MATLAB Implementation of a Multigrid Solver for Diffusion Problems:

Graphics Processing Unit vs. Central Processing Unit

Kristin Paulsen

Thesis submitted for the degree
Master of Science

Physics of Geological Processes

Department of Physics

University of Oslo

Norway June 2010

Acknowledgements

First of all I would like to thank my supervisors Dani Schmid and Marcin Dabrowski for inspiring discussions and the large amount of knowledge that they have sheared with me. I have never been pushed as hard and given so many challenges in a year. I would never have gotten this far without your help. Thank you!

There are many at PGP that deserve my gratitude for all the knowledge they have given me. I would especially like to thank Marcin Krotkiewski for helping me developing my programming skills, for all the technical information he has given to me and the programs he developed for testing the CUDA.

I would like to thank the "the lovely master students", Kristin, Elvira, Øystein, Håkon and Bodil, for their friendship and for taking care of me through hard times. I would like to thank them for all the interesting discussions, but most of all for making most days at the university a joy and for all the laughter. I hope we will stay friends for years after my time at PGP.

My soccer team, OSI (Oslo Studentenes Idrettsforening), has helped me to not loose focus on the life outside the university. I have never met a group of people with such a variety of interests, which allows me to gain incite to other subjects. Further more I would like to thank my other friends both in Oslo and back home in Trondheim.

I would like to thank my boyfriend Sigfred Sørensen for all the love he has given me. I would never have overcome the frustrations and challenges without his support.

Last but not least I would like to thank my family. My parents Anne Margrethe Paulsen and Tore Paulsen for making me believe I can manage anything I want, and for all their love and support. There are many things I would like to thank my brother for. The most important of which is helping me to keep focus on what I want to do, and not only on what is expected of me.

Contents

1 Introduction

Graphics Processing Units are immensely powerful processors and for variety applications they outperform the Central Processing Unit, CPU. The recent generations of GPU's have a flexible architecture than older generations and programming interface more user friendly, which makes them better suited for general purpose programming. A high end GPU can give a desktop computer the same computational power as a small cluster of CPU's.
Speedup of applications by using the GPU has been shown in a variety of research fields, including medicine, finance and earth science. 3D seismic imaging is extensively used in oil exploration, and imaging complex geological areas is heavy computational task involving terabytes of data. Seismic imaging software that utilizes GPU's is being developed by companies such as SeismicCity. They found that 20x performance increase can be achieved by utilizing GPU's in their computer setup.

In this thesis it is shown that the GPU architecture is well suited for solving partial differential equations on structured grids. A parallel multigrid method algorithm is implemented using Jacket that can harness the computational power of the GPU. Jacket uses MATLAB syntax, which allow for more rapid development of algorithms. This does, however, come at a price, implementations that are developed in high level languages is not as efficient as implementations developed in low level languages such as C.

The ideas used in multigrid have been adapted to solve a broad spectrum of problems that involves structures that do not necessarily resemble any form of physical grid. They can for example be used to solve problems characterized by matrix structures, particle structures and lattice structures. The collection of methods that build on the same ideas as the multigrid method is often called multilevel methods, but there is no official unified term for these methods.

The multigrid algorithm implemented in this thesis efficiently solves Poisson problems for homogenous systems in 2 and 3 dimensions. The GPU implementation is 60 to 70 times faster than the equivalent CPU implementation, and can solve systems of size 257^3 in less than a second.

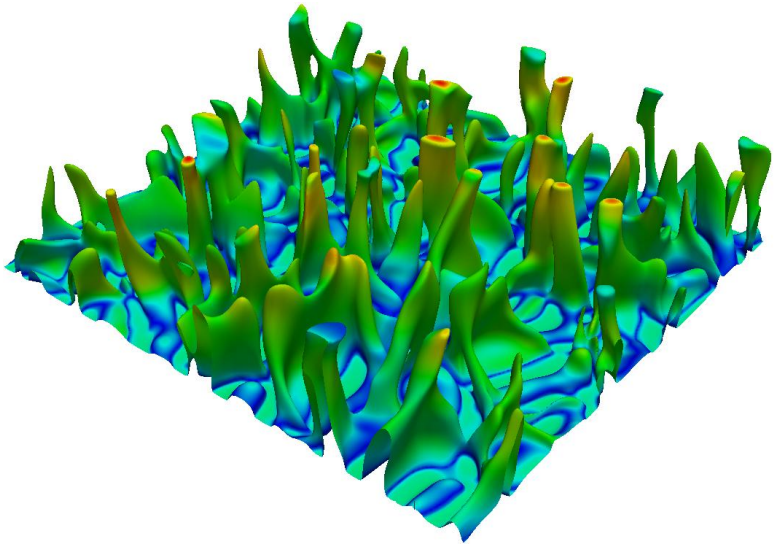

Figure 1.1: Simulation of porous convection made in this thesis.

Poisson solvers can be used to solve a variety of physical problems either as a stand alone solver or as a part of another solver. In this thesis it is shown that it can be used in an application where porous convection is simulated, see Figure 1.1. Porous convection can describe migration of ground water and hydrocarbons in the earth's crust.

The main aim of the thesis is to show how partial differential equations can be solved with the use of the multigrid algorithm and accelerated with the use of the graphics processing unit. The first chapter describes how a partial differential equation can be discretized on a regular grid, and solved using finite difference methods. In this chapter the heat diffusion equation is introduced, which is extensively used as an example throughout the text.

The rest of the thesis is divided into three main sections. The first one describes a series of both direct and iterative methods for solving linear sets of equations is presented and their strengths and weaknesses are discussed. The emphasis is on the multigrid algorithm that is implemented in the thesis. The second part describes the architecture of the GPU and techniques used to utilizing it. In the third part the applications made for this thesis is described and results are presented.

2 Discretization Approach

In this thesis the focus is on solving partial differential equations, PDE's, numerically. PDE's describe the rate of change of various physical quantities in relation to space and time. They can be used to describe a large variety of problems in science and engineering; for example physical phenomena such as propagation of sound or heat, fluid flow and electrodynamics. There are a limited number of systems described by partial differential equations that can be solved analytically. More complex systems can be analysed using a numerical approximations.

There are three main classes of partial differential equations,

- Elliptic, such as the Poisson equation $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f.$
- Hyperbolic, such as the wave equation $\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$
- Parabolic, such as the diffusion equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$

Elliptic partial differential equations result in boundary value problems, i.e. the solution is defined by the boundary conditions. Hyperbolic and parabolic equations describe time evolution problems. The solution of time evolution problems is defined by both the initial values and the boundary conditions. Whether the problem is a time evolution problem or a boundary value problem is more decisive for the numerical implementation of it than which class of equation it is.

There are several techniques that are used to solve partial differential equations, two of them are the finite difference and the finite element method. In this thesis the finite difference method is used, the reason for this choice of technique is elaborated on in the following subsection. The heat diffusion equation is chosen as a specific example for the finite difference discretization; it is presented in subsection 2.2. The heat diffusion equation can be solved as both a time evolution problem, i.e. transient heat diffusion, and as a boundary value problem, i.e. steady state heat diffusion see section 2.2.3.

2.1 Finite Difference Method

Finite difference and finite element methods are techniques for solving partial differential equations numerically. In the finite difference method the values of the function are defined at certain points in the domain and the derivatives are approximated locally using equations derived from Taylor expansion. In the finite element method the function is piecewise defined by polynomial functions. The partial differential equations are solved in integral form; using the weak formulation of the integrals reduces the restrictions on the polynomials.

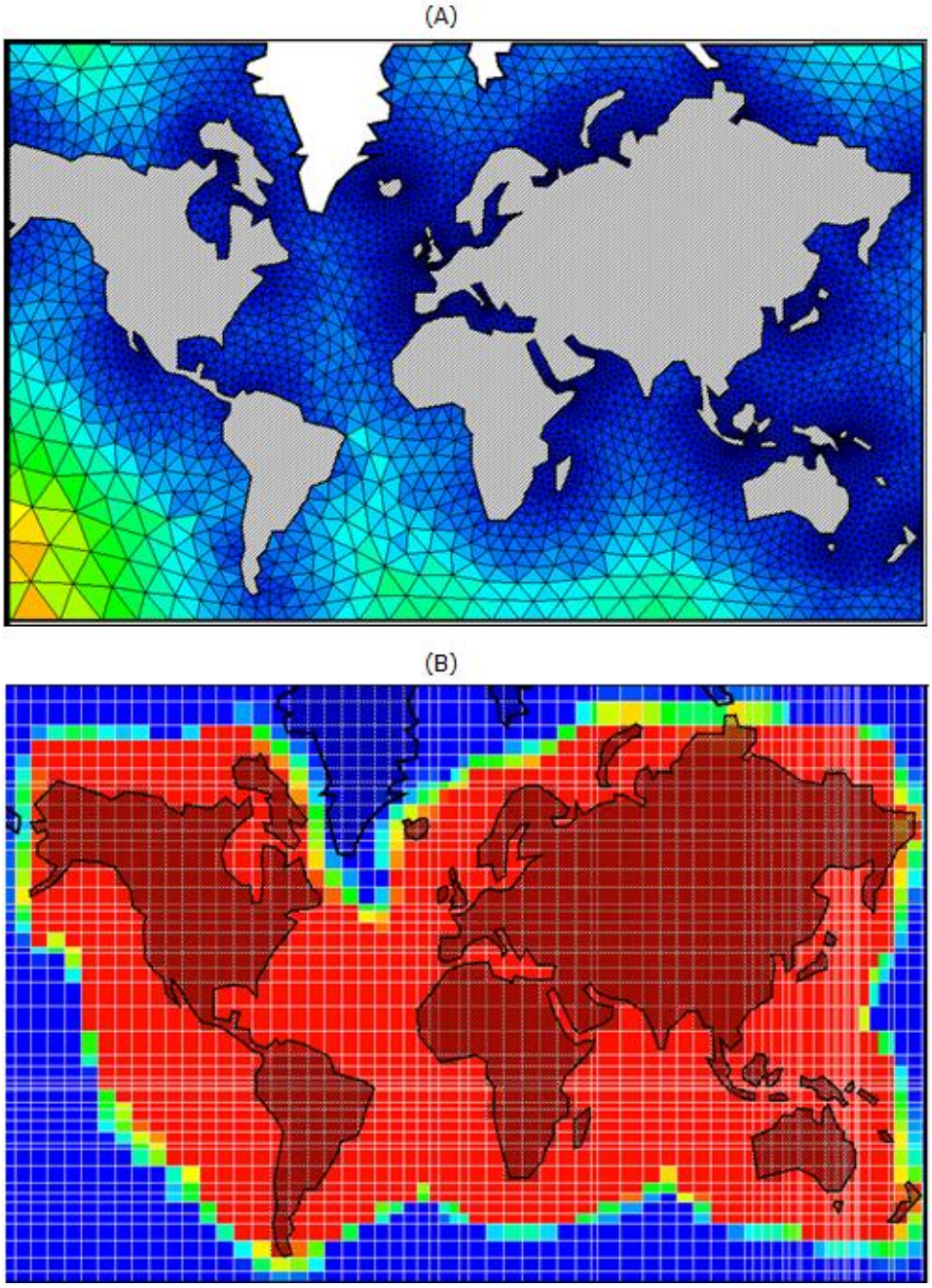

Figure 2.1: A) Body fitted mesh that can be applied when using finite element discretization PDE's, see figure. B) For finite differences a regular grid must be used. Source www.augsint.com /MeshGeneration.

The main argument for applying the finite element method is that it can handle complex geometries. This is because it allows for the use of body fitted meshes, see Figure 2.1A. A larger number of grid points in certain regions allow the complex geometry to be well represented, and computational power can be saved by having less grid points in simpler regions. The finite difference method is restricted to the use of regular grids, meaning that it is built up of rectangular blocks see Figure 2.2b.

A larger number of grid points are needed to represent a complex geometry using regular grids and finite difference methods. Finite difference approximations do, however, result in linear sets of equations were the coefficient matrix has more favourable properties, which allow for the use of more efficient solvers.

The discretization of a problem on a regular grid does in many ways resemble pixels in a digital image. This motivates the idea of them being well suited to the architecture of the graphics processing units; the graphics processing unit is a specialised processor that handles rendering of images on the computer screen.

In the multigrid algorithm a hierarchy of discretizations are used, with different levels of coarsening. Finding a reasonable set of coarser grids is more feasible for a regular grid than for body fitted meshes. There is ways to get around this problem, but this is beyond the scope of this thesis.

2.1.1 Discretization Grid

To solve the equations numerically on the computer they must be discretized on a grid, some examples of regular grids are shown in Figure 2.2. The vertices, i.e. the junctions between the blocks, are called grid points or nodes. Each node is numbered so that it can be identified. The function values are only defined at these nodes, as shown in Figure 2.2C.

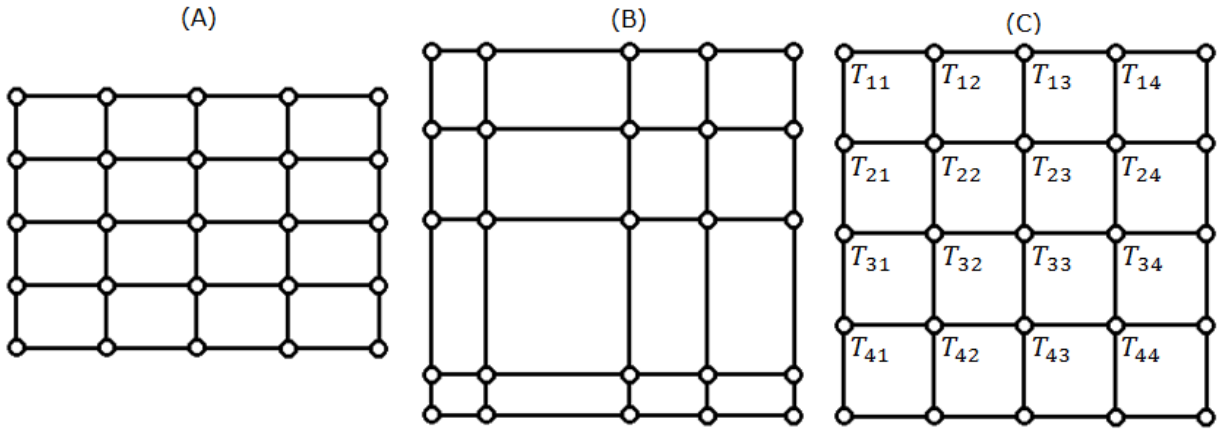

Figure 2.2: Different regular grids, all built up of rectangular blocks of different sizes with nodes at their junctions. (A) The distance between the nodes are at regular intervals in both spatial directions (x and y), i.e. Δx and Δy are constant. (B) Anisotropic grid, irregular spacing in both of the spatial directions. (C) The interval between the nodes in both of the spatial dimensions is constant and equal to each other. The physical quantity, for example temperature, is defined at the nodes. The values at the nodes are numbered.

2.1.2 Approximation of the Derivative

In the finite difference method partial differential equations are solved by replacing the derivatives with finite difference approximations of the derivatives. The approximations are derived from a Taylor expansion of the function

$$\boldsymbol{T}(x+\Delta x) = \boldsymbol{T}(x) + \frac{\boldsymbol{T}^{'}(x)}{1!}\Delta x + \frac{\boldsymbol{T}^{''}(x)}{2!}(\Delta x)^2 + \frac{\boldsymbol{T}^{'''}(x)}{3!}(\Delta x)^3 + \cdots \tag{2.1}$$

Δx is the distance between where the function value is approximated and the point where the function value is known. $\boldsymbol{T}(x)$ could be any function describing the behaviour of some physical quantity in the system, for example temperature. Rearranging this expression for the Taylor expansion of the function yields the following approximation for the first derivative

$$\boldsymbol{T}^{'}(x) = \frac{\boldsymbol{T}(x+\Delta x) - \boldsymbol{T}(x)}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.2}$$

$\mathcal{O}(\Delta x)$ means that the error caused by the truncation of the Taylor expansion is in the order of Δx. In the finite difference method this expression is used to approximate the derivative using the values at the grid points

$$\frac{\partial \boldsymbol{T}}{\partial x}|_{i+1/2} = \frac{\boldsymbol{T}_{i+1} - \boldsymbol{T}_{\boldsymbol{i}}}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.3}$$

$\frac{\partial \boldsymbol{T}}{\partial x}|_{i+1/2}$ means that the derivative is analysed between the grid points i and $i+1$. The truncation error decreases as the distance between the nodes decreases. This does, however, increase the chance to get round-off error as $\boldsymbol{T}_{i+1} - \boldsymbol{T}_{\boldsymbol{i}} \to 0$. This approximation for the first derivative is called forward differences or forward Euler. The backward Euler method is given by the following expression

$$\frac{\partial \boldsymbol{T}}{\partial x}|_{i-1/2} = \frac{\boldsymbol{T}_{i} - \boldsymbol{T}_{\boldsymbol{i-1}}}{\Delta x} + \mathcal{O}(\Delta x) \tag{2.4}$$

In the forward and backward Euler method the derivatives are evaluated at the midpoint between the grid points. To find the derivative in the nodes the central finite difference approximation can be used

$$\frac{\partial \boldsymbol{T}}{\partial x}|_{i} = \frac{\boldsymbol{T}_{i+1} - \boldsymbol{T}_{\boldsymbol{i-1}}}{2\Delta x} + \mathcal{O}(\Delta x^2) \tag{2.5}$$

The central finite difference method has a smaller truncation error, which is in the order of Δx^2.

By combining the Taylor expansion used to find the forward and backward Euler method we find a central finite difference approximation of the second derivative

$$\frac{\partial^2 \boldsymbol{T}}{\partial x^2}|_i = \frac{\boldsymbol{T}_{i+1} - 2\boldsymbol{T}_i + \boldsymbol{T}_{\boldsymbol{i-1}}}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{2.6}$$

Higher order derivatives may be used to find finite difference approximations that have a better accuracy.

The discretization approach can be applied to functions of several variables. In two dimensions this is the origin of the 5-point stencil for the derivative. Assuming that the intervals between the grid points are the same in both spatial directions, i.e. $\Delta x = \Delta y = h$, and using equation (2.6) to approximate the derivatives yields the following expression

$$\Delta T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{\boldsymbol{T}_{i+1,j} + \boldsymbol{T}_{i-1,j} + \boldsymbol{T}_{i,j+1} + \boldsymbol{T}_{i,j-1} - 4\boldsymbol{T}_{i,j}}{h^2} + \mathcal{O}(h^2) \tag{2.7}$$

It can be written in stencil notation

$$\Delta_5 = \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} + \mathcal{O}(h^2) \tag{2.8}$$

Some higher order approximations for the Laplacian, Δ, are discussed in section 2.2.4.

2.2 Heat Diffusion Equation

The heat diffusion equation is here used as an explicit example of a problem that can be solved using the algorithms implemented in this thesis, it is defined as

$$c_p \rho \frac{\partial T}{\partial t} = \nabla \cdot [k \nabla T] + q \tag{2.9}$$

Where c_p, k and ρ are the specific heat capacity, thermal conductivity and density respectively. These are material dependent properties. The specific heat capacity is a measure of how much energy is needed to rise temperature of the material, it has the units joule per kilogram per Kelvin ($[J/(kgK)]$). Thermal conductivity is a measure of the materials ability to conduct heat, it has the units watts per Kelvin per metre ($[W/(Km)]$).

Heat diffusion is only one example of a problem that can be solved using an equation of this form. The Darcy law describes pressure driven flow through a porous media. By assuming that the fluid is incompressible it can be formulated on the same form as the heat diffusion equation. Darcy law is used to describe hydrocarbon and water flows through reservoirs, and it can be used to find material parameters of the reservoir rocks such as permeability.

In the subsequent sections the discretization is done for the heat diffusion equation in two dimensions as an example

$$c_p \rho \frac{\partial T}{\partial t} = \left(\frac{\partial}{\partial x} k \frac{\partial}{\partial x} T + \frac{\partial}{\partial y} k \frac{\partial}{\partial y} T \right) + q \tag{2.10}$$

2.2.1 Homogeneous Case

The material properties are constant in the whole domain for homogeneous materials. The thermal conductivity is therefore unaffected by the spatial derivatives, which yields the simpler form of the heat diffusion equation

$$c_p \rho \frac{\partial T}{\partial t} = k \Delta T + q \tag{2.11}$$

Equation (2.6) is used to discretize spatial derivatives and forward Euler, equation (2.2), is used to approximate the time derivative. This yields the following discretized expression of the heat diffusion equation for homogeneous materials

$$c_p \rho \left(\frac{\boldsymbol{T}_{ij}^{l+1} - \boldsymbol{T}_{ij}^{l}}{\Delta t} \right) \approx k \left(\frac{\boldsymbol{T}_{i+1j}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{i-1j}^{l}}{\Delta x^2} + \frac{\boldsymbol{T}_{ij+1}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{ij-1}^{l}}{\Delta y^2} \right) + q_{ij} \tag{2.12}$$

i and j are the indices for the nodes in the x- and y-direction respectively. l denotes the indices for the time steps. The discretized equation has second order accuracy for spatial derivatives and first order accuracy for the time derivative, see section 2.1.2.

This is an explicit scheme, meaning that the unknowns, $\boldsymbol{T}_{ij}^{l+1}$, are explicitly given by the equation. The truncation error from the Taylor expansion will be amplified if the constant in front of the approximation scheme is larger than unity, this results in an unstable scheme. The explicit scheme is therefore stable if the following inequality is upheld

$$\Delta t \leq \frac{c_p \rho}{k} \Delta x^2 \tag{2.13}$$

An implicit scheme for the equation is found by using backward Euler approximation for the time derivative, which results in the following scheme

$$\frac{\boldsymbol{T}_{i+1j}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{i-1j}^{l}}{\Delta x^2} + \frac{\boldsymbol{T}_{ij+1}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{ij-1}^{l}}{\Delta y^2} + \frac{1}{k} q_{ij} = \frac{c_p \rho}{k} \left(\frac{\boldsymbol{T}_{ij}^{l} - \boldsymbol{T}_{ij}^{l-1}}{\Delta t} \right) \tag{2.14}$$

The unknowns $\boldsymbol{T}_{ij}^{l}$ are found by solving a set of linear equations given by

$$\boldsymbol{T}_{ij}^{l} - \frac{k \Delta t}{c_p \rho} \left(\frac{\boldsymbol{T}_{i+1j}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{i-1j}^{l}}{\Delta x^2} + \frac{\boldsymbol{T}_{ij+1}^{l} - 2\boldsymbol{T}_{ij}^{l} + \boldsymbol{T}_{ij-1}^{l}}{\Delta y^2} \right) + \frac{\Delta t}{c_p \rho} q_{ij} = \boldsymbol{T}_{ij}^{l-1} \tag{2.15}$$

The implicit scheme is stable even if the inequality in equation (2.13) is not upheld, and can therefore be used to solve steady state problems (see section 2.2.3).

2.2.2 Heterogeneous Case

In heterogeneous materials the material properties, that is the specific heat capacity, thermal conductivity and density, vary in space. The thermal conductivity is therefore affected by the spatial derivatives. The spatial derivatives are given by the following expressions for heterogeneous materials

$$\frac{\partial^2 T}{\partial x^2}\Big|_{ij} = \frac{k_{i+\frac{1}{2}j} \dfrac{\boldsymbol{T}_{i+1j}^{l} - \boldsymbol{T}_{ij}^{l}}{\Delta x} - k_{i-\frac{1}{2}j} \dfrac{\boldsymbol{T}_{ij}^{l} - \boldsymbol{T}_{i-1j}^{l}}{\Delta x}}{\Delta x} \tag{2.16}$$

$$\frac{\partial^2 T}{\partial y^2}\Big|_{ij} = \frac{k_{ij+\frac{1}{2}} \dfrac{\boldsymbol{T}_{ij+1}^{l} - \boldsymbol{T}_{ij}^{l}}{\Delta y} - k_{ij-\frac{1}{2}} \dfrac{\boldsymbol{T}_{ij}^{l} - \boldsymbol{T}_{ij-1}^{l}}{\Delta y}}{\Delta y} \tag{2.17}$$

$\frac{\partial^2 T}{\partial x^2}|_{ij}$ means that the derivative is approximated in the grid point specified by the indexes i and j, and $k_{i+\frac{1}{2}j}$ and $k_{ij+\frac{1}{2}}$ means that the thermal conductivity is defined in the midpoint between the grid points in the x-direction and y-direction respectively. The time derivative is unaffected by the varying thermal conductivities. The complete discretized expression for the heat diffusion equation for a heterogeneous system using an implicit scheme is given by

$$\frac{k_{i+\frac{1}{2}j}\boldsymbol{T}^l_{i+1j} - (k_{i+\frac{1}{2}j} + k_{i-\frac{1}{2}j})\boldsymbol{T}^l_{ij} - k_{i-\frac{1}{2}j}\boldsymbol{T}^l_{i-1j}}{\Delta x^2} + \frac{k_{ij+\frac{1}{2}}\boldsymbol{T}^l_{ij+1} - (k_{ij+\frac{1}{2}} + k_{ij-\frac{1}{2}})\boldsymbol{T}^l_{ij} - k_{ij-\frac{1}{2}}\boldsymbol{T}^l_{ij-1}}{\Delta y^2} + \frac{1}{k}q_{ij} = \frac{c_p\rho}{k}\left(\frac{\boldsymbol{T}^l_{ij} - \boldsymbol{T}^{l-1}_{ij}}{\Delta t}\right) \tag{2.18}$$

The conductivities must be defined between the nodes to evaluate the derivatives. This can be done by using a staggered grid, meaning that the material properties are defined where they are needed, or by using averages of conductivities defined in the nodes.

2.2.3 Steady State Heat Diffusion

In many cases we are not interested in how a system evolves with time but rather how the end result is going to be. This can not be found directly using the explicit scheme since the scheme is unstable for large time steps. For the implicit scheme, however, this is not a problem.

As the length of the time step approaches infinity we get the following equation based on the implicit scheme for the heat diffusion, see equation (2.14)

$$\frac{\boldsymbol{T}^l_{i+1j} - 2\boldsymbol{T}^l_{ij} + \boldsymbol{T}^l_{i-1j}}{\Delta x^2} + \frac{\boldsymbol{T}^l_{ij+1} - 2\boldsymbol{T}^l_{ij} + \boldsymbol{T}^l_{ij-1}}{\Delta y^2} + c_p\rho q_{ij} = 0 \tag{2.19}$$

2.2.4 Boundary Conditions

The boundary nodes must be treated separately since the schemes to approximate the derivatives utilize the values in the neighbouring points. There are three main types of boundary conditions, which are Dirichlet, von Neumann and periodic boundary conditions.

Dirichlet boundary condition specifies the function value, i.e. temperature for the heat diffusion equation, at the boundary. The first derivative of the function value is specified if von Neumann boundary conditions are used. The physical interpretation of von Neumann boundary conditions are that the flux at the boundary is set to some value, usually zero. Periodic boundary conditions are often used to simulate a system that is infinitely large.

Dirichlet boundary conditions are implemented by setting the values at the boundaries to the specified boundary values directly. To keep symmetry in the coefficient matrix it is useful to remove equations for the boundary points from the set of linear equations.

Von Neumann boundary conditions are handled by introducing ghost points just outside the domain, see Figure 2.3. The central finite difference scheme, equation (2.5), is used to approximate the first derivative at the boundary. The flux is set to zero at the boundary if the values in the ghost points are equal to the grid points inside the domain. This yields the following expression for the approximation of the second derivative in the x-direction at the left boundary

$$\frac{\partial^2 T}{\partial x^2}\Big|_{ij} = \frac{2\boldsymbol{T}^l_{i+1j} - 2\boldsymbol{T}^l_{ij}}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{2.20}$$

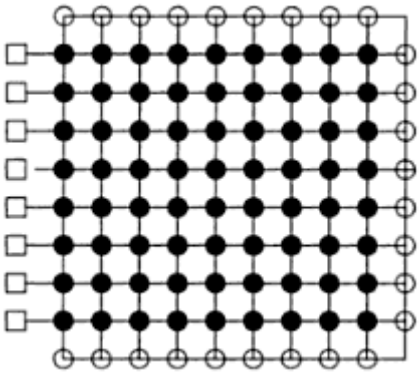

Figure 2.3: The normal stencil for the Laplacian is used for the grid points marked with filled circles and Dirichlet boundary conditions is applied to the grid points marked with open circles. For Neumann boundary conditions the values at the ghost points, open squares in the figure, are equal to the values to the right of the boundary points on the left side. The flux at the boundary is defined by the right hand side.For periodic boundary conditions this values are equal to the values to the left of the boundary grid points at the right side of the domain. Source Trottenberg et al. (2001).

Periodic boundary conditions can be implemented by setting the values at the boundary on one side of the domain equal to the values on the opposite side of the domain. Ghost points, which have the same values as the points just inside the domain on the opposite side of the domain see Figure 2.3, are used to calculate the values on the left side of the domain. The values at the right boundary are handled as Dirichlet boundary points when the values on the left side are known.

2.2.5 Higher Order Schemes for the Laplacian

The Laplace operator is a differential operator is used in the modelling of many physical problems such as wave propagation, heat diffusion and fluid mechanics. The standard 5-point stencil presented for the Laplacian has a truncation error of $\mathcal{O}(h^2)$. Stencils of higher order accuracy have been found. Numerical implementations favour using only the neighbouring nodes to approximating the derivatives, such stencils are called compact stencils.

An example of a compact stencil for the Laplacian with fourth order accuracy, $\mathcal{O}(\Delta x^4)$, is the Mehrstellen discretization, see Trottenberg et al. (2001). For the steady state problem in two dimensions it yields the following stencil

$$\Delta_9 T = -Rq \tag{2.21}$$

$$\text{where } \Delta_9 = \frac{1}{6h^2}\begin{bmatrix}1 & 4 & 1\\ 4 & -20 & 4\\ 1 & 4 & 1\end{bmatrix}, \text{and } R = \frac{1}{12}\begin{bmatrix}0 & 1 & 0\\ 1 & 8 & 1\\ 0 & 1 & 0\end{bmatrix}$$

Notice that the stencil has a correction for the right hand side. The Mehrstellen discretization in three dimensions is given by the following stencil

$$\Delta_{19} T = -Rq \tag{2.22}$$

$$\text{where } \Delta_{19} = \frac{1}{6h^2}\left[\begin{bmatrix}0 & 1 & 0\\ 1 & 2 & 1\\ 0 & 1 & 0\end{bmatrix}, \begin{bmatrix}1 & 2 & 1\\ 2 & -24 & 2\\ 1 & 2 & 1\end{bmatrix}, \begin{bmatrix}0 & 1 & 0\\ 1 & 2 & 1\\ 0 & 1 & 0\end{bmatrix}\right]$$

$$\text{and } R = \frac{1}{12}\begin{bmatrix}0 & 1 & 0\\ 1 & 6 & 1\\ 0 & 1 & 0\end{bmatrix}$$

Δ_{19} is a stencil of size 3x3x3 here each of the 3x3 matrixes is a plane in the stencil. Other stencils that were tested in 3 dimensions see Table 2.1.

	Stencil
Hale (2008)	$\begin{bmatrix}1/48 & 1/8 & 1/48\\ 1/8 & 5/12 & 1/8\\ 1/48 & 1/8 & 1/48\end{bmatrix}, \begin{bmatrix}1/8 & 5/12 & 1/8\\ 5/12 & -25/6 & 5/12\\ 1/8 & 5/12 & 1/8\end{bmatrix}, \begin{bmatrix}1/48 & 1/8 & 1/48\\ 1/8 & 5/12 & 1/8\\ 1/48 & 1/8 & 1/48\end{bmatrix}$
Patra et. al (2005)	$\begin{bmatrix}1/30 & 1/10 & 1/30\\ 1/10 & 7/15 & 1/10\\ 1/30 & 1/10 & 1/30\end{bmatrix}, \begin{bmatrix}1/10 & 7/15 & 1/10\\ 7/15 & -64/15 & 7/15\\ 1/10 & 7/15 & 1/10\end{bmatrix}, \begin{bmatrix}1/30 & 1/10 & 1/30\\ 1/10 & 7/15 & 1/10\\ 1/30 & 1/10 & 1/30\end{bmatrix}$
Finite element	$\begin{bmatrix}1/6 & 1/3 & 1/6\\ 1/3 & 0 & 1/3\\ 1/6 & 1/3 & 1/6\end{bmatrix}, \begin{bmatrix}1/3 & 0 & 1/3\\ 0 & -16/3 & 0\\ 1/3 & 0 & 1/3\end{bmatrix}, \begin{bmatrix}1/6 & 1/3 & 1/6\\ 1/3 & 0 & 1/3\\ 1/6 & 1/3 & 1/6\end{bmatrix}$

Table 2.1: Higher order stencils for the Laplacian.

2.2.6 Matrix Properties

Property	**Description**
Positive definite	A matrix A is positive definite if $x^T Ax > 0$, for all nonzero vectors x. All eigenvalues are positive.
Diagonally dominant	A matrix is diagonally dominant if $\left\|a_{jj}\right\| \geq \sum_{\substack{i=1\\ i\neq j}}^{i=n} \left\|a_{ij}\right\| \quad \text{for } j = 1,\ldots,n$ Where a_{ij} are the values in the matrix A. The matrix is strictly diagonally dominant if $\left\|a_{jj}\right\| > \sum_{\substack{i=1\\ i\neq j}}^{i=n} \left\|a_{ij}\right\| \quad \text{for } j = 1,\ldots,n$
Band matrix	All non-zero entries in the matrix are confined to diagonal bands in the matrix.
Symmetric	A matrix is symmetric if $A = A^T$.
Sparse	The matrix is mostly populated by zeros.

Table 2.2: Properties of matrices that arise when solving partial differential equations using finite difference.

Linear sets of equations must be solved when applying implicit schemes to partial differential equations. The coefficient matrices in the set of linear equations that arise from finite difference approximations of partial differential equations usually have some useful properties. Some of these properties are listed in Table 2.2. Linear sets of equations that have coefficient matrices with these properties can be solved more efficiently than linear sets of equations were the coefficient matrix is a full matrix.

3 Direct Methods

There are a large number of different methods that can be used to solve linear sets of equations. There are two main types of methods; the direct methods discussed here and the iterative methods which are discussed in the subsequent chapter. Direct methods use a finite number of operations to find a solution to a finite set of linear equations. In theory these methods would find an exact solution to a set of linear equations, provided that such a solution exists for the equations. However, round off errors will always arise when the methods are implemented numerically. Direct methods for solving linear sets of equations are based on Gaussian elimination.

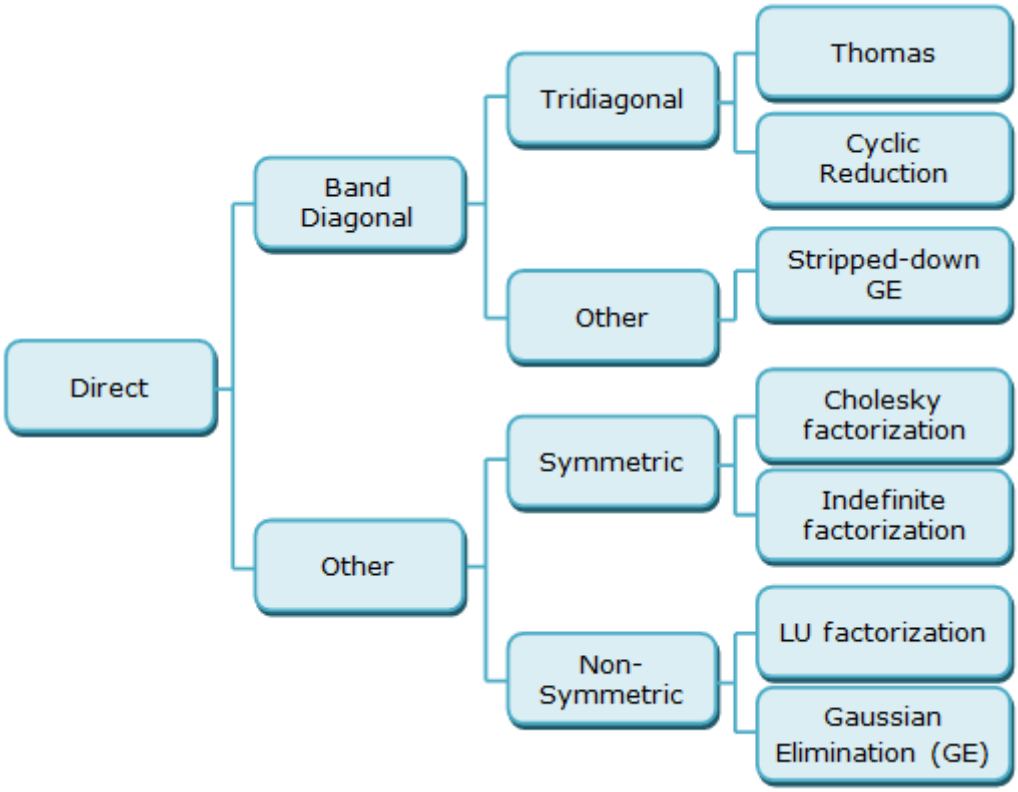

Figure 3.1: Decision tree for finding an algorithm that solves a set of linear equations. For band diagonal matrices a simplified version of Gaussian elimination can be used that only applies the row operations to the non-zero diagonals. For a tridiagonal matrix this approach is called Thomas algorithm. Cyclic reduction algorithm is a parallelizable algorithm that can be applied to all Toeplitz matrices, i.e. matrices where all elements on the same diagonal have the same value The Cyclic reduction algorithm is especially effective at solving tridiagonal matrices. Factorization is especially useful when several sets of equations with the same coefficient matrix are solved, as done when solving the transient diffusion equation (see section 2.2). Cholesky factorization can be carried out if the matrix is symmetric and positive definite. Indefinite factorization can be applied to symmetric non-positive definite systems. For non-symmetric systems LU factorization can be applied or Gaussian elimination on the full matrix can be carried out directly.

There is a variety of different direct solvers that can be used to solve linear sets of equations with coefficient matrices with different properties (see Figure 3.1). As discussed in section 2.2.6 the matrices that arise from the discretization of partial differential equations usually have the following properties; they are sparse, symmetric, positive definite, diagonally dominant and band diagonal. Band diagonal matrices have non-zero values only on diagonals within a certain range around the main diagonal. The total number of diagonals within the range is called the bandwidth of the banded matrix.

The efficiency of the algorithms depends on their complexity and their degree of parallelism. The complexity is the number of operations that is needed to solve the set of equations. The degree of parallelism is set by the number of processes the algorithm can be divided into, which can run simultaneously. Only cyclic reduction is directly parallelizable of the algorithms presented in Figure 3.1. Iterative methods are in general better suited for parallel processing.

3.1 Gaussian Elimination

Gaussian elimination uses elementary row operations to reduce the matrix to its reduced row echelon form. The algorithm has two main steps. The first one is forward elimination, which reduces the matrix to an upper triangular matrix, i.e. echelon form. The second step is backward substitution which reduces the matrix to a reduced row echelon form. Both the forward elimination and the backward substitution are done with elementary row operations.

Gaussian elimination is always stable for matrices that are diagonally dominant or positive definite. The algorithm is generally stable for other matrices as well, when used in combination with partial or full pivoting. Full pivoting means to interchange rows and columns such that the largest absolute values are found at the pivot positions. Only the rows are interchanged when using partial pivoting. The pivot positions correspond to the positions of the leading ones in the echelon form. It is preferable to have large values at these positions to avoid division by numbers that are close to zero.

3.2 Methods for Band Diagonal Matrices

The values at the nodes in finite difference discretization of partial differential equations are only coupled to the values at the neighbouring nodes. This result in banded diagonal matrices when they are discretized, see section 2.2. Discretization using finite differences of one dimensional problems results in matrices that are tridiagonal. Such systems can be solved with the Thomas algorithm and the cyclic reduction algorithm presented in the following two subsections.

Discretization of systems in two and three dimensions results in banded matrices that tend to be sparse, i.e. only a few of the diagonals contain non-zero values. Discretization of the Laplacian in two dimensions on a system with $n \times n$ nodes using a five-point stencil results in a matrix with a bandwidth of $2n+1$ where only five of the bands contain non-zero values. Applying Gaussian elimination or factorizing matrices does, however, fill up many of the diagonals with only zeros.

3.2.1 Thomas Algorithm

The algorithm is a stripped down version of Gaussian elimination, it has a complexity of $2n$ for tridiagonal matrices. The algorithm consists of one forward sweep which eliminates the lower diagonal, and the second step is backward substitution which solves the system. The algorithm will always succeed if the matrix is diagonally dominant. Only the non-zero diagonals are stored to save space in the computer memory. The algorithm is strictly sequential, i.e. all operations depend on values calculated in the previous step.

3.2.2 Cyclic Reduction

The cyclic reduction algorithm has roughly the same complexity as the Thomas algorithm for tridiagonal systems. The algorithm is applicable to all Toeplitz matrices, but the focus here is on tridiagonal matrices. For more general systems see Gander (1997).

The advantage of using cyclic reduction is that the algorithm is relatively easy to parallelize. This makes it suitable for processor architectures with several processing cores, such as the graphics processor architecture.

The idea is to eliminate the unknowns that have odd-numbered indices, regroup and repeat the process until there is one unknown left. Find this value and then retrace to find the other unknowns. Consider a set of equations, $A\boldsymbol{x} = \boldsymbol{b}$, of the following form

$$\begin{bmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \\ & e_3 & \ddots & \ddots & \\ & & \ddots & \ddots & f_{n-1} \\ & & & e_n & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \tag{3.1}$$

The outline of the cyclic reduction algorithm is presented here, with $n = 7$ as an example.

1. Permute the matrix with the following permutation

$$\Pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 5 & 7 & 2 & 4 & 6 \end{pmatrix} \tag{3.2}$$

This means that rows are interchanged such that all odd numbered rows are placed at the top of the matrix, and all even numbered rows are placed at the bottom. The same is done for the columns such that the coefficients that correspond to the unknowns with odd numbered indices are to the left in the matrix and even numbered ones are to the right.

This yields the following matrix,

$$A_\Pi = \begin{bmatrix} d_1 & & & & f_1 & & \\ & d_3 & & & e_3 & f_3 & \\ & & d_5 & & & e_5 & f_5 \\ & & & d_7 & & & e_7 \\ e_2 & f_2 & & & d_2 & & \\ & e_4 & f_4 & & & d_4 & \\ & & e_6 & f_6 & & & d_6 \end{bmatrix} \tag{3.3}$$

2. The permuted matrix can then be row reduced such that all elements in the lower-left quadrant is eliminated; which changes the values in the lower-right quadrant and generates new elements on the diagonal just over and under the main diagonal. The linear set of equations is of the following form after the row reduction

$$\begin{bmatrix} d_1 & & & & f_1 & & \\ & d_3 & & & e_3 & f_3 & \\ & & d_5 & & & e_5 & f_5 \\ & & & d_7 & & & e_7 \\ & & & & d_1^{(1)} & f_1^{(1)} & \\ & & & & e_2^{(1)} & d_2^{(1)} & f_2^{(1)} \\ & & & & & e_3^{(1)} & d_3^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_5 \\ x_7 \\ x_2 \\ x_4 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_3 \\ b_5 \\ b_7 \\ b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix} \tag{3.4}$$

Where the values l_i, $m_i, d_i^{(1)}, e_i^{(1)}$ and $f_i^{(1)}$ are given by the following expressions when there is an odd number of unknowns

$$l_i = \frac{e_{2i}}{d_{2i-1}}, \qquad m_i = \frac{f_{2i}}{d_{2i+1}}, \qquad d_i^{(1)} = d_{2i} - l_i f_{2i-1} - m_i e_{2i+1}$$

where $i = 1,2, \dots, \frac{n-1}{2}$.

$$e_{i+1}^{(1)} = -e_{2i+1} l_{i+1}, \qquad f_i^{(1)} = -f_{2i+1} m_i$$

where $i = 1,2, \dots, \frac{n-1}{2} - 1$

The row reductions yields the following corrections to the values of the right hand side, whose indices are even numbers

$$b_i^{(1)} = b_{2i} - l_i b_{2i-1} - m_i b_{2i+1} \tag{3.5}$$

Where $i = 1,2, \dots, \frac{n-1}{2}$. The set of equations for the unknowns with even numbered indices can now be solved independently of the unknowns with odd numbered indices. The set of equations for the unknowns with even numbered indices has the same form as the original set of equations, see (3.1).

3. Half of the unknowns in the set of equations for the unknowns with even numbered indices can now be removed since the coefficient matrix is tridiagonal. This can be done by repeating points 1 and 2 for this set of equations. This will in turn result in a matrix of the same form and the process of removing half of the unknowns can be repeated arbitrary many times.
4. Solve for the unknowns in the final system of equations.
5. Assuming that x_2, x_4 and x_6 are known the unknowns with odd indices can be found with the following equations

$$x_1 = b_1 - f_1 x_2$$

$$x_i = b_i - f_i x_{i+1} - e_i x_{i-1}$$

$$x_n = b_n - e_n x_{n-1}$$

Where $i = 1,3, \dots, n$.

3.3 Factorization

Factorization of a matrix is often used in numerical linear algebra; it means to write the matrix as a product of two or more matrices. Each of the matrices in the factorization has a useful structure that can be exploited to make more efficient algorithms to solve the linear set of equations. Factorization is especially useful when several sets of equations with the same coefficient matrix are solved.

Two commonly used factorizations are presented here; LU factorization and Cholesky factorization. Both of them factorize the matrix into an upper and a lower triangular matrix, called L and U respectively. Instead of solving the original equation, $A\boldsymbol{x} = \boldsymbol{b}$, the following set of equations is solved

$$L\boldsymbol{y} = \boldsymbol{b} \tag{3.6}$$

$$U\boldsymbol{x} = \boldsymbol{y} \tag{3.7}$$

Solving this set of equations is preferable since U and L are triangular matrices and therefore efficient to solve.

3.3.1 LU Factorization

The complexity of finding the LU factorization and then solve the set of equations is higher than applying Gaussian elimination directly. The LU factorization of a matrix of size $n \times n$ requires about $2n^3/3$ operations, and equation (3.6) and (3.7) can be solved with $2n^2$ operations when the LU factorization is known. Applying Gaussian elimination directly solves the system of equations with approximately $2n^3/3$ operations. The number of operations used by each algorithm has been obtained from Lay (2006).

The solution obtained by using LU factorization is often more accurate since the factorization accumulate less round off errors. The L and U matrix is quite likely to be sparse if the matrix A itself is sparse, but the inverse of A is likely to be dense. LU factorization is far more efficient than direct Gaussian elimination if this is the case.

The upper triangular matrix is found by reducing the matrix A to an echelon form with row operations. The lower triangular matrix is formed such that the same sequence of row operations reduces the lower triangular matrix to the identity matrix. The factorization is unique if the values on the diagonal for either the upper or the lower triangular matrix are all ones.

A LU factorization exist for a matrix if and only if the all the principal minors are non-zero. A minor is the determinant of some smaller matrix obtained by removing one or more rows and columns from the matrix. The principal minors of a matrix are the determinants of the matrices formed by removing the i'th column and row, for $i = 1,2,\dots,n$. A LU factorization may exist for a singular matrix.

3.3.2 Cholesky Factorization

Cholesky factorization can be applied to a matrix A, of size $n \times n$, if it is symmetric and positive definite. The upper triangular matrix U is equal to the conjugate transpose of the lower triangular matrix L for the Cholesky factorization. The number of operations

needed to calculate the Cholesky factorization is about $n^3/3$, that is half of the number operations needed to find the LU factorization or solving the system using Gaussian elimination.

The Cholesky factorization has the same advantages as the LU factorization when it comes to limiting round off errors and keeping the matrix sparse. The Cholesky factorization is more efficient than the LU factorization and it requires that less data is stored in the computer memory. The Cholesky factorization is always unique.

3.4 Limitations of Direct Methods

The complexity for the algorithms described in this chapter is summarized in Table 3.1. The number of operations needed to solve the equation sets increases rapidly as the size of the coefficient matrix increases. Iterative methods are usually the preferred choice for large systems where direct methods would be to computationally expensive. The system can be so large that it would be impossible to fit data needed in the computer memory or solve the system in a reasonable amount of time.

	FULL MATRIX	
Gaussian elimination	$2n^3/3$	
	TRIDIAGONAL MATRIX	
Thomas algorithm	$2n$	
Cyclic reduction	$\sim 2.7 \cdot 2n$	
	FACTORIZATION	
	Factorization	Solving equation (3.6) and (3.7)
LU factorization	$2n^3/3$	$2n^2$
Cholesky factorization	$n^3/3$	$2n^2$

Table 3.1: Number of operations needed to solve a system of equations with n unknowns. e.g. Gander (1997) and Lay (2006).

The direct methods presented here, with the exception of cyclic reduction, are traditionally perceived as sequential. Sequential algorithms are inefficient on processors with several cores and they tend to accumulate large round-off errors, which can make the final solution unusable for large systems.

However, in later years parallel implementations of direct methods has been developed. Packages that includes parallel implementations of factorizations and direct solvers are available, see for example,

- a MUltifrontal Massively Parallel sparse direct Solver, MUMPS
- superLU
- PETSc
- PARADISO

4 Iterative Methods

Iterative methods can be used to solve system of equations that arise from finite difference approximation of partial differential equations, which have large and sparse coefficient matrices. In the previous chapter it was shown that as the system become increasingly large and complex it is no longer feasible to solve the systems with direct methods due to limitations of computer memory and the number of arithmetic operations that must be carried out.

Another incentive for using iterative methods is that they are far easier to implement on parallel computers. This is becoming increasingly important as inexpensive powerful parallel computers become broadly available.

Iterative methods aim at finding a solution to a set of linear equations, $A\boldsymbol{x} = \boldsymbol{b}$, by finding successive approximations for its solution, until a sufficiently accurate one is found. As for the direct methods there are a large number of iterative methods to choose from, see Figure 4.1.

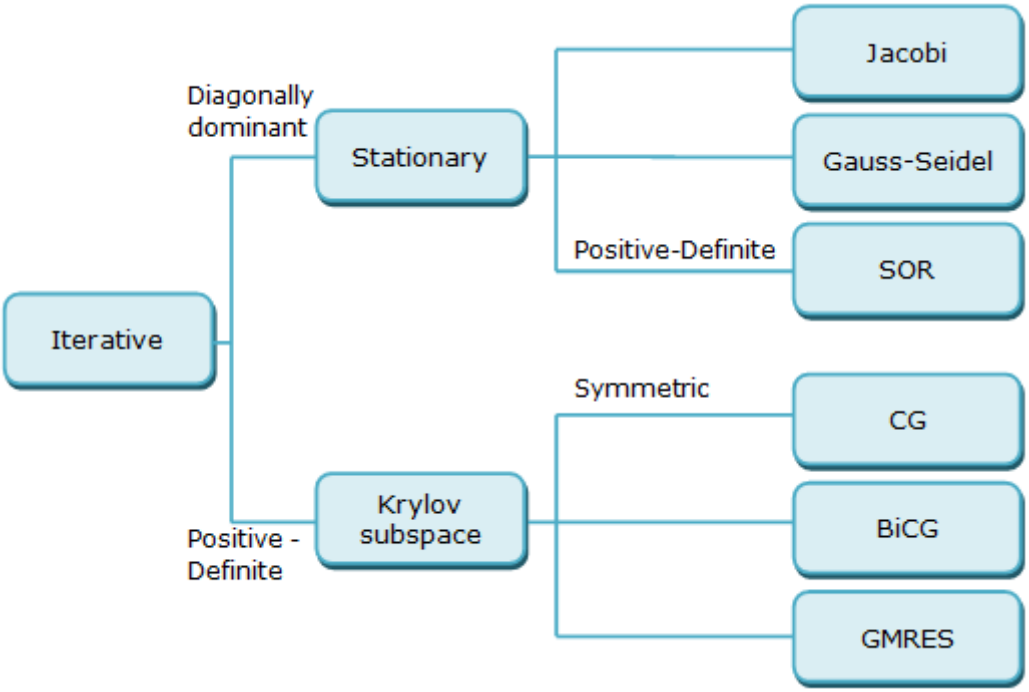

Figure 4.1: There are two main branches of iterative methods. The first is the stationary or classic methods, which include Jacobi, Gauss-Seidel and the Successive OverRelaxation method (SSOR). The other branch is the Krylov subspace methods, which include the Conjugate Gradient method (CG), the BiConjugate Gradient (BiCG) and Generalized Minimal RESidual method (GMRES).

There are two main types of iterative methods, which are the stationary or classic iterative methods and Krylov subspace methods. All the stationary methods that are presented in Figure 4.1 are described in the following subsection. There are several Krylov subspace methods as well, but the focus will be on the Conjugate Gradient (CG) method.

4.1 Classic Iterative Methods

The classic iterative methods are built on the principle that the matrix A can be written as a sum of other matrices. There are several ways to divide the matrix; two of them are the origin of the Jacobi and the Gauss-Seidel method. The successive overrelaxation method is an improved version of the Gauss-Seidel method. The classic iterative methods do in general have a quite low convergence rate compared to the Krylov subspace methods. They do, however, smooth the error efficiently and this makes them an important part of the Multigrid algorithm that is presented in the subsequent chapter. The smoothing effect of the classic iterative methods is elaborated on in subsection 4.1.6.

4.1.1 Jacobi Method

In the Jacobi method the matrix A is divided into two matrices D and E such that $D + E = A$. D is a diagonal matrix with the same entries as A has on the main diagonal, and E has zeros on the diagonal and the off diagonal entries are equal to the rest of the entries in A. Applying this to the to set of linear equations we find that

$$A\boldsymbol{x} = \boldsymbol{b}$$

$$(D + E)\boldsymbol{x} = \boldsymbol{b}$$

$$D\boldsymbol{x} = -E\boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{x} = -D^{-1}E\boldsymbol{x} + D^{-1}\boldsymbol{b}$$

$$\boldsymbol{x} = B\boldsymbol{x} + D^{-1}\boldsymbol{b} \tag{4.1}$$

Where $B = -D^{-1}E$. This expression may be used to find an iterative method based on recursion,

$$\boldsymbol{x}^{(i+1)} = B\boldsymbol{x}^{(i)} + D^{-1}\boldsymbol{b} \tag{4.2}$$

Where $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(i+1)}$ are successive approximations for the solution of the linear set of equations. From equation (4.2) we find that the exact solution, $\boldsymbol{x}$, is a stationary point, i.e. if $\boldsymbol{x}^{(i)}$ is equal to the exact solution of the equation set then $\boldsymbol{x}^{(i+1)}$ will be equal to the exact solution as well. Solvers based of this principle, such as the Jacobi and the Gauss-Seidel methods, are therefore called stationary methods.

Equation (4.2) may be written in component form, which yields the following expression

$$x_k^{(i+1)} = -\frac{1}{a_{kk}} \sum_{\substack{j=1 \\ j \neq k}}^{n} a_{kj}\, x_j^{(i)} + \frac{1}{a_{kk}} b_k \tag{4.3}$$

Where a_{kk} denotes the entries in the matrix A, $x_k^{(i)}$ and b_k are the entries in the approximation vector $\boldsymbol{x}^{(i)}$ and the right hand side vector $\boldsymbol{b}$ respectively.

The error is defined as the difference between the approximation of the solution and the exact solution

$$\boldsymbol{e}^{(i)} = \boldsymbol{x}^{(i)} - \boldsymbol{x} \tag{4.4}$$

The Jacobi method can be analysed further using this definition of the error. From equation (4.1) the following equation is found

$$\begin{aligned} \boldsymbol{x}^{(i+1)} &= B(\boldsymbol{x} + \boldsymbol{e}^{(i)}) + D^{-1}\boldsymbol{b} \\ \boldsymbol{x}^{(i+1)} &= B\boldsymbol{x} + B\boldsymbol{e}^{(i)} + D^{-1}\boldsymbol{b} \\ \boldsymbol{x}^{(i+1)} &= \boldsymbol{x} + B\boldsymbol{e}^{(i)} \\ \boldsymbol{e}^{(i+1)} &= B\boldsymbol{e}^{(i)} \end{aligned} \tag{4.5}$$

Equation (4.5) shows that each iterative step only affects the error, i.e. the incorrect part of the approximation. Whether or not the Jacobi method converges to the exact solution will therefore depend on the properties of the iteration matrix, B. To analyse how the operation in equation (4.5) affects the error we can decompose the error vector using the eigenvectors of B, provided that B which is of size $n \times n$ has n independent eigenvectors. The eigenvectors are denoted $\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_n$ with corresponding eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_n$. Rewriting equation (4.5) yields the following expression

$$\boldsymbol{e}_{(i+1)} = B(k_1\boldsymbol{v}_1 + k_2\boldsymbol{v}_2 + \cdots + k_n\boldsymbol{v}_n \tag{4.6}$$

Where $k_1, k_2, \dots, k_n$ are constants. The eigenvalues satisfies the eigenvalue equation, $A\boldsymbol{v_i} = \lambda_i\boldsymbol{v_i}$. Multiplying the equation with a scalar gives $kA\boldsymbol{v} = k\lambda\boldsymbol{v}$. Adding this result to equation (4.6) yields the following expression

$$\boldsymbol{e}_{(i+1)} = \lambda_1 k_1\boldsymbol{v}_1 + \lambda_2 k_2\boldsymbol{v}_2 + \cdots + \lambda_n k_n\boldsymbol{v}_n \tag{4.7}$$

From this equation we find that the method will converge to the exact solution if all eigenvalues are less than unity, i.e. the spectral radius of B is less than unity. The spectral radius is defined as

$$\rho(B) = \max|\lambda_i| \tag{4.8}$$

λ_i are the eigenvalues of B. This gives the condition for whether or not the Jacobi algorithm will converge, but does not specify rate of convergence. The Jacobi method will converge for linear systems of equations with coefficient matrices that are diagonally

dominant; this is usually the case for the set of equation that arises from finite difference and finite element discretizations of partial differential equations. Finding the rate of convergence requires a more detailed study. The convergence of the classic iterative methods is discussed in section 4.1.4.

4.1.2 Gauss-Seidel Method

In the Jacobi method all entries in the approximation are updated based on the values in the previous approximation, see equation (4.3). The Gauss-Seidel method on the other hand uses the previously updated values in the current approximation to find the rest of them.

This corresponds to dividing the matrix A into three matrices, D, F and G. Where D matrix is the same diagonal matrix as in the Jacobi method, which contains the diagonal entries in A. F is a lower triangular matrix and the G matrix is an upper triangular matrix containing the entries A has below and above the diagonal respectively. Carrying out the same derivations as for equation (4.2) for this splitting of the matrix A leads to the matrix form of the Gauss-Seidel method

$$\boldsymbol{x}^{(i+1)} = -(D+F)^{-1} G \boldsymbol{x}^{(i)} + (D-F)^{-1} \boldsymbol{b} \tag{4.9}$$

Writing it out in component form this yields

$$\boldsymbol{x}_k^{(i+1)} = -\frac{1}{a_{kk}} \sum_{j=1}^{k-1} a_{kj}\, \boldsymbol{x}_j^{(i+1)} - \frac{1}{a_{kk}} \sum_{j=k+1}^{n} a_{kj}\, \boldsymbol{x}_j^{(i)} + \frac{1}{a_{kk}} b_k \tag{4.10}$$

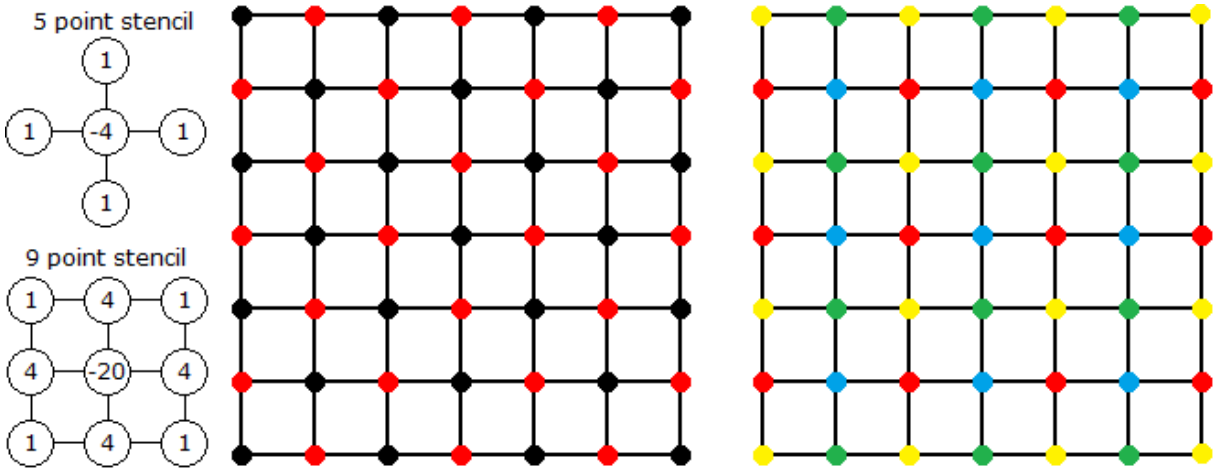

Figure 4.2: In the linear sets of equations that arise from the finite difference discretization of partial differential equations, such as the heat diffusion equation, the function values in the nodes are only coupled to the function values in their neighbouring nodes. In section 2.2.5 different stencils for the Laplacian were discussed. The 5 point and the 9 point stencil are visualized on the left hand side figure. For the 5 point stencil red-black ordering (see middle figure) can be used. The updated values in every grid point marked with red are found first then all values in the grid points marked with black are found. For the 9 point stencil four colour ordering (see figure on the right hand side) can be used.

In equation (4.10) the entries are solved for in the order they are numbered, called lexicographic ordering. In principal it is possible to solve for the entries in any order. In the equation sets formed by finite difference discretization of partial differential equations, such as the diffusion equation, the unknowns are found based on the values of the closest neighbours. This observation leads to the idea of the so called red-black ordering, see Figure 4.2. For the 5 point stencil for the Laplacian all grid points marked in red are uncoupled with the other red grid points and grid points marked in black are uncoupled with each other as well. This means that the values in the red grid points can be solved for first and then all values in the black grid points can be solved using the updated values in the red grid points. Solving for all red grid points can be done in parallel since they are independent of each other and the same goes for the black ones. For the 9 point stencil for the Laplacian four colour-ordering must be used to get the same effect.

4.1.3 Successive Overrelaxation Method

An improvement to the Gauss-Seidel method can be made by anticipating future corrections to the approximation by making an overcorrection at each iterative step. The method is called the successive overrelaxation method (SOR). The SOR method was the standard iterative method until the 1970s. This method is based on the matrix splitting

$$\omega A = (D + \omega F) + (\omega G + (\omega - 1)D) \tag{4.11}$$

The matrices D, F and G are the same as for the Gauss-Seidel method, and ω is the successive overrelaxation parameter. This matrix splitting results in the following iterative method

$$(D + \omega F)\boldsymbol{x}^{(i+1)} = -(\omega G + (\omega - 1)D)\boldsymbol{x}^{(i)} + \omega \boldsymbol{b} \tag{4.12}$$

This is equivalent to the following expression

$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{GS} + (1 - \omega)\boldsymbol{x}^{(i)} \tag{4.13}$$

$\boldsymbol{x}^{GS}$ is the approximation found with the Gauss-Seidel method, see equation (4.10).

To obtain convergence for symmetric, positive definite matrices the value of the overrelaxation parameter must be in the range $0 < \omega < 2$. For values smaller than one it is called underrelaxation. Only values larger than one result in a performance gain. There is an optimal value for the relaxation parameter where the rate of convergence is at its maximum. The weak point of the method is finding the optimal value of the relaxation parameter since the much of the performance gain is lost if the value is not chosen form a narrow window around the optimal value. Methods for finding the optimal values are discussed in section 4.1.5.

4.1.4 Convergence

The convergence rate indicates the number of iterations that is needed for an iterative method to find an approximation of the solution that is within a certain range of the exact solution. Both the rate of convergence and the choice of optimal overrelaxation pa-

rameter are heavily dependent on finding the spectral radius of the iteration matrix, or at least an upper bound for it. This section starts with a general discussion the convergence rates. The convergence is analysed for the steady state diffusion equation in one dimension with Dirichlet boundary conditions. Discretization of this equation is shown in section 2.2.1.

All the iterative methods discussed in the previous sections are on the form

$$\boldsymbol{x}^{(i+1)} = B\boldsymbol{x}^{(i)} + \widetilde{\boldsymbol{b}} \tag{4.14}$$

The matrix B is the iteration matrix, and $\widetilde{\boldsymbol{b}}$ is a vector. The iteration matrices for the three methods are given in table Table 4.1.

Method	Iteration matrix
Jacobi	$-D^{-1}E$
Gauss-Seidel	$-(D+F)^{-1}G$
SOR	$-(D+\omega F)^{-1}(\omega G+(\omega-1)D)$

Table 4.1: The iteration matrices of the classic iterative methods.

The iterative methods will converge if the spectral radius of the iterative matrix is less than unity; this fact is well documented, see for example Saad (2003). This was also shown in section 4.1.1 for the Jacobi method.

The number of iterations needed to reduce the error by a factor 10^{-d}, where d is a positive integer, is used to compare the efficiency of the different methods, i.e.

$$\frac{\left\|e^{(m)}\right\|}{\left\|e^{(0)}\right\|} < 10^{-d} \tag{4.15}$$

$e^{(0)}$ is the initial error and m is the smallest integer that satisfies the inequality. Equation (4.5) yields the following relation between error at the m^{th} iteration and the initial error

$$\left\|e^{(m)}\right\| = \left\|B^m e^{(0)}\right\| \tag{4.16}$$

Using that $\|AB\| \leq \|A\|\|B\|$ for any matrix A and B yields the following inequality

$$\left\|e^{(m)}\right\| \leq \ \|B\|^m \left\|e^{(0)}\right\| \tag{4.17}$$

For any matrix norm the following inequality is upheld see Saad (2003)

$$\rho(B) \leq \|B\| \tag{4.18}$$

Where $\rho(B)$ is the spectral radius of the matrix B. The condition in equation (4.15), (4.17) and equation (4.18) gives the following equation

$$[\rho(B)]^{(m)} \leq \ 10^{-d} \tag{4.19}$$

Solving for the number of iterations yields

$$m \geq -\frac{d}{\log_{10}[\rho(B)]} \tag{4.20}$$

The quantity $\log_{10}[\rho(B)]$ is called the asymptotic convergence rate. As $\rho(B)$ approaches 1 the convergence deteriorates.

The iteration matrix of the Jacobi method can be rewritten as a sum of the identity matrix, I, and the matrix A from the linear set of equations that is solved, $A\boldsymbol{x} = \boldsymbol{b}$,

$$B = -D^{-1}E = I + \frac{1}{2}A \tag{4.21}$$

All eigenvectors of the identity matrix equal to one. This means that the eigenvalues of B are given by

$$\lambda(B) = \frac{1}{2}\lambda(A) + 1 \tag{4.22}$$

This means that finding the eigenvectors of the iteration matrix is really a question of finding the eigenvalues of the A matrix.

For steady state heat diffusion in a homogeneous material in one dimension, where heat capacity, conductivity, grid spacing and density are all set to unity for simplicity, the matrix A is of the form

$$A = \begin{bmatrix} -2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix} \tag{4.23}$$

The eigenvalues of matrix A, which is of size $n \times n$, is given by the following equation

$$\lambda(A) = -2 - 2\cos\left(\frac{k\pi}{n+1}\right) \tag{4.24}$$

Where $k = 1,2,\ldots,n$. Using equation (4.22) we find that the eigenvalues of B are given by the following equation,

$$\lambda(B) = -\cos\left(\frac{k\pi}{n+1}\right) \tag{4.25}$$

The spectral radius is given by the largest absolute value of the eigenvalues,

$$\rho(B) = \max|-1\cos\left(\frac{k\pi}{n+1}\right)| = -1\cos\left(\frac{n\pi}{n+1}\right) \tag{4.26}$$

The asymptotic convergence rate is $\log_{10}\left[-1\cos\left(\frac{n\pi}{n+1}\right)\right]$. Number of iterations needed to achieve the predetermined level of convergence can now be found by the following equation,

$$m \geq -\frac{d}{\log_{10}\left[-1\cos\left(\frac{n\pi}{n+1}\right)\right]} \tag{4.27}$$

It is shown that the spectral radius of the iteration matrix of Gauss-Seidel method is simply the square of the spectral radius of the iteration matrix of the Jacobi method, see Young (1971), Stoer et al. (2002) and Varga (2000).

$$\rho_{GS} = \rho_{jacobi}^2 \tag{4.28}$$

The spectral radius of the SOR method is found in the following subsection since it is dependent on the choice of overrelaxation parameter.

4.1.5 Overrelaxation Parameter

The overelaxation parameter must be within a narrow range around the optimal value for the SOR algorithm to convergence considerably faster than the standard Gauss-Seidel algorithm. The optimal value for the overrelaxation parameter can be found based on the following analytical expression see Press et al. (2007)

$$\omega = \frac{2}{1+\sqrt{1-\rho_{GS}}} \tag{4.29}$$

The spectral radius of the SOR iteration matrix for the optimal choice of ω is given by the following expression,

$$\rho_{SOR} = \frac{\rho_{GS}}{\left(1+\sqrt{1-\rho_{GS}}\right)^2} \tag{4.30}$$

The overrelaxation parameter can be found experimentally. Finding it numerically is often the only choice since it is not always possible to derive the spectral radius of the Gauss-Seidel iteration matrix analytically.

4.1.6 Smoothing Properties

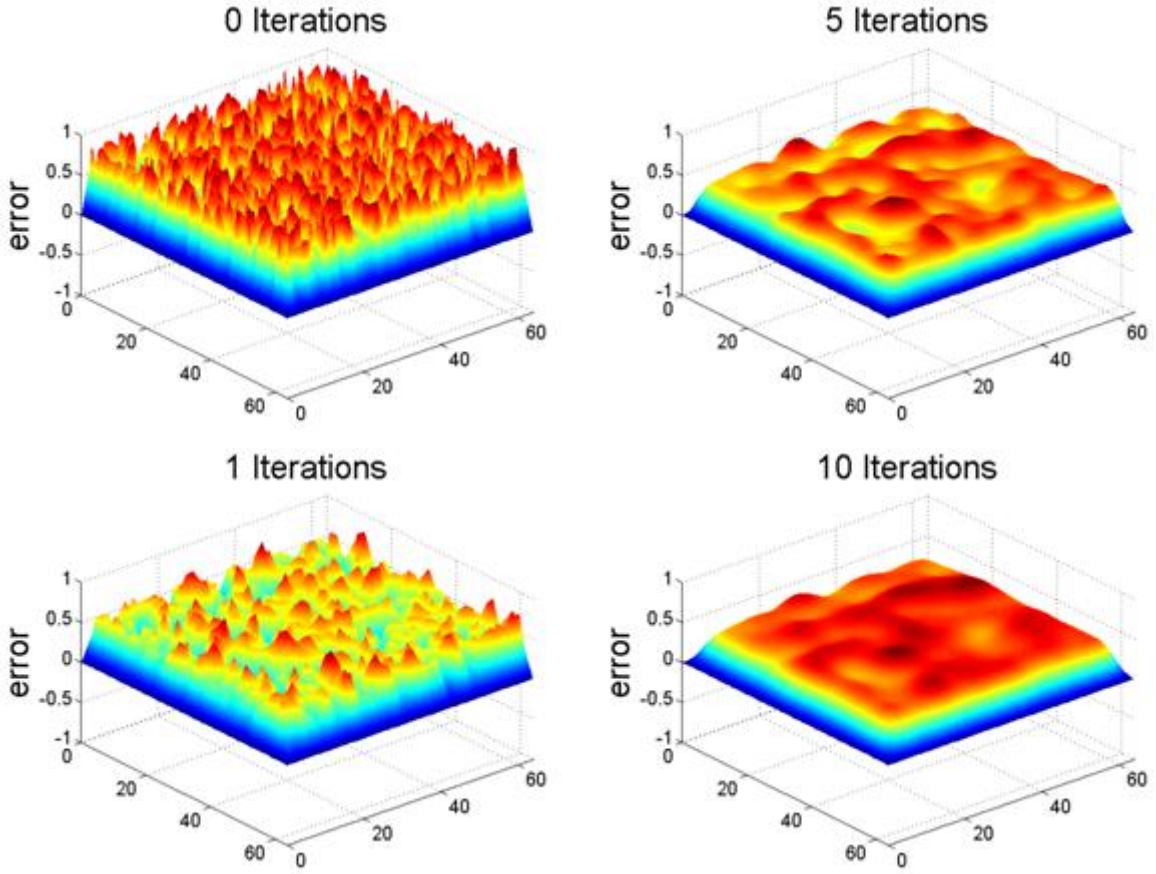

Figure 4.3: The image series shows how the error is smoothed as after certain numbers of iterations using the Gauss-Seidel algorithm with red-black ordering and a 5 point stencil for the Laplacian.

The classic iterative methods do not necessarily reduce the error of an approximation efficiently, but it does have a strong smoothing effect on it (see Figure 4.3). The smooth error term is well represented on a coarse grid. This property is exploited in the Multigrid algorithm where we solve for the error on the coarser grid to find a better approximation for the exact solution. It is clear from Table 3.1 that solving the equation on a coarser grid requires a substantially smaller number of operations.

4.1.7 Tests

For the Gauss-Seidel method convergence is tested for systems of different sizes, see Figure 4.4. The plot clearly shows that the rate of convergence deteriorates rapidly as the system size increases. It is this adverse affect on the convergence rate from the system size that can be avoided by using the multigrid algorithm.

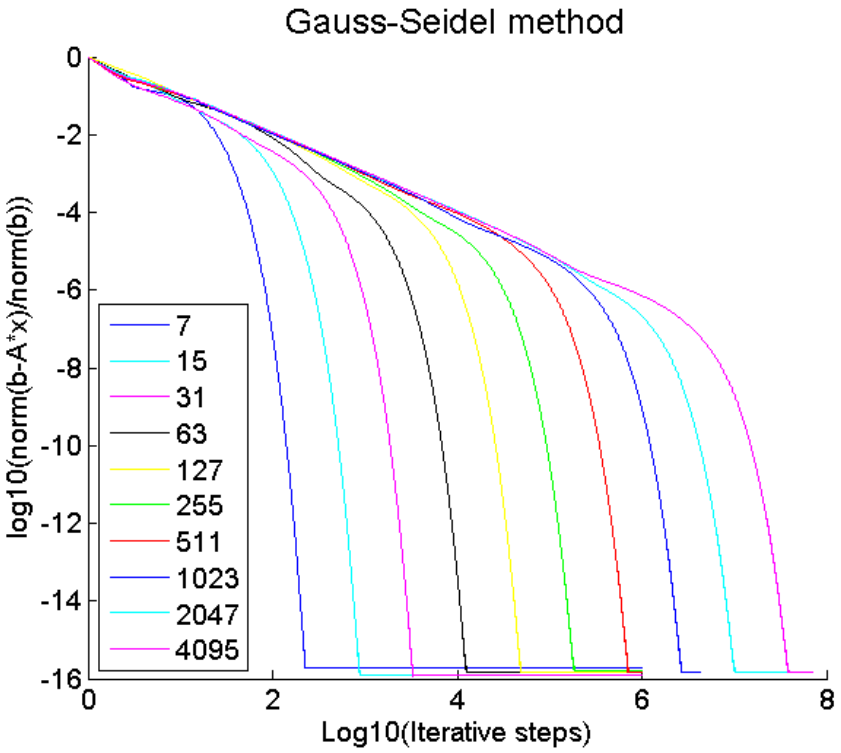

Figure 4.4: The number of iterative steps required to reach the maximum accuracy for the different grid sizes using the Gauss-Seidel method. Both the x- and the y-axis are plotted on logarithmic scales. The legend shows which colours corresponds to the different number of grid points in each spatial direction. The number of iterative steps needed to reach the stagnation point does increase with the number of grid points.

4.2 Krylov Subspace Methods

The Krylov subspace iterative methods are rated as one of the "*Top 10 Algorithms of the 20th Century*", see Cipra (2000) and Dongarra et al. (2000). As the other algorithms presented thus far these methods aim at solving equations of the form, $A\boldsymbol{x} = \boldsymbol{b}$, where A is a large and sparse matrix. The Krylov subspace is spanned by vectors formed by powers of the matrix A a being multiplied by the residual vector, $\boldsymbol{r}_0 = A\boldsymbol{x} - \boldsymbol{b}$,

$$\mathcal{K}_r(A, \boldsymbol{r}_0) = \text{span}\{\boldsymbol{r}_0, A\boldsymbol{r}_0, A^2\boldsymbol{r}_0, \dots, A^n\boldsymbol{r}_0\} \tag{4.31}$$

The most prominent algorithm of the Krylov subspace methods is the Conjugate Gradient algorithm, which can be applied to matrices that are symmetric and positive definite. A brief introduction to the idea behind this algorithm is presented in the following subsection. Generalizations of this method for non-symmetric matrices lead to the Generalized Minimum Residual (GMRES) method and BiConjugate Gradient method (BiCG), neither of which is covered in this thesis.

4.2.1 Conjugate Gradients

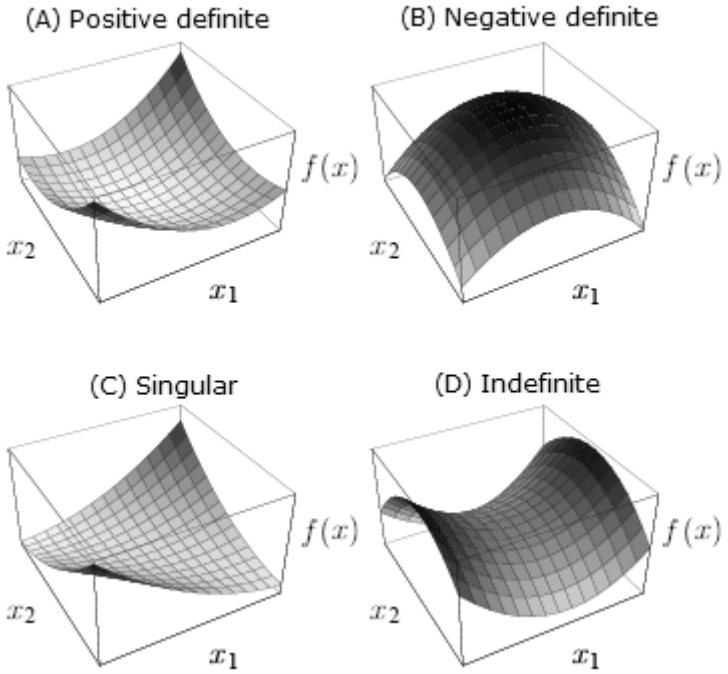

Figure 4.5: x_1 and x_2 indicates the possible values of the unknowns in a system with two unknowns. A) Shape of the quadratic form of a positive definite system. B) For a negative definite system is the shape of the quadratic form an "upside-down bowl". C) A singular (positive definite) matrix has an infinite number of solutions that are found in the bottom of the "valley" formed by the quadratic form. D) The solution of a linear set of equations with an indefinite coefficient matrix is at the saddle point of the quadratic form. Source Shewchuk (1994).

The Conjugate Gradient method (CG) method is built on the same ideas as the Steepest Decent (SD) method. As usual the aim is to solve an equation on the form, $A\boldsymbol{x} = \boldsymbol{b}$. For both the CG and SD method the matrix *A* must be positive definite or positive indefinite. The reason for this is that both methods exploit the shape of the quadratic form function

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{4.32}$$

c is some scalar. The quadratic function has its minimum value when $\boldsymbol{x}$ is the solution of the linear set of equations, $A\boldsymbol{x} = \boldsymbol{b}$, if the matrix A is positive definite. The quadratic form function is shaped as a "bowl" for positive definite systems, see Figure 4.5.

The SD method searches for the solution of the equation in the direction of the steepest decent, i.e. the direction that has the largest negative gradient. The gradient of the form function is given by

$$f(\boldsymbol{x})' = A\boldsymbol{x} - \boldsymbol{b} \tag{4.33}$$

For a symmetric matrix A. For the exact solution this gradient is zero. Each new approximation for the solution is found in the direction of the largest negative gradient in the SD method. Forming a line, called a search line, in the direction of the largest gradient the new approximation is found at the point of this line which has the smallest gradient. The gradient is orthogonal to the search line at the point where the gradient is at its minimum. This technique repeatedly applied until a sufficiently accurate approximation for the solution is found, but what is a sufficiently accurate approximation?

To find an answer to this question some definitions are needed. The error is still defined as

$$\boldsymbol{e}^{(i)} = \boldsymbol{x}^{(i)} - \boldsymbol{x} \tag{4.34}$$

Where $\boldsymbol{e}^{(i)}$ is the error and $\boldsymbol{x}^{(i)}$ is the approximate solution at some iteration i, and $\boldsymbol{x}$ is the exact solution. In addition to the error the definition of the residual is needed

$$\boldsymbol{r} = A\boldsymbol{x}^{(i)} - \boldsymbol{b} \tag{4.35}$$

Notice that the residual points in the direction that is opposite of the steepest gradient. For each successive approximation the residual and the error is reduced. Ideally the error should be reduced to zero before stopping the iterations. Finding the error is, unfortunately, as hard as finding the solution it self. Restrictions are therefore set on the residual instead. For the exact solution the residual is zero. The customary stopping criteria for the SD and CG method are the same as for the multigrid algorithm, the iterations are stopped when the residual is a small fraction of the initial error or when the norm of the residual divided by the norm of the right hand side is below some predefined value.

The convergence of the SD method is highly dependent on the condition number, κ, of the matrix A, which is the ratio between the largest and the smallest eigenvalue of A. For large condition numbers the shape of the quadratic form is close to a "Valley", and for small ones it is shaped like a perfect "bowl". In the case of where the condition number is

large the convergence will be bad if the choice of starting point for the iterations is unlucky. This is illustrated in Figure 4.6.

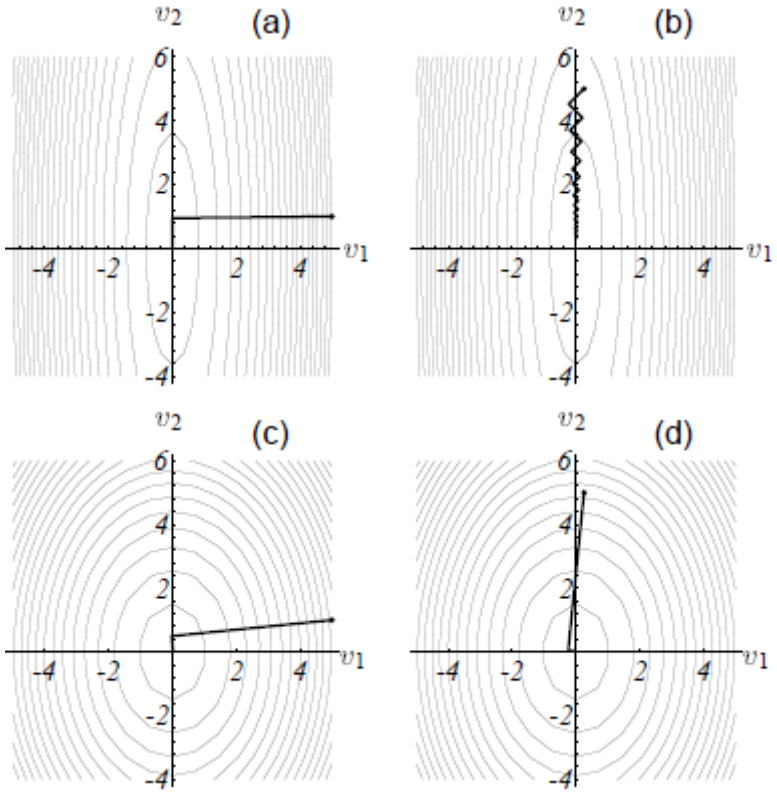

Figure 4.6: v_1 and v_2 is the eigenvectors of the matrix. μ is the slope of the error at the current iteration $\boldsymbol{e}^{(i)}$, i.e. the "quality" of the initial guess. κ is the condition number of the matrix A. (a) large κ, i.e. bad condition number, and small μ, i.e. a lucky initial guess. (b) large κ and large μ, result in bad convergence. (c) small κ and μ, and (d) small κ and large μ both yields a good convergence. Source Shewchuk (1994).

As illustrated in Figure 4.6 (b) the SD method can end up searching for the solution several times in the same direction, this is not the case for the CG method. Instead of searching in the direction of the largest negative gradient the CG method searches for the solution in directions, $\boldsymbol{d}_{(i)}$, that are A-orthogonal to each other, which means that

$$\boldsymbol{d}^{(i)} A \boldsymbol{d}^{(j)} = 0 \tag{4.36}$$

For $i \neq j$. Each new approximation of the solution is found at the point where the gradient on the search line is at its minima, as for the SD method. The CG algorithm avoids searching in the same direction several times as the SD can do since the search directions are predefined and span the same space as A. In the CG algorithm the initial residual is used as a basis for finding the search directions, and they are given by the vectors that span the Krylov subspace

$$\mathcal{K}_r\left(A, \boldsymbol{r}_{(i)}\right) = \text{span}\{\boldsymbol{r}^{(i)}, A\boldsymbol{r}^{(i)}, A^2\boldsymbol{r}^{(i)}, \dots, A^n\boldsymbol{r}^{(i)}\} \tag{4.37}$$

This concludes the short introduction to the idea behind CG method. The full explanation of this prominent algorithm is well presented in Shewchuk (1994). The outline of the algorithm is presented in the here:

If one has a good estimate for $\boldsymbol{x}$ one may use this as an initial step $\boldsymbol{x}_{(0)}$. Otherwise use $\boldsymbol{x}_{(0)}$=**0** as an initial guess.

1. $\boldsymbol{d}^{(0)} = \boldsymbol{r}^{(0)} = b - A\boldsymbol{x}^{(i)}$, find initial residual, i.e. the search direction
2. $\alpha^{(i)} = \frac{\left(\boldsymbol{r}^{(i)}\right)^T \boldsymbol{r}^{(i)}}{\left(\boldsymbol{d}^{(i)}\right)^T \mathrm{A}\boldsymbol{d}^{(i)}}$, Constant needed to find the step length
3. $\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} + \alpha\boldsymbol{d}^{(i)}$, find the next approximation
4. $\boldsymbol{r}^{(i+1)} = \boldsymbol{r}^{(i)} - \alpha^{(i)}A\boldsymbol{d}^{(i)}$, find residual for the new approximation
5. $\beta^{(i+1)} = \frac{\left(\boldsymbol{r}^{(i+1)}\right)^T \boldsymbol{r}^{(i+1)}}{\left(\boldsymbol{r}^{(i)}\right)^T \boldsymbol{r}^{(i)}}$
6. $\boldsymbol{d}^{(i+1)} = \boldsymbol{r}^{(i+1)} + \beta^{(i+1)}\boldsymbol{d}^{(i)}$, find search direction

Repeat 2-6 until convergence is achieved.

The convergence of the CG algorithm is heavily dependent on the condition number of the matrix A. Convergence can be improved by applying a technique called preconditioning. The idea is to multiply both sides of the equation, $A\boldsymbol{x} = \boldsymbol{b}$, with a matrix, i.e. $M^{-1}A\boldsymbol{x} = M^{-1}\boldsymbol{b}$, to obtain a more favourable condition number. The matrix M should be easy to invert and $M^{-1}A$ should have a significantly better condition number to get a performance gain from the preconditioning. The multigrid can in fact be used as preconditioner for the CG and the convergence of multigrid preconditioned CG is superior to the convergence of multigrid method in many cases, e.g. Trottenberg et al. (2001), Tatebe (1993) and Braess (1986).

5 Multigrid

In this thesis they are used to solve elliptic as well as parabolic equations. Multigrid algorithms are among the most efficient solvers for boundary value problems, see Trottenberg (2001). In recent years, however, multigrid methods have been used to solve a broad spectrum of problems, see section 5.5.

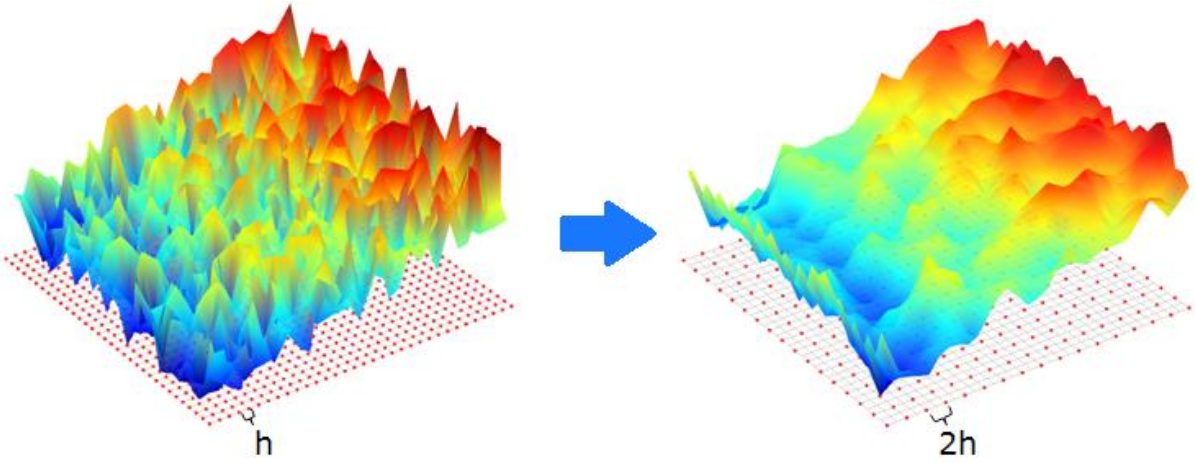

Figure 5.1: Classic iterative methods smooth the error efficiently and a smooth error term can be well represented on a coarse grid.

The remarkable feature of the multigrid algorithms is that the convergence rate does not deteriorate as the grid size increases, which is the problem for classic iterative problems. The computational cost of solving the partial differential equations with the multigrid algorithm is therefore roughly proportional with the number of unknowns $\mathcal{O}(N \log \varepsilon)$, where ε refers to the achieved accuracy of the solution. The drawback of the multigrid method is that it must be tailored to the specific problem at hand, as opposed to for example the conjugate gradient method.

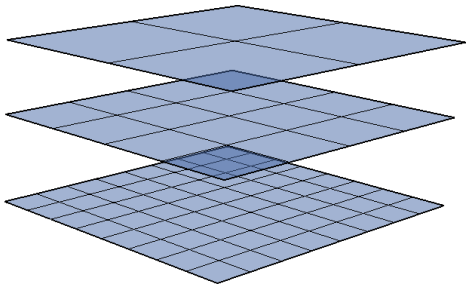

Figure 5.2: Shows a hierarchy of grids, where the grid spacing is halved for each level of coarsening.

This classic iterative method does have a central role has in the Multigrid algorithms. On their own these methods have a quite poor convergence, but they do have a strong smoothing effect on the error. The high frequency modes, corresponding to the large eigenvalues, are dampened rapidly. The low frequency modes on the other hand are dampened quite slowly. By transferring the problem to a coarser grid many of the low frequency modes are mapped onto the high frequency modes and can be efficiently dampened on the coarse grid, see Figure 5.1. By transferring the problem unto coarser

and coarser grids more and more of the low frequencies can be dampened. This is done with a hierarchy of coarser grids, see Figure 5.2.

5.1 Multigrid Cycle

The first step is to smooth the error using one of the classic iterative methods. The smooth error term is well represented on a coarser grid, which can be exploited by transferring the problem to a coarser grid. Consider a set of linear equations

$$A\boldsymbol{x} = \boldsymbol{b} \tag{5.1}$$

The solution of equation (5.1) is in general not a smooth function and is not necessarily well represented on a coarse grid. Some steps must be made to find an equation that is well defined on a coarse grid, which exploits the smooth error terms.

For the continued discussion the definition of the error and the residual is needed. As presented in the previous chapter these quantities are given by the following two expressions respectively

$$r = A\boldsymbol{x}^{(i)} - \boldsymbol{b} \tag{5.2}$$

$$\boldsymbol{e} = \boldsymbol{x}^{(i)} - \boldsymbol{x} \tag{5.3}$$

$\boldsymbol{x}^{(i)}$ is an approximation of the solution. Combining equation (5.1), (5.2) and (5.3) the residual equation can be found

$$A\boldsymbol{e} = \boldsymbol{r} \tag{5.4}$$

The residual is restricted from the fine grid to the coarser grids. The restriction is usually done in steps and the error is smoothed before each restriction, see section 5.1.1. The interval between the grid points is usually doubled for each step.

The equation is solved at the coarsest grid using one of the direct methods discussed in chapter 3 or one of the classic iterative methods. Solving the system on the coarsest level is efficient since the number of unknowns is small. The solution of the residual equation is interpolated back to the finest level, see section 5.1.2. This process is also done in steps and the solution is smoothed with one of the classic iterative methods on each coarsening level.

The process of restricting the equation down to the coarsest level, solving it and then interpolating it back to the finest level is called a V-cycle, see Figure 5.3. Multigrid is an iterative method and the V-cycle is repeated several times to find a sufficiently accurate approximation of the solution. Other possible cycles are the W-cycle and the full multigrid cycle, see Trottenberg et al. (2001).

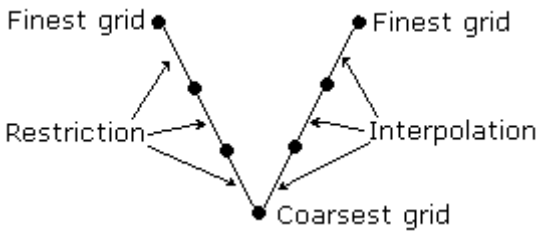

Figure 5.3: V-cycle. The residual is restricted from the finest grid to the coarsest grid at which equation (5.4) is solved and the solution is interpolated to the finest grid.

The goal of the algorithm is to minimize the error, $\boldsymbol{e}$, but finding the error is impossible without knowing the exact solution, see equation (5.3). In practice a norm of the residual is the measure of the quality of the approximation, as for the conjugate gradient algorithm. It is customary to stop running the cycles when the norm of the residual is a small fraction of the initial error or when the norm of the residual divided by the norm of the right hand side is below some predefined value. One should be aware that even if the residual is small the error is not necessarily so, a nice example of this is presented in Briggs et al. (2000) and reproduced here

Consider the following two linear equations

$$\begin{pmatrix} 1 & -1 \\ 21 & -20 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} -1 \\ -19 \end{pmatrix} \tag{5.5}$$

$$\begin{pmatrix} 1 & -1 \\ 3 & -1 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \tag{5.6}$$

The exact solution to both of these equations is $\boldsymbol{u} = (1,2)^T$. Suppose we have calculated the approximation $\boldsymbol{v} = (1.95,3)^T$. The error for this approximation is $\boldsymbol{e} = (-0.95,-1)^T$, for which $\|\boldsymbol{e}\|_2 = 1.379$. The norm of the residual for the first set of equations is $\|\boldsymbol{r_1}\|_2 = 0.071$, while the residual norm for the second system is $\|\boldsymbol{r_2}\|_2 = 1.851$. Clearly the relative small residual found for the first system does not reflect the rather large error.

An overview of the algorithm is presented as a flowchart in Figure 5.4. As indicated in the figure the restriction and the interpolation are described in further detail in the subsequent subsections. The direct solvers which are used on the coarsest level and the smoothers are discussed in chapter 3 and 4 respectively.

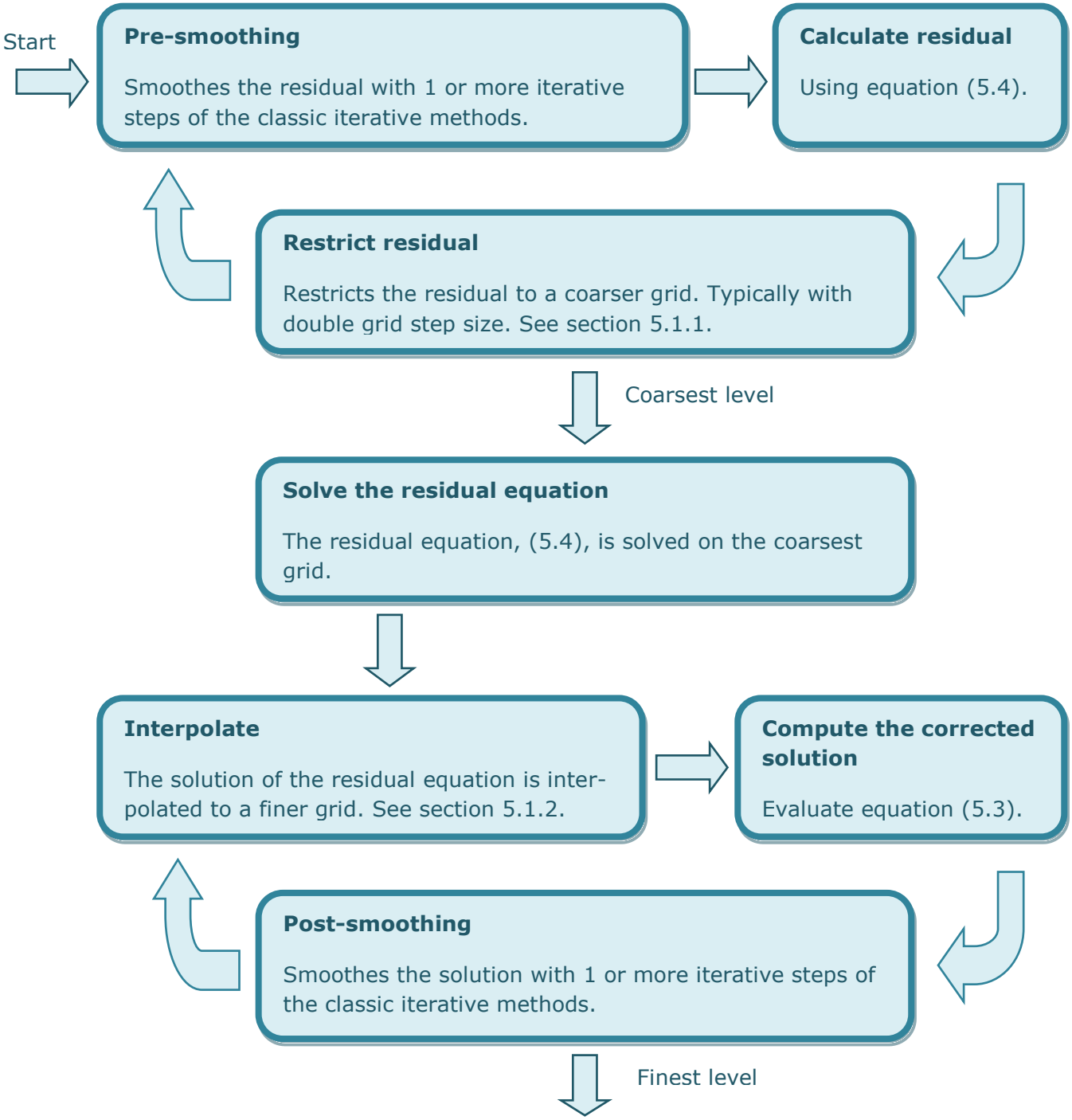

Figure 5.4: An overview of the multigrid cycle. The restriction and the interpolation are elaborated on in the following subsections. To get a sufficiently accurate approximation several multigrid cycles are run.

5.1.1 Restriction

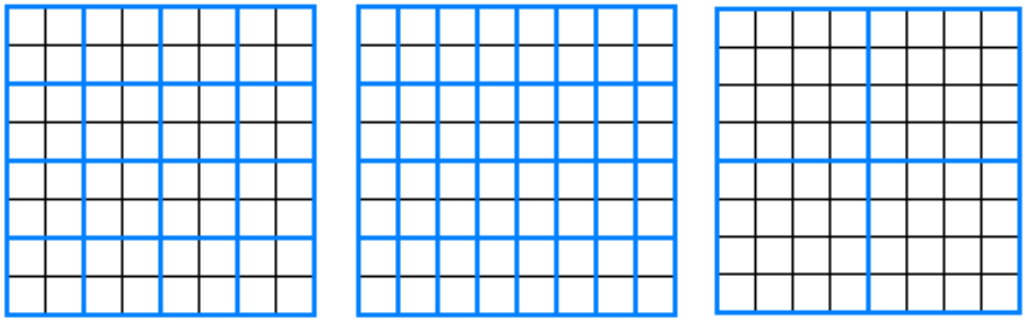

Figure 5.5: Choices for coarse grids. The whole grids show the fine grid and the blue part is the first coarsening level. The first image shows standard coarsening where the grid spacing is halved, the second one show semi-coarsening in the y-direction, and the last one shows coarsening where the grid spacing is four times larger than for the fine grid.

After the smoothing of the error term is transferred to a coarser grid. The multigrid algorithm uses a transfer operator that restricts the residual from a fine grid to a coarser grid. The choice of operator depends on the type of coarsening, see Figure 5.5. The transfer operators used in this thesis are applied to standard coarsening, where the grid spacing is doubled for each coarsening level. Semi-coarsening can be used in cases where there is a strong coupling between the unknowns in one direction and not the other.

In one dimension a common choice is the full weight operator (FW)

$$u_i^{2h} = \frac{1}{4}(u_{2i-2}^h + 2u_{2i-1}^h + u_{2i}^h) \tag{5.7}$$

u_i^h is the function values on the fine grid at grid point i with grid spacing h and u_i^{2h} is the function values on the coarse grid where the grid spacing is $2h$. Equation (5.7) can be written with stencil notation

$$I_h^{2h} = \frac{1}{4}[1\ 2\ 1]_h^{2h} \tag{5.8}$$

Where I_h^{2h} is a restriction operator that restricts the function values from a grid with where the interval between the grid points is h to one where the interval is of length $2h$.

In 2 dimensions the full weight operator is given as

$$I_h^{2h} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^{2h} \tag{5.9}$$

Another choice in 2 dimensions is to use the half weight operator (HW)

$$I_h^{2h} = \frac{1}{8}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}_h^{2h} \tag{5.10}$$

One should be aware that the half weight operator yields an incorrect scaling at the boundaries when periodic and Neumann boundary conditions are implemented. The restriction operator must be modified before applying them to the boundary points when periodic or Neumann boundary conditions is used. In 1 and 2 dimensions the Neumann boundary conditions on the left side of the domain gives the following restriction operators respectively

$$I_h^{2h} = \frac{1}{4}[0\ 2\ 2]_h^{2h} \tag{5.11}$$

$$I_h^{2h} = \frac{1}{16}\begin{bmatrix} 0 & 2 & 2 \\ 0 & 4 & 4 \\ 0 & 2 & 2 \end{bmatrix}_h^{2h} \tag{5.12}$$

For periodic boundary conditions the grid point values at the opposite side of the domain are required.

5.1.2 Interpolation

Figure 5.6: Values on the coarse, black, grid is interpolated to the fine grid, green, using linear interpolation.

The multigrid algorithm employs a transfer operator that interpolates a values at the gridpoints, $u_h(x)$, from a coarse grid to a finer grid. In the context of the multigrid algorithm the interpolation operator is often called a prolongation operator. As for the restriction there are several interpolation operators to choose from, in this thesis linear, bilinear and trilinear interpolation is used for 1, 2 and 3 dimensions respectively. Standard coarsening is still assumed, see Figure 5.5. For linear interpolation in 1 dimension, see Figure 5.6, is given by the following equations

$$u_{2i-1}^h = u_i^{2h} \tag{5.13}$$

$$u_{2i}^h = \frac{1}{2}\left(u_i^{2h} + u_{i+1}^{2h}\right)$$

Written in stencil notation this yields

$$I_{2h}^h = \frac{1}{2}]1\ 2\ 1[_{2h}^h \tag{5.14}$$

In two dimensions a bilinear interpolation scheme is used. This yields the following expression for the values on the grid points on the fine grid,

$$u^h_{2i-1,2j-1} = u^{2h}_{i,j}$$

$$u^h_{2i,2j-1} = \frac{1}{2}\left(u^{2h}_{i,j} + u^{2h}_{i+1,j}\right)$$

$$u^h_{2i-1,2j} = \frac{1}{2}\left(u^{2h}_{i,j} + u^{2h}_{i,j+1}\right)$$

$$u^h_{2i,2j} = \frac{1}{4}\left(u^{2h}_{i,j} + u^{2h}_{i,j+1} + u^{2h}_{i+1,j} + u^{2h}_{i+1,j+1}\right)$$

The colour coding is connected with the visualization of the fine grid in two dimensions in Figure 5.7. The black grid points are the grid points on the coarse grid.

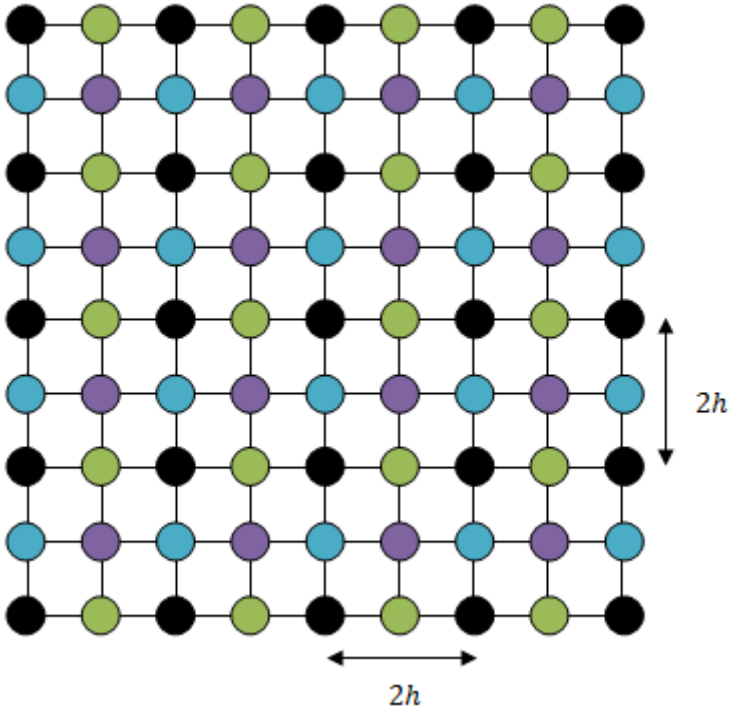

Figure 5.7: Fine grid in two dimensions. The grid points that exist on the coarser grid as well are marked in black.

The operator is then given by

$$I^h_{2h} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}^h_{2h} \tag{5.15}$$

The interpolation is unaffected by the choice of boundary conditions.

5.1.3 Coarse grid operator

The equation system must be solved on the coarsest grid. There are two main approaches to find the coarse grid operator that is used to build the system of equations on the coarsest level. The first one is to simply use the same finite difference operator as on the finest grid just corrected for the coarser grid spacing.

The other option is to use the so-called Galerkin coarse grid operator, which is defined by

$$L_H = I_h^H L_h I_H^h \tag{5.16}$$

Where L_H is the finite difference operator on the coarse grid, and L_h is the finite difference operator on the fine grid. I_h^H and I_H^h are the restriction and the interpolation operator respectively. The Galerkin coarse grid operator must be used when solving partial differential equation on an unstructured grid.

5.2 Inhomogeneous Systems

The materials that are of interest in earth sciences are inhomogeneous even down to the crystal scale. Their large degree of heterogeneity influences their bulk behaviour. Digital reconstruction of rock samples from oil reservoirs are being used in the petroleum industry to find material properties of the rocks, see Figure 5.8. The rock samples are heterogeneous down to microscopic levels, and the heterogeneity influences the material properties of the rock, such as porosity and permeability. These properties are in turn used to describe fluid flow through the rock samples.

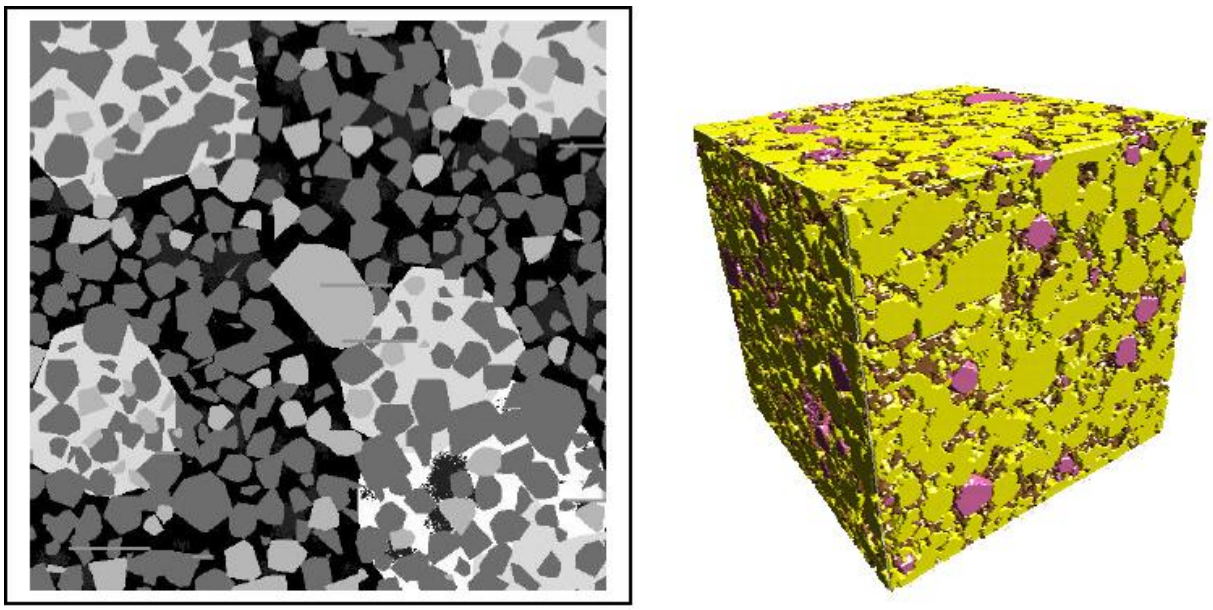

Figure 5.8: The images are found in a case study carried out by numericalRocks. To the left there is a micro-CT image of sandstone sample from an oil reservoir in the North Sea. The black areas are pore spaces. Micro-CT image is one of several cross sections that are used to create a 3D representation of the rock sample, as shown on the right hand side of the figure.

The heterogeneity of the material that is simulated has an adverse affect on the convergence of the multigrid algorithms. The smoothing properties of the classic iterative methods are strongly affected by this. Assuming for example that we have a strong coupling of unknowns in one direction, in the system where steady state heat diffusion is calculated. The equation is given as

$$\varepsilon \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{5.17}$$

In the equation the unknowns in the x-direction is weakly coupled if $\varepsilon \ll 1$. The stencil for this equation is given as follows

$$\frac{1}{h^2}\begin{bmatrix} & 1 & \\ \varepsilon & -2(1+\varepsilon) & \varepsilon \\ & 1 & \end{bmatrix} = 0 \tag{5.18}$$

Applying point wise smoothing with one of the classic iterative methods will smooth the error as efficiently as normal in the y-direction, but due to the weak coupling in the x-direction the error will hardly be affected by the smoothing in this direction. The result is illustrated in Figure 5.9.

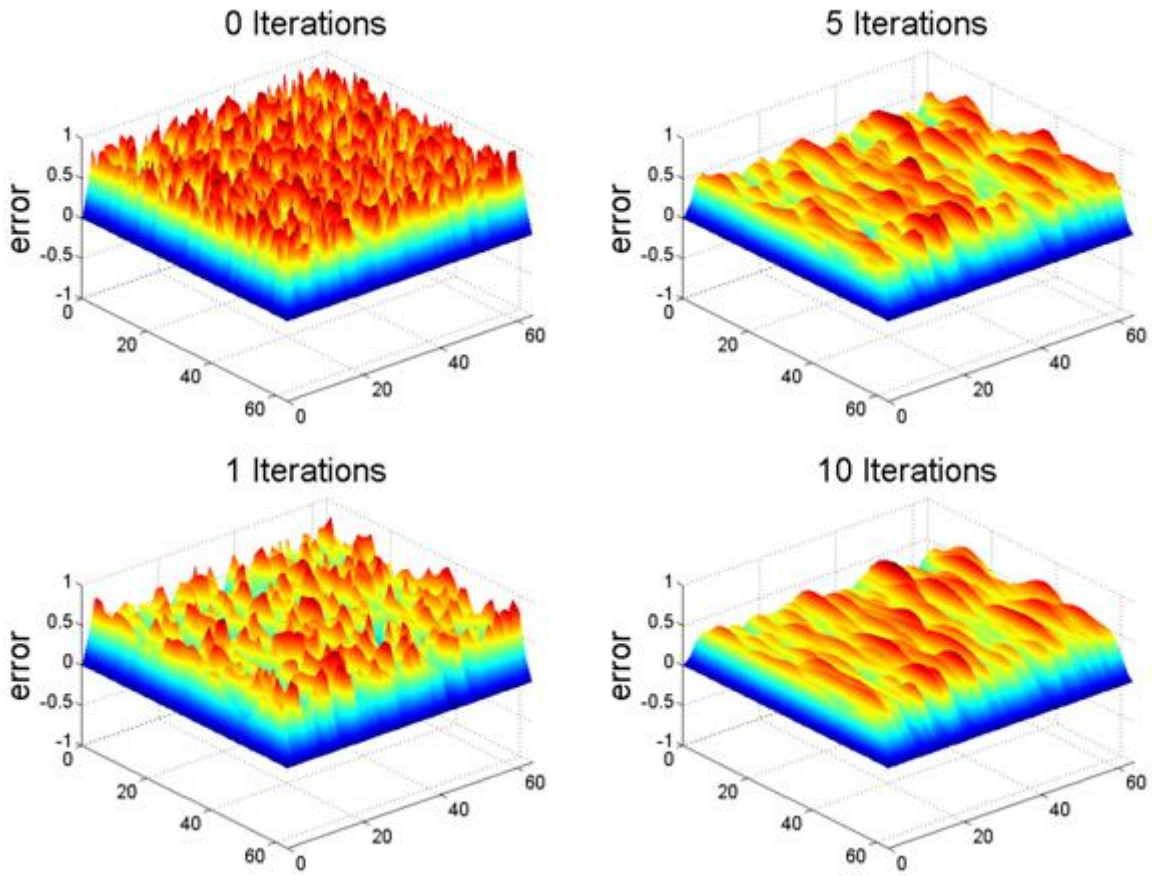

Figure 5.9: Pointwise Gauss-Seidel smoothing of a system with strong coupling of the unknowns in one direction.

This motivates the idea of line smoothing where smoothing is done in each spatial direction separately. In three dimensions there is the possibility to use plane smoothers as well. Both line smoothers and plane smoothers are implemented in this thesis for three dimensional systems.

The line and plane smoothers work well for systems where the unknowns are strongly coupled in one direction. In rocks, however, the strongly coupled unknowns are in general not restricted to certain spatial directions, which is clear from Figure 5.8. For cases were there are heterogeneities in all directions alternating line relaxation is a possibility. In alternating line relaxation smoothing a combination of line smoothing in all spatial directions is used. Alternating line relaxation efficiently smoothes the error when there is strong coupling in various directions.

5.2.1 Line smoothers

Line smoothers collectively update unknowns in lines in the grid. The contributions to the stencil in the other spatial directions go into the right hand side in the set of equations; this result in linear systems of equations with tridiagonal coefficient matrices. These sets

of equations can be solved with for example cyclic reduction, Thomas algorithm or with one of the classic iterative methods.

The lines can be updated in a lexicographic order or using so called zebra line smoothing, for which all odd numbered lines are updated first and then all even numbered lines are updated. Both types of ordering are illustrated in Figure 5.10.

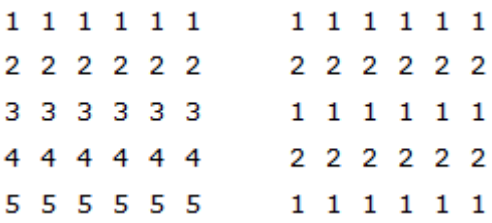

Figure 5.10: Line smoothers update all values in one line at the time. The figure to the left illustrates line smoothing with lexicographic ordering; and the figure to the right illustrates line smoothing with zebra ordering. Using the zebra ordering would be the analogue to Gauss-Seidel red-black ordering for point wise smoothers.

5.2.2 Plane smoothers

Plane smoothers are the analogue to line smothers in three dimensions, where unknowns in planes are updated collectively. Contributions from the final spatial dimension would be moved to the right hand side of the equation as for the line smoothers. The coefficient matrix that arises from this technique is banded with a bandwidth, see chapter 3, of $2n + 1$, where n is the number of unknowns in one spatial direction. As discussed in chapter 3, matrices with this bandwidth is considerably slower to solve than tridiagonal ones with direct methods. For large systems iterative methods is the preferable choice. As for the line smoothers it is possible to choose between lexicographic ordering and zebra ordering when smoothing the planes.

5.2.3 Alternating line relaxation

For systems where the heterogeneity is not connected to a certain direction, but rather to specific regions the line smoothing in specific directions will fail to smooth the error sufficiently. A remedy for this is to smooth the error with line smoothers in alternating directions. In the version that is implemented in this thesis the smoothing is done in all spatial directions with zebra ordered line smoothing. In two dimensions this means that smoothing is done for all odd numbered lines and then for all even numbered lines in the x-direction once; then the same procedure is done for the y-direction.

Using alternating line smoothers yields an efficient convergence for systems where the inclusions causing the heterogeneities have a larger thermal coefficient than the surrounding grid. For systems where the thermal coefficients in the inclusion are several orders of magnitude smaller than the thermal coefficients in the bulk the convergence starts to deteriorate.

5.3 Implementation

As mentioned in the introduction to this chapter the implementation of multigrid solvers must be tailored to the problem at hand. Several versions are therefore implemented in this thesis, which can handle different boundary conditions and for homogeneous and heterogeneous systems.

The multigrid function takes input parameters that define the problem and technical parameters that specify how the algorithm should solve the system. Tweaking the technical parameters is necessary to find the best possible convergence and the most efficient implementation. The input parameters are listed in Table 5.1.

Variable name	Iteration matrix
u	Previous approximation for the solution, i.e. temperature values
b	Right hand side
L	L is the finite difference stencil
v1	Number of pre-smoothing steps
v2	Number of post-smoothing steps
cl	Parameter which sets the maximum number of grid points on the coarsest grid level.

Table 5.1: Descriptions of the input variables to the multigrid function.

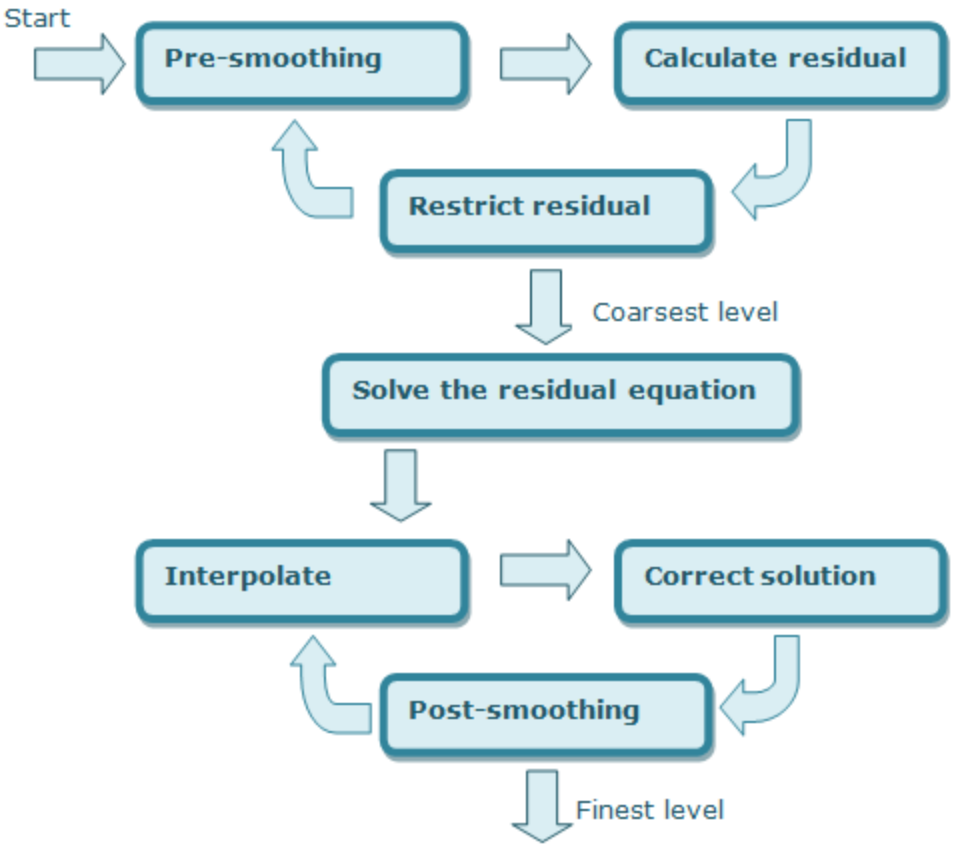

Figure 5.11: An overview of the multigrid algorithm.

A variety of different implementations was tested to find an efficient implementation of the multigrid algorithm. An overview of the multigrid algorithm is shown in Figure 5.11.

As shown in the figure there is 7 main components of the algorithm, where two of them are the same operation. These are the pre- and post smoothing.

The standard approach for implementing the smoothers, the calculation of the restriction and calculating the residual would be to use indexing and vector notation. For the interpolation the built in function interp3 is commonly used. Implementation with indexing is, unfortunately, inefficient when in Jacket and interp3 is not supported.

The solution to this problem was to use convolution which is supported by both MATLAB and Jacket. This function applies a stencil, often called a kernel, to the matrix. The stencil that should be applied to the matrix, A, can for example be the 5-point stencil, L, for the Laplacian in 2 dimensions

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} \tag{5.19}$$

$$L = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The convolution function will than calculate

$$a_{ij}^{new} = 4a_{ij} - a_{i-1j} - a_{i+1j} - a_{ij-1} - a_{ij+1} \tag{5.20}$$

It is called in the following way in MATLAB and Jacket

```
u_new = convn(u,L)
```

The convolution is used to calculate the residual, by applying the stencil for the Laplacian directly. The stencil used in the convolution for the smoothers is modified to have zero in the middle, which allow for the splitting of the matrix, $\boldsymbol{x}^{(i+1)} = -D^{-1}E\boldsymbol{x}^{(i)} + D^{-1}\boldsymbol{b}$. The boundaries are handled separately.

Calculating the interpolation and the restriction was also fount to be far more efficient when using convolution. For the restriction the convolution is applied to the fine grid using the full weight operator as a kernel, after which the grid points that exist on the coarser grid is used and the rest of them is discarded. For a regular grid we found that it is possible implement the interpolation using convolution. This was done by first creating a matrix with zeros where the values on the fine grid should be placed. The next step is to fill in the values that existed on the coarse grid as well, i.e. the grid points marked in black in Figure 5.7. On this finer grid now consisting of zeros and the values that existed on the coarser grid convolution is applied using the interpolation operator, see (5.15).

For the Gauss-Seidel method the convolution needs to be applied 2, 4 or 8 times depending on which stencil is used. Number of iterations needed to achieve convergence is less when using Gauss-Seidel method for the smoother in 2 dimensions, the difference is very

small in 3 dimensions. In both 2 and 3 dimensions the total number of times convolution applied to the grid to achieve the selected convergence was less when using Jacobi method for the smoothing. Introducing over relaxation results in a faster convergence, but the effect is too small to justify the number of extra mathematical operations that must be done in the multigrid cycle. Over relaxation was therefore removed in the final implementations. In the final implementations standard Jacobi method without overrelaxation is used as a smoother.

At the coarsest level the equation must be solved. This can be done by using one of the smoothers, by a direct solver or using conjugate gradient method. For singular systems direct solvers without modifications cannot be used. It was found that using smoothers yielded the fastest solver on the coarsest level since only a few iterations are needed to solve the system of equations on such a coarse grid. Solving the system that is singular on the coarsest level is more efficient if the solution is shifted by a constant such that the mean of it is zero.

Solving a singular system means that there are infinitely many solutions to the set of equations, these solutions are shifted by different constants. For the pressure solver used in the porous convection application this is not a problem since only the gradient of the pressure is needed to find the velocities.

5.3.1 Parallel properties of the multigrid components

The multigrid algorithm as a whole is not directly parallelizable, since it runs through the different grids sequentially and because the degree of parallelism changes from grid level to grid level. The smoothers, restriction, residual and interpolation functions do, however, have a large degree of parallelism.

- **Smoothers**: The Jacobi method is "*fully Ω_h parallel*", meaning that the value in each grid point can be calculated independently of all others. Ω_h means for all points on the grid level where the grid spacing is h. The notation of degree of parallelism is here introduced for the Jacobi method

$$\mathrm{par}-\deg(Jacobi)=\#\Omega_h \tag{5.21}$$

  For Gauss-Seidel method with lexicographic ordering the degree of parallelism is relatively low, since the updating of the value each grid point is dependent on previously calculated ones. The Gauss-Seidel method with red-black ordering, see Figure 4.2, on the other hand has a large degree of parallelism; the values in the red grid points can be updated simultaneously since they are uncoupled with each other and then al values in the black grid points can be updated simultaneously. The parallel degree of the Gauss-Seidel (GS) method is,

$$\mathrm{par}-\deg(GS_{red-black})=\frac{1}{2}\#\Omega_h \tag{5.22}$$

- **Residual**: The residual in each grid point can be calculated independent of the values in all other grid points. The calculation of the residual is therefore "*fully Ω_h parallel*"

$$\text{par} - \deg(\text{Residual}) = \#\Omega_h \tag{5.23}$$

- **Restriction:** The restriction operator is applied independently to all grid points on the coarser grid level, this goes for both the half weight and the full weight operator. Meaning that if the restriction operator is applied to the restriction from a grid Ω_h to a coarser grid Ω_{2h} then the degree of parallelism is

$$\text{par} - \deg(\text{Restriction}) = \#\Omega_{2h} \tag{5.24}$$

- **Solver**: A parallelizable solver can be used on the coarsest grid level for example the Gauss-Seidel method.
- **Interpolation**: The interpolation operator act differently on different grid points, but each operation can be done independently of the others.

The complexity of the multigrid algorithm is for a two dimensional Poisson solver, i.e. steady state heat diffusion, is $\mathcal{O}(N \log \varepsilon)$ for the sequential algorithm and $\mathcal{O}(\log N \log \varepsilon)$ for the parallel implementation; when Gauss-Seidel red-black ordering is used for the smoothers. The complexities for this algorithm are found in Trottenberg et al. (2001).

It is preferable to divide the discretization grid into partitions since the communication within each partition is efficient. The idea of partitioning is discussed in section 7.1 for the standard implementation of the finite difference discretized heat diffusion equation on the graphics processing unit. The size of the grid should be chosen such that the grid on each grid level can be divided into partitions.

5.4 Convergence Tests

The efficiency of the multigrid algorithm is depended on the various input values, such as the relaxation parameter, and number of pre- and post-smoothing steps (see TBL) and the choice of smoother. Each parameter must be tweaked to find the best convergence rate to the problem at hand. In this chapter the effect of changing each of the parameters are explored to find optimal values for problems in 2 and 3 dimensions with different boundary conditions for steady state heat diffusion. The results are used for the porous flow application presented in the following chapter. The different boundary conditions are listed in Table 5.2.

Boundary conditions	**2 dimensions**
Dirichlet	Dirichlet boundary conditions on all walls.
Neumann Dirichlet	Dirichlet boundary conditions on top and bottom wall and Neumann boundary conditions on lateral walls.
Periodic Dirichlet	Dirichlet boundary conditions on top and bottom wall and periodic boundary conditions on lateral walls.
Pure Neumann	Neumann boundary conditions on all walls.
Pure Periodic	Periodic boundary conditions on all walls.
Neumann periodic	Neumann boundary conditions on top and bottom wall and periodic boundary conditions on lateral walls.
	3 dimensions
Dirichlet	Dirichlet boundary conditions on all walls.
Neumann Dirichlet	Neumann boundary conditions on two lateral walls and Dirichlet boundary conditions on the other walls.
Periodic Dirichlet	Periodic boundary conditions on two lateral walls and Dirichlet boundary conditions on the other walls.
2 Neumann Dirichlet	Dirichlet boundary conditions on top and bottom wall and Neumann boundary conditions on lateral walls.
2 periodic Dirichlet	Dirichlet boundary conditions on top and bottom wall and periodic boundary conditions on lateral walls.
Pure periodic	Periodic boundary conditions on all walls.
Pure Neumann	Neumann boundary conditions on all walls.
Neumann periodic	Neumann boundary conditions on top and bottom wall and periodic boundary conditions on lateral walls.

Table 5.2: Notation and descriptions of the different boundary conditions discussed in this chapter. The flux is set to zero at the Neumann boundaries.

The multigrid algorithm was tested for a system with a random source term, with the mean value equal to zero. The same random source term was used to analyse each of the different boundary conditions for each system size. The Jacobi method is used as a smoother and a solver at the coarsest level. One pre-smoothing and one post-smoothing step is used. Based on preliminary test it was found that the ideal relaxation parameter was in the interval 0.7 to 1.0 of for 2 dimensional systems and between 1.0 and 1.3 for 3 dimensional systems.

Several different stencils for approximating the Laplacian can be used; some of these stencils are presented in section 2.2.5. The convergence rate is tested for four different stencils in 3 dimensions where Dirichlet boundary conditions were used. That is the Mehrstellen stencil, a stencil derived from the finite element approximation of the Laplacian on a regular grid, one presented by Hale (2008) and one presented by Patra et. al (2005). The relative residual after 10 iterations using different relaxation parameters is shown in Figure 5.12. The relative residual is the difference between the norm of the re-

sidual at the current iteration and the norm of the initial residual. Based on these results we chose to use the Mehrstellen stencil for the Laplacian.

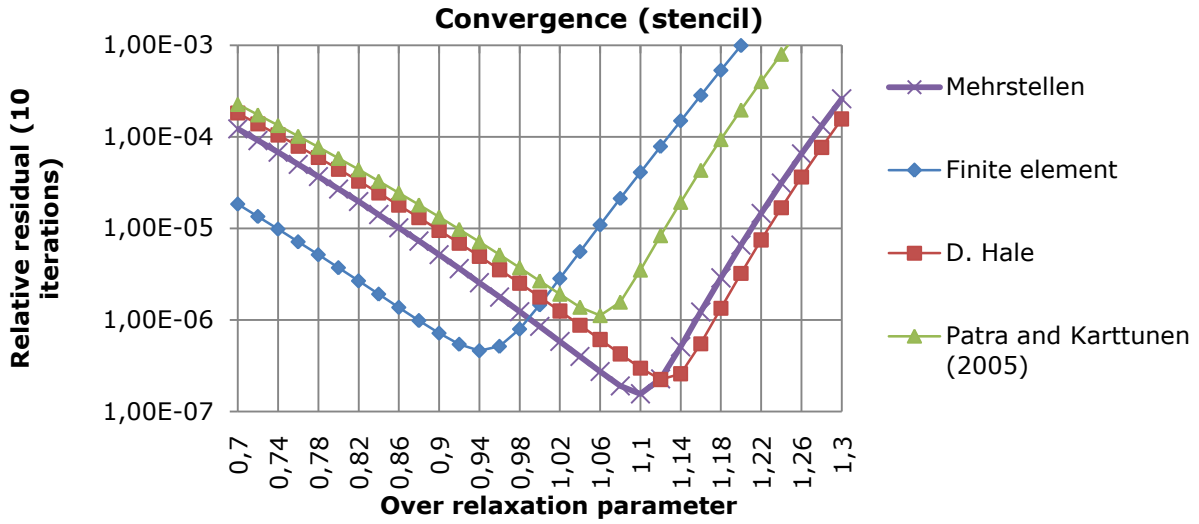

Figure 5.12: Relative residua after 10 iterations for different stencils for approximating the Laplacian. The different stencils are presented in section 2.2.5. The convergence is tested for a system of size 129x129x129.

The effect of the relaxation parameter is studied for all boundary conditions using the Mehrstellen stencil for the smallest system size, which is 1025x1025 in 2 dimensions and 129x129x129 in 3 dimensions. The relative residual after 10 iterations using different relaxation parameters is shown in Figure 5.13 and Figure 5.14, for 2 and 3 dimensions respectively.

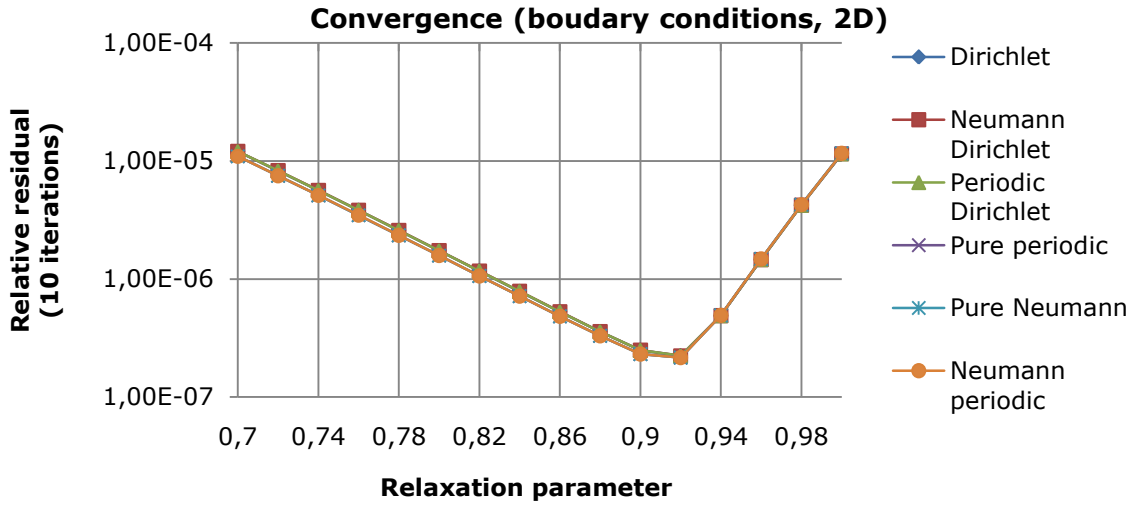

Figure 5.13: The relative residual, norm of the residual divided by the norm of the initial residual, after 10 iterations of the multigrid cycle for different relaxation parameters is shown for different boundary conditions in 2 dimensions. There is a slight difference between the convergence rate of the systems with Dirichlet boundary conditions and systems with no walls with Dirichlet boundary conditions, but it this effect is negligible in 2 dimensions and does not affect the choice of relaxation parameter.

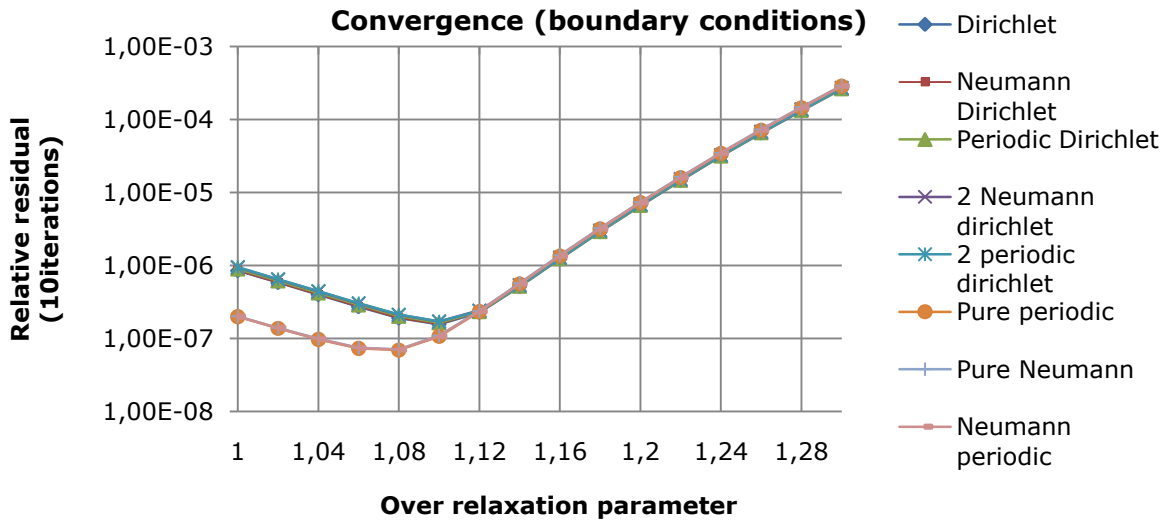

Figure 5.14: The relative residual after 10 iterations of the multigrid cycle for different relaxation parameters is shown for different boundary conditions in 3 dimensions. There is a clear difference in convergence for systems with Dirichlet boundary conditions at any of the walls and systems with no walls with Dirichlet boundary conditions. This has an effect on which relaxation parameter that yields the best convergence.

There is a small difference in the convergence between systems with Dirichlet boundary conditions on any of the walls and systems with no Dirichlet boundary conditions, but the difference it too small to affect the choice of relaxation parameter in 2 dimensions. Relaxation parameter equal to 0.91 yields the most efficient convergence for all boundary conditions that were tested. In 3 dimensions there is a distinct difference between the convergence rates of systems with Dirichlet boundary conditions on any of the walls and systems with no Dirichlet boundary conditions see Figure 5.14Figure 5.14. The best relaxation parameter when using Dirichlet boundary conditions is 1.1 and for systems with no Dirichlet boundary conditions the convergence is most efficient when the relaxation parameter is equal to 1.07.

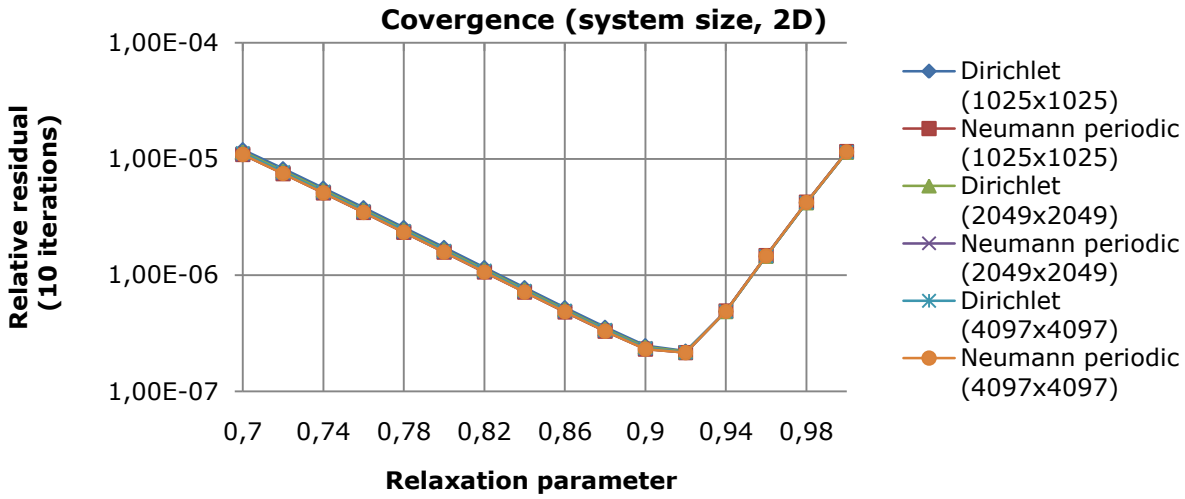

Figure 5.15: Relative residual is shown after 10 iterations for different system sizes with Dirichlet boundary conditions on all walls and a combination of Neumann and periodic boundary conditions in 2 dimensions. The convergence is hardly affected by the choice of boundary conditions or system size in 2 dimensions.

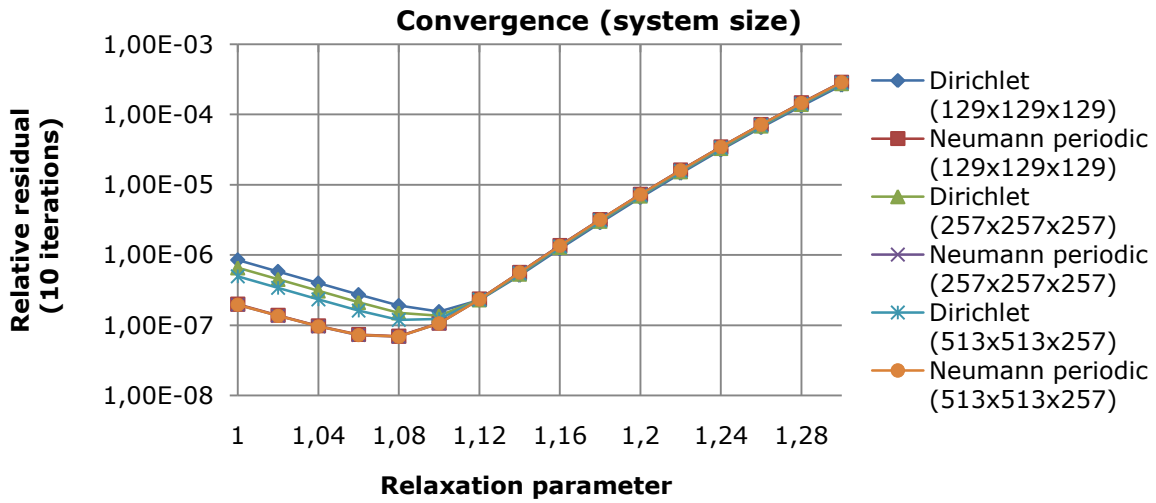

Figure 5.16: Relative residual is shown after 10 iterations for different system sizes with Dirichlet boundary conditions on all walls and a combination of Neumann and periodic boundary conditions in 3 dimensions. The convergence is affected by the choice of boundary conditions, but is relatively unaffected by the system size.

To find the best choice of relaxation parameter for larger systems a system with only Dirichlet boundary conditions and a system with periodic boundary conditions on the lateral walls and Neumann boundary conditions on the top and bottom wall was used. The same is done for the tests using Gauss-Seidel method as a smoother. This choice was made since the convergence for different relaxation parameters was only affected by whether or not there were Dirichlet boundary conditions in the system. The combination with Neumann and periodic boundary conditions was specifically chosen since it is used in the porous flow application presented in the following chapter.

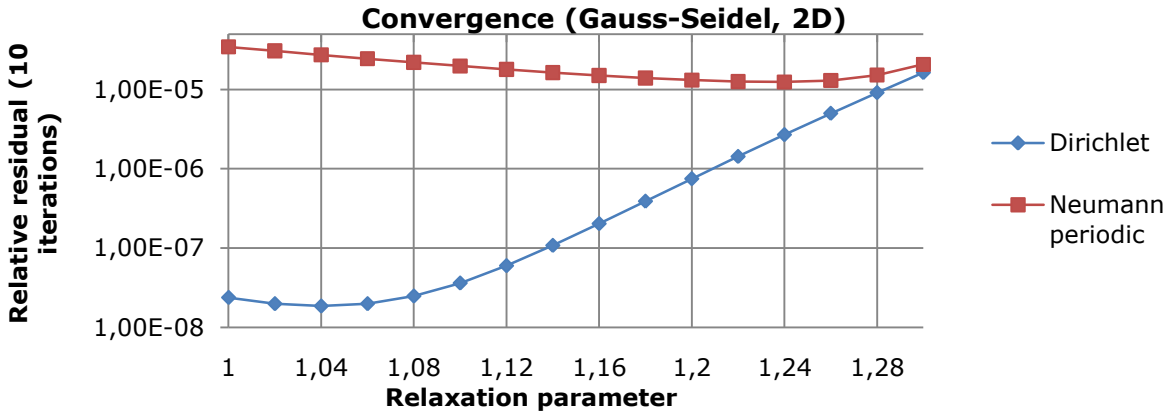

Figure 5.17: Relative residual is shown after 10 iterations for different system sizes with Dirichlet boundary conditions on all walls and a combination of Neumann and periodic boundary conditions in 3 dimensions. Gauss-Seidel with four colour ordering is used as a smoother. The convergence of the multigrid algorithm is best for when using a relaxation parameter equal to 1.04 and 1.24 for a system with Dirichlet and a combination of Neumann and periodic boundary conditions.

The convergence for different relaxation parameters is studied for three system sizes in both 2 and 3 dimension (see Figure 5.15 and Figure 5.16), that is 129x129x129, 257x257x257 and 513x513x257 in 3 dimensions and 1025x1025, 2049x2049 and

4097x4097 in 2 dimensions. The convergence is hardly affected by changing the system size.

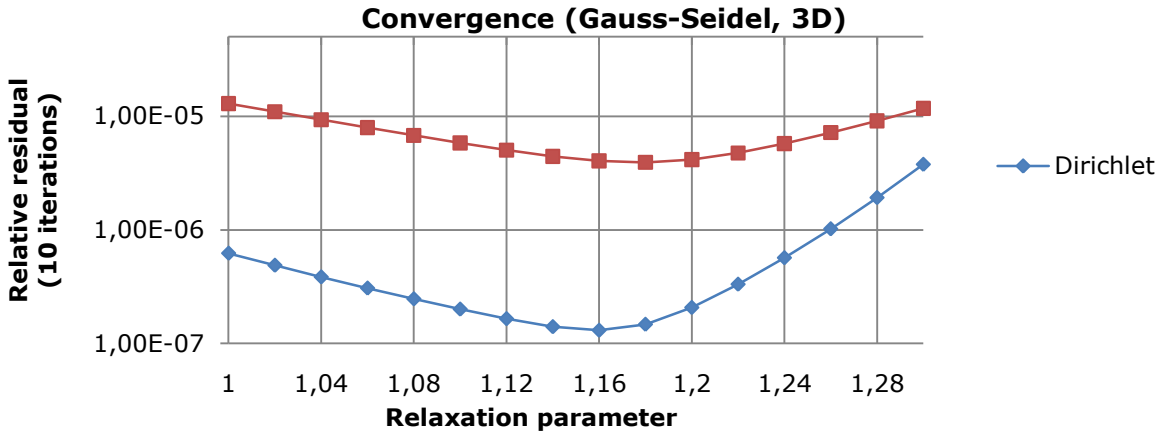

Figure 5.18: Relative residual is shown after 10 iterations for different system sizes with Dirichlet boundary conditions on all walls and a combination of Neumann and periodic boundary conditions in 3 dimensions. Gauss-Seidel with four colour ordering is used as a smoother. The relaxation parameter that yields the best convergence is 1.16 for a system with Dirichlet boundary conditions and 1.18 for a system with a combination of Neumann and periodic boundary conditions.

Test were carried out to find the best relaxation parameter when using the Gauss-Seidel method as a smoother. This was done in both 2 and 3 dimensions using one pre- and one post-smoothing step. The results are shown in Figure 5.17 and Figure 5.18, for 2 and 3 dimensions respectively. The relaxation parameters that yielded the best convergence are 1.04 and 1.24 in 2 dimensions for Dirichlet boundary conditions and a combination of Neumann and periodic boundary conditions respectively. The corresponding values in 3 dimensions are 1.16 and 1.18.

Using Gauss-Seidel method for the smoother yields a better convergence in 2 dimensions than the use of the Jacobi method for the smoother. In 3 dimensions the number of iterations needed to achieve convergence is relatively unaffected by the choice of smoother.

The efficiency of the algorithm is mainly dependent on the number of times convolution is applied on the finest level. Four colour ordering was used in the Gauss-Seidel method, since a 19 point stencil in 3 dimensions and a 9 point stencil in 2 dimensions was used to approximate the Laplacian. Convolution must be applied 4 times per smoothing step when four colour ordering is used, whereas the Jacobi method only requires that convolution is applied once. The calculating of the residual, restriction the residual and interpolation requires that the convolution function is applied three times. The efficiency of the implementation using Gauss-Seidel and four colour ordering will therefore roughly be proportional to

$$n \cdot (v1 \cdot 4 + v2 \cdot 4 + 3) \tag{5.25}$$

Where $v1$ is the number of pre-smoothing steps, $v2$ is the number of post-smoothing steps and n is the number of iterations that is needed to achieve the selected conver-

gence. The number of times the convolution function is applied on the finest grid using the Jacobi method is used as a smoother is given by the following expression

$$n \cdot (v1 + v2 + 3) \tag{5.26}$$

Tests were carried out using from 0 to 2 pre- and post-smoothing steps using both the Gauss-Seidel method and the Jacobi method to find the ideal number of pre- and post-smoothing steps for each method and the results were than compared to find out which method used for the smoothing resulted in the most efficient algorithm. The results are presented in Table 5.3 and Table 5.4 for 2 and 3 dimensions respectively. The relaxation parameters that was fount to give the best convergence were used.

It is clear from the tables that using Jacobi method as a smoother is more efficient in both 2 and 3 dimensions. With the Jacobi method it is preferable to use a total of 3 smoothing steps for both types of boundary conditions in 2 dimensions. In 3 dimensions it is most efficient to use two pre- and one post-smoothing step when solving a system with a combination of periodic and Neumann boundary conditions. When solving a system with Dirichlet boundary conditions on any of the walls it is most efficient to use a total of 3 pre- and post-smoothing steps.

$v1$	$v2$	**Number of times convolution is applied on the finest level (2D)**			
		Gauss-Seidel (four colour)		**Jacobi**	
		Dirichlet	Neumann Periodic	Dirichlet	Neumann Periodic
1	**0**	98	140	80	-
0	**1**	84	133	80	76
1	**1**	77	132	55	55
1	**2**	90	150	54	54
2	**1**	90	150	54	54
2	**2**	95	171	56	56

Table 5.3: Total number of times convolution is applied on the finest level to achieve a relative convergence of 1E-7 using Gauss-Seidel with four colour ordering and Jacobi method as smoothers. A 2 dimensional system with 1025x1025 unknowns is tested with Dirichlet boundary conditions and a combination of Neumann and periodic boundary conditions. $v1$ and $v2$ is the number of pre- and post-smoothing steps respectively.

$v1$	$v2$	**Number of times convolution is applied on the finest level (3D)**			
		Gauss-Seidel (four colour)		**Jacobi**	
		Dirichlet	Neumann Periodic	Dirichlet	Neumann Periodic
1	**0**	126	203	80	72
0	**1**	112	196	80	76
1	**1**	121	121	55	50
1	**2**	180	180	48	50
2	**1**	150	150	48	48
2	**2**	209	209	49	49

Table 5.4: Total number of times convolution is applied on the finest level to achieve a relative convergence of 1E-7 using Gauss-Seidel with four colour ordering and Jacobi method as smoothers. A 3 dimensional system with 129x129x129 number of unknowns is tested with Dirichlet boundary conditions and a combination of Neumann and periodic boundary conditions. In 3 dimensions the implementations using Jacobi method as a smoother is far more efficient.

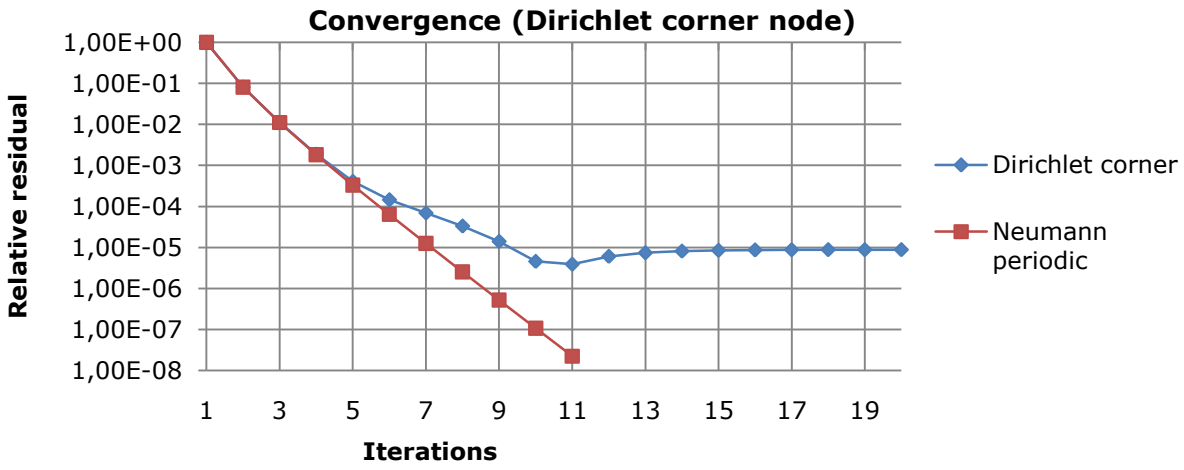

Figure 5.19: Convergence rate for a system with periodic boundary conditions on the lateral walls and Neumann boundary conditions on the top and bottom wall. Having one node, in this case the corner node, with a fixed value (Dirichlet boundary condition) resulted in poor convergence.

For the application presented in the following chapter, where flow through a porous material is studied, a combination of Neumann and periodic boundary conditions are used. The set of equations that results from using these boundary conditions is singular, i.e. it has infinitely many solutions. To find the correct solution one could fix one point in the domain using a Dirichlet boundary point or shift the solutions such that its mean value is zero. Using one Dirichlet point resulted in poor convergence for the multigrid algorithm, see Figure 5.19. A system with no Dirichlet point was therefore used in this application.

5.5 Future Outlook

As mentioned briefly in the introduction to this chapter, the multigrid algorithms have successfully been applied to a variety of problems.

Algebraic multigrid is designed to handle problems with unstructured grids and can indeed handle problems with no grid at all. The idea is that the coarsening is not defined by the grid, but rather from the properties of the matrix itself. Galerkins technique is used to define the set of equations on the coarser grid. The technique was mentioned in section 5.1.3 where the coarse grid operator where discussed, but it is not implemented in thesis. The idea is to algebraically define the coarse grid operator based on the interpolation and restriction operators. Implementing algebraic multigrid to solve PDE's using body fitted meshes and finite element discretization of the derivatives would be the natural step in further developing the work in this thesis. This type of implementation might, however, not be well suited for the GPU.

The multigrid method can be used to precondition the matrix for use in the conjugate gradient method. It would be useful to compare the convergence of direct multigrid and preconditioned conjugate gradients using multigrid for different physical problems. This is, however, beyond the scope of this thesis.

6 GPU Programming

Software applications have traditionally been written for serial computation, where one instruction could be executed at a time. Serial applications are limited by the number of instructions, the average runtime of each instruction and the rate at which the data can be transferred to the processing core.

Modern computers employ parallel computation to speed up applications. In parallel computing a large problem is divided into smaller ones that are divided between multiple processing cores. The hardware producers agree that parallel computing is the way of the future. Graphic Processing Unit, GPU, is a prime example of hardware that is designed for parallel computing where instructions are divided between the hundreds of processing cores in the GPU.

The GPU is a specialized processor which handles rendering of 2D and 3D graphics. The computational power of the GPU has increased rapidly in recent years due to the demand for ever more realistic computer games. Modern GPU's are therefore very powerful processors; they are in fact far more powerful than the Central Processing Unit, CPU. The GPU can give a desktop computer the computational power of a small to a medium sized cluster of CPU's.

The GPU has traditionally only handled graphics computations, but in recent years the interest for using GPU for general purpose programming has sky rocketed. Especially after NVIDIA released CUDA, which allowed users to use standard programming languages to utilize the GPU. Third party producers have opened up for utilizing the GPU through high-level languages such Python and MATLAB. This has made the computational power of the GPU available for more end users. Writing an efficient algorithm that utilizes the GPU does, however, require a solid understanding of its architecture.

There is a large range of algorithms that can exploit the highly parallel architecture of the GPU. Large speedups of simulations using GPU's have been shown in a variety of research fields, such as fluid dynamics, molecular dynamics, medicine, computational chemistry and finance.

Software that exploits the computational power of the GPU has been developed for use in oil exploration, where detailed geological models of the subsurface are needed. The information is based on large-scale seismic surveys. A few hundred square kilometres are covered in a typical marine survey, and this result in several terabytes of recovered data. By using GPU's in a hardware cluster, NVIDIA has shown speedups of a factor of 10 in the processing of these data, see Hubert Nguyen (2008).

Using finite differences to do simulations of geological processes is very well suited for the architecture of the GPU. The problem can easily be divided in to smaller problems, since it is discretized on a structured mesh and the operations done on each node are only directly affected by the nearest nodes.

6.1 Components in a PC

A computer is made up of a few basic components, as shown in Figure 6.1. A short description of these components is presented here. The hardware and operating system of the computer where the applications are tested out in this thesis is shown in Table 6.1.

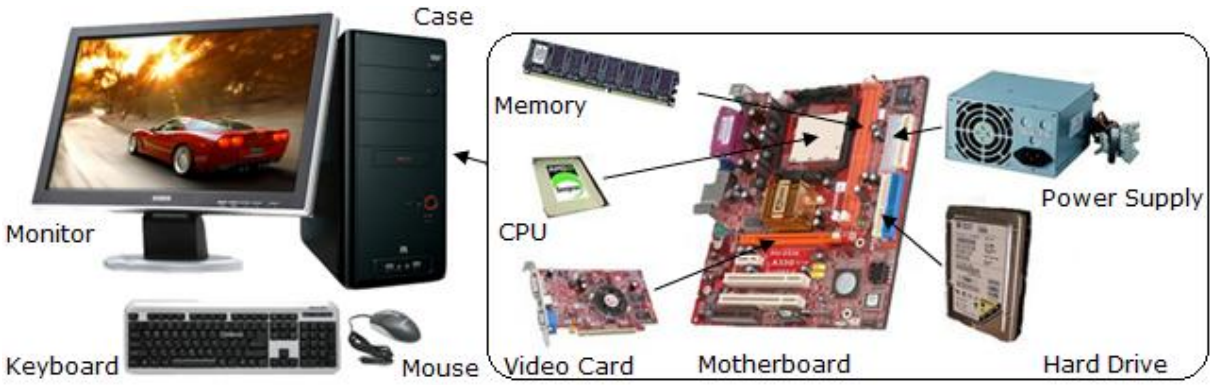

Figure 6.1: Main components in a computer. Adapted from limousincomputers and www.tncpcs.com.

- **The Case**, contains the following hardware components
  - **Central Processing Unit, CPU**: Basically the "brain" of the system. It carries out most of the instructions and does most of the computations. It will administer tasks to the GPU if it is utilized.
  - **Motherboard:** The centrepiece of the computer, it connects the components. It contains the interface between the CPU and the other components.
  - **Video Card:** It contains the Graphics Processing Unit, GPU, and a separate memory space dedicated to it. It is connected to the motherboard through a PSI express buss in the newest motherboards. Some computers may not have a dedicated Video Card for the GPU, but these GPU's are not suitable for general purpose programming.
  - **Random Access Memory, RAM**: stores data that is used by the CPU in all running applications on the computer.
  - **Hard Drive:** It stores data over a long period of time. The data is transferred to the RAM before it is used in the applications.
  - **Power SUpply, PSU**: It supplies power to the other hardware components in the case.
- **The monitor** is a visual interface between the user and the computer.
- **The keyboard and mouse** are used to issue commands to the computer.

HARDWARE	
Motherboard	HP 0A1Ch
Processor	2x Dual-Core AMD Opteron(tm) Processor 2220 (2.8GHz)
Video Card	NVIDIA GeForce GTX 285
RAM (DDR2)	4 x Samsung M3 93T5750EZA-CE6 (2GB)
Hard disk	2 x Seagate ST3500630AS (500 GB)
SOFTWARE	
Operating System	Microsoft Windows 7 Ultimate 6.01.7600 (x64)

Table 6.1: Hardware configuration for the test computer and operating system used.

6.2 The GPU

The two major producers of dedicated video cards are NVIDIA and ATI/AMD. Both of which have recognised that there is an increasing interest for using GPU's for general purpose programming and released API's, Application Programming Interface, especially designed for this. The major producers of CPU's are Intel and AMD. There are a large number of different models of both CPU's and GPU's with various number of processing cores and memory layout. The NVIDIA platform was chosen in this thesis since we had access one of an NVIDIA GeForce GTX 285 video card. An additional advantage is that NVIDIA's API, CUDA (which is described in section 6.3), is well documented and more commonly used than ATI/AMD's API, Brook+. AMD's 2.8GHz Opteron processor is used in the test computer and therefore used here as a specific example for comparison between the CPU and the GPU processor architecture.

The architecture of the GPU differs largely from the architecture of the CPU. The GPU is highly parallelized and the newest models contains hundreds of processing cores as apposed to the CPU which generally consist of 1,2, 4 or 8 powerful processing cores. The memory layout of the GPU is also unlike that of the CPU, which allows it to utilize the large number of processing cores more efficiently.

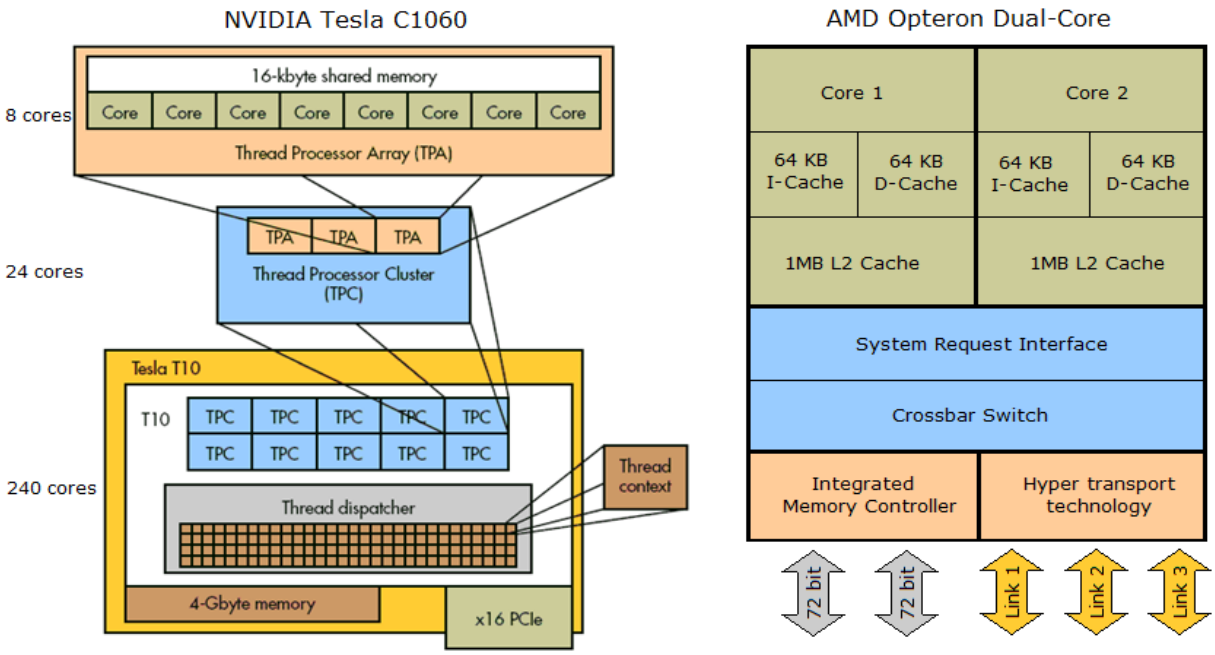

Figure 6.2: Schematic overview of the architecture of the NVIDIA Tesla C1060 video card, to the left, and an Opteron dual-core processor from AMD, to the right. The Tesla video card has 4 GB of dedicated memory (DRAM) and the video card is connected to the motherboard with a PCI express port (x 16 PCIe). The GPU can handle hundreds of threads, which are controlled by the thread dispatcher. The cores in the GPU are divided into groups of 8 cores which share a fast 16-kbyte on-chip memory space, called Thread Processor Arrays (TPA). This allows each group of cores to share data efficiently. There are a total of 30 thread processor arrays in the processor. To communicate to the other groups they have to use the DRAM. The Opteron processor has two powerful cores. They have access to their separate L1 and L2 caches. The CPU needs to use the RAM to store the data if it works on large systems and is in those cases limited by the memory bandwidth between the processor and the RAM. Adapted from images on electronicdesign.com and amd.com.

Both the GPU and the CPU can store small amounts of data on-chip, i.e. inside the processor. Data can be transferred very efficiently between the processor and these memory spaces. These memory spaces are often called caches in the CPU, and there can be several layers of caches with varying sizes. The caches are numbered L1, L2 ... etc. from the smallest and fastest to the slowest and largest cache. The GPU does also have several layers of on-chip memory. The on-chip memory is used to reduce so called latency problems. Latency is the amount of time it takes from the processor request data until it has access to it.

The specifications of the NVIDIA GeForce GTX 285 video card and the Opteron processor are presented in Table 6.2 and a schematic overview of the architecture of NVIDIA's Tesla C1060 and the Opteron processor is shown in Figure 6.2.

	SPECIFICATIONS (Video Card)
Graphics Processor	NVIDIA GeForce GTX 285
Number of GPU processors	1
Number of Streaming Processor Cores	240
Frequency of processor cores	1.48 GHz
Single Precision floating point performance (peak)	N/A GFLOP
Double Precision floating point performance (peak)	N/A GFLOP
Total Dedicated Memory (GDDR3)	1 GB
Memory Bandwidth	159 GB/s
Max Power Consumption	204 W
System Interface	PCI express x16 (gen 2.0)
Bandwidth (System Interface)	8 GB/s
	SPECIFICATIONS (CPU)
	Dual-Core AMD Opteron (tm) Processor 2220
Number of processing cores	2
Speed	2.8 GHz
Double Precision floating point performance (peak)	22.4 GFLOPS
Max Power Consumption	96.55W
Socket	Socket F
Integrated Data Cache	2x 64kB
L2 On-board Cache	2x 1MB
Memory Bandwidth	8.6 GB/s

Table 6.2: Specifications for the CPU and the GPU in the hardware configuration of the test computer.

The memory bandwidth is the rate at which data can be transferred from storage to the processor. It is usually measured in GigaBytes per second, GB/s. There are several benchmarks for measuring effective memory bandwidth. In this thesis the maximum effective bandwidth is measured by copying data from one location in the memory to another.

Floating point is a computer representation of a number. The double floating point needs twice as much space in memory as single floating point, which also yields twice the precision. FLoating point OPerations Per Second, FLOPS, is a performance measurement for a processor which shows the number of arithmetic operations a processor can carry out per second.

The key difference between the processors is the number of cores and the memory layout. The cores in the GPU are arranged into groups, called Thread Processor Arrays (TPA), which can communicate efficiently with each other. Tasks run by the GPU must be divided into a large number of processes, often called threads, to utilize all the cores in the GPU. The threads should be run in groups to exploit the efficient communication between the cores in one TPA. This is illustrated in Figure 6.3, the memory spaces the threads and the blocks have access to are also shown. Finite difference methods are very well suited for the GPU architecture.

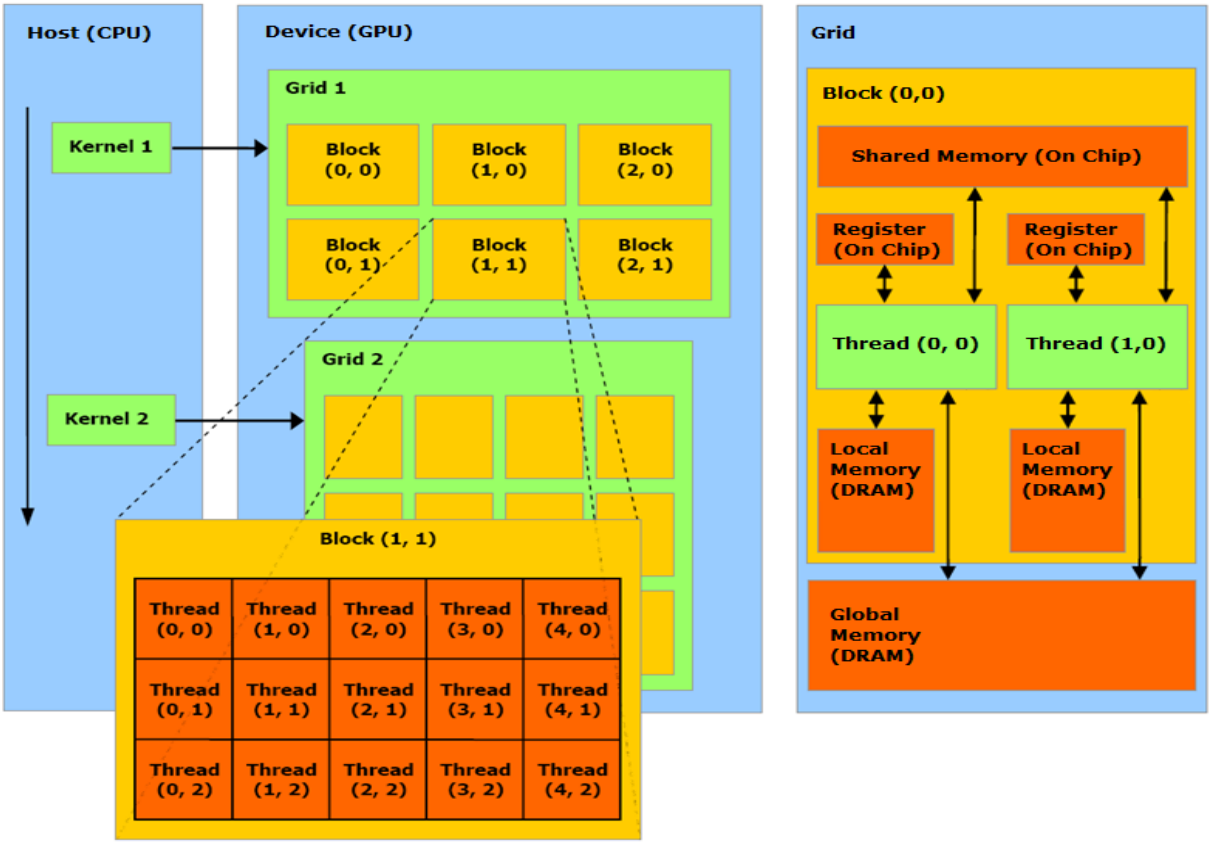

Figure 6.3: Adapted from www.ixbt.com. The host, i.e. the CPU, issues tasks that are to be solved by the GPU, called kernels. The kernels are divided into a large number of blocks that are handled by the TPA's. Each kernel must consist of 30 blocks or more to utilize all of them. The blocks consist of a series of threads which are handled by each of the 8 cores in the TPA. Each thread has access to a separate memory space (register), the shared memory for the TPA and the DRAM.

It is the division of the threads into groups which makes the development of algorithms for the GPU different from development of algorithms for the CPU. The thread switching on the GPU is also faster than on the CPU which means that implementations on the GPU benefits more from having a large number of light weight threads, i.e. simple processes.

Many algorithms can benefit from utilizing both the CPU and the GPU in the same implementations. These are typically algorithms that have portions of serial operations that

can be run efficiently on the powerful cores in the CPU and parallel portions that can be run on the hundreds of processors in the GPU. Combining the use of the GPU and the CPU in the same application is called heterogeneous computing.

6.3 CUDA

NVIDIA has developed Compute Unified Device Architecture, CUDA, and it was released in November 2006. CUDA exploits the parallel architecture of the Graphics Processing Unit to solve complex numerical problems. It allows the user to facilitate the computational power of the GPU for general-propose computing. Before CUDA the only API's available for making applications that tapped into the GPU resources were graphics APIs, such as OpenCL and DirectX. OpenCL and DirectX are very useful for working with graphics and developing computer games, but not suited for general-purpose computing.

The new API is supported by NVIDIAs graphics cards in the G8X series and onwards, which includes cards from the Fermi, Tesla, Quadro and GeForce series. The hardware in the newest video cards is designed to facilitate the use of CUDA.

CUDA enables the user to access the GPU through the programming language, C. NVIDIA's new Fermi series has support for C++, and NVIDIA is currently working on adding support for FORTRAN. A series of CUDA specific extensions is added to the C programming language that allows the user to run code on the GPU. Users that are familiar with the C programming structure should be able to learn CUDA syntax relatively quickly. The challenge is to find applications that can benefit from using the GPU and to develop codes that utilize the processor power efficiently.

One of the main advantages CUDA has over pre-existing APIs is that the memory access is a lot more flexible. Each thread may access any memory location on the graphics card as often as required at a certain cost. Threads in a block can cooperate efficiently with little penalty from latency since the architecture of the GPU is divided into blocks of processors which share a fast on-chip memory. This results in high effective memory bandwidth.

6.4 GPU Libraries for MATLAB

High-level programming languages such as MATLAB and Pyton are useful for rapid development and prototyping of algorithms. Third party developers have focused on accelerating functions in these languages using the GPU. CUDA provided the tools needed for rapid development of GPU accelerated functions for MATLAB and Pyton. Accelerated MATLAB functions were developed before CUDA using OpenCL, see Brodtkorb (2008), but only a handful of functions where supported.

Tech-X, Accelereyes and GP-you Group are three developers that have made MATLAB engines that support a large number of MATLAB functions. Tech-X and Accelereyes have commercialized their engines called GPUlib and Jacket, whereas GP-you Group distributes their engine called GPUmat as freeware[1]. Accelereyes' Jacket is used in this thesis since it has support for the largest number of MATLAB functions.

These GPU accelerated MATLAB engines allow the user to tap into the resources of the GPU through the user friendly interface of MATLAB. In this way the user can avoid the complexities of programming on a lower level programming language, such as C. This opens for rapid development of applications that make use of both the CPU and the GPU.

The algorithms are developed in the native M-language and run in the framework of MATLAB. Some specific engine commands are needed in addition to the normal MATLAB syntax to implement algorithms that utilize the GPU. The drawback of the CUDA based MATLAB engines is that they have support of only a fraction of the functions in MATLAB.

[1] Freeware is software which is available for use without cost or optional fees.

6.4.1 Implementation using MATLAB Engines that utilize the GPU

In this section we will focus on Accelereyes' engine, Jacket, since it is used for the implementations in this thesis. GPUmat and GPUlib engines do, however, work in much the same way. Jacket replaces some of the basic functions in MATLAB engine that run on the CPU. To use Jacket the user simply cast the variables into GPU data structures. This can be done with the functions listed in Table 6.3.

Jacket MATLAB function	Description	Example
gsingle	Casts a MATLAB matrix to a single precision floating point GPU matrix.	`A = gsingle(B);`
gdouble	Casts a MATLAB matrix to a double precision floating point GPU matrix.	`A = gdouble(B);`
glogical	Casts a MATLAB matrix to a binary GPU matrix. All non-zero values are set to '1'. The input matrix can be a GPU or CPU data type.	`A = glogical(B);` `A = glogical(0:4);`
guint32,gint32	Cast a MATLAB matrix to a signed and unsigned 32 bit integer GPU matrix respectively.	`A = gunit32(B);` `A = gint32(B);`
gzeros	Creates a matrix of zeros analogous to the MATLAB zeros function.	`A = gzeros(5);` `A = gzeros(2,6);`
gones	Creates a matrix of ones analogous to the MATLAB ones function.	`A = gones(5);` `A = gones([3 9]);`
geye	Creates an identity matrix analogous to the MATLAB eye.	`A = geye(5);`

Table 6.3: All included variables must be cast to GPU data structures before operations can run on the GPU; this is done with the listed functions. Notice that the library has support for double precision, which can be used by some of NVIDIA's top models.

All operations applied to GPU data structures are computed on the GPU, granted that Jacket has support for the function. Here is an example of how the code may be written to carry out an arithmetic operation using Jacket:

Normal MATLAB approach:	Jacket approach:

`A = rand(100);`
`B = rand(100);`
`C = A + B;`
	`A = rand(100);`
	`B = rand(100);`
	`A = gsingle(A);`
	`B = gsingle(B);`
	`C = A + B;`
	`C = single(C);`

Table 6.4: After the A and B matrices are given values they must be cast to GPU data structures when using Jacket, this is done with the gsingle statements. C will be calculated on the CPU in normal MATLAB, but in Jacket the computation is done on the GPU. When calculations are done on the GPU the matrix C will be a GPU data structure. It may be cast to a CPU data structure with the final statement.

Source codes written in for Jacket may be found in the Appendix, see chapter 11. For more extensive information see the Jacket User Guide[2].

[2] http://www.accelereyes.com/doc/JacketUserGuide.pdf.

6.5 Limitations of GPU Programming

It is clear that the GPU is a very powerful processor compared to the CPU. There is, however, a large range of applications where utilizing the GPU does not yield a speed up. The applications must be;

- **Parallelizable**. The application must be parallelizable as outlined in section 6.2 since each of the cores in the GPU is considerably weaker than each of the cores in the CPU. The application must be dividable into hundreds of threads that can run concurrently to utilize the full power of the GPU.
- **Partitioned**. The threads should be divided into blocks to exploit the shared memory spaces to reduce latency problems. There should be a multiple of 30 blocks in an application to utilize all thread processor arrays and there should be a multiple of 8 numbers of threads in each block.

In addition to the requirements on the applications there are a few technical issues that hinders the implementation process and execution of the applications;

- **Single precision**. The Tesla series was the first GPU's that had support for double precision floating points. It could be used, but it resulted in a considerable performance loss. The new Fermi series does, however, have full support for double precision.
- **Bandwidth between the CPU and the GPU**. Many applications rely on computations on both the CPU and the GPU. These applications are limited by the rate at which data can be transferred between the processors.
- **Programming limitations in CUDA**. The version of C that is used in CUDA does not have support for recursion of function pointers.
- **No unified standard**. NVIDIA and ATI/AMD does not support the same API for general purpose programming.

7 Standard Finite Difference Implementations on the GPU

Performance gain from utilizing the GPU has been shown in a variety of research fields, and it is clear from the specifications that there computational power of the GPU is immense. We wanted to carry out a few preliminary tests to get an idea of the potential performance gain when utilizing the GPU in our applications. The test was a strait forward implementation of the heat diffusion equation in two dimensions, i.e. without the use of multigrid.

As for traditional applications on the CPU the user can choose to either implement the applications using an efficient low-level language such as C or user-friendly high-level languages such as MATLAB or Python. Development time is considerably faster in high-level languages, but at it comes at the cost of severe performance loss.

The heat diffusion equation was implemented in MATLAB and C for tests on the CPU and in CUDA and Accelereyes' MATLAB engine Jacket for tests of the GPU. It was clear early on that implementations using Jacked suffered severe performance loss when indexing was used in the vector notations. A performance enhancement was achieved by using convolution for the homogeneous problems.

7.1 Heat Diffusion Equation

Implementations of the transient heat diffusion equation for both heterogeneous and homogeneous systems are carried out. The discretization of the heat diffusion equations are done as presented in chapter 0, which resulted in the following stencils for the homogeneous and the heterogeneous materials respectively

$$\frac{k}{c_p\rho}\left(\frac{\boldsymbol{T}^l_{i+1j}-2\boldsymbol{T}^l_{ij}+\boldsymbol{T}^l_{i-1j}}{\Delta x^2}+\frac{\boldsymbol{T}^l_{ij+1}-2\boldsymbol{T}^l_{ij}+\boldsymbol{T}^l_{ij-1}}{\Delta y^2}\right)+q_{ij}\approx\left(\frac{\boldsymbol{T}^{l+1}_{ij}-\boldsymbol{T}^l_{ij}}{\Delta t}\right) \tag{7.1}$$

$$\frac{k_{i+\frac{1}{2}j}\boldsymbol{T}^l_{i+1j}-(k_{i+\frac{1}{2}j}+k_{i-\frac{1}{2}j})\boldsymbol{T}^l_{ij}-k_{i-\frac{1}{2}j}\boldsymbol{T}^l_{i-1j}}{\Delta x^2}+ \tag{7.2}$$

$$\frac{k_{ij+\frac{1}{2}}\boldsymbol{T}^l_{ij+1}-(k_{ij+\frac{1}{2}}+k_{ij-\frac{1}{2}})\boldsymbol{T}^l_{ij}-k_{ij-\frac{1}{2}}\boldsymbol{T}^l_{ij-1}}{\Delta y^2}+\frac{c_p\rho}{k}q_{ij}=\frac{c_p\rho}{k}\left(\frac{\boldsymbol{T}^l_{ij}-\boldsymbol{T}^{l-1}_{ij}}{\Delta t}\right)$$

The material properties c_p and ρ are set equal to unity and there are no heat sources or sinks in the system. The thermal conductivity is set equal to unity for the homogeneous system.

As mentioned in chapter 0 the thermal conductivity values in the heterogeneous system are either stored where they are needed, i.e. staggered grid, or they are defined in the nodes and averages are used in the calculations. The first approach is more memory bandwidth intensive and the second one requires that more arithmetic operations are carried out. The performance of an implementation is capped by one of these two requirements, and the type of implementation can therefore affect the results.

7.1.1 Implementation on the GPU

The complexity of adapting the algorithm to exploit the full potential GPU, i.e. manually allocating memory ant utilizing processing cores, can be avoided with the use of Jacket. For the CUDA implementations, however, the adaptations from the CPU implementation must be made manually.

This means that for the CUDA implementations the system must manually be divided into blocks and threads. Dividing the algorithm into blocks is often referred to as partitioning. Each block consists of a series of threads that is handled by the cores in a thread processor array. The system should be divided into blocks that consist of a multiple of 8 nodes, which allows for all 8 cores in thread processor array to be utilized simultaneously. There is a total of 30 tread processor arrays, which means that it would be preferable to have a multiple of 30 blocks in the system. In the CUDA implementations that were made for these tests the blocks consists of 16 x 16 nodes. It is clear from the stencils that the thread processor array must have access to the values in the nodes nearest to the domain it is working on; these nodes are called the "halo" of the small domain. The idea is illustrated in Figure 7.1.

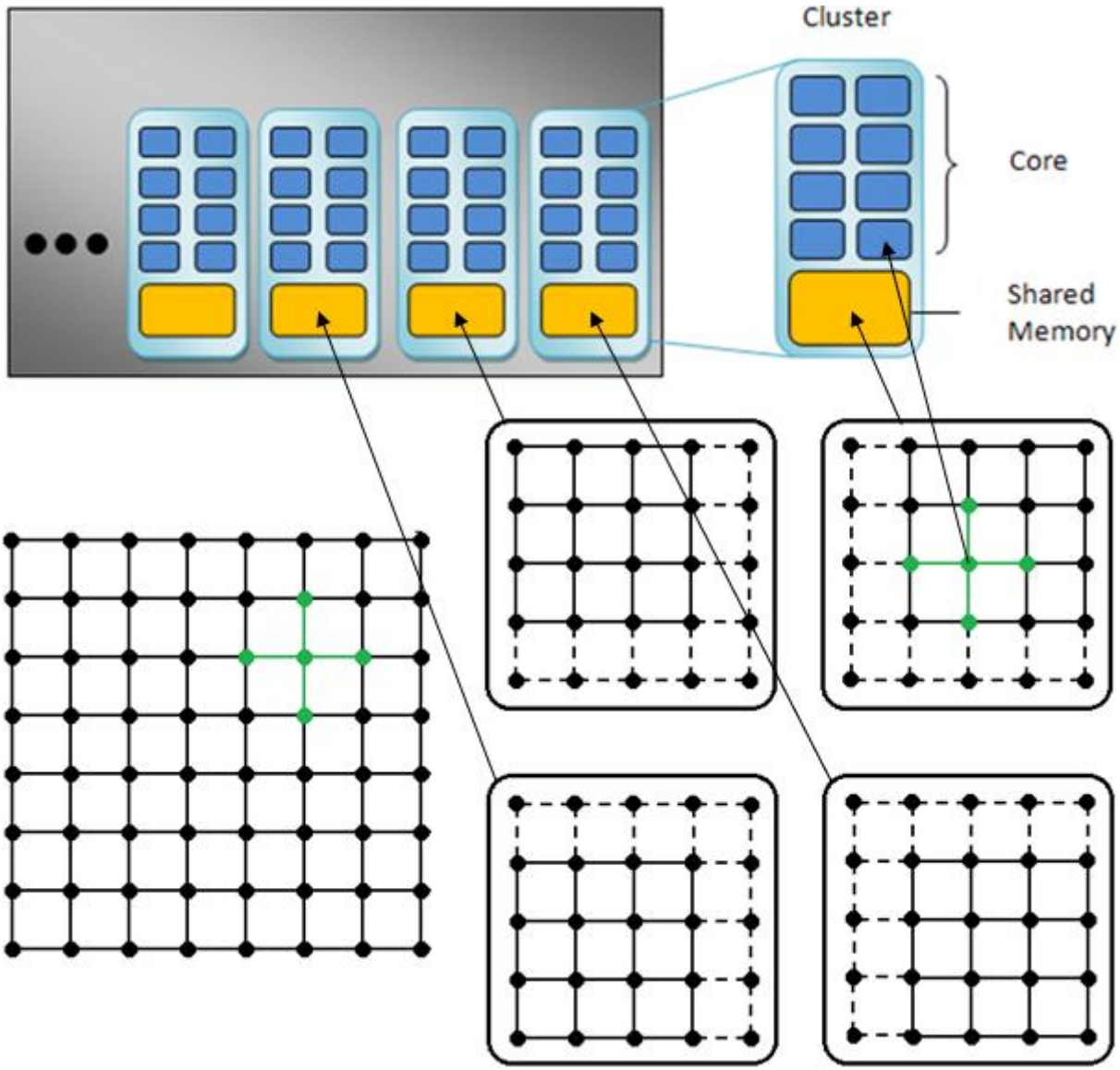

Figure 7.1: The heat diffusion equation is discretized on the grid shown in the bottom-left corner of the figure. To solve the system using the GPU it is divided into smaller domains, each of which is solved by one thread processor array. The stencil is applied to each node, illustrated with a green cross in the figure. The calculations for each node are carried out by one of the cores. Each thread processor array has access to their "halo".

7.2 Performance Measurements

The bottle neck of the implementations is the memory bandwidth between the RAM/DRAM to the processor. The effective memory bandwidth is calculated from the optimal memory load, which is the minimum amount of bytes that must be read and written by the processor to carry out the calculations, and the time it took to carry out them out. The amount of time it takes to carry out the equations is normalized by the number of time steps that the temperatures are calculated for.

Based on the stencils, see equation 6 and (7.2) for the homogeneous and heterogeneous materials respectively, the optimal memory load can be found. A two dimensional grid, where the number of grid points in the x- and y-direction is nx and ny respectively. For the homogeneous systems all material properties are set to unity and are therefore not needed for the computations. This means that the data needed to carry out the calculations are the temperature values in the nodes.

For heterogeneous materials the thermal conductivity values are needed in addition to the temperature values. Using a staggered grid requires that about twice as many thermal conductivity values must be transferred to the processor than if the thermal conductivity values are stored in the nodes. Only the temperature values must be written to memory after the calculations for both the heterogeneous and the homogeneous system since the thermal conductivity values are unaffected by the computations.

The optimal memory load is calculated in bytes. Single precision floating point numbers occupies 32 bits on binary format, i.e. 4 bytes. The memory load for the different systems is listed in Table 7.1.

System	Optimal memory load (bytes)
Homogeneous	$8 \cdot nx \cdot ny$
Heterogeneous (Staggered grid)	$\sim 16 \cdot nx \cdot ny$
Heterogeneous (On nodes)	$12 \cdot nx \cdot ny$

Table 7.1: The optimal memory load for the heat diffusion equation on two dimensions, the number of grid points is $nx \times ny$. For the heterogeneous systems the thermal conductivity values are stored either at the points where they are needed, staggered grid in the table, or at the nodes, on nodes in the table.

7.3 Results

7.3.1 Heat Diffusion

Utilizing the GPU, Graphics Processing Unit, instead of the CPU, Central Processing Unit, resulted in considerably more efficient implementation in both MATLAB and C for all test systems. Implementations on the GPU were up to 20–30 times more efficient than the equivalent implementations on the CPU using C for the large systems. The MATLAB implementations were about 15–25 times more efficient on the GPU than on the CPU. The Jacket implementations are about 4 times faster than the C implementations for the GPU. The ratios of execution time between the different implementations are presented in Table 7.2 for the homogeneous system.

Homogeneous					
N	C: GPU vs. CPU	MATLAB: GPU vs. CPU	CPU: C vs. MATLAB	GPU: C vs. MATLAB	C on CPU vs. MATLAB on GPU
16	0.16	0.06	17.47	46.43	294.66
32	0.54	0.09	8.02	47.62	88.79
64	0.69	0.19	12.25	43.63	63.15
128	3.28	0.54	8.27	49.92	15.23
256	9.42	1.90	6.52	32.31	3.43
512	26.20	7.59	4.33	14.94	0.57
1024	30.82	15.72	3.85	7.54	0.24
2096	23.48	23.14	5.07	5.15	0.22
4096	19.38	24.70	5.94	4.66	0.24

Table 7.2: Comparison between the implementations in both C and MATLAB show that there is a large performance gain when utilizing the GPU.

The efficiency of the different implementations of the heat diffusion equation for a homogeneous systems are presented in Figure 7.2, it shows the number of grid points the stencil is calculated for per second for various grid sizes. Notice that there is a logarithmic scale on the y-axis.

The implementations on the GPU are considerably more efficient than the implementations on the CPU for large systems; the situation is, however, reversed for the small systems. For the efficiency of the GPU implementations to surpass those on the CPU the system size must be over 64 x 64 for the C implementations and 256 x 256 for the MATLAB implementations.

Cache effects cause the efficiency to peak for grid sizes of about 128 x 128 for the CPU. For this grid size and smaller ones all data required for the calculations will fit in the cache. This reduces the latency considerably. For the GPU implementations we observe that the efficiency of codes improves rapidly for the smaller grids, but stagnates for grid sizes larger than 512 x 512 for C and 1024 x1024 for MATLAB.

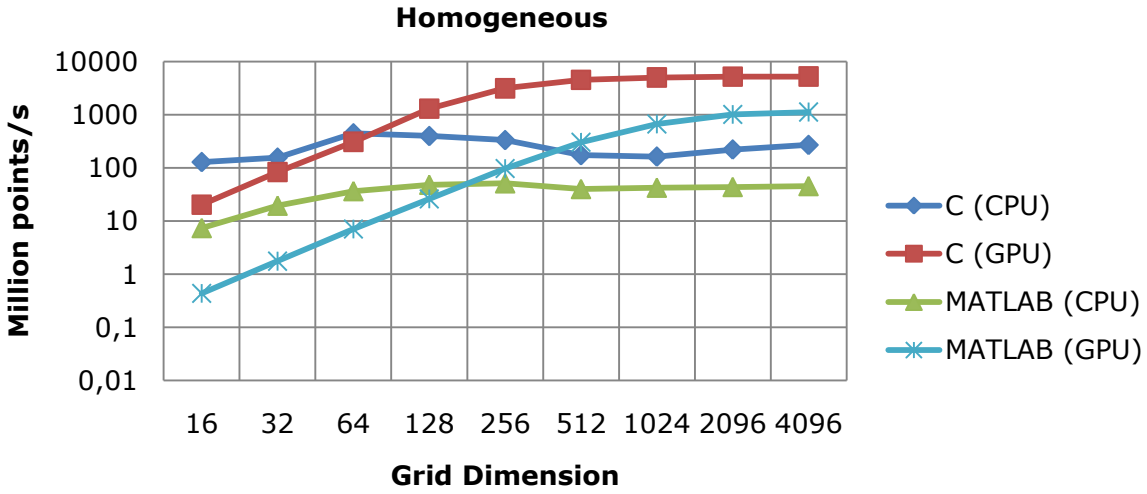

Figure 7.2: For small systems the CPU implementations are faster than the GPU implementations, but the situation is reversed for the large ones. The CPU implementations have a peak performance on grid sizes of about 128 x 128 due to cache effects. The efficiency of the GPU implementations stagnates for large systems.

The tests done on the heterogeneous systems are presented in Figure 7.3 and Figure 7.4. The performance peak for the CPU is on lower grid sizes since the material property values take up space in the cache. For the CUDA implementations the same curves are observed as for the homogeneous system. The efficiency of the Jacket implementation is reduced since indexing must be used to solve the problem. There is no clear difference in the performance of the two implementations of the heterogeneous system, but for the MATLAB implementations slightly more efficient when the thermal coefficients are defined on a staggered grid and the C implementations are slightly more efficient when they are defined at the nodes.

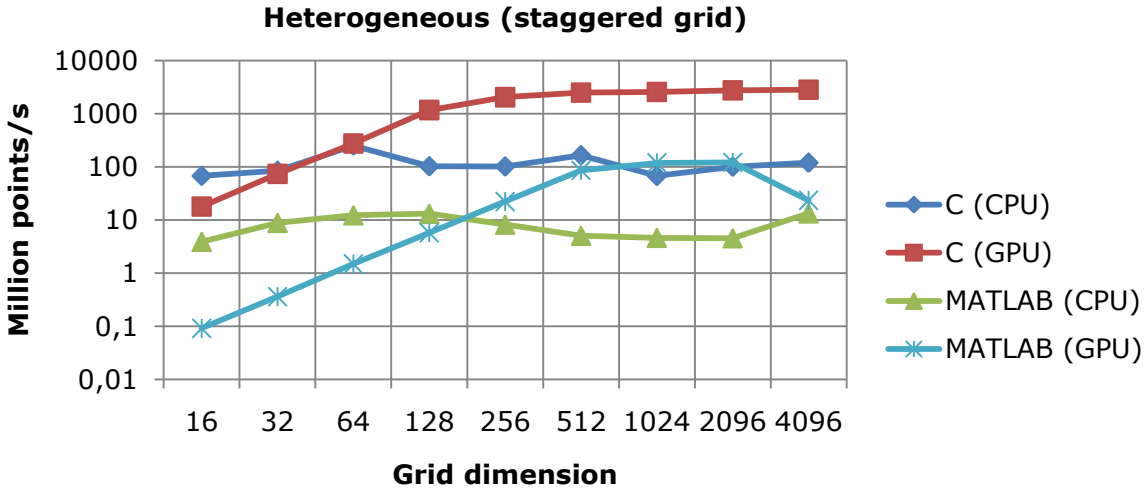

Figure 7.3: In these implementations the material properties are stored on a staggered grid; this requires two arrays with material properties. For the heterogeneous systems we see the same trend as for the homogeneous systems with the exception of the Jacket implementation were the efficiency is reduced, see Figure 7.2. The peak performance for the CPU is at a lower grid size for this system since the material properties takes up some space in the cache.

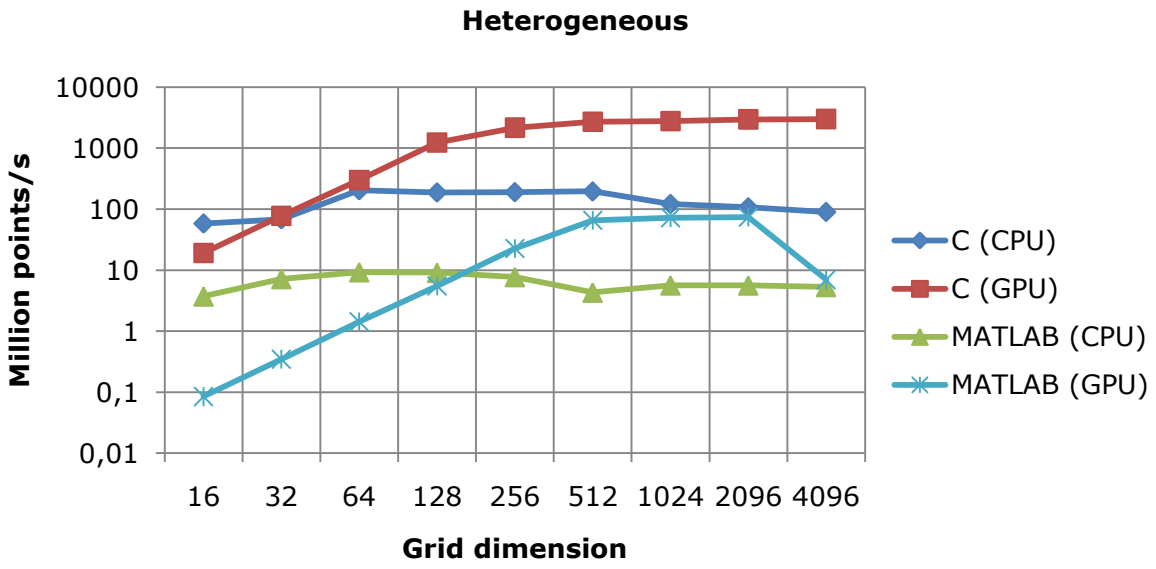

Figure 7.4: In this implementation the material properties are defined on the grid points, this requires one array with material properties. Here we see the same trends as for the staggered grid system, see Figure 7.3. The peak performance for this system on the CPU is wider for this system since it is more depended on floating point operations than on memory bandwidth compared to the heterogeneous system where a staggered grid is used.

The efficiency of the implementations is capped by the memory bandwidth. The measured effective memory bandwidth is presented for the different implementations for the CPU and the GPU in Table 7.3.

	Effective Memory Bandwidth (GB/s)			
	C		MATLAB	
	CPU	GPU	CPU	GPU
Homogeneous	1.7	41.1	0.35	7.43
Heterogeneous (off nodes)	1.5	43.2	0.13	0.98
Heterogeneous (on nodes)	1.3	34.9	0.07	0.61

Table 7.3: The values show the ratio between the effective memory bandwidth and the theoretical memory bandwidth in percentages. The Opteron CPU the theoretical memory bandwidth is 2.8 GB/s and for the Tesla C1060 GPU the maximum measured memory bandwidth is 75 GB/s. All values are averages for the three largest grid sizes for the respective implementations.

	Million Points per Second			
	C		MATLAB	
	CPU	GPU	CPU	GPU
Homogeneous	217	5131	44	929
Heterogeneous (off nodes)	95	2699	8	61
Heterogeneous (on nodes)	106	2912	6	58

Table 7.4: All values are averages for the three largest grid sizes for the respective implementations. Points per second are the number of grid points the diffusion stencil is applied to per second. Here we clearly see the speedups we achieved by executing the codes on the GPU instead of the CPU. For the C implementations it is slightly more effective to define the material properties on the grid points and for the MATLAB implementations it is the other way around.

Table 7.4 shows how many grid points the heat diffusion was calculated for per second. From the values we clearly see that the GPU implementations are faster than the CPU implementations. The homogeneous code is far more efficient than the implementation of diffusion in a inhomogeneous system. For the heterogeneous implementations in C with the material properties defined on the nodes we calculate the diffusion for more points per second, but the difference is quite small. For the MATLAB implementations we see the opposite trend. This is probably due to the difference in how the data is placed in the fast on chip memory spaces.

7.4 Conclusion

Implementations are considerably faster to execute on the GPU instead of on the CPU for both MATLAB and C implementations. For the C implementations the execution time is up to 30 times faster on the GPU and for the MATLAB implementation the execution time is about 25 times faster for the GPU.

The MATLAB implementation of diffusion in homogeneous systems is far more efficient than the implementations of diffusion in inhomogeneous systems. This is a result of them being implemented with convolution which is very efficient in both native MATLAB and Jacket.

The Jacket implementation for the homogeneous systems is 4 – 5 times faster than the equivalent C implementation. Despite the speedups found for the GPU implementations in MATLAB the execution time not faster than that for the CPU implementations in C for inhomogeneous systems.

8 Applications

8.1 Poisson Solver

In this thesis an efficient Poisson solver is developed. It is implemented using the multigrid algorithm and accelerated by utilizing the GPU. It can be used as a solver or a part of a solver for a variety of problems in earth sciences as discussed previously. In the following section it is used to solve for the pressure in a simulation of porous convection. The Poisson solver implemented in this thesis is very efficient and it can solve a system of size 257 cubed, that is roughly 16 million unknowns, in less than a second. This is possible through the utilization of the GPU. The function that was implemented for solving the Poisson problem with Dirichlet boundary conditions is relatively short, i.e. 75 lines, and transparent. It can be found in the appendix, section 11.1.

Dirichlet boundary conditions are imposed on all walls in the test system used in this application. The efficiency of the algorithm that utilizes the GPU is tested for various system sizes and the results are compared with a similar implementation for the CPU. The speedups achieved from utilizing the GPU is immense, see Table 8.1. Implementations utilizing the GPU are about 60 to 70 times faster than the corresponding implementations on the CPU.

N	CPU	GPU	Speed up
129x129x129	8.55 sec	0.13 sec	x 66
257x257x257	69.36 sec	0.93 sec	x 74
257x257x513	143.30 sec	2.32 sec	x 62
321x321x257	119.25 sec	1.74 sec	x 68

Table 8.1: The computation time for solving a Poisson problem on systems with various resolutions is shown in the table and the speedups is calculated.

The GPUs that are available to day is best suited for use of single precision. Some have support for double precision, but the performance when using double precision is not satisfactory. Implementations written with double precision for the CPU are about half as efficient as corresponding implementations using single precision. The performances loss from using double precision on the GPU is far greater than this. NVIDIA does, however, claim that full support for double precision is going to be fully supported the next generation of video cards.

The multigrid method is an iterative method and the quality of the approximation of the solution found by it is measured with the relative residual, i.e. the ratio between the initial residual and the residual at the current approximation. As the number of operations needed to calculate the solution increases the round off errors in the solution increases. For traditional iterative methods such as Gauss-Seidel or conjugate gradient method the number of iteration needed to find a solution of the problem increases as the system size is increased. This can make the solution found for large systems unusable due to large round off errors, especially when single precision is used. The multigrid method is likely to be less vulnerable by round off errors since the number of iterations needed to achieve the selected accuracy is relatively unaffected by the system size. This means that the number of mathematical operations applied to the value in each grid point is independent of the system size.

Convergence rates for different system sizes using single precision are shown in the graph in Figure 8.1, and the convergence rate is compared to the convergence rate for a system solved with double precision. The rate of convergence and accuracy of the final solution appear to be relatively unaffected by the system size. The accuracy that can be achieved is as expected much smaller when using single precision than when using double precision, but the rate of convergence is the unaffected. The system sizes that could be tested were limited by the amount of dedicated memory on the graphics card. The NVIDIA GeForce GTX 285 video card that is used in these tests has 1 GB of dedicated memory. It is therefore not possible to test the level of accuracy that can be achieved for larger system than those presented in the graph.

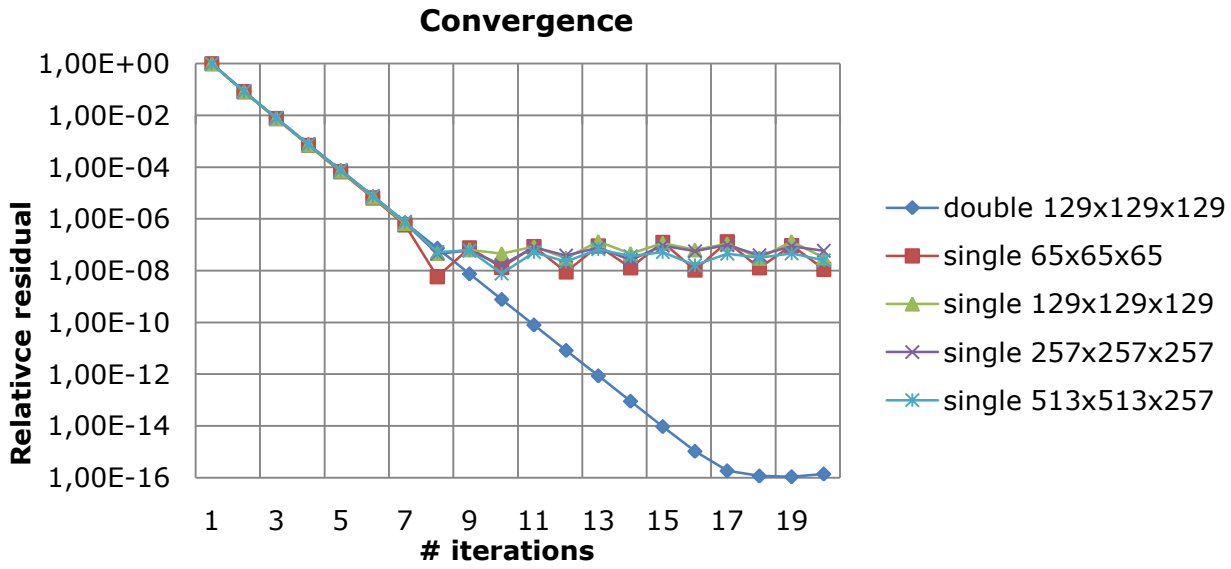

Figure 8.1: Convergence rates for different system sizes using single and double precision. The convergence does not deteriorate as the system size increases. The accuracy that can be achieved is of the same magnitude for all system sizes that can be solved on the NVIDIA GeForce GTX 285 video card with the multigrid algorithm. In these tests 2 pre- and 2 post-smoothing steps is used.

Video cards produced for scientific purposes such as the Tesla c1060 does, however, have 4 GB of dedicated memory and could therefore be used to solve larger systems. The price on the video cards produced for games are considerably less than the price of the video cards made for scientific calculations, i.e. the NVIDIA GeForce GTX 285 video card costs $450 whereas the Tesla c1060 costs $1 300. The extra expense can be justified since it is likely that larger systems can be solved efficiently with an acceptable accuracy. To get equivalent efficiency using CPU's would require a cluster of processors, which is far more expensive and power consuming than a graphics card that can be utilized directly from a desktop computer.

8.2 Porous Convection

Sedimentary rocks are naturally porous. The matrix porosity together with fractures and voids in the rock allow water, oil and gas to migrate through the earths crust. To consider the rock as homogeneous the factures and voids must be relatively evenly distributed and relatively small compared to the scale of the flow.

Flow in a porous medium is driven by pressure gradients, and the flow is characterized by the permeability of the rock and the viscosity of the fluid. The permeability is a measure of the porous rocks ability to transmit fluids, which is the dependent on the number and size voids and pathways in the rock. The viscosity is a measure of a fluids resistance to being deformed.

The flow through a porous material is in many cases found to be proportional to the pressure gradient and the permeability, and inversely proportional to the viscosity of the fluid migrating through it. Fluid flow that exhibits this behaviour can be described by the Darcy law. It can be used to model a variety of problems in earth sciences, such as flow of hydrocarbons in reservoirs, flow of groundwater and migration of magma.

It is believed that the heat from magma bodies triggers hydrothermal circulation. The heated fluid expands, and due to the decreased density, it raises up trough the earths crust, whereas the cooler fluids will sink. This process is called convection. In this application convection in a porous material is simulated and other transport phenomena.

The equations used here for simulating flow though porous media are well documented in literature see for example Turcotte (2002). In the simulations incompressible flow is assumed, meaning that the divergence of the velocity ($\bar{V}$) is zero

$$\frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y} + \frac{\partial V_z}{\partial z} = 0 \tag{8.1}$$

We do, however, allow for thermal expansion resulting in buoyancy driven flow. The model includes transfer of heat and the conservation of energy law therefore applies, for a homogeneous material it is written as

$$\frac{\partial T}{\partial t} + \left(V_x \frac{\partial T}{\partial x} + V_y \frac{\partial T}{\partial y} + V_z \frac{\partial T}{\partial z}\right) = \lambda\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{8.2}$$

Where V_x, V_y and V_z is the velocity components and λ is the thermal diffusivity constant (which has the units $[m^2/s]$). The conservation of energy law is a coupled advection and transient heat diffusion equation. The fluid is driven by the pressure gradient and its velocity is found with Darcy's law. It is found to be given by the following equations in 3D

$$V_x = -\frac{\kappa}{\mu}\frac{\partial P}{\partial x} \tag{8.3}$$

$$V_y = -\frac{\kappa}{\mu}\frac{\partial P}{\partial y} \tag{8.4}$$

$$V_z = -\frac{\kappa}{\mu}\left(\frac{\partial P}{\partial z} - \alpha_T \rho g T\right) \tag{8.5}$$

Where κ is the permeability and μ is the dynamic viscosity, α is the thermal expansion coefficient, g is the gravitational constant and P is the pressure.

Equations 1, (8.2), (8.3), (8.4) and (8.5) are the governing equations for the problem. These equations are non-dimensionalized based on the following non-dimensional quantities

$$T^* = \frac{T}{\Delta T} \tag{8.6}$$

$$V_x^* = \frac{V_x}{\lambda}H, V_y^* = \frac{V_x}{\lambda}H, V_z^* = \frac{V_x}{\lambda}H \tag{8.7}$$

$$dx^* = \frac{dx}{H}, dy^* = \frac{dy}{H}, dz^* = \frac{dz}{H} \tag{8.8}$$

$$dt^* = \frac{dt}{H^2}\lambda \tag{8.9}$$

$$Ra = \frac{\alpha_T \rho g H \kappa \Delta T}{\mu \lambda} \tag{8.10}$$

Where H is the height of the system, and Ra is the Rayleigh number. For small Rayleigh numbers the heat transfer is mainly driven by convection whereas for large Rayleigh number it is mainly driven by diffusion. Rewriting equation 1, (8.2), (8.3), (8.4) and (8.5) with the dimensionless variables yields

$$\frac{\partial^2 P}{\partial x^{*2}} + \frac{\partial^2 P}{\partial y^{*2}} + \frac{\partial^2 P}{\partial z^{*2}} = Ra\frac{\partial T}{\partial z^*} \tag{8.11}$$

$$\frac{\partial T^*}{\partial t^*} + \left(V_x^*\frac{\partial T^*}{\partial x^*} + V_y^*\frac{\partial T^*}{\partial y^*} + V_z^*\frac{\partial T^*}{\partial z^*}\right) = \left(\frac{\partial^2 T^*}{\partial x^{*2}} + \frac{\partial^2 T^*}{\partial y^{*2}} + \frac{\partial^2 T^*}{\partial z^{*2}}\right) \tag{8.12}$$

$$V_x = -\frac{\partial P^*}{\partial x^*} \tag{8.13}$$

$$V_y = -\frac{\partial P^*}{\partial y^*} \tag{8.14}$$

$$V_z = -\left(\frac{\partial P^*}{\partial z^*} - RaT\right) \tag{8.15}$$

The equations are solved for the system which is shown in Figure 8.2. The transient diffusion is decoupled from the advection in equation (8.12). This means that the primary variables in the problem are temperature and pressure. The velocities directly with equation (8.13), (8.14) and (8.15) which means that boundary conditions can not be applied directly on the velocities.

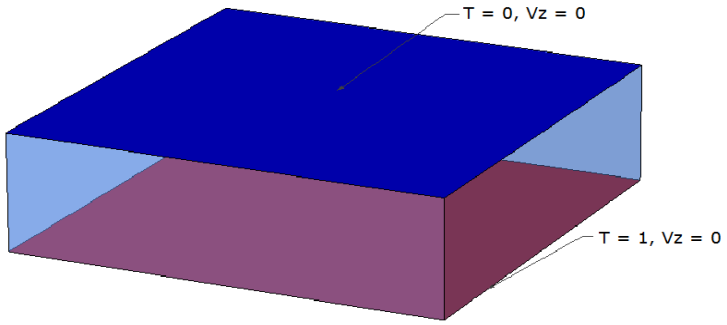

Figure 8.2: The system setup for the simulations. The temperature is one at the bottom wall and zero on the top wall, the velocity in the z-direction at both of them is zero. There are periodic boundary conditions on all lateral walls.

To solve for the porous flow both pressure and temperature is discretized in the following way

$$P = \sum_{j=1}^{n} N_j P_j \tag{8.16}$$

$$T = \sum_{j=1}^{n} N_j T_j \tag{8.17}$$

Where $N_1, N_2, .., N_n$ are shape functions and $T_1, T_2, T_3, \ldots$ and $P_1, P_2, P_3, \ldots$ are the temperature and the pressure in the nodes. Shape functions are simple functions we use to approximate the variables inside each of the elements.

To solve the problem we want to evaluate the integral of the weighted residual. For equation (8.11) this yields

$$\int_\Omega w \frac{\partial^2 P}{\partial x^{*2}} + \frac{\partial^2 P}{\partial y^{*2}} + \frac{\partial^2 P}{\partial z^{*2}} dxdydz = Ra \int_\Omega w \frac{\partial T}{\partial z^*} dxdydz \tag{8.18}$$

The Ω implies that the integral should be taken over the whole domain. As weighting functions, w, we use the shape functions $N_1, N_2, .., N_n$, i.e. Galerkin's method. Appling this and substitution of the pressure and temperature by their discretized version yields

$$\int_\Omega N_i \left(\frac{\partial^2 N_j}{\partial x^{*2}} + \frac{\partial^2 N_j}{\partial y^{*2}} + \frac{\partial^2 N_j}{\partial z^{*2}} \right) P_j \, dxdydz = Ra \int_\Omega N_i \frac{\partial N_j}{\partial z^*} T_j \, dxdydz \tag{8.19}$$

The second derivative of the shape function must differ from zero for this integral to be meaningful. By applying integration by part we may reduce the constraints on the shape functions. Applying this technique yields,

$$\begin{aligned} &- \int_\Omega \left(\frac{\partial N_i}{\partial x^*} \frac{\partial N_j}{\partial x^*} \right) P_j \, dxdydz + \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial x^*} \right) P_j \, dxdydz \\ &- \int_\Omega \left(\frac{\partial N_i}{\partial y^*} \frac{\partial N_j}{\partial y^*} \right) P_j \, dxdydz + \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial y^*} \right) P_j \, dxdydz \\ &- \int_\Omega \left(\frac{\partial N_i}{\partial z^*} \frac{\partial N_j}{\partial z^*} \right) P_j \, dxdydz + \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial z^*} \right) P_j \, dxdydz \\ &= -Ra \int_\Omega \frac{\partial N_i}{\partial z^*} N_j T_j \, dxdydz + Ra \int_{\mathrm{d}\Omega} N_i N_j T_j \, dxdydz \end{aligned} \tag{8.20}$$

$$\begin{aligned} &\Rightarrow \int_\Omega \left(\frac{\partial N_i}{\partial x^*} \frac{\partial N_j}{\partial x^*} \right) P_j \, dxdydz + \int_\Omega \left(\frac{\partial N_i}{\partial y^*} \frac{\partial N_j}{\partial y^*} \right) P_j \, dxdydz + \int_\Omega \left(\frac{\partial N_i}{\partial z^*} \frac{\partial N_j}{\partial z^*} \right) P_j \, dxdydz \\ &= Ra \int_\Omega \frac{\partial N_i}{\partial z^*} N_j T_j \, dxdydz + \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial x^*} \right) P_j \, dxdydz + \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial y^*} \right) P_j \, dxdydz \\ &+ \int_{\mathrm{d}\Omega} \left(N_i \frac{\partial N_j}{\partial z^*} \right) P_j \, dxdydz - Ra \int_{\mathrm{d}\Omega} N_i N_j T_j \, dxdydz \end{aligned} \tag{8.21}$$

The integral over $d\Omega$ means that the integral is taken over the boundaries. If we discretize and take the integral over the weighted residuals of the velocities in the x- y- and z-direction, see equation (8.13), (8.14) and (8.15), we find the following expressions

$$V_x = - \int_\Omega \left(N_i \frac{\partial N_j}{\partial x^*} \right) P_j \, dxdydz \tag{8.22}$$

$$V_y = - \int_\Omega \left(N_i \frac{\partial N_j}{\partial y^*} \right) P_j \, dxdydz \tag{8.23}$$

$$V_z = - \int_\Omega \left(N_i \frac{\partial N_j}{\partial z^*} \right) P_j \, dxdydz + Ra \int_{\mathrm{d}\Omega} N_i N_j T_j \, dxdydz \tag{8.24}$$

As described in Figure 8.2, the velocity in the z-direction is set to zero on the top and bottom boundary and periodic boundary conditions is applied to the lateral walls. This means that the last four terms in equation (8.21) will cancel out. This result in the following discretized version of equation (8.11)

$$\int_{\Omega}\left(\frac{\partial N_i}{\partial x^*}\frac{\partial N_j}{\partial x^*}+\frac{\partial N_i}{\partial y^*}\frac{\partial N_j}{\partial y^*}+\frac{\partial N_i}{\partial z^*}\frac{\partial N_j}{\partial z^*}\right)P_j \; dxdydz = Ra\int_{\Omega}\frac{\partial N_i}{\partial z^*}N_j T_j \; dxdydz \tag{8.25}$$

The multigrid algorithm requires that the stencils are assembled into compact stencils; this can be done when the finite element discretization is done on a regular grid.

In the implementation the advection and the diffusion is handled in two separate steps. The first step is to calculate the advection directly by using velocities found with equation (8.22), (8.23) and (8.24). In the second step diffusion is calculated by with the following equation

$$\frac{\partial^2 T^*}{\partial {x^*}^2}+\frac{\partial^2 T^*}{\partial {y^*}^2}+\frac{\partial^2 T^*}{\partial {z^*}^2}=\frac{\partial T^*}{\partial t^*} \tag{8.26}$$

A finite difference discretization is applied to the heat diffusion equation as described in section 2.2.

8.2.1 Implementation

To implementation of the solver for the porous convection consist of four main steps, which is

1. Solve the pressure equation (8.25).
2. Find the velocities implicitly with equation (8.22), (8.23) and (8.24).
3. Calculate the advection.
4. Solve the heat diffusion equation (8.26).

Before the pressure equation can be solved the right hand side of the equation must be calculated. It is found by applying the assembled stencil based on the finite element discretization of the equation. The pressure equation is solved using the implementation of the multigrid algorithm as described in section 5.3. Both of these are implemented using convolution for the heaviest calculations which is efficiently calculated by the GPU.

The velocities are using a central finite difference approximation of the derivatives of the pressure in equation (8.13), (8.14) and (8.15).

Implicit markers are used to calculate the advection. The idea is to find the position where the point that now is in the node would have been before the advection. Interpolation is used to find the value in that point and this value is set in the node. The built in function in MATLAB used for the interpolation is not supported Jacket and this calculations must therefore be done with the use of the CPU. Developing an algorithm to solve the advection on the GPU would be the natural next step in the further development of the code.

The heat diffusion equation is well conditioned and can be solved efficiently with the use of a few iterations with the Jacobi smoother.

8.2.2 Results

Porous convection is one of the main processes for heat transfer in sedimentary basins. The influence of the different parameters characterizing the fluid flow in sedimentary basins can be better understood by systematically analysing and quantifying the patterns in simulations of porous convection.

As mentioned, the Rayleigh number indicates whether diffusion or advection is the primary process for transferring heat through the rocks. For small Rayleigh numbers the heat transfer is mainly driven by advection. A large number of simulations with different parameter values must be run and analysed before conclusions can be drawn. A detailed study of the patterns formed in the simulations is beyond the scope of this study.

Simulations of porous convection have traditionally required heavy calculations that were to run on clusters of CPU's to be solved in a reasonable amount of time. A considerable amount of server time had to be devoted to the simulations. This is expensive, both in terms of time and power consumption.

Relatively efficient implementations of porous convection can be developed in low level languages such as C or FORTRAN. Compared to MATLAB this does require a longer development time. The implementation in this thesis was made using Jacket, which allows for relatively fast development since MATLAB syntax can be used. Jacket utilizes the GPU to accelerate the computations. Solving for one time step is done in just under 8 seconds for the largest system that was studied, 321x321x81. Between 1000 and 2000 time steps should be calculated to get well developed features that are unaffected by the initial conditions and allow for pattern analysis. This means that a 2-3 simulations with this resolution can be made in one day on a desktop computer.

The initial conditions for the simulations are a linear temperature profile from the top to the bottom wall, with small random perturbations. In this section the results from three simulations are presented. The simulations are run with different Rayleigh numbers. Details for each run are presented in Table 8.2.

Results from the simulations are presented in Figure 8.3 to Figure 8.5 for Rayleigh number equal to 250, 500 and 1000 respectively. Three visualizations of the system are shown for each of the simulations. An iso-surface for the temperature is presented in figure A. On top of the surface the magnitude of the velocities are visualized in colours. The magnitude of the temperature and velocity are visualized in figures B and C.

	Simulation 1	**Simulation 2**	**Simulation 3**
Resolution	321x321x81	257x257x65	321x321x81
Rayleigh number	250	500	1000
Time step	1E-5	5E-5	1E-5

Table 8.2: Details on the simulations presented in this section.

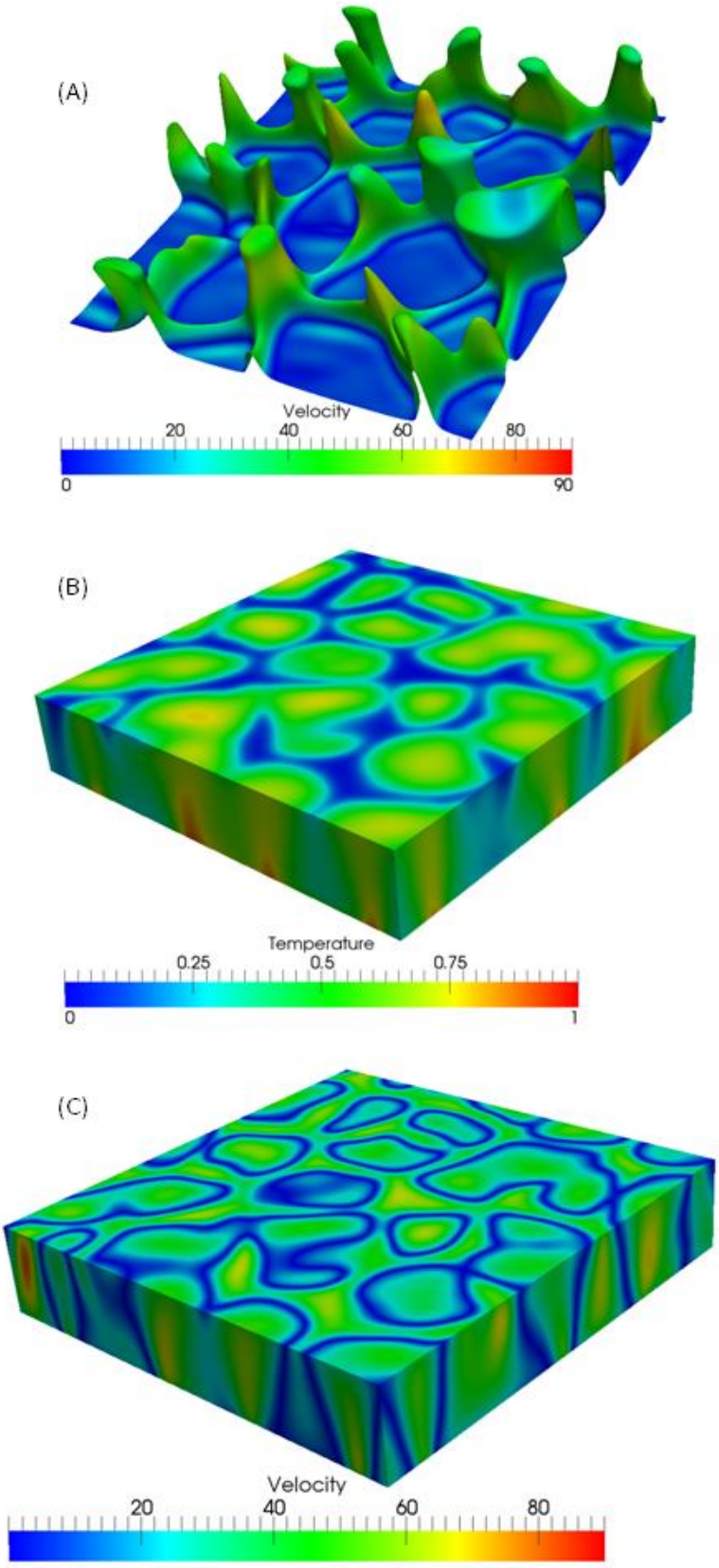

Figure 8.3: Simulation 1, Rayleigh number is 250. A) An iso-surface of the temperature, at T = 0.75, is visualized and the magnitude of the velocity on this surface is shown in colours. B) The magnitude of the temperature is visualized in colours. The top layers of the domain are removed. C) The magnitude of the velocity is shown in colours. The top layer of the domain is removed.

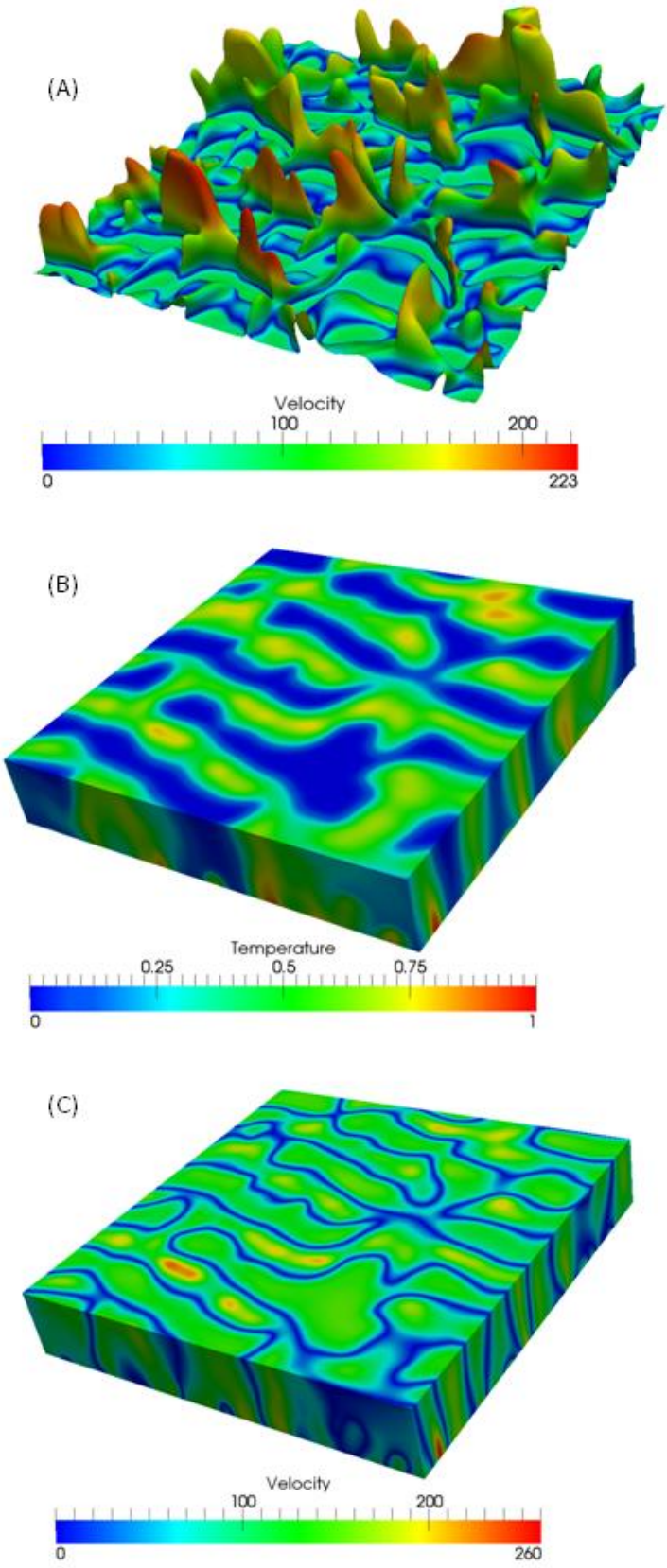

Figure 8.4: Simulation 2, Rayleigh number is 500. A) An iso-surface of the temperature, at T = 0.75, is visualized and the magnitude of the velocity on this surface is shown in colours. B) The magnitude of the temperature is visualized in colours. The top layers of the domain are removed. C) The magnitude of the velocity is shown in colours. The top layer of the domain is removed.

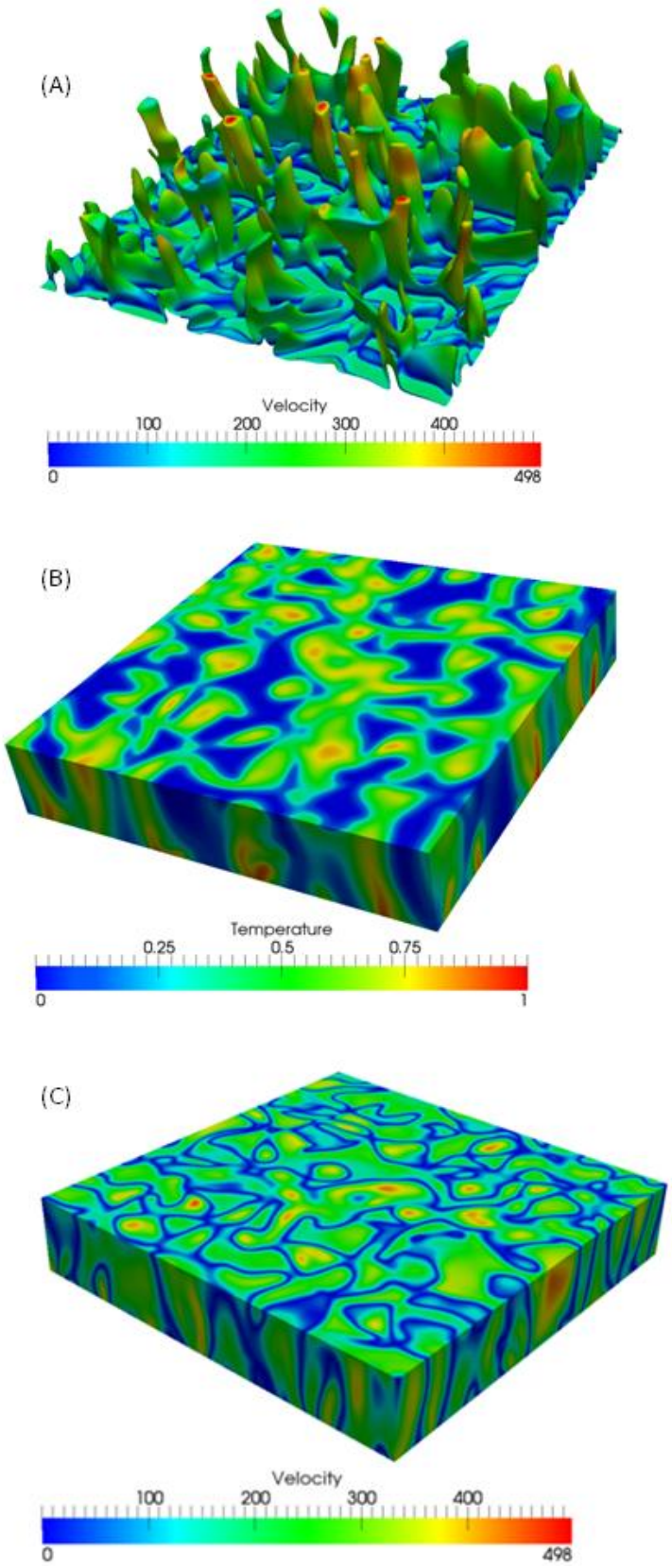

Figure 8.5: Simulation 3, Rayleigh number is 1000. A) An iso-surface of the temperature, at T = 0.75, is visualized and the magnitude of the velocity on this surface is shown in colours. B) The magnitude of the temperature is visualized in colours. The top layers of the domain are removed. C) The magnitude of the velocity is shown in colours. The top layer of the domain is removed.

9 Conclusion

A series of both direct and iterative methods for solving linear sets of equations have been presented and their strengths and weaknesses have been discussed. The choice fell on the multigrid algorithm for the main implementation in this thesis. The multigrid algorithm is written in Jacket, which utilizes the GPU.

Some preliminary tests using Jacket were carried out to find out if substantial speedups for numerical partial differential equation solvers on regular grids can be achieved. In these tests a straight forward implementation, i.e. without the use of multigrid, of transient diffusion was tested. The results showed that the heat diffusion equation for homogeneous systems could be solved efficiently by using convolution for the stencil application. The usually used indexing performs badly in both MATLAB and Jacket. This represents a problem for heterogeneous systems, where convolution cannot be employed.

The bad performance of indexing in MATLAB and Jacket made it necessary to avoid this as much as possible. We found that it was possible to implement all the major components of the multigrid algorithm using convolution. The efficiency of the multigrid algorithm is also strongly dependent on it being tailored to the problem at hand. In this thesis several implementations are developed for solving elliptic and parabolic equations with different boundary conditions. Each of the algorithm parameters must be tweaked as well. The convergence for different input parameters are tested and presented in section 5.4. Based on these results the speed tests for the Poisson solver and porous convection application were implemented.

The Poisson solver developed in this thesis is efficient and can solve for a system of size 257^3 in less than one second on a desktop computer. The Jacket implementation was 60 – 70 times more efficient than the implementation of the algorithm in native MATLAB. These speedups were achieved by harnessing the computational power of the GPU.

The Poisson solver can be used as a part of the porous convection solver which is developed in this thesis. Simulations of this phenomenon have traditionally required heavy computations that have been run on clusters of CPU's. An understanding of the patterns that are formed in the simulations can give insight to the physical processes that forms them. This can in turn be used to understand the history of sedimentary basins and to find parameters such as permeability of the rocks and the viscosity of the fluids that migrate through them.

A large number of simulations are required to study the physical parameters in porous convection. For traditional implementations much server time must be used for the simulations. This is expensive from a time and power consumption point of view. The implementation presented here uses less than 8 seconds to solve one time step when looking at a system of size 321x321x81. This means that roughly 3 simulations can be done in a day on a desktop computer and therefore large parameters studies become feasible.

Overall the developed Multigrid-MATLAB-Jacket-GPU approach reduces project time and cost drastically. Not only does it take less resources for code development, but also the run times and hardware requirements are substantially reduced.

10 Bibliography

BRAESS, D. 1986. On the combination of the multigrid method and conjugate gradients *Multigrid Methods II*.

BRIGGS, W. L., HENSON, V. E. & MCCORMICK, S. F. 2000. *A multigrid tutorial,* Philadelphia, Siam.

BRODTKORB, A. R. 2008. The Graphics Processor as a Mathematical Coprocessor in MATLAB. *International Conference on Complex, Intelligent and Software Intensive Systems, 2008*.

CIPRA, B. 2000. The Best of the 20th Century: Editors Name Top 10 Algorithms. *SIAM News.* SIAM News.

DONALD L. TURCOTTE, G. S. 2002. *Geodynamics,* New York, USA, Cambridge University Press.

HALE, D. 2008. Compact finite-difference approximations for anisotropic image smoothing and painting.

HUBERT NGUYEN, C. Z., EVAN HART, IGNACIO CASTAÑO, KEVIN BJORKE, KEVIN MYERS, AND NOLAN GOODNIGHT 2008. *GPU Gems 3,* Boston, Pearson Education, Inc.

JACK DONGARRA, F. S. 2000. Guest Editors' Introduction: The Top 10 Algorithms. *Computing in Science and Engineering.*

JOSEF STOER, R. B. 2002. *Introduction to numerical analysis,* New York, Springer.

LAY, D. C. 2006. *Linear Algebra and its applications*, Addison-Wesley.

PATRA, M. & KARTTUNEN, M. 2005. Stencils with Isotropic Discretization Error for Differential Operators. *Wiley InterScience*.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T. & FLANNERY, B. P. 2007. *Numerical Recipes Third Edition,* New York, Cambridge University Press.

SAAD, J. 2003. *Iterative Methods for Sparse Linear Systems,* Philadelphia, Society for Industrial and Applied Mathematics.

SHEWCHUK, J. R. 1994. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain

TATEBE, O. 1993. The Multigrid Preconditioned Conjugate Gradient Method.

TROTTENBERG, U., OOSTERLEE, C. & SCHÜLLER, A. 2001. *Multigrid,* London, Academic Press.

VARGA, R. S. 2000. *Matrix iterative analysis,* Berlin Springer.

WALTER GANDER, G. H. G. 1997. Cyclic Reduction - History and Applications.

YOUNG, D. M. 1971. *Iterative Solution of Large Linear Systems,* New York, Academic Press.

11 Appendix

11.1 Multigrid Solver for Poisson Problems

This code solves Poisson problems with Dirichlet boundary conditions. With small changes it can be adapted to solving system with other boundary conditions and transient heat diffusion problems. CUDA and Jacket must be installed on the computer run the codes. The implementation is divided into two functions. The first one, `diffusion_dirichlet`, only contains the algorithm parameters presented in Table 5.1. The first input parameter for this function is the source term, and the second one is the initial guess for the solution. The second function,`mgcyc_dir` , is the multigrid cycle, the implementation of it is described in section 5.3. An additional script is needed to define the problem that should be solved. An example of such a script is presented in the following section, for the porous convection application.

```
function [ u ] = diffusion_dirichlet( b,u )
    % Algorithm parameters
    v1       = 2;               % Number of pre smoothing steps.
    v2       = 2;               % Number of post smoothing steps.
    cl       = 5;               % Coarsest level
    n_it     = 5;

    % Stencil
    L        = zeros(3,3,3);
    L(:,:,1) = [ 0 -1  0; -1 -2 -1;  0 -1  0];
    L(:,:,2) = [-1 -2 -1; -2 24 -2; -1 -2 -1];
    L(:,:,3) = [ 0 -1  0; -1 -2 -1;  0 -1  0];
    L        = (1/6)*L;

    % Iteration loop
    for i = 1:n_it
        [u]           = mgcyc_dir( u, b, L, v1, v2, cl);
    end
end

function [ u ] = mgcyc_dir(u, f, L, v1, v2, cl )
    [m n o] = size(u);

    % Boundary values
    Ux          = u([1 m],:,:);
    Uy          = u(:,[1 n],:);
    Uz          = u(:,:,[1 o]);

    %% Pre-smoothing
    L_sm        = L;
    L_sm(2,2,2) = 0;

    for j = 1:v1
        u           = 1/L(2,2,2)*(f - convn(u,L_sm,'same'));

        % Dirichlet boundary conditions
        u([1 m],:,:) = Ux;
        u(:,[1 n],:) = Uy;
        u(:,:,[1 o]) = Uz;
    end
```

```
    %% Compute the residual
    df          = f - convn(u,L,'same');

    %% Restriction
    % Define restriction operator
    KER         = zeros(3,3,3);
    KER(:,:,1) = 1/28*[0 1 0;1 2 1;0 1 0];
    KER(:,:,2) = 1/28*[1 2 1;2 4 2;1 2 1];
    KER(:,:,3) = 1/28*[0 1 0;1 2 1;0 1 0];

    df          = convn(df,KER,'same');
    df          = 4*df(1:2:end,1:2:end,1:2:end);

    vc = gzeros(size(df));
    if (ceil(numel(u).^(1/3)) > cl)  % Continue coaresning
        [vc] = mgcyc_dir(vc, df, L, v1, v2, cl );
    else                                % Solve the restriction equation.
        n_it        = 3;
        [mc nc oc] = size(vc);
        for i = 1:n_it
            vc              = 1/L(2,2,2)*(df - convn(vc,L_sm,'same'));
            % Dirichlet boundary conditions
            vc([1 mc],:,:) = 0;
            vc(:,[1 nc],:) = 0;
            vc(:,:,[1 oc]) = 0;
        end
    end

    clear df
    %% Interpolation
    vf                                  = gzeros(size(u));
    vf(1:2:end  ,1:2:end  ,1:2:end  )   = vc;
    clear vc

    % Define interpolation operator
    KER(:,:,1) = 1/8*[1 2 1;2 4 2;1 2 1];
    KER(:,:,2) = 1/8*[2 4 2;4 8 4;2 4 2];
    KER(:,:,3) = 1/8*[1 2 1;2 4 2;1 2 1];

    vf = convn(vf,KER,'same');

    %% Compute the corrected approximation
    u = u + vf;

    %% Post-smoothing
    for j = 1:v2
        u = 1/L(2,2,2)*(f - convn(u,L_sm,'same'));

        % Dirichlet boundary conditions
        u([1 m],:,:) = Ux;
        u(:,[1 n],:) = Uy;
        u(:,:,[1 o]) = Uz;
    end
end
```

11.2 Porous convection

This is the script for the porous convection algorithm. It utilizes a modified version of the functions presented in the previous section, where a combination of periodic and Neumann boundary conditions are used. Details on how the code is implemented are presented in section 8.2.1.

```
%% Numerics
n      = 8;              % Grid size
nx     = 2^n + 2;        % Nr of grid points
n      = 8;              % Grid size
ny     = 2^n + 2;        % Nr of grid points
n      = 6;              % Grid size
nz     = 2^n + 1;        % Nr of grid points
h      = 1/(nz-1);             % Spatial step length
nt     = 2000;

x      = linspace( 0,(nx-1)/(nz-1)  ,nx  );
y      = linspace( 0,(ny-1)/(nz-1)  ,ny  );
z      = linspace( 0,1  ,nz  );

LX     = max(X(:)) - min(X(:)) - h;
LY     = max(Y(:)) - min(Y(:)) - h;

%% Physics
dt     = 5e-5;
Ra     = 5e2;

%% Initialize
PRE                           = zeros(nx,ny,nz);
RHS_PRE                       = zeros(size(PRE));
V                             = zeros(size(PRE));
TEMP                          = (1-Z);
TEMP(2:end-1,2:end-1,2:end-1) = TEMP(2:end-1,2:end-1,2:end-1) + .1*rand(nx-
2,ny-2,nz-2)-.05;

% Periodic boundary conditions
TEMP(1,:,:)                   = TEMP(end-1,:,:);
TEMP(:,1,:)                   = TEMP(:,end-1,:);

%% Stencils velocity
[GCOORD, ELEM2NODE] = generate_mesh_8_brick(x,y,z);
[Y X Z]             = meshgrid(y,x,z);
[Kzs]               = thermal3d_brick_mat(ELEM2NODE, GCOORD);
clear ELEM2NODE GCOORD

KERvx         = zeros(3,3,3);
KERvx(1,2,2)  = -1/(2*h);
KERvx(3,2,2)  =  1/(2*h);

KERvy         = zeros(3,3,3);
KERvy(2,1,2)  = -1/(2*h);
KERvy(2,3,2)  =  1/(2*h);

KERvz         = zeros(3,3,3);
KERvz(2,2,1)  = -1/(2*h);
KERvz(2,2,3)  =  1/(2*h);
```

```
L             = 1+(dt/(6*h^2))*24;

L_temp        = zeros(3,3,3);
L_temp(:,:,1) = [ 0 -1  0; -1 -2 -1;  0 -1  0];
L_temp(:,:,2) = [-1 -2 -1; -2 24 -2; -1 -2 -1];
L_temp(:,:,3) = [ 0 -1  0; -1 -2 -1;  0 -1  0];
L_temp        = (1/(6*h^2))*L_temp;
L_temp(2,2,2) = 0;

[KzsaT KzsaT_bot KzsaT_top] = Stencil_brick_3d(Kzs');

%% Boundary conditions
% Temperature (dirichlet)
TEMPz        = TEMP(:,:,[1 nz]);

%% Allocate memory on the GPU
TEMP = gsingle(TEMP);
PRE  = gsingle(PRE );
gforce(TEMP);
gforce(PRE );
%% Time loop
for i = 1:nt
    i
    %% Right hand side pressure
    TEMP(end  ,:,:) = TEMP(2,:,:);
    TEMP(:,end  ,:) = TEMP(:,2,:);

    DTDz            = convn(TEMP,KzsaT,'same');

    % Periodic boundary conditions
    DTDz(1  ,:,:)    = DTDz(end-1,:,:);
    DTDz(:,1  ,:)    = DTDz(:,end-1,:);
    DTDz(end,:,:)    = DTDz(2    ,:,:);
    DTDz(:,end,:)    = DTDz(:,2    ,:);

    tmp              = conv2(TEMP(:,:,1    ), KzsaT_top(:,:,2),'same') +
                       conv2(TEMP(:,:,2  ), KzsaT_top(:,:,3),'same');
    DTDz(:,:,1  )    = tmp;

    tmp              = conv2(TEMP(:,:,end-1), KzsaT_bot(:,:,1),'same') +
                       conv2(TEMP(:,:,end), KzsaT_bot(:,:,2),'same');
    DTDz(:,:,end)    = tmp;

    DTDz(1  ,:,1  )  = DTDz(end-1,:,1  );
    DTDz(:,1  ,1  )  = DTDz(:,end-1,1  );
    DTDz(1  ,:,end)  = DTDz(end-1,:,end);
    DTDz(:,1  ,end)  = DTDz(:,end-1,end);

    %% Pressure solver
    PRE              = pressure_solver(-Ra/h*DTDz,0*PRE );

    %% Velocity field
    PRE(:,end,:)     = PRE(:,2  ,:);
    PRE(end,:,:)     = PRE(2  ,:,:);

    % x-direction
    % ---------------------------------------------------------------------
    V                = -convn(PRE,KERvx,'same');
```

```
% Periodic boundary conditions
V(1  ,:,:)        = V(end-1,:,:);
V(:,1  ,:)        = V(:,end-1,:)
xadv              = X-V*dt;

% y-direction
% ------------------------------------------------------------------------
V                 = -convn(PRE,KERvy,'same');

% Periodic boundary conditions
V(1  ,:,:)        = V(end-1,:,:);
V(:,1  ,:)        = V(:,end-1,:);
yadv              = Y-V*dt;

% z-direction
% ------------------------------------------------------------------------
V                 = -(convn(PRE,KERvz,'same') - Ra*TEMP);

% Dirichlet boundary conditions
V(:,:,1  )        = 0;
V(:,:,end)        = 0;
% Periodic boundary conditions
V(1  ,:,:)        = V(end-1,:,:);
V(:,1  ,:)        = V(:,end-1,:);

zadv              = Z-V*dt;

TEMP_OLD        = TEMP;

%% Heat diffusion
for j = 1:50
    TEMP(:,end,:)      = TEMP(:,2,:);
    TEMP(end,:,:)      = TEMP(2,:,:);

    TEMP               = 1/L*(TEMP_OLD - dt*convn(TEMP,L_temp,'same'));

    % Dirichlet boundary conditions
    TEMP(:,:,[1 nz])   = TEMPz;

    % Periodic boundary conditions
    TEMP(1,:,:)        = TEMP(end-1,:,:);
    TEMP(:,1,:)        = TEMP(:,end-1,:);
    TEMP(:,end,:)      = TEMP(:,2,:);
    TEMP(end,:,:)      = TEMP(2,:,:);
end

%% transfere data to the CPU
TEMP =  single(TEMP);

xadv = single(xadv(1:end-1,1:end-1,:));
yadv = single(yadv(1:end-1,1:end-1,:));
zadv = single(zadv(1:end-1,1:end-1,:));

%% Advection
% Periodic boundary conditions
xadv(xadv<min(X(:))  ) =  xadv(xadv<min(X(:))  ) + LX;
```

```
    xadv(xadv>max(X(:))-h) =  xadv(xadv>max(X(:))-h) - LX;
    yadv(yadv<min(Y(:))   ) =  yadv(yadv<min(Y(:))   ) + LY;
    yadv(yadv>max(Y(:))-h) =  yadv(yadv>max(Y(:))-h) - LY;

    TEMP(1:end-1,1:end-1,:) = interp3(Y,X,Z,TEMP,yadv,xadv,zadv,'linear');

    %% Save temperature
    if (mod(i,30) == 0)
        filename = ['temperature' num2str(i) '_257.mat'];
        %tmp_temp = TEMP(1:end-1,1:end-1,:);
        save(filename,'TEMP', 'V');
    end

    TEMP = gsingle(TEMP);
end
```