An Investigation of Iterative Methods for Large Scale Linear Systems

Catalin-Andrei Ilie

Oriel College

University of Oxford

Project Coordinator: Professor David Kay

A thesis submitted for

Part B Student Projects

Trinity 2018

Acknowledgements

I would like to express my gratitude to Professor David Kay for the continuous help and teaching he has been providing me. He contributed tremendously both towards my education in the numerical methods field and towards getting the project at the current state.

Abstract

Several methods are presented to computationally approximate numerical solutions to a class of differential equations which are discretized to a linear system and kept in memory using a sparse structure. We have implemented, tested and compared both classic and more recent methods for solving the resulting linear equations. The main results are about the convergence rates of the Multigrid method and of the Preconditioned Conjugate Gradients method. An emphasis is put on the setting of the parameters and their effect on the solvers. Some interesting results are presented towards the end regarding the comparison of different parameter selections. We propose a new adaptive method for the smoothing parameter used in Multigrid methods and briefly investigate it.

Contents

Chapter 1

Motivation

1.1 Introduction

There are plenty of differential equations which arise from modelling real life phenomena. Most of them do not exhibit closed form solutions, so a numerical computational approach is required. These approaches, more often than not, lead to large scale linear algebra problems arising from certain discretization techniques.

In general, the linear systems of such problems are large, sparse and possess some interesting properties (e.g. diagonal dominance, positive-definiteness), discussed in the subsequent chapters.

It is clear that the finer a discretization is, the better it can approximate the (continuous) solution of a differential equation. However, there is an obvious trade-off between how many points the discretization has and the computational resources used. Therefore, the discretization size should be chosen in such a way to get a good enough approximation of the continuous solution (by interpolating the discrete approximation) and to keep the computations tractable.

Desirable methods and data structures used in such linear algebra tasks need to be adapted to sparse formats. As we will see shortly, for a 2D discretization grid of size $N \times N$ the resulting linear system will have $\mathcal{O}(N^4)$ entries, out of which only $\mathcal{O}(N^2)$ are non-zero.

We investigate the convergence of several approximation approaches for fixed discretization (grid) sizes, using two related classes of partial differential equations which are frequently used to model physical and chemical processes. The choice of the discretization size will not be discussed, as it depends on the specific requirement of approximation precision and on the computational resources available.

1.2 A class of differential equations

The Heat Equation models the distribution of heat in time over a region, given some initial conditions and a function describing the distribution of the source of heat over time:

$$\frac{\partial u}{\partial t} - k\nabla^2 u = f, \tag{1.1}$$

where $u(\mathbf{x}, t)$ models the distribution of temperature and $f(\mathbf{x}, t)$ is the source term at position $\mathbf{x}$ and time t ($\mathbf{x}$ is a vector in any number of dimensions).

For a set A, let us denote by δA the accumulation points of A which are not in it (the "border points") and let $\overline{A} = A \cup \delta A$. Let Ω be the open domain $(0,1)^2$, then $\delta\Omega = (\{0,1\} \times [0,1]) \cup ([0,1] \times \{0,1\})$ and $\overline{\Omega} = [0,1]^2$.

We will model the 2D homogeneous version of the Heat Equation with Dirichlet boundary condition and with an initial solution. The function u is defined for $((x,y),t) \in \overline{\Omega_T}$, where $\Omega_T = \Omega \times [0,1] = (0,1)^2 \times [0,1]$, so $\overline{\Omega_T} = \overline{\Omega} \times [0,1] = ([0,1] \times [0,1]) \times [0,1]$:

$$\begin{aligned} \frac{\partial u}{\partial t} - k(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = h, \text{ for } ((x,y),t) \in \Omega_T, \\ u(x,y,t) = e(x,y,t), \text{ for } ((x,y),t) \in \delta\Omega \times [0,1], \\ u(x,y,t_0) = u_0(x,y), \text{ at } t_0 = 0, \text{ for } (x,y) \in \overline{\Omega}, \end{aligned} \tag{1.2}$$

where k is a strictly positive constant.

Another strongly related PDE to the Heat Equation is the Poisson equation:

$$\nabla^2 u = f. \tag{1.3}$$

Similarly, we will model the 2D version of the Poisson Equation with Dirichlet boundary condition on the domain $(x,y) \in \overline{\Omega} = [0,1] \times [0,1]$ ($\Omega = (0,1) \times (0,1)$):

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= f, \text{ in } \Omega, \\ u(x,y) &= e(x,y), \text{ on } \delta\Omega. \end{aligned} \tag{1.4}$$

A special emphasis will be put on numerically approximating (1.4), as it is sufficient to prove the power of the proposed methods. We will shortly see how to extend techniques used to numerically approximate solutions of the Poisson Equation (1.4) in order to numerically approximate solutions of the Heat Equation (1.2), using the Backward Euler method (section 2.1.2).

The 1D version of the Poisson equation is:

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &= f, \text{ in } \Omega = (0,1), \\ u(x) &= e(x), \text{ on } \delta\Omega = \{0,1\}. \end{aligned} \tag{1.5}$$

1.3 Note on implementation

The experiments in this thesis have been carried on our implementation, provided at the end as an appendix. The code is highly customizable, allowing one to easily particularize and combine the solver methods among themselves and with discretization techniques. We will not refer much to the code in the current project, putting an emphasis on the theoretical aspect and on experimental observations.

Chapter 2

Discretization of the equations

2.1 Deducing the approximations

"Truth is much too complicated to allow anything but approximations."

(John von Neumann)

2.1.1 Finite differences method

A usual method when numerically approximating partial differential equations is to discretize their domain and approximate the differential terms using finite difference operators (e.g. [11], Chapter 8).

We discretize the space domain with an evenly spaced grid in both directions at coordinates $x_i = \frac{i}{N}$, for $i = 0, 1, \ldots, N$, and $y_i = \frac{i}{N}$, for $i = 0, 1, \ldots, N$ (we call this *the N-discretization grid*). When solving the Heat Equation at (1.2), we also introduce the concept of time discretization, at evenly sampled times $t_i = \frac{i}{T}$, for $i = 0, 1, \ldots, T$.

In figure 2.3, the solution for a Poisson differential equation with explicit form $f(x, y) = \sin(x)\sin(y)$ is shown for different levels of discretization. Clearly, higher values of N yield better approximations, but require more computational resources.

For given N and T, denote $h = \frac{1}{N}$ and $\tau = \frac{1}{T}$. For the Heat Equation (1.2), the Finite Difference approximations are:

$$\begin{gathered}\frac{\partial^2 u(x_m, y_n, t_p)}{\partial x^2} \approx \frac{\frac{u(x_{m+1}, y_n, t_p) - u(x_m, y_n, t_p)}{x_{p+1} - x_p} - \frac{u(x_m, y_n, t_p) - u(x_{m-1}, y_n, t_p)}{x_p - x_{p-1}}}{t_p - t_{p-1}} = \\ \frac{u(x_{m+1}, y_n, t_p) + u(x_{m-1}, y_n, t_p) - 2u(x_m, y_n, t_p)}{h^2}\end{gathered} \tag{2.1}$$

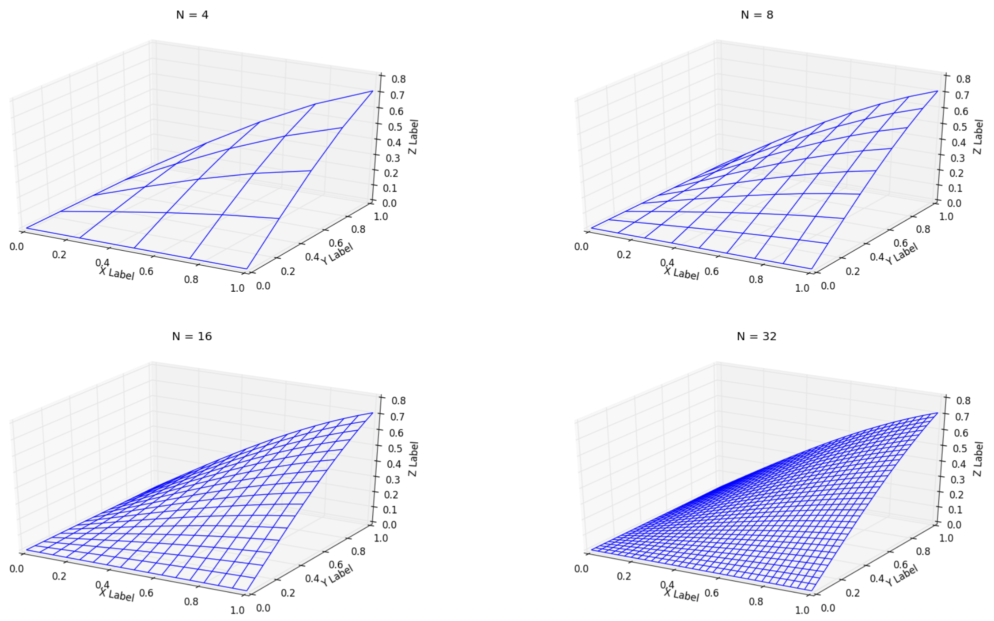


Figure 2.1: Different levels of discretization

and

$$\frac{\partial u(x_m, y_n, t_p)}{\partial t} \approx \frac{u(x_m, y_n, t_p) - u(x_m, y_n, t_{p-1})}{\tau}. \tag{2.2}$$

Approximating similarly for y-coordinates leads to the Finite Difference approximation for the Heat Equation:

$$\begin{gathered} f(x_m, y_n, t_p) = \frac{\partial u(x_m, y_n, t_p)}{\partial t} - k(\frac{\partial^2 u(x_m, y_n, t_p)}{\partial x^2} + \frac{\partial^2 u(x_m, y_n, t_p)}{\partial y^2}) \approx \\ \frac{u(x_m, y_n, t_p) - u(x_m, y_n, t_{p-1})}{\tau} - \\ k\frac{u(x_{m+1}, y_n, t_p) + u(x_{m-1}, y_n, t_p) + u(x_m, y_{n+1}, t_p) + u(x_m, y_{n-1}, t_p) - 4u(x_m, y_n, t_p)}{h^2} \end{gathered} \tag{2.3}$$

Similarly we can deduce the Finite Difference approximation for the Poisson Equation (1.4):

$$\begin{gathered} f(x_m, y_n) = \frac{1}{h^2}(u(x_{m+1}, y_n, t_p) + u(x_{m-1}, y_n, t_p) + \\ u(x_m, y_{n+1}, t_p) + u(x_m, y_{n-1}, t_p) - 4u(x_m, y_n, t_p)) \end{gathered} \tag{2.4}$$

2.1.2 Backward Euler

In order to approach (2.3), we iterate over t_k, for $k = 1, 2, \ldots, T$, using the solution at the previous time step in order to generate the next one (the solution at t_0 is given). Thus, we write the equation in the following form which also guarantees a better numerical stability:

$$\begin{aligned}(4k + \frac{h^2}{\tau})u(x_m, y_n, t_p) - k(u(x_{m+1}, y_n, t_p) + u(x_{m-1}, y_n, t_p) + \\ u(x_m, y_{n+1}, t_p) + u(x_m, y_{n-1}, t_p)) = h^2 f(x_m, y_n, t_p) + \frac{h^2}{\tau} u(x_m, y_n, t_{p-1})\end{aligned} \tag{2.5}$$

We denote the right hand side of (2.5) as: $L(x_m, y_n, t_p) = h^2 f(x_m, y_n, t_p) + \frac{h^2}{\tau} u(x_m, y_n, t_{p-1})$. Note that L can be computed in terms of f, which is given, and of the solution for u at one time step before p, which leads us to the following approach:

Algorithm 1: Backward Euler for the time variable

```
Data: $f$, $u_{\text{init}}$, $h$
/* $u_i$ is the solution for $u$ at time step $i$ */
/* $f_i$ is $f$ at time step $i$ */
/* $e_i$ is $e$ from (1.2), modelling the border values of $u$ at time step $i$ */
1 $u_0 \leftarrow u_{\text{init}}$
2 for $i = 1; i <= T; i++$ do
3     $L_i \leftarrow h^2 f_i + \frac{h^2}{\tau} u_{i-1}$     // compute the new RHS of (2.5) for all $x, y$
4     $u_i \leftarrow SolveHeat(L_i, e_i)$     // solve for the current time step
5 $u \leftarrow (u_0, u_1, \cdots, u_T)$
6 return $u$     // where $u(x_m, y_n, t_p) = u_p(x_m, y_n)$
```

The algorithm 1 calls a procedure $SolveHeat(L_i, e_i)$ which takes as input the right-hand side function of equation (2.5) and the border value of u at time step i. It returns the approximation for u at time step i.

Similarly, in order to approach (2.4), we write it in the more convenient way:

$$\begin{aligned}4u(x_m, y_n) - (u(x_{m+1}, y_n) + u(x_{m-1}, y_n) + u(x_m, y_{n+1}) + u(x_m, y_{n-1})) \\ = -h^2 f(x_m, y_n), \text{ for } m = 1, \ldots, N-1, \text{ and } n = 1\ldots, N-1.\end{aligned} \tag{2.6}$$

A solution approximation will be the output of a procedure $SolvePoisson(L, e)$, where L is $h^2 f$ and e is the function which gives the border values of u.

We will see in section 2.1.3 how both $SolvePoisson$ and $SolveHeat$ reduce to solving a large, sparse, (semi-)positive definite linear system. Therefore, most of this project is based on both theoretical and practical aspects regarding the procedure $SolvePoisson$.

Similarly, the 1D version of Poisson (2.7) is discretized as :

$$2u(x_m) - (u(x_{m+1}) + u(x_{m-1})) = -h^2 f(x_m), \text{ for } m = 1, \ldots, N-1. \tag{2.7}$$

2.1.3 Creating a linear system

It is common practice when solving systems of equations which resemble (2.5) and (2.6) to arrange them in a convenient way, capturing their left hand side in a matrix ($\mathbf{A}$) and their right hand side in a vector ($\mathbf{b}$). For each point on the discretization (x_m, y_n) there is a line in $\mathbf{A}$ describing its linear combination and an entry in $\mathbf{b}$ equal to the value yielded by that linear combination.

First of all, in the Poisson Equation we create a matrix and a vector. We enumerate the points in the discretization lexicographically, each corresponding to a linear equation centred around it. The equations are captured in this order in the lines of $\mathbf{A}_{\text{Poisson}}$.

For interior (x_m, y_n) which are adjacent to some locations on the borders of the domain, we push the border values from the (x_m, y_n) centered equation to the right-hand side, as their value is known. In the case of (x_m, y_n) lying on the border, for which the value of u is given, we can either skip adding it to $\mathbf{A}_{\text{Poisson}}$, or just add 1 to its corresponding line and column, 0 everywhere else on the line, and the border value $e(x_m, y_n)$.

For example, we illustrate how to obtain (conveniently arrange the linear equations) $\mathbf{A}$ and $\mathbf{b}$ in the 1D case (2.7) for $N = 4$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u(x_0) \\ u(x_1) \\ u(x_2) \\ u(x_3) \\ u(x_4) \end{bmatrix} = \begin{bmatrix} e(x_0) \\ e(x_0) - \frac{1}{16} f(x_1) \\ -\frac{1}{16} f(x_2) \\ e(x_4) - \frac{1}{16} f(x_3) \\ e(x_4) \end{bmatrix} \tag{2.8}$$

When we store the border values in the matrix, the solver methods can be implemented such that their effect is the same as when the border values are not stored in the matrix. We constructed our implementations in this way, so it suffices to argue about the case without border values in the matrix, which, in the example (2.8), is: $\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u(x_1) \\ u(x_2) \\ u(x_3) \end{bmatrix} = \begin{bmatrix} e(x_0) - \frac{1}{16} f(x_1) \\ -\frac{1}{16} f(x_2) \\ e(x_4) - \frac{1}{16} f(x_3) \end{bmatrix}$. It does not mention $u(x_0), u(x_4)$, as these can be recovered easily from the border condition : $u(x_0) = e(x_0), u(x_4) = e(x_4)$.

For the Heat Equation, we need a linear equation at each time step. However, the matrix is the same for all time steps, $\mathbf{A}^{\mathbf{t}}_{\text{Heat}} = \mathbf{A}_{\text{Heat}}$, so we adopt he same construction described above for $\mathbf{A}_{\text{Heat}}$, but the vector $\mathbf{b}_t$ depends on the time step t. This is captured in algorithm 1, line 3 performing the vector update.

2.1.4 Properties of matrices arising from discretization

It is easy to see that the constructed matrices are symmetric: if a row r has a non-diagonal, non-zero entry on column c, it must be -1 (or $-k$). This is justified by the variable on which the r^{th} row is centred being a neighbour of the variable on which the c^{th} row is centered (both variables are not on the boundary, as we "pushed" the boundary values in the vectors $\mathbf{b}$). As this relation is symmetric, row c must also have a -1 (or $-k$) entry on column r, therefore the matrices created above are symmetric. It is well known that ([17], 331):

Theorem 2.1.1. *Real symmetric matrices have real eigenvalues.*

Therefore, both $\mathbf{A}_{\text{Heat}}$ and $\mathbf{A}_{\text{Poisson}}$ have real eigenvalues. We also examine if the matrices are (strictly) diagonally dominant ([6], 155):

Definition 2.1.1. *A square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is called "(row) diagonally dominant" if:*

$$|a_{ii}| \geq \sum_{j=1, j \neq i}^{n} |a_{ij}|, i = 1, 2, \ldots, n. \tag{2.9}$$

The matrix is called "strictly (row) diagonally dominant" if the inequalities above hold strictly.

A desirable property of matrices when numerically solving linear equations is "(semi) positive definite" ([19], 87):

Definition 2.1.2. *A matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is called positive definite if $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ for every vector $\mathbf{x} \in R^n - \{\mathbf{0}\}$. ($\mathbf{A}$ is semi-positive definite if $\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0$ for every $\mathbf{x}$.)*

An important characterization of real, symmetric, positive definite matrices is given by the following theorem ([18], 318):

Theorem 2.1.2. *Each of the following is a necessary and sufficient condition for the real symmetric matrix A to be positive definite:*

1. $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ *for all nonzero real vectors* $\mathbf{x}$,
2. *all the eigenvalues of* $\mathbf{A}$ *are strictly positive,*
3. *all the upper left submatrices* $\mathbf{A}_k$ *have strictly positive determinants.*

Theorem 2.1.2 has its natural correspondent for semi-positive definite matrices, allowing equality in 1., nonnegative eigenvalues in 2. and nonnegative determinants ($det(\mathbf{A}_k) \geq 0$) in 3..

It can be shown using Gershgorin's theorem that real, symmetric, (strictly) diagonally dominant matrices are (strictly) positive definite as long as their diagonal entries are positive. Gershgorin's theorem is stated below ([9], 388):

Theorem 2.1.3. *Let* $\mathbf{A} = [a_{ij}] \in \mathbb{M}_n$ *and* $R_i(\mathbf{A}) = \sum_{j=1,j\neq i}^{n} |a_{ij}|, i = 1, 2, \ldots, n$. *Consider the* n *Gershgorin discs:*

$$D_i(A) = \{z \in \mathbb{C} \ : |z - a_{ii}| \leq R_i(\mathbf{A})\} \tag{2.10}$$

Then, the eigenvalues of $\mathbf{A}$ *are in the union of Gershgorin discs* $\bigcup_{i=1}^{i=n} D_i$ *.*

Combining (2.10), theorem 2.1.1 and theorem 2.1.3, we get:

Theorem 2.1.4. *A symmetric, (strictly) diagonally dominant real matrix with positive entries on its diagonal has (strictly) positive eigenvalues - is (semi-)positive definite.*

Proof. Let $\mathbf{A}$ be a symmetric, diagonally dominant real matrix. By theorem 2.1.1, its eigenvalues are real. As it is diagonally dominant, $|a_{ii}| \geq \sum_{j=1,j\neq i}^{n} |a_{ij}|, i = 1, 2, \ldots, n$. By Gershgorin's theorem (2.1.3), each eigenvalue of $\mathbf{A}$ is in some Gershgorin disc. Then, for each eigenvalue λ there is some row i such that $|\lambda - a_{ii}| \leq R_i(A)$. As a_{ii} is real, positive, this means that λ, which is also real, is in the intersection of the Gershgorin disc centred at a_{ii} with radius $R_i(A)$ and the real axis. By the diagonal dominance, we get $|a_{ii}| = a_{ii} \geq R_i$, so λ lies in the interval $[a_{ii} - R_i, a_{ii} + R_i]$, which consists only of positive, real values. Therefore, all the eigenvalues of $\mathbf{A}$ are positive. The proof goes exactly the same if we take its strict form. □

The matrix constructed for the 2D Heat Equation, $\mathbf{A}_{\text{Heat}}$ is both symmetric and strictly diagonally dominant. Notice that each line either has 1 on the main diagonal and 0 everywhere else, or $4k + \frac{h^2}{\tau}$ on the main diagonal, and at most 4 other entries on that row of value $-k$, where k is a positive, real number (at most 4 entries because some of the 4 neighbours may be on the border and get pushed to the RHS). We illustrate $\mathbf{A}_{\text{Heat}}$ below for $N = T = 4$, with no border points in the matrix, and the linear equation at time step $j + 1$ (for $j = 0, 1, \ldots, N - 1 = 0, 1, 2, 3$). Below, we use the exponent notation to show the value of a function at time step q ($t_q = \frac{q}{N}$) : $f^q(x, y) = f(x, y, t_q)$. The matrix we construct is:

$$\mathbf{A}_{\text{Heat}} = \begin{bmatrix} 4k+\frac{1}{4} & -k & 0 & -k & 0 & 0 & 0 & 0 & 0 \\ -k & 4k+\frac{1}{4} & -k & 0 & -k & 0 & 0 & 0 & 0 \\ 0 & -k & 4k+\frac{1}{4} & 0 & 0 & -k & 0 & 0 & 0 \\ -k & 0 & 0 & 4k+\frac{1}{4} & -k & 0 & -k & 0 & 0 \\ 0 & -k & 0 & -k & 4k+\frac{1}{4} & -k & 0 & -k & 0 \\ 0 & 0 & -k & 0 & -k & 4k+\frac{1}{4} & 0 & 0 & -k \\ 0 & 0 & 0 & -k & 0 & 0 & 4k+\frac{1}{4} & -k & 0 \\ 0 & 0 & 0 & 0 & -k & 0 & -k & 4k+\frac{1}{4} & -k \\ 0 & 0 & 0 & 0 & 0 & -k & 0 & -k & 4k+\frac{1}{4} \end{bmatrix},$$

and the linear equation is:

$$\mathbf{A}_{\text{Heat}} \begin{bmatrix} u^{j+1}(x_1,y_1) \\ u^{j+1}(x_1,y_2) \\ u^{j+1}(x_1,y_3) \\ u^{j+1}(x_2,y_1) \\ u^{j+1}(x_2,y_2) \\ u^{j+1}(x_2,y_3) \\ u^{j+1}(x_3,y_1) \\ u^{j+1}(x_3,y_2) \\ u^{j+1}(x_3,y_3) \end{bmatrix} = \begin{bmatrix} \frac{1}{16}f^{j+1}(x_1,y_1) + u^j(x_1,y_1) + ke^{j+1}(x_0,y_1) + ke^{j+1}(x_1,y_0) \\ \frac{1}{16}f^{j+1}(x_1,y_2) + u^j(x_1,y_2) + ke^{j+1}(x_0,y_2) \\ \frac{1}{16}f^{j+1}(x_1,y_3) + u^j(x_1,y_3) + ke^{j+1}(x_0,y_3) + ke^{j+1}(x_1,y_4) \\ \frac{1}{16}f^{j+1}(x_2,y_1) + u^j(x_2,y_1) + ke^{j+1}(x_2,y_0) \\ \frac{1}{16}f^{j+1}(x_2,y_2) + u^j(x_2,y_2) \\ \frac{1}{16}f^{j+1}(x_2,y_3) + u^j(x_2,y_3) + ke^{j+1}(x_2,y_4) \\ \frac{1}{16}f^{j+1}(x_3,y_1) + u^j(x_3,y_1) + ke^{j+1}(x_3,y_0) + ke^{j+1}(x_4,y_1) \\ \frac{1}{16}f^{j+1}(x_3,y_2) + u^j(x_3,y_2) + ke^{j+1}(x_4,y_2) \\ \frac{1}{16}f^{j+1}(x_3,y_3) + u^j(x_3,y_3) + ke^{j+1}(x_3,y_4) + ke^{j+1}(x_4,y_3) \end{bmatrix}.$$

$\mathbf{A}_{\text{Heat}}$ is indeed strictly diagonally dominant and symmetric, with all diagonal entries positive, so, by theorem 2.1.4, it is positive definite. The same argument applies to $\mathbf{A}_{\text{Poisson}}$, but in this case we get that it is semi-positive definite, as for some entries we have 4 on the diagonal and 4 other elements with value -1 on the same row. It can be shown by direct calculation of $\mathbf{x}^T\mathbf{A}_{\text{Poisson}}\mathbf{x}$ that $\mathbf{A}_{\text{Poisson}}$ is positve definite as well ([15]).

To conclude, both $\mathbf{A}_{\text{Heat}}$ and $\mathbf{A}_{\text{Poisson}}$ are real, symmetric and positive definite matrices. Also, the size of the matrices is $(N+1)^2 \times (N+1)^2 \implies \mathcal{O}(N^4)$ entries, but there are at most 5 elements per line which are non-zero, so $\mathcal{O}(N^2)$ non-zero entries. This is why we call them "sparse". In general the following definition describes well the class of sparse matrices [2] : "a matrix is sparse if it contains enough zero entries to be worth taking advantage of them to reduce both the storage and work required in solving a linear system.". The linear equations that arise from discretizing our model problems are summarized by:

Problem 2.1.1. *Solve* $\mathbf{Ax} = \mathbf{b}$*, where* A *is a real, symmetric, positive definite, sparse matrix.*

2.2 Sparse data structures

2.2.1 Requirements

One of the main aims is to optimize the memory. As mentioned before, there are only $\mathcal{O}(N^2)$ non-zero entries in a matrix consisting of $\mathcal{O}(N^4)$ entries. For example, for $N = 256$, keeping the whole matrix in memory in a float format in Python, on a 64 bit machine, requires approximately 96 GB of data, out of which the 0 entries take approximately 95.995 GB.

Another very important aim is optimizing the time of the operations used in the linear solvers. For the methods to be discussed, a fast computation of a matrix product with a vector and fast access of rows/elements of the matrix are crucial.

2.2.2 CSR format

The Compressed Sparse Row format (CSR) is an efficient way of storing sparse matrices which is described in ([14], 3.4). For a sparse matrix $\mathbf{M}$, it uses an array M_{data} of the non-zero elements in the matrix in row major order, an array J_M of column indices of the elements in the same order as M_{data}, and an array I_M which points to the start of each row of the matrix $\mathbf{M}$ in the 2 arrays described above.

The matrix $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$, which arose in example (2.8) by discretizing the 1D version of (1.4), would be stored as:

- $M_{data} = [1, 2, -1, -1, 2, -1, -1, 2, 1]$,
- $J_M = [0, 1, 2, 1, 2, 3, 2, 3, 4]$,
- $I_M = [0, 1, 3, 6, 8, 9]$,

where the last entry in I_M is by convention the number of non-zero elements.

Let the size of $\mathbf{M}$ be $K \times K$ and let NNZ be the number of non-zero elements in $\mathbf{M}$. The i^{th} row non-zero elements and their column indices can be retrieved from the arrays M_{data} and J_M, between the indices $I_M[i], I_M[i+1] - 1$, hence we get fast matrix-vector product: we only perform one multiplication and one addition for each non-zero element in $\mathbf{M}$, with $\mathcal{O}(1)$ amortized access per element, therefore $\mathcal{O}(NNZ)$. Multiplying the matrix in the usual way, by iterating through all of its elements, would otherwise be in $\mathcal{O}(K^2)$ time.

The size of M_{data} if NNZ, the size of J_M is NNZ as well and the size of I_M is $K+1$, therefore the CSR format stores $NNZ+K+1$ integers and NNZ floats, while keeping the whole matrix would be $K \times K$ floats.

2.3 Measuring the approximation algorithms performance

Before presenting the approximation algorithms, it is important to discuss when we consider that an approximation method "converges" for our model problems and which are the important performance metrics.

After using the Finite Difference approximations for a certain grid spacing $h = \frac{1}{N}$, we are left to approximate the solution $\mathbf{u}^N$ of a linear equation $\mathbf{A}^N\mathbf{u} = \mathbf{f}^N$, where the components of $\mathbf{u}^N$ approximate the values of a continuous function at some points in the domain. Below we discuss only the case when the linear algebra approximation methods converge (we take care of conditions for this to happen in chapter 3).

2.3.1 Convergence of methods

Definition 2.3.1. *Let $\mathbf{x}^*$ be the exact solution of a linear system $\mathbf{Ax} = \mathbf{b}$ and let $\mathbf{z}$ be an approximation of $\mathbf{x}^*$. We call the vector $\mathbf{e} = \mathbf{x}^* - \mathbf{z}$ the error and $\mathbf{r} = \mathbf{b} - \mathbf{Az}$ the residual error of the approximation.*

Note that the approximation accuracy we are measuring is that of approximating the solution of the linear system. For an approximation $\mathbf{u}_k^N$ of $\mathbf{u}^N$, the error is as hard to determine as the exact solution : if $\mathbf{e} = \mathbf{u}^N - \mathbf{u}_k^N$ is the error, then $\mathbf{u}^N = \mathbf{e} + \mathbf{u}_k^N$. Therefore, the accuracy of an approximation will be measured in terms of how close the relative residual error $\frac{\|\mathbf{b}^N - \mathbf{A}^N\mathbf{u}_k^N\|}{\|\mathbf{b}^N\|}$ is to 0.

We iterate the linear algebra approximation algorithms until the relative residual error becomes smaller than a tolerance constant $\epsilon_{\text{tolerance}}$ (which is 10^{-5} in our experiments), or until a certain number of iterations is reached.

The number of floating point operations (FLOPs) required by certain approximation algorithms in order to converge is the best estimate for the time cost of the methods (the CPU time depends heavily on the environment, the libraries used for linear algebra operations, etc.).

2.3.2 Continuous approximation

Let $\mathbf{u}^{N,k}$ be the approximation obtained after k updates of an approximation algorithm for the linear system of grid size N. Note that $\lim_{k\to\infty} \mathbf{u}^{N,k} = \mathbf{u}^N$ (where $\mathbf{u}^N$ is the discrete approximation at level N). Also, $\lim_{N\to\infty} \mathbf{u}^N = u$ (u is the true, continuous solution, see figure 2.2). $\|\mathbf{u} - \mathbf{u}^{N,k}\| \leq \|\mathbf{u} - \mathbf{u}^N\| + \|\mathbf{u}^N - \mathbf{u}^{N,k}\|$ (where $\mathbf{u}$ in $\|\mathbf{u} - \mathbf{u}^{N,k}\|$ stands for projecting the real solution u onto the grid of level h, such that we can compute the vector differences involved). It is a classic result that Finite Difference approximations have the property $\|\mathbf{u} - \mathbf{u}^N\| = \mathcal{O}(\frac{1}{N})$ ([15]). As $\lim_{k\to\infty} \mathbf{u}^{N,k} = \mathbf{u}^N$, for any arbitrarily small positive ϵ we can choose a number of iterations k_N such that $\|\mathbf{u}^N - \mathbf{u}^{N,k_N}\| < \epsilon$, therefore $\lim_{k\to\infty} \|\mathbf{u} - \mathbf{u}^{N,k}\| = \mathcal{O}(\frac{1}{N})$.

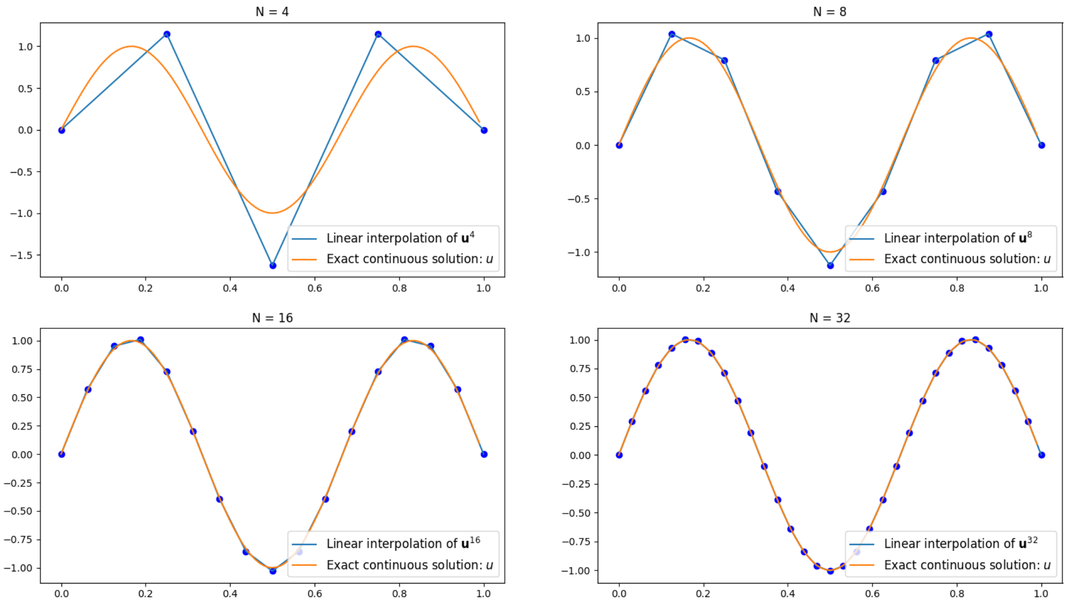


Figure 2.2: Linear interpolations of the exact solutions of Finite Difference approximations for different grid sizes (blue) versus the exact continuous solution (orange) of a 1D Poisson Equation with exact solution $f(x) = \sin 3\pi x$

Therefore, the closest we can get to the projection of the real, continuous solution on the grid of grid spacing $h = \frac{1}{N}$ is $\mathcal{O}(h)$ (and this bound is reachable for a sufficiently large number of iterations).

For suitably chosen grid size N and iteration count k, the approximation $\mathbf{u}^{N,k}$ of $\mathbf{u}^N$ can be interpolated in order to approximate the exact solution u.

For the rest of this thesis, we use Fourier analysis terminology interchangeably for continuous functions and their projections on certain grids (see figure 2.3 for 1D example, see [4] for a more detailed discussion). Note that writing (or approximating) a function as a series of sine terms yields the same result for its projection onto a grid: it is a linear combination of (or approximated by) the projections of the sine terms onto the same grid.

For example, in 1D the series terms are of the form $\sin k\pi x$ and $\cos k\pi x$: continuous functions $F : [0, 1] \to \mathbb{R}$ can be written as a combination of such terms: $F(x) = a_0 + \sum_{k=1}^{\infty}(a_k \sin k\pi x + b_k \cos k\pi x)$.

We say that $\sin k\pi x$ is the sine term with *wave number* k and with *frequency* $\frac{k}{2}$ (number of complete periods on $[0, 1]$). For a grid spacing of size N, only the sine terms with wave numbers in $i = 1, 2, \dots, N$ can be captured (figure 2.4), as greater wave numbers suffer from *aliasing*: the grid is not fine enough to capture such oscillatory modes, they will look as smoother components. Let us adopt the terminology in [4] and refer to the modes $\sin k\pi x$ with $1 \leq k < \frac{N}{2}$ as being low frequency (smooth) modes, and to the ones with $\frac{N}{2} \leq k \leq N$ as being high frequency (oscillatory) modes. The same definitions go for the cos terms. This discussion generalizes naturally to any dimension.

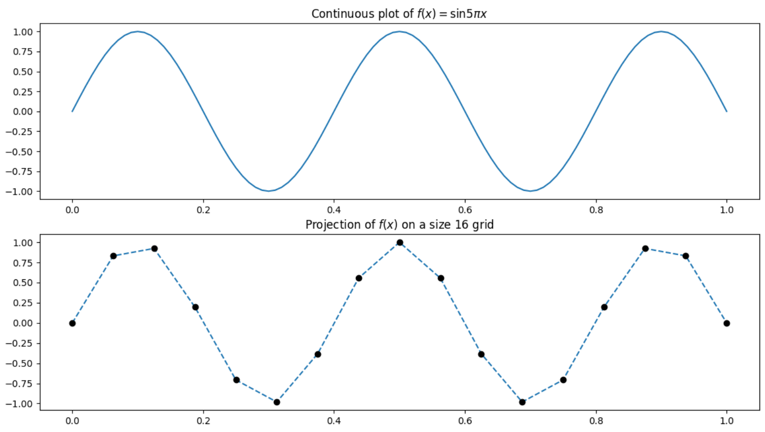


Figure 2.3: The first plot shows the continuous $f(x) = \sin 5\pi x$. It has wave number 5 and frequency 2.5 (2 complete sine cycles and a half). The second plot represents the projection of $f(x)$ onto a grid with $h = \frac{1}{16}$ spacing of the interval $[0, 1]$. We will also be referring to the vector containing this discretization ($[\sin(5\pi * \frac{0}{16}), \sin(5\pi * \frac{1}{16}), \cdots, \sin(5\pi * \frac{16}{16})]^T$) as having the same frequency (2.5) and wave number (5) as its continuous model.

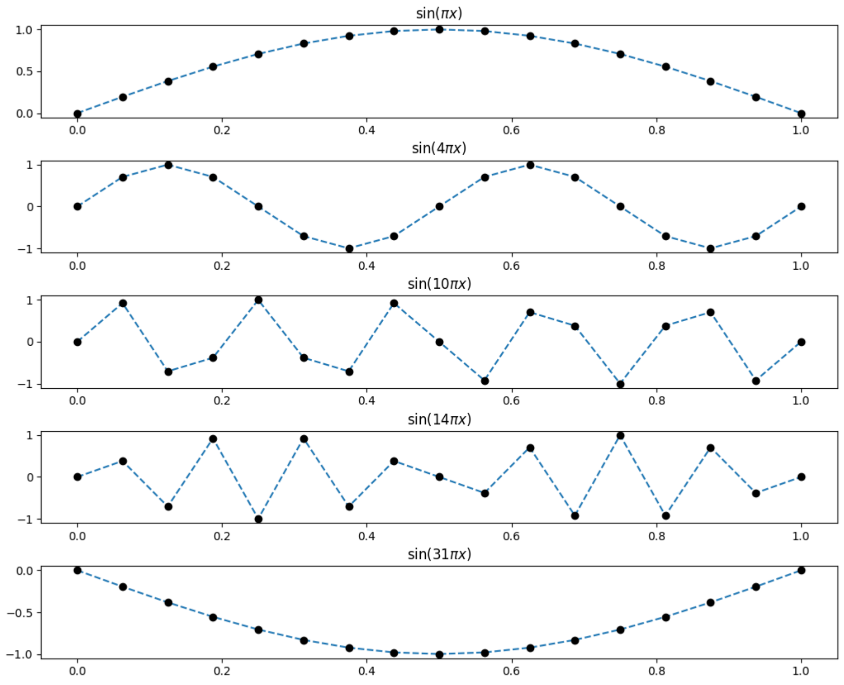


Figure 2.4: Sine modes with wave numbers $1, 4, 10, 14$ *and* 31*. Note that the* 31 *mode represents the same frequency as the* 1 *mode because of aliasing, we should not consider modes with wave number above* 16*.*

Chapter 3

Classic iterative methods

3.1 Matrix splitting methods

3.1.1 General Formulation

The main idea behind a wide class of iterative methods for numerically approximating the solution of $\mathbf{Ax} = \mathbf{b}$, which is described in ([6], 11.2.3) and [7], is to split the matrix $\mathbf{A}$ in $\mathbf{A} = \mathbf{M} - \mathbf{N}$ and rewrite the linear equation as $\mathbf{Mx} = \mathbf{Nx} + \mathbf{b}$. The choice must be made such that $\mathbf{M}$ is invertible and easy to invert, case in which we get:

$$\mathbf{x} = \mathbf{Tx} + \mathbf{c}, \text{ where } \mathbf{T} = \mathbf{M}^{-1}\mathbf{N}, \mathbf{c} = \mathbf{M}^{-1}\mathbf{b}, \tag{3.1}$$

which can naturally be interpreted as an iterative scheme:

$$\mathbf{x}^{k+1} = \mathbf{Tx}^k + \mathbf{c}, \text{ with } \mathbf{x}^0 \text{ an initial approximation.} \tag{3.2}$$

In standard texts, $\mathbf{T}$ is called *iteration matrix* and c is called *iteration vector.*

The following result (e.g. [6], [4], [7]) gives a necessary and sufficient condition for the convergence of such methods, based on the *spectral radius* of T (denoted by $\rho(\mathbf{T})$, equal to the highest absolute value of the eigenvalues of $\mathbf{T}$):

Theorem 3.1.1. *Assuming that* $\mathbf{M}$ *is invertible, the iteration (3.2) converges for any initial choice of* $\mathbf{x}^0$ *if and only if* $\rho(\mathbf{T}) < 1$.

Let $\mathbf{D}$ be the diagonal part of $\mathbf{A}$ and $\mathbf{L}, \mathbf{U}$ be the strictly lower and strictly upper parts of $\mathbf{A}$ (obviously, $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$).

3.1.2 Jacobi

The Jacobi iteration is the simplest instance of the method described above. It splits the matrix $\mathbf{A}$ in $\mathbf{M} = \mathbf{D}$ and $\mathbf{N} = -\mathbf{L} - \mathbf{U}$, so we get the form:

$$\mathbf{x}^{k+1} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^k + \mathbf{D}^{-1}\mathbf{b}. \tag{3.3}$$

The method converges for any initial solution $\mathbf{x}^0$ if $\mathbf{A}$ is strictly diagonally dominant ([6], 11.2.2). This is the case for the matrix obtained by discretizing the Heat Equation, but the matrix obtained by discretizing the Poisson equation is only diagonally dominant. However, for our purposes, the Jacobi method will also work in the second case (theorem 3.1.3 in the next subsection shows that Jacobi works for irreducibly diagonally dominant matrices).

The weighted version of Jacobi (damped Jacobi) depends on a parameter $\omega \in (0, 1]$:

$$\begin{gathered} \mathbf{x}^{k+1} = \omega(-\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^k + \mathbf{D}^{-1}\mathbf{b}) + (1-\omega)x^k, \text{ therefore} \\ \mathbf{x}^{k+1} = \mathbf{T}_\omega \mathbf{x}^k + \omega\mathbf{D}^{-1}\mathbf{b}, \text{ where } \mathbf{T}_\omega = -\omega\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) + (1-\omega)\mathbf{I}. \end{gathered} \tag{3.4}$$

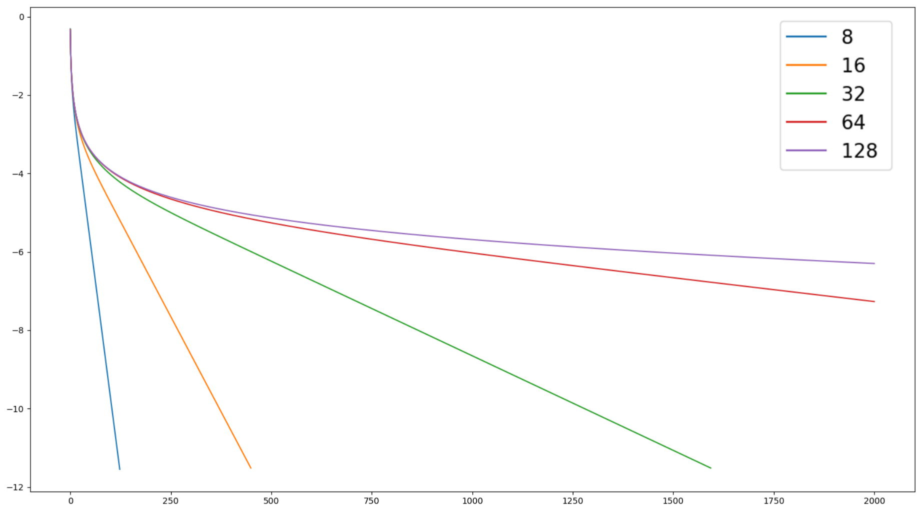


Figure 3.1: Logarithmic scale of the residual Euclidian norm in terms of the number of Jacobi ($\omega = 1$) iterations for different grid sizes of the 2D Poisson Equation discretization with exact solution $f(x, y) = \sin(x)\sin(y)$. The legend describes the sample number N. Note that the number of unknowns is quadratic in the value of N for our 2D model problem discretization.

The main advantage of weighted Jacobi is that it can easily be parallelized, but it possesses the so called *smoothing property*: it eliminates fast the high frequency components of the error (for well chosen ω), but performs badly on the low frequency components (for any ω).

A spectral analysis of how different frequency components decrease can be found in [4], chapter 2. For the ease of notation, let $\mathbf{A} \in \mathbb{R}^{n\times n}$. The main idea is that the eigenvectors of A correspond to different frequency Fourier modes $\mathbf{w_k}, k = 1, 2, \ldots, n$ with associated eigenvalues $\theta_k, k = 1, 2, \ldots, n$. As $\mathbf{T}_\omega = -\omega\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}+\mathbf{D}) + \omega\mathbf{D}^{-1}\mathbf{D} + (1-\omega)\mathbf{I} = -\omega\mathbf{D}^{-1}\mathbf{A} + \mathbf{I} = -\frac{\omega}{4}\mathbf{A} + \mathbf{I}$, it follows that $\mathbf{T}_\omega$ and $\mathbf{A}$ have the same eigenvectors, and the eigenvalues of $\mathbf{T}_\omega$ are $\lambda_k = 1 - \frac{\omega}{4}\theta_k, k = 1, 2, \ldots, n$.
As $\mathbf{A}$ is symmetric, its eigenvectors are a basis for $\mathbb{R}^n$ (classic result, see [17], [18]), so there exist real coefficients c_i such that $\mathbf{e_0} = \sum_{i=1}^{i=n} c_i\mathbf{w_i}$, where $\mathbf{e_k}$ is the error at n^{th} step, $\mathbf{e_k} = \mathbf{x} - \mathbf{x}^k$, and $\mathbf{x}$ is the exact solution of the linear sytem. As $\mathbf{e_k} = {\mathbf{T}_\omega}^k\mathbf{e_0}$, we get:

$$\mathbf{e_k} = {\mathbf{T}_\omega}^k \sum_{i=1}^{i=n} c_i\mathbf{w_i} = \sum_{i=1}^{i=n} c_i\lambda_i^k\mathbf{w_i}. \tag{3.5}$$

It is proved in [4] that, for any choice of ω, the λ_i associated with low frequency $\mathbf{w_i}$ are close to 1, so we can adjust ω to get certain higher frequency components diminished faster.

The figure 3.1 is suggestive for how some (high) frequency components of the error get smoothed at the same rate, independently of the mesh size. As soon as they are eliminated, the differences are clearly noticeable, and the bigger the mesh size, the bigger the spectrum of (low) frequencies that are hardly captured by Jacobi, therefore the performance worsens.

Adjusting ω does not mean improving the overall convergence of the Jacobi method; it is a trade-off between the general convergence and being biased towards diminishing certain modes. This is useful if the error is mainly made up of high frequencies, but in our case the problem that arises has a large spectrum of frequencies, so we do not expect to get a better overall convergence for $\omega \neq 1$ (see figure 3.2).

3.1.3 Gauss-Seidel and SOR

Gauss-Seidel is another matrix splitting iteration, based on choosing $\mathbf{M} = \mathbf{D} + \mathbf{L}$ and $\mathbf{N} = -\mathbf{U}$ ([6], [7], [14]). Then, the iteration (3.2) becomes:

$$\mathbf{x}^{k+1} = -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}\mathbf{x}^k + (\mathbf{D}+\mathbf{L})^{-1}\mathbf{c}. \tag{3.6}$$

The main results ([6], 11.2.3 and [14], 4.9) about the convergence of this method are theorems 3.1.2 and 3.1.3 below.

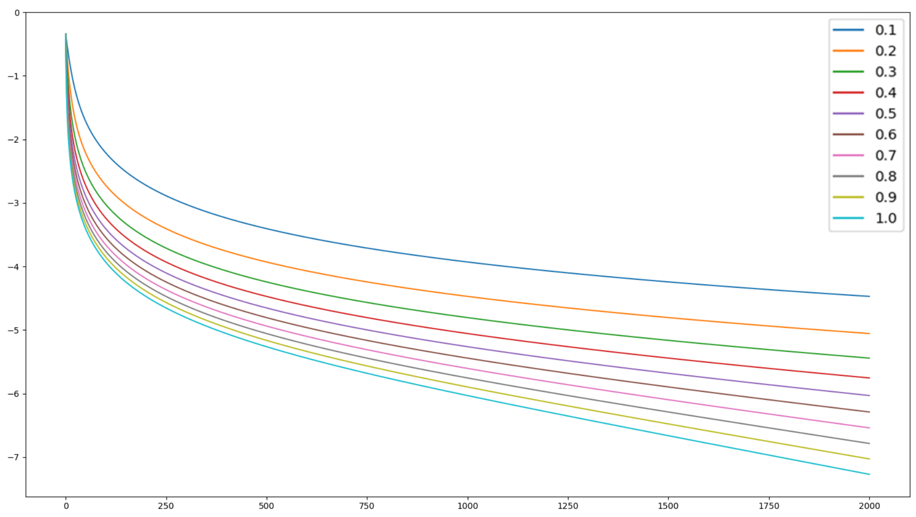


Figure 3.2: Logarithmic scale of the residual Euclidian norm in terms of the number of damped Jacobi iterations with varying ω.The bigger ω is, the better general convergence we get, as the spectrum of frequencies that arises is large.

Theorem 3.1.2. *If $\mathbf{A} \in \mathbb{R}^{n \times n}$ is symmetric and positive definite, then the Gauss-Seidel iteration (3.6) converges for any initial guess $\mathbf{x}^0$.*

Definition 3.1.1. *A square matrix $\mathbf{C}$ is irreducible if the directed graph which has the adjacency matrix $\mathbf{C}$ (i.e. the graph with number of rows of $\mathbf{C}$ vertices and with directed edges between vertices v_x and v_y iff $\mathbf{C}[x][y]$ is not 0) is strongly connected.*

Theorem 3.1.3. *If $\mathbf{A}$ is a strictly diagonally dominant or an irreducibly diagonally dominant matrix, then the associated Jacobi and Gauss-Seidel iterations converge for any $\mathbf{x}_0$.*

The matrices that arise from our discretizations (without border storage) are irreducible, as the graphs that result are just the adjacency graphs of interior points in our grids, which are strongly connected. As they are diagonally dominant, the convergence of Gauss-Seidel and Jacobi also follows from theorem 3.1.3.

As stated in chapter 2, our model problems reduce to problem 2.1.1. Therefore, the matrix we are working with fulfills the conditions of (3.1.2), so Gauss-Seidel converges for any initial guess.

Figure 3.3 shows that Gauss-Seidel converges approximately twice as fast as Jacobi, as the Gauss-Seidel iteration matrix for such systems has eigenvalues equal to the square of the eigenvalues of the iteration matrix for Jacobi. However, it cannot be parallelized, unlike Jacobi.

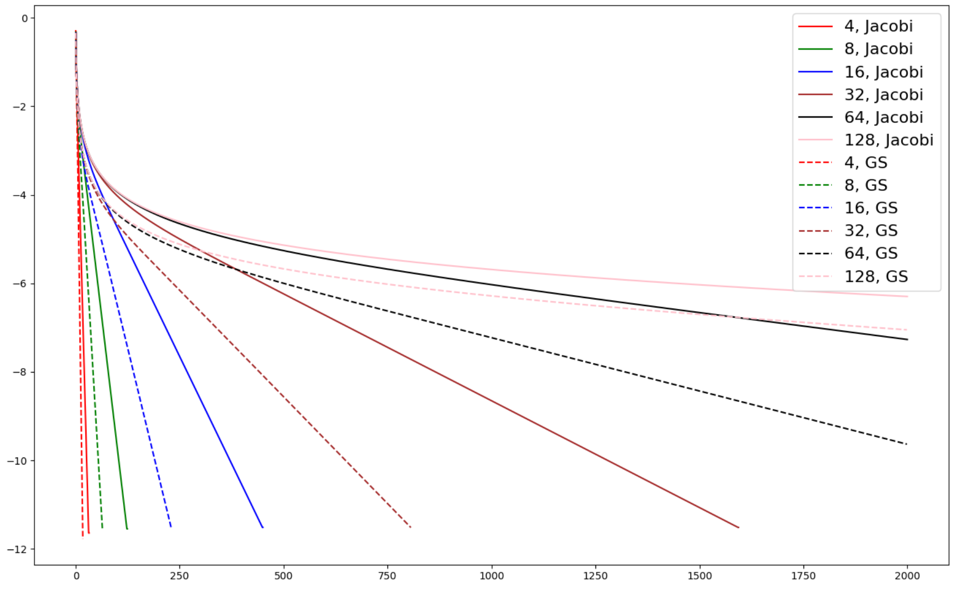


Figure 3.3: Comparison in logarithmic scale of the convergence of Gauss-Seidel (dashed line) and Jacobi (complete line) in the limit of 2000 *iterations under the same setup as figure (3.1).*

Gauss-Seidel gets slowed down if the spectral radius of the iteration matrix is close to 1. This follows from (3.5):

$$\mathbf{e_k} = \sum_{i=1}^{i=n} c_i \lambda_i^k \mathbf{w_i},$$

so the closer a λ_i is to 1, the worse the convergence of the method is. In order to avoid this, one might try an averaged splitting of the matrix $\mathbf{A}$, parameterized by a real number ω, similar to damped Jacobi. This approach is called Successive Over-Relaxation (SOR) (see [6], [14]). The matrix splitting $\mathbf{A} = \mathbf{M}_\omega - \mathbf{N}_\omega$ is given by [6], 11.2.7:

$$\begin{aligned} \mathbf{M}_\omega &= \frac{1}{\omega}\mathbf{D} + \mathbf{L}, \\ \mathbf{N}_\omega &= (\frac{1}{\omega} - 1)\mathbf{D} - \mathbf{U}. \end{aligned} \tag{3.7}$$

Hence, we get the following iteration, parameterized by ω:

$$\mathbf{x}^{k+1} = -(\frac{1}{\omega}\mathbf{D} + \mathbf{L})^{-1}((\frac{1}{\omega} - 1)\mathbf{D} - \mathbf{U})\mathbf{x}^k + (\frac{1}{\omega}\mathbf{D} + \mathbf{L})^{-1}\mathbf{c}. \tag{3.8}$$

By setting ω to 1 we recover Gauss-Seidel. We can adjust the parameter ω to minimize the spectral radius of the iteration matrix $\mathbf{T}_\omega = \mathbf{M}_\omega{}^{-1}\mathbf{N}_\omega$. Minimizing the largest eigenvalue of $\mathbf{T}_\omega$ is not a trivial task. However, for the Poisson discretization, [22] gives an optimal parameter which holds for any dimension in which we state the differential equation:

$$\omega_{\text{optimal}} = \frac{2}{1+\sin \pi h}, \text{ where } h \text{ is the mesh spacing, } h = \frac{1}{N}. \tag{3.9}$$

This optimal value of SOR for our model problem is obtained from a more general result of Young ([23], chapter 6):

Theorem 3.1.4. *For a matrix* $\mathbf{E}$*, let* $\mathbf{F}$ *be its Jacobi iteration matrix. If* $\mathbf{E}$ *is consistently ordered with nonvanishing diagonal elements and* $\rho(\mathbf{F}) < 1$*, then the optimal value for the parameter is* $\omega_{optimal} = \frac{2}{1+\sqrt{1-\rho(\mathbf{F})^2}}$.

The class of consistently ordered matrices is defined in ([23], definition 3.2) and it contains the block tri-diagonal matrices ([23], theorem 3.1), which naturally arise from certain orderings of elliptic PDEs discretizations ([3], 2.2.3).

The matrices we constructed so far are tri-diagonal and their diagonal elements are strictly positive. Furthermore, Jacobi converges for our model $\mathbf{A}$ (theorem 3.1.3). This happens if and only if the spectral radius of the Jacobi iteration matrix is strictly smaller than 1 (theorem 3.1.1), therefore conditions of theorem 3.1.4 hold. Therefore we can use the form $\omega_{\text{optimal}} = \frac{2}{1+\sqrt{1-\rho(\mathbf{J})^2}}$ (where $\mathbf{J}$ is the Jacobi iteration matrix of $\mathbf{A}$) to deduce (3.9) ([22]).

For a general problem, determining the spectral radius of the Jacobi iteration matrix is very expensive. There is no closed form for any eigenvalue of a general matrix, so numerical approximation methods need to be used in order to approximate the spectral radius.

The following result is an important characterization of the SOR method, following directly from ([14], 4.10). It is referred to as Ostrowski-Reich theorem ([12]):

Theorem 3.1.5. *For any symmetric positive definite* $\mathbf{A}$*, SOR converges for any initial solution* $\mathbf{x}^0$ *if and only if* ω *is in the range* $(0, 2)$.

The results for different values of ω in the range $(0, 2)$ can be seen in figure 3.4. For unsuitable choices of ω (yielding spectral radii close to 1) the iteration range is not enough to reach convergence.

The inversions and matrix multiplications in (3.8) do not need to be performed explicitly. The iteration can be written in an elegant way which takes advantage of the fast row access and product of the CSR format (see algorithm 2).

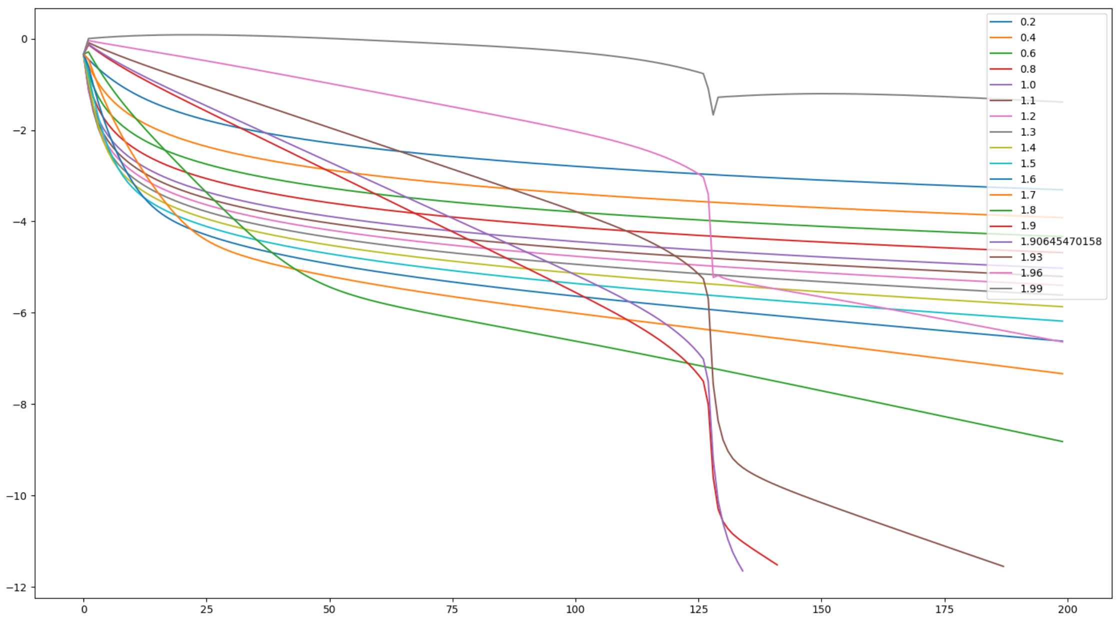


Figure 3.4: Logarithmic scale of the residual norm in terms of the number of SOR iterations for different ω values, for a grid size of $(N+1)\times(N+1)$, where $N = 64$. It can be seen that $\omega = \frac{2}{1+\sin \pi h} \approx 1.906455$ yields indeed the best convergence rate. For $\omega = 2$ the method would diverge, so for very close values to 2 the method behaves unnatural, as computational errors also intervene (e.g.: $\omega = 1.99$ in the figure).

3.1.4 SSOR

As noted in ([6], 11.2.7), we can see the SOR algorithm as updating $\mathbf{x}$ "top-to-bottom". This yields a non-symmetric iteration matrix, as the method itself is non-symmetric. It will be shown that the preconditioned Conjugate Gradient requires a symmetric preconditioner. Therefore, in the case of a Multigrid preconditioner (presented in section 4.8), a symmetric smoother (iteration matrix) is required. The natural approach to alleviate this is to update $\mathbf{x}$ twice each time: once top-to-bottom, and once bottom-to-top.

The bottom-to-top update described above is obtained by interchanging the roles of $\mathbf{U}$ and $\mathbf{L}$ in (3.8). We can describe the updates in the convenient way ([7], 4.1) given by (3.10):

$$\begin{aligned} \mathbf{x}^{(k+1/2)} &= -(\frac{1}{\omega}\mathbf{D}+\mathbf{L})^{-1}((\frac{1}{\omega}-1)\mathbf{D}-\mathbf{U})\mathbf{x}^{k} + (\frac{1}{\omega}\mathbf{D}+\mathbf{L})^{-1}\mathbf{c}, \\ \mathbf{x}^{k+1} &= -(\frac{1}{\omega}\mathbf{D}+\mathbf{U})^{-1}((\frac{1}{\omega}-1)\mathbf{D}-\mathbf{L})\mathbf{x}^{(k+1/2)} + (\frac{1}{\omega}\mathbf{D}+\mathbf{U})^{-1}\mathbf{c}. \end{aligned} \tag{3.10}$$

The cost of one SSOR iteration is exactly twice the cost of an SOR iteration. In general, SSOR converges slower than SOR, but its iteration matrix is symmetric

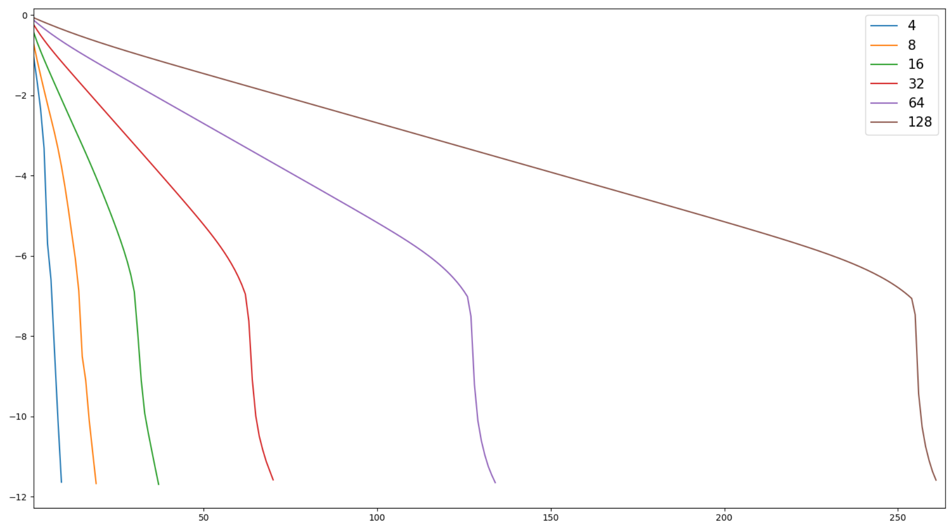


Figure 3.5: Logarithmic scale of the residual norm in terms of the number of SOR iterations for different N values, where the grid size is $(N+1) \times (N+1)$. The optimal ω is used for each N. There is clearly a strong mesh dependence of the convergence rate.

and thus will work as a smoother for a Multigrid preconditioner in preconditioned Conjugate Gradient.

Considering the nature of the subiterations in an SSOR iteration, we expect a similar result to (3.1.5) about the convergence of the method. Indeed, the following holds ([7], 4.2) :

Theorem 3.1.6. *Let $\mathbf{A} \in \mathbb{C}^{n,n}$ be a Hermitian matrix with positive diagonal elements (in our case, $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{A}$ symmetric suffice). SSOR method converges for any initial value of $\mathbf{x}^0$ if and only if $\mathbf{A}$ is positive definite and $\omega \in (0, 2)$.*

It is sensible that the optimal parameter ω_{SSOR} should be close to the optimal parameter for SOR, $\omega_{\text{SOR}} = \frac{2}{1+\sin(\pi h)}$. We approximated the optimal parameter for SSOR experimentally, and the results are in the following table.

Algorithm 2: Successive Over-Relaxation

```
    Data: $\mathbf{A}$, $\mathbf{x}_{\text{init}}$, $\mathbf{b}$, $\omega$, $iterationCount$
1  $\mathbf{x}_{\text{previous}}, \mathbf{x}_{\text{current}} \leftarrow \mathbf{x}_{\text{init}}$
2  $M \leftarrow size(\mathbf{A})$
3  for $iteration = 1; iteration \leq iterationCount; iteration{+}{+}$ do
4      $\mathbf{x}_{\text{current}} \leftarrow (0, 0, \cdots 0)$
5      for $i = 0; i < M; i{+}{+}$ do
6          currentRow $\leftarrow$ $\mathbf{A}$.getRow(i)
7          newUnweighted $\leftarrow$ (b[i] - $\mathbf{x}_{\text{current}}$[:i] $\cdot$ currentRow[:i] $-$
8              $\mathbf{x}_{\text{previous}}$[(i+1):] $\cdot$ currentRow[(i+1):]) / currentRow[i]
9          newWeighted $\leftarrow$ $\mathbf{x}_{\text{previous}}$[i] + $\omega$(newUnweighted $-$ $\mathbf{x}_{\text{previous}}$[i])
10     $\mathbf{x}_{\text{previous}} \leftarrow \mathbf{x}_{\text{current}}$
11 return $\mathbf{x}_{current}$
```

N	ω_{SOR}	# iterations ω_{SOR}	ω_{SSOR}	# iterations ω_{SSOR}
8	1.4465	19	1.503	19
16	1.6735	35	1.720	34
32	1.8215	61	1.852	59
64	1.9065	106	1.923	103
128	1.9521	182	1.961	176

Table 3.1: Comparing convergence of SSOR for the 2D Poisson model problem discretization when using the optimal ω of SOR against using a good approximation of the optimal ω of SSOR, which I discovered empirically.

A comparison of the convergence of SOR and SSOR methods using optimal parameters can be seen in figure 3.6. A comparison of how many floating point operations SOR and SSOR do until convergence on the model problem is shown in table 3.2.

N	# flops SOR	# flops SSOR
8	16440	32508
16	129732	244398
32	992154	1747308
64	7640730	12407892
128	59699844	85576050

Table 3.2: Comparing number of floating point operations of SOR and SSOR with optimal parameters ω_{SOR} and ω_{SSOR} for the convergence of the 2D Poisson model problem discretization.

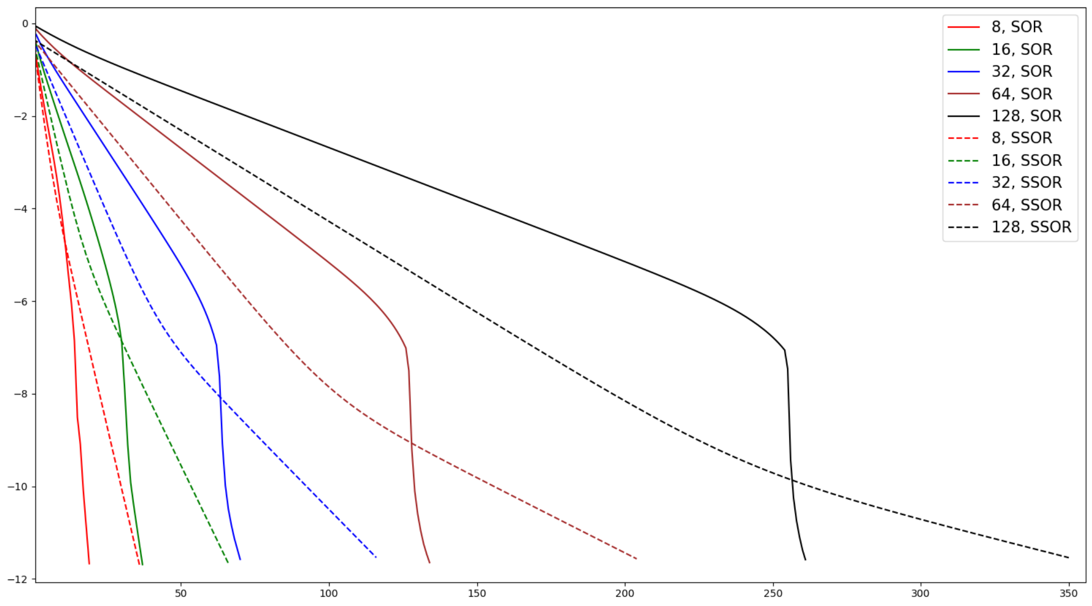


Figure 3.6: Logarithmic scale of the residual norm in terms of SOR iterations using the optimal ω_{SOR} for varying grid size of the 2D model problem discretization plotted using continuous lines. Logarithmic scale of the residual norm in terms of SSOR subiterations (2 subiterations per SSOR iteration) using the optimal ω_{SSOR} plotted using dashed lines for the same 2D model problem. The same colour corresponds to the same grid size in both methods. Notice that both SOR and SSOR graphs are plotted in terms of "SOR steps" (1 "SOR step" per SOR iteration, 2 "SOR steps" per SSOR iteration).

3.1.5 Comparison of Jacobi, SOR and SSOR

The (damped) Jacobi and Gauss-Seidel (SOR) have the same costs per iteration (approximately N additions/subtractions, N multiplications and 1 division). Gauss-Seidel converges in a smaller number of iterations in general, as intuitively each iteration has access to more updated information on the approximation (approximately twice faster than Jacobi, see figure 3.3).

An advantage of Jacobi is that it is highly parallelizable, as updating $\mathbf{x}^{k+1}$ depends only on $\mathbf{x}^k$. This is not true for Gauss-Seidel (SOR), as updating the $(i+1)^{\text{th}}$ component of $\mathbf{x}^{k+1}$ also depends on the first i components of $\mathbf{x}^{k+1}$. This is important for Jacobi as a solver on its own, but the Multigrid method (discussed in chapter 4) only requires a few SOR smoothing operations, so parallel smoothing is not necessarily required.

SSOR suffers from the same lack of parallelization as SOR. In terms of CPU operations, the dominant component of an SSOR iteration is given by 2 dot products, so twice the cost of an SOR iteration. In general, SOR with an optimal parameter requires fewer FLOPs than SSOR with optimal parameter to converge (see figure 3.6

and table 3.2). Intuitively, this is because there needs to be a trade-off when choosing the optimal parameter of SSOR between optimizing the top-to-bottom convergence rate and the bottom-to-top convergence rate, while for SOR there is no such trade-off.

However, the symmetric structure of SSOR makes it usable in preconditioned approximation methods which require symmetric smoothers, as we will see in chapter 4, section 4.8).

3.2 Gradient methods

The following theorem ([6], section 11.3.1) establishes a connection between computing a solution of $\mathbf{Ax} = \mathbf{b}$ for a symmetric positive definite matrix $\mathbf{A}$ and a convex optimization problem.

Theorem 3.2.1. *Suppose* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *is a symmetric positive definite,* $\mathbf{b} \in \mathbb{R}^n$ *and let the function* $\phi : \mathbb{R}^n \to \mathbb{R}$ *be* $\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}$*. Then, the solution* $\mathbf{x}^*$ *of* $\mathbf{Ax} = \mathbf{b}$ *is the same as the minimum point of* ϕ *(i.e.* $\arg\min_{x\in\mathbb{R}^n} \phi(\mathbf{x}) = \mathbf{x}^*$*). Furthermore, an iteration that produces a sequence of ever-better approximate minimizers for* ϕ *is an iteration that produces ever-better approximate solutions to* $\mathbf{Ax} = \mathbf{b}$ *(as measured in* $\mathbf{A}$*-norm).*

(The $\mathbf{A}$-norm of a vector $\mathbf{v}$ is defined as $\|\mathbf{v}\|_\mathbf{A} := \sqrt{\mathbf{v}^T\mathbf{A}\mathbf{v}}$. As $\mathbf{A}$ is positive definite, $\|\mathbf{v}\|_\mathbf{A} \geq 0$, with equality iff $\mathbf{v} = \mathbf{0}$.)

Proof. ϕ is a strictly convex function as $\nabla^2_\mathbf{x}\phi(\mathbf{x}) = \nabla_\mathbf{x}(\frac{1}{2}(\mathbf{Ax}+\mathbf{A}^T\mathbf{x})-\mathbf{b}) = \nabla_\mathbf{x}(\mathbf{Ax}-\mathbf{b}) = \mathbf{A}$ and $\mathbf{A}$ is positive definite. As $\mathbf{A}$ is positive definite, it is invertible (none of the eigenvalues can be 0), so there exists $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$. Then, $\nabla_\mathbf{x}\phi(\mathbf{x}^*) = \mathbf{A}\mathbf{x}^* - \mathbf{b} = \mathbf{0}$, so $\mathbf{x}^*$ is a stationary point of ϕ. As ϕ is strictly convex, $\mathbf{x}^*$ is the unique minimum point of ϕ, therefore $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b} = \arg\min_{x\in\mathbb{R}^n} \phi(\mathbf{x})$.

$$\begin{aligned}
\phi(\mathbf{x}^* + \mathbf{e}) - \phi(\mathbf{x}^*) &= \frac{1}{2}(\mathbf{x}^{*T} + \mathbf{e}^T)\mathbf{A}(\mathbf{x}^* + \mathbf{e}) - (\mathbf{x}^{*T} + \mathbf{e}^T)\mathbf{b} - (\frac{1}{2}\mathbf{x}^{*T}\mathbf{A}\mathbf{x}^* - \mathbf{x}^{*T}\mathbf{b}) \\
&= \frac{1}{2}\mathbf{x}^{*T}\mathbf{A}\mathbf{e} + \frac{1}{2}\mathbf{e}^T\mathbf{A}\mathbf{x}^* + \frac{1}{2}\mathbf{e}^T\mathbf{A}\mathbf{e} - \mathbf{e}^T\mathbf{b} \\
&= \frac{1}{2}(\mathbf{b}^T(\mathbf{A}^{-1})^T\mathbf{A}\mathbf{e} + \mathbf{e}^T\mathbf{A}\mathbf{A}^{-1}\mathbf{b} + \mathbf{e}^T\mathbf{A}\mathbf{e}) - \mathbf{e}^T\mathbf{b} \\
&= \frac{1}{2}\mathbf{e}^T\mathbf{A}\mathbf{e} = \frac{1}{2}\|\mathbf{e}\|_\mathbf{A}
\end{aligned} \tag{3.11}$$

Therefore $\phi(\mathbf{x}) = \phi(\mathbf{x}^*) + \frac{1}{2}\|\mathbf{x} - \mathbf{x}^*\|_{\mathbf{A}}$ for any vector $\mathbf{x}$. This establishes the second part of the theorem: the closer $\mathbf{x}$ is to minimizing ϕ, the closer $\mathbf{x}$ is to $\mathbf{x}^*$ under $\mathbf{A}$-norm.

$\square$

We use the classic result that all norms on $\mathbb{R}^n$ are "equivalent", so convergence under $\mathbf{A}$-norm is the same as convergence under any norm ([6], 2.2.4):

Theorem 3.2.2. *If $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ are norms on $\mathbb{R}^n$, then there exist positive constants c_1 and c_2 such that $c_1\|x\|_\alpha \leq \|x\|_\beta \leq c_2\|x\|_\alpha$.*

Theorem 3.2.1 enables the use of a class of (convex) optimizing techniques called "gradient" methods for numerically approximating solutions of our model problems. These methods seek to minimize convex functions by starting from an initial point and moving in a sequence of directions until the gradient in the current point is close enough to 0.

3.2.1 Steepest Descent

Steepest Descent is a greedy gradient descent algorithm used to find the global minimum of the convex function ϕ by "sliding" through a series of points. Recall $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$ is the residual error of the approximation $\mathbf{x}_k$.

It starts at a point $\mathbf{x}_0$ and moves in the direction of the steepest descent: $-\nabla_{\mathbf{x}}\phi(\mathbf{x}_0) = \mathbf{b} - \mathbf{A}\mathbf{x}_0 = \mathbf{r}_0$. The update can be described by $\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{r}_0$, where $\alpha \in \mathbb{R}$ is chosen to minimize the value of ϕ at $\mathbf{x}_1$: let $g(\alpha) = \phi(\mathbf{x}_0 + \alpha\mathbf{r}_0)$ and set $\frac{dg}{d\alpha} = 0$ in order to find that the optimal value of α is $\alpha_0 = \frac{\mathbf{r}_0^T\mathbf{r}_0}{\mathbf{r}_0^T\mathbf{A}\mathbf{r}_0}$. Iterate the update rule $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{r}_k$ to get a series of better approximations of the optimal point.

The convergence condition of the while loop in the algorithm 3 is tested as mentioned in the beginning of the thesis: it converges if the residual relative error gets below a threshold.

For a general result about the convergence of Steepest Descent, we need the notion of condition number for a matrix (it is defined more generally, using Singular Value Decomposition, but we restrict to matrices we encounter in the definition below).

Definition 3.2.1. *The condition number of a real, invertible matrix $\mathbf{A}$ is $k(\mathbf{A}) = \frac{|\lambda_{max}|}{|\lambda_{min}|}$, where λ_{max} and λ_{min} are the eigenvalues of $\mathbf{A}$ with the maximum, respective minimum absolute values.*

Algorithm 3: Steepest Descent

```
Data: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}_0$
/* We approximate x in $\mathbf{Ax} = \mathbf{b}$                          */
/* $\mathbf{A}$ is a symmetric positive definite matrix                     */
/* $\mathbf{x}_0$ is the initial solution, if not given we take it to be full of zero */
1 $k \leftarrow 0$
2 while not convergence do
3     $\mathbf{r}_k \leftarrow \mathbf{b} - \mathbf{Ax}_k$
4     $\alpha_k = (\mathbf{r}_k^T\mathbf{r}_k)/(\mathbf{r}_k^T\mathbf{Ar}_k)$
5     $\mathbf{x}_{(k+1)} = \mathbf{x}_k + \alpha_k\mathbf{r}_k$
6     $k \leftarrow k + 1$
7 return $\mathbf{x}_k$
```

We will see that the closer the condition number of $\mathbf{A}$ is to 1, the better convergence bounds for both Steepest Descent and Conjugate Gradient we get. We call matrices with condition number close to 1 *well-conditioned* and matrices with condition number considerably greater than 1 *ill-conditioned*.

The following result characterizes the convergence of the Steepest Descent method ([6],11.3.9):

Theorem 3.2.3. *For any symmetric positive definite problem* $\mathbf{Ax} = \mathbf{b}$*, if* $\mathbf{x}_{k+1}$ *is obtained by a Steepest Descent step from* $\mathbf{x}_k$ *and* $\mathbf{x}^*$ *is the true solution, we have:* $\|\mathbf{x}_{k+1} - \mathbf{x}^*\|_{\mathbf{A}}^2 \leq (1 - \frac{1}{k(\mathbf{A})})\|\mathbf{x}_k - \mathbf{x}^*\|_{\mathbf{A}}^2$

As $k(\mathbf{A}) \geq 1$, theorem 3.2.3 implies the convergence of the method. However, unless $k(\mathbf{A})$ is small enough, the convergence of the method is slow. The behaviour of the method is best illustrated on a 2×2 linear equation. Let us use Steepest Descent to approximate a solution for $\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$ (the equation matrix is symmetric and positive definite and the exact solution is $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$) - see figure 3.7.

The performance of this method on the discretization of the 2D Poisson Equation sample problem used so far is shown in figure 3.8. The convergence rate flattens as the dimension grows, making steepest descent a fairly slow method.

3.2.2 Conjugate Gradient Method

“When it is obvious that the goals cannot be reached, don’t adjust the goals, adjust the action steps.”

(Confucius)

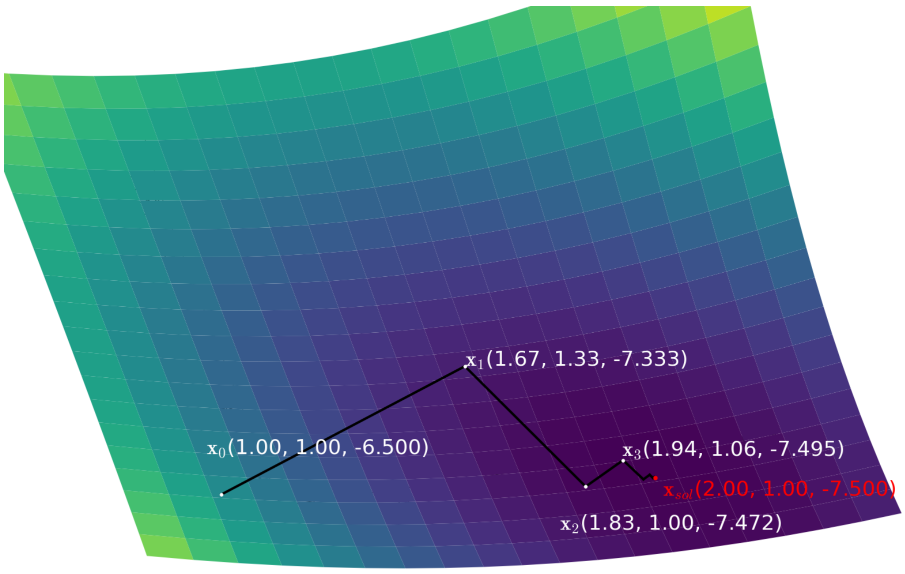


Figure 3.7: The first three steps ($\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$) *of Steepest Descent, the initial guess* ($\mathbf{x}_0 = [1,1]^T$) *and the minimum point* ($\mathbf{x}_{sol}$) *are shown on the plot of the example function* $\phi_1(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x} - \mathbf{x}^T \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. *The method converges in* 14 *steps, but only the first ones are visible at this image scale. This greedy method yields a zigzag movement towards the optimal point of the associated convex function* ϕ_1.

The weakness of Steepest Descent is that the greedy choice of directions results in making repeated moves in the same direction (see figures 3.7, 3.8). An idea to alleviate this is to use a set of mutually orthogonal vectors $\mathbb{D} = \{\mathbf{d}_0, \mathbf{d}_1, \cdots, \mathbf{d}_{n-1}\}$ and to iteratively compute step lengths α_k for each search direction $\mathbf{d}_k$, updating $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$. The goal is to choose the coefficients such that convergence is guaranteed in at most n steps (as $\mathbb{D}$ is a basis over $\mathbb{R}^n$, the exact solution $\mathbf{x}^*$ can be written as $\mathbf{x}^* = \mathbf{x}_0 + \sum_{i=0}^{i=n-1} \beta_i \mathbf{d}_i$ for suitable real coefficients β_i).

Not all details of deducing an exact formulation of Conjugate Gradient are presented here (see [8] for the original formulation, [16] for an intuitive approach and [6], chapter 11). The optimality condition of this method is to choose α_i such that $\mathbf{e}_{i+1}$ is orthogonal to $\mathbf{d}_i$. However, using the Euclidean norm yields $\alpha_i = \frac{\mathbf{d}_i^T \mathbf{e}_i}{\mathbf{d}_i^T \mathbf{d}_i}$ and this requires $\mathbf{e}_i = \mathbf{x}^* - \mathbf{x}_{(i)}$, which we cannot determine.

The trick is to use the $\mathbf{A}$-norm for orthogonality ($\mathbf{v}$ and $\mathbf{w}$ are $\mathbf{A}$-orthogonal if $\mathbf{v}^T \mathbf{A} \mathbf{w} = 0$) : $\mathbb{D}$ is a set of mutually $\mathbf{A}$-orthogonal vectors and we pick coefficients α_i such that $\mathbf{e}_{i+1}^T \mathbf{A} \mathbf{d}_i = 0$. As $\mathbf{A}\mathbf{e}_k = \mathbf{r}_k$, we get:

$$\alpha_i = \frac{\mathbf{d}_i^T \mathbf{r}_i}{\mathbf{d}_i^T \mathbf{A} \mathbf{d}_i}. \tag{3.12}$$

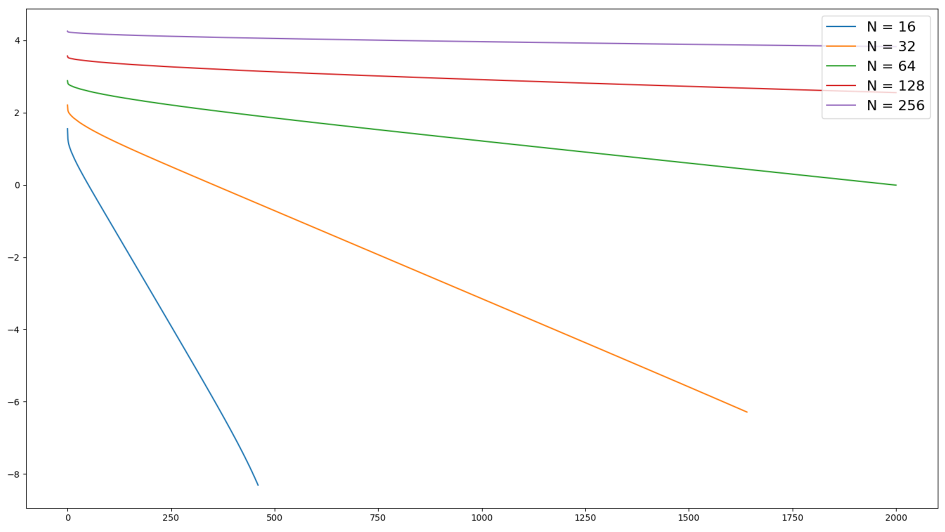


Figure 3.8: Logarithmic scale of the exact error $\mathbf{e}_k$ Euclidean norm of the Steepest Descent method on the sample 2D Poisson equation. The plot is relevant to see how fast we are approaching the actual solution, but in general this quantity is not available. We have chosen the functions in the model 2D Poisson equation such that the exact solution has an explicit form which we can use to compute $\mathbf{e}_k$. However, we are not cheating on the convergence condition: the algorithm stops once the relative residual norm is below a tolerance threshold of 10^{-5} (or when the iteration count gets to 2000).

Each step of the algorithm developed so far eliminates one term of the sum in $\mathbf{x}^* = \mathbf{x}_0 + \sum_{i=0}^{i=n-1} \beta_i \mathbf{d}_i$, so $\mathbf{e}_k = \sum_{i=k}^{i=n-1} \beta_i \mathbf{d}_i$ (β_i's are the coefficients of $\mathbf{x}^* - \mathbf{x}_0$ in the base $\mathbb{D}$), so $\alpha_i = \beta_i$, for any i. By multiplying to the left the last equation with $\mathbf{d}_j^T \mathbf{A}$ for a $j < k$, we get the nice property that $\mathbf{d}_j^T \mathbf{A} \mathbf{e}_k = \sum_{i=k}^{i=n-1} \beta_i \mathbf{d}_j^T \mathbf{A} \mathbf{d}_i = 0$, because of the $\mathbf{A}$-orthogonality of the directions, so

$$\mathbf{d}_j^T \mathbf{A} \mathbf{e}_k = \mathbf{d}_j^T \mathbf{r}_k = 0, \text{ for any } 0 \le j < k \le n-1. \tag{3.13}$$

We are left to determine a set $\mathbb{D}$ of n mutually $\mathbf{A}$-orthogonal vectors. This can be done by a modified version of the Gram-Schmidt orthogonalization process (see [1] and [16], 7.2): start with any set of n independent vectors $\mathbb{V} = \{\mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_{n-1}\}$, for $i = 0, 1, \ldots, n-1$ construct $\mathbf{d}_i$ by eliminating from $\mathbf{v}_i$ the $\mathbf{d}_0 \cdots \mathbf{d}_{i-1}$ components under $\mathbf{A}$-projection of $\mathbf{v}_i$. Start with $\mathbf{d}_0 = \mathbf{v}_0$ and, for $i > 0$, construct

$$\mathbf{d}_i = \mathbf{v}_i - \sum_{k=0}^{i-1} \delta_{ik} \mathbf{d}_k. \tag{3.14}$$

By taking the dot product of $\mathbf{d}_i$ and $\mathbf{A}\mathbf{d}_j$ and using $\mathbf{d}_i^T \mathbf{A} \mathbf{d}_j = 0$ for $i \ne j$, we get $0 = \mathbf{v}_i^T \mathbf{A} \mathbf{d}_j - \delta_{ij} \mathbf{d}_j^T \mathbf{A} \mathbf{d}_j$, so $\delta_{ij} = \frac{\mathbf{v}_i^T \mathbf{A} \mathbf{d}_j}{\mathbf{d}_j^T \mathbf{A} \mathbf{d}_j}$. This construction requires storing all previous search directions and it is no better than the Gaussian elimination, requiring $\mathcal{O}(n^3)$ operations in general.

This approach is known as *The Method of Conjugate Directions* (see [8] or [16], 7) and is not much better in practice than a solver one using Guassian elimination. Its main improvement comes from M. Hestenes and E. Stiefel and was published in the famous paper [8]: "Methods of Conjugate Gradients for Solving Linear Systems". The idea is to construct the directions by conjugating the residuals (i.e. apply on the go the Gram-Schmidt process above to the $\mathbf{r}_k$'s obtained so far). For such a construction, we denote $\mathbb{D}_k = \text{span}\{\mathbf{d}_0, \cdots, \mathbf{d}_{k-1}\} = \text{span}\{\mathbf{r}_0, \cdots, \mathbf{r}_{k-1}\}$ for any $1 \leq k \leq n-1$, and by (3.13) we get that $\mathbf{r}_k^T\mathbf{d}_i = 0$, for all $i < k$, so $\mathbf{r}_k$ is orthogonal to $\mathbb{D}_{k-1}$, and therefore linearly independent to $\{\mathbf{d}_0, \cdots, \mathbf{d}_{k-1}\}$ and $\{\mathbf{r}_0, \cdots, \mathbf{r}_{k-1}\}$. This means that the residuals are indeed independent (in fact, they are even orthogonal) and the Gram-Schmidt process for obtaining a basis of vectors which are mutually **A**-orthogonal works.

Note that

$$\mathbf{r}_{k+1} = \mathbf{A}\mathbf{e}_{k+1} = \mathbf{A}(\mathbf{e}_k - \beta_k\mathbf{d}_k) = \mathbf{r}_k - \beta_k\mathbf{A}\mathbf{d}_k, \tag{3.15}$$

so $\mathbf{r}_{k+1}$ is a linear combination of $\mathbf{r}_k$ and $\mathbf{A}\mathbf{d}_k$, so $\mathbb{D}_{k+1} = \mathbb{D}_k \bigcup \mathbf{A}\mathbb{D}_k$. Hence, we get ([16], 8):

$$\mathbb{D}_k = \text{span}\{\mathbf{d}_0, \mathbf{A}\mathbf{d}_0, \cdots, \mathbf{A}^{k-1}\mathbf{d}_0\} = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \cdots, \mathbf{A}^{k-1}\mathbf{r}_0\}. \tag{3.16}$$

Observation 3.2.1. $\mathbb{D}_k = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \cdots, \mathbf{A}^{k-1}\mathbf{r}_0\}$ *is called a "Krylov subspace" ([6], 11.3.3), and many authors (including Golub in [6]) deduce the Conjugate Gradient formulation as a "Subspace Strategy", by successively finding the optimal point in each Krylov subspace and, as the last such subspace* $\mathbb{D}_{n-1} = \mathbb{R}^n$*, the optimal point which belongs to it is the solution to our problem. The layout of the deduction here is inspired by Shewchuk's one in [16].*

Recall the coefficients in the Gram-Schmidt process are $\delta_{ij} = \frac{\mathbf{v}_i^T\mathbf{A}\mathbf{d}_j}{\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j}$, where we chose $\mathbf{v}_i = \mathbf{r}_i$. From (3.15), we get $\mathbf{r}_i^T\mathbf{r}_{k+1} = \mathbf{r}_i^T\mathbf{r}_k - \beta_k\mathbf{r}_i^T\mathbf{A}\mathbf{d}_k$, so for $i \neq k$ and $i \neq k+1$ we have $\beta_k\mathbf{r}_i^T\mathbf{A}\mathbf{d}_k = 0$, hence $\delta_{ik} = 0$. For $i = k+1$ we get $\mathbf{r}_{k+1}^T\mathbf{r}_{k+1} = \beta_k\mathbf{r}_{k+1}^T\mathbf{A}\mathbf{d}_k$, so $\delta_{k+1k} = -\frac{1}{\beta_k}\frac{\mathbf{r}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{d}_k\mathbf{A}\mathbf{d}_k}$ are the only non-zero coefficients in the expressions which construct the search directions $\mathbf{d}_i = \mathbf{v}_i - \sum_{k=0}^{i-1}\delta_{ik}\mathbf{d}_k$, so, by also using (3.12) and the fact that $\alpha_i = \beta_i$, the direction update rule is described by:

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \frac{\mathbf{r}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{d}_k^T\mathbf{r}_k}\mathbf{d}_k. \tag{3.17}$$

By taking the dot product of $\mathbf{d}_i$ and (3.14), we get $\mathbf{d}_i^T\mathbf{d}_i = \mathbf{d}_i^T\mathbf{v}_i$ (for any i), where $\mathbf{v}_i = \mathbf{r}_i$ under Conjugate Gradient choices of $\mathbf{v}$'s, so $\alpha_i = \frac{\mathbf{d}_i^T\mathbf{r}_i}{\mathbf{d}_i^T\mathbf{A}\mathbf{d}_i} = \frac{\mathbf{d}_i^T\mathbf{d}_i}{\mathbf{d}_i^T\mathbf{A}\mathbf{d}_i}$ and (3.18) can be written in the final, widely used form:

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \frac{\mathbf{r}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{r}_k^T\mathbf{r}_k}\mathbf{d}_k = \mathbf{r}_{k+1} + \gamma_k\mathbf{d}_k. \tag{3.18}$$

We can now put together a method which reaches the exact solution (neglecting precision errors) in at most n steps for any symmetric positive definite matrix in $\mathbb{R}^{n\times n}$. It is often referred to as being a *direct solver*, similar to Gaussian elimination. The Conjugate Gradient method is given by algorithm 4. However, we try to stop the method before reaching the n steps limit, as for very large systems even n iterations are not feasible. We test the convergence as before, by checking if the relative residual error is below a small threshold.

Algorithm 4: The Conjugate Gradient Algorithm

```
   Data: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}_0$
   /* We approximate x in $\mathbf{Ax} = \mathbf{b}$                                  */
   /* $\mathbf{A}$ is a symmetric positive definite matrix                       */
   /* $\mathbf{x}_0$ is the initial solution, if not given we take it to be full of zero */
 1 $k \leftarrow 0$
 2 $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
 3 $\mathbf{d}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$
 4 while not convergence do
 5     $\alpha_k \leftarrow \frac{\mathbf{d}_k^T\mathbf{d}_k}{\mathbf{d}_k^T\mathbf{A}\mathbf{d}_k}$
 6     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k\mathbf{d}_k$
 7     $\mathbf{r}_{k+1} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_{k+1}$
 8     $\gamma_k \leftarrow \frac{\mathbf{r}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{r}_k^T\mathbf{r}_k}$
 9     $\mathbf{d}_{k+1} \leftarrow \mathbf{r}_{k+1} + \gamma_k\mathbf{d}_k$
10     $k \leftarrow k + 1$
11 return $\mathbf{x}_k$
```

The main result about Conjugate Gradient's convergence is given by [21] , theorem 38.5:

Theorem 3.2.4. *Let the Conjugate Gradient iteration be applied to a symmetric positive definite matrix problem* $\mathbf{Ax} = \mathbf{b}$*. The* $\mathbf{A}$*-norms of the errors satisfy:*

$$\frac{\|\mathbf{e}_n\|_\mathbf{A}}{\|\mathbf{e}_0\|_\mathbf{A}} \leq 2\left(\frac{\sqrt{k(\mathbf{A})}-1}{\sqrt{k(\mathbf{A})}+1}\right)^n .$$

Conjugate Gradient is performing much better in practice than Steepest Descent (see table 3.3 for an experimental comparison of the two methods and figure 3.9 for how CG eliminates error components on an example problem). In the light of theorem 3.2.4, it is obvious that the closer the condition number of $\mathbf{A}$ is to 1, the better convergence bound we get.

N	# iterations SD	# iterations CG	# flops SD	# flops CG
8	124	20	174098	25199
16	460	38	2743858	187477
32	1640	74	40484818	1448801
64	5710	145	574008898	11317982
128	19560	278	7937609298	86671205

Table 3.3: Comparing number of floating point operations and the iteration count of SD and CG for convergence of the 2D Poisson model problem discretization.

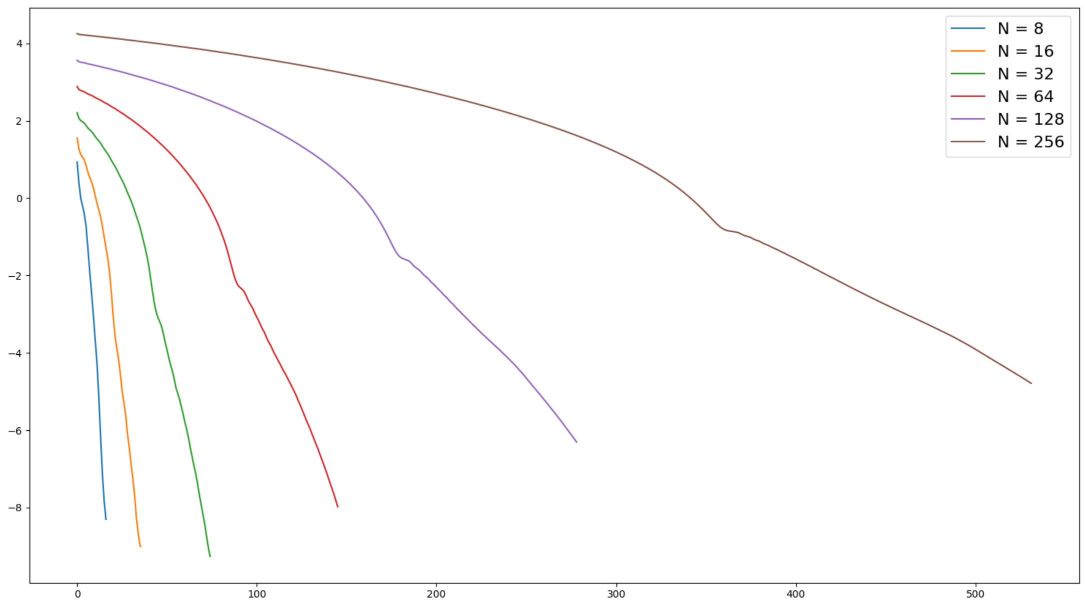


Figure 3.9: Logarithmic scale of the exact error $\mathbf{e}_k$ Euclidean norm of the Conjugate Gradient method on the same sample 2D Poisson equation as before. As in figure 3.8, the exact errors are not available in general, but in our case we know an explicit solution and it helps to illustrate how the method eliminates at each step an error component. Again, we are not cheating on the convergence condition: the algorithm stops once the relative residual norm is below a tolerance threshold of 10^{-5}.

3.2.2.1 Asymptotic behaviour

Multiplying a sparse matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with m nonzero entries (using the CSR format) and a vector can be done in $\mathcal{O}(m+n)$. Using this and (3.2.3) we can show that Steepest Descent converges in $\mathcal{O}((m+n)\cdot k(\mathbf{A}))$ for any symmetric positive definite $\mathbf{A}$. Similarly, using (3.2.4), we can show that Conjugate Gradient converges in $\mathcal{O}((m+n)\cdot \sqrt{k(\mathbf{A})})$.

For matrices $\mathbf{A} \in \mathbb{R}^{n\times n}$ arising in our discretizations, $m = \mathcal{O}(n)$. Article [16] notes that finite difference methods applied on second-order elliptic boundary value problems (such as our model differential equations) on d-dimensions often have the condition number of the arising matrix $k \in O(n^{2/d})$.

Summing it up, on the 1D model equations, Steepest Descent converges in $\mathcal{O}(n^3)$ and Conjugate Gradient converges in $\mathcal{O}(n^2)$. On the 2D model equations, Steepest Descent converges in $\mathcal{O}(n^2)$ and Conjugate Gradient converges in $\mathcal{O}(n\sqrt{n})$.

In the discussion above, recall that n is the arising number of variables in the discretization. So in 1D, for an N spaced grid, $n \approx N$ and in 2D, for an N spaced grid on each axis, $n \approx N^2$.

3.2.2.2 Preconditioned Conjugate Gradient Method

We have seen in theorem 3.2.4 that the convergence is faster when the condition number of the matrix is closer to 1. It is natural to ask whether a symmetric positive definite linear problem $\mathbf{Ax} = \mathbf{b}$ can be transformed to another symmetric positive definite linear problem $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ such that $\mathbf{x}$ is "easy" to obtain from $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{A}}$ is better-conditioned than $\mathbf{A}$ ($k(\tilde{\mathbf{A}}) < k(\mathbf{A})$). This is possible via *preconditioning.*

We adopt the construction from [6], chapter 11.5, below: let $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$ be a nonsingular matrix. Note that $\mathbf{Ax} = \mathbf{b} \iff \mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}\mathbf{M}_2\mathbf{x} = \mathbf{M}_1^{-1}\mathbf{b}$. Consider the linear system $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, where $\tilde{\mathbf{A}} = \mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}$ and $\tilde{\mathbf{b}} = \mathbf{M}_1^{-1}\mathbf{b}$. We can solve this by Conjugate Gradient and then determine $\mathbf{x}$ from $\mathbf{M}_2\mathbf{x} = \tilde{\mathbf{x}}$. A requirement of choosing $\mathbf{M}$ is that it should "approximate" $\mathbf{A}$, such that $\tilde{\mathbf{A}} \approx \mathbf{I}$ (so the resulting equation is "easier" to solve).

It is a classic result that for a symmetric positive definite matrix $\mathbf{M}$ there exists a unique symmetric positive definite $\mathbf{C}$ such that $\mathbf{M} = \mathbf{C}^2$. Choose such an $\mathbf{M}$ and let $\mathbf{M}_1 = \mathbf{M}_2 = \mathbf{C}$. This preserves the symmetry and positive definiteness of the problem $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. The advantage of such a preconditioning is that the resulting Conjugate Gradient method only involves $\mathbf{M}$ (after some computational simplifications, there is no occurrence of $\mathbf{C}$ in the update rules) and has the form described in algorithm 5 (see [6], algorithm 11.5.1 and [20], program 1).

Algorithm 5: The Preconditioned Conjugate Gradient Algorithm

```
   Data: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{x}_0$, $\mathbf{M}$
   /* We approximate x in $\mathbf{Ax} = \mathbf{b}$                                    */
   /* $\mathbf{A}$ is a symmetric positive definite matrix                              */
   /* $\mathbf{x}_0$ is the initial solution, if not given we take it to be full of zero */
   /* $\mathbf{M}$ is the preconditioning matrix.  We will see later that, in practice,
      instead of passing it as a parameter, we just simulate its inverse action.
      */
1  $k \leftarrow 0$
2  $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{Ax}_0$
3  $\mathbf{d}_0 \leftarrow \mathbf{b} - \mathbf{Ax}_0$
4  while not convergence do
5      $\alpha_i \leftarrow \frac{\tilde{\mathbf{r}}_i^T \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}$
6      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$
7      $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$
8      Solve $\mathbf{M}\tilde{\mathbf{r}}_{k+1} = \mathbf{r}_{k+1}$
9      $\beta_k \leftarrow \frac{\tilde{\mathbf{r}}_{k+1}^T \mathbf{r}_{k+1}}{\tilde{\mathbf{r}}_k^T \mathbf{r}_k}$
10     $\mathbf{p}_{k+1} \leftarrow \tilde{\mathbf{r}}_{k+1} + \beta_k \mathbf{p}_k$
11     $k \leftarrow k + 1$
12 return $\mathbf{x}_k$
```

Many linear solvers applied for a small, constant number of iterations yield good preconditioners for Conjugate Gradient. By applying k_{it} iterations of a certain linear method $\mathbb{L}$ to $\mathbf{Ax} = \mathbf{b}$ we derive $\mathbf{S}^{-1}\mathbf{Ax} = \mathbf{S}^{-1}\mathbf{b}$, where $\mathbf{S}^{-1}\mathbf{A}$ is "closer" to the identity matrix $\mathbf{I}$ (therefore, has a smaller condition number) than $\mathbf{A}$, so $\mathbf{S}^{-1}\mathbf{b}$ is an approximation of $\mathbf{x}$ ($\mathbf{S}^{-1}$ simulates the effect of the approximation method by pre-multiplying a vector, in this case $\mathbf{b}$, with it). Pick $\mathbf{M} = \mathbf{S}$ and assume for the moment that $\mathbf{S}$ is symmetric positive definite. This means that solving the linear system on line 8 in algorithm 5 ($\mathbf{M}\tilde{\mathbf{r}}_{k+1} = \mathbf{r}_{k+1}$) can be done by applying the same method $\mathbb{L}$ for k_{it} iterations on the system $\mathbf{Ay} = \mathbf{r}_{k+1}$, as the effect of this is $\mathbf{S}^{-1}\mathbf{Ay} = \mathbf{S}^{-1}\mathbf{r}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$, which is the solution of the aforementioned system on line 8.

Jacobi and SSOR are such linear methods, which yield symmetric positive definite matrices $\mathbf{S}$ (see [6], 11.5 for valid preconditioners). In the next chapter, we present a powerful method obtained by preconditioning with Multigrid the Conjugate Gradient algorithm.

Chapter 4

The Multigrid Method

4.1 Motivation

We have seen that Jacobi, SOR and SSOR all have the smoothing property, meaning that they eliminate fast the high frequency (oscillatory) components of the error and perform badly after on the remaining, lower frequency components. We call these methods *relaxation schemes* or *smoothers*.

The intuition for Multigrid (MG) methods comes from noting that a low frequency on a certain grid size appears to be more oscillatory on a coarser grid (1D example in figure 4.1). Therefore, we can apply one of the relaxation schemes above to a certain problem until we are left mainly with low frequency components and the relaxation starts to stagnate. Then, we can move the problem to a coarser grid, where the remaining error components appear more oscillatory, and continue with the relaxation scheme, which will perform better on this coarser grid. If the dimension of the coarsened grid is small enough, we can use a direct solver on it.

Suppose we want to approximate a solution for any model problem on the grid of size N: Ω^h (in 1D case this is the evenly spaced grid of size N, and in the 2D case this is the evenly spaced grid of size $N \times N$). Let the linear problem arising from finite differences be $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ and let us apply a relaxation scheme μ_1 times and obtain an approximate $\mathbf{x}^h_{\mu_1}$ such that we diminish the high frequency components of the error. Note that $\mathbf{A}^h\mathbf{e}^h = \mathbf{A}^h(\mathbf{x}^h - \mathbf{x}^h_{\mu_1}) = \mathbf{b}^h - \mathbf{A}^h\mathbf{x}^h_{\mu_1} = \mathbf{r}^h$, so we wish to solve $\mathbf{A}^h\mathbf{e}^h = \mathbf{r}^h$, where $\mathbf{e}^h$ contains mainly low frequency components. We should try to solve/approximate this problem on a coarser grid, as the low frequency of $\mathbf{e}^h$ cannot be captured by a smoother on Ω^h and they appear to be more oscillatory on a coarser grid, and then move the result back to Ω^h and combine the approximation of $\mathbf{e}^h$ and $\mathbf{x}^h_{\mu_1}$ to get a better estimate of the solution of $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$.

In order to make this method feasible, we need to define how to transfer information (matrices, vectors) between grids.

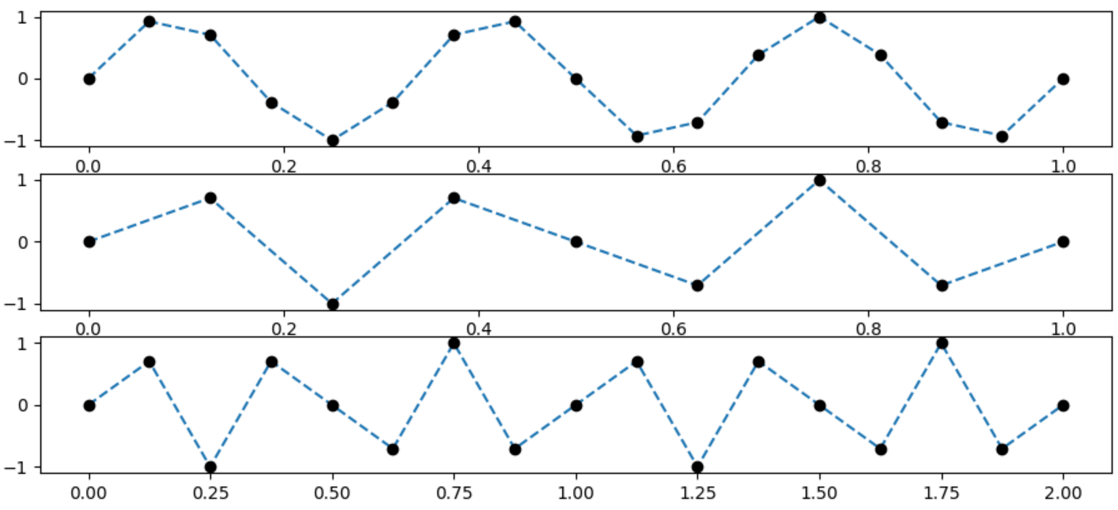


Figure 4.1: The first plot represents $\sin 6\pi x$ *on the grid* Ω^h*, where* $h = \frac{1}{16}$*. The second plot represents the same function on the coarser grid* Ω^{2h}*. The third plot represents the same function on a grid of spacing* $2h$ *on the interval* $[0, 2]$*. Note that its restriction to* $[0, 1]$ *is the same as the second plot, but the* x*-axis is shown at a different scale, such that the arising grid is "aligned" with the grid in the first plot and we can compare them. It is clear that the third plot, hence the second as well, show more oscillatory functions. Notice that the projection of* $\sin 6\pi x$ *has wave number* 6 *on both grids. However, on* Ω^h *it is the* 6^{th} *mode out of* 16 *modes, and on* Ω^{2h} *it is the* 6^{th} *mode out of* 8 *modes.*

4.2 Prolongation and restriction operators

We will only be interested in information transfers of two kinds: from Ω^{2h} to Ω^h (*prolongation* or *interpolation*) and from Ω^h to Ω^{2h} (*restriction* or *coarsening*). We are lucky enough to have a simple transfer for the matrix involved in the linear equation $\mathbf{A}$, as we can just apply the discretization process on any grid size, so we can easily obtain $\mathbf{A}^h$ and $\mathbf{A}^{2h}$. Therefore, we are interested in how to transfer vectors (representing discretizations of continuous functions) between grids. We mainly adopt the notations and definitions from [4], chapter 3 in the rows below.

Suppose that all the involved grid sizes are powers of 2 for convenience. We will see that the way Multigrid works means that the interpolation and restriction operators will always be passed discretizations with value 0 everywhere on the borders.

4.2.1 Interpolation

We define the interpolation from Ω^{2h} to Ω^h as a linear operator $\mathbf{I}_{2h}^h$ that takes a vector $\mathbf{v}^{2h}$ representing the discretization of a function on the grid Ω^{2h} and returns a vector $\mathbf{v}^h = \mathbf{I}_{2h}^h \mathbf{v}^{2h}$ of the discretized interpolation, on grid Ω^h. We use linear interpolation.

Let $h = \frac{1}{N}$ and $\mathbf{v}^{2h} = [v_0, v_1, \cdots, v_{N/2}]$ be a vector representing a 1D discretization on grid Ω^{2h}. Its linear interpolation to grid Ω^h is $\mathbf{v}^h = [w_0, w_1, \cdots, w_N]$ (see figure 4.2), where:

$$\begin{aligned} w_{2k} &= v_k, k = 0, 1, \dots, N/2, \\ w_{2k+1} &= \frac{v_k + v_{k+1}}{2}, k = 0, 1, \dots, N/2 - 1. \end{aligned} \tag{4.1}$$

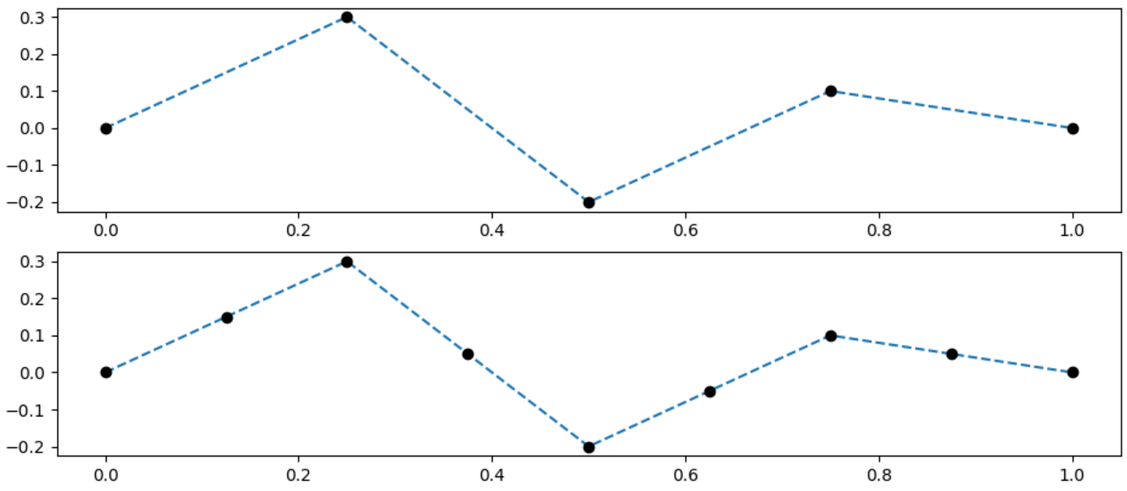


Figure 4.2: Linearly interpolating a discretized function from grid $\Omega^{1/4}$ to grid $\Omega^{1/8}$.In this case $h = 1/8$ and $\mathbf{I}_{2h}^h = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$.

Similarly, let $\mathbf{v}^{2h} = [v_{0,0}, v_{0,1}, \cdots v_{0,N/2}, \cdots v_{N/2,0}, v_{N/2,1}, \cdots v_{N/2,N/2}]$ be the lexicographically ordered vector representing a 2D discretization on grid Ω^{2h}. Its linear interpolation to grid Ω^h is $\mathbf{v}^h = [w_{0,0}, w_{0,1}, \cdots w_{0,N}, \cdots w_{N,0}, w_{N,1}, \cdots w_{N,N}]$ where:

$$\begin{aligned} w_{2i,2j} &= v_{i,j}, i, j = 0, 1, \dots, N/2, \\ w_{2i+1,2j} &= \frac{v_{i,j} + v_{i+1,j}}{2}, i = 0, 1, \dots, N/2 - 1 \text{ and } j = 0, 1, \dots, N/2, \\ w_{2i,2j+1} &= \frac{v_{i,j} + v_{i,j+1}}{2}, i = 0, 1, \dots, N/2 \text{ and } j = 0, 1, \dots, N/2 - 1, \\ w_{2i+1,2j+1} &= \frac{v_{i,j} + v_{i+1,j} + v_{i,j+1} + v_{i+1,j+1}}{4}, i, j = 0, 1, \dots, N/2 - 1. \end{aligned} \tag{4.2}$$

Considering the border values fixed with value 0, we can see the interpolation operator as taking $N/2-1$ and returning $N-1$ points in the 1D case and as taking $(N/2-1)\times(N/2-1)$ points in the 2D case and returning $(N-1)\times(N-1)$. This is important because, instead of $\mathbf{I}_{2h}^{h} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$ in the example above (figure 4.2), we can just use $\mathbf{I}_{2h}^{h} = \begin{bmatrix} 0.5 \\ 1 \\ 0.5 \end{bmatrix}$. In general, we will consider the discretization vectors as not keeping the border values in them. This will be important when using the transpose of $\mathbf{I}_{2h}^{h}$ to create a restriction operator.

4.2.2 Restriction

The simplest way of transferring a discretized function from Ω^h to Ω^{2h} is to just take the values of it at the common grid points, discarding the other values. However, we will use the *full weighting* restriction which averages common points and their neighbours. We will apply the restriction onto the smoothed error, which will consist mainly of smooth components. This is important because an oscillatory function cannot be transferred in such a way to a coarsened grid (see figure 4.3). In 1D, let

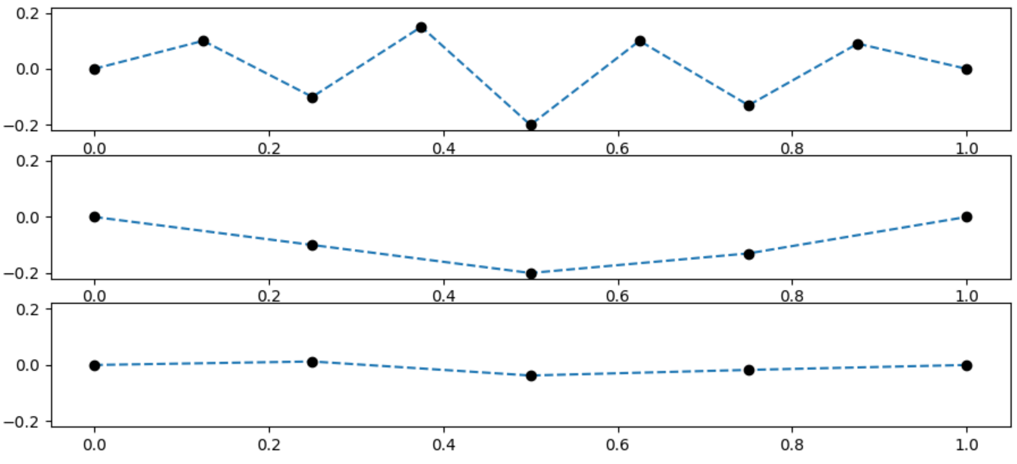


Figure 4.3: The first plot represents a discretized function f of high frequency on grid $\Omega^{1/8}$. The second plot is the trivial restriction of f on $\Omega^{1/4}$ and the third plot is the full weighting restriction of f on $\Omega^{1/4}$. Note that the restrictions do not capture the behaviour of f because of its high frequency components.

$\mathbf{v}^h = [w_0, w_1, \cdots, w_N]$ be the vector we wish to restrict. The trivial restriction gives $\mathbf{v}^{2h} = [v_0, v_1, \cdots, v_{N/2}]$, where $v_i = w_{2i}$ for $i = 0, 1, \ldots, N/2$. The full weighting restriction is $v_i = (2w_{2i} + w_{2i-1} + w_{2i+1})/4$, for interior points $i = 0, 1, \dots, N/2 - 1$ (see figure 4.4).

Similarly, in 2D the trivial restriction only keeps the points in the original grid with even indices and the full weighting restriction gives: $v_{i,j} = (4w_{2i,2j} + 2w_{2i-1,2j} + 2w_{2i+1,2j} + 2w_{2i,2j-1} + 2w_{2i,2j+1} + w_{2i-1,2j-1} + w_{2i+1,2j-1} + w_{2i-1,2j+1} + w_{2i+1,2j+1})/16$ for interior points. We refer to this in *stencil notation* as $\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$, with the obvious meaning of giving weight of 4/16 to the central points, 2/16 to its direct neighbours and 1/16 to its diagonal neighbours.

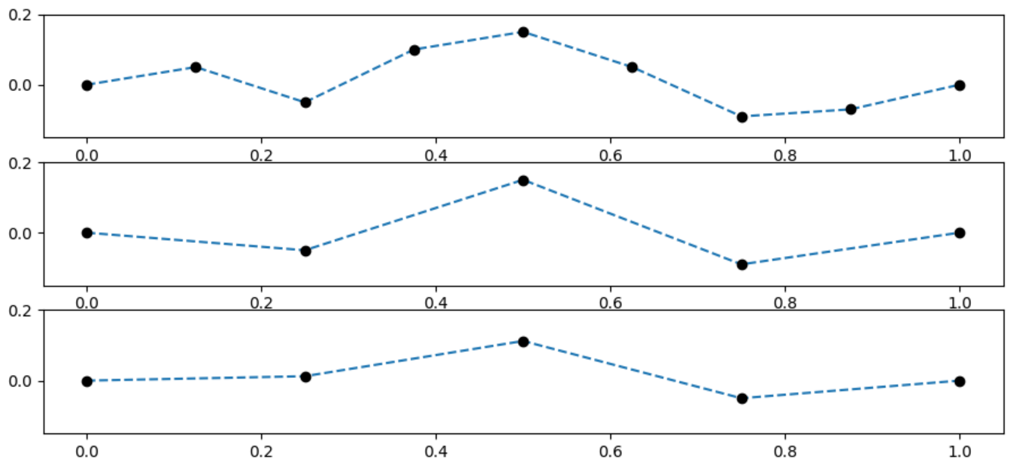


Figure 4.4: The first plot represents a discretized function f on grid $\Omega^{1/8}$. The second plot is the trivial restriction of f on $\Omega^{1/4}$ and the third plot is the full weighting restriction of f on $\Omega^{1/4}$. The restrictions are fairly good approximations of the discretization on grid $\Omega^{1/8}$ as the function is fairly smooth, there are not very big and frequent variations which get lost in the restricting process.

We define $\mathbf{I}_h^{2h}$ as taking a discretized vector of interior points (consider all border values to be 0) on the grid Ω^h, and returning a discretized vector of interior points on grid Ω^{2h} (all border points still 0) through the full weighting process described above. Notice that $\mathbf{I}_h^{2h} = c\mathbf{I}_{2h}^{h}$ in both 1D and 2D cases, where $c \in \mathbb{R}$.

4.3 The 2-grid algorithm

Now, it is easy to fill in the details of the method we started describing.

After applying μ_1 smoother steps on $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$, we got $\mathbf{x}_{\mu_1}^h$ and we were left with $\mathbf{A}^h\mathbf{e}^h = \mathbf{b}^h$, where $\mathbf{e}^h$ contains mainly smooth components which cannot easily be captured by the relaxation schemes discussed. Now, using the restriction defined above, move the problem to a coarser grid in order to get the equation $\mathbf{A}^{2h}\mathbf{e}^{2h} = \mathbf{b}^{2h}$. We use a direct solver, such as Gaussian elimination, on this smaller dimensional problem to get $\mathbf{e}^{2h}$. After this, use the interpolation defined above to approximate $\mathbf{e}^h$ by $\mathbf{I}_{2h}^{h}\mathbf{e}^{2h}$. Then, the true solution $\mathbf{x}^h$ is approximated by $\mathbf{w} = \mathbf{x}_{\mu_1}^h + \mathbf{e}^h$.

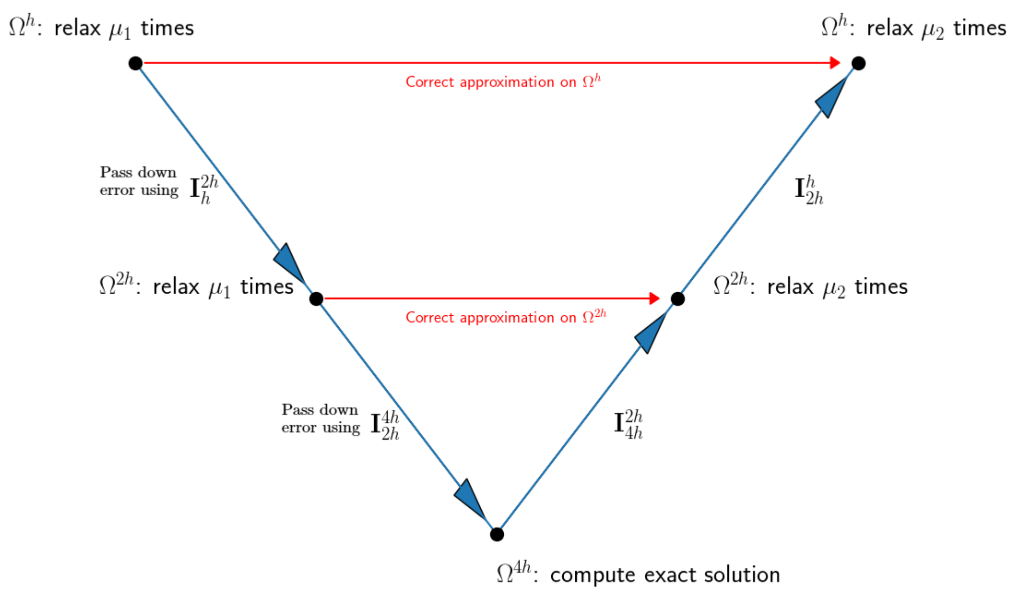
When interpolating, new error components may appear, so it is a good idea to apply a relaxation scheme μ_2 times on $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ with the initial guess for $\mathbf{x}^h$ given by $\mathbf{w}$ in order to further smooth out the error. Let the vector obtained in such a way be $\mathbf{x}_{\mu_1,\mu_2}^h$.

4.4 The V-Cycle

The 2-grid idea assumes that we can apply a direct solver on the coarsened grid, but it may be the case that the problem is still too big and we cannot solve it directly. The V-cycle is a natural generalization of the 2-grid algorithm: instead of applying a direct solver on Ω^{2h}, recursively apply the 2-grid algorithm to get an approximation of $\mathbf{e}^{2h}$. Keep recursively coarsening the grid until the resulting problem is small enough such that we can apply a direct solver:

- Relax $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ for μ_1 times to get an approximation $\mathbf{x}_{\mu_1}^h$,
- Compute residual $\mathbf{r}^h = \mathbf{b}^h - \mathbf{A}^h\mathbf{x}_{\mu_1}^h$ and its restriction $\mathbf{b}^{2h} = \mathbf{I}_h^{2h}\mathbf{r}^h$,
- Relax $\mathbf{A}^{2h}\mathbf{x}^{2h} = \mathbf{b}^{2h}$ for μ_1 times to get an approximation $\mathbf{x}_{\mu_1}^{2h}$,
- Compute residual $\mathbf{r}^{2h} = \mathbf{b}^{2h} - \mathbf{A}^{2h}\mathbf{x}_{\mu_1}^{2h}$ and its restriction $\mathbf{b}^{4h} = \mathbf{I}_{2h}^{4h}\mathbf{r}^{2h}$,
- $\cdots$
- On the coarsest grid $\Omega^{2^k h}$, use a direct solver for $\mathbf{A}^{2^k h}\mathbf{x}^{2^k h} = \mathbf{b}^{2^k h}$ and transfer $\mathbf{x}^{2^k h}$ to grid $\Omega^{2^{k-1}h}$: $\mathbf{w}^{2^{k-1}h} = \mathbf{I}_{2^k h}^{2^{k-1}h}\mathbf{x}^{2^k h}$,
- Correct $\mathbf{x}_{\mu_1}^{2^{k-1}h}$: let $\mathbf{u}^{2^{k-1}h} = \mathbf{x}_{\mu_1}^{2^{k-1}h} + \mathbf{w}^{2^{k-1}h}$. Relax $\mathbf{A}^{2^{k-1}h}\mathbf{x}^{2^{k-1}h} = \mathbf{b}^{2^{k-1}h}$ with $\mathbf{u}^{2^{k-1}h}$ as initial guess for μ_2 times to get $\mathbf{u}_{\mu_2}^{2^{k-1}h}$ and transfer it to grid $\Omega^{2^{k-2}h}$: $\mathbf{w}^{2^{k-2}h} = \mathbf{I}_{2^{k-1}h}^{2^{k-2}h}\mathbf{u}_{\mu_2}^{2^{k-1}h}$,
- $\cdots$
- Correct $\mathbf{x}_{\mu_1}^h$: let $\mathbf{u}^h = \mathbf{x}_{\mu_1}^h + \mathbf{w}^h$. Relax $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$ with $\mathbf{u}^h$ as initial guess for μ_2 times to get $\mathbf{u}_{\mu_2}^h$,
- Return $\mathbf{x}_{\mu_1,\mu_2}^h = \mathbf{u}_{\mu_2}^h$ as an approximation for $\mathbf{A}^h\mathbf{x}^h = \mathbf{b}^h$.

Intuitively, each frequency component corresponds to some grid level and, as a V-cycle explores a multitude of different grids, it will eventually explore the one on which the relaxation scheme reduces the fastest any frequency components. We can apply more V-cycle iterations to a problem in order to get a better approximation of the solution. This yields the Multigrid V-cycle scheme.



Ω^h: relax μ_1 times
Ω^h: relax μ_2 times
Correct approximation on Ω^h
Pass down error using $\mathbf{I}_h^{2h}$
$\mathbf{I}_{2h}^h$
Ω^{2h}: relax μ_1 times
Ω^{2h}: relax μ_2 times
Correct approximation on Ω^{2h}
Pass down error using $\mathbf{I}_{2h}^{4h}$
$\mathbf{I}_{4h}^{2h}$
Ω^{4h}: compute exact solution

Figure 4.5: One Multigrid V-cycle for a problem with 3 grid levels.

4.5 Multigrid setup

There are some details we need to fill in order to get an actual implementation. We first need to choose which relaxation scheme to use, its parameter and the number of relaxation steps μ_1, μ_2. We also need to decide what is the coarsest grid and which direct solver we apply on it. These choices constitute the *setup* of the Multigrid method.

We have considered an evenly spaced grid over the functions domain, but in general there are many other choices which work better on different problems. In the general case, the way of choosing the grid is also part of the setup.

Considering these, Multigrid is a method which requires extensive human intervention in setting it up. It is the opposite case of direct solvers such as Gaussian elimination and Conjugate Gradient, which require no parameter setup and are, thus, more robust.

In certain problems, Multigrid setup can be adjusted by an experimental search. Our implementation is general and customizable enough to easily change between different setups and track their performance.

4.6 Cost of Multigrid V-cycle

4.6.1 Space

For a d-dimensional problem on the grid Ω^h, where $N = \frac{1}{h}$, there are approximately N^d points in the discretization (depending on how we deal with the border points). We need to store the discretization matrix which, in CSR format, occupies $\mathbb{O}(N^d)$ space (the matrix dimension is $N^d \times N^d$, but only has $\mathbb{O}(N^d)$ nonzero elements, so $\mathbb{O}(N^d)$ storage is required - as seen in chapter 2). Let this number of variables be n. We also need to store two current approximation vectors and two residual vectors for each grid level (one of each for Ω^h on the first half of the V-cycle and one of each for Ω^h on the second half of the V-cycle), which amount in total to $\mathbb{O}(N^d)$.

Reasoning similarly for the coarser grids and taking into account that we can reuse space from a V-cycle iteration to the next one, we get that the space cost is:

$$S(N,d) = cN^d + c(N/2)^d + c(N/4)^d + \cdots \approx c\frac{2^d}{2^d-1}N^d, \qquad (4.3)$$

where c is a constant. Therefore, the storage cost is worse than the number of variables we wish to approximate only by a constant factor, so the algorithm is asymptotically optimal from this point of view : $\mathbb{O}(n)$ space. In fact, the storage space is greater only by a factor of $\frac{2^d}{2^d-1}$ than the storage used by Jacobi/SOR/SSOR on Ω^h.

4.6.2 Time

On grid Ω^h we perform μ_1 smoothing steps on the first half of a V-cycle and μ_2 smoothing steps on the second half of a V-cycle. The cost of an iteration of any relaxation scheme is $\mathbb{O}(N^d)$ (see Chapter 3: Jacobi, SOR, SSOR), let this be cN^d.

Reasoning similarly for all grid levels, the cost of one V-cycle is :

$$T_V(N,d) = c(\mu_1+\mu_2)(N^d + (N/2)^d + (N/4)^d + \cdots) \approx c\frac{2^d}{2^d-1}(\mu_1+\mu_2)N^d, \quad (4.4)$$

so the cost of one V-cycle is only the cost of a smoother iteration times a constant: $\mathbb{O}(n)$ time per V-cycle.

With the right setup, Multigrid with V-cycles can converge in a constant number of V-cycle iterations, independent of the grid size, so the computational cost is $\mathbb{O}(n)$ (recall n = number of variables in discretization $\approx N^d$), thus being asymptotically optimal for a class of problems (see [5], Chapter 3), including the discretizations that arise in our model problems (see figure 4.6).

4.7 Discussion

For the rest of this thesis, the 2D Poisson experiments will be run on the differential equation with the exact continuous solution $f(x, y) = \sin 2\pi x \sin 5\pi y$.

The relaxation on each grid level should capture the high frequency components of the error, thus leaving coarser grids to capture the smoother ones. It can be shown using spectral analysis similar to the one we used for Jacobi in Chapter 3 (see any numerical methods text analysis on SOR/SSOR, e.g. [6], [4], [22]) that higher values of ω in SOR/SSOR stand for a better elimination of the higher frequency modes of the error.

Recall that for the 2D Poisson discretization we have a formula for the optimal SOR parameter (3.9) depending on the grid size: $\omega_{\text{SOR}}^{N} = \frac{2}{1+\sin(\pi/N)}$, and we saw that the optimal SSOR parameter is approximated fairly well by ω_{SOR} (see table 3.1).

Having these optimal values in mind, we consider the following scenarios:

- using SOR/SSOR with its optimal value on the finest grid as a relaxation scheme (see figure 4.7),
- using an adaptive SOR/SSOR as a relaxation scheme, i.e. using a different relaxation parameter at each grid level, equal to the optimal for that discretization size (see figure 4.8).

We propose the second scenario as a way of obtaining a setup for large scale Multigrid systems and examine it to some extent.

These scenarios give a faster convergence in some cases, but, asymptotically, both lack the main point of Multigrid. The optimal parameter for SOR/SSOR as a solver on its own is chosen in such a way to diminish fast all the components of the error, but Multigrid seeks to eliminate the highest frequency error component per each grid level. Intuitively, the two proposals above will perform well on each level for a wide spectre of error components, but they may miss the highest frequencies components, on each grid level.

However, using SSOR with fixed, high ω does not perform well for small grid sizes, as these grids cannot even capture such high frequencies. In this case, we see that our proposal of an adaptive SSOR parameter scheme works better.

Multigrid with the right setup is far more powerful than the methods proposed before, being able to reach order optimal convergence ($\mathbb{O}(n)$). The right setup can

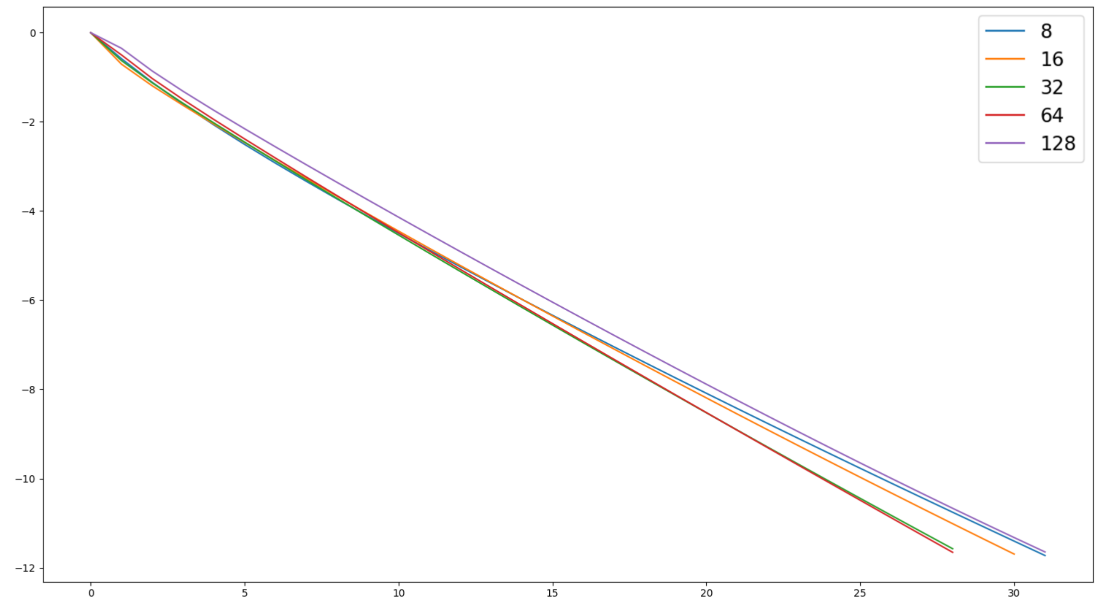


Figure 4.6: Residual error of the 2D Poisson model problem discretization in terms of number of V-cycles for varying grid sizes. The Multigrid V-cycle scheme is applied with SSOR ($\omega = 1.93$) as a relaxation scheme, $\mu_1 = \mu_2 = 2$ relaxation steps. The number of V-cycles is (approximately) independent of the grid size.

be approximated theoretically for the simplest model problems, but in general experimental results are needed for setting up the method. These experiments can be very expensive computationally.

We notice in table 4.1 that the relaxation parameter plays a more important role in the convergence costs than the numbers of pre and post smoothings, μ_1 and μ_2. We obtain the best results on the finest grid for $\mu_1 = \mu_2 = 2, \omega = 1.93$: 6.3×10^8 FLOPs. For $\omega = 1.93$ we get fairly good results even when we use no post-smoothing steps at all ($\mu_1 = 2, \mu_2 = 0$) - 6.9×10^8 FLOPs - and when we use no pre-smoothing steps at all ($\mu_1 = 0, \mu_2 = 2$) - 6.9×10^8. For $\mu_1 = \mu_2 = 1$ and $\omega = 1.93$, we get approximately 6.86×10^8 FLOPs. However, for $\mu_1 = \mu_2 = 2$ and $\omega = 1.85$, the performance decreases drastically - approximately 7.8×10^8 FLOPs. The smoothing factor is so important because it corresponds to the spectrum of frequencies that get eliminated most rapidly and the idea of Multigrid is to choose it in such a way that it eliminates the high frequency components. A good choice eliminates even from the first $1-2$ iterations a considerable amount of high frequency components of the error.

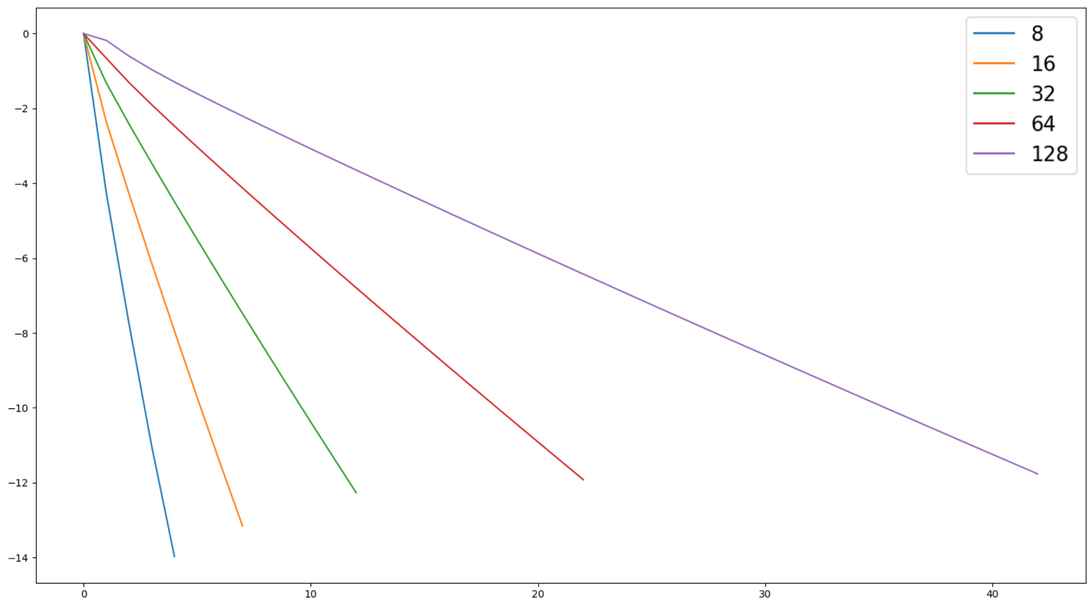


Figure 4.7: Choosing ω to be the optimal relaxation factor of SSOR on the finest grid on the same problem and setup as in figure 4.6.

N	$\mu_1 = 2, \mu_2 = 2$ $\omega = 1.9$	$\mu_1 = 2, \mu_2 = 2$ $\omega = 1.93$	$\mu_1 = 2, \mu_2 = 2$ $\omega = 1.85$	$\mu_1 = 2, \mu_2 = 2,$ $\omega = w_{\text{OPT}},$ fixed	$\mu_1 = 2, \mu_2 = 0$ $\omega = 1.91$
8	189980	264320	123900	41300	223104
16	927168	1306464	632160	337152	1114840
32	3779580	5219420	2699700	2339740	4457376
64	16203792	21359544	12521112	16940328	18240992
128	83266288	95161472	86240084	127873228	92059880
256	668949120	633112560	776458800	979532640	712693968

N	$\mu_1 = 2, \mu_2 = 2$ $\omega = w_{\text{OPT}},$ adaptive	$\mu_1 = 2, \mu_2 = 2$ $\omega = 1.89$	$\mu_1 = 2, \mu_2 = 0$ $\omega = 1.93$	$\mu_1 = 1, \mu_2 = 1,$ $\omega = 1.93$	$\mu_1 = 0, \mu_2 = 2$ $\omega = 1.93$
8	41300	173460	288176	288176	288176
16	295008	842880	1423200	1423200	1423200
32	2159760	3419620	5774328	5673024	5673024
64	15467256	14730720	23215808	23215808	23215808
128	110030452	83266288	102102776	102102776	102102776
256	692840160	692840160	692523384	685799856	692523384

Table 4.1: Flops count for Multigrid with SSOR (ω relaxation parameter) as a smoother, with μ_1 and μ_2 pre and post smoothing steps.

We presented a proposal of adapting the relaxation parameter per each grid level. "Guessing" a parameter close to eliminating the highest frequencies remaining on each grid level (which is not the optimal parameter for the relaxation scheme as a

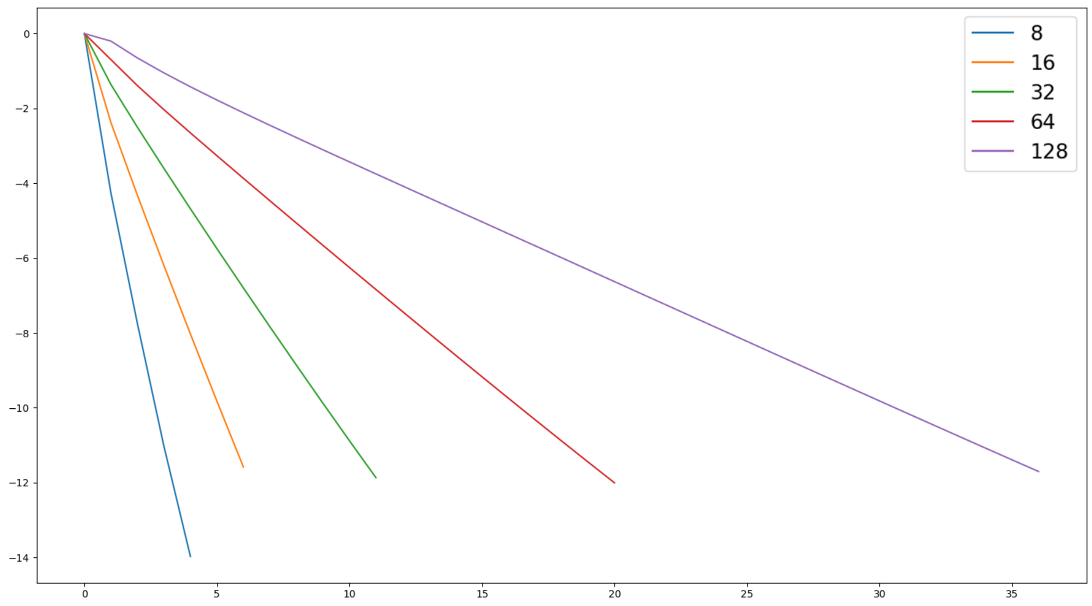


Figure 4.8: The behaviour of the adaptive method we propose. Changing ω on each grid of a V-cycle to its optimal value for SSOR on that grid size on the same problem and setup as in figure 4.6.

solver on its own!) would most probably give better results than any fixed parameter choice. However, this adaptive value is not accessible in general. Our proposal, which takes for each level the optimal value for the relaxation scheme as a solver on its own (see figure 4.8) already proves to work better on the smaller grid sizes, but slightly worsens for big values. However, there are plenty of methods for approximating the optimal parameter for a smoother ([6]) and very few and limited for approximating optimal smoothing parameters for large scale Multigrid systems involving varying meshes. Therefore, using the adaptive method after precomputing approximations for the parameters per each grid level seems a sensible approach.

Multigrid with V-cycle is just one method from the general class of Multigrid methods, all of which share the property of exploring various grids, smoothing the error and passing information between levels. There are different cycles, approaches for obtaining and adapting the grid and other modifications to the method we proposed (see [4] for some other Multigrid methods).

4.8 Multigrid as a preconditioner for Conjugate Gradient Method

In some cases, Multigrid is order optimal, as we have seen, thus the best algorithm we could asymptotically hope for. [10] notes that an optimally setup Multigrid works so well in practice in these cases that it is the basis for an HPC (High Performance Computing) benchmark used to measure the maximum capacity of a supercomputer to solve real life physics problems (HPGMG - https://hpgmg.org/). Unfortunately, there are problems for which the optimal Multigrid method is unknown or is inherently impossible/very hard to determine (main errors appear at transfer between grids and at the level of the exact solver precision). However, if the error of a Multigrid method is contained in a low-dimensional subspace, a Krylov method such as Conjugate Gradient can solve for this error in a small number of steps, thus cleaning up after Multigrid ([10]), as the number of iterations of Conjugate Gradient is at most the number of dimensions.

A Multigrid preconditioned Conjugate Gradient (MGCG) was first proposed by Kettler and Meijerink in [13] and is studied in detail and generalized by Tatebe in [20]. It consists of preconditioning a linear system with a few iterations (in general 1 or 2) in order to speed up the Conjugate Gradient method. The following is a result proved in [20]:

Theorem 4.8.1. *The Multigrid V-cycle with projection and restriction related by $\mathbf{I}_h^{2h} = c\mathbf{I}_{2h}^h$, where c is a positive constant, with damped Jacobi or SSOR as relaxation schemes and with equal pre and post smoothing steps ($\mu_1 = \mu_2$) works as a preconditioner for Conjugate Gradient, as it yields a symmetric positive definite preconditioning matrix.*

In practice, this method works really well and has the main advantage of being more robust. The setup of the Multigrid preconditioner does not impact the convergence as heavily as the setup of the Multigrid solver on its own, which is sensible to the mesh refinement and parameter selection. In fact, for certain problems with the right setup it is order optimal as well, converging in a number of steps independent of the grid size (figure 4.9).

We examine different Multigrid (with SSOR as relaxation scheme) preconditioned Conjugate Gradient schemes to see which converges the fastest in terms of floating point operations count in the table 4.2. It appears that the best choice is having 1 pre-smoothing and 1 post-smoothing steps of SSOR. We also note that the fastest convergence in terms of FLOPs is reached for $\mu = 1$ and $\omega \in \{1.9, 1.91, 1.93\}$.

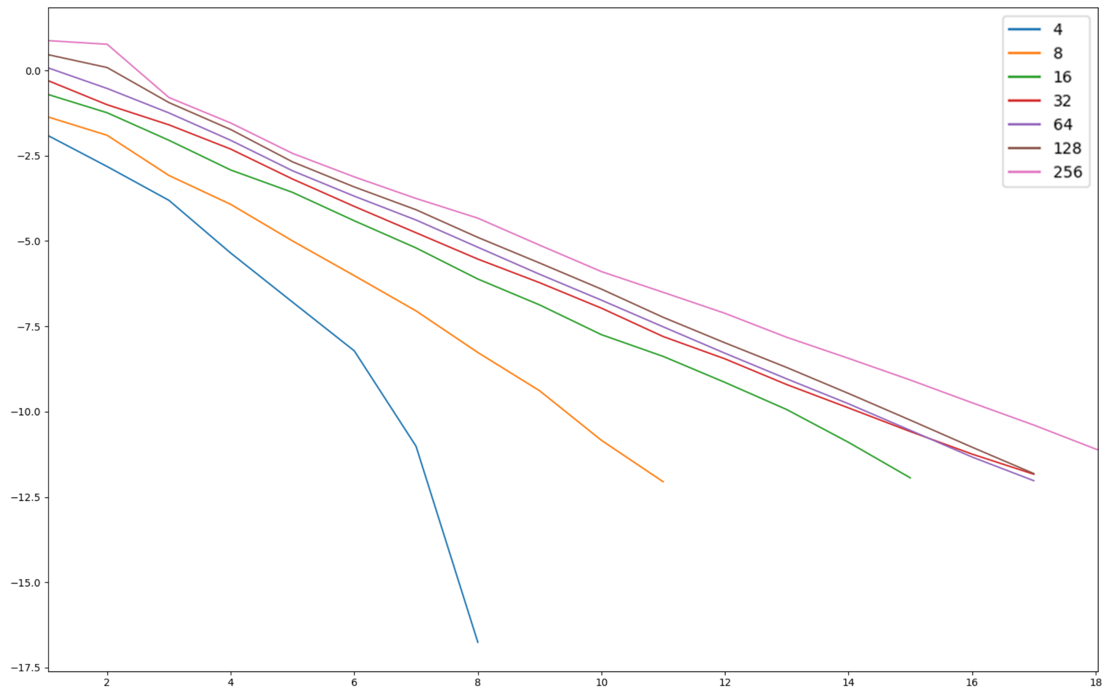


Figure 4.9: Logarithmic scale of the residual error of the 2D Poisson model problem discretization in terms of number of MG preconditoned CG iterations. Per each iteration, we apply 2 MG iterations with SSOR as a smoother ($\omega = 1.97, \mu_1 = \mu_2 = 2$). The number of steps is almost independent of the grid size, being smaller only for the coarse grids, and converging towards 18 *steps for the fine grids.*

4.9 Comparison between solvers

Let us mention that not all the setups we experimented on were presented in tables 4.1 and 4.2. We tried various values of ω and μ which make sense in light of the theory we discussed and increased the rate of experiments (which are shown) in the relevant domains. For example, small values of ω do not make sense in MG, as we seek to eliminate high frequencies. Another example is that using too many pre-smoothing or post-smoothing steps for MGCG loses the essence of performing few FLOPs.

We have seen that both MG and MGCG can be order optimal under the right setup. However, that optimal setup is hard to determine. In the experiments we performed, MGCG performs significantly better on the model problem. For the $256^2 \times 256^2$ grid, MGCG reaches convergence with the setups described in table 4.2 in the range of $1.8 \times 10^8 - 2.7 \times 10^8$ FLOPs. The best convergence we got for MG with V-cycle (table 4.1) on the same problem and grid takes 6.4×10^8 FLOPs ($\mu_1 = \mu_2 = 2, \omega = 1.93$).

The main general advantage of MGCG is that its setup does not impact as hard the

N	$\mu = 1$, $\omega = 1.9$	$\mu = 1$, $\omega = 1.93$	$\mu = 1$, $\omega = 1.95$	$\mu = 1$, $\omega = 1.97$	$\mu = 2$, $\omega = 1.95$	$\mu = 2$, $\omega = 1.97$
8	75670	82608	82608	89546	125214	136656
16	409764	468632	468632	498066	680188	777688
32	1682180	2044514	2286070	2648404	2796132	3597516
64	7304350	7793560	9261190	12196450	11335180	14583460
128	37285206	39254592	41223978	49101522	48920742	58731516
256	189151336	189151336	197054658	220764624	248869522	261995728
N	$\mu = 1$, $\omega = 1.91$	$\mu = 1$, $\omega = 1.8$	$\mu = 1$, $\omega = 1.85$	$\mu = 2$, $\omega = 1.8$	$\mu = 2$, $\omega = 1.85$	$\mu = 1$, ω varying
8	75670	61794	68732	79446	90888	40980
16	409764	292028	321462	338938	387688	233160
32	1802958	1319846	1319846	1594056	1594056	1319846
64	7304350	6815140	6815140	8086900	8898970	6815140
128	37285206	37285206	37285206	45650484	45650484	37285206
256	189151336	220764624	204957980	261995728	248869522	212861302

Table 4.2: Flops count for MGCG with 1 V-cycle per CG step. MG uses SSOR with $\mu_1 = \mu_2 = \mu$ smoothing steps and with relaxation parameter ω. The ω varying header represents the experiments performed with the MG with the adaptive strategy proposed before, using an approximation of the optimal parameter per each grid level.

convergence as it happens in the MG case. This is because CG is a "fixed" method, which requires no setup and converges fast on well-conditioned matrices (and in at most number of dimensions of the error steps), while MG depends heavily on the smoother. MG relies on the fact that the smoother iterations eliminate the highest frequency errors on each grid level, otherwise there are components which decrease very slowly through the iterations.

For example, note that we got the best results for MG (on the finest grid) using $\mu_1 = \mu_2 = 2, \omega = 1.93$ - approximately 6.3×10^8 FLOPs. However, for $\mu_1 = \mu_2 = 2, \omega = 1.9$ we need approximately 6.7×10^8 FLOPs. Meanwhile, for MGCG we get approximately 1.9×10^8 FLOPs for $\mu = 1$ and for any $\omega \in \{1.9, 1.91, 1.93\}$. Thus, we see how MGCG is not as sensitve as MG to the setup.

In MGCG we can see CG as "cleaning" the components of the error not eliminated by MG, which lie in a lower-dimensional space. Thus, the method does not rely anymore on MG eliminating specifically the highest frequency components, as CG decreases the remaining error components.

The advantage regarding the easiness of the setup becomes useful in practice, as we do not have to worry so much about finding the "perfect" setup for MGCG, unlike in the case of MG. We can try using various heuristics for MGCG, such as our proposal of an approximation for the adaptive SSOR relaxation scheme: for each grid level, choose ω close to the optimal value of SSOR on that grid size.

The adaptive strategy performs surprisingly well for both MG and MGCG on the finest grid of the model problem: 6.9×10^8 FLOPs versus the optimal of 6.7×10^8 FLOPs for MG; and 2.1×10^8 FLOPs versus the optimal of 1.9×10^8 FLOPs for MGCG. There are many strategies of approximating the optimal parameter for relaxation methods, so the approach we used may be generalized for other classes of problems.

Chapter 5

Conclusions

Multigrid (MG) and Multigrid preconditioned Conjugate Gradient (MGCG) methods are far superior to the classic linear approximation schemes. With the right setup, both can be order optimal for wide classes of problems, including discretizations arising from most PDEs.

MG methods behave impressively, but their performance depends heavily on the setup. We have seen that SSOR behaves well as a relaxation scheme and that its smoothing parameter is the main factor affecting the convergence of the MG method.

As approximating optimal smoothing parameters for MG is hard, we proposed an adaptive method which uses for each discretization level an approximation of the optimal value of the smoother as a solver on its own. The exact value for each grid level is known for few problems, but we can use approximation algorithms, which are simpler than approximating the optimal smoothing parameter for MG.

MGCG is a robust method, being less sensitive to the setup. This is because MG eliminates most of the high frequency components in a small number of iterations, and Conjugate Gradient is left to deal with a lower dimensional error, which it can eliminate fast. This is the reason why approximations of the Multigrid setup work well in practice. Our proposal of an adaptive relaxation parameter works surprisingly well with MGCG.

In the experiments we performed, both MG an MGCG reached order optimality, but it seems that MGCG performed significantly better in terms of floating point operations. It seems difficult to find a MG setup much better to use than MGCG.

It is most likely that an extension to this project will analyze how both MG and MGCG perform when we use an adaptive grid spacing as well (which becomes more frequent at sensible points of the modelled functions) and how our proposal of an adaptive smoothing parameter can be combined with the adaptive spacing.

Appendix A

Implementation

This appendix is provided as an extra material. The project report does not rely on it, but it includes the practical component of the project.

The implementations were run in Python 2.7.12. The solver methods can be found in the 2 files: SolverMethods.py and MGMethods.py. SolverMethods.py provides an encapsulation of different classic iterative solvers. MGMethods.py contains more classes encapsulating Multigrid based methods, including a Multigrid standalone solver class and a Multigrid class for preconditioning the Conjugate Gradient method. It also contains the preconditioned Conjugate Gradient using the Multigrid preconditioner encapsulation.

Several discretizers are available (encapsulated in classes in the files TimeEquationDiscretizer.py, SimpleEquationDiscretizer.py and EquationDiscretizer1D.py) which can be passed to the different solver methods available.

The experiments were run using the testing utilities in TestTools.py and the auxiliary functions in posDefPlot.py. FunctionExamples.py provides different instances of our our model problems on which we performed the experiments.

The code is provided in the current appendix, in the following pages.

```python
# SolverMethods.py

#  ______________________________________
# |CLASSIC CUSTOMIZABLE ITERATIVE SCHEMES|
# |______________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *
import math

tol = 0.00001

# Class encapsulating iterative methods for solving a linear system
class SolverMethods:
    def __init__(self, iterationConstant, eqDiscretizer, b = [], initSol = [], actualSol = [], initB = []):
        self.iterationConstant = iterationConstant
        self.M = eqDiscretizer.M
        self.D = eqDiscretizer.D
        self.R = eqDiscretizer.R
        self.L = eqDiscretizer.L
        self.U = eqDiscretizer.U
        if(b == []) :
            self.b = eqDiscretizer.valueVector2D
        else:
            self.b = b
        self.initB = self.b
        self.initSol = initSol
        self.actualSol = actualSol

    # Jacobi iterative method
    def JacobiIterate(self, dampFactor = 1.0):
        errorDataJacobi = []
        x = []
        d = self.D.diagonal()
        iterationConstant = self.iterationConstant
        # Initial guess is x = (0,0,...0) if not provided as a parameter
        if(self.initSol == []):
            x = np.zeros_like(self.b)
        else:
            x = self.initSol

        # Iterate constant number of times
        for i in range(iterationConstant):
            err = np.subtract(self.M.dot(x), self.b)
            absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))

            if(absErr < tol):
                break
            errorDataJacobi.append(math.log(absErr))

            y = self.R.dot(x)
            r = np.subtract(self.b, y)
            xPrev = np.copy(x)
            x = [r_i / d_i for r_i, d_i in zip(r, d)]
            xNew = np.add(np.multiply(xPrev, (1.0 - dampFactor)), np.multiply(x, dampFactor))
            x = np.copy(xNew)



        err = np.subtract(self.b, self.M.dot(x))
        absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
        errorDataJacobi.append(math.log(absErr))
        return x, absErr, errorDataJacobi, err

    # Jacobi iterative method v2, these 2 implementations have been used
    # to decide the more efficient way of implementing it
    def JacobiIterate2(self, omega = 1.0):
        errorDataJacobi = []
        x = []
        currentLowerRows = []
        currentUpperRows = []
        d = self.D.diagonal()
        iterationConstant = self.iterationConstant
        # Initial guess is x = (0,0,...0) if not provided as a parameter
        if(self.initSol == []):
            x = np.zeros_like(self.b)
        else:
            x = self.initSol

        for j in range(self.L.shape[0]):
            currentLowerRows.append(self.L.getrow(j))
            currentUpperRows.append(self.U.getrow(j))

        # Iterate constant number of times (TODO: iterate while big error Mx-b)
        for i in range(iterationConstant):
            err = np.subtract(self.M.dot(x), self.b)
            absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
            errorDataJacobi.append(math.log(absErr))

            if(absErr < tol):
                break

            xNew = np.zeros_like(x)
            for j in range(self.L.shape[0]):
                currentLowerRow = currentLowerRows[j]
                currentUpperRow = currentUpperRows[j]

                rowSum = currentLowerRow.dot(x) + currentUpperRow.dot(x) - x[j] * d[j]
                rowSum = 1.0 * (self.b[j] - rowSum) / d[j]
                xNew[j] = x[j] + omega * (rowSum - x[j])

            if np.allclose(x, xNew, rtol=1e-6):
                 break

            x = np.copy(xNew)



        err = np.subtract(self.b, self.M.dot(x))
        absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
        errorDataJacobi.append(math.log(absErr))
        return x, absErr, errorDataJacobi, err

    # Gauss Seidel iterative method
    def GaussSeidelIterate(self, omega = 1.0):
        errorDataGaussSeidel = []
        x = []
        d = self.L.diagonal()
        iterationConstant = self.iterationConstant

        flops = 0

        currentLowerRows = []
        currentUpperRows =[]

        if(self.initSol == []):
            x = np.zeros_like(self.b)
        else:
            x = self.initSol

        for j in range(self.L.shape[0]):
            currentLowerRows.append(self.L.getrow(j))
            currentUpperRows.append(self.U.getrow(j))

        for i in range(iterationConstant):
            err = np.subtract(self.M.dot(x), self.b)
            absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
            errorDataGaussSeidel.append(math.log(absErr))

            xNew = np.zeros_like(x)
            for j in range(self.L.shape[0]):
                currentLowerRow = currentLowerRows[j]
                currentUpperRow = currentUpperRows[j]

                rowSum = currentLowerRow.dot(xNew) + currentUpperRow.dot(x)
                flops += max(2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) - 1, 0)

                rowSum = 1.0 * (self.b[j] - rowSum) / d[j]
                flops += 2

                xNew[j] = x[j] + omega * (rowSum - x[j])
                flops += 3

            if(absErr < tol):
                break

            x = np.copy(xNew)

        print("Flops: ",flops)
        print("Iterations: ", len(errorDataGaussSeidel) - 1)
        return x, absErr, errorDataGaussSeidel, err

    # SSOR Iteartive method
    def SSORIterate(self, omega = 1.0, debugOn = False):
        errorDataSSOR = []
        x = []
        d = self.D.diagonal()
        iterationConstant = self.iterationConstant

        flops = 0

        currentLowerRows = []
        currentUpperRows = []

        if(self.initSol == []):
            x = np.zeros_like(self.b)
        else:
            x = np.copy(self.initSol)

        for j in range(self.L.shape[0]):
            currentLowerRows.append(self.L.getrow(j))
            currentUpperRows.append(self.U.getrow(j))


        err = np.subtract(self.M.dot(x), self.b)
        absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
        errorDataSSOR.append(math.log(absErr))


        for k in range(iterationConstant):

            xNew = np.zeros_like(x)

            for i in range(self.L.shape[0]):
                currentLowerRow = currentLowerRows[i]
                currentUpperRow = currentUpperRows[i]

                currSum = currentLowerRow.dot(xNew) + currentUpperRow.dot(x)
                flops += max(2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) - 1, 0)

                currSum = 1.0 * (self.b[i] - currSum) / d[i]
                flops += 2

                xNew[i] = x[i] + omega * (currSum - x[i])
                flops += 3

            x = np.copy(xNew)

            if(debugOn and k % 10 == 0):
                print("Iteration: ", k)
                print("After top to bottom: ", x)

            xNew = np.zeros_like(x)
            for i in reversed(range(self.L.shape[0])):
                currSum = 0.0
                currentLowerRow = currentLowerRows[i]
                currentUpperRow = currentUpperRows[i]

                currSum = currentLowerRow.dot(x) + currentUpperRow.dot(xNew) - d[i] * x[i]
                flops += 2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) + 1

                currSum = 1.0 * (self.b[i] - currSum) / d[i]
                flops += 2

                xNew[i] = x[i] + omega * (currSum - x[i])
                flops += 3

            x = np.copy(xNew)

            if(debugOn and k%10 ==0):
                print("After bottom to top: ", x)
                print("______________")

            err = np.subtract(self.M.dot(x), self.b)
            absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
            errorDataSSOR.append(math.log(absErr))

            if(absErr < tol):
                break

        err = np.subtract(self.b, self.M.dot(x))
        absErr = np.linalg.norm(err) / np.linalg.norm(self.b)

        return x, absErr, errorDataSSOR, err, flops

    # Conjugate Gradient using the Hestenes Stiefel formulation
    def ConjugateGradientsHS(self):
        flops = 0
        matrixDots = 0
        vectorAddSub = 0
        vectorDotVector = 0

        M = self.M
        b = self.b
        actualSol = self.actualSol

        avoidDivByZeroError = 0.0000000001
        errorDataConjugateGradients = []
        x = np.zeros_like(b, dtype=np.float)
        r = np.subtract(b, M.dot(x))
        d = np.copy(r)
        matrixDots += 1
        vectorAddSub += 1

        convergence = False
        beta_numerator = r.dot(r)
        vectorDotVector += 1

        while(not convergence):
            solutionError = np.subtract(M.dot(x), b)
            relativeResidualErr = np.linalg.norm(solutionError) / np.linalg.norm(self.initB)

            if(relativeResidualErr < tol):
                convergence = True
                break

            if(actualSol != []):
                err = np.subtract(actualSol, x)
                absErr = np.linalg.norm(err)
                errorDataConjugateGradients.append(math.log(absErr))
            else:
                errorDataConjugateGradients.append(math.log(relativeResidualErr))


            Md = M.dot(d)
            alpha_numerator = beta_numerator
            alpha_denominator = d.dot(Md)
            vectorDotVector += 1
            matrixDots += 1

            if(alpha_denominator < avoidDivByZeroError):
                convergence = True
                break

            alpha = 1.0 * alpha_numerator / alpha_denominator
            flops += 1

            x = np.add(x, np.multiply(d, alpha))
            flops += len(d)
            vectorAddSub += 1

            r_new = np.subtract(r, np.multiply(Md, alpha))
            flops += len(Md)
            vectorAddSub += 1

            beta_numerator = r_new.dot(r_new)
            beta_denominator = alpha_numerator
            vectorDotVector += 1

            if(beta_denominator < avoidDivByZeroError):
                convergence = True
                break

            beta = 1.0 * beta_numerator / beta_denominator
            flops += 1

            d = r_new + np.multiply(d, beta)
            vectorAddSub += 1
            flops += len(d)

            r = r_new

        nonZero = M.nonzero()
        NNZ = len(nonZero[0])
        flops += vectorAddSub * len(x) + vectorDotVector * (2 * len(x) - 1) + matrixDots * (2 * NNZ - len(x))
        return x, relativeResidualErr, errorDataConjugateGradients

    # Steepest descent method
    def SteepestDescent(self):
        avoidDivByZeroError = 0.0000000000000000001

        flops = 0
        matrixDots = 0
        vectorAddSub = 0
        vectorDotVector = 0
        actualSol = self.actualSol

        M = self.M
        b = self.b

        x = np.zeros_like(b)
        r = np.subtract(b, M.dot(x))
        matrixDots += 1
        vectorAddSub += 1

        errorDataSteepestDescent = []
        iterationConstant = self.iterationConstant
        for i in range(iterationConstant):
            err = np.subtract(M.dot(x), b)
            divide = np.linalg.norm(b)
            if(actualSol != []):
                err = np.subtract(x, actualSol)
                divide = 1.0
            absErr = np.linalg.norm(err) / divide
            errorDataSteepestDescent.append(math.log(absErr))

            alpha_numerator = r.dot(r)
            alpha_denominator = r.dot(M.dot(r))
            vectorDotVector += 2
            matrixDots +=1

            if(alpha_denominator < avoidDivByZeroError):
                break

            alpha = alpha_numerator / alpha_denominator
            flops += 1

            x = np.add(x, np.dot(r, alpha))
            flops += len(x)
            vectorAddSub+=1

            r = np.subtract(b, M.dot(x))
            vectorAddSub += 1
            matrixDots += 1

            if(np.linalg.norm(r) / np.linalg.norm(b) < tol):
                break

        NNZ = M.getnnz()
        flops += vectorAddSub * len(x) + vectorDotVector * (2 * len(x) - 1) + matrixDots * (2 * NNZ - len(x))

        err = np.subtract(M.dot(x), b)
        divide = np.linalg.norm(b)
        if(actualSol != []):
            err = np.subtract(x, actualSol)
            divide = 1.0
        absErr = np.linalg.norm(err) / divide
        errorDataSteepestDescent.append(math.log(absErr))

        return x, absErr, errorDataSteepestDescent
```

```python
# MGMethods.py

#  ________________________________________________
# |Multigrid based advanced methods, including MGCG|
# |________________________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *
import numpy.linalg as la
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import SimpleEquationDiscretizer as sed
import EquationDiscretizer1D as sed1D
import SolverMethods as sm
import FunctionExamples as fe
import TimeEquationDiscretizer as ted

tol = 0.00001

# Sanity check function for determining if a matrix is positive definite
def is_pos_def(x):
    return np.all(np.linalg.eigvals(x) > 0)

flops = 0

# Class encapsulating the Multigrid method for a given 2D Poisson equation instance
class MultiGrid2D:

    def __init__(self, maxN, borderFunction, valueFunction, niu1 = 4, niu2 = 4, omega = 1.95):
        # BorderFunction and valueFunction pass the function values in the Poisson equation
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction
        self.omega = omega

        # Niu1 and niu2 are the number of relaxation steeps per going-down and going-up parts of the v-cycle
        self.niu1 = niu1
        self.niu2 = niu2

        # N is the discretization grid size
        self.N = maxN

        # Create and store grid for different levels of discretization
        self.discrLevel = []
        self.flops = 0
        i = 0
        while(maxN > 2):
            assert(maxN % 2 == 0)
            self.discrLevel.append(sed.SimpleEquationDiscretizer(maxN, self.borderFunction, self.valueFunction))
            i += 1
            maxN /= 2

    # Helper function for grid level N
    def getCoordinates(self, row, N):
        return int(row / (N + 1)), row % (N + 1)

    # Helper function for grid level N
    def getRow(self, i, j, N):
        return(i * (N + 1) + j)

    # Restrict function from grid level fineN to grid level coarseN using trivial injection
    def restrict(self, r, fineN, coarseN):
        restr = []

        for(i, elem) in enumerate(r):
            (x, y) = self.getCoordinates(i, fineN)

            if(x % 2 == 0 and y % 2 == 0):
                restr.append(elem)

        return restr


    # Interpolate function from grid level coarseN to grid level fineN with full weight stencil
    def interpolate(self, r, fineN, coarseN):
        interp = []
        flops = 0
        for i in range((fineN + 1) * (fineN + 1)):
            (x, y) = self.getCoordinates(i, fineN)

            if(x % 2 == 0 and y % 2 == 0):
                index = self.getRow(x / 2, y / 2, coarseN)
                value = r[index]
                flops += 1

            elif(x % 2 == 1 and y % 2 == 0):
                index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
                index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
                value = (r[index1] + r[index2]) / 2.0
                flops += 2

            elif(x % 2 == 0 and y % 2 == 1):
                index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
                index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
                value = (r[index1] + r[index2]) / 2.0
                flops += 2

            else:
                index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
                index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
                index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
                index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
                value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4.0
                flops += 4

            if(x == 0 or y == 0 or x == fineN or y == fineN):
                value = 0

            interp.append(value)

        self.flops += flops
        return interp

    # Restrict function from grid level fineN to grid level coarseN using
    # the transpose action of interpolate (full weighting operator)
    def restrictTransposeAction(self, r, fineN, coarseN):
        restr = []
        flops = 0

        for i in range((coarseN + 1) * (coarseN + 1)):
            (x, y) = self.getCoordinates(i, coarseN)
            (x, y) = (2 * x, 2 * y)
            newEntry = r[self.getRow(x, y, fineN)]

            divideFactor = 1.0

            for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                newX = x + dX
                newY = y + dY
                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fineN):
                    index = self.getRow(newX, newY, fineN)
                    newEntry += 0.5 * r[index]
                    divideFactor += 0.5
                    flops += 2

            for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
                newX = x + dX
                newY = y + dY
                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fineN):
                    index = self.getRow(newX, newY, fineN)
                    newEntry += 0.25 * r[index]
                    divideFactor += 0.25
                    flops += 2

            newEntry = 1.0 * newEntry / divideFactor
            if(divideFactor < 4.0):
                if(not(x == 0 or y == 0 or x == fineN or y == fineN)):
                    print("Error code #1")

            if(x == 0 or y == 0 or x == fineN or y == fineN):
                newEntry = 0.0

            restr.append(newEntry)

        self.flops += flops
        return restr

    # Vcycle iterates once a V-relaxation scheme of multigrid starting at level L for a grid size N
    def vcycle(self, N, L, f, initSol = [], omega = 1.95):
        level = L
        discr = self.discrLevel[level]
        fSize = len(f)

        if(level == len(self.discrLevel) - 1):
            v = la.solve(discr.M.todense(), f)
            return v

        else:
            solver1 = sm.SolverMethods(
                iterationConstant = self.niu1,
                eqDiscretizer = discr,
                b = f,
                initSol = initSol,
                )

            # Omega is the smoother parameter
            omega = self.omega
            v, _, _, _, flops= solver1.SSORIterate(omega)
            self.flops += flops

            coarseN = N  / 2

            Mv = discr.M.dot(v)
            self.flops += (2 * discr.M.getnnz() - len(v))

            residual = np.subtract(f, Mv)
            self.flops += len(f)

            coarseResidual = self.restrictTransposeAction(residual, N, coarseN)

            coarseV = self.vcycle(coarseN, level + 1, coarseResidual)

            fineV = self.interpolate(coarseV, N, coarseN)
            w = np.add(v, fineV)
            self.flops += len(v)

            solver2 = sm.SolverMethods(
                iterationConstant = self.niu2,
                eqDiscretizer = discr,
                b = f,
                initSol = w,
                )

            v2, _, _, _, flops= solver2.SSORIterate(omega)
            self.flops += flops
            return v2

    # IterateVCycles iterates the function vcycle() for t times to obtain a better approximation
    def iterateVCycles(self, t):
        initSol = []
        N = self.N

        for i in range((N + 1) * (N + 1)):
            (x, y) = self.getCoordinates(i, N)
            if(x == 0 or y == 0 or x == N or y == N):
                initSol.append(self.borderFunction(1.0 * x / N, 1.0 * y / N))
            else:
                initSol.append(0.0)

        vErrors = []
        discr = self.discrLevel[0]
        f = np.copy(discr.valueVector2D)
        normF = la.norm(f)

        currSol = np.copy(initSol)

        for i in range(t):
            if(i % 10 == 0):
                print(i)

            residual = np.subtract(f, discr.M.dot(currSol))
            absErr = 1.0 * la.norm(residual) / (normF)
            vErrors.append(math.log(absErr))

            resSol = self.vcycle(N, 0, residual, np.zeros_like(currSol))
            prevSol = np.copy(currSol)
            currSol = np.add(currSol, resSol)

            if(absErr < tol):
                break

        return currSol, vErrors, self.flops


# Class encapsulating the Multigrid method for a given 1D Poisson equation instance
class MultiGrid:

    def __init__(self, maxN, borderFunction, valueFunction, niu1 = 4, niu2 = 4):
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction
        self.niu1 = niu1
        self.niu2 = niu2
        self.discrLevel = []
        self.N = maxN

        self.flops = 0

        i = 0
        while(maxN > 2):
            assert(maxN % 2 == 0)
            self.discrLevel.append(sed1D.EquationDiscretizer1D(maxN, self.borderFunction, self.valueFunction))
            i += 1
            maxN /= 2

    # Helper function for grid level N
    def getCoordinates(self, row, N):
        return int(row / (N + 1)), row % (N + 1)

    # Helper function for grid level N
    def getRow(self, i, j, N):
        return(i * (N + 1) + j)

    # Restriction operator
    def restrict1D(self, r, fineN, coarseN):
        restr = []
        for(i, elem) in enumerate(r):
            if(i%2 == 0):
                restr.append(elem)
        return restr

    # Interpolation operator
    def interpolate1D(self, r, fineN, coarseN):
        interp = []
        for i in range(fineN + 1):
            if(i%2 == 0):
                interp.append(r[i/2])
            else:
                interp.append((r[(i-1)/2]+r[(i+1)/2])/2.0)
        return interp

    # Vcycle scheme
    def vcycle(self, N, level, f, initSol = [], omega = 1.95):
        discr = self.discrLevel[level]
        fSize = len(f)

        if(level == len(self.discrLevel) - 1):
            # If the discretization is small enough, use a direct solver
            v = la.solve(discr.M.todense(), f)
            return v

        else:
            # If the discretization is still too fine, recursively V-cycle over it

            solver1 = sm.SolverMethods(
                iterationConstant = self.niu1,
                eqDiscretizer = discr,
                b = f,
                initSol = initSol,
                )

            # Omega is the smoothing parameter
            omega = 1.0

            # Using SSOR as a smoother
            v, _, _, flops = solver1.SSORIterate(omega)
            self.flops += flops

            # Sanity checks verifying that the error on the border stays 0
            if(not(v[0] == 0)):
                print("Error code #2")
            if(not(v[N] == 0)):
                print("Error code #3")

            assert(N % 2 == 0)
            coarseN = N  / 2

            Mv = discr.M.dot(v)
            flops += (2 * discr.M.getnnz() - len(v))

            residual = np.subtract(f, Mv)
            flops += len(f)

            # Project on a coarser grid the current error
            coarseResidual = self.restrict1D(residual, N, coarseN)

            # Sanity checks verifying that the error on the border stays 0
            if(not(coarseResidual[0] == 0)):
                print("Error code #4")
            if(not(coarseResidual[coarseN] == 0)):
                print("Error code #5")

            # Recursively V-cycle
            coarseV = self.vcycle(coarseN, level + 1, coarseResidual)

            # Sanity checks verifying that the error on the border stays 0
            if(not(coarseV[0] == 0)):
                print("Error code #6")
            if(not(coarseV[coarseN] == 0)):
                print("Error code #7")

            # Interpolate the vector obtained recursively to a finer level
            fineV = self.interpolate1D(coarseV, N, coarseN)

            # Correct the current approximation using the interpolation passed from a coarser level
            w = np.add(v, fineV)
            flops += len(v)

            solver2 = sm.SolverMethods(
                iterationConstant = self.niu2,
                eqDiscretizer = discr,
                b = f,
                initSol = w,
                )

            # Apply extra smoothing to flatten out error components that might have been
            # introduced in the process so far
            v2, _, _, _, flops = solver2.SSORIterate(omega)
            self.flops += flops
            return v2

    # Iterate t Vcycles in order to get a better approximate of the solution
    def iterateVCycles(self, t):
        initSol = []
        N = self.N

        for i in range((N + 1)):
            x = i
            if(x == 0 or x == N):
                initSol.append(self.borderFunction(1.0 * x / N))
            else:
                initSol.append(0.0)

        vErrors = []
        discr = self.discrLevel[0]
        f = np.copy(discr.valueVector2D)
        normF = la.norm(f)

        currSol = np.copy(initSol)

        for i in range(t):
            residual = np.subtract(f, discr.M.dot(currSol))
            absErr = 1.0 * la.norm(residual) / (normF)
            vErrors.append(math.log(absErr))

            resSol = self.vcycle(N, 0, residual, np.zeros_like(currSol))
            prevSol = np.copy(currSol)
            currSol = np.add(currSol, resSol)

            if(absErr < tol):
                break
        print(vErrors)
        return currSol, vErrors


# Multigrid class implemented such that it can easily be adapted
# as a preconditioner, previous implementations regard it as
# being a solver on its own and put an emphasis on it
class MultiGridAsPreconditioner:

    def __init__(self, borderFunction, valueFunction, maxN, bVector = [], niu1 = 2 , niu2 = 2, omega = 1.93):
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction
        self.niu1 = niu1
        self.niu2 = niu2
        self.maxN = maxN
        self.bVector = bVector
        self.discrLevel = []
        self.omega = omega

        self.flops = 0

        i = 0
        while(maxN >= 2):
            assert(maxN % 2 == 0)
            self.discrLevel.append(sed.SimpleEquationDiscretizer(maxN, self.borderFunction, self.valueFunction))
            i += 1
            maxN /= 2


    def getCoordinates(self, row, N):
        return int(row / (N + 1)), row % (N + 1)

    def getRow(self, i, j, N):
        return(i * (N + 1) + j)


    def restrict(self, r, fineN, coarseN):
        restr = []

        for(i, elem) in enumerate(r):
            (x, y) = self.getCoordinates(i, fineN)

            if(x % 2 == 0 and y % 2 == 0):
                restr.append(elem)

        return restr

    def interpolate(self, r, fineN, coarseN):
        interp = []
        flops = 0
        for i in range((fineN + 1) * (fineN + 1)):
            (x, y) = self.getCoordinates(i, fineN)

            if(x % 2 == 0 and y % 2 == 0):
                index = self.getRow(x / 2, y / 2, coarseN)
                value = r[index]
                flops += 1

            elif(x % 2 == 1 and y % 2 == 0):
                index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
                index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
                value = (r[index1] + r[index2]) / 2.0
                flops += 2

            elif(x % 2 == 0 and y % 2 == 1):
                index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
                index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
                value = (r[index1] + r[index2]) / 2.0
                flops += 2

            else:
                index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
                index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
                index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
                index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
                value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4.0
                flops += 4

            if(x == 0 or y == 0 or x == fineN or y == fineN):
                value = 0

            interp.append(value)

        self.flops += flops

        return interp

    def restrictTransposeAction(self, r, fineN, coarseN):
        restr = []
        flops = 0

        for i in range((coarseN + 1) * (coarseN + 1)):
            (x, y) = self.getCoordinates(i, coarseN)
            (x, y) = (2 * x, 2 * y)
            newEntry = r[self.getRow(x, y, fineN)]

            divideFactor = 1.0

            for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                newX = x + dX
                newY = y + dY
                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fineN):
                    index = self.getRow(newX, newY, fineN)
                    newEntry += 0.5 * r[index]
                    divideFactor += 0.5
                    flops += 2

            for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
                newX = x + dX
                newY = y + dY
                if(0 <= newX and newX <= fineN and 0 <= newY and newY <= fineN):
                    index = self.getRow(newX, newY, fineN)
                    newEntry += 0.25 * r[index]
                    divideFactor += 0.25
                    flops += 2

            newEntry = 1.0 * newEntry / divideFactor
            if(divideFactor < 4.0):
                if(not(x == 0 or y == 0 or x == fineN or y == fineN)):
                    print("Error1")

            if(x == 0 or y == 0 or x == fineN or y == fineN):
                newEntry = 0.0

            restr.append(newEntry)

        self.flops += flops
        return restr


    def vcycle(self, N, level, f, initSol = []):

        discr = self.discrLevel[level]
        fSize = len(f)
        if(fSize < 20):
            v = la.solve(discr.M.todense(), f)
            return v

        omega = self.omega

        solver1 = sm.SolverMethods(self.niu1, discr, f, initSol)
        v, _, _, _, flops = solver1.SSORIterate(omega)
        self.flops += flops

        assert(N % 2 == 0)
        coarseN = N  / 2

        Mv = discr.M.dot(v)
        self.flops += (2*discr.M.getnnz() - len(v))

        residual = np.subtract(np.array(f), Mv)
        self.flops += len(f)

        coarseResidual = self.restrictTransposeAction(residual, N, coarseN)

        coarseV = self.vcycle(coarseN, level + 1, coarseResidual)
        fineV = self.interpolate(coarseV, N, coarseN)
        v = np.add(v, fineV)
        self.flops += len(v)

        solver2 = sm.SolverMethods(self.niu2, discr, f, v)
        v2, _, _, _, flops = solver2.SSORIterate(omega)
        self.flops += flops
        return v2

    def iterateVCycles(self, N, t):
        self.flops = 0
        initSol = []
        vErrors = []
        discr = sed.SimpleEquationDiscretizer(N, self.borderFunction, self.valueFunction)

        if(self.bVector == []):
            f = discr.valueVector2D
        else:
            f = self.bVector

        for i in range(t):
            currSol = self.vcycle(N, 0, f, initSol)

            err = np.subtract(discr.M.dot(currSol), f)
            absErr = np.linalg.norm(err) / np.linalg.norm(f)
            vErrors.append(math.log(absErr))

            if(absErr < tol):
                break

            initSol = currSol

        return currSol, vErrors, self.flops

# MultiGridPrecondCG is the function that performs the Multigrid preconditioned  Conjugate Gradient method
def MultiGridPrecondCG(borderFunction, valueFunction, N, niu1 =1, niu2 = 1, omega = 1.95):
    avoidDivByZeroError = 0.000000000001
    errorDataMGCG = []

    totalFlops = 0
    matrixDots = 0
    vectorAddSub = 0
    vectorDotVector = 0

    mg = MultiGridAsPreconditioner(borderFunction, valueFunction, N, niu1 =niu1, niu2 = niu2, omega = omega)
    f = mg.discrLevel[0].valueVector2D
    M = mg.discrLevel[0].M

    x = np.zeros_like(f, dtype = np.float)
    r = np.subtract(f, M.dot(x))
    matrixDots += 1
    vectorAddSub += 1

    mg.bVector = np.copy(r)
    rTilda, _, flops = mg.iterateVCycles(N, 1)
    totalFlops += flops

    rTilda = np.array(rTilda)

    p = np.copy(rTilda)

    convergence = False

    while(not convergence):
        solutionError = np.subtract(M.dot(x), f)
        absErr = 1.0 * np.linalg.norm(solutionError) / np.linalg.norm(f)
        errorDataMGCG.append(math.log(absErr))

        if(absErr < tol):
            convergence = True
            break

        Mp = M.dot(p)
        alpha_numerator = rTilda.dot(r)
        alpha_denominator = p.dot(Mp)

        vectorDotVector += 1
        matrixDots += 1

        if(alpha_denominator < avoidDivByZeroError):
            convergence = True
            break

        alpha = 1.0 * alpha_numerator / alpha_denominator
        totalFlops += 1

        x = np.add(x, np.multiply(p, alpha))
        vectorAddSub += 1
        totalFlops += len(p)

        newR = np.subtract(r, np.multiply(Mp, alpha))
        totalFlops += len(Mp)

        vectorAddSub += 1

        mg.bVector = np.copy(newR)
        newR_tilda, _, flops = mg.iterateVCycles(N, 1)
        totalFlops += flops

        newR_tilda = np.array(newR_tilda)

        beta_numerator = newR_tilda.dot(newR)
        beta_denominator = rTilda.dot(r)
        vectorDotVector += 1

        if(beta_denominator < avoidDivByZeroError):
            convergence = True
            break

        beta = 1.0 * beta_numerator / beta_denominator
        p = newR_tilda + np.multiply(p, beta)
        totalFlops += 1
        totalFlops += len(p)

```

```python
        r = newR
        rTilda = newR_tilda

    NNZ = M.getnnz()
    totalFlops += vectorAddSub * len(x) + vectorDotVector * (2*len(x) -1) + matrixDots * (2* NNZ - len(x))
    return x, absErr, errorDataMGCG, totalFlops

# JacobiPrecondCG is the function that performs the Jacobi preconditioned  Conjugate Gradient method
# This has not been included in the project report, as the results of this method are not impressive
def JacobiPrecondCG(borderFunction, valueFunction, N):
    avoidDivByZeroError = 0.0000000000000000000001
    errorDataMGCG = []

    mg = sed.SimpleEquationDiscretizer(N, borderFunction, valueFunction)
    solver = sm.SolverMethods(5, mg)
    f = mg.valueVector2D
    M = mg.M

    x = np.zeros_like(f, dtype = np.float)
    r = np.subtract(f, M.dot(x))

    solver.b = r
    rTilda, _ , _, _= solver.JacobiIterate(0.2)
    rTilda = np.array(rTilda)

    p = np.copy(rTilda)

    convergence = False

    while(not convergence):
        solutionError = np.subtract(M.dot(x), f)
        absErr = 1.0 * np.linalg.norm(solutionError) / np.linalg.norm(f)
        errorDataMGCG.append(math.log(absErr))
        print(absErr)

        if(absErr < tol):
            convergence = True
            break

        alpha_numerator = rTilda.dot(r)
        alpha_denominator = p.dot(M.dot(p))

        if(alpha_denominator < avoidDivByZeroError):
            convergence = True
            break

        alpha = 1.0 * alpha_numerator / alpha_denominator

        x = np.add(x, np.multiply(p, alpha))

        newR = np.subtract(r, np.multiply(M.dot(p), alpha))

        solver.b = newR
        newR_tilda, _, _, _ = solver.JacobiIterate(0.2)
        newR_tilda = np.array(newR_tilda)

        beta_numerator = newR_tilda.dot(newR)
        beta_denominator = rTilda.dot(r)

        if(beta_denominator < avoidDivByZeroError):
            convergence = True
            break

        beta = 1.0 * beta_numerator / beta_denominator
        p = newR_tilda + np.multiply(p, beta)

        r = newR
        rTilda = newR_tilda

    return x, absErr, errorDataMGCG


# Using Conjugate Gradient to solve the Heat Equation
# using the Backward Euler method to generate the RHS at
# each time step
def solveHeatEquationForAllTimeSteps(discr):
    solHeat = discr.initialHeatTimeSolution()
    valueVector = discr.computeVectorAtTimestep(1, solHeat)
    solver = sm.SolverMethods(1000, discr, valueVector)
    sol =[solHeat]

    for k in range(1, discr.T + 1):
        t = k * discr.dT
        valueVector = discr.computeVectorAtTimestep(k, solHeat)
        solver.b = valueVector
        (solHeat, err, _) = solver.ConjugateGradientsHS()
        sol.append(solHeat)
        print(err)

    return sol
```

```python
# TimeEquationDiscretizer.py

# ______________________________________________
# |USED TO GENERATE HEAT EQUATION DISCRETIZATION|
# |_____________________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *


COMPLETE_MATRIX = 'COMPLETE_MATRIX'
LOWER_MATRIX = 'LOWER_MATRIX'
STRICTLY_UPPER_MATRIX = 'STRICTLY_UPPER_MATRIX'
DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
REMAINDER_MATRIX = 'REMAINDER_MATRIX'

# Class encapsulating the discretization arising from
# a 2D Heat Equation, including methods to get the
# RHS vector for the Backward Euler method
class TimeEquationDiscretizer:

    # Current value vector i.e. current value (at current time k * dT) of f(x,y,t) in du/dt - laplace(u) = f
    def __init__(self, N, T, borderTimeFunction, rhsHeatEquationFunction, initialHeatTimeFunction):
        self.N = N
        self.h = 1.0 / N

        self.T = T
        self.dT  = 1.0 / T

        self.borderTimeFunction = borderTimeFunction
        self.rhsHeatEquationFunction = rhsHeatEquationFunction
        self.initialHeatTimeFunction = initialHeatTimeFunction

        self.rowList = []
        self.colList = []
        self.dataList = []

        self.rowListDiagonal = []
        self.colListDiagonal = []
        self.dataListDiagonal = []

        self.rowListRemainder = []
        self.colListRemainder = []
        self.dataListRemainder = []

        self.rowListUpper = []
        self.colListUpper = []
        self.dataListUpper = []

        self.rowListLower = []
        self.colListLower = []
        self.dataListLower = []

        self.computeMatrixHeatTimeEquation()
        self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1) * (N + 1), (N + 1) * (N +
        self.D = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.rowListDiagonal), np.array(self.colListDiagonal))), shape = ((N + 1) *
        self.R = csr_matrix((np.array(self.dataListRemainder), (np.array(self.rowListRemainder), np.array(self.colListRemainder))), shape = ((N + 1)

        # Lower and strictly upper matrices L, U with L + U = M
        self.L = csr_matrix((np.array(self.dataListLower), (np.array(self.rowListLower), np.array(self.colListLower))), shape = ((N + 1) * (N + 1),
        self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.rowListUpper), np.array(self.colListUpper))), shape = ((N + 1) * (N + 1),

        print(self.M.todense())
    # Helper functions for creating the complete, lower, upper and diagonal matrices
    def addEntry(self, type, row, column, value):
        if(type == COMPLETE_MATRIX):
            self.rowList.append(row)
            self.colList.append(column)
            self.dataList.append(value)
        elif(type == LOWER_MATRIX):
            self.rowListLower.append(row)
            self.colListLower.append(column)
            self.dataListLower.append(value)
        elif(type == STRICTLY_UPPER_MATRIX):
            self.rowListUpper.append(row)
            self.colListUpper.append(column)
            self.dataListUpper.append(value)
        elif(type == DIAGONAL_MATRIX):
            self.rowListDiagonal.append(row)
            self.colListDiagonal.append(column)
            self.dataListDiagonal.append(value)
        elif(type == REMAINDER_MATRIX):
            self.rowListRemainder.append(row)
            self.colListRemainder.append(column)
            self.dataListRemainder.append(value)


    def addEntryToMatrices(self, row, column, value):
        self.addEntry(COMPLETE_MATRIX, row, column, value)
        if(row == column):
            self.addEntry(DIAGONAL_MATRIX, row, column, value)
            self.addEntry(LOWER_MATRIX, row, column, value)
        if(row > column):
            self.addEntry(LOWER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)
        if(row < column):
            self.addEntry(STRICTLY_UPPER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)

    # Check if a(i,j) is on border
    def isOnBorder(self, i, j):
        # print(i, j)
        if(i == 0 or j == 0 or i == (self.N) or j == (self.N)):
            return True
        else:
            return False


    # Get the coordinates of the variable around which the row-th row is created
    def getCoordinates(self, row):
        return int(row / (self.N + 1)), row % (self.N + 1)


    # Get the row of a(i, j)'s equation
    def getRow(self, i, j):
        return(i * (self.N + 1) + j)

    valueVector = []

    # Returns a vector with the initial solution at t = 0 for all x,y
    def initialHeatTimeSolution(self):
        initSol = []
        for currentIndex in range((self.N + 1) * (self.N + 1)):
            (x, y) = self.getCoordinates(currentIndex)
            initSol.append(self.initialHeatTimeFunction(x, y, 0))
        return initSol


    # Compute M and valueVector2D (in Mx = valueVector2D) and
    # compute L, U, D (lower, strictly upper and diagonal matrices of M)
    def computeMatrixHeatTimeEquation(self):
        for currentRow in range((self.N + 1) * (self.N + 1)):
            self.computeRowHeatTimeEquation(currentRow)


    # Computes the RHS vector at time k*dT w.r.t. the prevSol (sol vector at time = (k-1)*dT)
    def computeVectorAtTimestep(self, k, prevSol):
        valueVector = []
        for currentIndex in range((self.N + 1) * (self.N + 1)):
            (x, y) = self.getCoordinates(currentIndex)

            if(self.isOnBorder(x, y)):
                value = self.borderTimeFunction((1.0) * x / (self.N + 1), (1.0) * y / (self.N + 1), (1.0) * k * self.dT)
            else:
                value = 1.0* (self.h * self.h) * self.rhsHeatEquationFunction((1.0) * x / (self.N + 1), (1.0) * y / (self.N + 1), (1.0) * k * self.d
                value = value + 1.0 * (self.h * self.h) / self.dT * prevSol[self.getRow(x, y)]

                for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    if((self.isOnBorder(x + dX, y + dY))):
                        localValue = self.borderTimeFunction((1.0) * (x + dX) / (self.N + 1), (1.0) * (y + dY) / (self.N + 1), k * self.dT)
                        value += localValue

            valueVector.append(value)

        return valueVector


    # Compute the elements of row-th row in (rowList, colList, dataList) for the heat equation with time
    def computeRowHeatTimeEquation(self, row):
        (x, y) = self.getCoordinates(row)
        if(self.isOnBorder(x, y)):
            self.addEntryToMatrices(row, row, 1)
        else:
            self.addEntryToMatrices(row, row, 4 + (1.0) * (self.h * self.h) / self.dT)

            for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                if(not(self.isOnBorder(x + dX, y + dY))):
                    self.addEntryToMatrices(row, self.getRow(x + dX, y + dY), -1)
```

```python
# SimpleEquationDiscretizer.py

#  ___________________________________________________
# |USED TO GENERATE POISSON 2D EQUATION DISCRETIZATION|
# |___________________________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *


COMPLETE_MATRIX = 'COMPLETE_MATRIX'
LOWER_MATRIX = 'LOWER_MATRIX'
STRICTLY_UPPER_MATRIX = 'STRICTLY_UPPER_MATRIX'
DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
REMAINDER_MATRIX = 'REMAINDER_MATRIX'

# Class encapsulating the sparse matrix
# arising from a 2D poisson model problem
class SimpleEquationDiscretizer:

    def __init__(self, N, borderFunction, valueFunction):
        self.N = N
        self.h = 1.0 / N
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction

        self.rowList = []
        self.colList = []
        self.dataList = []

        self.rowListDiagonal = []
        self.colListDiagonal = []
        self.dataListDiagonal = []

        self.rowListRemainder = []
        self.colListRemainder = []
        self.dataListRemainder = []

        self.rowListUpper = []
        self.colListUpper = []
        self.dataListUpper = []

        self.rowListLower = []
        self.colListLower = []
        self.dataListLower = []

        self.valueVector2D = []

        self.computeMatrixAndVector()
        self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1) * (N + 1), (N + 1) * (N +
        self.D = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.rowListDiagonal), np.array(self.colListDiagonal))), shape = ((N + 1) *
        self.R = csr_matrix((np.array(self.dataListRemainder), (np.array(self.rowListRemainder), np.array(self.colListRemainder))), shape = ((N + 1)

        # Lower and strictly upper matrices L, U with L + U = M
        self.L = csr_matrix((np.array(self.dataListLower), (np.array(self.rowListLower), np.array(self.colListLower))), shape = ((N + 1) * (N + 1),
        self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.rowListUpper), np.array(self.colListUpper))), shape = ((N + 1) * (N + 1),

    # Helper functions for creating the complete, lower, upper and diagonal matrices
    def addEntry(self, type, row, column, value):
        if(type == COMPLETE_MATRIX):
            self.rowList.append(row)
            self.colList.append(column)
            self.dataList.append(value)
        elif(type == LOWER_MATRIX):
            self.rowListLower.append(row)
            self.colListLower.append(column)
            self.dataListLower.append(value)
        elif(type == STRICTLY_UPPER_MATRIX):
            self.rowListUpper.append(row)
            self.colListUpper.append(column)
            self.dataListUpper.append(value)
        elif(type == DIAGONAL_MATRIX):
            self.rowListDiagonal.append(row)
            self.colListDiagonal.append(column)
            self.dataListDiagonal.append(value)
        elif(type == REMAINDER_MATRIX):
            self.rowListRemainder.append(row)
            self.colListRemainder.append(column)
            self.dataListRemainder.append(value)


    def addEntryToMatrices(self, row, column, value):
        self.addEntry(COMPLETE_MATRIX, row, column, value)
        if(row == column):
            self.addEntry(DIAGONAL_MATRIX, row, column, value)
            self.addEntry(LOWER_MATRIX, row, column, value)
        if(row > column):
            self.addEntry(LOWER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)
        if(row < column):
            self.addEntry(STRICTLY_UPPER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)

    valueVector2D = []


    # Check if a(i,j) is on border
    def isOnBorder(self, i, j):
        # print(i, j)
        if(i == 0 or j == 0 or i == self.N or j == self.N):
            return True
        else:
            return False


    # Get the coordinates of the variable around which the row-th row is created
    def getCoordinates(self, row):
        return int(row / (self.N + 1)), row % (self.N + 1)


    # Get the row of a(i, j)'s equation
    def getRow(self, i, j):
        return(i * (self.N + 1) + j)


    # Compute M and valueVector2D (in Mx = valueVector2D) and
    # computer L, U, D (lower, strictly upper and diagonal matrices of M)
    def computeMatrixAndVector(self):
        for currentRow in range((self.N + 1) * (self.N + 1)):
            self.computeRow(currentRow)


    # Compute the elements of row-th row in (rowList, colList, dataList) for -nabla f(x,t) = g(x,t) problem
    def computeRow(self, row):
        (x, y) = self.getCoordinates(row)
        if(self.isOnBorder(x, y)):
            self.addEntryToMatrices(row, row, 1.0)
            # The value of the border on point x/N, y/N is known,
            # so append the equation variable = value to the system
            self.valueVector2D.append(self.borderFunction((1.0) * x / self.N, (1.0) * y / self.N))
        else:
            value = - self.valueFunction((1.0) * x / self.N, (1.0) * y / self.N) * self.h * self.h
            self.addEntryToMatrices(row, row, 4.0)

            for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                if(not(self.isOnBorder(x + dX, y + dY))):
                    self.addEntryToMatrices(row, self.getRow(x + dX, y + dY), -1.0)
                else:
                    localValue = self.borderFunction((1.0) * (x + dX) / self.N, (1.0) * (y + dY) / self.N)
                    value += localValue
            self.valueVector2D.append(value)
```

```python
# EquationDiscretizer1D.py

#  ___________________________________________________
# |USED TO GENERATE POISSON 1D EQUATION DISCRETIZATION|
# |___________________________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *


COMPLETE_MATRIX = 'COMPLETE_MATRIX'
LOWER_MATRIX = 'LOWER_MATRIX'
STRICTLY_UPPER_MATRIX = 'STRICTLY_UPPER_MATRIX'
DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
REMAINDER_MATRIX = 'REMAINDER_MATRIX'

# Class encapsulating the sparse matrix
# arising from discretizing a 1D model problem
class EquationDiscretizer1D:

    def __init__(self, N, borderFunction, valueFunction, printMatrix = False):
        self.N = N
        self.h = 1.0 / N
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction

        self.rowList = []
        self.colList = []
        self.dataList = []

        self.rowListDiagonal = []
        self.colListDiagonal = []
        self.dataListDiagonal = []

        self.rowListRemainder = []
        self.colListRemainder = []
        self.dataListRemainder = []

        self.rowListUpper = []
        self.colListUpper = []
        self.dataListUpper = []

        self.rowListLower = []
        self.colListLower = []
        self.dataListLower = []

        self.valueVector1D = []

        self.computeMatrixAndVector()
        self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1), (N + 1)))
        self.D = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.rowListDiagonal), np.array(self.colListDiagonal))), shape = ((N + 1), (
        self.R = csr_matrix((np.array(self.dataListRemainder), (np.array(self.rowListRemainder), np.array(self.colListRemainder))), shape = ((N + 1)

        # Lower and strictly upper matrices L, U with L + U = M
        self.L = csr_matrix((np.array(self.dataListLower), (np.array(self.rowListLower), np.array(self.colListLower))), shape = ((N + 1), (N + 1)))
        self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.rowListUpper), np.array(self.colListUpper))), shape = ((N + 1), (N + 1)))

        if(printMatrix):
            print("Discretization matrix: ")
            print(self.M.todense())
            print("RHS vector: ")
            print(self.valueVector1D)

    # Helper functions for creating the complete, lower, upper and diagonal matrices
    def addEntry(self, type, row, column, value):
        if(type == COMPLETE_MATRIX):
            self.rowList.append(row)
            self.colList.append(column)
            self.dataList.append(value)
        elif(type == LOWER_MATRIX):
            self.rowListLower.append(row)
            self.colListLower.append(column)
            self.dataListLower.append(value)
        elif(type == STRICTLY_UPPER_MATRIX):
            self.rowListUpper.append(row)
            self.colListUpper.append(column)
            self.dataListUpper.append(value)
        elif(type == DIAGONAL_MATRIX):
            self.rowListDiagonal.append(row)
            self.colListDiagonal.append(column)
            self.dataListDiagonal.append(value)
        elif(type == REMAINDER_MATRIX):
            self.rowListRemainder.append(row)
            self.colListRemainder.append(column)
            self.dataListRemainder.append(value)


    def addEntryToMatrices(self, row, column, value):
        self.addEntry(COMPLETE_MATRIX, row, column, value)
        if(row == column):
            self.addEntry(DIAGONAL_MATRIX, row, column, value)
            self.addEntry(LOWER_MATRIX, row, column, value)
        if(row > column):
            self.addEntry(LOWER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)
        if(row < column):
            self.addEntry(STRICTLY_UPPER_MATRIX, row, column, value)
            self.addEntry(REMAINDER_MATRIX, row, column, value)

    valueVector1D = []


    # Check if a(i) is on border
    def isOnBorder(self, i):
        if(i == 0 or i == self.N):
            return True
        else:
            return False



    # Compute M and valueVector1D (in Mx = valueVector1D) and
    # computer L, U, D (lower, strictly upper and diagonal matrices of M)
    def computeMatrixAndVector(self):
        for currentRow in range((self.N + 1)):
            self.computeRow(currentRow)


    # Compute the elements of row-th row in (rowList, colList, dataList) for -nabla f(x,t) = g(x,t) problem
    def computeRow(self, row):
        x = row
        if(self.isOnBorder(x)):
            self.addEntryToMatrices(row, row, 1.0)
            # The value of the border on point x/N, y/N is known,
            # so append the equation variable = value to the system
            self.valueVector1D.append(self.borderFunction((1.0) * x / self.N))
        else:
            value = - self.valueFunction((1.0) * x / self.N) * self.h * self.h
            self.addEntryToMatrices(row, row, 2.0)

            for (dX) in [-1, 1]:
                if(not(self.isOnBorder(x + dX))):
                    self.addEntryToMatrices(row, x + dX, -1.0)
                else:
                    localValue = self.borderFunction((1.0) * (x + dX) / self.N)
                    value += localValue
            self.valueVector1D.append(value)
```

```python
# testTools.py

#  _____________________________________________________________________________
# |USED FOR CARRYING TESTS ON ALL THE STRUCTURES AND METHODS DEFINED IN THE PROJECT|
# |_____________________________________________________________________________|

import numpy as np
from scipy.sparse import *
from scipy import *
import numpy.linalg as la
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import SimpleEquationDiscretizer as sed
import EquationDiscretizer1D as sed1D
import SolverMethods as sm
import FunctionExamples as fe
import TimeEquationDiscretizer as ted
import MGMethods as MG

# Default line colors used in some testing utilities
lineColor = ["red", "green", "blue", "brown", "black", "pink", "gray"]

# Empirically discovered approximation of the optimal SSOR parameter for the class of problems
# used in our tests
wSSOR2D = {}
wSSOR2D[8] = 1.503
wSSOR2D[16] = 1.720
wSSOR2D[32] = 1.852
wSSOR2D[64] = 1.923
wSSOR2D[128] = 1.961


# Heat Equation solver testing utility
def testHeatEquationSolver():
    discrTest = ted.TimeEquationDiscretizer(32, 32, fe.heatSinBorderFunction, fe.heatRhsFunction, fe.heatInitialFunction)
    sol = MG.solveHeatEquationForAllTimeSteps(discrTest)
    xSol = sol[16]
    return xSol

# Multigrid Preconditioned Conjugate Gradient solver testing utility
def testMGCG():
    # Niu1 and niu2 are the number of pre and post smoothing steps
    niu1 = 1
    niu2 = 1

    # The paramter of the smoothing iteration used in MG as a preconditioner
    omega = 1.92

    print("Testing MGCG, niu1 = niu2 = ", niu1, ", omega = ", omega)

    for N in [4, 8, 16, 32, 64, 128, 256]:
        print(N)
        xSolPrecond, errPrecond, errDataPrecond, flops = MG.MultiGridPrecondCG(
                                                            fe.sin2BorderFunction,
                                                            fe.sin2ValueFunction,
                                                            N,
                                                            niu1= niu1,
                                                            niu2= niu2,
                                                            omega = omega,
                                                         )
        plt.plot(errDataPrecond, label=str(N))

        print("FLOPS:", flops)

    plt.legend(loc='upper right')
    plt.title("Testing MGCG, niu1 = niu2 = " + str(niu1) + ", omega = " + str(omega))
    plt.show()

# PlotGraph is a helper function which interpolates a 3D graph of a function, given values at the grid knots
def plotGraph(N, valuesVector):
    h = 1.0 / N
    fig = plt.figure()
    ax = fig.add_subplot(111, projection = '3d')
    x = y = np.arange(0.0, 1.0 + h, h)

    X, Y = np.meshgrid(x, y)
    Z = np.reshape(valuesVector, (N+1, N+1))

    ax.plot_wireframe(X, Y, Z)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')

    plt.show()

# Multigrid solver testing utility for 2D problem instances
def testMG2D():
    N = 256
    n = 4

    i = 0

    niu1 = 1
    niu2 = 1

    omega = 1.5
    while(n < N):
        n = n * 2
        # Select omega = 2.0 / (1.0 + math.sin(math.pi / n)) to test the optimal SSOR parameter as a smoothing parameter
        print("Testing MG2D, niu1, niu2 = ", niu1," ", niu2,", omega = ", omega)
        print(n, ':')

        mg = MG.MultiGrid2D(n, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = niu1, niu2 = niu2)
        solMG, vErrors, flops = mg.iterateVCycles(1000)

        print("Flops: ",flops)

        plt.plot(vErrors, label = str(n))

    plt.title("Testing MG2D, niu1, niu2 = " + str(niu1) + " "+str(niu2) + ", omega = " + str(omega))
    plt.legend(loc='upper right', prop={'size':'16'})
    plt.show()

# Multigrid Preconditioned Conjugate Gradient solver testing utility for 2D problem instances for different smoothing parameters
def testMG2DvarW():
    N = 128
    listOmega = [1.89, 1.9, 1.91, 1.92, 1.93, 1.94, 1.95, 1.96]

    for omega in listOmega:
        mg = MG.MultiGrid2D(N, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = 2, niu2 = 2)
        solMG, vErrors, flops = mg.iterateVCycles(1000)

        print(N, ":", flops)
        plt.plot(vErrors, label = str(omega))

    plt.legend(loc='upper right', prop={'size':'16'})
    plt.show()

# Multigrid solver testing utility for 1D problem instances
def testMG1D():
    N = 128
    n = 2
    i = 0
    while(n < N):
        n = n * 2
        mg = MG.MultiGrid(n, fe.sin1DBorderFunction2, fe.sin1DValueFunction2)
        solMG, vErrors = mg.iterateVCycles(1000)
        plt.plot(vErrors, label = str(n))

    plt.legend(loc='upper right')
    plt.show()

# Multigrid Conjugate Gradient solver testing utility for 1D problem instances for different smoothing parameters
def testDifferentParamIterations1D():
    N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))

    optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
    for omega in [optSOROmega]:
        print(omega)
        sinEquationDiscr = sed1D.EquationDiscretizer1D(N, fe.sin1DBorderFunction, fe.sin1DValueFunction)

        initSol = []

        for i in range((N+1)):
            if(i == 0 or i == N):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * i/ N))
            else:
                initSol.append(0)

        solver = sm.SolverMethods(700, sinEquationDiscr, initSol = initSol)
        (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega, debugOn = False)
        plt.plot(errorDataJacobi, label=str(omega))
        print(len(errorDataJacobi))

    plt.legend(loc = "upper right", prop={'size':'15'})
    plt.show()

# SSOR solver testing utility for 2D problem instances for different smoothing parameters
def testDifferentParamIterations():
    N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))


    optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
    for omega in [optSOROmega, 1.503]:
        print(omega)
        sinEquationDiscr = sed.SimpleEquationDiscretizer(N, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []

        for i in range((N + 1) * (N + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == N or y == N):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / N, 1.0 * y / N))
            else:
                initSol.append(0.0)

        solver = sm.SolverMethods(300, sinEquationDiscr, initSol = initSol)
        (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega)
        plt.plot(errorDataJacobi, label=str(omega))

    plt.legend(loc = "upper right", prop={'size':'15'})
    plt.show()

# Conjugate Gradient testing utility
def testConjugateGradient():
    N = 128
    n = 4
    index = 0
    print("Testing Conjugate Gradient")

    while(n < N):
        n = 2 * n
        print(n)
        sinEquationDiscr = sed.SimpleEquationDiscretizer(n, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []
        actualSol = fe.actualSinSolution(n)

        for i in range((n + 1) * (n + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == n or y == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
            else:
                initSol.append(0.0)

        solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol, actualSol = actualSol)

        (x, absErr, errorData) = solver.ConjugateGradientsHS()
        plt.plot(errorData, label = "N = " + str(n))
        index = index + 1

    plt.legend(loc = "upper right", prop={'size':'16'})
    plt.show()

# Steepest Descent testing utility
def testSteepestDescent():
    N = 128
    n = 4
    index = 0
    print("Testing Steepest Descent")

    while(n < N):
        n = 2 * n
        print(n)
        sinEquationDiscr = sed.SimpleEquationDiscretizer(n, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []

        for i in range((n + 1) * (n + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == n or y == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
            else:
                initSol.append(0.0)

        actualSol = fe.actualSinSolution(n)
        solver = sm.SolverMethods(200000, sinEquationDiscr, initSol = initSol, actualSol = [])

        (x, absErr, errorData) = solver.SteepestDescent()
        plt.plot(errorData, label = "N = " + str(n))
        index = index + 1

    plt.legend(loc = "upper right", prop={'size':'16'})
    plt.show()

# Jacobi solver testing utility on a 1D model problem
def testJacobiSmoothing1D():
    N = 32
    h = 1.0 / N
    x = np.arange(0.0, 1.0 + h, h)
    k = 3.0
    initSol = np.sin(math.pi * 8 * x) + np.sin(math.pi * 5 * x) + np.sin(math.pi * 3 *x)

    plt.plot(x,initSol)

    sinEquationDiscr = sed1D.EquationDiscretizer1D(N, fe.zero1D, fe.zero1D)
    solver = sm.SolverMethods(20, sinEquationDiscr, initSol = initSol)
    (y, absErr, errorData, r) = solver.JacobiIterate(1.0)

    plt.plot(x,y)
    plt.show()

# Jacobi solver testing utility
def testJacobi():
    N = 128
    n = 4
    index = 0
    print("Testing Jacobi")
    while(n < N):
        n = 2 * n
        print(n)
        sinEquationDiscr = sed.SimpleEquationDiscretizer(n, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []

        for i in range((n + 1) * (n + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == n or y == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
            else:
                initSol.append(0.0)

        solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
        (x, absErr, errorData, r) = solver.JacobiIterate()
        plt.plot(errorData, label = str(n)+", Jacobi")
        index +=1

    plt.legend(loc = "upper right", prop={'size':'16'})
    plt.show()

# Gauss Seidel solver testing utility
def testGaussSeidel():
    N = 128
    n = 4
    index = 0

    while(n < N):
        n = 2 * n
        print(n, ':')
        sinEquationDiscr = sed.SimpleEquationDiscretizer(n, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []

        for i in range((n + 1) * (n + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == n or y == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
            else:
                initSol.append(0.0)

        solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
        wOpt = 2.0/(1.0 + math.sin(math.pi / n))
        (x, absErr, errorData, r) = solver.GaussSeidelIterate(wOpt)
        plt.plot(errorData, label = str(n) +", SOR", color=lineColor[index])
        index += 1

    plt.show()

# SSOR solver testing utility
def testSSOR():
    N = 256
    n = 4
    index = 0
    print("SSOR TEST")
    while(n < N):
        n = 2 * n
        print(n, ':')
        sinEquationDiscr = sed.SimpleEquationDiscretizer(n, fe.sinBorderFunction, fe.sinValueFunction)

        initSol = []

        for i in range((n + 1) * (n + 1)):
            (x, y) = sinEquationDiscr.getCoordinates(i)
            if(x == 0 or y == 0 or x == n or y == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
            else:
                initSol.append(0.0)

        solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
        wOpt = wSSOR2D[n]

        (x, absErr, errorData, r) = solver.SSORIterate(wOpt)

        plt.plot(errorData, label = str(n) + ", SSOR")
        index += 1

    plt.legend(loc = "upper right", prop={'size':'15'})
    plt.show()

# Helper function which prints the resulting discretization matrix in dense format
# Note: only use this for small values, otherwise making a dense matrix from our
# sparse format becomes too expensive
def printDiscretization():
    f1=open('./testfile', 'w+')

    eqDiscr = sed.SimpleEquationDiscretizer(3, fe.sinBorderFunction, fe.sinValueFunction)
    f1.write(str(eqDiscr.M.todense()))

    f1.close()

# Helper function to plot the exact solution of a 1D model problem
def plotExactSol1D():
    plt.figure(1)
    n = 2
    N = 32
    t = np.arange(0.0, 1.0, 0.01)
    k = 3.0
    s = np.sin(k * np.pi * t)
    index = 0
    while(n < N):
        n = 2 * n
        h = 1.0 / n
        print(n, ':')
        x  = np.arange(0.0, 1.0 + h, h)
        sinEquationDiscr = sed1D.EquationDiscretizer1D(n, fe.sin1DBorderFunction2, fe.sin1DValueFunction2)

        initSol = []

        for i in range(n + 1):
            if(i == 0 or i == n):
                initSol.append(sinEquationDiscr.borderFunction(1.0 * i / n))
            else:
                initSol.append(0)
        M = sinEquationDiscr.M.todense()
        v = sinEquationDiscr.valueVector2D
        exactSol = np.linalg.solve(M, v)
        index = index + 1
        plt.subplot('22'+str(index))
        plt.plot(x,exactSol, 'bo')
        if(index == 1):
            plt.plot(x,exactSol, label = "Linear interpolation of $\mathbf{u}^{4}$")

        elif(index == 2):
            plt.plot(x,exactSol, label = "Linear interpolation of $\mathbf{u}^{8}$")

        elif(index == 3):
            plt.plot(x,exactSol, label = "Linear interpolation of $\mathbf{u}^{16}$")

        elif(index == 4):
            plt.plot(x,exactSol, label = "Linear interpolation of $\mathbf{u}^{32}$")

        plt.plot(t, s, label = "Exact continuous solution: $u$")

    plt.legend(loc = "lower right", prop={'size':'12'})
    plt.title('N = '+str(n))
    plt.show()

# Backward Euler method for solving the Heat Equation (which includes the time variable)
def solveHeatEquationForAllTimeSteps(discr):
    solHeat = discr.initialHeatTimeSolution()
    valueVector = discr.computeVectorAtTimestep(1, solHeat)
    solver = sm.SolverMethods(1000, discr, valueVector)
    sol =[solHeat]

    for k in range(1, discr.T + 1):
        t = k * discr.dT
        valueVector = discr.computeVectorAtTimestep(k, solHeat)
        solver.b = valueVector
        (solHeat, err, _) = solver.ConjugateGradientsHS()
        sol.append(solHeat)
        print(err)
        plotGraph(discr.N, solHeat)

    return sol

# Heat Equation solver testing utility
def testHeatEquationSolver():
    discr = ted.TimeEquationDiscretizer(
        N = 32,
        T = 6,
        borderTimeFunction = fe.heatSinBorderFunction,
        rhsHeatEquationFunction = fe.heatRhsFunction,
        initialHeatTimeFunction = fe.heatInitialFunction,
    )

    solveHeatEquationForAllTimeSteps(discr)
```

```python
# FunctionExamples.py

#  _________________________________________
# |USED FOR SAMPLE MODEL PROBLEMS FUNCTIONS|
# |_________________________________________|

import math

# Function examples for a simple equation discretizer

# Value of the border function on values x,y
def sinBorderFunction(x, y):
    # Assert (x,y) is on border
    value = 1.0 * math.sin(x) * math.sin(y)
    return value

def actualSinSolution(N):
    actualSolution = []
    for i in range(N + 1):
        for j in range(N + 1):
            actualSolution.append(math.sin((1.0) * i / N) * math.sin((1.0) * j / N))
    return(actualSolution)


# RHS value of the differential equation at points x, y
def sinValueFunction(x, y):
    value = - 2.0 * math.sin(x) * math.sin(y)
    return value


def borderFunction1(x, y):
    value = 1.0 * (x * x * x + y * y * y + x + y + 1.0)
    return value

def laplaceValueFunction1(x, y):
    value = 6.0 * x + 6.0 * y
    return value

def sin2BorderFunction(x, y):
    # Assert (x,y) is on border
    k = 2.0
    j = 5.0
    value = 1.0 * math.sin(math.pi * k * x) * math.sin(math.pi * j * y)
    return value


# RHS value of the differential equation at points x, y
def sin2ValueFunction(x, y):
    k = 2.0
    j = 5.0
    value = - 1.0 * math.sin(math.pi * k * x) * math.sin(math.pi * j * y) * math.pi * math.pi * (k * k + j * j)
    return value


# Function examples for a time equation discretizer (heat equation)

def heatSinBorderFunction(x, y, t):
    value = 1.0 * math.sin(x) * math.sin(y)
    value = 0.0
    return value

def heatRhsFunction(x, y, t):
    value = -2.0 * math.sin(x) * math.sin(y)
    value = 0.0
    return value

def heatInitialFunction(x, y, t):
    # Assert t == 0
    value = math.sin(x) * math.sin(y)
    return value



def sin1DValueFunction(x):
    value = - math.sin(x)
    return value

def sin1DBorderFunction(x):
    value = math.sin(x)
    return value


def sin1DValueFunction2(x):
    k = 3.0
    value = - math.sin(math.pi * k * x) * k * k * math.pi * math.pi
    return value

def sin1DBorderFunction2(x):
    k = 3.0
    value = math.sin(math.pi * k * x)
    return value


def zero1D(x):
    return 0
```

```python
# posdefplot.py

#   _______________________________________
# |USED FOR AUXILLIARY PLOTS IN THE THESIS|
# |_______________________________________|

from mpl_toolkits import mplot3d
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import math
matplotlib.rcParams['text.usetex'] = True
matplotlib.rcParams['text.latex.unicode'] = True
tol = 0.00001

A = np.array([[2.0, 1.0], [1.0,3.0]], dtype = np.float)
b = np.array([5.0,5.0], dtype = np.float)

def f(x, y):
    return (1 * x ** 2 +  1.5 * y ** 2 + 1 * x*y - 5 * x - 5 * y)

# ax = plt.axes(projection='3d')
# plt.axis('off')
def plotF():
    x = np.linspace(0.8, 2.5, 20)
    y = np.linspace(0.8, 2, 20)

    X, Y = np.meshgrid(x, y)
    Z = f(X, Y)
    fig = plt.figure()

    ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                    cmap='viridis', edgecolor='none')

    # ax.plot([0,1],[0,1],[0,1])
    # plt.show()

Xs = []
def SteepestDescent(M, b, iterationConstant = 100, plotFirstIterations = False):
    # ax = plt.axes()
    plt.xlim(-2, 3)
    plt.ylim(-2, 2)
    avoidDivByZeroError = 0.0000000000000000001
    x = np.ones_like(b)
    r = np.subtract(b, M.dot(x))
    errorDataSteepestDescent = []
    iterationConstant = iterationConstant
    for i in range(iterationConstant):
        Xs.append(x)

        err = np.subtract(M.dot(x), b)
        absErr = np.linalg.norm(err) / np.linalg.norm(b)
        errorDataSteepestDescent.append(math.log(absErr))

        alpha_numerator = r.dot(r)
        alpha_denominator = r.dot(M.dot(r))
        if(alpha_denominator < avoidDivByZeroError):
            break
        alpha = alpha_numerator / alpha_denominator

        xOld = np.copy(x)
        x = np.add(x, np.dot(r, alpha))

        # Used for plotting first iterations of Steepest descent
        fSize = 26
        strCoord = "("+format(xOld[0], ".2f")+", "+ format(xOld[1], ".2f")+", "+format(f(xOld[0], xOld[1]), ".3f")+")"
        if(i>=0):
            ax.plot([xOld[0], x[0]], [xOld[1], x[1]], [f(xOld[0], xOld[1]), f(x[0], x[1])], color="k", linewidth = 3)
        if(plotFirstIterations):
            if(i == 0 ):
                ax.plot([xOld[0]],[xOld[1]],[f(xOld[0], xOld[1])], markerfacecolor='white', markeredgecolor='white', marker='o', markersize=4, alpha
                ax.text(xOld[0],xOld[1] +0.1,f(xOld[0], xOld[1]),"$\mathbf{x}_0$" + strCoord, fontsize = fSize, color = "white")
            if(i == 1 ):
                ax.plot([xOld[0]],[xOld[1]],[f(xOld[0], xOld[1])], markerfacecolor='white', markeredgecolor='white', marker='o', markersize=4, alpha
                ax.text(xOld[0],xOld[1],f(xOld[0], xOld[1]),"$\mathbf{x}_1$" + strCoord, fontsize = fSize, color = "white")
            if(i == 2 ):
                ax.plot([xOld[0]],[xOld[1]],[f(xOld[0], xOld[1])], markerfacecolor='white', markeredgecolor='white', marker='o', markersize=4, alpha
                ax.text(xOld[0]- 0.1,xOld[1]- 0.1,f(xOld[0], xOld[1]),"$\mathbf{x}_2$" + strCoord, fontsize = fSize, color = "white")
            if(i == 3):
                ax.plot([xOld[0]],[xOld[1]],[f(xOld[0], xOld[1])], markerfacecolor='white', markeredgecolor='white', marker='o', markersize=4, alpha
                ax.text(xOld[0],xOld[1]+0.01,f(xOld[0], xOld[1]),"$\mathbf{x}_3$" + strCoord, fontsize = fSize, color = "white")



        r = np.subtract(b, M.dot(x))
        if(np.linalg.norm(r) < tol):
            break

    err = np.subtract(M.dot(x), b)
    Xs.append(x)
    absErr = np.linalg.norm(err) / np.linalg.norm(b)
    errorDataSteepestDescent.append(math.log(absErr))

    #Plot solution for model problem
    ax.plot([2.0],[1.0],[f(2.0, 1.0)], markerfacecolor='red', markeredgecolor='red', marker='o', markersize=5, alpha=1)
    xSol = 2.0
    ySol = 1.0
    strCoord = "("+format(xSol, ".2f")+", "+ format(ySol, ".2f")+", "+format(f(xSol, ySol), ".3f")+")"
    ax.text(2.0, 0.95, f(2.0, 1.0), "$\mathbf{x}_{sol}$"+strCoord, fontsize = fSize, color = "red")
    plt.show()

    return x, absErr, errorDataSteepestDescent


def ConjugateGradientsHS(M, b):
    avoidDivByZeroError = 0.0000000001
    errorDataConjugateGradients = []
    x = np.zeros_like(b, dtype=np.float)
    r = np.subtract(b, M.dot(x))
    d = np.subtract(b, M.dot(x))
    convergence = False

    while(not convergence):
        solutionError = np.subtract(M.dot(x), b)
        absErr = np.linalg.norm(solutionError)
        try:
            errorDataConjugateGradients.append(math.log(absErr))
        except:
            convergence = True

        if(absErr < tol):
            convergence = True
            break

        alpha_numerator = r.dot(r)
        alpha_denominator = d.dot(M.dot(d))
        if(alpha_denominator < avoidDivByZeroError):
            convergence = True
            break
        alpha = 1.0 * alpha_numerator / alpha_denominator

        x = np.add(x, np.multiply(d, alpha))
        r_new = np.subtract(r, np.multiply(M.dot(d), alpha))

        beta_numerator = r_new.dot(r_new)
        beta_denominator = r.dot(r)
        if(beta_denominator < avoidDivByZeroError):
            convergence = True
            break

        beta = 1.0 * beta_numerator / beta_denominator

        d = r_new + np.multiply(d, beta)
        r = r_new

    return x, absErr, errorDataConjugateGradients

def ConjugateGradients_Golub(M, b):
    errorDataConjugateGradients = []
    tol = 0.000001
    k = 0
    x = np.zeros_like(b)
    r = np.subtract(b, A.dot(x))
    ro_c = r.dot(r)
    delta = tol * np.linalg.norm(b)
    while math.sqrt(ro_c) > delta:
        err = np.subtract(M.dot(x), b)
        absErr = np.linalg.norm(err)
        errorDataConjugateGradients.append(absErr)
        k = k + 1
        if(k == 1):
            p = r
        else:
            tau = ro_c / ro_minus
            p = np.add(r, np.multiply(p, tau))
        w = A.dot(p)
        miu_nominator = ro_c
        miu_denominator = w.dot(p)
        miu = miu_nominator / miu_denominator
        x = np.add(x, np.multiply(p, miu))
        r = np.subtract(r, np.multiply(w, miu))
        ro_minus = ro_c
        ro_c = r.dot(r)

    err = np.subtract(M.dot(x), b)
    absErr = np.linalg.norm(err)
    errorDataConjugateGradients.append(absErr)
    return x, absErr, errorDataConjugateGradients


def plotDiscretizedSine1D():
    N = 16
    h = 1.0 / N
    r = 1.0 / 100.0
    xCont = np.arange(0.0, 1.0 + r, r)
    xDiscr = np.arange(0.0, 1.0 + h, h)

    xCont2 = np.arange(0.0, 1.0 + r, r)
    xDiscr2 = np.arange(0.0, 1.0 + h, h)


    k = 5.0
    contFunction = np.sin(math.pi * k * xCont)
    discrFunction = np.sin(math.pi * k * xDiscr)

    contFunction2 = np.sin(math.pi * 8.0 * xCont2)
    discrFunction2 = np.sin(math.pi * 8.0 * xDiscr2)
    plt.subplot(211)
    plt.plot(xCont, contFunction)
    plt.plot(xCont, contFunction2)

    plt.subplot(212)
    plt.plot(xDiscr, discrFunction, linestyle='dashed')
    plt.plot(xDiscr, discrFunction, 'ko')

    plt.plot(xDiscr, discrFunction2, linestyle='dashed')
    plt.plot(xDiscr, discrFunction2, 'ko')
    plt.show()

def plotSineModes():
    N = 16
    h = 1.0 / N
    xDiscr = np.arange(0.0, 1.0 + h, h)

    discrFunction1 = np.sin(math.pi * 1.0  * xDiscr)
    discrFunction2 = np.sin(math.pi * 4.0 * xDiscr)
    discrFunction3 = np.sin(math.pi * 10.0 * xDiscr)
    discrFunction4 = np.sin(math.pi * 14.0 * xDiscr)
    discrFunction5 = np.sin(math.pi * 16.0 * xDiscr)

    plt.subplot(511)
    plt.title("$\sin{(\pi x)}$")
    plt.plot(xDiscr, discrFunction1, linestyle='dashed')
    plt.plot(xDiscr, discrFunction1, 'ko')

    plt.subplot(512)
    plt.title("$\sin{(4 \pi x)}$")
    plt.plot(xDiscr, discrFunction2, linestyle='dashed')
    plt.plot(xDiscr, discrFunction2, 'ko')

    plt.subplot(513)
    plt.title("$\sin{(10 \pi x)}$")
    plt.plot(xDiscr, discrFunction3, linestyle='dashed')
    plt.plot(xDiscr, discrFunction3, 'ko')

    plt.subplot(514)
    plt.title("$\sin{(14 \pi x)}$")
    plt.plot(xDiscr, discrFunction4, linestyle='dashed')
    plt.plot(xDiscr, discrFunction4, 'ko')

    plt.subplot(515)
    plt.title("$\sin{(31 \pi x)}$")
    plt.plot(xDiscr, discrFunction5, linestyle='dashed')
    plt.plot(xDiscr, discrFunction5, 'ko')

    plt.show()


def plotProjectedSine():
    N1 = 16
    h1 = 1.0 / N1
    xDiscr1 = np.arange(0.0, 1.0 + h1, h1)

    N2 = 8
    h2 = 1.0 / N2
    xDiscr2 = np.arange(0.0, 1.0 + h2, h2)
    xDiscr22 = np.arange(0.0, 2.0 + h2, h2)

    discrFunction1 = np.sin(math.pi * 6.0 * xDiscr1)
    discrFunction2 = np.sin(math.pi * 6.0 * xDiscr2)
    discrFunction22 = np.sin(math.pi * 6.0 * xDiscr22)

    plt.subplot(311)
    plt.plot(xDiscr1, discrFunction1, linestyle='dashed')
    plt.plot(xDiscr1, discrFunction1, 'ko')

    plt.subplot(312)
    plt.plot(xDiscr2, discrFunction2, linestyle='dashed')
    plt.plot(xDiscr2, discrFunction2, 'ko')

    plt.subplot(313)
    plt.plot(xDiscr22, discrFunction22, linestyle='dashed')
    plt.plot(xDiscr22, discrFunction22, 'ko')
    plt.show()

def plotGrid():
     plt.plot([0.0, 0.0, 0.0, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0],[0.0, 0.5, 1.0, 0.0, 0.5, 1.0, 0.0, 0.5, 1.0], 'ro', marker = 'o', markersize = 10, mark
     plt.plot([0, 0, 0.25, 0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 0.75, 0.75, 0.75, 1, 1], [0.25, 0.75, 0, 0.25, 0.5, 0.75, 1, 0.25, 0.75, 0,
     plt.show()

def plotInterpolatedExample():
    N = 4.0
    h1 = 1.0 / N
    h2 = 0.5 / N
    xDiscr1 = np.arange(0.0, 1.0 + h1, h1)
    xDiscr2 = np.arange(0.0, 1.0 + h2, h2)
    discrFunction1 = [0.0, 0.3, -0.2, 0.1, 0.0]
    discrFunction2 = []
    for i in range(len(discrFunction1) -1):
        discrFunction2.append(discrFunction1[i])
        discrFunction2.append((discrFunction1[i] + discrFunction1[i+1])/2.0)
    discrFunction2.append(discrFunction1[len(discrFunction1)-1])
    plt.subplot(211)
    plt.plot(xDiscr1, discrFunction1, linestyle='dashed')
    plt.plot(xDiscr1, discrFunction1, 'ko')

    plt.subplot(212)
    plt.plot(xDiscr2, discrFunction2, linestyle='dashed')
    plt.plot(xDiscr2, discrFunction2, 'ko')
    plt.show()

def plotRestrictionExample():
    N = 4.0
    h1 = 0.5 / N
    h2 = 1.0 / N
    xDiscr1 = np.arange(0.0, 1.0 + h1, h1)
    xDiscr2 = np.arange(0.0, 1.0 + h2, h2)

    discrFunction1 = [0.0, 0.1, -0.1, 0.15, -0.2, 0.1, -0.13, 0.09, 0.0]
    discrFunction2 = []

    for i in range(len(discrFunction1)):
        if(i % 2 == 0):
            discrFunction2.append(discrFunction1[i])

    discrFunction3 = []
    for i in range(len(discrFunction1)):
        if(i == 0 or i == len(discrFunction1)-1):
            discrFunction3.append(0.0)
        elif(i%2 == 0):
            discrFunction3.append((2 * discrFunction1[i] + discrFunction1[i-1]+discrFunction1[i+1]) / 4.0)

    plt.subplot(311)
    plt.ylim(-0.22, 0.22)
    plt.plot(xDiscr1, discrFunction1, linestyle='dashed')
    plt.plot(xDiscr1, discrFunction1, 'ko')

    plt.subplot(312)
    plt.ylim(-0.22, 0.22)
    plt.plot(xDiscr2, discrFunction2, linestyle='dashed')
    plt.plot(xDiscr2, discrFunction2, 'ko')

    plt.subplot(313)
    plt.ylim(-0.22, 0.22)
    plt.plot(xDiscr2, discrFunction3, linestyle='dashed')
    plt.plot(xDiscr2, discrFunction3, 'ko')
    plt.show()


def plotVCycle():
    plt.axis('off')
    h = 0.1
    plt.xlim(0,6)
    plt.ylim(0.5, 3.5)
    xs = [1, 2, 3, 4, 5]
    ys = [3, 2, 1, 2, 3]
    plt.plot(xs, ys)
    plt.plot(xs, ys, 'ro', markerfacecolor='k', markeredgecolor='k', markersize = 10)
    plt.text(1.0 - 7 * h, 3.0 + h,"$\Omega^{h}$: relax $\mu_1$ times", fontsize = 20)
    plt.text(2.0 - 12 * h, 2.0 ,"$\Omega^{2h}$: relax $\mu_1$ times", fontsize = 20)
    plt.text(3.0, 1.0 - 2 * h, "$\Omega^{4h}$: compute exact solution", fontsize = 20)
    plt.text(4.0 + 2 * h, 2.0,"$\Omega^{2h}$: relax $\mu_2$ times", fontsize = 20)
    plt.text(5.0 - 2 * h, 3.0 + h,"$\Omega^{h}$: relax $\mu_2$ times", fontsize = 20)

    plt.text(1.5 - 2 * h, 2.5 - h, "$\mathbf{I}_{h}^{2h}$", fontsize = 20)
    plt.text(2.5 - 2 * h, 1.5 - h, "$\mathbf{I}_{2h}^{4h}$", fontsize = 20)

    plt.text(3.5 + 1.5 * h, 1.5 - h, "$\mathbf{I}_{4h}^{2h}$", fontsize = 20)
    plt.text(4.5 + 1.5 * h, 2.5 - h, "$\mathbf{I}_{2h}^{h}$", fontsize = 20)

    plt.arrow(1.0, 3.0, 0.8, -0.8, head_width = 0.1, head_length = 0.2)
    plt.arrow(2.0, 2.0, 0.8, -0.8, head_width = 0.1, head_length = 0.2)

    plt.arrow(3.0, 1.0, 0.8, 0.8, head_width = 0.1, head_length = 0.2)
    plt.arrow(4.0, 2.0, 0.8, 0.8, head_width = 0.1, head_length = 0.2)

    plt.arrow(1.05, 3.0, 3.8, 0.0, head_width = 0.05, head_length = 0.05, color = "red")
    plt.arrow(2.05, 2.0, 1.8, 0.0, head_width = 0.05, head_length = 0.05, color = "red")

    plt.text(3 - 5 * h, 3 - h, "Correct approximation on $\Omega^{h}$", color = "red", fontsize = 13)
    plt.text(3 - 5 * h, 2 - h, "Correct approximation on $\Omega^{2h}$", color = "red", fontsize = 13)

    plt.show()

plotVCycle()
```

Bibliography

[1] Albl, C. Conjugate gradients explained. http://cmp.felk.cvut.cz/ alblcene/CG/CG_explained.pdf.

[2] Allen, R. C., Bottcher, C., et al. Computational science education project. https://www.phy.ornl.gov/csep/la/node14.html.

[3] Barrett, R. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics.

[4] Briggs, W. L., Henson, V. E., and McCormick, S. F. *A Multigrid Tutorial, Second Edition.* Society for Industrial and Applied Mathematics, 2000.

[5] Gai, X. ., Acklam, E., Langtangen, H. P., and Tveito, A. . *Advanced Topics in Computational Partial Differential Equations.* Springer, 2003.

[6] Golub, G. H., and Loan, C. F. V. *Matrix Computations, 4th ed.* The John Hopkins University Press, 2715 North Charles Street, Baltimore, Maryland, 2013.

[7] Hadjidimos, A. Successive overrelaxation (sor) and related methods. *Journal of Computational and Applied Mathematics 123* (2000), 177–199.

[8] Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49*, 6 (1952).

[9] Horn, R. A., and Johnsons, C. R. *Matrix Analysis, 2nd ed.* Cambridge University Press, 2013.

[10] (https://scicomp.stackexchange.com/users/3824/patrick sanan), P. S. How is krylov-accelerated multigrid (using mg as a preconditioner) motivated? Computational Science Stack Exchange. URL:https://scicomp.stackexchange.com/q/19946 (version: 2017-04-13).

[11] Iserles, A. *A First Course in the Numerical Analysis of Differential Equations.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK, 2009.

[12] Li, R. C. Iterative schemes and their convergence for a symmetric positive definite matrix.

[13] R.Kettler, and Meijerink, J. A multigrid method and a combined multigrid-conjugate gradient method for elliptic problems with strongly discontinuous coefficients in general domains.

[14] Saad, Y. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2003.

[15] Shankar, V. The finite difference method for elliptic problems. Tech. rep., The University of Utah, Department of Mathematics, 2016.

[16] Shewchuk, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* 1994.

[17] Strang, G. *Introduction to Linear Algebra, 4th ed.* Wellesley-Cambridge Press, Box 812060 Wellesley MA 02482, 2009.

[18] Strang, G. *Linear Algebra and its Applications, Fourth Edition.* Thomson Higher Education, 2009.

[19] Suli, E., and Mayers, D. *An Introduction to Numerical Analysis.* Cambridge University Press, 2003.

[20] Tatebe, O. The multigrid preconditioned conjugate gradient method. *NASA Conference Publication*, 3224 (1993), 621–634.

[21] Trefethen, L. N., and Bau, D. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, United States of America, 1997.

[22] Yang, S., and Gobbert, M. K. The optimal relaxation parameter for the sor method applied to a classical model problem.

[23] Young, D. M. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.