

An Investigation of Iterative Methods for Large Scale Linear Systems



Catalin-Andrei Ilie
Oriol 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

1	Motivation	1
1.1	Introduction	1
1.2	A class of differential equations	2
1.3	Note on implementation	3
2	Discretization of the equations	4
2.1	Deducing the approximations	4
2.1.1	Finite differences method	4
2.1.2	Backward Euler	5
2.1.3	Creating a linear system	7
2.1.4	Properties of matrices arising from discretization	8
2.2	Sparse data structures	11
2.2.1	Requirements	11
2.2.2	CSR format	11
2.3	Measuring the approximation algorithms performance	12
2.3.1	Convergence of methods	12
2.3.2	Continuous approximation	13
3	Classic iterative methods	16
3.1	Matrix splitting methods	16
3.1.1	General Formulation	16
3.1.2	Jacobi	17
3.1.3	Gauss-Seidel and SOR	18
3.1.4	SSOR	22
3.1.5	Comparison of Jacobi, SOR and SSOR	25
3.2	Gradient methods	26
3.2.1	Steepest Descent	27
3.2.2	Conjugate Gradient Method	28

3.2.2.1	Asymptotic behaviour	34
3.2.2.2	Preconditioned Conjugate Gradient Method	34
4	The Multigrid Method	36
4.1	Motivation	36
4.2	Prolongation and restriction operators	37
4.2.1	Interpolation	38
4.2.2	Restriction	39
4.3	The 2-grid algorithm	40
4.4	The V-Cycle	41
4.5	Multigrid setup	42
4.6	Cost of Multigrid V-cycle	43
4.6.1	Space	43
4.6.2	Time	43
4.7	Discussion	44
4.8	Multigrid as a preconditioner for Conjugate Gradient Method	48
4.9	Comparison between solvers	49
5	Conclusions	52
A	Implementation	53
	Bibliography	71

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:

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

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 $\overline{\Omega_T} = \overline{\Omega} \times [0, 1] = ([0, 1] \times [0, 1]) \times [0, 1]$:

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

where k is a strictly positive constant.

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

$$\nabla^2 u = f. \quad (1.3)$$

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

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

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 $i = 0, 1, \dots, N$, and $y_i = \frac{i}{N}$, for $i = 0, 1, \dots, N$ (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 $i = 0, 1, \dots, T$.

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:

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

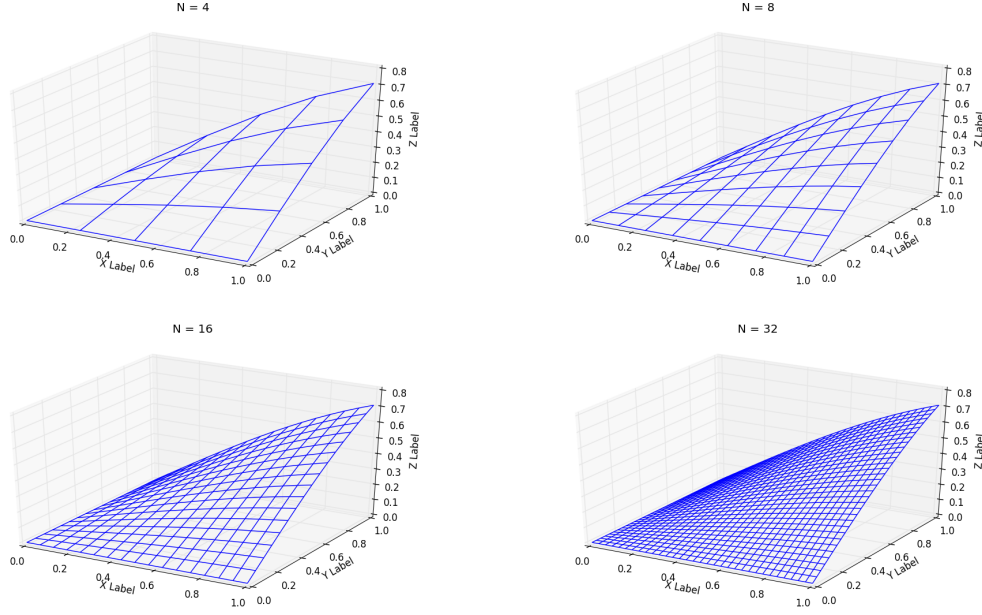


Figure 2.1: Different levels of discretization

and

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

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

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

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

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

2.1.2 Backward Euler

In order to approach (2.3), we iterate over t_k , for $k = 1, 2, \dots, T$, 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:

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

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  $i = 1; i \leq T; i++$  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    $u_i \leftarrow \text{SolveHeat}(L_i, e_i)$  // solve for the current time step
5  $u \leftarrow (u_0, u_1, \dots, u_T)$ 
6 return  $u$  // where  $u(x_m, y_n, t_p) = u_p(x_m, y_n)$ 

```

The algorithm 1 calls a procedure $\text{SolveHeat}(L_i, e_i)$ 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:

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

A solution approximation will be the output of a procedure $\text{SolvePoisson}(L, e)$, 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 :

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

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 $N = 4$:

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

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, $\mathbf{A}_{\text{Heat}}^t = \mathbf{A}_{\text{Heat}}$, so we adopt the same construction described above for \mathbf{A}_{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}_{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:*

$$|a_{ii}| \geq \sum_{j=1, j \neq i}^n |a_{ij}|, i = 1, 2, \dots, n. \quad (2.9)$$

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 $\mathbf{x} \in \mathbb{R}^n - \{\mathbf{0}\}$. (\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 \mathbf{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 $R_i(\mathbf{A}) = \sum_{j=1, j \neq i}^n |a_{ij}|, i = 1, 2, \dots, n$. Consider the n Gershgorin discs:*

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

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

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

The matrix constructed for the 2D Heat Equation, \mathbf{A}_{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}_{Heat} below for $N = T = 4$, with no border points in the matrix, and the linear equation at time step $j + 1$ (for $j = 0, 1, \dots, N - 1 = 0, 1, 2, 3$). 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}_{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 positive definite as well ([15]).

To conclude, both \mathbf{A}_{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} is 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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ and let \mathbf{z} be an approximation of \mathbf{x}^* . We call the vector $\mathbf{e} = \mathbf{x}^* - \mathbf{z}$ the error and $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{z}$ 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 $\lim_{k \rightarrow \infty} \mathbf{u}^{N,k} = \mathbf{u}^N$ (where \mathbf{u}^N is the discrete approximation at level N). Also, $\lim_{N \rightarrow \infty} \mathbf{u}^N = u$ (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 $\lim_{k \rightarrow \infty} \mathbf{u}^{N,k} = \mathbf{u}^N$, 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 $\lim_{k \rightarrow \infty} \|\mathbf{u} - \mathbf{u}^{N,k}\| = \mathcal{O}(\frac{1}{N})$.

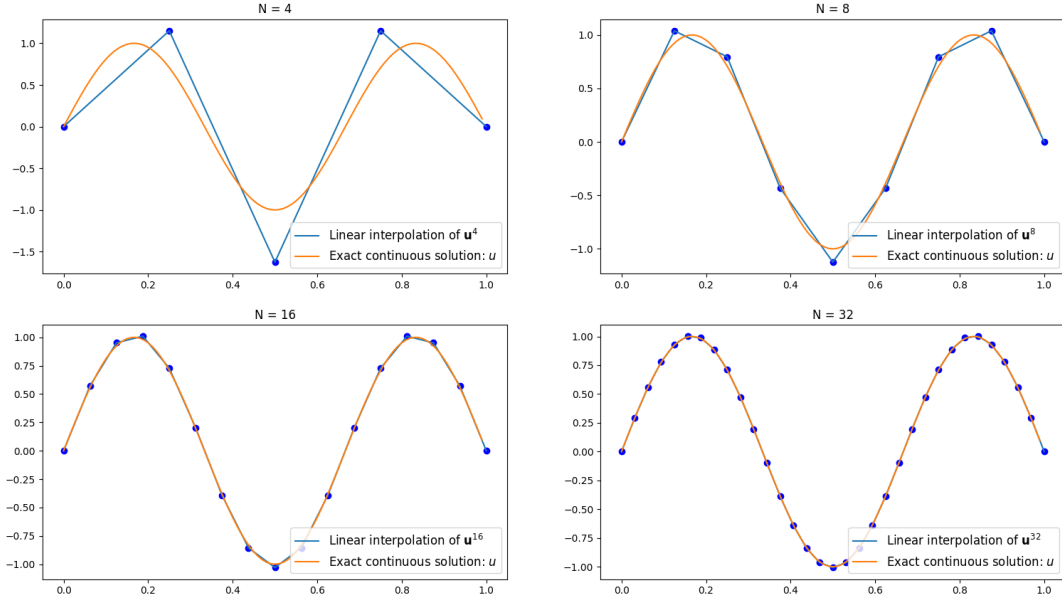


Figure 2.2: Linear interpolations of the exact solutions of Finite Difference approximations for different grid sizes (blue) versus the exact continuous solution (orange) of a 1D Poisson Equation with exact solution $f(x) = \sin 3\pi x$

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 $F : [0, 1] \rightarrow \mathbb{R}$ 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.

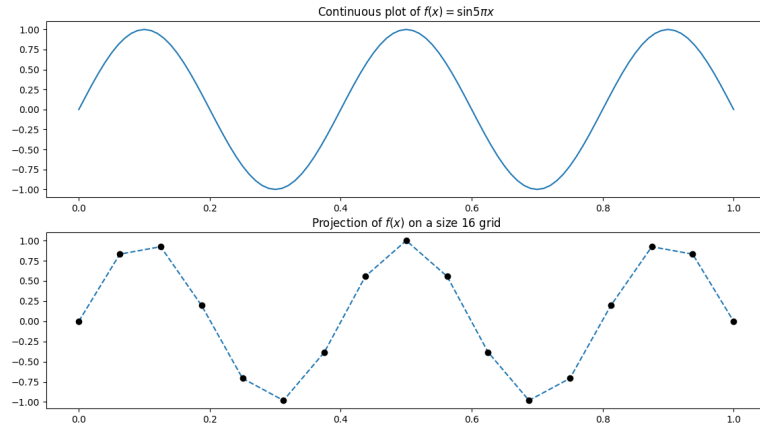


Figure 2.3: The first plot shows the continuous $f(x) = \sin 5\pi x$. It has wave number 5 and frequency 2.5 (2 complete sine cycles and a half). The second plot represents the projection of $f(x)$ onto a grid with $h = \frac{1}{16}$ spacing of the interval $[0, 1]$. We will also be referring to the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

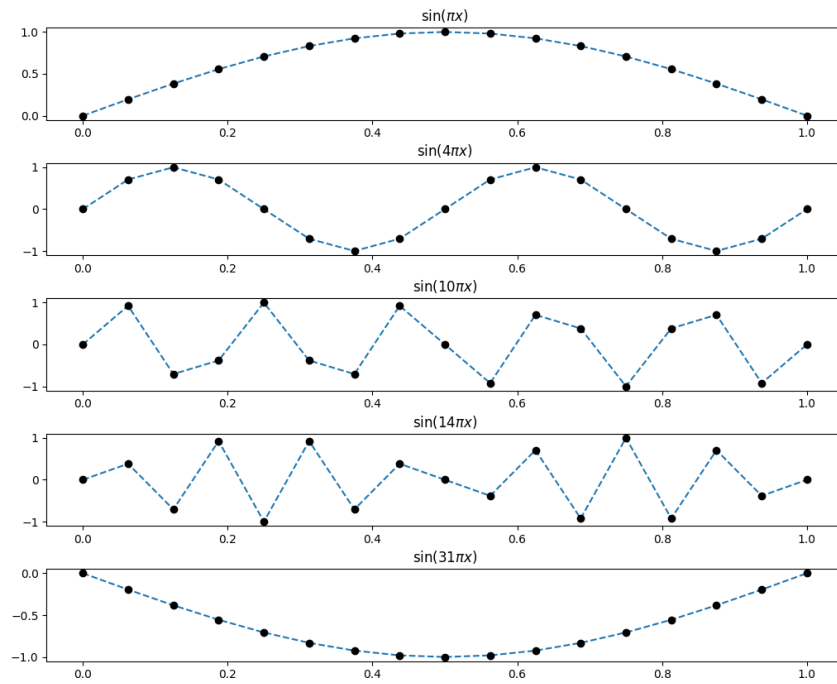


Figure 2.4: Sine modes with wave numbers 1, 4, 10, 14 and 31. Note that the 31 mode represents the same frequency as the 1 mode because of aliasing, we should not consider modes with wave number above 16.

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:

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

which can naturally be interpreted as an iterative scheme:

$$\mathbf{x}^{k+1} = \mathbf{T}\mathbf{x}^k + \mathbf{c}, \text{ with } \mathbf{x}^0 \text{ an initial approximation.} \quad (3.2)$$

In standard texts, \mathbf{T} is called *iteration matrix* and \mathbf{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:

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

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 $\omega \in (0, 1]$:

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

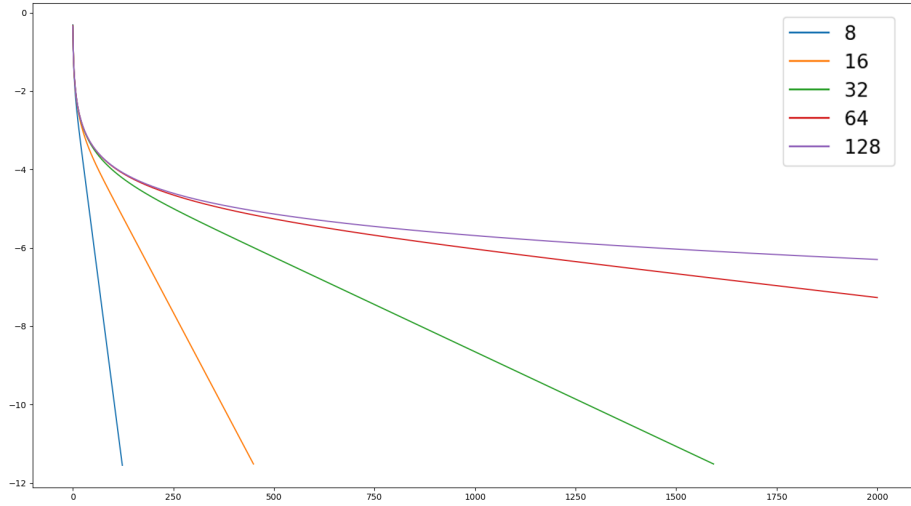


Figure 3.1: Logarithmic scale of the residual Euclidean norm in terms of the number of Jacobi ($\omega = 1$) iterations for different grid sizes of the 2D Poisson Equation discretization with exact solution $f(x, y) = \sin(x)\sin(y)$. The legend describes the sample number N . Note that the number of 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 $\mathbf{w}_k, k = 1, 2, \dots, n$ with associated eigenvalues $\theta_k, k = 1, 2, \dots, n$. 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}_ω and \mathbf{A} have the same eigenvectors, and the eigenvalues of \mathbf{T}_ω are $\lambda_k = 1 - \frac{\omega}{4} \theta_k, k = 1, 2, \dots, n$. 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:

$$\mathbf{e}_k = \mathbf{T}_\omega^k \sum_{i=1}^{i=n} c_i \mathbf{w}_i = \sum_{i=1}^{i=n} c_i \lambda_i^k \mathbf{w}_i. \quad (3.5)$$

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:

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

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.

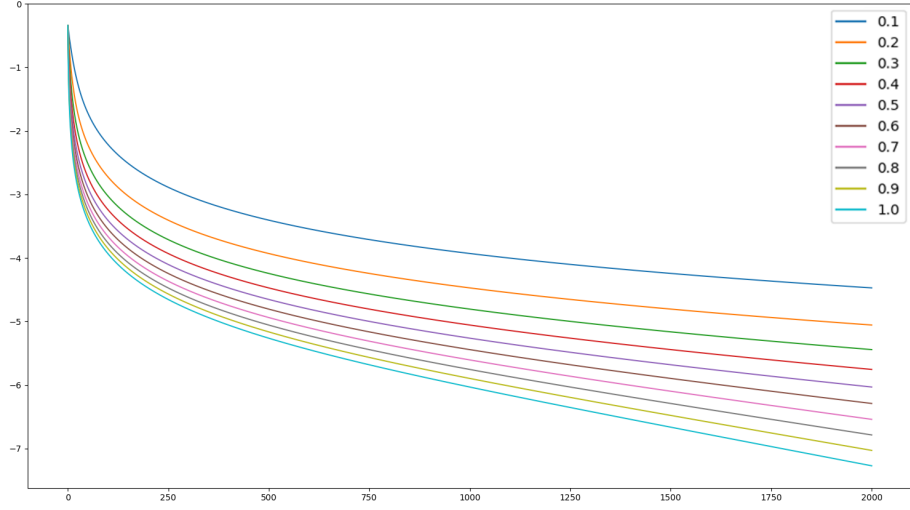


Figure 3.2: Logarithmic scale of the residual Euclidian norm in terms of the number of damped Jacobi iterations with varying ω . The bigger ω is, the better general convergence we get, as the spectrum of frequencies that arises is large.

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.

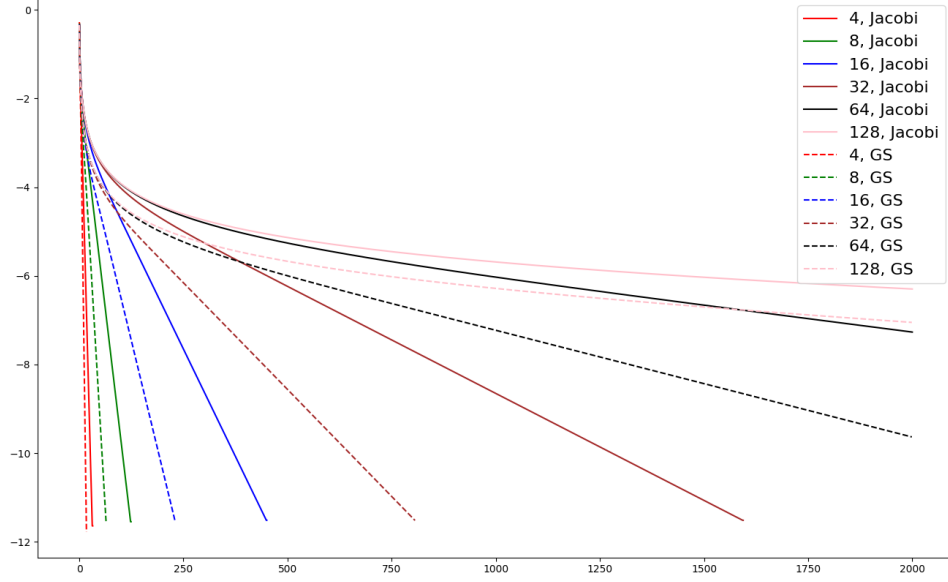


Figure 3.3: Comparison in logarithmic scale of the convergence of Gauss-Seidel (dashed line) and Jacobi (complete line) in the limit of 2000 iterations under the same setup as figure (3.1).

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:

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

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

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

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}_ω 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:

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

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 $\omega_{\text{optimal}} = \frac{2}{1 + \sqrt{1 - \rho(\mathbf{F})^2}}$.*

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).

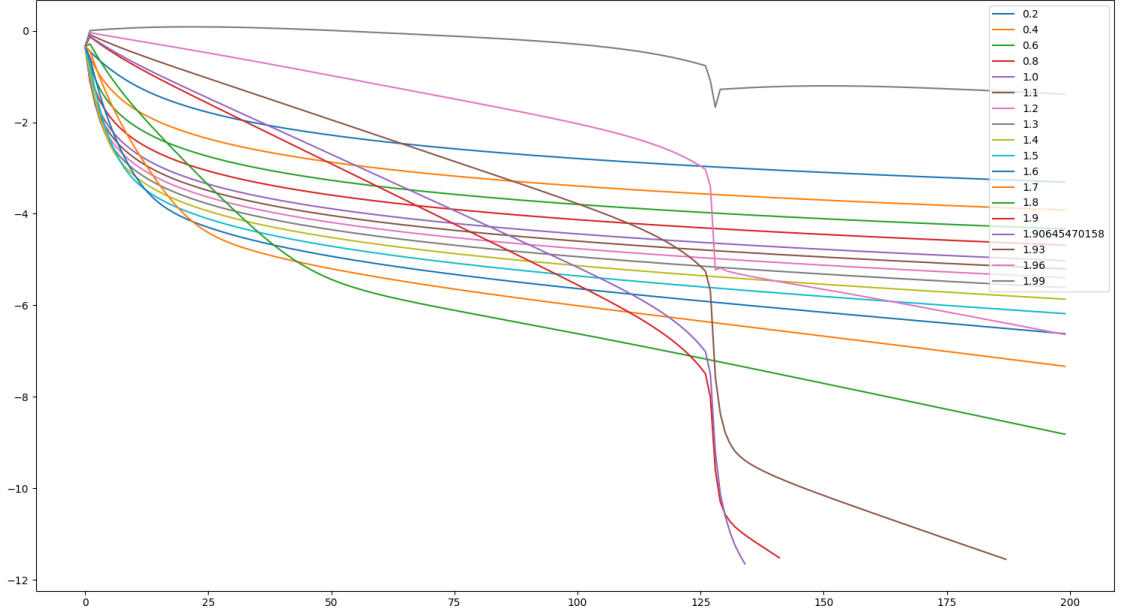


Figure 3.4: Logarithmic scale of the residual norm in terms of the number of SOR iterations for different ω values, for a grid size of $(N + 1) \times (N + 1)$, where $N = 64$. It can be seen that $\omega = \frac{2}{1 + \sin \pi h} \approx 1.906455$ yields indeed the best convergence rate. For $\omega = 2$ 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):

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

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

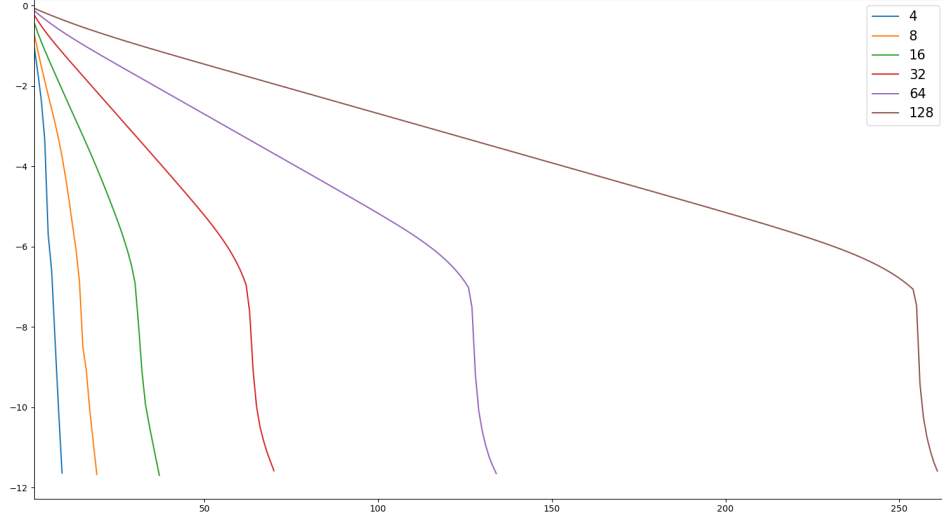


Figure 3.5: Logarithmic scale of the residual norm in terms of the number of SOR iterations for different N values, where the grid size is $(N + 1) \times (N + 1)$. The optimal ω is used for each N . There is clearly a strong mesh dependence of the convergence rate.

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}_{init} , \mathbf{b} , ω , iterationCount

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

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.

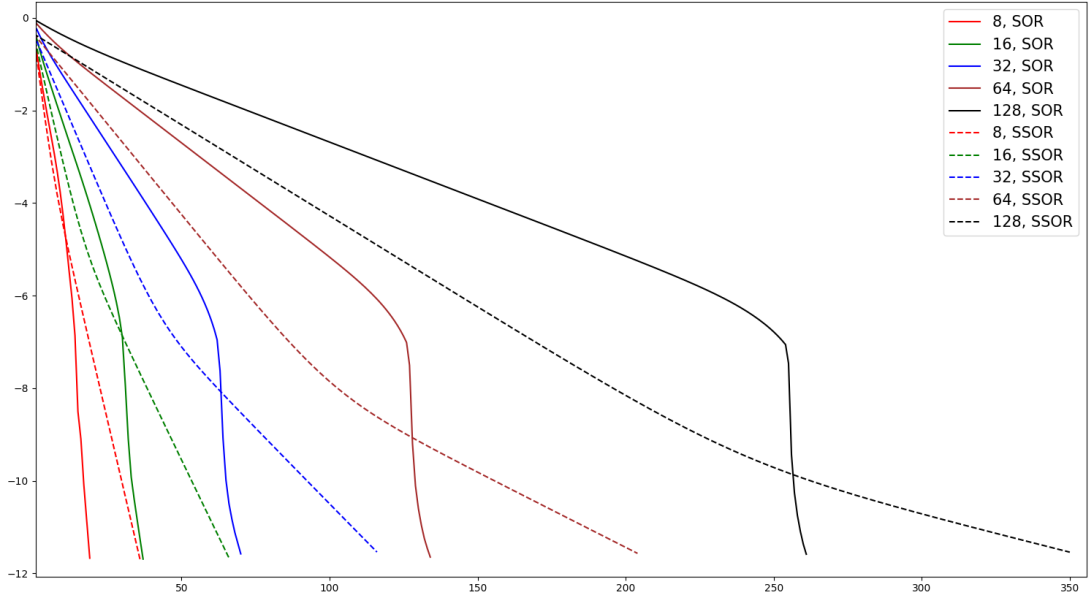


Figure 3.6: Logarithmic scale of the residual norm in terms of SOR iterations using the optimal ω_{SOR} for varying grid size of the 2D model problem discretization plotted using continuous lines. Logarithmic scale of the residual norm in terms of SSOR subiterations (2 subiterations per SSOR iteration) 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 $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$ be $\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{Ax} - \mathbf{x}^T \mathbf{b}$. Then, the solution \mathbf{x}^* of $\mathbf{Ax} = \mathbf{b}$ is the same as the minimum point of ϕ (i.e. $\arg \min_{\mathbf{x} \in \mathbb{R}^n} \phi(\mathbf{x}) = \mathbf{x}^*$). 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 $\nabla_{\mathbf{x}}^2 \phi(\mathbf{x}) = \nabla_{\mathbf{x}}(\frac{1}{2}(\mathbf{Ax} + \mathbf{A}^T \mathbf{x}) - \mathbf{b}) = \nabla_{\mathbf{x}}(\mathbf{Ax} - \mathbf{b}) = \mathbf{A}$ 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, $\nabla_{\mathbf{x}} \phi(\mathbf{x}^*) = \mathbf{Ax}^* - \mathbf{b} = \mathbf{0}$, so \mathbf{x}^* is a stationary point of ϕ . As ϕ is strictly convex, \mathbf{x}^* is the unique minimum point of ϕ , therefore $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b} = \arg \min_{\mathbf{x} \in \mathbb{R}^n} \phi(\mathbf{x})$.

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

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. □

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  $\mathbf{x}$  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)

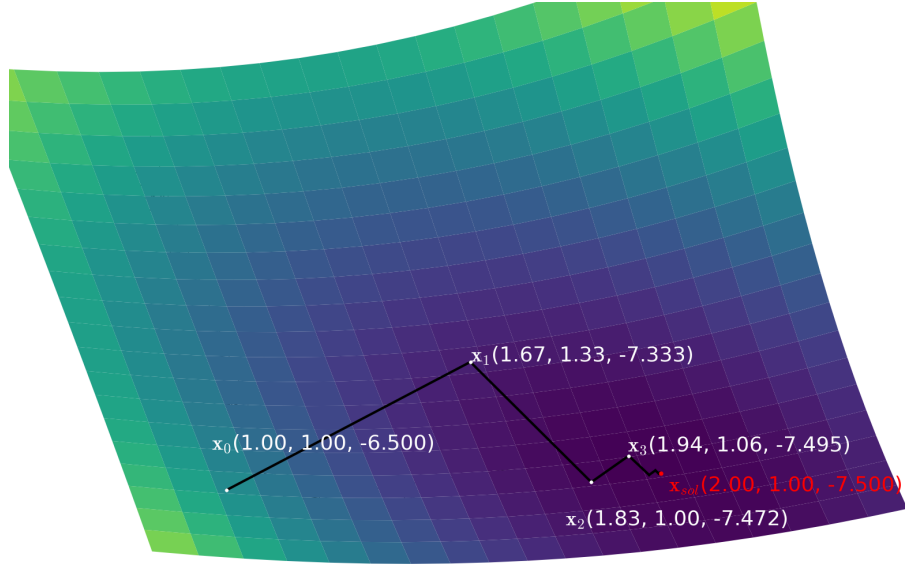


Figure 3.7: The first three steps ($\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$) of Steepest Descent, the initial guess ($\mathbf{x}_0 = [1, 1]^T$) and the minimum point (\mathbf{x}_{sol}) are shown on the plot of the example function $\phi_1(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \mathbf{x} - \mathbf{x}^T \begin{bmatrix} 2 \\ 1 \end{bmatrix}$. 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 $\mathbb{D} = \{\mathbf{d}_0, \mathbf{d}_1, \dots, \mathbf{d}_{n-1}\}$ 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 $\mathbf{x}^* = \mathbf{x}_0 + \sum_{i=0}^{n-1} \beta_i \mathbf{d}_i$ 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:

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

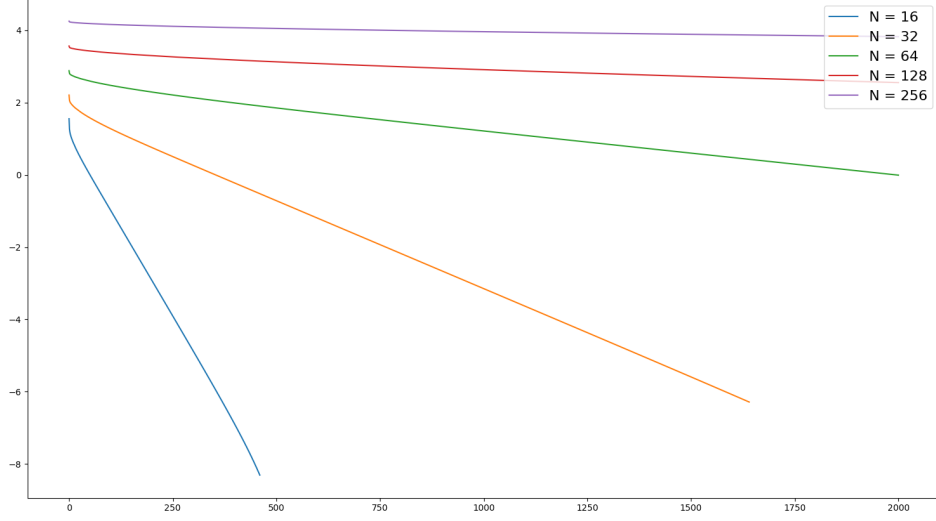


Figure 3.8: Logarithmic scale of the exact error \mathbf{e}_k Euclidean norm of the Steepest Descent method on the sample 2D Poisson equation. The plot is relevant to see how fast we are approaching the actual solution, but in general this quantity is not available. We have chosen the functions in the 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

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

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 $\mathbb{V} = \{\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_{n-1}\}$, for $i = 0, 1, \dots, n-1$ construct \mathbf{d}_i by eliminating from \mathbf{v}_i the $\mathbf{d}_0 \dots \mathbf{d}_{i-1}$ components under \mathbf{A} -projection of \mathbf{v}_i . Start with $\mathbf{d}_0 = \mathbf{v}_0$ and, for $i > 0$, construct

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

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 $i \neq j$, 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 Gaussian 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 $\mathbb{D}_k = \text{span}\{\mathbf{d}_0, \dots, \mathbf{d}_{k-1}\} = \text{span}\{\mathbf{r}_0, \dots, \mathbf{r}_{k-1}\}$ 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 $\{\mathbf{d}_0, \dots, \mathbf{d}_{k-1}\}$ and $\{\mathbf{r}_0, \dots, \mathbf{r}_{k-1}\}$. 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 \mathbf{A} -orthogonal works.

Note that

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

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

$$\mathbb{D}_k = \text{span}\{\mathbf{d}_0, \mathbf{A}\mathbf{d}_0, \dots, \mathbf{A}^{k-1} \mathbf{d}_0\} = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{k-1} \mathbf{r}_0\}. \quad (3.16)$$

Observation 3.2.1. $\mathbb{D}_k = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \dots, \mathbf{A}^{k-1} \mathbf{r}_0\}$ 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 $i \neq k+1$ 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 $\delta_{k+1k} = -\frac{1}{\beta_k} \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{d}_k^T \mathbf{A}\mathbf{d}_k}$ 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:

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

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:

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

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  $\mathbf{x}$  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  $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{Ax}_0$ 
3  $\mathbf{d}_0 \leftarrow \mathbf{b} - \mathbf{Ax}_0$ 
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    $\mathbf{r}_{k+1} \leftarrow \mathbf{b} - \mathbf{Ax}_{k+1}$ 
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.

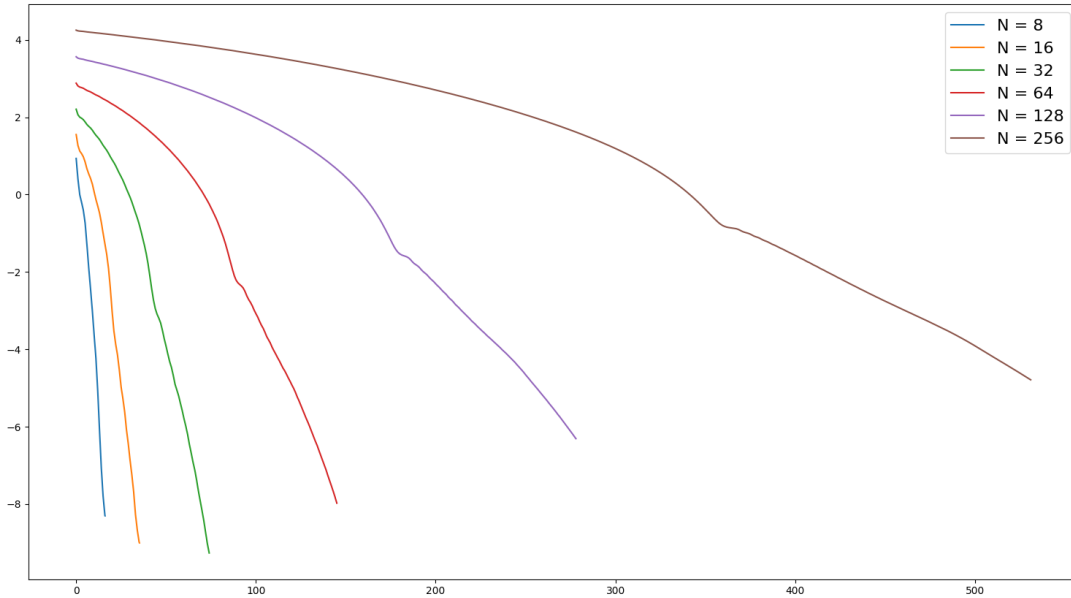


Figure 3.9: Logarithmic scale of the exact error \mathbf{e}_k Euclidean norm of the Conjugate Gradient method on the same sample 2D Poisson equation as before. As in figure 3.8, the exact errors are not available in general, but in our case we know an explicit solution and it helps to illustrate how 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 $k \in \mathcal{O}(n^{2/d})$.

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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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 $\mathbf{A}\mathbf{x} = \mathbf{b} \iff \mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}\mathbf{M}_2\mathbf{x} = \mathbf{M}_1^{-1}\mathbf{b}$. 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  $\mathbf{x}$  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      $\alpha_i \leftarrow \frac{\tilde{\mathbf{r}}_i^T \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{Ap}_i}$ 
6      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$ 
7      $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{Ap}_k$ 
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 $\mathbf{x}_{\mu_1}^h$ such that we diminish the high frequency components of the error. Note that $\mathbf{A}^h \mathbf{e}^h = \mathbf{A}^h (\mathbf{x}^h - \mathbf{x}_{\mu_1}^h) = \mathbf{b}^h - \mathbf{A}^h \mathbf{x}_{\mu_1}^h = \mathbf{r}^h$, 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 $\mathbf{x}_{\mu_1}^h$ 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.

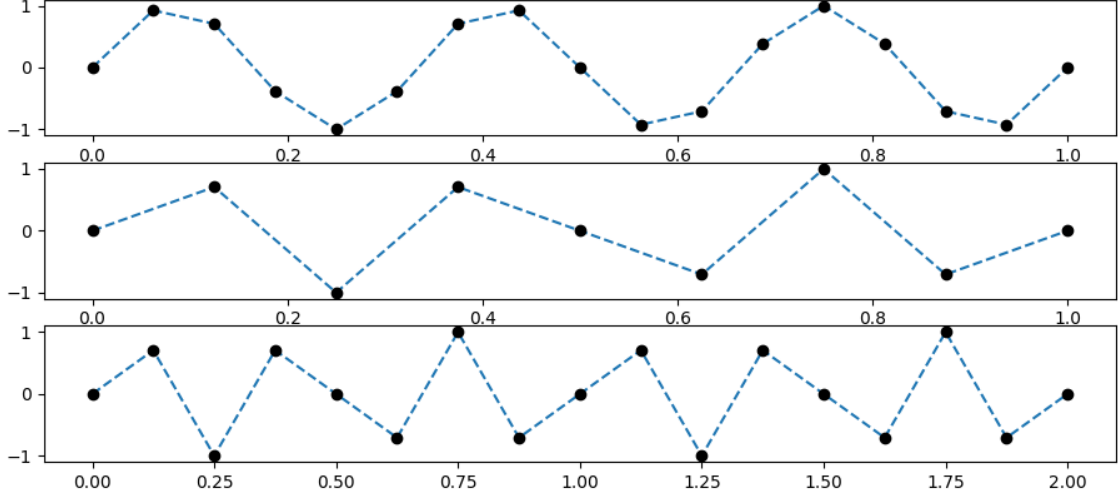


Figure 4.1: The first plot represents $\sin 6\pi x$ on the grid Ω^h , where $h = \frac{1}{16}$. The second plot represents the same function on the coarser grid Ω^{2h} . The third plot represents the same function on a grid of spacing $2h$ on the interval $[0, 2]$. Note that its restriction to $[0, 1]$ 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 6th mode out of 16 modes, and on Ω^{2h} it is the 6th 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 $\mathbf{v}^{2h} = [v_0, v_1, \dots, v_{N/2}]$ be a vector representing a 1D discretization on grid Ω^{2h} . Its linear interpolation to grid Ω^h is $\mathbf{v}^h = [w_0, w_1, \dots, w_N]$ (see figure 4.2), where:

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

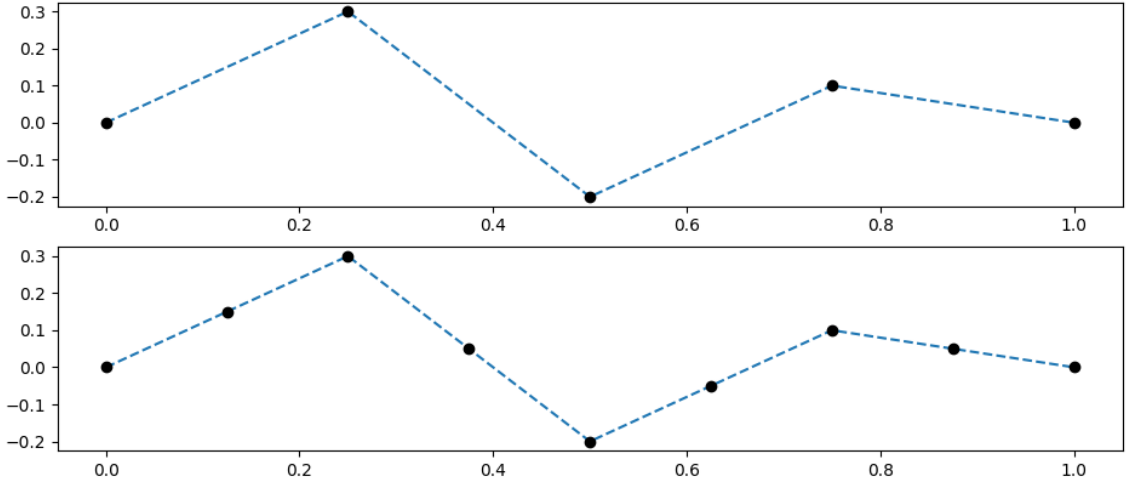


Figure 4.2: Linearly interpolating a discretized function from grid $\Omega^{1/4}$ to grid $\Omega^{1/8}$. In this

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

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

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

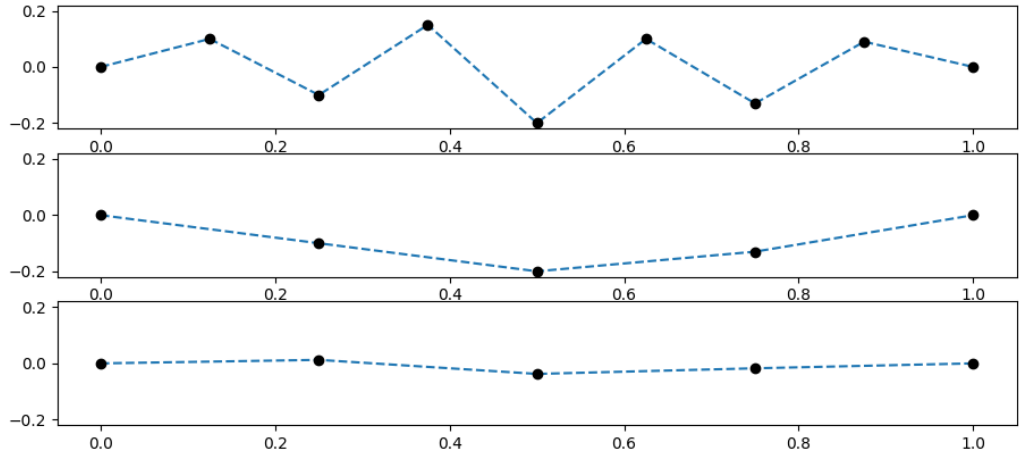


Figure 4.3: The first plot represents a discretized function f of high frequency on grid $\Omega^{1/8}$. The second plot is the trivial restriction of f on $\Omega^{1/4}$ and the third plot is the full weighting restriction of f on $\Omega^{1/4}$. Note that the restrictions do not capture the behaviour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

$\mathbf{v}^h = [w_0, w_1, \dots, w_N]$ be the vector we wish to restrict. The trivial restriction gives $\mathbf{v}^{2h} = [v_0, v_1, \dots, v_{N/2}]$, where $v_i = w_{2i}$ for $i = 0, 1, \dots, N/2$. 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.

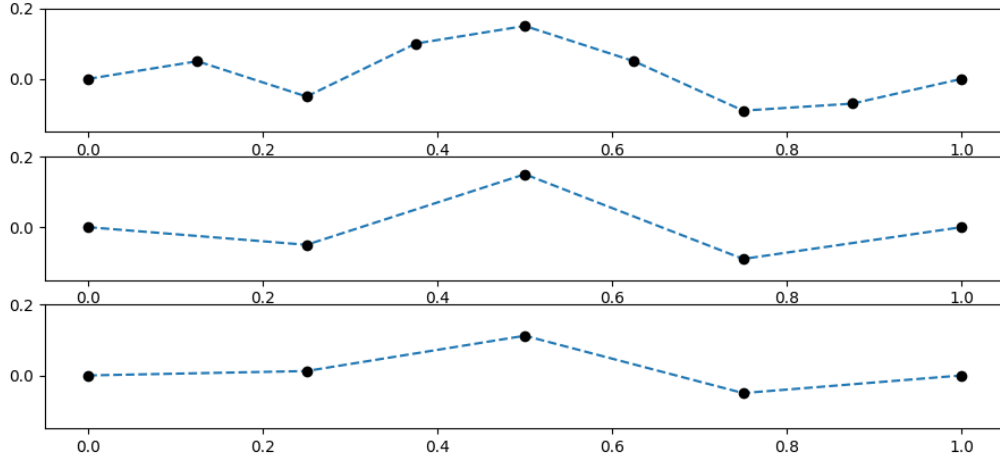


Figure 4.4: The first plot represents a discretized function f on grid $\Omega^{1/8}$. The second plot is the trivial restriction of f on $\Omega^{1/4}$ and the third plot is the full weighting restriction of f on $\Omega^{1/4}$. The restrictions are fairly good approximations of the discretization on 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 $\mathbf{b}^{4h} = \mathbf{I}_{2h}^{4h}\mathbf{r}^{2h}$,
- ...
- 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 $\Omega^{2^{k-2} h}$: $\mathbf{w}^{2^{k-2} h} = \mathbf{I}_{2^{k-1} h}^{2^{k-2} h}\mathbf{u}_{\mu_2}^{2^{k-1} h}$,
- ...
- 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.

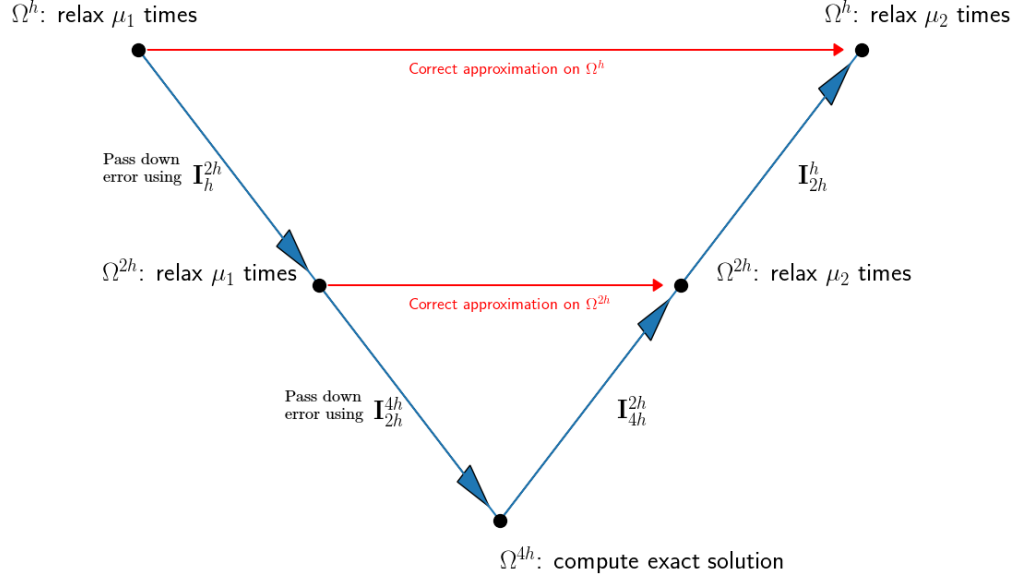


Figure 4.5: One Multigrid V-cycle for a problem with 3 grid levels.

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:

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

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 :

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

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 ($\mathcal{O}(n)$). The right setup can

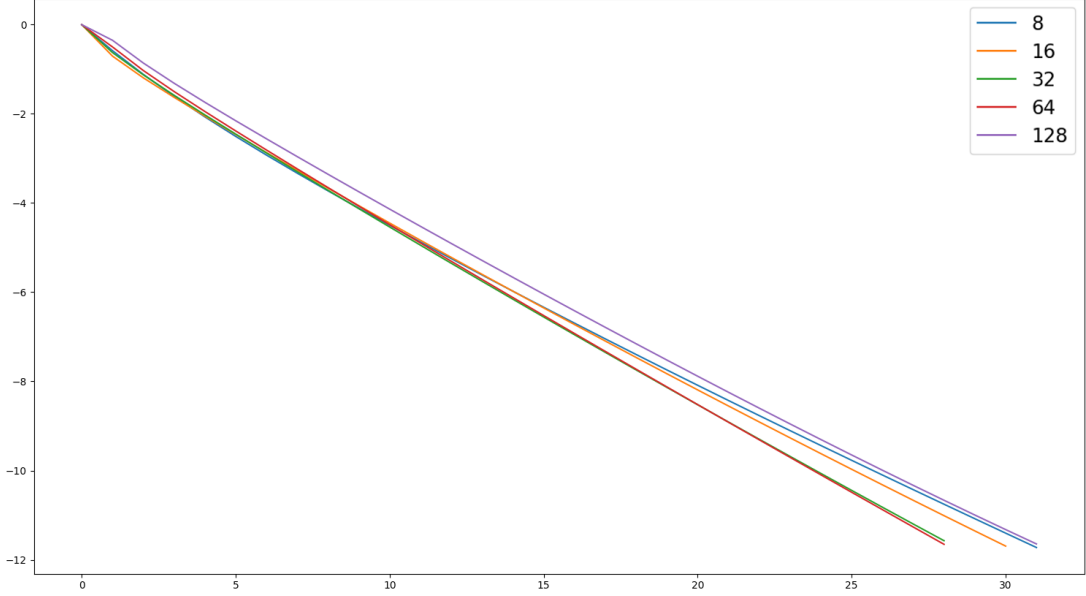


Figure 4.6: Residual error of the 2D Poisson model problem discretization in terms of number of V-cycles for varying grid sizes. The Multigrid V-cycle scheme is applied with SSOR ($\omega = 1.93$) as a relaxation scheme, $\mu_1 = \mu_2 = 2$ relaxation steps. The number of V-cycles is (approximately)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

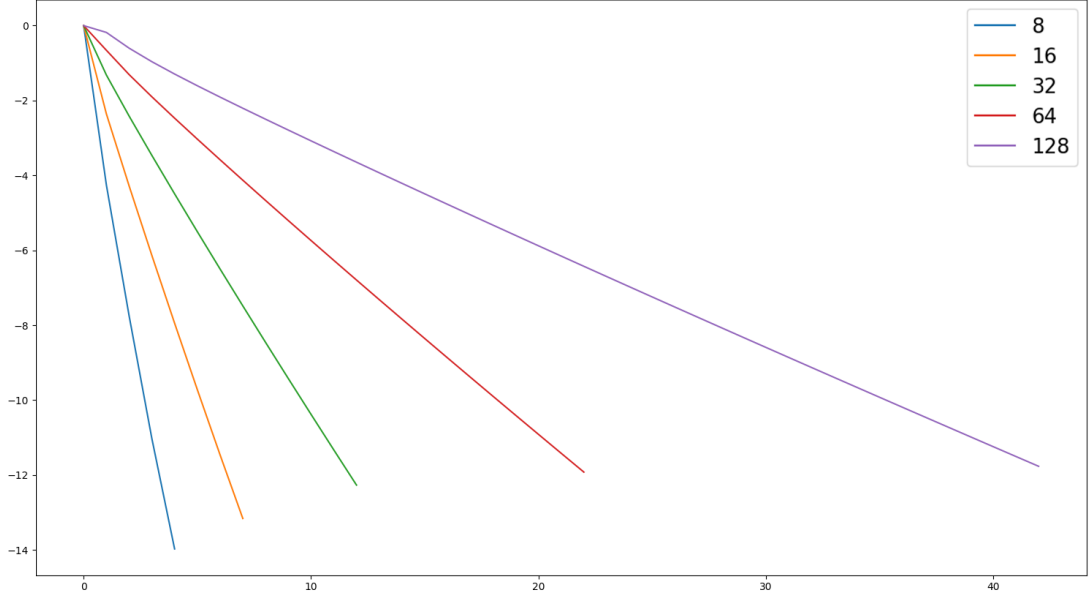


Figure 4.7: Choosing ω to be the optimal relaxation factor of SSOR on the finest grid on the same problem and setup as in figure 4.6.

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

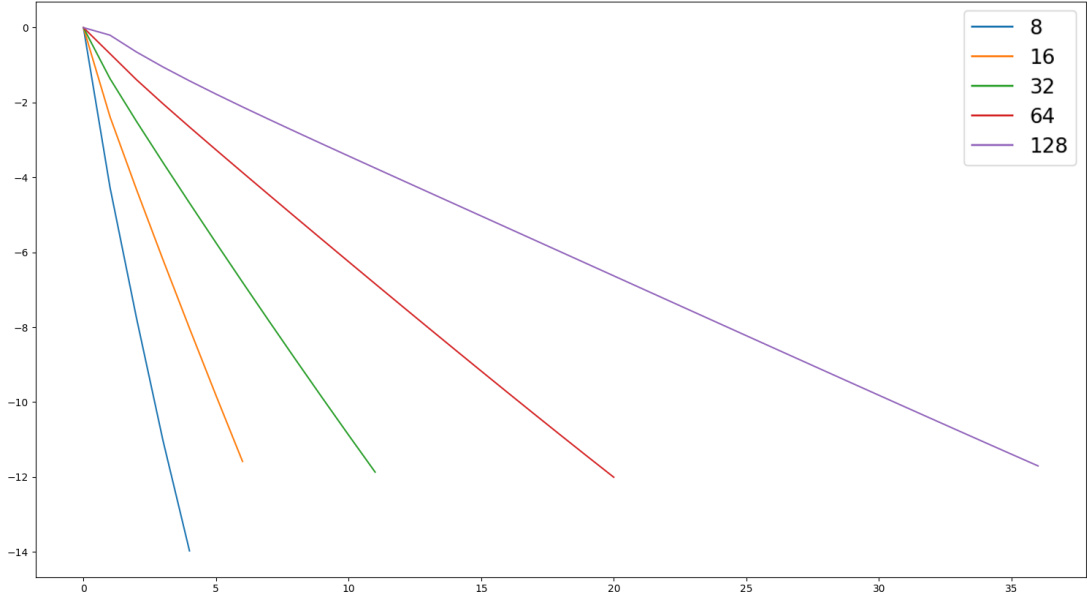


Figure 4.8: The behaviour of the adaptive method we propose. Changing ω on each grid of a V-cycle to its optimal value for SSOR on that grid size on the same problem and setup as in figure 4.6.

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 $\omega \in \{1.9, 1.91, 1.93\}$.

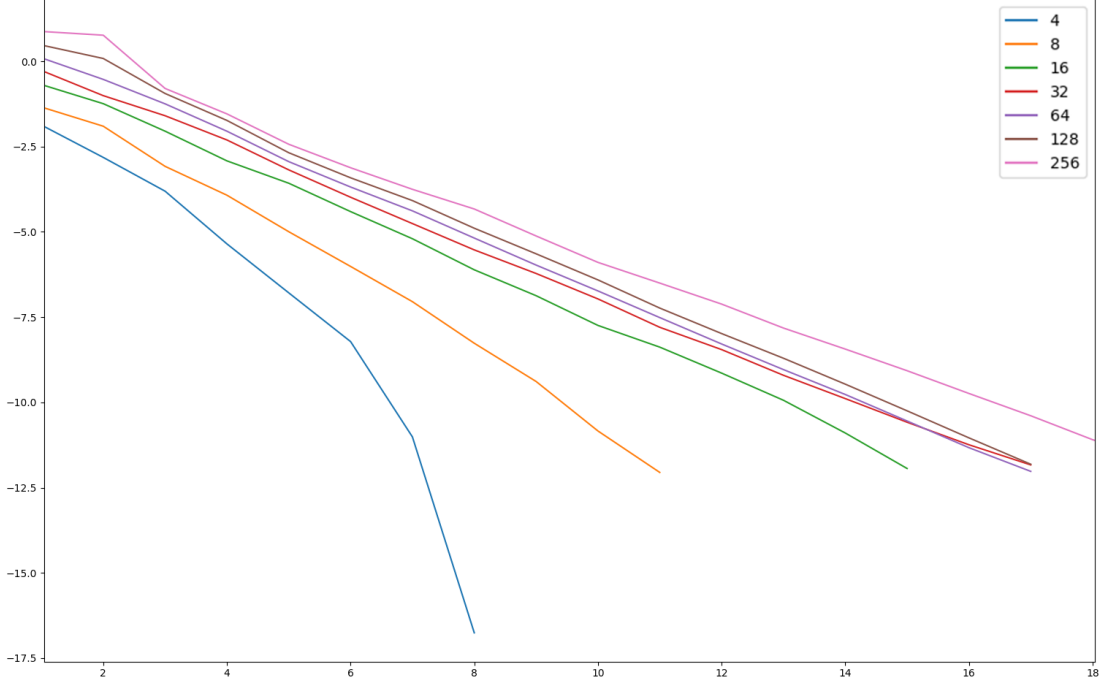


Figure 4.9: Logarithmic scale of the residual error of the 2D Poisson model problem discretization in terms of number of MG preconditioned CG iterations. Per each iteration, we apply 2 MG iterations with SSOR as a smoother ($\omega = 1.97, \mu_1 = \mu_2 = 2$). The number of steps is almost independent 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 sensitive 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 and 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 model problems on which we performed the experiments.

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

```

1 # SolverMethods.py
2
3 # [CLASSIC CUSTOMIZABLE ITERATIVE SCHEMES]
4
5 #
6
7 import numpy as np
8 from scipy.sparse import *
9 from scipy import *
10 import math
11
12 tol = 0.00001
13
14 # Class encapsulating iterative methods for solving a linear system
15 class SolverMethods:
16     def __init__(self, iterationConstant, eqDiscretizer, b = [], initSol = [], actualSol = [], actualSol = [], initB = []):
17         self.iterationConstant = iterationConstant
18         self.M = eqDiscretizer.M
19         self.D = eqDiscretizer.D
20         self.R = eqDiscretizer.R
21         self.L = eqDiscretizer.L
22         self.U = eqDiscretizer.U
23         if b == []:
24             self.b = eqDiscretizer.valueVector2D
25         else:
26             self.b = b
27         self.initB = self.b
28         self.initSol = initSol
29         self.actualSol = actualSol
30
31         # Jacobi iterative method
32         def JacobiIterate(self, dampFactor = 1.0):
33             x = []
34             d = self.D.diagonal()
35             iterationConstant = self.iterationConstant
36             # Initial guess is x = (0,0,...,0) if not provided as a parameter
37             if(self.initSol == []):
38                 x = np.zeros_like(self.b)
39             else:
40                 x = self.initSol
41
42         # Iterate constant number of times
43         for i in range(iterationConstant):
44             err = np.subtract(self.M.dot(x), self.b)
45             absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
46             if(absErr < tol):
47                 break
48             errorDataJacobi.append(math.log(absErr))
49
50             y = self.R.dot(x)
51             r = np.subtract(self.b, y)
52             xPrev = np.copy(x)
53             x = [r_i / d_i for r_i, d_i in zip(r, d)]
54             xNew = np.add(np.multiply(xPrev, (1.0 - dampFactor)), np.multiply(x, dampFactor))
55             x = np.copy(xNew)
56
57         err = np.subtract(self.b, self.M.dot(x))
58         absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
59         errorDataJacobi.append(math.log(absErr))
60         return x, absErr, errorDataJacobi, err
61
62         # Jacobi iterative method v2, these 2 implementations have been used
63         # to decide the more efficient way of implementing it
64         def JacobiIterate2(self, omega = 1.0):
65             x = []
66             currentLowerRows = []
67             currentUpperRows = []
68             d = self.D.diagonal()
69             iterationConstant = self.iterationConstant
70             # Initial guess is x = (0,0,...,0) if not provided as a parameter
71             if(self.initSol == []):
72                 x = np.zeros_like(self.b)
73             else:
74                 x = self.initSol
75
76         for j in range(self.L.shape[0]):

```

```

82         currentLowerRows.append(self.L.getrow(j))
83         currentUpperRows.append(self.U.getrow(j))
84
85         # Iterate constant number of times (TODO: iterate while big error (x-b))
86         for i in range(iterationConstant):
87             err = np.subtract(self.M.dot(x), self.b)
88             absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
89             errorDataJacobi.append(math.log(absErr))
90
91             if(absErr < tol):
92                 break
93
94             xNew = np.zeros_like(x)
95             for j in range(self.L.shape[0]):
96                 currentLowerRow = currentLowerRows[j]
97                 currentUpperRow = currentUpperRows[j]
98
99                 rowSum = currentLowerRow.dot(x) + currentUpperRow.dot(x - x[j] * d[j])
100                 rowSum = 1.0 * (self.b[j] - rowSum) / d[j]
101                 xNew[j] = x[j] + omega * (rowSum - x[j])
102
103             if np.allclose(x, xNew, rtol=1e-6):
104                 break
105
106             x = np.copy(xNew)
107
108
109             err = np.subtract(self.b, self.M.dot(x))
110             absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
111             errorDataJacobi.append(math.log(absErr))
112             return x, absErr, errorDataJacobi, err
113
114         # Gauss Seidel iterative method
115         def GaussSeidelIterate(self, omega = 1.0):
116             x = []
117             d = self.L.diagonal()
118             iterationConstant = self.iterationConstant
119             flops = 0
120
121             currentLowerRows = []
122             currentUpperRows = []
123
124             if(self.initSol == []):
125                 x = np.zeros_like(self.b)
126             else:
127                 x = self.initSol
128
129             for j in range(self.L.shape[0]):
130                 currentLowerRows.append(self.L.getrow(j))
131                 currentUpperRows.append(self.U.getrow(j))
132
133                 for i in range(iterationConstant):
134                     err = np.subtract(self.M.dot(x), self.b)
135                     absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
136                     errorDataGaussSeidel.append(math.log(absErr))
137
138                     xNew = np.zeros_like(x)
139                     for j in range(self.L.shape[0]):
140                         currentLowerRow = currentLowerRows[j]
141                         rowSum = currentLowerRow.dot(xNew) + currentUpperRow.dot(x)
142                         flops += max(2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) - 1, 0)
143                         rowSum = 1.0 * (self.b[j] - rowSum) / d[j]
144                         xNew[j] = x[j] + omega * (rowSum - x[j])
145                         flops += 3
146
147                     if(absErr < tol):
148                         break
149
150                     x = np.copy(xNew)
151
152             print("Flops: ", flops)
153             print("Iterations: ", len(errorDataGaussSeidel) - 1)
154             return x, absErr, errorDataGaussSeidel, err
155
156
157
158
159
160
161
162
163

```

```

164 # SSOR Iterative method
165 def SSORIterate(self, omega = 1.0, debugOn = False):
166     x = []
167     d = self.D.diagonal()
168     iterationConstant = self.iterationConstant
169
170     flops = 0
171
172     currentLowerRows = []
173     currentUpperRows = []
174
175     if(self.initSol == []):
176         x = np.zeros_like(self.b)
177     else:
178         x = np.copy(self.initSol)
179
180     for i in range(self.L.shape[0]):
181         currentLowerRows.append(self.L.getrow(i))
182         currentUpperRows.append(self.U.getrow(i))
183
184     err = np.subtract(self.M.dot(x), self.b)
185     absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
186     errorDataSSOR.append(math.log(absErr))
187
188     for k in range(iterationConstant):
189         xNew = np.zeros_like(x)
190
191         for i in range(self.L.shape[0]):
192             currentLowerRow = currentLowerRows[i]
193             currentUpperRow = currentUpperRows[i]
194
195             currSum = currentLowerRow.dot(xNew) + currentUpperRow.dot(x)
196             flops += max(2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) - 1, 0)
197
198             currSum = 1.0 * (self.b[i] - currSum) / d[i]
199             flops += 2
200
201             xNew[i] = x[i] + omega * (currSum - x[i])
202             flops += 3
203
204         x = np.copy(xNew)
205
206         if(debugOn and k % 10 == 0):
207             print("Iteration: ", k)
208             print("After top to bottom: ", x)
209
210         xNew = np.zeros_like(x)
211         for i in reversed(range(self.L.shape[0])):
212             currSum = 0.0
213             currentLowerRow = currentLowerRows[i]
214             currentUpperRow = currentUpperRows[i]
215
216             currSum = currentLowerRow.dot(x) + currentUpperRow.dot(xNew) - d[i] * x[i]
217             flops += 2 * (currentLowerRow.getnnz() + currentUpperRow.getnnz()) + 1
218
219             currSum = 1.0 * (self.b[i] - currSum) / d[i]
220             flops += 2
221
222             xNew[i] = x[i] + omega * (currSum - x[i])
223             flops += 3
224
225         x = np.copy(xNew)
226
227         if(debugOn and k%10 ==0):
228             print("After bottom to top: ", x)
229             print("_____")
230
231         err = np.subtract(self.M.dot(x), self.b)
232         absErr = np.linalg.norm(err) / (np.linalg.norm(self.b))
233         errorDataSSOR.append(math.log(absErr))
234
235         if(absErr < tol):
236             break
237
238     err = np.subtract(self.b, self.M.dot(x))
239     absErr = np.linalg.norm(err) / np.linalg.norm(self.b)
240
241     return x, absErr, errorDataSSOR, err, flops
242
243
244
245
246
247 # Conjugate Gradient using the Hestenes Stiefel formulation
248 def ConjugateGradientsHS(self):
249     flops = 0
250     matrixDots = 0
251     vectorAddSub = 0
252     vectorDotVector = 0
253
254     M = self.M
255     b = self.b
256     actualSol = self.actualSol
257
258     avoidDivByZeroError = 0.0000000001
259     errorDataConjugateGradients = []
260     x = np.zeros_like(b, dtype=np.float)
261     r = np.subtract(b, M.dot(x))
262     d = np.copy(r)
263     matrixDots += 1
264     vectorAddSub += 1
265
266     convergence = False
267     beta_numerator = r.dot(r)
268     vectorDotVector += 1
269
270     while(not convergence):
271         solutionError = np.subtract(M.dot(x), b)
272         relativeResidualErr = np.linalg.norm(solutionError) / np.linalg.norm(self.initB)
273
274         if(relativeResidualErr < tol):
275             convergence = True
276             break
277
278         if(actualSol != []):
279             err = np.subtract(actualSol, x)
280             absErr = np.linalg.norm(err)
281             errorDataConjugateGradients.append(math.log(absErr))
282         else:
283             errorDataConjugateGradients.append(math.log(relativeResidualErr))
284
285         Md = M.dot(d)
286         alpha_numerator = beta_numerator
287         alpha_denominator = d.dot(Md)
288         vectorDotVector += 1
289         matrixDots += 1
290
291         if(alpha_denominator < avoidDivByZeroError):
292             convergence = True
293             break
294
295         alpha = 1.0 * alpha_numerator / alpha_denominator
296         flops += 1
297
298         x = np.add(x, np.multiply(d, alpha))
299         flops += len(d)
300         vectorAddSub += 1
301
302         r_new = np.subtract(r, np.multiply(Md, alpha))
303         flops += len(Md)
304         vectorAddSub += 1
305
306         beta_numerator = r_new.dot(r_new)
307         beta_denominator = alpha_numerator
308         vectorDotVector += 1
309
310         if(beta_denominator < avoidDivByZeroError):
311             convergence = True
312             break
313
314         beta = 1.0 * beta_numerator / beta_denominator
315         flops += 1
316
317         d = r_new + np.multiply(d, beta)
318         vectorAddSub += 1
319         flops += len(d)
320
321         r = r_new
322
323     nonZero = M.nonzero()
324     MNZ = len(nonZero[0])
325     flops += vectorAddSub * len(x) + vectorDotVector * (2 * len(x) - 1) + matrixDots * (2 * MNZ - len(x))
326
327

```

```

328         return x, relativeResidualErr, errorDataConjugateGradients
329
330     # Steepest descent method
331     def SteepestDescent(self):
332         avoidDivByZeroError = 0.000000000000000000000001
333
334         flops = 0
335         matrixDots = 0
336         vectorAddSub = 0
337         vectorDotVector = 0
338         actualSol = self.actualSol
339
340         M = self.M
341         b = self.b
342
343         x = np.zeros_like(b)
344         r = np.subtract(b, M.dot(x))
345         matrixDots += 1
346         vectorAddSub += 1
347
348         errorDataSteepestDescent = []
349         iterationConstant = self.iterationConstant
350         for i in range(iterationConstant):
351             err = np.subtract(M.dot(x), b)
352             divide = np.linalg.norm(b)
353             if(actualSol != []):
354                 err = np.subtract(x, actualSol)
355                 divide = 1.0
356             absErr = np.linalg.norm(err) / divide
357             errorDataSteepestDescent.append(math.log(absErr))
358
359             alpha_numerator = r.dot(r)
360             alpha_denominator = r.dot(M.dot(r))
361             vectorDotVector += 2
362             matrixDots += 1
363
364             if(alpha_denominator < avoidDivByZeroError):
365                 break
366
367             alpha = alpha_numerator / alpha_denominator
368             flops += 1
369
370             x = np.add(x, np.dot(r, alpha))
371             flops += len(x)
372             vectorAddSub+=1
373
374             r = np.subtract(b, M.dot(x))
375             vectorAddSub += 1
376             matrixDots += 1
377
378             if(np.linalg.norm(r) / np.linalg.norm(b) < tol):
379                 break
380
381             NNZ = M.getnnz()
382             flops += vectorAddSub * len(x) + vectorDotVector * (2 * len(x) - 1) + matrixDots * (2 * NNZ - len(x))
383
384             err = np.subtract(M.dot(x), b)
385             divide = np.linalg.norm(b)
386             if(actualSol != []):
387                 err = np.subtract(x, actualSol)
388                 divide = 1.0
389             absErr = np.linalg.norm(err) / divide
390             errorDataSteepestDescent.append(math.log(absErr))
391
392         return x, absErr, errorDataSteepestDescent

```

```

1 # MGMethods.py
2
3 #
4 # |Multigrid based advanced methods, including MGCG|
5 # |-----|
6
7 import numpy as np
8 from scipy.sparse import *
9 from scipy import *
10 import numpy.linalg as la
11 import matplotlib.pyplot as plt
12 from mpl_toolkits.mplot3d import Axes3D
13
14 import SimpleEquationDiscretizer as sed
15 import EquationDiscretizerID as sedID
16 import SolverMethods as sm
17 import FunctionExamples as fe
18 import TimeEquationDiscretizer as ted
19
20 tol = 0.00001
21
22 #
23 # Sanity check function for determining if a matrix is positive definite
24 def is_pos_def(x):
25     return np.all(np.linalg.eigvals(x) > 0)
26
27 flops = 0
28
29 # Class encapsulating the Multigrid method for a given 2D Poisson equation instance
30 class Multigrid2D:
31     def __init__(self, maxN, borderFunction, valueFunction, niu1 = 4, niu2 = 4, omega = 1.95):
32         # BorderFunction and valueFunction pass the function values in the Poisson equation
33         self.borderFunction = borderFunction
34         self.valueFunction = valueFunction
35         self.omega = omega
36
37         # niu1 and niu2 are the number of relaxation steps per going-down and going-up parts of the v-cycle
38         self.niu1 = niu1
39         self.niu2 = niu2
40
41         # N is the discretization grid size
42         self.N = maxN
43
44         # Create and store grid for different levels of discretization
45         self.discrLevel = []
46         self.flops = 0
47         i = 0
48         while(maxN > 2):
49             assert(maxN % 2 == 0)
50             self.discrLevel.append(sed.SimpleEquationDiscretizer(maxN, self.borderFunction, self.valueFunction))
51             i += 1
52             maxN /= 2
53
54         # Helper function for grid level N
55         def getCoordinates(self, row, N):
56             return int(row / (N + 1)), row % (N + 1)
57
58         # Helper function for grid level N
59         def getRow(self, i, j, N):
60             return(i * (N + 1) + j)
61
62         # Restrict function from grid level finel to grid level coarseN using trivial injection
63         def restrict(self, r, finel, coarseN):
64             restr = []
65
66             for(i, elem) in enumerate(r):
67                 (x, y) = self.getCoordinates(i, finel)
68                 if(x % 2 == 0 and y % 2 == 0):
69                     restr.append(elem)
70
71             return restr
72
73         #
74         # Interpolate function from grid level coarseN to grid level finel with full weight stencil
75         def interpolate(self, r, finel, coarseN):
76             interp = []
77             flops = 0
78             for i in range((finel + 1) * (finel + 1)):
79                 (x, y) = self.getCoordinates(i, finel)
80                 if(x % 2 == 0 and y % 2 == 0):
81                     index = self.getRow(x / 2, y / 2, coarseN)
82
83

```

```

84
85         value = r[index]
86         flops += 1
87
88         elif(x % 2 == 1 and y % 2 == 0):
89             index1 = self.getRow((x - 1) / 2, y / 2, coarseN)
90             index2 = self.getRow((x + 1) / 2, y / 2, coarseN)
91             value = (r[index1] + r[index2]) / 2.0
92             flops += 2
93
94         elif(x % 2 == 0 and y % 2 == 1):
95             index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
96             index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
97             value = (r[index1] + r[index2]) / 2.0
98             flops += 2
99
100         else:
101             index1 = self.getRow((x - 1) / 2, (y - 1) / 2, coarseN)
102             index2 = self.getRow((x + 1) / 2, (y - 1) / 2, coarseN)
103             index3 = self.getRow((x - 1) / 2, (y + 1) / 2, coarseN)
104             index4 = self.getRow((x + 1) / 2, (y + 1) / 2, coarseN)
105             value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4.0
106             flops += 4
107
108         if(x == 0 or y == 0 or x == finel or y == finel):
109             value = 0
110
111         interp.append(value)
112
113         self.flops += flops
114         return interp
115
116         # Restrict function from grid level finel to grid level coarseN using
117         # the transpose action of interpolate (full weighting operator)
118         def restrictTransposeAction(self, r, finel, coarseN):
119             restr = []
120             flops = 0
121
122             for i in range((coarseN + 1) * (coarseN + 1)):
123                 (x, y) = self.getCoordinates(i, coarseN)
124                 (x, y) = (2 * x, 2 * y)
125                 newEntry = r[self.getRow(x, y, finel)]
126
127                 divideFactor = 1.0
128
129                 for (dX, dY) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
130                     newX = x + dX
131                     newY = y + dY
132                     if(0 <= newX and newX <= finel and 0 <= newY and newY <= finel):
133                         index = self.getRow(newX, newY, finel)
134                         newEntry += 0.5 * r[index]
135                         divideFactor += 0.5
136                         flops += 2
137
138                 for (dX, dY) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
139                     newX = x + dX
140                     newY = y + dY
141                     if(0 <= newX and newX <= finel and 0 <= newY and newY <= finel):
142                         index = self.getRow(newX, newY, finel)
143                         newEntry += 0.25 * r[index]
144                         divideFactor += 0.25
145                         flops += 2
146
147                 newEntry = 1.0 * newEntry / divideFactor
148                 if(divideFactor < 4.0):
149                     if(not(x == 0 or y == 0 or x == finel or y == finel)):
150                         print("Error code #1")
151
152                 if(x == 0 or y == 0 or x == finel or y == finel):
153                     newEntry = 0.0
154
155                 restr.append(newEntry)
156
157             self.flops += flops
158             return restr
159
160         # Wcycle iterates once a V-relaxation scheme of multigrid starting at level L for a grid size N
161         def wcycle(self, N, L, f, initSol = [], omega = 1.95):
162             level = L
163             discr = self.discrLevel[level]
164             fSize = len(f)
165
166             if(level == len(self.discrLevel) - 1):
167                 v = la.solve(discr.H, todense(f), f)
168                 return v

```

```

168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251

else:
    solver1 = sm.SolverMethods(
        iterationConstant = self.niul,
        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.getnmz() - 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.niul,
        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, niul = 4, niu2 = 4):
        self.borderFunction = borderFunction
        self.valueFunction = valueFunction

```

```

252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335

self.niul = niul
self.niu2 = niu2
self.discrLevel = []
self.N = maxN
self.flops = 0

i = 0
while (maxN > 2):
    assert (maxN % 2 == 0)
    self.discrLevel.append(sedID.EquationDiscretizerID(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 restrictID(self, r, fineN, coarseN):
    restr = []
    for (i, elem) in enumerate(r):
        if (i % 2 == 0):
            restr.append(elem)
    return restr

# Interpolation operator
def interpolateID(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.niul,
            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.getnmz() - len(v))

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

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

```

```

336 # Sanity checks verifying that the error on the border stays 0
337 if(not(coarseResidual[0] == 0)):
338     print("Error code #4")
339 if(not(coarseResidual[coarseN] == 0)):
340     print("Error code #5")
341
342 # Recursively V-cycle
343 coarseV = self.vcycle(coarseN, level + 1, coarseResidual)
344
345 # Sanity checks verifying that the error on the border stays 0
346 if(not(coarseV[0] == 0)):
347     print("Error code #6")
348 if(not(coarseV[coarseN] == 0)):
349     print("Error code #7")
350
351 # Interpolate the vector obtained recursively to a finer level
352 fineV = self.interpolateID(coarseV, N, coarseN)
353
354 # Correct the current approximation using the interpolation passed from a coarser level
355 w = np.add(v, fineV)
356 flops += len(v)
357
358 solver2 = sm.SolverMethods(
359     iterationConstant = self.niu2,
360     eqDiscretizer = discr,
361     b = f,
362     initSol = w,
363 )
364
365 # Apply extra smoothing to flatten out error components that might have been
366 # introduced in the process so far
367 v2, _, _, flops = solver2.SSORIterate(omega)
368 self.flops += flops
369 return v2
370
371 # Iterate t Vcycles in order to get a better approximate of the solution
372 def iterateVCycles(self, t):
373     initSol = []
374     N = self.N
375
376     for i in range((N + 1)):
377         x = i
378         if(x == 0 or x == N):
379             initSol.append(self.borderFunction(1.0 * x / N))
380         else:
381             initSol.append(0.0)
382
383     vErrors = []
384     discr = self.discretLevel[0]
385     f = np.copy(discr.valueVector2D)
386     normF = la.norm(f)
387
388     currSol = np.copy(initSol)
389
390     for i in range(t):
391         residual = np.subtract(f, discr.M.dot(currSol))
392         absErr = 1.0 * la.norm(residual) / (normF)
393         vErrors.append(math.log(absErr))
394
395     resSol = self.vcycle(N, 0, residual, np.zeros_like(currSol))
396     prevSol = np.copy(currSol)
397     currSol = np.add(currSol, resSol)
398
399     if(absErr < tol):
400         break
401     print(vErrors)
402     return currSol, vErrors
403
404
405 # Multigrid class implemented such that it can easily be adapted
406 # as a preconditioner; previous implementations regard it as
407 # being a solver on its own and put an emphasis on it
408 class MultigridAsPreconditioner:
409     def __init__(self, borderFunction, valueFunction, maxN, bVector = [], niu1 = 2, niu2 = 2, omega = 1.93):
410         self.borderFunction = borderFunction
411         self.valueFunction = valueFunction
412         self.niu1 = niu1
413         self.niu2 = niu2
414         self.maxN = maxN
415         self.bVector = bVector
416         self.discretLevel = []
417         self.omega = omega
418
419
420 self.flops = 0
421 i = 0
422 while(maxN >= 2):
423     assert(maxN % 2 == 0)
424     self.discretLevel.append(sed.SimpleEquationDiscretizer(maxN, self.borderFunction, self.valueFunction))
425     i += 1
426     maxN /= 2
427
428
429 def getCoordinates(self, row, N):
430     return int(row / (N + 1)), row % (N + 1)
431
432 def getRow(self, i, j, N):
433     return(i * (N + 1) + j)
434
435 def restrict(self, r, fineN, coarseN):
436     restr = []
437     for(i, elem) in enumerate(r):
438         (x, y) = self.getCoordinates(i, fineN)
439         if(x % 2 == 0 and y % 2 == 0):
440             restr.append(elem)
441     return restr
442
443 def interpolate(self, r, fineN, coarseN):
444     interp = []
445     flops = 0
446     for i in range((fineN + 1) * (fineN + 1)):
447         (x, y) = self.getCoordinates(i, fineN)
448         if(x % 2 == 0 and y % 2 == 0):
449             index = self.getRow(x / 2, y / 2, coarseN)
450             value = r[index]
451             flops += 1
452         elif(x % 2 == 1 and y % 2 == 0):
453             index1 = self.getRow(x - 1) / 2, y / 2, coarseN)
454             index2 = self.getRow(x + 1) / 2, y / 2, coarseN)
455             value = (r[index1] + r[index2]) / 2.0
456             flops += 2
457         elif(x % 2 == 0 and y % 2 == 1):
458             index1 = self.getRow(x / 2, (y - 1) / 2, coarseN)
459             index2 = self.getRow(x / 2, (y + 1) / 2, coarseN)
460             value = (r[index1] + r[index2]) / 2.0
461             flops += 2
462         else:
463             index1 = self.getRow(x - 1) / 2, (y - 1) / 2, coarseN)
464             index2 = self.getRow(x - 1) / 2, (y - 1) / 2, coarseN)
465             index3 = self.getRow(x - 1) / 2, (y + 1) / 2, coarseN)
466             index4 = self.getRow(x + 1) / 2, (y + 1) / 2, coarseN)
467             value = (r[index1] + r[index2] + r[index3] + r[index4]) / 4.0
468             flops += 4
469
470     if(x == 0 or y == 0 or x == fineN or y == fineN):
471         value = 0
472     interp.append(value)
473     self.flops += flops
474     return interp
475
476 def restrictTransposeAction(self, r, fineN, coarseN):
477     restr = []
478     flops = 0
479     for i in range((coarseN + 1) * (coarseN + 1)):
480         (x, y) = self.getCoordinates(i, coarseN)
481         newEntry = (2 * x / 2 * y)
482         newEntry = r[self.getRow(x, y, fineN)]
483         divideFactor = 1.0
484         for (dx, dy) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
485             newx = x + dx
486             newy = y + dy
487             if(0 <= newx and newx <= fineN and 0 <= newy and newy <= fineN):

```



```

504 index = self.getRow(newX, newY, fineN)
505 newEntry += 0.5 * r[index]
506 divideFactor += 0.5
507 flops += 2
508
509 for (dx, dy) in [(1, 1), (-1, 1), (-1, -1), (1, -1)]:
510     newX = x + dx
511     newY = y + dy
512     if 0 <= newX and newX <= fineN and 0 <= newY and newY <= fineN:
513         index = self.getRow(newX, newY, fineN)
514         newEntry += 0.25 * r[index]
515         divideFactor += 0.25
516         flops += 2
517
518 newEntry = 1.0 * newEntry / divideFactor
519 if divideFactor < 4.0:
520     if not (x == 0 or y == 0 or x == fineN or y == fineN):
521         print("Error!")
522
523 if (x == 0 or y == 0 or x == fineN or y == fineN):
524     newEntry = 0.0
525
526 restr.append(newEntry)
527
528 self.flops += flops
529 return restr
530
531 def vcycle(self, N, level, f, initSol = []):
532     discr = self.discrLevel[level]
533     fSize = len(f)
534     if fSize < 20:
535         v = la.solve(discr.M, todense(), f)
536         return v
537
538 omega = self.omega
539
540 solver1 = sm.SolverMethods(self.niu1, discr, f, initSol)
541 v, _, _, flops = solver1.SSORIterate(omega)
542 self.flops += flops
543
544 assert(N % 2 == 0)
545 coarseN = N / 2
546
547 Mv = discr.M.dot(v)
548 self.flops += (2*discr.M.getnnz() - len(v))
549
550 residual = np.subtract(np.array(f), Mv)
551 self.flops += len(f)
552
553 coarseResidual = self.restrictTransposeAction(residual, N, coarseN)
554
555 coarsev = self.vcycle(coarseN, level + 1, coarseResidual)
556 finev = self.interpolate(coarsev, N, coarseN)
557 v = np.add(v, finev)
558 self.flops += len(v)
559
560 solver2 = sm.SolverMethods(self.niu2, discr, f, v)
561 v2, _, _, flops = solver2.SSORIterate(omega)
562 self.flops += flops
563 return v2
564
565 def iterateVCycles(self, N, t):
566     self.flops = 0
567     initSol = []
568     vErrors = []
569     discr = sed.SimpleEquationDiscretizer(N, self.borderFunction, self.valueFunction)
570
571 if (self.bVector == []):
572     f = discr.valueVector2D
573 else:
574     f = self.bVector
575
576 for i in range(t):
577     currSol = self.vcycle(N, 0, f, initSol)
578     err = np.subtract(discr.M.dot(currSol), f)
579     absErr = np.linalg.norm(err) / np.linalg.norm(f)
580     vErrors.append(math.log(absErr))
581     if (absErr < tol):
582         break
583

```

```

588 initSol = currSol
589 return currSol, vErrors, self.flops
590
591 # MultiGridPrecondCG is the function that performs the MultiGrid preconditioned Conjugate Gradient method
592 def MultiGridPrecondCG(borderFunction, valueFunction, N, niu1 = 1, niu2 = 1, omega = 1.95):
593     avoidDivByZeroError = 0.0000000000001
594     errorDataMCG = []
595     totalFlops = 0
596     matrixDots = 0
597     vectorAddSub = 0
598     vectorDotVector = 0
599     mg = MultiGridAsPreconditioner(borderFunction, valueFunction, N, niu1 = niu1, niu2 = omega = omega)
600     M = mg.discrLevel[0].valueVector2D
601     x = np.zeros_like(f, dtype = np.float)
602     r = np.subtract(f, M.dot(x))
603     matrixDots += 1
604     vectorAddSub += 1
605     mg.bVector = np.copy(r)
606     rTilda, _, flops = mg.iterateVCycles(N, 1)
607     totalFlops += flops
608     rTilda = np.array(rTilda)
609     p = np.copy(rTilda)
610     convergence = False
611     while(not convergence):
612         solutionError = np.subtract(M.dot(x), f)
613         absErr = 1.0 * np.linalg.norm(solutionError) / np.linalg.norm(f)
614         errorDataMCG.append(math.log(absErr))
615         if (absErr < tol):
616             convergence = True
617             break
618
619 Mp = M.dot(p)
620 alpha_numerator = rTilda.dot(r)
621 alpha_denominator = p.dot(Mp)
622 vectorDotVector += 1
623 matrixDots += 1
624 if (alpha_denominator < avoidDivByZeroError):
625     convergence = True
626     break
627
628 alpha = 1.0 * alpha_numerator / alpha_denominator
629 totalFlops += 1
630 x = np.add(x, np.multiply(p, alpha))
631 vectorAddSub += 1
632 totalFlops += len(p)
633 newR = np.subtract(r, np.multiply(Mp, alpha))
634 totalFlops += len(Mp)
635 vectorAddSub += 1
636 mg.bVector = np.copy(newR)
637 newR_tilda, _, flops = mg.iterateVCycles(N, 1)
638 totalFlops += flops
639 newR_tilda = np.array(newR_tilda)
640 beta_numerator = newR_tilda.dot(newR)
641 beta_denominator = rTilda.dot(r)
642 vectorDotVector += 1
643 if (beta_denominator < avoidDivByZeroError):
644     convergence = True
645     break
646
647 beta = 1.0 * beta_numerator / beta_denominator
648 p = newR_tilda + np.multiply(p, beta)
649 totalFlops += 1
650 totalFlops += len(p)
651

```



```

1  # TimeEquationDiscretizer.py
2
3  # USED TO GENERATE HEAT EQUATION DISCRETIZATION
4
5  #
6
7  Import numpy as np
8  from scipy.sparse import *
9
10
11 11 COMPLETE_MATRIX = 'COMPLETE_MATRIX'
12 12 LOWER_BOUNDARY_MATRIX = 'LOWER_BOUNDARY_MATRIX'
13 13 STRICTLY_UPPER_MATRIX = 'STRICTLY_UPPER_MATRIX'
14 14 DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
15 15 REMAINDER_MATRIX = 'REMAINDER_MATRIX'
16
17 # Class encapsulating the discretization arising from
18 # a 2D Heat Equation, including methods to get the
19 # RHS vector for the Backward Euler method
20 #
21 21 class TimeEquationDiscretizer:
22
23     # Current value vector i.e. current value (at current time k * dt) of fix,v,t in du/dt - Laplace(u) = f
24     def __init__(self, M, T, borderTimeFunction, rhsHeatEquationFunction, initialHeatTimeFunction):
25         self.M = M
26         self.h = 1.0 / M
27
28         self.T = T
29         self.dt = 1.0 / T
30
31         self.borderTimeFunction = borderTimeFunction
32         self.rhsHeatEquationFunction = rhsHeatEquationFunction
33         self.initialHeatTimeFunction = initialHeatTimeFunction
34
35         self.rowList = []
36         self.colList = []
37         self.dataList = []
38
39         self.rowListDiagonal = []
40         self.colListDiagonal = []
41         self.dataListDiagonal = []
42
43         self.rowListRemainder = []
44         self.colListRemainder = []
45         self.dataListRemainder = []
46
47         self.rowListUpper = []
48         self.colListUpper = []
49         self.dataListUpper = []
50
51         self.rowListLower = []
52         self.colListLower = []
53         self.dataListLower = []
54
55         self.computeHeatTimeEquation()
56
57         self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1), (N + 1)) * (N + 1))
58         self.h = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.colListDiagonal), np.array(self.colListDiagonal))), shape = ((N + 1), (N + 1)) * (N + 1))
59         self.L = csr_matrix((np.array(self.dataListRemainder), (np.array(self.colListRemainder), np.array(self.colListRemainder))), shape = ((N + 1), (N + 1)) * (N + 1))
60         self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.colListUpper), np.array(self.colListUpper))), shape = ((N + 1), (N + 1)) * (N + 1))
61
62         # Lower and strictly upper matrices L, U with L + U = M
63         self.L = csr_matrix((np.array(self.dataListLower), (np.array(self.colListLower), np.array(self.colListLower))), shape = ((N + 1), (N + 1)) * (N + 1))
64         self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.colListUpper), np.array(self.colListUpper))), shape = ((N + 1), (N + 1)) * (N + 1))
65
66         print(self.M.todense())
67
68     # Helper functions for creating the complete, lower, upper and diagonal matrices
69     def __createMatrix(self, row, column, value):
70         if type == COMPLETE_MATRIX:
71             self.rowList.append(row)
72             self.colList.append(column)
73             self.dataList.append(value)
74         elif type == LOWER_BOUNDARY_MATRIX:
75             self.rowListLower.append(row)
76             self.colListLower.append(column)
77             self.dataListLower.append(value)
78         elif type == STRICTLY_UPPER_MATRIX:
79             self.rowListUpper.append(row)
80             self.colListUpper.append(column)
81             self.dataListUpper.append(value)
82         elif type == DIAGONAL_MATRIX:
83             self.colListDiagonal.append(row)
84             self.colListDiagonal.append(column)
85             self.dataListDiagonal.append(value)
86         elif type == REMAINDER_MATRIX:
87             self.colListRemainder.append(row)
88             self.colListRemainder.append(column)
89             self.dataListRemainder.append(value)
90
91     def addEntryToMatrices(self, row, column, value):
92         self.addEntryToCOMPLETE_MATRIX(row, column, value)
93         if row == column:
94             self.addEntryToDIAGONAL_MATRIX(row, column, value)
95             self.addEntryToLOWER_BOUNDARY_MATRIX(row, column, value)
96             if row > column:
97                 self.addEntryToSTRICTLY_UPPER_MATRIX(row, column, value)
98                 self.addEntryToREMAINDER_MATRIX(row, column, value)
99             else:
100                 self.addEntryToLOWER_BOUNDARY_MATRIX(row, column, value)
101
102     # Check if a(i,j) is on border
103     def isOnBorder(self, i, j):
104         if (i == 0 or j == 0 or i == (self.N) or j == (self.N)):
105             return True
106         else:
107             return False
108
109

```

```

1 # SimpleEquationDiscretizer.py
2
3 # USED TO GENERATE POISSON 2D EQUATION DISCRETIZATION
4
5 #
6
7 Import numpy as np
8 from scipy.sparse import *
9
10
11 COMPLETE_MATRIX = 'COMPLETE_MATRIX'
12 LOWER_MATRIX = 'LOWER_MATRIX'
13 STRICTLY_UPPER_MATRIX = 'STRICTLY_UPPER_MATRIX'
14 STRICTLY_LOWER_MATRIX = 'STRICTLY_LOWER_MATRIX'
15 DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
16 REMINDER_MATRIX = 'REMAINDER_MATRIX'
17
18 # Class encapsulating the sparse matrix
19 # arising from a 2D poisson model problem
20 class SimpleEquationDiscretizer:
21     def __init__(self, N, borderFunction, valueFunction):
22         self.N = N
23         self.h = 1.0 / N
24         self.borderFunction = borderFunction
25         self.valueFunction = valueFunction
26
27         self.rowList = []
28         self.colList = []
29         self.dataList = []
30
31         self.rowListDiagonal = []
32         self.colListDiagonal = []
33         self.dataListDiagonal = []
34
35         self.rowListRemainder = []
36         self.colListRemainder = []
37         self.dataListRemainder = []
38
39         self.rowListUpper = []
40         self.colListUpper = []
41         self.dataListUpper = []
42
43         self.rowListLower = []
44         self.colListLower = []
45         self.dataListLower = []
46
47         self.valueVector2D = []
48
49     def computeMatrixAndVector()
50         self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1), (N + 1)) * (N + 1))
51         self.b = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.colListDiagonal), np.array(self.colListDiagonal))), shape = ((N + 1) * (N + 1), (N + 1) * (N + 1)))
52         self.A = csr_matrix((np.array(self.dataListRemainder), (np.array(self.colListRemainder), np.array(self.colListRemainder))), shape = ((N + 1) * (N + 1), (N + 1) * (N + 1)))
53
54         # Lower and strictly upper matrices L, U with L + U = M
55         self.L = csr_matrix((np.array(self.dataListUpper), (np.array(self.colListUpper), np.array(self.colListUpper))), shape = ((N + 1) * (N + 1), (N + 1) * (N + 1)))
56         self.U = csr_matrix((np.array(self.dataListLower), (np.array(self.colListLower), np.array(self.colListLower))), shape = ((N + 1) * (N + 1), (N + 1) * (N + 1)))
57
58     # Helper functions for creating the complete, lower, upper and diagonal matrices
59     def addEntry(self, type, row, column, value):
60         if type == 'COMPLETE_MATRIX':
61             self.colList.append(column)
62             self.colList.append(row)
63             self.dataList.append(value)
64
65         elif type == 'LOWER_MATRIX':
66             self.colListLower.append(row)
67             self.colListLower.append(column)
68             self.dataListLower.append(value)
69
70         elif type == 'STRICTLY_UPPER_MATRIX':
71             self.colListUpper.append(row)
72             self.colListUpper.append(column)
73             self.dataListUpper.append(value)
74
75         elif type == 'DIAGONAL_MATRIX':
76             self.colListDiagonal.append(row)
77             self.colListDiagonal.append(column)
78             self.dataListDiagonal.append(value)
79
80         elif type == 'REMAINDER_MATRIX':
81             self.colListRemainder.append(row)
82             self.colListRemainder.append(column)
83             self.dataListRemainder.append(value)
84
85     def addEntryToMatrices(self, row, column, value):
86         self.addEntry('COMPLETE_MATRIX', row, column, value)
87         self.addEntry('DIAGONAL_MATRIX', row, column, value)
88         self.addEntry('LOWER_MATRIX', row, column, value)
89         self.addEntry('STRICTLY_UPPER_MATRIX', row, column, value)
90         self.addEntry('REMAINDER_MATRIX', row, column, value)
91
92         if (row < column):
93             self.addEntry('STRICTLY_UPPER_MATRIX', row, column, value)
94             self.addEntry('REMAINDER_MATRIX', row, column, value)
95
96         valueVector2D = []
97
98     # Check if a(i,j) is on border
99     def isOnBorder(self, i, j):
100         # print(i, j)
101         if (i == 0 or j == 0 or i == self.N or j == self.N):
102             return True
103         else:
104             return False
105
106     # Get the coordinates of the variable around which the row-th row is created
107     def getCoordinates(self, row):
108         return int(row / (self.N + 1)), row % (self.N + 1)
109

```

```

1  # EquationDiscretizerID.py
2
3  # USED TO GENERATE POISSON ID EQUATION DISCRETIZATION
4
5  #
6
7  Import numpy as np
8  from scipy.sparse import *
9
10
11 11 COMPLETE MATRIX = 'COMPLETE_MATRIX'
12 12 COMPLETE MATRIX = 'COMPLETE_MATRIX'
13 13 COMPLETE MATRIX = 'COMPLETE_MATRIX'
14 14 STRICTLY UPPER MATRIX = 'STRICTLY_UPPER_MATRIX'
15 15 DIAGONAL_MATRIX = 'DIAGONAL_MATRIX'
16 16 REMAINDER_MATRIX = 'REMAINDER_MATRIX'
17
18 # Class encapsulating the sparse matrix
19 # arising from discretizing a 1D model problem
20 class EquationDiscretizerID:
21
22     def __init__(self, N, borderFunction, valueFunction, printMatrix = False):
23         self.N = N
24         self.h = 1.0 / N
25         self.borderFunction = borderFunction
26         self.valueFunction = valueFunction
27
28         self.rowList = []
29         self.colList = []
30         self.dataList = []
31
32         self.rowListDiagonal = []
33         self.colListDiagonal = []
34         self.dataListDiagonal = []
35
36         self.rowListRemainder = []
37         self.colListRemainder = []
38         self.dataListRemainder = []
39
40         self.rowListUpper = []
41         self.colListUpper = []
42         self.dataListUpper = []
43
44         self.rowListLower = []
45         self.colListLower = []
46         self.dataListLower = []
47
48         self.valueVectorID = []
49
50     def computeMatrixAndVector()
51         self.M = csr_matrix((np.array(self.dataList), (np.array(self.rowList), np.array(self.colList))), shape = ((N + 1), (N + 1)))
52         self.b = csr_matrix((np.array(self.dataListDiagonal), (np.array(self.rowListDiagonal), np.array(self.colListDiagonal))), shape = (N + 1), (
53             N + 1))
54         self.A = csr_matrix((np.array(self.dataListRemainder), (np.array(self.rowListRemainder), np.array(self.colListRemainder))), shape = ((N + 1),
55             (N + 1))
56         # Lower and strictly upper matrices L, U with L + U = M
57         self.L = csr_matrix((np.array(self.dataListLower), (np.array(self.rowListLower), np.array(self.colListLower))), shape = ((N + 1), (N + 1)))
58         self.U = csr_matrix((np.array(self.dataListUpper), (np.array(self.rowListUpper), np.array(self.colListUpper))), shape = ((N + 1), (N + 1)))
59         if printMatrix:
60             print('Discretization matrix: ')
61             print(self.M.todense())
62             print(self.L.todense())
63             print(self.U.todense())
64
65     # Helper functions for creating the complete, lower, upper and diagonal matrices
66     def addEntryToMatrix(self, row, column, value):
67         if type == COMPLETE_MATRIX:
68             self.rowList.append(row)
69             self.colList.append(column)
70             self.dataList.append(value)
71         elif type == STRICTLY_UPPER_MATRIX:
72             self.rowListLower.append(row)
73             self.colListLower.append(column)
74             self.dataListLower.append(value)
75         elif type == STRICTLY_LOWER_MATRIX:
76             self.rowListUpper.append(row)
77             self.colListUpper.append(column)
78             self.dataListUpper.append(value)
79         elif type == DIAGONAL_MATRIX:
80             self.colListDiagonal.append(row)
81             self.colListDiagonal.append(column)
82             self.dataListDiagonal.append(value)
83         elif type == REMAINDER_MATRIX:
84             self.colListRemainder.append(row)
85             self.colListRemainder.append(column)
86             self.dataListRemainder.append(value)
87
88     def addEntryToMatrices(self, row, column, value):
89         self.addEntryToMatrix(self, row, column, value)
90         if row == column:
91             self.addEntryToMatrix(self, row, column, value)
92             self.addEntryToMatrix(self, row, column, value)
93             self.addEntryToMatrix(self, row, column, value)
94         if row > column:
95             self.addEntryToMatrix(self, row, column, value)
96             self.addEntryToMatrix(self, row, column, value)
97             self.addEntryToMatrix(self, row, column, value)
98         if row < column:
99             self.addEntryToMatrix(self, row, column, value)
100             self.addEntryToMatrix(self, row, column, value)
101             self.addEntryToMatrix(self, row, column, value)
102
103     valueVectorID = []
104
105     # Check if a(i) is on border
106     def isOnBorder(self, i):
107         if (i == 0 or i == self.N):
108             return True
109         else:
110             return False
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

1 # testTools.py
2
3 #
4 # USED FOR CARRYING TESTS ON ALL THE STRUCTURES AND METHODS DEFINED IN THE PROJECT
5 #
6
7 import numpy as np
8 from scipy.sparse import *
9 from math import pi
10 import numpy.linalg as la
11 import matplotlib.pyplot as plt
12 from mpl_toolkits.mplot3d import Axes3D
13
14 import SimpleEquationDiscretizer as sed
15 import EquationDiscretizer1D as sed1D
16 import SolverMethods as sm
17 import FunctionExamples as fe
18 import SimpleEquationDiscretizer as sed
19 import MGMethods as MG
20
21 # Default line colors used in some testing utilities
22 lineColor = ["red", "green", "blue", "brown", "black", "pink", "gray"]
23
24 # Empirically discovered approximation of the optimal SSOR parameter for the class of problems
25 # used in our tests
26 wSSOR20[1] = 1.503
27 wSSOR20[31] = 1.973
28 wSSOR20[16] = 1.720
29 wSSOR20[32] = 1.852
30 wSSOR20[64] = 1.923
31 wSSOR20[128] = 1.961
32
33
34 # Heat Equation solver testing utility
35 def testHeatEquationSolver():
36     # MG solve heat equation for all time steps (discretTest)
37     sol = MG.solveHeatEquationForAllTimeSteps(discretTest)
38     xSol = sol[i6]
39     return xSol
40
41 # Multigrid Preconditioned Conjugate Gradient solver testing utility
42 def testMCGG():
43     # niu1 and niu2 are the number of pre and post smoothing steps
44     niu1 = 1
45     niu2 = 1
46
47     # The parameter of the smoothing iteration used in MG as a preconditioner
48     omega = 1.92
49
50     print("Testing MCGG, niu1 = niu2 = ", niu1, ", omega = ", omega)
51
52     for N in [4, 8, 16, 32, 64, 128, 256]:
53         print(N)
54         xSolPrecond, errPrecond, errDataPrecond, fUps = MG.MultiGridPrecondCG(
55             fe.sin2BorderFunction,
56             fe.sin2ValueFunction,
57             N,
58             niu1= niu1,
59             niu2= niu2,
60             omega = omega,
61
62             plt.plot(errDataPrecond, label=str(N))
63
64             print("FLOPS: ", fUps)
65
66         plt.legend(loc='upper right')
67         plt.title("Testing MCGG, niu1 = niu2 = " + str(niu1) + ", omega = " + str(omega))
68         plt.show()
69
70 # PlotGraph is a helper function which interpolates a 3D graph of a function, given values at the grid knots
71 def plotGraph(N, valuesVector):
72     fig = plt.figure()
73     ax = fig.add_subplot(111, projection = '3d')
74     x = y = np.arange(0.0, 1.0 + h, h)
75
76     X, Y = np.meshgrid(x, y)
77     Z = np.reshape(valuesVector, (N+1, N+1))
78
79     ax.plot_wireframe(X, Y, Z)
80     ax.set_xlabel('X Axis')
81     ax.set_ylabel('Y Axis')
82     ax.set_zlabel('Z Label')
83
84     plt.show()
85
86 # Multigrid solver testing utility for 2D problem instances
87 def testMG2D():
88     N = 256
89     n = 4
90
91     i = 0
92
93     niu1 = 1
94     niu2 = 1
95
96     omega = 1.5
97     while(n < N):
98         n = n * 2
99         # Select omega = 2.0 / (1.0 + math.sin(math.pi / n)) to test the optimal SSOR parameter as a smoothing parameter
100         print("Testing MG2D, niu1, niu2 = ", niu1, ", niu2 = ", omega = ", omega)
101         print(n, ".")
102
103         mg = MG.MultiGrid2D(n, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = niu1, niu2 = niu2)
104         solMG, vErrors, fUps = mg.iterateCycles(1000)
105
106         print("Flops: ", fUps)
107
108         plt.plot(vErrors, label = str(n))
109
110         plt.title("Testing MG2D, niu1, niu2 = " + str(niu1) + ", omega = " + str(omega))
111         plt.legend(loc='upper right', prop={'size':16})
112         plt.show()
113
114 # Multigrid Preconditioned Conjugate Gradient solver testing utility for 2D problem instances for different smoothing parameters
115 def testMG2DvarK():
116     N = 128
117     listOmega = [1.89, 1.9, 1.91, 1.92, 1.93, 1.94, 1.95, 1.96]
118
119     for omega in listOmega:
120         mg = MG.MultiGrid2D(N, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = 2, niu2 = 2)
121         solMG, vErrors, fUps = mg.iterateCycles(1000)
122
123         print(N, " ", fUps)
124         plt.plot(vErrors, label = str(omega))
125
126         plt.legend(loc='upper right', prop={'size':16})
127         plt.show()
128
129 # Multigrid solver testing utility for 1D problem instances
130 def testMG1D():
131     N = 128
132     i = 0
133     while(n < N):
134         n = n * 2
135         mg = MG.MultiGrid1D(n, fe.sin1BorderFunction2, fe.sin1ValueFunction2)
136         solMG, vErrors = mg.iterateCycles(1000)
137         plt.plot(vErrors, label = str(n))
138
139         plt.legend(loc='upper right')
140         plt.show()
141
142 # Multigrid Conjugate Gradient solver testing utility for 1D problem instances for different smoothing parameters
143 def testDiffParamIterationsID():
144     N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))
145
146     optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
147     for omega in [optSOROmega]:
148         sineQuationDiscr = sed1D.EquationDiscretizer1D(N, fe.sin1BorderFunction, fe.sin1ValueFunction)
149         initSol = []
150         for i in range((N+1)):
151             if(i == 0 or i == N):
152                 initSol.append(sineQuationDiscr.borderFunction(1.0 * i / N))
153             else:
154                 initSol.append(0)
155
156         solver = sm.SolverMethods(700, sineQuationDiscr, initSol = initSol)
157         (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega, debugOn = False)
158         plt.plot(errorDataJacobi, label=str(omega))
159         print(len(errorDataJacobi))
160
161         plt.legend(loc = "upper right", prop={'size':15})
162         plt.show()
163
164 # SSOR solver testing utility for 2D problem instances for different smoothing parameters
165 def testDiffParamIterations():
166     N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))
167
168     optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
169     for omega in [optSOROmega, 1.503]:
170         sineQuationDiscr = sed.SimpleEquationDiscretizer(N, fe.sinBorderFunction, fe.sinValueFunction)
171         initSol = []
172         for i in range((N + 1) * (N + 1)):
173             (x, y) = sineQuationDiscr.getCoordinates(i)
174             if(x == 0 or y == 0 or x == N or y == N):
175                 initSol.append(sineQuationDiscr.borderFunction(1.0 * x / N, 1.0 * y / N))
176             else:
177                 initSol.append(0.0)
178
179         solver = sm.SolverMethods(300, sineQuationDiscr, initSol = initSol)
180         (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega)
181         plt.plot(errorDataJacobi, label=str(omega))
182
183         plt.legend(loc = "upper right", prop={'size':15})
184         plt.show()
185
186 # PlotGraph is a helper function which interpolates a 3D graph of a function, given values at the grid knots
187 def plotGraph(N, valuesVector):
188     fig = plt.figure()
189     ax = fig.add_subplot(111, projection = '3d')
190     x = y = np.arange(0.0, 1.0 + h, h)
191
192     X, Y = np.meshgrid(x, y)
193     Z = np.reshape(valuesVector, (N+1, N+1))
194
195     ax.plot_wireframe(X, Y, Z)
196     ax.set_xlabel('X Axis')
197     ax.set_ylabel('Y Axis')
198     ax.set_zlabel('Z Label')
199
200     plt.show()
201
202 # Multigrid solver testing utility for 2D problem instances
203 def testMG2D():
204     N = 256
205     n = 4
206
207     i = 0
208
209     niu1 = 1
210     niu2 = 1
211
212     omega = 1.5
213     while(n < N):
214         n = n * 2
215         # Select omega = 2.0 / (1.0 + math.sin(math.pi / n)) to test the optimal SSOR parameter as a smoothing parameter
216         print("Testing MG2D, niu1, niu2 = ", niu1, ", niu2 = ", omega = ", omega)
217         print(n, ".")
218
219         mg = MG.MultiGrid2D(n, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = niu1, niu2 = niu2)
220         solMG, vErrors, fUps = mg.iterateCycles(1000)
221
222         print("Flops: ", fUps)
223
224         plt.plot(vErrors, label = str(n))
225
226         plt.title("Testing MG2D, niu1, niu2 = " + str(niu1) + ", omega = " + str(omega))
227         plt.legend(loc='upper right', prop={'size':16})
228         plt.show()
229
230 # Multigrid Preconditioned Conjugate Gradient solver testing utility for 2D problem instances for different smoothing parameters
231 def testMG2DvarK():
232     N = 128
233     listOmega = [1.89, 1.9, 1.91, 1.92, 1.93, 1.94, 1.95, 1.96]
234
235     for omega in listOmega:
236         mg = MG.MultiGrid2D(N, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = 2, niu2 = 2)
237         solMG, vErrors, fUps = mg.iterateCycles(1000)
238
239         print(N, " ", fUps)
240         plt.plot(vErrors, label = str(omega))
241
242         plt.legend(loc='upper right', prop={'size':16})
243         plt.show()
244
245 # Multigrid solver testing utility for 1D problem instances
246 def testMG1D():
247     N = 128
248     i = 0
249     while(n < N):
250         n = n * 2
251         mg = MG.MultiGrid1D(n, fe.sin1BorderFunction2, fe.sin1ValueFunction2)
252         solMG, vErrors = mg.iterateCycles(1000)
253         plt.plot(vErrors, label = str(n))
254
255         plt.legend(loc='upper right')
256         plt.show()
257
258 # Multigrid Conjugate Gradient solver testing utility for 1D problem instances for different smoothing parameters
259 def testDiffParamIterationsID():
260     N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))
261
262     optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
263     for omega in [optSOROmega]:
264         sineQuationDiscr = sed1D.EquationDiscretizer1D(N, fe.sin1BorderFunction, fe.sin1ValueFunction)
265         initSol = []
266         for i in range((N+1)):
267             if(i == 0 or i == N):
268                 initSol.append(sineQuationDiscr.borderFunction(1.0 * i / N))
269             else:
270                 initSol.append(0)
271
272         solver = sm.SolverMethods(700, sineQuationDiscr, initSol = initSol)
273         (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega, debugOn = False)
274         plt.plot(errorDataJacobi, label=str(omega))
275         print(len(errorDataJacobi))
276
277         plt.legend(loc = "upper right", prop={'size':15})
278         plt.show()
279
280 # SSOR solver testing utility for 2D problem instances for different smoothing parameters
281 def testDiffParamIterations():
282     N = int(input("Enter inverse of coordinates sample rate for the coarser grid\n"))
283
284     optSOROmega = 2.0/(1.0 + math.sin(math.pi / N))
285     for omega in [optSOROmega, 1.503]:
286         sineQuationDiscr = sed.SimpleEquationDiscretizer(N, fe.sinBorderFunction, fe.sinValueFunction)
287         initSol = []
288         for i in range((N + 1) * (N + 1)):
289             (x, y) = sineQuationDiscr.getCoordinates(i)
290             if(x == 0 or y == 0 or x == N or y == N):
291                 initSol.append(sineQuationDiscr.borderFunction(1.0 * x / N, 1.0 * y / N))
292             else:
293                 initSol.append(0.0)
294
295         solver = sm.SolverMethods(300, sineQuationDiscr, initSol = initSol)
296         (xFine, absErr, errorDataJacobi, rFine) = solver.SSORIterate(omega)
297         plt.plot(errorDataJacobi, label=str(omega))
298
299         plt.legend(loc = "upper right", prop={'size':15})
300         plt.show()
301
302 # PlotGraph is a helper function which interpolates a 3D graph of a function, given values at the grid knots
303 def plotGraph(N, valuesVector):
304     fig = plt.figure()
305     ax = fig.add_subplot(111, projection = '3d')
306     x = y = np.arange(0.0, 1.0 + h, h)
307
308     X, Y = np.meshgrid(x, y)
309     Z = np.reshape(valuesVector, (N+1, N+1))
310
311     ax.plot_wireframe(X, Y, Z)
312     ax.set_xlabel('X Axis')
313     ax.set_ylabel('Y Axis')
314     ax.set_zlabel('Z Label')
315
316     plt.show()
317
318 # Multigrid solver testing utility for 2D problem instances
319 def testMG2D():
320     N = 256
321     n = 4
322
323     i = 0
324
325     niu1 = 1
326     niu2 = 1
327
328     omega = 1.5
329     while(n < N):
330         n = n * 2
331         # Select omega = 2.0 / (1.0 + math.sin(math.pi / n)) to test the optimal SSOR parameter as a smoothing parameter
332         print("Testing MG2D, niu1, niu2 = ", niu1, ", niu2 = ", omega = ", omega)
333         print(n, ".")
334
335         mg = MG.MultiGrid2D(n, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = niu1, niu2 = niu2)
336         solMG, vErrors, fUps = mg.iterateCycles(1000)
337
338         print("Flops: ", fUps)
339
340         plt.plot(vErrors, label = str(n))
341
342         plt.title("Testing MG2D, niu1, niu2 = " + str(niu1) + ", omega = " + str(omega))
343         plt.legend(loc='upper right', prop={'size':16})
344         plt.show()
345
346 # Multigrid Preconditioned Conjugate Gradient solver testing utility for 2D problem instances for different smoothing parameters
347 def testMG2DvarK():
348     N = 128
349     listOmega = [1.89, 1.9, 1.91, 1.92, 1.93, 1.94, 1.95, 1.96]
350
351     for omega in listOmega:
352         mg = MG.MultiGrid2D(N, fe.sin2BorderFunction, fe.sin2ValueFunction, omega = omega, niu1 = 2, niu2 = 2)
353         solMG, vErrors, fUps = mg.iterateCycles(1000)
354
355         print(N, " ", fUps)
356         plt.plot(vErrors, label = str(omega))
357
358         plt.legend(loc='upper right', prop={'size':16})
359         plt.show()
360
361 # Multigrid solver testing utility for 1D problem instances
362 def testMG1D():
363     N = 128
364     i = 0
365     while(n < N):
366         n = n * 2
367         mg = MG.MultiGrid1D(n, fe.sin1BorderFunction2, fe.sin1ValueFunction2)
368         solMG, vErrors = mg.iterateCycles(1000)
369         plt.plot(vErrors, label = str(n))
370
371         plt.legend(loc='upper right')
372         plt.show()
373
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

192 plt.legend(loc = "upper right", prop={'size':15})
193 plt.show()
194
195 # Conjugate Gradient testing utility
196 def testConjugateGradient():
197     N = 128
198     index = 0
199     solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
200     plt.plot(errorData, label = "Testing Conjugate Gradient")
201     index += 1
202     while(n < N):
203         n = 2 * n
204         print(n)
205         sineQuationDiscr = sed.SimpleEquationDiscrizer(n, fe.sinBorderFunction, fe.sinValueFunction)
206         initSol = []
207         actualSol = fe.actualSinSolution(n)
208         for i in range((n + 1) * (n + 1)):
209             (x, y) = sineQuationDiscr.getCoordinates(i)
210             if(x == 0 or y == 0 or x == n or y == n):
211                 initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
212             else:
213                 initSol.append(0.0)
214
215         solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol, actualSol = actualSol)
216         (x, absErr, errorData) = solver.ConjugateGradientsHS()
217         plt.plot(errorData, label = "N = " + str(n))
218         index = index + 1
219
220         plt.legend(loc = "upper right", prop={'size':16})
221         plt.show()
222
223 # Steepest Descent testing utility
224 def testSteepestDescent():
225     N = 128
226     index = 0
227     while(n < N):
228         n = 2 * n
229         print(n)
230         sineQuationDiscr = sed.SimpleEquationDiscrizer(n, fe.sinBorderFunction, fe.sinValueFunction)
231         initSol = []
232         for i in range((n + 1) * (n + 1)):
233             (x, y) = sineQuationDiscr.getCoordinates(i)
234             if(x == 0 or y == 0 or x == n or y == n):
235                 initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
236             else:
237                 initSol.append(0.0)
238
239         actualSol = fe.actualSinSolution(n)
240         solver = sm.SolverMethods(2000000, sinEquationDiscr, initSol = initSol, actualSol = [])
241         (x, absErr, errorData) = solver.SteepestDescent()
242         plt.plot(errorData, label = "N = " + str(n))
243         index = index + 1
244
245         plt.legend(loc = "upper right", prop={'size':16})
246         plt.show()
247
248 # Jacobi solver testing utility on a 1D model problem
249 def testJacobiSmoothingID():
250     N = 32
251     h = 1.0 / N
252     x = np.arange(0.0, 1.0 + h, h)
253     k = 3.0
254     initSol = np.sin(math.pi * 8 * x) + np.sin(math.pi * 5 * x) + np.sin(math.pi * 3 * x)
255     plt.plot(x, initSol)
256
257     sineQuationDiscr = sed.ID.EquationDiscrizerID(N, fe.zeroID, fe.zeroID)
258     solver = sm.SolverMethods(20, sinEquationDiscr, initSol = initSol)
259     (y, absErr, errorData, r) = solver.JacobiIterate(1.0)
260
261     plt.plot(x, y)
262     plt.show()
263
264 # Jacobi solver testing utility
265 def testJacobi():
266     N = 128
267     index = 0
268     while(n < N):
269         n = 2 * n
270         print("Testing Jacobi")
271         sineQuationDiscr = sed.SimpleEquationDiscrizer(n, fe.sinBorderFunction, fe.sinValueFunction)
272         initSol = []
273         for i in range((n + 1) * (n + 1)):

```

```

288     (x, y) = sineQuationDiscr.getCoordinates(i)
289     if(x == 0 or y == 0 or x == n or y == n):
290         initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
291     else:
292         initSol.append(0.0)
293
294     solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
295     (x, absErr, errorData, r) = solver.JacobiIterate()
296     plt.plot(errorData, label = str(n) + ", Jacobi")
297     index += 1
298
299     plt.legend(loc = "upper right", prop={'size':16})
300     plt.show()
301
302 # Gauss Seidel solver testing utility
303 def testGaussSeidel():
304     N = 128
305     n = 4
306     index = 0
307     while(n < N):
308         n = 2 * n
309         print(n, ':')
310         sineQuationDiscr = sed.SimpleEquationDiscrizer(n, fe.sinBorderFunction, fe.sinValueFunction)
311         initSol = []
312         for i in range((n + 1) * (n + 1)):
313             (x, y) = sineQuationDiscr.getCoordinates(i)
314             if(x == 0 or y == 0 or x == n or y == n):
315                 initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
316             else:
317                 initSol.append(0.0)
318
319         solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
320         wOpt = 2.0 / (1.0 + math.sin(math.pi / n))
321         (x, absErr, errorData, r) = solver.GaussSeidelIterate(wOpt)
322         plt.plot(errorData, label = str(n) + ",SOR", color=LineColor[index])
323         index += 1
324
325         plt.show()
326
327 # SSOR solver testing utility
328 def testSSOR():
329     N = 256
330     n = 4
331     index = 0
332     while(n < N):
333         n = 2 * n
334         print("SSOR TEST")
335         sineQuationDiscr = sed.SimpleEquationDiscrizer(n, fe.sinBorderFunction, fe.sinValueFunction)
336         initSol = []
337         for i in range((n + 1) * (n + 1)):
338             (x, y) = sineQuationDiscr.getCoordinates(i)
339             if(x == 0 or y == 0 or x == n or y == n):
340                 initSol.append(sinEquationDiscr.borderFunction(1.0 * x / n, 1.0 * y / n))
341             else:
342                 initSol.append(0.0)
343
344         solver = sm.SolverMethods(2000, sinEquationDiscr, initSol = initSol)
345         wOpt = wSSOR2D(n)
346         (x, absErr, errorData, r) = solver.SSORIterate(wOpt)
347         index += 1
348
349         plt.legend(loc = "upper right", prop={'size':15})
350         plt.show()
351
352 # Helper function which prints the resulting discretization matrix in dense format
353 # Note: only use this for small values, otherwise making a dense matrix from our
354 # sparse format becomes too expensive
355 def printDiscretization():
356     f1=open('./testfile', 'w')
357     eqDiscr = sed.SimpleEquationDiscrizer(3, fe.sinBorderFunction, fe.sinValueFunction)
358     f1.write(str(eqDiscr.M.todense()))
359     f1.close()
360
361 # Helper function to plot the exact solution of a 1D model problem
362 def plotExactSolID():
363     N = 2
364     t = np.arange(0.0, 1.0, 0.01)
365     k = 5.0
366     index = 0
367     while(n < N):
368         n = 2 * n
369         h = 1.0 / n

```

```

384 print(n, ' :')
385 x = np.arange(0,0.0, 1.0 + h, h)
386 sinEquationDiscr = sed1DEquationDiscrizer1D(n, fe.sin1DBorderFunction2, fe.sin1DValueFunction2)
387
388 initSol = []
389
390 for i in range(n + 1):
391     if i == 0 or i == n:
392         initSol.append(sinEquationDiscr.borderFunction(1.0 + i / n))
393     else:
394         initSol.append(0)
395     M = sinEquationDiscr.Mtodense()
396     v = sinEquationDiscr.valueVector()
397     exactSol = np.linalg.solve(M, v)
398     index = index + 1
399     plt.subplot('22-'+str(index))
400     plt.plot(x,exactSol, 'bo')
401     if index == 1:
402         plt.plot(x,exactSol, label = "Linear interpolation of  $\sin(x)$ ")
403     elif index == 2:
404         plt.plot(x,exactSol, label = "Linear interpolation of  $\sin(x)$ ")
405     elif index == 3:
406         plt.plot(x,exactSol, label = "Linear interpolation of  $\sin(x)$ ")
407     elif index == 4:
408         plt.plot(x,exactSol, label = "Linear interpolation of  $\sin(x)$ ")
409         plt.plot(t, s, label = "Exact continuous solution:  $\sin(x)$ ")
410     plt.legend(loc = "lower right", prop={'size':12})
411     plt.title('N = '+str(n))
412     plt.show()
413
414 # Backward Euler method for solving the Heat Equation (which includes the time variable)
415 def solveHeatEquationForAllTimeSteps(discr):
416     valueVector = discr.computeVectorAtTimeStep(1, solHeat)
417     solver = sm.SolverMethods(1000, discr, valueVector)
418     sol = [solHeat]
419     for k in range(1, discr.T + 1):
420         t = k * discr.dt
421         valueVector = discr.computeVectorAtTimeStep(k, solHeat)
422         solver.b = valueVector
423         solver = sm.SolverMethods(1000, solver.ConjugateGradientsHS(),
424                                     sol.append(solHeat))
425         print(err)
426     plotGraph(discr.N, solHeat)
427     return sol
428
429 # Heat Equation solver testing utility
430 def testHeatEquationSolver():
431     discr = sed1DEquationDiscrizer(
432         n = 32,
433         T = 6,
434         borderTimeFunction = fe.heats1DBorderFunction,
435         rhsHeatEquationFunction = fe.heatRhsFunction,
436         initialHeatTimeFunction = fe.heatInitialFunction,
437     )
438     solveHeatEquationForAllTimeSteps(discr)

```



```

83 def sinIDValueFunction2(x):
84     k = 3.0
85     value = - math.sin(math.pi * k * x) * k * k * math.pi * math.pi
86     return value
87
88 def sinIDBorderFunction2(x):
89     k = 3.0
90     value = math.sin(math.pi * k * x)
91     return value
92
93
94 def zeroID(x):
95     return 0

```

```

1 # FunctionExamples.py
2
3 #
4 # USED FOR SAMPLE MODEL PROBLEMS FUNCTIONS
5 #
6
7 import math
8
9 # Function examples for a simple equation discretizer
10
11 # Value of the border function on values x,y
12 def sinBorderFunction(x, y):
13     # Assert (x,y) is on border
14     value = 1.0 * math.sin(x) * math.sin(y)
15     return value
16
17 def actualSinSolution(N):
18     actualSolution = []
19     for i in range(N + 1):
20         for j in range(N + 1):
21             actualSolution.append(math.sin((1.0) * i / N) * math.sin((1.0) * j / N))
22     return actualSolution
23
24
25 # RHS value of the differential equation at points x, y
26 def sinValueFunction(x, y):
27     value = - 2.0 * math.sin(x) * math.sin(y)
28     return value
29
30
31 def borderFunction1(x, y):
32     value = 1.0 * (x * x * x + y * y * y + x * y + 1.0)
33     return value
34
35 def laplaceValueFunction1(x, y):
36     value = 0.0 * x + 0.0 * y
37     return value
38
39 def sin2BorderFunction(x, y):
40     # Assert (x,y) is on border
41     k = 2.0
42     j = 5.0
43     value = 1.0 * math.sin(math.pi * k * x) * math.sin(math.pi * j * y)
44     return value
45
46
47 # RHS value of the differential equation at points x, y
48 def sin2ValueFunction(x, y):
49     k = 2.0
50     j = 5.0
51     value = - 1.0 * math.sin(math.pi * k * x) * math.sin(math.pi * j * y) * math.pi * math.pi * (k * k + j * j)
52     return value
53
54
55 # Function examples for a time equation discretizer (heat equation)
56
57 def heatSinBorderFunction(x, y, t):
58     value = 1.0 * math.sin(x) * math.sin(y)
59     value = 0.0
60     return value
61
62 def heatRhsFunction(x, y, t):
63     value = -2.0 * math.sin(x) * math.sin(y)
64     value = 0.0
65     return value
66
67 def heatInitialFunction(x, y, t):
68     # Assert t == 0
69     value = math.sin(x) * math.sin(y)
70     return value
71
72
73
74 def sinIDValueFunction(x):
75     value = - math.sin(x)
76     return value
77
78 def sinIDBorderFunction(x):
79     value = math.sin(x)
80     return value
81
82

```

```

1 # possible.py
2
3 # USED FOR AUXILIARY PLOTS IN THE THESIS
4 #
5
6
7 from mpl_toolkits import mplot3d
8 import numpy as np
9 import matplotlib.pyplot as plt
10 import matplotlib.pyplot as plt
11 import math
12 matplotlib.rcParams['text.usetex'] = True
13 matplotlib.rcParams['text.latex.unicode'] = True
14 tol = 0.00001
15
16 A = np.array([[-2.0, 1.0], [1.0, 3.0]]).dtype = np.float
17 b = np.array([5.0, 5.0]).dtype = np.float
18
19 def f(x, y):
20     return (1 * x ** 2 + 1.5 * y ** 2 + 1 * x*y - 5 * x - 5 * y)
21
22 # ax = plt.axes(projection='3d')
23 # plt.axis('off')
24 def plotf():
25     x = np.linspace(0.8, 2.5, 20)
26     y = np.linspace(0.8, 2, 20)
27
28     X, Y = np.meshgrid(x, y)
29     Z = f(X, Y)
30     fig = plt.figure()
31     ax = plot_surface(X, Y, Z, rstride=1, cstride=1,
32                      cmap='viridis', edgecolor='none')
33
34     # ax.plot([0.1], [0.1], [0.1])
35     # plt.show()
36
37
38 xs = []
39 def SteepestDescent(M, b, iterationConstant = 100, plotFirstIterations = False):
40     # iterationConstant = 100
41     plt.xlim(-2, 3)
42     plt.ylim(-2, 2)
43     avoidDivByZeroError = 0.000000000000000001
44     r = np.zeros_like(b)
45     r = np.subtract(b, M.dot(x))
46     errorDataSteepestDescent = []
47     iterationConstant = iterationConstant
48     for i in range(iterationConstant):
49         xs.append(x)
50
51         err = np.subtract(M.dot(x), b)
52         absErr = np.linalg.norm(err) / np.linalg.norm(b)
53         errorDataSteepestDescent.append(math.log(absErr))
54
55         alpha_numerator = r.dot(r)
56         alpha_denominator = r.dot(M.dot(r))
57         if(alpha_denominator < avoidDivByZeroError):
58             alpha = alpha_numerator / alpha_denominator
59         xOld = np.copy(x)
60         x = np.add(x, np.dot(r, alpha))
61
62     # Used for plotting first iterations of Steepest Descent
63     fSize = 26
64     fSize = 26
65     fSize = 26
66     fSize = 26
67     fSize = 26
68     fSize = 26
69     fSize = 26
70     fSize = 26
71     fSize = 26
72     fSize = 26
73     fSize = 26
74     fSize = 26
75     fSize = 26
76     fSize = 26
77     fSize = 26
78     fSize = 26
79     fSize = 26
80     fSize = 26
81     fSize = 26
82     fSize = 26
83     fSize = 26
84     fSize = 26
85     fSize = 26
86     fSize = 26
87     fSize = 26
88     fSize = 26
89     fSize = 26
90     fSize = 26
91     fSize = 26
92     fSize = 26
93     fSize = 26
94     fSize = 26
95     fSize = 26
96     fSize = 26
97     fSize = 26
98     fSize = 26
99     fSize = 26
100     fSize = 26
101     fSize = 26
102     fSize = 26
103     fSize = 26
104     fSize = 26
105     fSize = 26
106     fSize = 26
107     fSize = 26
108     fSize = 26
109     fSize = 26
110     fSize = 26
111     fSize = 26
112     fSize = 26
113     fSize = 26
114     fSize = 26
115     fSize = 26
116     fSize = 26
117     fSize = 26
118     fSize = 26
119     fSize = 26
120     fSize = 26
121     fSize = 26
122     fSize = 26
123     fSize = 26
124     fSize = 26
125     fSize = 26
126     fSize = 26
127     fSize = 26
128     fSize = 26
129     fSize = 26
130     fSize = 26
131     fSize = 26
132     fSize = 26
133     fSize = 26
134     fSize = 26
135     fSize = 26
136     fSize = 26
137     fSize = 26
138     fSize = 26
139     fSize = 26
140     fSize = 26
141     fSize = 26
142     fSize = 26
143     fSize = 26
144     fSize = 26
145     fSize = 26
146     fSize = 26
147     fSize = 26
148     fSize = 26
149     fSize = 26
150     fSize = 26
151     fSize = 26
152     fSize = 26
153     fSize = 26
154     fSize = 26
155     fSize = 26
156     fSize = 26
157     fSize = 26
158     fSize = 26
159     fSize = 26
160     fSize = 26
161     fSize = 26
162     fSize = 26
163     fSize = 26
164     fSize = 26
165     fSize = 26
166     fSize = 26
167     fSize = 26
168     fSize = 26
169     fSize = 26
170     fSize = 26
171     fSize = 26
172     fSize = 26
173     fSize = 26
174     fSize = 26
175     fSize = 26
176     fSize = 26
177     fSize = 26
178     fSize = 26
179     fSize = 26
180     fSize = 26
181     fSize = 26
182     fSize = 26
183     fSize = 26
184     fSize = 26
185     fSize = 26
186     fSize = 26
187     fSize = 26
188     fSize = 26
189     fSize = 26
190     fSize = 26
191     fSize = 26
192     fSize = 26
193     fSize = 26
194     fSize = 26
195     fSize = 26
196     fSize = 26
197     fSize = 26
198     fSize = 26
199     fSize = 26
200     fSize = 26
201     fSize = 26
202     fSize = 26
203     fSize = 26
204     fSize = 26
205     fSize = 26
206     fSize = 26
207     fSize = 26
208     fSize = 26
209     fSize = 26
210     fSize = 26
211     fSize = 26
212     fSize = 26
213     fSize = 26
214     fSize = 26
215     fSize = 26
216     fSize = 26
217     fSize = 26
218     fSize = 26
219     fSize = 26
220     fSize = 26
221     fSize = 26
222     fSize = 26
223     fSize = 26
224     fSize = 26
225     fSize = 26
226     fSize = 26
227     fSize = 26
228     fSize = 26
229     fSize = 26
230     fSize = 26
231     fSize = 26
232     fSize = 26
233     fSize = 26
234     fSize = 26
235     fSize = 26
236     fSize = 26
237     fSize = 26
238     fSize = 26
239     fSize = 26
240     fSize = 26
241     fSize = 26
242     fSize = 26
243     fSize = 26
244     fSize = 26
245     fSize = 26
246     fSize = 26
247     fSize = 26
248     fSize = 26
249     fSize = 26
250     fSize = 26
251     fSize = 26
252     fSize = 26
253     fSize = 26
254     fSize = 26
255     fSize = 26
256     fSize = 26
257     fSize = 26
258     fSize = 26
259     fSize = 26
260     fSize = 26
261     fSize = 26
262     fSize = 26
263     fSize = 26
264     fSize = 26
265     fSize = 26
266     fSize = 26
267     fSize = 26
268     fSize = 26
269     fSize = 26
270     fSize = 26
271     fSize = 26
272     fSize = 26
273     fSize = 26
274     fSize = 26
275     fSize = 26
276     fSize = 26
277     fSize = 26
278     fSize = 26
279     fSize = 26
280     fSize = 26
281     fSize = 26
282     fSize = 26
283     fSize = 26
284     fSize = 26
285     fSize = 26
286     fSize = 26
287     fSize = 26
288     fSize = 26
289     fSize = 26
290     fSize = 26
291     fSize = 26
292     fSize = 26
293     fSize = 26
294     fSize = 26
295     fSize = 26
296     fSize = 26
297     fSize = 26
298     fSize = 26
299     fSize = 26
300     fSize = 26
301     fSize = 26
302     fSize = 26
303     fSize = 26
304     fSize = 26
305     fSize = 26
306     fSize = 26
307     fSize = 26
308     fSize = 26
309     fSize = 26
310     fSize = 26
311     fSize = 26
312     fSize = 26
313     fSize = 26
314     fSize = 26
315     fSize = 26
316     fSize = 26
317     fSize = 26
318     fSize = 26
319     fSize = 26
320     fSize = 26
321     fSize = 26
322     fSize = 26
323     fSize = 26
324     fSize = 26
325     fSize = 26
326     fSize = 26
327     fSize = 26
328     fSize = 26
329     fSize = 26
330     fSize = 26
331     fSize = 26
332     fSize = 26
333     fSize = 26
334     fSize = 26
335     fSize = 26
336     fSize = 26
337     fSize = 26
338     fSize = 26
339     fSize = 26
340     fSize = 26
341     fSize = 26
342     fSize = 26
343     fSize = 26
344     fSize = 26
345     fSize = 26
346     fSize = 26
347     fSize = 26
348     fSize = 26
349     fSize = 26
350     fSize = 26
351     fSize = 26
352     fSize = 26
353     fSize = 26
354     fSize = 26
355     fSize = 26
356     fSize = 26
357     fSize = 26
358     fSize = 26
359     fSize = 26
360     fSize = 26
361     fSize = 26
362     fSize = 26
363     fSize = 26
364     fSize = 26
365     fSize = 26
366     fSize = 26
367     fSize = 26
368     fSize = 26
369     fSize = 26
370     fSize = 26
371     fSize = 26
372     fSize = 26
373     fSize = 26
374     fSize = 26
375     fSize = 26
376     fSize = 26
377     fSize = 26
378     fSize = 26
379     fSize = 26
380     fSize = 26
381     fSize = 26
382     fSize = 26
383     fSize = 26
384     fSize = 26
385     fSize = 26
386     fSize = 26
387     fSize = 26
388     fSize = 26
389     fSize = 26
390     fSize = 26
391     fSize = 26
392     fSize = 26
393     fSize = 26
394     fSize = 26
395     fSize = 26
396     fSize = 26
397     fSize = 26
398     fSize = 26
399     fSize = 26
400     fSize = 26
401     fSize = 26
402     fSize = 26
403     fSize = 26
404     fSize = 26
405     fSize = 26
406     fSize = 26
407     fSize = 26
408     fSize = 26
409     fSize = 26
410     fSize = 26
411     fSize = 26
412     fSize = 26
413     fSize = 26
414     fSize = 26
415     fSize = 26
416     fSize = 26
417     fSize = 26
418     fSize = 26
419     fSize = 26
420     fSize = 26
421     fSize = 26
422     fSize = 26
423     fSize = 26
424     fSize = 26
425     fSize = 26
426     fSize = 26
427     fSize = 26
428     fSize = 26
429     fSize = 26
430     fSize = 26
431     fSize = 26
432     fSize = 26
433     fSize = 26
434     fSize = 26
435     fSize = 26
436     fSize = 26
437     fSize = 26
438     fSize = 26
439     fSize = 26
440     fSize = 26
441     fSize = 26
442     fSize = 26
443     fSize = 26
444     fSize = 26
445     fSize = 26
446     fSize = 26
447     fSize = 26
448     fSize = 26
449     fSize = 26
450     fSize = 26
451     fSize = 26
452     fSize = 26
453     fSize = 26
454     fSize = 26
455     fSize = 26
456     fSize = 26
457     fSize = 26
458     fSize = 26
459     fSize = 26
460     fSize = 26
461     fSize = 26
462     fSize = 26
463     fSize = 26
464     fSize = 26
465     fSize = 26
466     fSize = 26
467     fSize = 26
468     fSize = 26
469     fSize = 26
470     fSize = 26
471     fSize = 26
472     fSize = 26
473     fSize = 26
474     fSize = 26
475     fSize = 26
476     fSize = 26
477     fSize = 26
478     fSize = 26
479     fSize = 26
480     fSize = 26
481     fSize = 26
482     fSize = 26
483     fSize = 26
484     fSize = 26
485     fSize = 26
486     fSize = 26
487     fSize = 26
488     fSize = 26
489     fSize = 26
490     fSize = 26
491     fSize = 26
492     fSize = 26
493     fSize = 26
494     fSize = 26
495     fSize = 26
496     fSize = 26
497     fSize = 26
498     fSize = 26
499     fSize = 26
500     fSize = 26
501     fSize = 26
502     fSize = 26
503     fSize = 26
504     fSize = 26
505     fSize = 26
506     fSize = 26
507     fSize = 26
508     fSize = 26
509     fSize = 26
510     fSize = 26
511     fSize = 26
512     fSize = 26
513     fSize = 26
514     fSize = 26
515     fSize = 26
516     fSize = 26
517     fSize = 26
518     fSize = 26
519     fSize = 26
520     fSize = 26
521     fSize = 26
522     fSize = 26
523     fSize = 26
524     fSize = 26
525     fSize = 26
526     fSize = 26
527     fSize = 26
528     fSize = 26
529     fSize = 26
530     fSize = 26
531     fSize = 26
532     fSize = 26
533     fSize = 26
534     fSize = 26
535     fSize = 26
536     fSize = 26
537     fSize = 26
538     fSize = 26
539     fSize = 26
540     fSize = 26
541     fSize = 26
542     fSize = 26
543     fSize = 26
544     fSize = 26
545     fSize = 26
546     fSize = 26
547     fSize = 26
548     fSize = 26
549     fSize = 26
550     fSize = 26
551     fSize = 26
552     fSize = 26
553     fSize = 26
554     fSize = 26
555     fSize = 26
556     fSize = 26
557     fSize = 26
558     fSize = 26
559     fSize = 26
560     fSize = 26
561     fSize = 26
562     fSize = 26
563     fSize = 26
564     fSize = 26
565     fSize = 26
566     fSize = 26
567     fSize = 26
568     fSize = 26
569     fSize = 26
570     fSize = 26
571     fSize = 26
572     fSize = 26
573     fSize = 26
574     fSize = 26
575     fSize = 26
576     fSize = 26
577     fSize = 26
578     fSize = 26
579     fSize = 26
580     fSize = 26
581     fSize = 26
582     fSize = 26
583     fSize = 26
584     fSize = 26
585     fSize = 26
586     fSize = 26
587     fSize = 26
588     fSize = 26
589     fSize = 26
590     fSize = 26
591     fSize = 26
592     fSize = 26
593     fSize = 26
594     fSize = 26
595     fSize = 26
596     fSize = 26
597     fSize = 26
598     fSize = 26
599     fSize = 26
600     fSize = 26
601     fSize = 26
602     fSize = 26
603     fSize = 26
604     fSize = 26
605     fSize = 26
606     fSize = 26
607     fSize = 26
608     fSize = 26
609     fSize = 26
610     fSize = 26
611     fSize = 26
612     fSize = 26
613     fSize = 26
614     fSize = 26
615     fSize = 26
616     fSize = 26
617     fSize = 26
618     fSize = 26
619     fSize = 26
620     fSize = 26
621     fSize = 26
622     fSize = 26
623     fSize = 26
624     fSize = 26
625     fSize = 26
626     fSize = 26
627     fSize = 26
628     fSize = 26
629     fSize = 26
630     fSize = 26
631     fSize = 26
632     fSize = 26
633     fSize = 26
634     fSize = 26
635     fSize = 26
636     fSize = 26
637     fSize = 26
638     fSize = 26
639     fSize = 26
640     fSize = 26
641     fSize = 26
642     fSize = 26
643     fSize = 26
644     fSize = 26
645     fSize = 26
646     fSize = 26
647     fSize = 26
648     fSize = 26
649     fSize = 26
650     fSize = 26
651     fSize = 26
652     fSize = 26
653     fSize = 26
654     fSize = 26
655     fSize = 26
656     fSize = 26
657     fSize = 26
658     fSize = 26
659     fSize = 26
660     fSize = 26
661     fSize = 26
662     fSize = 26
663     fSize = 26
664     fSize = 26
665     fSize = 26
666     fSize = 26
667     fSize = 26
668     fSize = 26
669     fSize = 26
670     fSize = 26
671     fSize = 26
672     fSize = 26
673     fSize = 26
674     fSize = 26
675     fSize = 26
676     fSize = 26
677     fSize = 26
678     fSize = 26
679     fSize = 26
680     fSize = 26
681     fSize = 26
682     fSize = 26
683     fSize = 26
684     fSize = 26
685     fSize = 26
686     fSize = 26
687     fSize = 26
688     fSize = 26
689     fSize = 26
690     fSize = 26
691     fSize = 26
692     fSize = 26
693     fSize = 26
694     fSize = 26
695     fSize = 26
696     fSize = 26
697     fSize = 26
698     fSize = 26
699     fSize = 26
700     fSize = 26
701     fSize = 26
702     fSize = 26
703     fSize = 26
704     fSize = 26
705     fSize = 26
706     fSize = 26
707     fSize = 26
708     fSize = 26
709     fSize = 26
710     fSize = 26
711     fSize = 26
712     fSize = 26
713     fSize = 26
714     fSize = 26
715     fSize = 26
716     fSize = 26
717     fSize = 26
718     fSize = 26
719     fSize = 26
720     fSize = 26
721     fSize = 26
722     fSize = 26
723     fSize = 26
724     fSize = 26
725     fSize = 26
726     fSize = 26
727     fSize = 26
728     fSize = 26
729     fSize = 26
730     fSize = 26
731     fSize = 26
732     fSize = 26
733     fSize = 26
734     fSize = 26
735     fSize = 26
736     fSize = 26
737     fSize = 26
738     fSize = 26
739     fSize = 26
740     fSize = 26
741     fSize = 26
742     fSize = 26
743     fSize = 26
744     fSize = 26
745     fSize = 26
746     fSize = 26
747     fSize = 26
748     fSize = 26
749     fSize = 26
750     fSize = 26
751     fSize = 26
752     fSize = 26
753     fSize = 26
754     fSize = 26
755     fSize = 26
756     fSize = 26
757     fSize = 26
758     fSize = 26
759     fSize = 26
760     fSize = 26
761     fSize = 26
762     fSize = 26
763     fSize = 26
764     fSize = 26
765     fSize = 26
766     fSize = 26
767     fSize = 26
768     fSize = 26
769     fSize = 26
770     fSize = 26
771     fSize = 26
772     fSize = 26
773     fSize = 26
774     fSize = 26
775     fSize = 26
776     fSize = 26
777     fSize = 26
778     fSize = 26
779     fSize = 26
780     fSize = 26
781     fSize = 26
782     fSize = 26
783     fSize = 26
784     fSize = 26
785     fSize = 26
786     fSize = 26
787     fSize = 26
788     fSize = 26
789     fSize = 26
790     fSize = 26
791     fSize = 26
792     fSize = 26
793     fSize = 26
794     fSize = 26
795     fSize = 26
796     fSize = 26
797     fSize = 26
798     fSize = 26
799     fSize = 26
800     fSize = 26
801     fSize = 26
802     fSize = 26
803     fSize = 26
804     fSize = 26
805     fSize = 26
806     fSize = 26
807     fSize = 26
808     fSize = 26
809     fSize = 26
810     fSize = 26
811     fSize = 26
812     fSize = 26
813     fSize = 26
814     fSize = 26
815     fSize = 26
816     fSize = 26
817     fSize = 26
818     fSize = 26
819     fSize = 26
820     fSize = 26
821     fSize = 26
822     fSize = 26
823     fSize = 26
824     fSize = 26
825     fSize = 26
826     fSize = 26
827     fSize = 26
828     fSize = 26
829     fSize = 26
830     fSize = 26
831     fSize = 26
832     fSize = 26
833     fSize = 26
834     fSize = 26
835     fSize = 26
836     fSize = 26
837     fSize = 26
838     fSize = 26
839     fSize = 26
840     fSize = 26
841     fSize = 26
842     fSize = 26
843     fSize = 26
844     fSize = 26
845     fSize = 26
846     fSize = 26
847     fSize = 26
848     fSize = 26
849     fSize = 26
850     fSize = 26
851     fSize = 26
852     fSize = 26
853     fSize = 26
854     fSize = 26
855     fSize = 26
856     fSize = 26
857     fSize = 26
858     fSize = 26
859     fSize = 26
860     fSize = 26
861     fSize = 26
862     fSize = 26
863     fSize = 26
864     fSize = 26
865     fSize = 26
866     fSize = 26
867     fSize = 26
868     fSize = 26
869     fSize = 26
870     fSize = 26
871     fSize = 26
872     fSize = 26
873     fSize = 26
874     fSize = 26
875     fSize = 26
876     fSize = 26
877     fSize = 26
878     fSize = 26
879     fSize = 26
880     fSize = 26
881     fSize = 26
882     fSize = 26
883     fSize = 26
884     fSize = 26
885     fSize = 26
886     fSize = 26
887     fSize = 26
888     fSize = 26
889     fSize = 26
890     fSize = 26
891     fSize = 26
892     fSize = 26
893     fSize = 26
894     fSize = 26
895     fSize = 26
896     fSize = 26
897     fSize = 26
898     fSize = 26
899     fSize = 26
900     fSize = 26
901     fSize = 26
902     fSize = 26
903     fSize = 26
904     fSize = 26
905     fSize = 26
906     fSize = 26
907     fSize = 26
908     fSize = 26
909     fSize = 26
910     fSize = 26
911     fSize = 26
912     fSize = 26
913     fSize = 26
914     fSize = 26
915     fSize = 26
916     fSize = 26
917     fSize = 26
918     fSize = 26
919     fSize = 26
920     fSize = 26
921     fSize = 26
922     fSize = 26
923     fSize = 26
924     fSize = 26
925     fSize = 26
926     fSize = 26
927     fSize = 26
928     fSize = 26
929     fSize = 26
930     fSize = 26
931     fSize = 26
932     fSize = 26
933     fSize = 26
934     fSize = 26
935     fSize = 26
936     fSize = 26
937     fSize = 26
938     fSize = 26
939     fSize = 26
940     fSize = 26
941     fSize = 26
942     fSize = 26
943     fSize = 26
944     fSize = 26
945     fSize = 26
946     fSize = 26
947     fSize = 26
948     fSize = 26
949     fSize = 26
950     fSize = 26
951     fSize = 26
952     fSize = 26
953     fSize = 26
954     fSize = 26
955     fSize = 26
956     fSize = 26
957     fSize = 26
958     fSize = 26
959     fSize = 26
960     fSize = 26
961     fSize = 26
962     fSize = 26
963     fSize = 26
964     fSize = 26
965     fSize = 26
966     fSize = 26
967     fSize = 26
968     fSize = 26
969     fSize = 26
970     fSize = 26
971     fSize = 26
972     fSize = 26
973     fSize = 26
974     fSize = 26
975     fSize = 26
976     fSize = 26
977     fSize = 26
978     fSize = 26
979     fSize = 26
980     fSize = 26
981     fSize = 26
982     fSize = 26
983     fSize = 26
984     fSize = 26
985     fSize = 26
986     fSize = 26
987     fSize = 26
988     fSize = 26
989     fSize = 26
990     fSize = 26
991     fSize = 26
992     fSize = 26
993     fSize = 26
994     fSize = 26
995     fSize = 26
996     fSize = 26
997     fSize = 26
998     fSize = 26
999     fSize = 26
1000    fSize = 26

```

```

110 convergence = False
111 while(not convergence):
112     solutionError = np.subtract(M.dot(x), b)
113     absErr = np.linalg.norm(solutionError)
114     errorDataConjugateGradients.append(math.log(absErr))
115
116 except:
117     convergence = True
118
119 if(absErr < tol):
120     convergence = True
121
122 break
123
124 alpha_numerator = r.dot(r)
125 alpha_denominator = d.dot(M.dot(d))
126 if(alpha_denominator < avoidDivByZeroError):
127     convergence = True
128
129 alpha = 1.0 * alpha_numerator / alpha_denominator
130
131 x = np.add(x, np.multiply(d, alpha))
132 r_new = np.subtract(r, np.multiply(M.dot(d), alpha))
133
134 beta_numerator = r_new.dot(r_new)
135 beta_denominator = r.dot(r)
136 if(beta_denominator < avoidDivByZeroError):
137     convergence = True
138
139 break
140
141 beta = 1.0 * beta_numerator / beta_denominator
142 d = r_new + np.multiply(d, beta)
143 r = r_new
144
145 return x, absErr, errorDataConjugateGradients
146
147 def ConjugateGradients_GolubM, b):
148     errorDataConjugateGradients = []
149     tol = 0.000001
150     x = np.zeros_like(b)
151     r = np.subtract(b, A.dot(x))
152     ro_c = r.dot(r)
153     data = ro_c / np.linalg.norm(b)
154     while(not convergence):
155         err = np.subtract(M.dot(x), b)
156         absErr = np.linalg.norm(err)
157         errorDataConjugateGradients.append(absErr)
158         if(k == 1):
159             p = r
160         else:
161             tau = ro_c / ro_mins
162             p = np.add(r, np.multiply(p, tau))
163             w = A.dot(p)
164             mlu_nominator = ro_c
165             mlu_nominator = w.dot(p) / denominator
166             x = np.add(x, np.multiply(p, mlu))
167             r = np.subtract(r, np.multiply(w, mlu))
168             ro_minus = ro_c
169             ro_c = r.dot(r)
170             err = np.subtract(M.dot(x), b)
171             absErr = np.linalg.norm(err)
172             errorDataConjugateGradients.append(absErr)
173
174 err = np.subtract(M.dot(x), b)
175 absErr = np.linalg.norm(err)
176 errorDataConjugateGradients.append(absErr)
177 return x, absErr, errorDataConjugateGradients
178
179 def plotDiscretizedSineID():
180     h = 1.0 / N
181     h = 1.0 / N
182     xCont = np.arange(0.0, 1.0 + r, r)
183     xDiscr = np.arange(0.0, 1.0 + h, h)
184     xCont2 = np.arange(0.0, 1.0 + r, r)
185     xDiscr2 = np.arange(0.0, 1.0 + h, h)
186
187 k = 5.0
188 confFunction = np.sin(math.pi * k * xCont)
189 discrFunction = np.sin(math.pi * k * xDiscr)
190
191 confFunction2 = np.sin(math.pi * 8.0 * xCont2)
192 discrFunction2 = np.sin(math.pi * 8.0 * xDiscr2)
193
194 plt.subplot(211)
195 plt.plot(xCont, confFunction)
196 plt.plot(xDiscr, discrFunction)
197
198 plt.subplot(212)
199 plt.plot(xCont, confFunction2)
200 plt.plot(xDiscr, discrFunction2)
201
202 plt.plot(xDiscr, discrFunction, linestyle='dashed')
203
204 plt.plot(xDiscr, discrFunction2, linestyle='dashed')
205
206 plt.show()
207
208 def plotSineNodes():
209     N = 16
210     h = 1.0 / N
211     xDiscr = np.arange(0.0, 1.0 + h, h)
212     discrFunction1 = np.sin(math.pi * 1.0 * xDiscr)
213     discrFunction2 = np.sin(math.pi * 4.0 * xDiscr)
214     discrFunction3 = np.sin(math.pi * 9.0 * xDiscr)
215     discrFunction4 = np.sin(math.pi * 16.0 * xDiscr
```

```

330 plt.plot(x0IsCr2, discrFunction2, 'ko')
331
332 plt.subplot(313)
333 plt.ylim(-0.22, 0.22)
334 plt.plot(x0IsCr2, discrFunction3, linestyle='dashed')
335 plt.plot(x0IsCr2, discrFunction3, 'ko')
336
337 plt.show()
338
339 def plotVCycle(h):
340     h = 0.1
341
342     plt.xlim(0,6)
343     plt.ylim(0.5, 3.5)
344     ys = [3, 2, 1, 2, 3]
345     plt.plot(ys, ys)
346
347     plt.plot(ys, ys, 'r-', markerfacecolor='k', markeredgecolor='k', markersize = 10)
348     plt.text(1.5, 2, h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
349     plt.text(2.5, 1.5, h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
350     plt.text(3.5, 1.5, h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
351     plt.text(4.0 + 2 * h, 2.0, h, s="$\Omega$mega~$2h$%; compute exact solution", fontsize = 20)
352     plt.text(4.0 + 2 * h, 2.0, h, s="$\Omega$mega~$2h$%; relax $\nu$mu 28 times", fontsize = 20)
353     plt.text(4.5 + 2 * h, 3.0 + h, s="$\Omega$mega~$2h$%; relax $\nu$mu 28 times", fontsize = 20)
354
355     plt.text(1.5, 2 * h, 2.5 * h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
356     plt.text(2.5, 2 * h, 1.5 * h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
357
358     plt.text(0.5 + 1.5 * h, 1.5 * h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
359     plt.text(1.5 + 1.5 * h, 2.5 * h, s="$\omega$athb(I1, 2h)~$2h$%", fontsize = 20)
360
361     plt.arrow(1.0, 3.0, 0.8, -0.8, head_width = 0.2, head_length = 0.2)
362     plt.arrow(2.0, 2.0, 0.8, -0.8, head_width = 0.1, head_length = 0.2)
363
364     plt.arrow(3.0, 1.0, 0.8, 0.8, head_width = 0.1, head_length = 0.2)
365     plt.arrow(4.0, 2.0, 0.8, 0.8, head_width = 0.1, head_length = 0.2)
366
367     plt.arrow(1.05, 3.0, 3.8, 0.0, head_width = 0.05, head_length = 0.05, color = "red")
368     plt.arrow(2.05, 2.0, 1.8, 0.0, head_width = 0.05, head_length = 0.05, color = "red")
369
370     plt.text(3 - 5 * h, 3 - h, "Correct approximation on $\Omega$mega~$2h$%", color = "red",
371            fontsize = 13)
372     plt.text(3 - 5 * h, 2 - h, "Correct approximation on $\Omega$mega~$2h$%", color = "red",
373            fontsize = 13)
374
375     plt.show()
376
377 plotVCycle(h)

```

```

220 plt.subplot(511)
221 plt.title('%sin(%pi x)%')
222 plt.plot(xbscr, discFunction1, linestyle='dashed')
223 plt.plot(xbscr, discFunction1, 'ko')
224
225 plt.subplot(512)
226 plt.title('%sin((4/pi x))%')
227 plt.plot(xbscr, discFunction2, linestyle='dashed')
228 plt.plot(xbscr, discFunction2, 'ko')
229
230 plt.subplot(513)
231 plt.title('%sin(10/pi x)%')
232 plt.plot(xbscr, discFunction3, linestyle='dashed')
233 plt.plot(xbscr, discFunction3, 'ko')
234
235 plt.subplot(514)
236 plt.title('%sin(14/pi x)%')
237 plt.plot(xbscr, discFunction4, linestyle='dashed')
238 plt.plot(xbscr, discFunction4, 'ko')
239
240 plt.subplot(515)
241 plt.title('%sin(31/pi x)%')
242 plt.plot(xbscr, discFunction5, linestyle='dashed')
243 plt.plot(xbscr, discFunction5, 'ko')
244
245 plt.show()
246
247
248 def plotProjectedSine():
249     N1 = 16
250     h1 = 1.0 / N1
251     xbscr1 = np.arange(0.0, 1.0 + h1, h1)
252
253     N2 = 8
254     h2 = 1.0 / N2
255     xbscr2 = np.arange(0.0, 1.0 + h2, h2)
256     xbscr22 = np.arange(0.0, 2.0 + h2, h2)
257
258     discFunction1 = np.sin(math.pi * 6.0 * xbscr1)
259     discFunction2 = np.sin(math.pi * 6.0 * xbscr2)
260     discFunction22 = np.sin(math.pi * 6.0 * xbscr22)
261
262     plt.subplot(311)
263     plt.plot(xbscr1, discFunction1, linestyle='dashed')
264     plt.plot(xbscr1, discFunction1, 'ko')
265
266     plt.subplot(312)
267     plt.plot(xbscr2, discFunction2, linestyle='dashed')
268     plt.plot(xbscr2, discFunction2, 'ko')
269
270     plt.subplot(313)
271     plt.plot(xbscr22, discFunction22, linestyle='dashed')
272     plt.plot(xbscr22, discFunction22, 'ko')
273     plt.show()
274
275 def plotGrid():
276     plt.plot([0.0, 0.0, 0.0, 0.0, 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], marker = 'o', markersize = 10, markeredgewidth = 10,
277             color = 'r', linestyle = 'solid')
278     plt.show()
279
280 def plotInterpolatedExample():
281     N = 4.0
282     h = 0.5 / N
283     h2 = 0.5 / N
284     xbscr1 = np.arange(0.0, 1.0 + h1, h1)
285     xbscr2 = np.arange(0.0, 1.0 + h2, h2)
286     discFunction1 = [0.0, 0.3, -0.2, 0.1, 0.0]
287     discFunction2 = [0.0, 0.3, -0.2, 0.1, 0.0]
288     for i in range(len(discFunction1) - 1):
289         discFunction2.append(discFunction1[i])
290     discFunction2.append(discFunction1[-1])
291     discFunction2.append(discFunction1[-1])
292     plt.subplot(211)
293     plt.plot(xbscr1, discFunction1, linestyle='dashed')
294     plt.plot(xbscr1, discFunction1, 'ko')
295
296     plt.subplot(212)
297     plt.plot(xbscr2, discFunction2, linestyle='dashed')
298     plt.plot(xbscr2, discFunction2, 'ko')
299     plt.show()
300
301 def plotRestrictionExample():
302     N = 4.0
303     h1 = 0.5 / N
304     h2 = 1.0 / N
305     xbscr1 = np.arange(0.0, 1.0 + h1, h1)
306     xbscr2 = np.arange(0.0, 1.0 + h2, h2)
307
308     discFunction1 = [0.0, 0.1, -0.1, 0.15, -0.2, 0.1, -0.13, 0.09, 0.0]
309     discFunction2 = []
310
311     for i in range(len(discFunction1)):
312         discFunction2.append(discFunction1[i])
313
314     discFunction3 = []
315     for i in range(len(discFunction1)):
316         if i == 0 or i == len(discFunction1) - 1:
317             discFunction3.append(0.0)
318         else:
319             discFunction3.append((2 * discFunction1[i] - discFunction1[i+1]) * discFunction1[i+1]) / 4.0)
320
321     plt.subplot(311)
322     plt.plot(xbscr1, discFunction1, linestyle='dashed')
323     plt.plot(xbscr1, discFunction1, 'ko')
324
325     plt.subplot(312)
326     plt.plot(xbscr2, discFunction2, linestyle='dashed')
327     plt.plot(xbscr2, discFunction2, 'ko')
328
329     plt.subplot(313)
330     plt.plot(xbscr2, discFunction3, linestyle='dashed')
331     plt.plot(xbscr2, discFunction3, 'ko')
332
333     plt.show()
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914

```

Bibliography

- [1] ALBL, C. Conjugate gradients explained. http://cmp.felk.cvut.cz/~alblcene/CG/CG_explained.pdf.
- [2] ALLEN, R. C., BOTTCHEER, 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](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 MELJERINK, 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.