A PARALLEL ELLIPTIC PDE SOLVER

Jesper Grooss

Kgs. LYNGBY 2001
EKSAMENSPROJEKT
NR. 08/2001

IMM

Preface

This thesis constitutes my Masters thesis project for a degree in Master of Science in Engineering (M.Sc.Eng.). It was written during the period of 1st of February to 31st of July 2001 at Informatics and Mathematical Modelling (IMM), Technical University of Denmark (DTU).

Supervisor on the project have been Associate Research Professor Stefan Mayer, located at IMM and member of the Computational Hydrodynamics Group, a group based on a Research Frame Program on Computational Hydrodynamics financed by the Danish Technical Research Council (STVF). This thesis arise primarily due to Stefans needs and knowledge in the field of computational hydrodynamics.

The reader is assumed to have knowledge of numerical methods and mathematics at the level of advanced undergraduates or graduates.

I thank Stefan for his big enthusiasm and interest in the project, and his willingness to always take his time to answer questions and discuss problems. I also thank Jan M. Rasmussen and Michael Jacobsen for good advice and reading through the thesis, thereby making it more readable.

Kgs. Lyngby, July 27th, 2001
Jesper Grooss

Abstract

The problem considered is how to parallelize an elliptic PDE solver, or to be specific: How to parallelize a Poisson solver based on a finite volume discretization. The Poisson problem arises as a subproblem in computational fluid dynamics (CFD). The motivation is a wish to parallelize an existing CFD solver called NS3.

Different methods from domain decomposition are presented, and their properties are outlined. First is presented the original method of Schwarz, the classical alternating Schwarz method, which is based on overlapping domains. Secondly is presented a non overlapping approach, where Dirichlet data and Neumann data are exchanged over the boundary in odd and even iterations respectively. Finally Schur Complement methods and the BDD preconditioner are presented. In the literature the latter shows for a finite element approach nice properties from a parallelization point of view.

These methods of domain decomposition are adapted to fit into restrictions given by NS3. Numerical experiments show that the BDD preconditioner still has the same properties using a finite volume approach, hence it is applicable to the Poisson problem at hand.

Resume

Problemet, som betragtes, er hvordan man paralleliserer en elliptisk PDE løser, eller for at være mere specifik: Hvordan man paralleliserer en Poisson løser baseret på en finite volume diskretisering. Poisson problemet opstår som et delproblem i numerisk strømningsdynamik (CFD). Motivationen kommer fra et ønske om at parallelisere en eksisterende CFD løser ved navn NS3.

Forskellige metoder fra domæne dekomposition bliver præsenteret sammen med hver deres egenskaber. Først præsenteres den originale metode af Schwarz, classical alternating Schwarz metoden, som er baseret på overlappende domæner. Dernæst præsenteres en ikke overlappende metode, hvor Dirichlet data og Neumann data udveksles over domæne grænsen i hhv. ulige og lige iterationer. Til slut præsenteres Schur komplement metoder og BDD prækonditioneren. I litteraturen vises at BDD prækonditioneren ved en finite element diskretisering har gode egenskaber, set fra et paralleliserings synspunkt.

Disse metoder fra domæne dekomposition bliver tilpasset til begrænsningerne givet af NS3. Numeriske eksperimenter viser, at BDD prækonditioneren har de samme egenskaber ved brug af en finite volume tilgang, og den er derfor anvendelig til at løse det originale Poisson problem.

Contents

CHAPTER 1

Introduction

How to parallelize an elliptic PDE[1] solver, or to be specific; how to parallelize a Poisson solver, is what this thesis is about. The Poisson problem is described by the PDE

$$\nabla^2 u = f, \tag{1.1}$$

or in cartesian (x, y, z) coordinates

$$\frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u + \frac{\partial^2}{\partial z^2}u = f. \tag{1.2}$$

It is an elliptic PDE, hence the title of the thesis.

But the first question to be answered is: Why at all solve a Poisson problem?

A Poisson problem arise as a subproblem in computational fluid dynamics (CFD), how is explained in Chapter 2. It is furthermore a difficult part of a CFD solver to parallelize, due to its elliptic nature. Therefor the first step towards a parallel CFD solver is to parallelize the Poisson solver.

[1] Partial Differential Equation

1.1 Motivation

In many areas of engineering fluid models are basis for design of structures with certain functionality and / or durability.

A real fluid flow problem often include fluctuations in time and space on scales ranging over many orders of magnitude, from long waves to tiny turbulent eddies. To include all scales, fluid models must have a very high resolution. In practice however, even if the highest possible resolution is applied, it is often not possible to include the smallest scales. This has in many cases limited the use and liability of computational fluid dynamics (CFD).

To get the best possible results, the highest possible resolution have always been applied, hence CFD have always utilized the power of computers to the limit. And with the power of computers increasing almost day by day, computational fluid dynamics has become a growing discipline in engineering science.

At IMM, DTU exists a CFD program to solve fluid models. The CFD program is special in its capability to handle moving geometries, especially a free surface. It exists only in a serial version, and it have been the wish for some time to make a parallel version.

And that is the main motivation for this project.

The main goal of this project is therefor to :

- Explore different methods for solving an elliptic PDE, the Poisson problem, in parallel.
- Adapt, if possible, the methods to work within the existing CFD solver.
- Verify that the methods parallelize well and hence will be an improvement of the existing serial program.

Even though the motivation is from CFD, usability of this thesis is not necessarily limited to this area. The Poisson equation arise in many other interesting fields, and there it will usually be equally difficult to solve. In that case a parallel approach, like the one presented here, might be of interest. Furthermore, the process presented here may be

applicable to other elliptic PDE's than the Poisson problem.

1.2 Examples of Application

A Fluid is a substance which deforms when exerted to even the smallest force. It describes a continuum, and it covers gasses and liquids. To link this to practical aspect, consider the following applications of fluid modelling.

The Bumblebee Paradox. "According to aerodynamic theory, a bumblebee cannot fly." This was the answer a Swiss aerodynamicist gave to a biologist in the 1930th after a back-of-the-napkin calculation during a dinner party, the rumor says. The saying is usually continued by: "Nobody just ever told it so."

Time have shown however that the application of aerodynamic principles in the 1930th, at least in the bumblebee case, were based on assumptions, which do not hold when modelling the flight of a bumblebee.

But still today the bumblebee challenges scientists. Modelling its flight is not simple: It includes a moving geometry (the wing) and the model must be of very high resolution in order to include the small vortices around the wing, which is essential for the bumblebee to gain the necessary lift. Recently [Wang00] have produced results of a 2D model using "hundreds of hours of number-crunching by a supercomputer" [Sege00], which show sufficient lift for the bumblebee to fly. They should at present be working on a 3D model [Sege00].

Wind Turbine Wing Design. According to an article in the Danish engineering magazine, Ingeniøren [Gods01], DTU and Risø have together developed a computer program to test the design of wind turbine wings. Result from the program have been held against measurements in a big wind tunnel of NASAs in California, and the program have produced very fine results.

When this type of computations become more practically usable, it is expected to cut down expenses for development of new and more effective wing profiles.

The program is based on 3.000.000 cells, each contributing with 6 equations and 6 unknowns, which have to be solved for each timestep.

Free surface Waves. An example is the design of ships: How should a ship hull be created such that it has certain properties, i.e. small water resistance, the ability to sail fast even in high waves, or strength to withstand the load from waves.

1.3 Outline

The order of the chapters follows to a great extend the path of recognitions / realizations that have led me through the project.

Chapter 2 sets the plot: The equations of fluid motion, the Navier-Stokes equations, are presented. It is shown how the equations can be handled and why a Poisson equation becomes important. How to apply a finite volume approximation is described, and the chapter ends by describing the structure of the discrete Poisson operator.

Chapter 3 presents the victim: NS3, the name of the CFD solver at hand. How NS3 decomposes the domain into blocks, creates a grid, solves, and performs, is described. In the end of the chapter an outline of the restrictions that this project has to work within is addressed.

Chapter 4, 5, and 6 describe the accused; different methods of domain decomposition. How do they work, and especially how well do they work? And do they fit into our restrictions? Three methods will be considered, Chapter 4 describe the classical alternating Schwarz method, which in some sense is the original, and based on overlapping domains. Chapter 5 presents a non-overlapping method. Chapter 6 turns to Schur complement methods, and especially the balancing domain decomposition (BDD) method, which in a finite element context shows almost optimal properties from a parallelization point of view.

Chapter 7 consist of the testimonies: The different methods is confronted with the witnesses, a suite of examples, and properties of the methods are verified experimentally. Especially is it verified that the almost optimal properties of the BDD method also apply in a finite volume context.

Chapter 8 finally pronounces a sentence upon the accused: It summarizes the results for the different methods, and argue that the BDD method is applicable and hence is guilty. Also an outline of further work is given, first of all of work to finish before a parallel implementation is started, but also some potentially interesting loose ends is mentioned.

The Appendix consists of a number of sections, all referenced from somewhere in the thesis. I will however mention a few here: Appendix A presents the witnesses, a suite of examples that have been used to test the different methods. When the text refers to Example A.1, it will actually be to Figure A.1 in Appendix A. Appendix B gives a short survey of theory for preconditioners and Krylov subspace methods. And finally Appendix E: I have a number of times been asked what the “elliptic” in elliptic PDE means, and this appendix is devoted to that.

CHAPTER 2

The Plot

The purpose of this chapter is to introduce the equation of fluid motion which is the underlying basis of this work. The chapter shows that solving a Poisson problem is a vital part when solving fluid models. The creation and structure of the discrete Poisson operator is presented, including how to implement the most common boundary conditions. Guidelines for how to solve the fluid dynamic equations in total are described, without giving specific algorithms.

Notation and concepts from this chapter will be used throughout the thesis.

2.1 Navier-Stokes Equations

The basic equations describing fluid flows are the Navier-Stokes equations. The equations will only be stated here, for a derivation turn to e.g. [Ande95], which has a thorough tutorial of how to derive different versions of the equations.

Only flows of incompressible fluids will be considered. Liquids can often be assumed incompressible, and so can a gas with veloc-

ities somewhat below the speed of sound, when $M < 0.3$.[1] When the compressibility is neglected, the fluid density is usually assumed constant. Also we assume the fluid to be isothermal, which implies constant viscosity. In that case we end up with what is usually called the Navier-Stokes equations for incompressible flow:

$$\nabla \cdot \mathbf{v} = 0, \tag{2.1a}$$

$$\frac{\partial u_i}{\partial t} + \nabla \cdot (u_i \mathbf{v}) = \nabla \cdot (\mu \nabla u_i) - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + g_i, \quad i = 1, \ldots, d \tag{2.1b}$$

where $\mathbf{v}$ is the fluid velocity vector, u_i is the ith cartesian component of $\mathbf{v}$ in the x_i direction, p is the pressure, μ viscosity, ρ density, g gravity, and i ranges from 1 to d the dimension of the domain, usually 2 or 3. The unknowns are the pressure p and the velocity components in the vector $\mathbf{v}$.

The equations are here presented on differential form in their cartesian coordinates and in conservation form.[2] Equation (2.1a) is usually referred to as the Mass equations, while the latter Equation (2.1b) is referred to as the Momentum equation, since they describe conservation of Mass and Momentum respectively.

Note that the equations are coupled and nonlinear and in general impossible to solve analytically.

Integrating Equations (2.1a) and (2.1b) over a volume V having the boundary $S = \delta V$, and applying the divergence theorem (D.1), transform the equations into integral form,

$$\int_S \mathbf{v} \cdot \mathbf{n} \, \mathrm{d}S = 0, \tag{2.2a}$$

$$\frac{\partial}{\partial t}\int_V u_i \, \mathrm{d}V + \int_S (u_i \mathbf{v}) \cdot \mathbf{n} \, \mathrm{d}S = \int_S (\mu \nabla u_i) \cdot \mathbf{n} - \frac{p}{\rho} n_i \, \mathrm{d}S + \int_V g_i \, \mathrm{d}V. \tag{2.2b}$$

The integral form is the form used when applying a finite volume discretization.

[1] M is the Mach number. It is a dimensionless number describing the speed of the fluid relative to the speed of sound in the fluid, $M = 1$ is the speed of sound.

[2] Conservation form is when the PDE can be written as $\partial u_i / \partial t + \nabla \cdot \mathbf{a}(\mathbf{v}) = 0$.

2.2 Finite Volume Discretization

In a finite volume (FV) discretization the domain is subdivided into a number of small, disjoint, finite sized control volumes, abbreviated CVs. We will consider a cell centered FV, where in the center of each CV a node is defined at which the value of the unknown variables are to be found. These node values represent the mean of the variables over the CV.

Figure 2.1 shows a part of a grid that arise from a uniform, rectangular subdivision of a 2D domain. The boundary of one CV consists of several faces: A face is the boundary between two neighbouring CVs. In a rectangular 2D grid the faces are named after the four corners of the world, $k = e, w, n, s$, while 3D also have $k = f, b$ (front and back). In general the CVs can have any shape, though only quadrilateral CVs will be treated here.

On each CV the integral form of the equations for Mass (2.2a) and Momentum (2.2b) are imposed. The equations contain integrals which cannot be calculated exactly since only values at the nodes are known. Therefor these surface and volume integrals must be approximated.

In the following, approximations for the CV labeled I in Figure 2.1 is given, where φ_I denotes the value of φ at the center of cell I.

2.2.1 Approximation of Integrals and Derivatives

Assume a quantity φ is known at the center of each CV. This will show how to achieve second order accurate approximations of the integrals when having an orthogonal, equidistant grid.

Surface Integrals. The integral of a quantity φ over a CV surface can be split into a sum of integrals over each face of the CV

$$\int_S \varphi \, \mathrm{d}S = \sum_k \int_{S_k} \varphi \, \mathrm{d}S. \tag{2.3}$$

Only the $k = w$ (west) face will be considered here, other faces are treated analogously. To calculate the integral of φ on the face requires

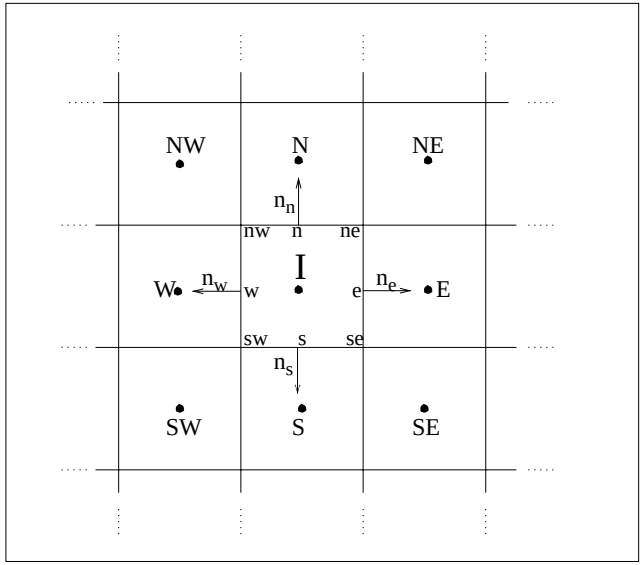


Figure 2.1: *Finite Volume mesh*

φ to be known on the entire surface. That is not available. To second order the integral can be approximated by the midpoint rule,

$$\int_{S_w} \varphi \, \mathrm{d}S \approx \overline{\varphi}_w S_w, \tag{2.4}$$

$\overline{\varphi}_w$ being the midpoint value of φ on the face w and S_w the area. However, $\overline{\varphi}_w$ is still not known, and to maintain second order accuracy, this must be approximated also to second order. Using linear interpolation, an average $\overline{\varphi}_w \approx \frac{1}{2}(\varphi_I + \varphi_W)$ will accomplish this on the rectangular equidistant grid, giving

$$\int_{S_w} \varphi \, \mathrm{d}S \approx \frac{1}{2}(\varphi_I + \varphi_W) S_w. \tag{2.5}$$

The quantity φ should be replaced by the flux of some quantity through the surface, e.g. mass $\varphi = \rho \mathbf{v} \cdot \mathbf{n}$. On a nonorthogonal grid, the normal vector $\mathbf{n}_w$ needs to be taken into consideration, and also a simple average for $\overline{\varphi}_w$ will no longer give second order accuracy.

Volume Integrals. The integral of a quantity q over a CV volume can also be approximated to second order by the midpoint rule. Since q_I is defined to be the center value of q in the CV, the approximation becomes

$$\int_V q \, \mathrm{d}V \approx \overline{q} V_I \approx q_I V_I, \tag{2.6}$$

V_I being the volume of the CV. This is also a second order approximation.

Derivatives. The integrals contain gradients and space derivatives which are not known exact and also need to be approximated. To keep second order accuracy of the integrals, the derivatives need to be approximated to at least second order. For that, a central approximation scheme can be used, so e.g. the gradient of φ at the midpoint of a face

$$\left(\frac{\partial \varphi}{\partial x_1}\right)_w \approx \frac{\varphi_W - \varphi_I}{x_W - x_I}. \tag{2.7}$$

Analogously for the x_2 and x_3 directions.

Notes on Approximations. In the case of a nonorthogonal or non equidistant grid, more points are needed to calculate the midpoint value of φ to achieve second order accuracy. Otherwise the approximations are in the worst case first order only.

If higher order approximations for these integrals are sought, more points are needed in the integration domain, and higher order approximations of the integrals must be used. Some can be found in [Ferzi97].

2.2.2 Setting Up the System

The procedure is to approximate all integrals of the Mass equation (2.2a) and Momentum equation (2.2b) by a midpoint rule, and furthermore approximate all needed values and derivatives of the unknowns at this midpoint to second order.

Then it is possible to write the Mass and Momentum equation as a function of unknowns at cell centers only. The result is one equation for each cell and unknown variable, some containing a time derivative. Each equation for one CV will have contributions from neighbouring CV cell centers.

If the problem is to solve a steady state problem, all time derivatives can be removed from the equations. In that case only an algebraic system of equations needs to be solved. The system nonlinear and hence not straight forward to solve. One could use techniques with temporary linearization to transform the problem into a system of linear algebraic equations which then is solved iteratively until convergence.

Many methods to solve the nonlinear steady state problem exist, and often a subproblem is solving a Poisson problem or Poisson-like problem, as e.g. the SIMPLE[3] method [Flet01].

Note that applying a conservation equation on each CV is the same as applying the equation on the entire domain. This is seen by summing up all contributions of surface and volume integrals from each CV. The volume integrals sum up to the volume integral of the entire domain while surface integrals for inner CV faces cancel out, leaving exactly surface integrals over the domain boundary.

2.3 Time Stepping

In the unsteady case there is also a time dimension which needs to be discretized. A simple time discretization can be written as $t_{i+1} = t_i + \Delta t_i$. Usually a solver follows the natural perception of time in that

[3] Semi Implicit Method for Pressure Linked Equations

sense that is solves completely for one time t_i before proceeding to the next t_{i+1}.

The difficulty in the incompressible Navier Stokes equations arise due to the lack of an independent equation for the pressure p. While each of the Momentum equations (2.1b) involve a time derivative of one of the velocity components, the Mass equation (2.1a) does not provide something alike for the last unknown, the pressure. In simplified form:

$$0 = G(\mathbf{v}), \tag{2.8a}$$

$$\frac{\partial u_i}{\partial t} = F(\mathbf{v}, p). \tag{2.8b}$$

Instead the pressure must in some way be constructed, so that the velocity field computed from the Momentum equation (2.8b) also satisfy the Mass equation (2.8a). Some way of linking pressure and velocity is needed.

2.3.1 The Pressure-Velocity Linking

Consider the differential form of one Momentum equation (2.1b)

$$\begin{aligned} \frac{\partial u_i}{\partial t} &= -\nabla \cdot (u_i \mathbf{v}) + \nabla \cdot (\mu \nabla u_i) - \frac{1}{\rho}\frac{\partial p}{\partial x_i} + g_i \\ &= F_i - \frac{1}{\rho}\frac{\partial p}{\partial x_i}, \qquad i = 1, \ldots, d, \end{aligned} \tag{2.9}$$

where F_i is just an abbreviation for terms not including the pressure. One way of solving this is to approximate the time derivative with some scheme, let us here consider a simple explicit Euler scheme:

$$u_i^{n+1} - u_i^n = \Delta t \left(F_i^n - \frac{1}{\rho}\frac{\partial p^n}{\partial x_i} \right), \qquad i = 1, \ldots, d, \tag{2.10}$$

where the n superscript indicate that the values from time level n is used. Assume that the old velocity field satisfies the Mass equation (2.1a), then the new velocity field u_i^{n+1} does in general not do that.

Note that satisfying the Mass equation (2.1a) corresponds to making the velocity field divergence free.

Consider each of the d Momentum equations (2.10) as one vector equation, taking the divergence of the vector equation gives

$$\nabla \cdot \mathbf{v}^{n+1} - \nabla \cdot \mathbf{v}^n = \Delta t \nabla \cdot \left(\mathbf{F}^n - \frac{1}{\rho} \nabla p^n \right). \tag{2.11}$$

The first term is the divergence of the new velocity field. The Mass equation (2.1a) demands this to be zero. The second term is alike just for the former time level, which is assumed to satisfy the Mass equation and is therefore already zero. This means that the new velocity field will satisfy the Mass equation when constructing the pressure so that the right hand side is zero,

$$0 = \rho \nabla \cdot \mathbf{F}^n - \nabla^2 p^n. \tag{2.12}$$

This is a Poisson equation for the pressure. Note that this was derived using an explicit Euler scheme, but any other approximation scheme would also have produced a Poisson equation for the pressure. Note also that the pressure has superscript n as if it belongs to timelevel n but its dependency is not important, and will change with the time derivative approximation scheme used.

2.3.2 Explicit Time Stepping

This give rice to the following "algorithm" which uses explicit time stepping. Assume that the velocity field $\mathbf{v}$ satisfies the Mass equation at time level n. To calculate the new velocity field $\mathbf{v}^{n+1}$:

1. Calculate $\mathbf{F}^n$ and solve the Poisson equation (2.12) to obtain p^n.
2. Use p^n in the Momentum equation (2.10) to calculate the new velocity field $\mathbf{v}^{n+1}$ which will also satisfy the Mass equation.

2.3.3 Implicit Time Stepping

Consider instead the Momentum equation (2.9) discretized using implicit time stepping, e.g. a backward Euler.

$$\mathbf{v}^{n+1} - \mathbf{v}^n = \Delta t \left(\mathbf{F}^{n+1} - \frac{1}{\rho} \nabla p^{n+1} \right). \tag{2.13}$$

The problem of making the new velocity field satisfy the Mass equation is similarly done by requiring the divergence of the right hand side to be zero, so again p must satisfy a Poisson equation. The difficulty here is that to calculate $\mathbf{F}^{n+1}$ and thereby p^{n+1}, $\mathbf{v}^{n+1}$ is needed and similar to calculate $\mathbf{v}^{n+1}$, p^{n+1} is needed. So it is necessary to solve for both simultaneously. This can be done in some iterative manner having the pressure updating as an inner loop and the velocity updating as an outer loop.

The next section will describe a way to circumvent an iterative procedure in what is called the pressure correction method.

2.3.4 Pressure Correction Scheme

Consider again Equation (2.13). Write the pressure as $p^{n+1} = p^n + \Delta p$, and add and subtract a temporary solution $\mathbf{v}^*$ to the left hand side,

$$\mathbf{v}^{n+1} - \mathbf{v}^* + \underbrace{(\mathbf{v}^* - \mathbf{v}^n) = \Delta t \left(\mathbf{F}^* - \frac{1}{\rho} \nabla p^n \right)}_{\text{solve for } \mathbf{v}^*} - \Delta t \frac{1}{\rho} \nabla (\Delta p). \tag{2.14}$$

First solve the under-braced part: Use the pressure from the previous time level p^n to calculate $\mathbf{v}^*$. The solution $\mathbf{v}^*$ will not satisfy the Mass equation and be divergence free. Consider the remaining part

$$\mathbf{v}^{n+1} = \mathbf{v}^* - \Delta t \frac{1}{\rho} \nabla (\Delta p). \tag{2.15}$$

Requiring that $\mathbf{v}^{n+1}$ is divergence free again end up in solving a Poisson equation, this time for the pressure correction Δp

$$0 = \nabla \cdot \mathbf{v}^* - \Delta t \frac{1}{\rho} \nabla^2 (\Delta p). \tag{2.16}$$

This leads to the following "algorithm":

1. Solve the nonlinear system for $\mathbf{v}^*$ using old pressure values.
2. Solve the Poisson equation (2.16) for the pressure correction Δp
3. Update the velocity field using Equation (2.15). The new velocity field $\mathbf{v}^{n+1}$ will be divergence free and hence satisfy the Mass equation.

Note that the solution using the pressure correction approach is not exactly the same as solving the whole system simultaneously, hence the calculation of $\mathbf{v}^*$ as noted in in equation (2.14) is based on $\mathbf{F}^*$ and not $\mathbf{F}^{n+1}$ as in equation (2.13).

Implicit versus Explicit

Con's This implicit "algorithm" needs to solve two systems, and will therefore usually be approximately double as expensive to solve as the explicit counterpart per time step.
It is fairly complicated to implement since it requires a solver for a nonlinear system as well, while the explicit one only needs a Poisson solver.

Pro's Due to the implicit approach, stability is maintained for much larger time steps Δt, implying considerably fewer timesteps for a given time interval and hence using less computing time.

2.4 The Discrete Poisson Operator

The previous sections have shown how a Poisson problem arise. This section will show what it looks like and how to implement different boundary conditions.

Consider the Poisson problem,

$$\nabla^2 u = f. \tag{2.17}$$

The following assumes a 2D domain. As before, integrate over a CV and apply the divergence theorem (D.1) to the left hand side,

$$\int_S \nabla u \cdot \mathbf{n} \, \mathrm{d}S = \int_V f \, \mathrm{d}V. \tag{2.18}$$

Assume for now that an orthogonal, equidistant grid is used, so $\nabla u \cdot \mathbf{n}_w$ just picks out the gradient over the face w. Let h_n be the grid spacing between cell I and N, then an approximation to (2.18) is

$$\sum_{k=e,w,n,s} \frac{u_k - u_I}{h_k} S_k = fV. \tag{2.19}$$

Due to the regular grid: $S_w = S_e = h_n = h_s$, and $S_s = S_n = h_w = h_e$, and also $S_w S_n = h_n h_w = V$, hence the approximation (2.19) becomes

$$\frac{S_n}{h_n}(-2u_I + u_s + u_n) + \frac{S_w}{h_w}(-2u_I + u_w + u_e) = f_I V. \tag{2.20}$$

If also $h_w = h_n = h$, then it is further reducible

$$-4u_I + u_s + u_n + u_w + u_e = h^2 f_I. \tag{2.21}$$

This equation is called the stencil for a cell. If the grid is stretched, coefficient will be different from -4 and 1 since h_n will not equal h_w. Non-orthogonal grids implies that also corner neighbours, e.g. u_{NW}, must be included. In general a Poisson operator in a 2D finite volume grid creates a 9 point stencil. For interior cells, the coefficients in the stencil sum to zero.

Note that for a rectangular grid, the operator will be the same for finite volume as for a finite difference and finite element approach.

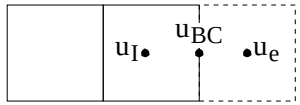


Figure 2.2: *Face boundary condition*

2.4.1 Dirichlet BC

Dirichlet boundary conditions (BC) give the value of the unknown at the boundary

$$u|_{\partial\Omega} = g. \tag{2.22}$$

Boundary conditions are applied at a cell boundary, the cell face, as depicted in Figure 2.2. Here u_e is not known, but if assuming linearity then u_e can be approximated by $u_e \approx 2u_{BC} - u_I$, giving

$$u_e - u_I = 2(u_{BC} - u_I). \tag{2.23}$$

Replacing this into the stencil (2.21) gives

$$-5u_I + u_s + u_n + u_w = h^2 f_I - 2u_{BC}. \tag{2.24}$$

Note that the boundary condition is moved to the right hand side, since it is a known quantity. The only difference is that u_e is removed from the stencil, and the coefficient for u_I is updated with -1.

The above is only a first order approximation. To be consistent and get overall second order accuracy, also boundary condition should be second order accurate. A second order approximation of u_e looks like; $u_e \approx \frac{8}{3}u_{BC} - 2u_I + \frac{1}{3}u_w$. The equation for the boundary cell will then become:

$$-6u_I + u_s + u_n + \frac{4}{3}u_w = h^2 f_I - \frac{8}{3}u_{BC}. \tag{2.25}$$

2.4.2 Neumann BC

Neumann BC give the value of the gradient of the unknown at the boundary

$$\left.\frac{\partial u}{\partial}\mathbf{n}\right|_{\partial\Omega} = g. \tag{2.26}$$

Also the Neumann boundary conditions are applied at a cell face. The gradient over the face can be approximated by

$$u_e - u_I \approx h_{Ie} \left.\frac{\partial u}{\partial n}\right|_{BC}, \tag{2.27}$$

which is a central difference second order approximation. Replacing this in the stencil 2.21 gives

$$-3u_I + u_s + u_n + u_w = h^2 f_I - h \left.\frac{\partial u}{\partial n}\right|_{BC} \tag{2.28}$$

Note again that the boundary condition is moved to the right hand side, since it is a known quantity. Only difference is that u_e is removed from the stencil, and u_I is updated with 1.

2.4.3 Robin BC

A Robin condition is a mixed Dirichlet and Neumann condition and can be expressed on the form

$$c_1 u|_{BC} + c_2 \left.\frac{\partial u}{\partial n}\right|_{BC} = g. \tag{2.29}$$

For simplicity, only first order will be considered here. Using the approximations (2.23) and (2.27), then

$$u_e = \frac{g}{\frac{c_1}{2} + \frac{c_2}{h}} - \frac{\frac{c_1}{2} - \frac{c_2}{h}}{\frac{c_1}{2} + \frac{c_2}{h}} u_I. \tag{2.30}$$

Substituting this into the stencil (2.21) gives

$$-(4 + \frac{\frac{c_1}{2} - \frac{c_2}{h}}{\frac{c_1}{2} + \frac{c_2}{h}})u_I + u_s + u_n + u_w = h^2 f_I - \frac{g}{\frac{c_1}{2} + \frac{c_2}{h}} \tag{2.31}$$

so this will produce something in between the Dirichlet and Neumann BC stencil.

It is now possible to create one equation for each CV, producing n linear equations with n unknowns. How to solve this efficiently in parallel is what the thesis is really about.

CHAPTER 3

The Victim

Many methods have been developed throughout the years to solve the equations of fluid motion. Each method has usually its own advantages and describes certain properties of the fluid motion well, while other properties are neglected, modelled badly or not at all. Here one method will be described, a package called NS3, which was developed at the International Research Center for Computational Hydrodynamics (ICCH) at the Danish Hydrolic Institute in Hørsholm. For a reference to NS3, see [Mayer98].

3.1 An NS3 Walkover

The NS3 package solves the incompressible Navier Stokes equations. It is based on a finite volume discretization and uses a variant of the pressure correction scheme from Section 2.3.4. NS3 can handle moving geometries including a free surface, which makes the procedure somewhat more complicated: The domain is discretized by a time varying curvilinear grid, and a finite volume method needs to take the movement of the grid into consideration.

3.1.1 Block Decomposition and Grid

The fluid domain is decomposed into a number of disjoint subdomains, blocks, on which a structured curvilinear grid can be applied. The grid continues naturally over a block face, meaning that the grid lines are continuous over block faces. One side of a block must have the same block as neighbour over the entire side. An example of how to split up and discretize a kind of T-shaped domain is shown in Figure 3.1.

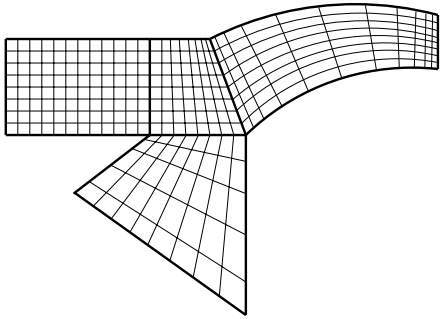


Figure 3.1: *Block decomposition and grid for a T-shaped domain*

On this grid, a cell centered finite volume discretization is applied.

3.1.2 The Solver

The Poisson problem in NS3 is solved by a multigrid method using standard finite volume coarsening and a V-cycle scheme.[1] The coarsest level consist of one cell per block. On every multigrid level either point relaxation, line relaxation or a ILLU smoother is applied, depending on the grid.

Some multigrid operations are local to a block, meaning that values from neighbouring blocks are not needed to perform the operation.

[1]For multigrid references, see [Brig87] or [McCor94]

Other operations are not. Those that are not local, need values from the first layers of cells of the neighbouring blocks. To accommodate this, there is a shadow layer around every block in every multigrid level. When an operation needs values from the neighbouring block, the values are copied to the shadow layer beforehand, and the values from the shadow layer are used to complete the operation. If a second order scheme is used, the shadow layer needs to be only one cell wide. Higher order schemes imply either a wider shadow layers or multiple consecutive 1 layer communications. The shadow layers for the blocks in Figure 3.1 are shown as gray in Figure 3.2.

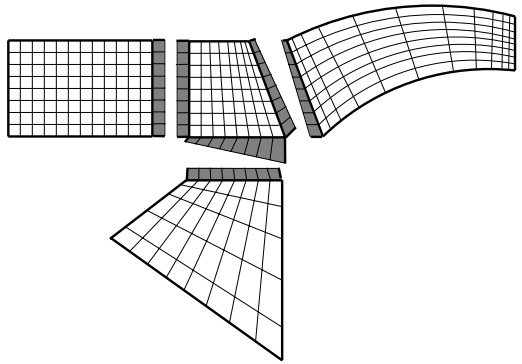


Figure 3.2: *Shadow layer*

To simplify the copying of values to the shadow layer, the scheme is created in a way, so only blocks that have direct face contact need to copy values. Blocks that only share corner points, in 2D a vertex, 3D as well an edge as a vertex, do not couple directly and no copying between these blocks is needed. E.g. in a 2 by 2 block setup, the diagonal blocks do not couple directly. This implies that the method at block corners in general no longer is of second (or higher) order.

The different operations in the multigrid procedure are best identified by presenting a typical multigrid scheme. Below is presented the V-cycle scheme.

```
function V-cycle(v^h,f^h)
    if h = coarsest level
        relax v^h sufficiently (*)
    else
        relax v^h μ1 times (*)
        compute residual: r^h = (f^h − A^h v^h) (*)
        restrict residual to level 2h: r^2h = R_h^2h r^h
        v^2h = V-cycle(0^2h,r^2h)
        Correct: v^h = v^h + R_2h^h v^2h
        relax v^h μ2 times (*)
    end
    return v^h
end
```

Operations marked with a (*) need to update the shadow layer beforehand.

3.1.3 A Parallel Setup

The very first part of this project was to make the multigrid method parallel in the most straight forward way.

The natural way of setting NS3 up in parallel, is to assign each block to a computational node. In case there are more blocks than nodes, some nodes are assigned more than one block. In case of more nodes than blocks, the biggest blocks may be subdivided into several smaller blocks. If only one node is available, all blocks are assigned to that node. Blocks on the same node are called local blocks.

To update the shadow layer now implies to communicate values between nodes. All operations in NS3 that need to communicate, are build on the same scheme using a message passing approach. Consider as an example the relax operation:


```
function Relax()
    Update all shadow layers
    for each local block
        Relax local block
        Update local block shadow layers
    end
end
```

Note that local blocks are treated slightly different than the rest. In the setup from previous section, an operation uses only old values from neighbouring blocks, it is a block additive operation. If already calculated values are used when available, the convergence rate increases and the operation is called block multiplicative. This difference is similar to the one between the standard iterative methods Jacobi (additive) and Gauss-Seidel (multiplicative). Communication between nodes in the middle of an operation is not wanted, so this is only possible between local blocks, where updating of shadow layers is just copying of memory. To exploit this, neighbouring blocks must be assigned to one node when possible.

3.1.4 Performance

The performance of NS3 on a single node machine is according to its creators quite good. NS3 relies on multigrid. Multigrid on “ugly” grids requires strong smoothers. Strong smoothers are difficult to implement over block boundaries, and fairly quickly the residual concentrates on the block boundaries. Some results can be found in [Mayer98].

The parallel version has not yet been thoroughly tested. However it does not show immediate scalability. The solver on the coarsest level presents some problems: The coarsest level is usually solved by a relative large number of relaxation sweeps. A relaxation of the entire domain on the coarsest level involves on each block only few cells and require therefor few flops. The number of unknowns that need to be communicated is of same order as the number of cells, and since communication is usually far more expensive computationally, this presents a potential bottleneck.

In case of a distributed memory machine, where the network is relatively slow, the above bottleneck is no longer potential: Most time is used on communication on the coarsest level.

On a shared memory machine that bottleneck might no longer exist. Before each relaxation step the nodes need to synchronize, and the following communication between nodes should be just a copying of memory. Apart from the synchronization this should be about as effective as the communication in the single node case, though convergence will be slower due to the additive approach.

Unfortunately a shared memory machine for us to use alone, with an effective and optimized version of the parallel package MPI, Message Passing Interface, has not been available. During the spring 2001 it should have been installed here at DTU, but at present this is not yet ready.

3.2 Problem Formulation

This brings it all back to the main purpose of this project. NS3 is an effective serial solver, but as other fluid flows modelling packages it is very expensive computationally. The goal of this project is to find, modify or create methods which fit into the NS3 package, are parallel and scalable, and off course efficient.

To fit into the NS3 package means:

- The domain is decomposed in a number of blocks which do not overlap.
- Each block uses a cell centered finite volume discretization. Diagonal blocks do not couple.
- Shadow variables are used to hold extra information. These do not necessarily have to be copies of values from the neighbouring blocks, but anything that improves convergence will be allowed.
- Due to the smoother, the multigrid solver is very efficient on blocks only, so multigrid should be used on blocks only. We will assume however, that an exact solution at each block can be found effectively using multigrid.
- We will allow the coefficients in the operator to be changed, as long as the solution is not changed.
- We will allow the presence of a global coarse grid correction, if the coarse grid dimension is small.

To be scalable, it must not suffer from communication overhead as is the case with the present parallel version. On a single node machine it should compare with the existing implementation.

CHAPTER 4

Schwarz Methods

Domain decomposition is technique where the original domain is decomposed into a set of smaller sub-domains. This is a rather old technique, the first known method was introduced by Schwarz:[1]

1869 H.A. Schwarz “invented” domain decomposition as a mathematical tool to proof existence and uniqueness theorems for solutions of general partial differential equations. Analytical solutions on simple subdomains in each iteration was used to prove solutions for complicated domains.

1965 Keith Miller proposed Schwarz method for computational purposes. There existed efficient methods for solving PDEs on simple domains, but not on domains like cars and airplanes. So the complicated domains were decomposed into simple ones and using the Schwarz method the solution could be obtained globally.

1980- The parallel computer emerged and Schwarz methods are well suited for it. Each computational node of the parallel computer is assigned a subdomain to work on and the Schwarz method

[1] For references, see [Smith96] or [Chan94]

gives the coupling of the domains and how to obtain the correct solution over all subdomains.

Today It is well known that Schwarz methods have convergence rates independent of the mesh parameters and thus are "optimal". If one refines the mesh, the convergence rate of the parallel algorithm does not change. Nevertheless the convergence rates are slow compared to other methods.

The next section will describe the original method of Schwarz and why the convergence rates are slow.

4.1 Classical Alternating Schwarz Method

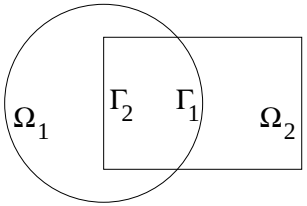


Figure 4.1: *Schwarz's original figure*

The original method by Schwarz is what today is called the classical alternating Schwarz method, and it works as follows: Let the domain be decomposed as in Figure 4.1 into two overlapping subdomains, $\Omega = \Omega_1 \cup \Omega_2$, on which we want to solve the system

$$Lu = f \quad \text{in } \Omega, \tag{4.1a}$$
$$u = g \quad \text{on } \partial\Omega, \tag{4.1b}$$

where L is some linear PDE operator. Set Γ_1 and Γ_2 to be the artificial boundaries of Ω_1 and Ω_2 respectively. The classical alternating Schwarz method iteratively solves for u_1^{n+1} in Ω_1 using as BC on Γ_1

old values from Ω_2, u_2^n,

$$Lu_1^{n+1} = f \qquad \text{in } \Omega_1, \tag{4.2a}$$
$$u_1^{n+1} = g \qquad \text{on } \partial\Omega_1 \setminus \Gamma_1, \tag{4.2b}$$
$$u_1^{n+1} = u_2^n|_{\Gamma_1} \quad \text{on } \Gamma_1, \tag{4.2c}$$

followed by a solve for for u_2^{n+1} in Ω_2 now using the newly computed values of u_1^{n+1} as BC on Γ_2,

$$Lu_2^{n+1} = f \qquad \text{in } \Omega_2, \tag{4.3a}$$
$$u_2^{n+1} = g \qquad \text{on } \partial\Omega_2 \setminus \Gamma_2, \tag{4.3b}$$
$$u_2^{n+1} = u_1^{n+1}|_{\Gamma_2} \quad \text{on } \Gamma_2. \tag{4.3c}$$

This is a multiplicative method since values already calculated are used when solving the next block, $u_1^{n+1}|_{\Gamma_2}$. If instead only old values are used in the second solve, $u_1^n|_{\Gamma_2}$, the method turns additive. This procedure is easily extendible to several subdomains.

4.2 Performance

The convergence speed of this method depends strongly on the size of the overlap $\Omega_1 \cap \Omega_2$, and the number of blocks, which is shown in the example below.

Example 4.1: Convergence of 1D Laplace Problem Consider the following one dimensional Laplace problem:

$$u'' = 0 \quad \text{in } 0 < x < 1, \qquad u(0) = u(1) = 1, \tag{4.4}$$

with the obvious solution $u(x) = 1$. Figure 4.2 shows how the iterative solution behaves when starting with zero initial guess, $u^0 = 0$, and using a classical multiplicative alternating Schwarz method decomposed into two overlapping subdomains.

Doing the same with varying sizes of the overlap, it is easily seen, e.g. in the left part of Figure 4.3, that a smaller overlap implies slower convergence.

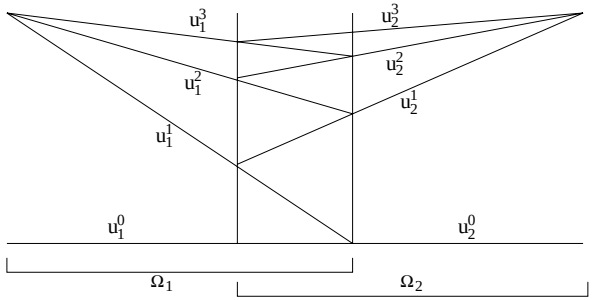


Figure 4.2: *Evolution of solution*

If the overlap is complete, that is $\Omega = \Omega_1 = \Omega_2$, then the exact solution is achieved in one iteration. If there is no overlap, the method does not converge at all.

Also, if the domain is decomposed into many subdomains, convergence rapidly decreases as the number of subdomains increases, as is seen in the right part of the figure.

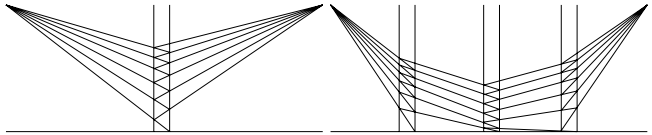


Figure 4.3: *Evolution of solution for the first 6 iterations*

End of Example 4.1.

Note, that the convergence is independent of how each subdomain is discretized and solved, and this is why it is called "optimal". The grids in each subdomain do not have to match, that is the grid nodes do not have to coincide in the overlapping regions. They most likely will not in domains like in Figure 4.1. One difference in a such non-matching case is, that an interpolating operator is needed to calculate the values of u on the artificial boundaries $u_i^n|_{\Gamma_j}$, $i \neq j$. If the grids

match though, it is possible to improve convergence by e.g. a Krylov subspace method.[2]

Example 4.2: Improve Convergence by Krylov Subspace Method
The problem considered is a Poisson problem

$$\begin{aligned} -\Delta u &= xe^y \qquad \text{in } \Omega =]0,2[\times]0,1[\\ -u &= xe^y \qquad \text{on } \partial\Omega \end{aligned} \tag{4.5}$$

The problem is discretized using a centered finite difference approximation, second order, with a spacing of $h = \frac{1}{N-1}$. This is solved for several values of N and the size of overlap measured in number of overlapping cells, n_o. A "simple" classical alternating Schwarz is compared with a Krylov subspace accelerated version (GMRES with a restart of 10). Both the additive and multiplicative versions are tested. Details about the preconditioners used can be found in [Smith96], where also a similar example is presented.

The number of iterations used to decrease the residual by a factor 10^{-10} are recorded and the results are listed in Table 4.1. Numbers in parenthesis indicate that convergence is almost achieved already at that point.

Table 4.1 show that convergence is independent of the grid spacing. Going from $N = 11$ to $N = 21$ halves the grid spacing. To get the same amount of overlap, double as many cells are needed, n_o should be doubled. To get independence of the mesh spacing, $(N, n_o) = (11, 2)$ should match $(N, n_o) = (21, 4)$ and $(N, n_o) = (41, 8)$. This is the case for all 4 methods.

The most important to notice here is that the accelerated versions are less sensitive to the amount of overlap than the simple. This becomes more evident as the problem grows bigger and the overlap decreases. The simple method about doubles the number of iterations when the overlap is halved. The accelerated version only increase slightly.

Notice the factor of 2 between the simple additive and multiplicative versions, as is also the case for the classical Jacobi and Gauss-Seidel methods. This is almost also the case for the accelerated version, though not as evident.

This should not be used to compare the methods against each other, since the GMRES method uses quite more computation time for each iteration compared to the simple one. It is the increase in iteration count as the problem grows, that is interesting.

[2] See Appendix B for an introduction to Krylov subspace methods.

N	n_o	Simple additive	Simple multiplicative	Krylov additive	Krylov multiplicative
	1	35	18	13	7(6)
11	2	19	10(9)	8	5
	4	10	5	6	3
	8	5	3	4	2
	1	>50	34	21	9
21	2	35	17	12	6
	3	19	9	8	5(4)
	8	10	5	6	3
	1		>50	35	12
41	2	>50	35	21	8
	3	35	17	12	6
	8	19	9	8	4

Table 4.1:

At last, the multiplicative methods are not parallelizable without some kind of additive grouping (coloring), which will give the method a parallel performance somewhere in between the additive and multiplicative, depending on the number of groups (colors).

End of Example 4.2.

A more thorough motivation than given in Example 4.1 for the increasing iteration count as function of the size of overlap and the number of blocks can be found in [Smith96] p. 24ff.

Convergence can be improved by a coarse grid correction or another multigrid approach, and [Smith96] shows that convergence can be made independent of as well the grid spacing as the block size, but will still depend on the size of the overlap.

The classical alternating Schwarz method cannot directly be implemented in NS3. It is based on overlapping blocks, and the NS3 grid generator need to be updated to accommodate this. The method itself is based on local block solves, and fits in that sense fine into NS3: In

every step, $u_i^n|_{\Gamma_j}$, $i \neq j$ is placed in the shadow of block i, and the block is solved using this as BC.

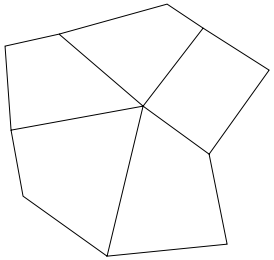


Figure 4.4: *Domain decomposition*

CHAPTER 5

Non Overlapping Domain Decomposition

The main interest for this project is in non overlapping domain decomposition. The extra calculation due to the needed overlap is not wanted, and the grid generator in NS3 can as mentioned earlier not handle overlapping blocks.

The classical alternating Schwarz method does not converge without overlap, so another approach is necessary.

Inspired at first by a simple 1D example like Example 5.1 presented to me by Stefan Mayer, and secondly by [Rice98a] which formalized this to any dimension, the following approach is investigated: At odd iterations Dirichlet values are exchanged between blocks, while at even iterations Neumann values are exchanged. Let us formalize this.

Decompose the domain into two blocks, $\Omega = \Omega_1 \cup \Omega_2$, $\Omega_1 \cap \Omega_2 = 0$, separated by the boundary $\Gamma = \partial\Omega_1 \cap \partial\Omega_2$. A such interior boundary Γ will be called an interface. A solution to the system (4.1) is sought.

At odd iteration, solve the following system, exchanging Dirichlet

data over Γ

$$Lu_1^{n+1} = f \text{ in } \Omega_1, \quad u_1^{n+1} = \alpha u_1^n + (1-\alpha)u_2^n \text{ on } \Gamma, \tag{5.1a}$$
$$Lu_2^{n+1} = f \text{ in } \Omega_2, \quad u_2^{n+1} = \alpha u_1^n + (1-\alpha)u_2^n \text{ on } \Gamma, \tag{5.1b}$$

while a even iterations solve instead the following exchanging Neumann data over Γ

$$Lu_1^{n+1} = f \text{ in } \Omega_1, \quad \frac{\partial u_1^{n+1}}{\partial n_1} = \beta\frac{\partial u_1^n}{\partial n_1} + (1-\beta)\frac{\partial u_2^n}{\partial n_1} \text{ on } \Gamma, \tag{5.2a}$$
$$Lu_2^{n+1} = f \text{ in } \Omega_2, \quad \frac{\partial u_2^{n+1}}{\partial n_2} = \beta\frac{\partial u_1^n}{\partial n_2} + (1-\beta)\frac{\partial u_2^n}{\partial n_2} \text{ on } \Gamma, \tag{5.2b}$$

where n_i is the outward normal vector to Γ. The α and β parameters decide how to weight the values from the two blocks. This will be referred to as the DN (Dirichlet-Neumann) method.

Example 5.1: Convergence of 1D Laplace Problem Using DN Method Consider the following one dimensional Laplace problem:

$$u'' = 0 \quad \text{in } 0 < x < 1 \qquad u(0) = 0, u(1) = 1 \tag{5.3}$$

Figure 5.1 shows how the iterative solution behaves when starting with zero initial guess, $u^0 = 0$, and the DN method with $\alpha = \beta = \frac{1}{2}$ is used.

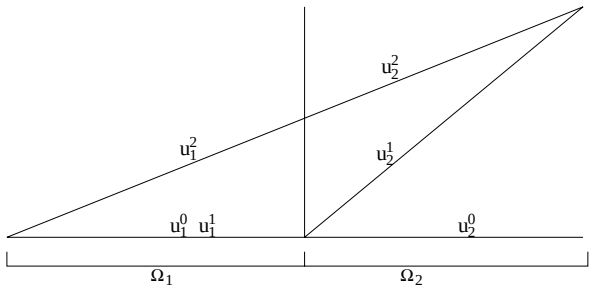


Figure 5.1: *Convergence of 1D Laplace problem using DN method*

In the first iteration, $u(\Gamma) = 0$ is used as BC. The second iteration uses an average of the derivative of u^1 at Γ weighted between the two blocks which equals exactly the derivative of the solution. Hence the correct solution is found in 2 iterations.

If the two blocks are not equally big, other values of α and β can produce the correct result in also 2 iterations. Though, α and β are subject for optimization.

Discretization within the block does not change the convergence behavior.

End of Example 5.1.

The rest of this chapter is devoted to implementing the DN method in a linear algebra context. It will be based on the formulation of stationary iterative methods, which therefor will be introduced in the first section. Secondly the notion of shadow variables will be formalized, in order to model the shadow variables as the are used in NS3.

5.1 Stationary Iterative Block Methods

This section will state some basic theory for stationary iterative methods, for reference see [Bark92].

Consider the system

$$\mathbf{Au} = \mathbf{f}. \tag{5.4}$$

Split $\mathbf{A}$ into two matrices $\mathbf{A} = \mathbf{M} + \mathbf{N}$, and rearrange.

$$\mathbf{Mu} = \mathbf{f} - \mathbf{Nu} \tag{5.5}$$

If $\mathbf{M}$ is nonsingular, then this can be the basis for an iterative solver, namely:

$$\mathbf{Mu}^{n+1} = \mathbf{f} - \mathbf{Nu}^n \tag{5.6}$$

Subtracting (5.5) from (5.6) and introducing the error vector $\mathbf{e}^k = \mathbf{u}^k - \mathbf{u}$ gives

$$\mathbf{Me}^{n+1} = -\mathbf{Ne}^n \tag{5.7}$$

The evolution of the error is determined by

$$\mathbf{e}^{n+1} = \mathbf{G}\mathbf{e}^n \tag{5.8}$$

where $\mathbf{G} = -\mathbf{M}^{-1}\mathbf{N}$. The matrix $\mathbf{G}$ is usually called the iteration matrix. This can be shown to converge if and only if $\rho(\mathbf{G})$, the spectral radius of $\mathbf{G}$, is less than one: The largest absolute eigenvalue of $\mathbf{G}$ must be less than one,

$$\rho(\mathbf{G}) = \max_i |\lambda_i| < 1. \tag{5.9}$$

Proof: Denote the eigensolutions of $\mathbf{G}$ by

$$(\lambda_i, \mathbf{w}_i), \quad i = 1, \ldots, n \tag{5.10}$$

Assume for simplicity that the eigenvectors are linearly independent. Then any initial error can be expressed on the form

$$\mathbf{e}^0 = c_1\mathbf{w}_1 + c_2\mathbf{w}_2 + \cdots + c_n\mathbf{w}_n \tag{5.11}$$

and

$$\mathbf{e}^k = \mathbf{G}^k\mathbf{e}^0 = \sum_{i=1}^{n} c_i\lambda_i^k\mathbf{w}_i \tag{5.12}$$

where $\mathbf{e}^k \to 0$ if and only if $|\lambda_i| < 1$, all i □

The proof indicates that the smaller spectral radius, the faster $\mathbf{e}^k$ goes towards zero, and the faster convergence. The task is to find a splitting, where the System (5.6) is easy to solve, i.e. a property of $\mathbf{M}$, while $\rho(\mathbf{G})$ is as small as possible.

Consider a splitting of $\mathbf{A}$ on the form

$$\mathbf{A} = \mathbf{A}_b + \mathbf{A}_L + \mathbf{A}_U \tag{5.13}$$

where $\mathbf{A}_b$ is the block diagonal, $\mathbf{A}_L$ the strictly lower block triangular and $\mathbf{A}_U$ the strictly upper block triangular part of $\mathbf{A}$. Below is presented some examples of the use of these in stationary iterative methods.

Example 5.2: The block Jacobi method is an additive method and uses only old values to calculate new values, which is equivalent to

$$\mathbf{M} = \mathbf{A}_b, \qquad \mathbf{N} = \mathbf{A}_L + \mathbf{A}_U \tag{5.14}$$

The block Gauss-Seidel method is a multiplicative method and reuses values already calculated, which is equivalent to

$$\mathbf{M} = \mathbf{A}_b + \mathbf{A}_L, \qquad \mathbf{N} = \mathbf{A}_U \tag{5.15}$$

As is the case with the standard iterative methods of Jacobi and Gauss-Seidel, Jacobi uses approximately twice as many iterations to converge, while the Gauss-Seidel being multiplicative is not possible to parallelize without some kind of additive grouping (coloring).

End of Example 5.2.

From a parallelization point of view, stationary iterative methods are in general not optimal. If $\mathbf{A}$ is dense, then usually also $\mathbf{N}$ will be dense for $\mathbf{M}$ to be easy to invert. For every block to be able to update the right hand side of the iterative procedure (5.6), it is required that the entire preliminary solution $\mathbf{u}_k$ is known to this block, in order to calculate the relevant part of $\mathbf{N}\mathbf{u}^k$. In a parallel environment this will imply a lot of communication, which in many cases will introduce a bottleneck.

However, if $\mathbf{A}$ arises from a second order approximation of a linear partial differential operator of at most second order (Poisson operator), it will be sparse and only nearest neighbour interaction is included. $\mathbf{N}$ will have entries corresponding to cells at the block interface which depend on cells from the neighbouring blocks. Only values at the interface need to be communicated between only neighbouring blocks to be able to calculate the relevant part of $\mathbf{N}\mathbf{u}^k$.

In case of the Poisson operator, the splittings in Example 5.2 used in a stationary iterative method produce a method which is equivalent to the classical alternating Schwarz method with a one cell overlap.

5.2 Shadow Variables

This section will formalize the notion of shadow variables in a linear algebra context. It introduces a way to construct the shadow variables to represent the block coupling, and make each block solver independent of values from other blocks. Consider the system

$$\mathbf{A}\mathbf{u} = \mathbf{f}. \tag{5.16}$$

Example 5.3 presents the idea of how to create and use shadow variables.

Example 5.3: Construction of Shadow Variables in 1D Consider a 1D problem decomposed into two blocks. Split the solution vector into two parts corresponding to the two blocks, $\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 \end{bmatrix}^T$. Write system (5.16) as

$$\left[\begin{array}{c|c} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \hline \mathbf{A}_{21} & \mathbf{A}_{22} \end{array}\right] \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}. \tag{5.17}$$

Since the system only include nearest neighbour interaction, the matrices $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ have only one nonzero column each. Therefor let us look at each column of the matrices $\mathbf{A}_{ij}$:

$$\left[\begin{array}{ccc|ccc} \mathbf{a}_{11}^1 & \cdots & \mathbf{a}_{11}^{n_1} & \mathbf{a}_{12}^1 & & \\ \hline & & \mathbf{a}_{21}^{n_1} & \mathbf{a}_{22}^1 & \cdots & \mathbf{a}_{22}^{n_2} \end{array}\right] \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix}, \tag{5.18}$$

where n_1 and n_2 are the number of unknowns in $\mathbf{u}_1$ and $\mathbf{u}_1$ respectively.

Introduce the two shadow variables u_1^s and u_2^s: Set u_1^s to equal the first value of the vector $\mathbf{u}_2$, while u_2^s is set to equal the last value of $\mathbf{u}_1$, $u_1^s = (\mathbf{u}_2)_1$ and $u_2^s = (\mathbf{u}_1)_{n_1}$. Add these new variables and equations to the system

$$\left[\begin{array}{ccc|ccc|cc} \mathbf{a}_{11}^1 & \cdots & \mathbf{a}_{11}^{n_1} & \mathbf{a}_{12}^1 & & & & \\ \hline & & \mathbf{a}_{21}^{n_1} & \mathbf{a}_{22}^1 & \cdots & \mathbf{a}_{22}^{n_2} & & \\ \hline & & & 1 & & & -1 & \\ & & 1 & & & & & -1 \end{array}\right] \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ u_1^s \\ u_2^s \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ 0 \\ 0 \end{bmatrix}. \tag{5.19}$$

Since the shadow variables are exactly copies of variables next to the interface, the following system will have the same solution:

$$\left[\begin{array}{ccc|ccc|cc} \mathbf{a}_{11}^1 & \cdots & \mathbf{a}_{11}^{n_1} & & & & \mathbf{a}_{12}^1 & \\ \hline & & & \mathbf{a}_{22}^1 & \cdots & \mathbf{a}_{22}^{n_2} & & \mathbf{a}_{21}^{n_1} \\ \hline & & & 1 & & & -1 & \\ & & 1 & & & & & -1 \end{array}\right] \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ u_1^s \\ u_2^s \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ 0 \\ 0 \end{bmatrix}. \tag{5.20}$$

Note that no change have been made to the block diagonal part of $\mathbf{A}$, the two matrices $\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ are not touched. However, we have made each block independent of variables from the other block, and instead dependent on its own shadow variables, thereby represented the coupling between the blocks through the shadow variables.

End of Example 5.3.

Let us generalize the procedure in Example 5.3, using a more compact notation. Assume the domain is decomposed into k non-overlapping blocks, and assume furthermore that the elements of the solution vector $\mathbf{u}$ are ordered block wise.

Besides the unknowns in $\mathbf{u}$, each block furthermore has a shadow layer, whose unknowns we will add to the set of unknowns. Let the vector $\mathbf{u}_s$ consist of all the shadow unknowns, then define a new expanded solution vector as $\tilde{\mathbf{u}} = \begin{bmatrix} \mathbf{u} & \mathbf{u}_s \end{bmatrix}^T$.

For this new solution vector $\tilde{\mathbf{u}}$ we wish to make a system matrix $\mathbf{A}_s$ of the form

$$\mathbf{A}_s\tilde{\mathbf{u}} = \begin{bmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{R} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix}, \tag{5.21}$$

where: The matrix $\mathbf{B}$ is the block diagonal part of $\mathbf{A}$ consisting of k blocks

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & & \\ & \ddots & \\ & & \mathbf{B}_k \end{bmatrix}. \tag{5.22}$$

The $\begin{bmatrix} \mathbf{R} & -\mathbf{I} \end{bmatrix}$ rows correspond to the copying of variables to the shadow variables and $\mathbf{C}$ is the use of the shadow variables in each block.

It is a requirement that the $\mathbf{u}$ part of the expanded solution vector also solves the original system (5.16). Note especially that a system of the form $\mathbf{B}\mathbf{u} = \mathbf{f}$ can be solved in parallel because of the block diagonal structure of $\mathbf{B}$.

To solve the expanded system (5.21), invert $\mathbf{A}_s$. This can be done analytically in a block context and the result is[1]

$$\begin{bmatrix} \mathbf{u} \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{B}}^{-1} & \tilde{\mathbf{B}}^{-1}\mathbf{C} \\ \mathbf{R}\tilde{\mathbf{B}}^{-1} & -\mathbf{I} + \mathbf{R}\tilde{\mathbf{B}}^{-1}\mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{0} \end{bmatrix}, \quad \tilde{\mathbf{B}} = \mathbf{B} + \mathbf{CR}. \tag{5.23}$$

Consider only the part of the solution vector, $\mathbf{u}$, corresponding to the solution of the original system:

$$\mathbf{u} = \tilde{\mathbf{B}}^{-1}\mathbf{f} = (\mathbf{B} + \mathbf{CR})^{-1}\mathbf{f}. \tag{5.24}$$

This should have the same solution as before the decomposition $\mathbf{u} = \mathbf{A}^{-1}\mathbf{f}$, so a requirement is that

$$\mathbf{B} + \mathbf{CR} = \mathbf{A} \tag{5.25}$$

Note that the new formulation allow a more general usage of the shadow variables than in Example 5.3: The matrices $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{R}$ can be chosen freely as long as they fulfill the requirement (5.25).

The following will continue Example 5.3 in case of more than 1D, and show how to create and use the shadow variables in general.

A straight forward way to produce the form of Equation (5.21) is by moving the non block diagonal part of $\mathbf{A}$ "to the right". In case of 2 blocks, this will look like

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \rightarrow \mathbf{A}_s = \left[\begin{array}{cc|cc} \mathbf{A}_{11} & \mathbf{0} & \mathbf{C}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{0} & \mathbf{C}_{21} \\ \hline \mathbf{0} & \tilde{\mathbf{I}}_{12} & -\mathbf{I} & \mathbf{0} \\ \tilde{\mathbf{I}}_{21} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{array}\right], \tag{5.26}$$

where $\mathbf{C}_{12}$ is $\mathbf{A}_{12}$ with all columns having only zero elements removed, and $\tilde{\mathbf{I}}_{12}$ is an identity matrix where rows have been removed corresponding to the removed columns of $\mathbf{A}_{12}$. Similar for $\mathbf{C}_{21}$ and $\tilde{\mathbf{I}}_{21}$.

[1] For the inversion, see Equation (D.15) in Appendix D.2

The system matrix $\mathbf{A}_s$ obeys Equation (5.25), since the $\tilde{\mathbf{I}}_{ij}$ take each columns of $\mathbf{C}_{ij}$ and place it where it originally came from in $\mathbf{A}$.

Instead of removing zero columns, it is possible to remove zero rows. Then the non block diagonal is moved down instead of right, producing:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \rightarrow \mathbf{A}_s = \left[\begin{array}{cc|cc} \mathbf{A}_{11} & \mathbf{0} & \tilde{\mathbf{J}}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{0} & \tilde{\mathbf{J}}_{21} \\ \hline \mathbf{0} & \mathbf{R}_{12} & -\mathbf{I} & \mathbf{0} \\ \mathbf{R}_{21} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{array}\right]. \tag{5.27}$$

If $\mathbf{A}$ is symmetric, then $\mathbf{R}_{ij} = \mathbf{C}_{ij}^T$ and $\tilde{\mathbf{I}}_{ij} = \tilde{\mathbf{J}}_{ij}^T$. It is possible to change between Equation (5.26) and (5.27) by a suitable series of row and column operations on rows and columns corresponding to the shadow variables, using a kind of Gaussian elimination. However, we will mostly use the procedure in Equation (5.26).

The process of creating shadow variables and form a system like in Equation (5.26) can be generalized to systems decomposed in more than two blocks, giving 4 matrices $\tilde{\mathbf{I}}_{ij}$, $\tilde{\mathbf{I}}_{ji}$, $\mathbf{C}_{ij}$ and $\mathbf{C}_{ji}$ for all pair of blocks (i, j) with a common interface.

In this context, the classical alternating Schwarz method can be implemented, as is shown in Example 5.4 below.

Example 5.4: Classical Alternating Schwarz This example will show how to implement the classical alternating Schwarz method with one cell overlap, using a stationary iterative method and the setup in Equation (5.26). That is we want to split up the system matrix in Equation (5.26) into a sum of two matrices, $\mathbf{A}_s = \mathbf{M} + \mathbf{N}$.

The purpose of $\mathbf{N}$ is to copy the necessary values of $\mathbf{u}^k$ to the shadow variables for the next iteration $\mathbf{u}_s^{k+1}$,

$$\mathbf{u}_s^{k+1} = \begin{bmatrix} & \tilde{\mathbf{I}}_{12} \\ \tilde{\mathbf{I}}_{21} & \end{bmatrix} \mathbf{u}^k. \tag{5.28}$$

The splitting of $\mathbf{A}_s$ into

$$\mathbf{M} = \left[\begin{array}{cc|cc} \mathbf{A}_{11} & & \mathbf{C}_{12} & \\ & \mathbf{A}_{22} & & \mathbf{C}_{21} \\ \hline & & -\mathbf{I} & \\ & & & -\mathbf{I} \end{array}\right], \qquad \mathbf{N} = \left[\begin{array}{cc|cc} & & & \\ & & \tilde{\mathbf{I}}_{12} & \\ \hline \tilde{\mathbf{I}}_{21} & & & \end{array}\right], \quad (5.29)$$

will accomplish this in the Jacobi case.

Note that the coefficients in Equation (5.28) for $\tilde{\mathbf{u}}^k$ go into the lower part of $\mathbf{N}$, while coefficients for $\tilde{\mathbf{u}}^{k+1}$ go into the lower part of $\mathbf{M}$. That is how it generally works when making an $\mathbf{M} + \mathbf{N}$ splitting.

The block Gauss-Seidel method reuses values already calculated, which is equivalent to

$$\mathbf{M} = \left[\begin{array}{cc|cc} \mathbf{A}_{11} & & \mathbf{C}_{12} & \\ & \mathbf{A}_{22} & & \mathbf{C}_{21} \\ \hline \tilde{\mathbf{I}}_{21} & & -\mathbf{I} & \\ & & & -\mathbf{I} \end{array}\right], \qquad \mathbf{N} = \left[\begin{array}{cc|cc} & & & \\ & & \tilde{\mathbf{I}}_{12} & \\ \hline & & & \end{array}\right] \quad (5.30)$$

These two splittings used in a stationary iterative method are equivalent to the additive and multiplicative versions of the classical alternating Schwarz method with a one cell overlap.

End of Example 5.4.

The next Example 5.5 below is intended to show how the different matrices actually look for a specific 2 by 2 block decomposition of a square.

Example 5.5: The Structure of the system matrix A The left part of Figure 5.2 shows a 2×2 block domain decomposition with the block numbering. The vector in the left bottom indicate where the numbering of cells start within each block, and which is the main direction for the numbering. In this case the numbering continues naturally over the vertical interface between block 1 and 2, but over the horizontal interface, the numbering order is no longer natural.

The right part of the figure shows a "spy of $\mathbf{A}$": At which positions of the matrix $\mathbf{A}$ that have nonzero elements.[2] The elements far from the main diagonal corresponds exactly to cells at the horizontal interface where the numbering of cells does not continue naturally.

[2]The word "spy" comes from Matlab: To plot in Matlab all nonzero elements of a matrix $\mathbf{A}$ as in the right part of Figure 5.2 is done by typing `spy(A);` In some Matlab circles spy has become a concept more than a command.

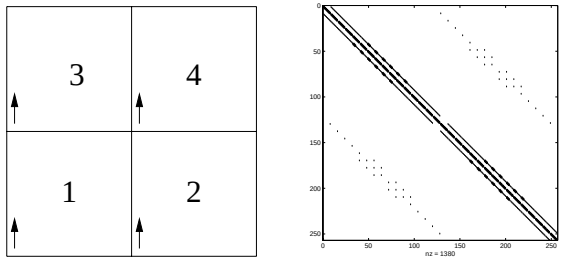


Figure 5.2: *Block domain decomposition and structure of system matrix*

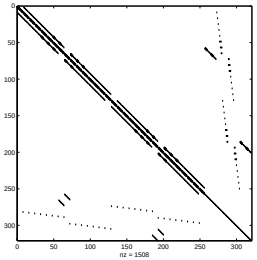


Figure 5.3: *Structure of system matrix with shadow variables*

In Figure 5.3, shadow variables have been introduced, and a structure like Equation (5.26) has been achieved.

End of Example 5.5.

Let us summarize what we have done so far in this section:

- The use of shadow variables as a copy of values from neighbouring blocks have been formalized. A procedure to create the matrix $\mathbf{A}_s$ has been given. The one in Equation (5.26) will be used if not otherwise specified.
- Given the system (5.21) and a splitting of $\mathbf{A}_s$ into $\mathbf{M} + \mathbf{N}$, the iterative procedure (5.6) can solve this system. A basic step in this procedure is the solution of

$$\mathbf{u}^{n+1} = \mathbf{M}^{-1}\mathbf{f}(\mathbf{u}^n). \tag{5.31}$$

  The matrix $\mathbf{M}$ consists of the block diagonal matrix $\mathbf{B}$, which can be solved in parallel using local block solvers. It will be assumed that local block solvers are available.

Note that the solution to the original system is intact.

We have shown how to implement a classical alternating Schwarz method with one cell overlap. However, we want to obtain better convergence, hence a question that arise is: What can be done to $\mathbf{A}_s$ without changing the solution to the original system? The intention is to only apply operations on $\mathbf{A}_s$ that do not change the solution of the original system, to guarantee that the new system produces the correct solution. Secondly, how is the splitting into $\mathbf{M} + \mathbf{N}$ done so that the iterative procedure (5.6) converges as fast as possible to the correct solution? And finally, how can this implement the DN method?

First of all: What can be done directly to $\mathbf{A}_s$?

- Row operations do not change the solution. To avoid updating the right hand side also, only rows with zero right hand side will be scaled or added to other rows. The bottom rows of Equation (5.21) corresponding to the shadow variable equations are created with zero right hand side, which implies that the $\begin{bmatrix} \mathbf{R} & -\mathbf{I} \end{bmatrix}$ rows of $\mathbf{A}_s$ can be added to any other row.
- Column operations on columns corresponding to the shadow variables, $\begin{bmatrix} \mathbf{C}^T & -\mathbf{I} \end{bmatrix}^T$. The shadow variables were originally introduced as copies of values from neighbouring blocks, but any meaning given to these variables will be allowed as long as it improves convergence.

How the splitting is done to implement the DN method will be addressed in the following sections. The presentation is based on a two block system, but can be extended to many blocks by repeating the procedure for every block to block interface.

We will start by introducing the notation which will be used in the sections to come.

Figure 5.4: *Boundary cell and shadow layer*

Two blocks, X and Y will be considered. Cells in X are split into cells in the interior and cells on the interface, $\mathbf{x} = [\mathbf{x}_i\ \mathbf{x}_b]^T$. The shadow variables of X will be called $\mathbf{x}_s$. Similar for Y. This is depicted in Figure 5.4. The system (5.21) or its special case (5.26) will be written as

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & \mathbf{C}_x & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 & \mathbf{0} & \mathbf{C}_y \\ \mathbf{0} & \tilde{\mathbf{I}}_y & -\mathbf{I} & \mathbf{0} \\ \tilde{\mathbf{I}}_x & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{x}_s \\ \mathbf{y}_s \end{bmatrix} = \begin{bmatrix} \mathbf{f}_x \\ \mathbf{f}_y \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}. \tag{5.32}$$

The $\tilde{\mathbf{I}}_x$ picks out the cells at the interface, $\mathbf{x}_b = \tilde{\mathbf{I}}_x\mathbf{x}$, and similar $\mathbf{y}_b = \tilde{\mathbf{I}}_y\mathbf{y}$. The last rows of the system matrix in Equation (5.32), the copying of values from neighbouring blocks, simply state element-vice that $(y_b)_i - (x_s)_i = 0$ or $(x_s)_i = (y_b)_i$.

5.3 An Approximate DN Method

This section will show how to implement the DN method in the context from previous section. The DN method consist of two steps, a Dirichlet step and a Neumann step. The two steps is presented one by one.

5.3.1 A Dirichlet Step

The stationary iterative methods used so far are based on an update of the shadow variables using the newly calculated values from the neighbouring block,

$$\mathbf{x}_s^{k+1} = \mathbf{y}_b^k, \tag{5.33a}$$
$$\mathbf{y}_s^{k+1} = \mathbf{x}_b^k. \tag{5.33b}$$

A generalization is to update with a mixture of the old shadow value and the interface value from the neighbouring block,

$$\begin{aligned} \mathbf{x}_s^{k+1} &= \alpha \mathbf{y}_b^k + (1-\alpha)\mathbf{x}_s^k, \\ \mathbf{y}_s^{k+1} &= \alpha \mathbf{x}_b^k + (1-\alpha)\mathbf{y}_s^k. \end{aligned} \tag{5.34a}$$

This can be accomplished by multiplying the shadow rows by α

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & \mathbf{C}_x & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 & \mathbf{0} & \mathbf{C}_y \\ \mathbf{0} & \alpha\tilde{\mathbf{I}}_y & -\alpha\mathbf{I} & \mathbf{0} \\ \alpha\tilde{\mathbf{I}}_x & \mathbf{0} & \mathbf{0} & -\alpha\mathbf{I} \end{bmatrix}, \tag{5.34b}$$

and split using the same $\mathbf{M}$ as for the classical alternating Schwarz splitting (5.29),

$$\mathbf{M} = \begin{bmatrix} \mathbf{B}_1 & & \mathbf{C}_x & \\ & \mathbf{B}_2 & & \mathbf{C}_y \\ & & -\mathbf{I} & \\ & & & -\mathbf{I} \end{bmatrix}, \tag{5.34c}$$

$$\mathbf{N} = \begin{bmatrix} & & & \\ & \alpha\tilde{\mathbf{I}}_y & (1-\alpha)\mathbf{I} & \\ \alpha\tilde{\mathbf{I}}_x & & & (1-\alpha)\mathbf{I} \end{bmatrix}. \tag{5.34d}$$

Note that the updating of shadow variables (5.34a) can be written as

$$\mathbf{0} = \begin{bmatrix} -\mathbf{I} & \\ & -\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x}_s \\ \mathbf{y}_s \end{bmatrix}^{k+1} + \begin{bmatrix} & \alpha\tilde{\mathbf{I}}_y & (1-\alpha)\mathbf{I} & \\ \alpha\tilde{\mathbf{I}}_x & & & (1-\alpha)\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{x}_s \\ \mathbf{y}_s \end{bmatrix}^k, \tag{5.35}$$

where as in Example 5.4 the coefficients for the unknowns at level $k+1$ go into the lower part of $\mathbf{M}$ and the coefficients for the unknowns at level k go into the lower part of $\mathbf{N}$.

Alone, the procedure can be viewed as a kind of underrelaxation. Choosing e.g. $\alpha = \frac{1}{2}$ will correspond to implementing a classical alternating Schwarz method with only $\alpha = \frac{1}{2}$ cell overlap, hence giving slower convergence.

5.3.2 A Neumann Step

As in the DN method, the even steps should implement a Neumann boundary condition on the block interface. Consider the difference over the interface, $\mathbf{y}_b - \mathbf{x}_b$. From each side of the interface this can be approximated by

$$\begin{aligned} \mathbf{x}_s^{k+1} - \mathbf{x}_b^{k+1} &= \beta(\mathbf{y}_b^k - \mathbf{y}_s^k) + (1-\beta)(\mathbf{x}_s^k - \mathbf{x}_b^k), \\ \mathbf{y}_s^{k+1} - \mathbf{y}_b^{k+1} &= \beta(\mathbf{x}_b^k - \mathbf{x}_s^k) + (1-\beta)(\mathbf{y}_s^k - \mathbf{y}_b^k). \end{aligned} \tag{5.36a}$$

This can be accomplished in a similar way as in the Dirichlet step. First add up the two shadow block rows and multiply these new rows by β,

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & \mathbf{C}_x & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 & \mathbf{0} & \mathbf{C}_y \\ \beta\tilde{\mathbf{I}}_x & \beta\tilde{\mathbf{I}}_y & -\beta\mathbf{I} & -\beta\mathbf{I} \\ \beta\tilde{\mathbf{I}}_x & \beta\tilde{\mathbf{I}}_y & -\beta\mathbf{I} & -\beta\mathbf{I} \end{bmatrix}. \tag{5.36b}$$

The splitting will this time be different, since we want $\mathbf{x}_s - \mathbf{x}_b$ in $\mathbf{M}$ to implement the Neumann condition,

$$\mathbf{M} = \begin{bmatrix} \mathbf{B}_1 & & \mathbf{C}_x & \\ & \mathbf{B}_2 & & \mathbf{C}_y \\ \tilde{\mathbf{I}}_x & & -\mathbf{I} & \\ & \tilde{\mathbf{I}}_y & & -\mathbf{I} \end{bmatrix} \tag{5.36c}$$

$$\mathbf{N} = \begin{bmatrix} & & & \\ & & & \\ -(1-\beta)\tilde{\mathbf{I}}_x & \beta\tilde{\mathbf{I}}_y & (1-\beta)\mathbf{I} & -\beta\mathbf{I} \\ \beta\tilde{\mathbf{I}}_x & -(1-\beta)\tilde{\mathbf{I}}_y & -\beta\mathbf{I} & (1-\beta)\mathbf{I} \end{bmatrix} \tag{5.36d}$$

The matrix in Equation (5.36b) is singular, since the last block rows are the same. To complete the system, equations telling e.g. $\mathbf{x}_s = \mathbf{y}_b$ are needed. $\mathbf{M}$ is regular, hence the stationary iterative method can be applied regardless of the singularity of the system, though convergence is no longer guaranteed.

[Bark92] states the following theory for stationary iterative methods when applied to singular systems:

- The system must be consistent, that is having a right hand side which belongs to the range of $\mathbf{A}_s$, $f \in R(\mathbf{A}_s)$.
- The iteration matrix $\mathbf{G} = \mathbf{M}^{-1}\mathbf{N}$ for a singular system has as many eigenvectors with modulus 1 as the dimension of the nullspace, and the method will converge if
  - no eigenvalues are bigger than one, and
  - all eigenvalues with modulus 1 are real,

  i.e. no complex eigenvalue of length 1 must exist.

It have been experimentally verified that all matrices of the form in Equation (5.36b) satisfy the latter. It can most likely be proven always to be the case, but no effort has been put into that here. It can be motivated by the fact that the system in Equation (5.36b) for each block is an approximation of a real problem with Neumann boundary conditions, and since nature always provides a solution to a real problem, the system should also have a solution.

The Dirichlet and Neumann steps presented do not strictly implement the two steps in the DN method as formulated in the beginning of this chapter, for two reasons: First of all, the DN method uses a global weighting for both blocks, e.g. for the Dirichlet step,

$$\begin{aligned} \mathbf{x}_s^{k+1} &= \alpha\mathbf{y}_b^k + (1-\alpha)\mathbf{x}_s^k, \\ \mathbf{y}_s^{k+1} &= (1-\alpha)\mathbf{x}_b^k + \alpha\mathbf{y}_s^k. \end{aligned} \tag{5.37a}$$

while the Dirichlet and Neumann steps presented here uses a local weighting, which is not alike for both blocks, compare above equation with Equation (5.34a). To implement a global weighting instead, the system matrix is handed slightly different. Only the Dirichlet step will be presented here,

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & \mathbf{C}_x & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 & \mathbf{0} & \mathbf{C}_y \\ \mathbf{0} & \alpha\tilde{\mathbf{I}}_y & -\alpha\mathbf{I} & \mathbf{0} \\ (1-\alpha)\tilde{\mathbf{I}}_x & \mathbf{0} & \mathbf{0} & (-1+\alpha)\mathbf{I} \end{bmatrix}. \tag{5.37b}$$

The $\mathbf{M}$ is the same,

$$\mathbf{M} = \begin{bmatrix} \mathbf{B}_1 & & \mathbf{C}_x & \\ & \mathbf{B}_2 & & \mathbf{C}_y \\ & & -\mathbf{I} & \\ & & & -\mathbf{I} \end{bmatrix}, \tag{5.37c}$$

$$\mathbf{N} = \begin{bmatrix} & & & \\ & & \alpha\tilde{\mathbf{I}}_y & (1-\alpha)\mathbf{I} \\ (1-\alpha)\tilde{\mathbf{I}}_x & & & \alpha\mathbf{I} \end{bmatrix}. \tag{5.37d}$$

Again, compare with the splitting in equation (5.34c) and (5.34d). In the case of $\alpha = \beta = \frac{1}{2}$, the two formulations coincide.

Secondly, the two steps both implement a boundary condition, but not in a way that boundary conditions are usually implemented. The Dirichlet step uses the value of the neighbouring cell, and not on the interface where the boundary condition usually would be applied. We will motivate later that this is an approximation of the usual way to implementing boundary conditions, hence the name of the method.

5.4 The DN Method

This section will show how to implement boundary conditions on the interface, using shadow variables and applying only operations that do not change the original solution.

While in the previous section no changes have been made to the block diagonal system $\mathbf{B}$ of the expanded system (5.21), this section will change this.

Section 2.4 shows that adding or subtracting from the diagonal at interface cell equations is equivalent to some kind of a boundary condition on the interface. Both Dirichlet (1st order), Neumann (2nd order) and Robin (1st order) boundary conditions can be implemented this way.

Adding γ to all interface cell diagonal elements is done through the operation

$$\tilde{\mathbf{B}}_1 = \mathbf{B}_1 + \gamma \tilde{\mathbf{I}}_x^T \tilde{\mathbf{I}}_x. \tag{5.38a}$$

The operation adds γ times a shadow rows to specific rows in $\mathbf{B}_1$. Replacing γ by a matrix having $\gamma_1, \ldots, \gamma_n$ on the correct diagonal entries corresponds to using a different value of γ for each shadow row. The use of a γ matrix will be necessary in case of a nonorthogonal grid.

Equation (5.38a) adds only to the $\mathbf{B}$ part of Equation (5.21), the $\mathbf{C}$ part has to be updated similarly for this to be a valid row operation on the entire system. The complete procedure carried out by multiplying the system matrix in Equation (5.32) from the left with

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} & \gamma\tilde{\mathbf{I}}_x^T \\ \mathbf{0} & \mathbf{I} & \gamma\tilde{\mathbf{I}}_y^T & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{5.38b}$$

resulting in a new system matrix

$$\begin{bmatrix} \tilde{\mathbf{B}}_1 & \mathbf{0} & \mathbf{C}_x & -\gamma\tilde{\mathbf{I}}_x^T \\ \mathbf{0} & \tilde{\mathbf{B}}_2 & -\gamma\tilde{\mathbf{I}}_y^T & \mathbf{C}_y \\ \mathbf{0} & \tilde{\mathbf{I}}_y & -\mathbf{I} & \mathbf{0} \\ \tilde{\mathbf{I}}_x & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{bmatrix}. \tag{5.38c}$$

The natural splitting of the new system matrix (5.38c) will be:

$$\mathbf{M} = \begin{bmatrix} \tilde{\mathbf{B}}_1 & & \mathbf{C}_x & -\gamma\tilde{\mathbf{I}}_x^T \\ & \tilde{\mathbf{B}}_2 & -\gamma\tilde{\mathbf{I}}_y^T & \mathbf{C}_y \\ & & -\mathbf{I} & \\ & & & -\mathbf{I} \end{bmatrix} \tag{5.38d}$$

$$\mathbf{N} = \begin{bmatrix} & & & \\ & \tilde{\mathbf{I}}_y & & \\ \tilde{\mathbf{I}}_x & & & \end{bmatrix}. \tag{5.38e}$$

On a regular grid $\gamma = -1$ will implement a Dirichlet step while $\gamma = 1$ will implement a Neumann step.

The classical alternating Schwarz method with a half cell overlap is implemented using $\gamma = -1$. The DN method is implemented using $\gamma = -1$ at odd iterations and $\gamma = 1$ at even iterations. There are no parameters deciding a weighting of values from each block as in the section about the approximative version. A weighting might be added to the method by implementing the boundary condition not on the actual block interface but on an artificial boundary moved slightly toward one of the blocks.

The DN method in Example 5.1 shows 2 iteration convergence for a 1D problem, and [Hadj00] proofs that this is the case for a hypercube in any dimension decomposed in two equally big hypercubes. A somewhat similar proof based on system (5.38c) is presented in Example C.1, while a proof for any dimension is given in Example C.2. The proofs are a bit lengthy, which is the reason to their position in the appendix.

The proofs show that the combination of a Dirichlet step and Neumann step works well for two blocks.

Let us just for a moment consider the Dirichlet step and the Neumann step alone.

Example 5.6: Eigenvalues of Iteration Matrices Consider a 2D quadrilateral domain of size 2×1 having pure Dirichlet boundaries. The domain is discretized using 32×16 cells, and decomposed into two equally big blocks.

We wish to find the eigenvalues of the iteration matrix, since this will tell us about convergence for each step separately.

The matrices $\mathbf{M}$ and $\mathbf{N}$ are created and the iteration matrix $\mathbf{G} = \mathbf{M}^{-1}\mathbf{N}$ is explicitly formed for both steps, named $\mathbf{G}_D$ and $\mathbf{G}_N$ for the Dirichlet and Neumann step respectively.

Figure 5.5 plots eigenvalue distribution for $\mathbf{G}_D$, $\mathbf{G}_N$, and their product.

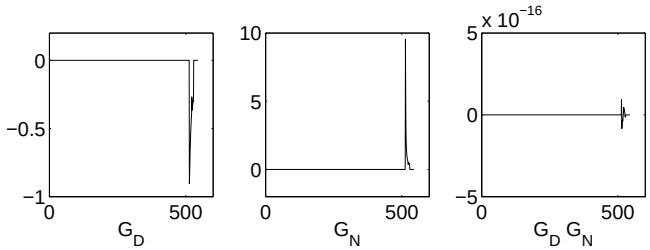


Figure 5.5: *Eigenvalues for DN iteration matrices*

The left plot shows the absolute value of all eigenvalues to be less than one: The Dirichlet step alone will converge, slowly though, since the absolute biggest eigenvalue is fairly close to one, having the largest at 0.905.

The middle plot shows many eigenvalues with absolute value above 1, the biggest of 9.5. The Neumann step alone do not converge, on contrary it diverges very rapidly. It would have been nice if both steps were guaranteed to converge, but that is not the case.

The right plot (notice the factor 10^{-16} on the vertical axis) shows the product of the two to be zero apart from rounding errors.

If refining the discretization, e.g. using 64×32 cells, the same picture will arise, differing only in scale: The Dirichlet step eigenvalues will be closer to one, the Neumann step eigenvalues will be scaled by about a factor two, but the product of the two is still zero.

End of Example 5.6.

The eigenvalues presented in Example 5.6 show that especially the

Neumann step alone is not stable when using a stationary iterative procedure.

5.5 A Generalized DN Method

The DN method works fine for a 2 block setup. However a 2×2 block system is somewhat more tricky to handle. As mentioned, NS3 does not include any coupling between blocks diagonally over a vertex point, and there would usually not be so on a regular grid anyway. Hence, using the names of blocks and interfaces in Figure 5.6, information from block A must pass either block B or C before block D get to know about it. The information is in a sense delayed by one iteration, and this delay may cause growing oscillations, as Example 5.7 shows.

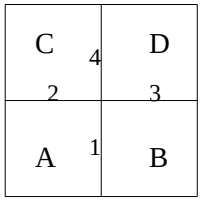


Figure 5.6: *Interface numbering*

Example 5.7: Divergence of DN Method Consider a 2×2 block setup as in Example A.1, with Dirichlet boundary data on all boundaries.

Applying the DN method shows to diverge. Figure 5.7 shows a plot of 6 consecutive preliminary solutions. The Neumann steps (even iterations) create oscillations while the Dirichlet steps try to even the oscillations out.

This is also the case for approximative DN method, unless under relaxation values of the parameters are chosen on the Neumann step, $\beta < 0.5$.

End of Example 5.7.

The Neumann step in a 2×2 block setup, or in general when an

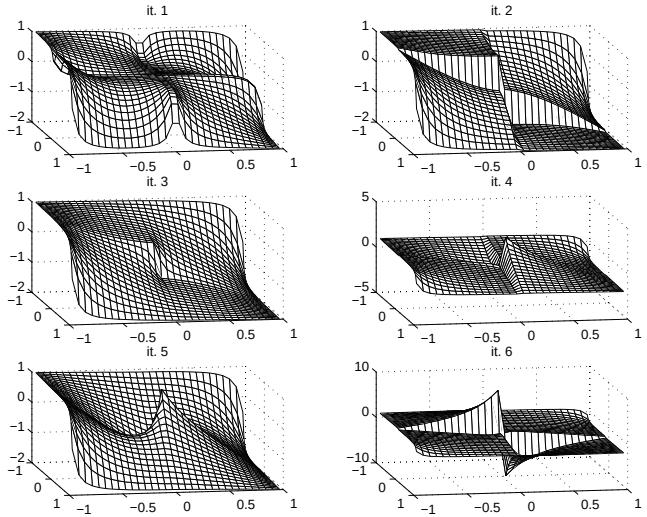


Figure 5.7: *Divergence for 2 step method*

interior vertex is present, is highly unstable. However the Dirichlet step is always stable. The idea is to mix the two steps to get sufficient stability and fast convergence. For that we need to generalize the DN method.

Consider a 2×2 block setup and number the interfaces as in Figure 5.6. The generalized DN method will allow different data to be exchanged over different interfaces in the same step. Introduce the following notation, $\mathbf{O} = [\ 1 \quad 0 \quad 0 \quad 1\ ]$: A row with a zero at the ith position means that Dirichlet data is exchanged over ith interface, and similar a one at jth position indicates the exchange of Neumann data over interface j. So the row vector $\mathbf{O}$ implies exchange of Dirichlet data over second and third interface and Neumann data over first and fourth. Furthermore several rows of this type may be collected in a matrix, telling for each row how data is exchanged in each step. Such a matrix

will be called a direction matrix.

The DN method as presented so far will have the direction matrix

$$\mathbf{O}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \tag{5.39a}$$

since only Dirichlet data is exchanged in the first step and only Neumann in the second.

The following will present the direction matrices, that will used in numerical experiments later.

The two block case shows that it is possible to control oscillation over one Neumann interface. Therefor we want in each step to either;

- let the horizontal interfaces 2 and 3 exchange Dirichlet data and the vertical interfaces 1 and 4 exchange Neumann data, or
- let the horizontal interfaces 2 and 3 exchange Neumann data and the vertical interfaces 1 and 4 exchange Dirichlet data.

Thereby only one long interface at a time exchanges Neumann data. This can be implemented by using the direction matrix

$$\mathbf{O}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}. \tag{5.39b}$$

We will also consider the following direction matrix

$$\mathbf{O}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{5.39c}$$

The motivation for $\mathbf{O}_3$ is as follows: In the case of a regular grid, if a two block system can obtain a solution in 2 iterations, then a 2×2 block system may obtain a solution in 4 iterations by:

- Set Dirichlet BC at the horizontal interfaces 2 and 3 and use the 2 iteration DN method for a two block setup to produce an exact solution for both the top and bottom pairs of blocks.

- Set Neumann BC at the horizontal interfaces 2 and 3 and use again the 2 iteration DN method for a two block setup to produce an exact solution for both the top and bottom pairs of blocks.

In total four iterations when having a regular grid.

To complete the investigation for the 2×2 block setup, the last case that will be considered is the direction matrix

$$\mathbf{O}_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{5.39d}$$

These are the direction matrices that will be tested in the 2×2 block case. As the result section shows later, this can produce a solution quite effectively.

However, for more general setup it has not been possible to produce iteration matrices that behave significantly better than the classical alternating Schwarz method, and is independent of the mesh spacing.

This way of mixing the exchange of Dirichlet and Neumann data can be applied to as well the approximative DN method, as to the DN method it self. Therefor in the rest of the thesis, any DN method must given with a direction matrix in order to make the method well defined.

CHAPTER 6

Schur complement methods

Theory and notation is based on the presentation in [Smith96], which itself is based on a finite element approach. This implies that the interface between two blocks is represented by a number of nodes in the grid as depicted in Figure 6.1. This approach will be used in the

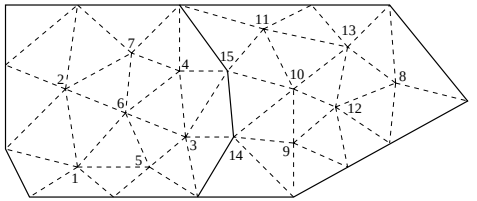


Figure 6.1: *Finite element grid*

presentation of the Schur complement methods.

Consider the symmetric positive definite system

$$\mathbf{A}\mathbf{u} = \mathbf{f}. \tag{6.1}$$

A Poisson problem discretized by a finite element method produces a symmetric positive definite system, since the Laplace operator behaves equally in any direction and any region.[1] The system may be only semidefinite, e.g. for a Poisson problem with pure Neumann BC.

Split up the domain Ω in several non overlapping blocks Ω_i. Denote nodes on the interface as $\mathbf{u}_B$ and nodes in the interior as $\mathbf{u}_I$. This partitioning can also be applied on the unknowns for each block $\mathbf{u}^{(i)} = [\mathbf{u}_I^{(i)}\ \mathbf{u}_B^{(i)}]^T$, where each element of $\mathbf{u}_B$ occur as unknown in at least two blocks. For example will the nodes numbered 14 and 15 in Figure 6.1 be unknowns to both blocks, and hence a part of as well $\mathbf{u}^{(1)}$ as $\mathbf{u}^{(2)}$.

Let $\mathbf{R}_i$ pick out the nodes relating to block i, $\mathbf{u}^{(i)} = \mathbf{R}_i\mathbf{u}$, while $\tilde{\mathbf{R}}_i$ pick out the nodes on $\mathbf{u}_B$ relating to block i, $\mathbf{u}_B^{(i)} = \tilde{\mathbf{R}}_i\mathbf{u}_B$. The tilde is used to indicate that $\tilde{\mathbf{R}}_i$ works on the interface variables.

In a two block the system (6.1) may be written on the form

$$\begin{bmatrix} \mathbf{A}_{II}^{(1)} & \mathbf{0} & \mathbf{A}_{IB}^{(1)} \\ \mathbf{0} & \mathbf{A}_{II}^{(2)} & \mathbf{A}_{IB}^{(2)} \\ \mathbf{A}_{BI}^{(1)} & \mathbf{A}_{BI}^{(2)} & \mathbf{A}_{BB}^{(1)} + \mathbf{A}_{BB}^{(2)} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(1)} \\ \mathbf{u}_I^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(1)} \\ \mathbf{f}_I^{(2)} \\ \mathbf{f}_B \end{bmatrix}. \tag{6.2}$$

For each block a local system matrix can be obtained of the form

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{A}_{II}^{(i)} & \mathbf{A}_{IB}^{(i)} \\ \mathbf{A}_{BI}^{(i)} & \mathbf{A}_{BB}^{(i)} \end{bmatrix}. \tag{6.3}$$

A direct factorization of the two block system (6.2) would start by eliminating the $\mathbf{A}_{BI}^{(i)}$'s,

$$\begin{bmatrix} \mathbf{A}_{II}^{(1)} & \mathbf{0} & \mathbf{A}_{IB}^{(1)} \\ \mathbf{0} & \mathbf{A}_{II}^{(2)} & \mathbf{A}_{IB}^{(2)} \\ \mathbf{0} & \mathbf{0} & \mathbf{S}^{(1)} + \mathbf{S}^{(2)} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(1)} \\ \mathbf{u}_I^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(1)} \\ \mathbf{f}_I^{(2)} \\ \mathbf{g} \end{bmatrix}, \tag{6.4}$$

[1] For references, see in [Bren94] or
`http://math.nist.gov/mcsd/savg/tutorial/ansys/FEM/`

where the Schur complement matrices are defined as

$$\mathbf{S}^{(i)} = \mathbf{A}_{BB}^{(i)} - \mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{A}_{IB}^{(i)}, \tag{6.5}$$

and the right hand side

$$\mathbf{g} = \mathbf{f}_B - \sum_{i=1}^{2} \mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{f}_I^{(i)}. \tag{6.6}$$

If solving first the Schur complement for $\mathbf{u}_B$

$$(\mathbf{S}^{(1)} + \mathbf{S}^{(2)})\mathbf{u}_B = \mathbf{g}, \tag{6.7}$$

what is left, is a back solve for the interior nodes in each block in Equation (6.4).

This we will generalize to an arbitrary number of blocks, in which case the system matrix will have the form

$$\begin{bmatrix} \mathbf{A}_{II}^{(1)} & & & \mathbf{A}_{IB}^{(1)}\tilde{\mathbf{R}}_1 \\ & \ddots & & \vdots \\ & & \mathbf{A}_{II}^{(k)} & \mathbf{A}_{IB}^{(k)}\tilde{\mathbf{R}}_k \\ \tilde{\mathbf{R}}_1^T\mathbf{A}_{BI}^{(1)} & \cdots & \tilde{\mathbf{R}}_k^T\mathbf{A}_{BI}^{(k)} & \sum_{i=1}^{k}\tilde{\mathbf{R}}_i^T\mathbf{A}_{BB}^{(i)}\tilde{\mathbf{R}}_i \end{bmatrix}. \tag{6.8}$$

The $\tilde{\mathbf{R}}_i$ are necessary since in general every element of $\mathbf{u}_B$ no longer relates to every block.

A direct factorization will again start by eliminating the blocks below the diagonal, the $\tilde{\mathbf{R}}_i^T\mathbf{A}_{BI}^i$'s, and as in Equation (6.4) produce the Schur complement system

$$\left(\sum_{i=1}^{k}\tilde{\mathbf{R}}_i^T\mathbf{S}^{(i)}\tilde{\mathbf{R}}_i\right)\mathbf{u}_B = \mathbf{f}_B - \sum_{i=1}^{k}\tilde{\mathbf{R}}_i^T\mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{f}_I^{(i)}, \tag{6.9}$$

or in short just

$$\mathbf{S}\mathbf{u}_B = g. \tag{6.10}$$

Having solved the Schur complement system and obtained the values on the interface $\mathbf{u}_B$, what is left, is for each block to back solve for the interior variables,

$$\mathbf{A}_{II}^{(i)}\mathbf{u}_I^{(i)} = \mathbf{f}_I^{(i)} - \mathbf{A}_{IB}^{(i)}\tilde{\mathbf{R}}_i\mathbf{u}_B, \quad \forall\, i. \tag{6.11}$$

The explicit formation and inversion of the Schur complement is expensive, since each of the Schur complements $\mathbf{S}^{(i)}$ in general are dense, though of much smaller size than the original $\mathbf{A}$. For large systems an iterative procedure to solve the Schur complement should be applied.

The next section introduces the Neumann-Neumann method to solve the Schur complement system. This and other iterative methods for solving the Schur complement system can be found in [Smith96].

The method is presented in the context of Krylov subspace methods: According to Appendix B, to apply a Krylov subspace method to the system (6.10), it is necessary to provide routines to:

- Compute the matrix vector product $\mathbf{w} = \mathbf{S}\mathbf{v}$.
- Compute an approximative solution $\mathbf{w} = \mathbf{B}\mathbf{v}$, where the preconditioner $\mathbf{B}$ approximates $\mathbf{S}^{-1}$.

6.1 Neumann-Neumann Method

Consider first the ability to apply $\mathbf{S}$ to a vector. By using the definition of the Schur complements (6.5) and (6.9),

$$\mathbf{S}\mathbf{v} = \left(\sum_{i=1}^{k} \tilde{\mathbf{R}}_i^T \left(\mathbf{A}_{BB}^{(i)} - \mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{A}_{IB}^{(i)}\right)\tilde{\mathbf{R}}_i\right)\mathbf{v} \tag{6.12}$$

produces this without having to form any of the $\mathbf{S}^{(i)}$'s explicitly.

Secondly, a Krylov method needs a preconditioner. The Neumann-Neumann method uses a preconditioner $\mathbf{B}$, which in a two block case is

$$\mathbf{S}^{-1} = (\mathbf{S}^{(1)} + \mathbf{S}^{(2)})^{-1} \approx ({\mathbf{S}^{(1)}}^{-1} + {\mathbf{S}^{(2)}}^{-1}) = \mathbf{B}. \tag{6.13}$$

The motivation for using this preconditioner is simple: If the two blocks are mirrors of each others, then $\frac{1}{2}\mathbf{S} = \mathbf{S}^{(1)} = \mathbf{S}^{(2)}$, hence $\mathbf{B} = \mathbf{S}^{-1}$ and $\mathbf{B}$ will be an optimal preconditioner for $\mathbf{S}$. If the two blocks are not mirrors, the preconditioner does no longer produce the exact solution, but it will usually still be a good preconditioner.

In the general case of more than two blocks,

$$\mathbf{S}^{-1} \approx \mathbf{B} = \mathbf{D}\left(\sum_{i=1}^{k} \tilde{\mathbf{R}}_i^T {\mathbf{S}^{(i)}}^{-1}\tilde{\mathbf{R}}_i\right)\mathbf{D}, \tag{6.14}$$

where $\mathbf{D}$ is a diagonal scaling matrix, $(\mathbf{D})_{i,i}^{-1}$ is the number of blocks that share node $(\mathbf{u}_B)_i$. In general, one may use

$$\mathbf{B} = \left(\sum_{i=1}^{k} \mathbf{D}_i\tilde{\mathbf{R}}_i^T {\mathbf{S}^{(i)}}^{-1}\tilde{\mathbf{R}}_i\mathbf{D}_i\right), \tag{6.15}$$

to make a weighting between blocks possible. Best convergence seems to be obtained when $\sum_{i=1}^{k}\mathbf{D}_i = \mathbf{I}$, according to [Smith96].

The action of the inverse of $\mathbf{S}^{(i)}$ on a vector can be calculated without explicitly forming each $\mathbf{S}^{(i)}$. Consider the factorization

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{II}^{(i)} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{I} & {\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{A}_{IB}^{(i)} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{6.16}$$

one can calculate the inverse

$${\mathbf{A}^{(i)}}^{-1} = \begin{bmatrix} \mathbf{I} & -{\mathbf{A}_{II}^{(i)}}^{-1}\mathbf{A}_{IB}^{(i)} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} {\mathbf{A}_{II}^{(i)}}^{-1} & \mathbf{0} \\ \mathbf{0} & {\mathbf{S}^{(i)}}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{A}_{BI}^{(i)}{\mathbf{A}_{II}^{(i)}}^{-1} & \mathbf{I} \end{bmatrix} \tag{6.17}$$

Putting this together produces in short form

$${\mathbf{A}^{(i)}}^{-1} = \begin{bmatrix} \mathbf{xx} & \mathbf{xx} \\ \mathbf{xx} & {\mathbf{S}^{(i)}}^{-1} \end{bmatrix}, \tag{6.18}$$

hence

$$\mathbf{S}^{(i)^{-1}}\mathbf{v} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \end{bmatrix} \mathbf{A}^{(i)^{-1}} \begin{bmatrix} \mathbf{0} \\ \mathbf{v} \end{bmatrix}, \tag{6.19}$$

which reads: Expand $\mathbf{v}$ by padding with zeros for interior unknowns, solve the local block problem for all unknowns, and take out only the solution at the interface.

Equations (6.12) and (6.19) provide what is sufficient to apply a Krylov subspace method to the Schur system (6.10).

To summarize the Neumann-Neumann method: The application of $\mathbf{S}^{(i)}$ on a vector (6.12) involves a solve of the system $\mathbf{A}_{II}^{(i)}$ for the interior variables $\mathbf{u}_I^{(i)}$ using Dirichlet boundary conditions on $\mathbf{u}_B^{(i)}$. The application of $\mathbf{S}^{(i)^{-1}}$ on a vector (6.19) involves a solve of the system $\mathbf{A}^{(i)}$ for all the variables $\mathbf{u}^{(i)}$ using a Neumann boundary condition on $\mathbf{u}_B^{(i)}$. Since the preconditioner is based on solutions to Neumann problems on all blocks, the method is called the Neumann-Neumann method.

If a block Ω_i is an interior block or has Neumann conditions on the original domain boundaries, then $\mathbf{S}^{(i)}$ and also $\mathbf{A}^{(i)}$ are singular. A pseudo inverse or some kind of regularization must be applied.

For a smaller number of blocks, the Neumann-Neumann preconditioner used in a Krylov subspace method shows quite good performance.

However, for large number of blocks, convergence decrease rapidly as $\frac{1}{k}$, k being the number of blocks, see [Smith96]. The following section on balancing domain decomposition will show how to add a coarse grid problem to improve convergence.

6.2 Balancing Domain Decomposition Method

The Balancing domain decomposition (BDD) method[2] was originally introduced in [Mand93a]. The method adds a coarse grid problem, using the null space of $\mathbf{S}^{(i)}$. Let $\mathbf{Z}_i$ span the null space of $\mathbf{S}^{(i)}$, $N(\mathbf{S}^{(i)}) \subset R(\mathbf{Z}_i)$. Since the null space of the Poisson problem is known, $N(\mathbf{S}^{(i)}) = R(\mathbf{Z}_i)$ will be used. Then $\mathbf{g}$ is said to be balanced if

$$\mathbf{Z}_i^T \tilde{\mathbf{R}}_i \mathbf{D}\mathbf{g} = 0, \quad \forall\, i. \tag{6.20}$$

That is, the restriction (and weighting) of $\mathbf{g}$ to each block must be orthogonal to all columns of $\mathbf{Z}_i$, i.e. be orthogonal to the null space of $\mathbf{S}^{(i)}$. The process of balancing a vector $\mathbf{g}$ is to find a vector $\mathbf{w}$ such that $\mathbf{s} = \mathbf{g} - \mathbf{S}\mathbf{w}$ is balanced.

The BDD preconditioner computing $\mathbf{z} = \mathbf{B}\mathbf{g}$ is based on the following steps: Balance the right hand side $\mathbf{g}$ by solving for λ_j;

$$\mathbf{Z}_i^T \tilde{\mathbf{R}}_i \mathbf{D} \left(\mathbf{g} - \mathbf{S} \sum_{j=1}^{k} \mathbf{D}\tilde{\mathbf{R}}_j^T \mathbf{Z}_j \lambda_j \right) = 0, \quad i = 1, \ldots, k, \tag{6.21a}$$

and set

$$\mathbf{s} = \mathbf{g} - \mathbf{S} \sum_{j=1}^{k} \mathbf{D}\tilde{\mathbf{R}}_j^T \mathbf{Z}_j \lambda_j, \quad \mathbf{s}^{(i)} = \tilde{\mathbf{R}}_i \mathbf{D}\mathbf{s}, \tag{6.21b}$$

which is now balanced. The balancing of $\mathbf{g}$ will be called the prebalancing step. Find any solution to the local problems

$$\mathbf{S}^{(i)} \mathbf{u}_B^{(i)} = \mathbf{s}^{(i)}, \quad \forall\, i. \tag{6.21c}$$

Each of these are consistent due to the balancing step. Average the solution

$$\mathbf{u}_B = \sum_{i=1}^{k} \mathbf{D}\tilde{\mathbf{R}}_i \mathbf{u}_B^{(i)}. \tag{6.21d}$$

[2] Sometimes called Balancing Neumann-Neumann

Since any solution to (6.21c) is allowed, this is in general not a feasible solution. So balance the residual by solving for μ_j;

$$\mathbf{Z}_i^T \tilde{\mathbf{R}}_i \mathbf{D} \left(\mathbf{g} - \mathbf{S} \left(\mathbf{u}_B + \sum_{j=1}^{k} \mathbf{D} \tilde{\mathbf{R}}_j^T \mathbf{Z}_j \mu_j \right) \right) = 0, \quad i = 1, \ldots, k, \tag{6.21e}$$

and finally update the result

$$\mathbf{z} = \mathbf{u}_B + \sum_{i=1}^{k} \mathbf{D} \tilde{\mathbf{R}}_i \mathbf{Z}_i \mu_i. \tag{6.21f}$$

The last balancing procedure will be called the post-balancing step.

Usually the input $\mathbf{g}$ will be the residual for the preliminary solution at hand, while the output $\mathbf{z}$ is an estimate of the error of the preliminary solution.

Let us take a look at the two balancing steps. Note that for each block i, $\mathbf{u}_B$ is balanced when

$$\mathbf{0} = \mathbf{Z}_i^T \tilde{\mathbf{R}}_1 \mathbf{D} \mathbf{u}_B = (\mathbf{D} \tilde{\mathbf{R}}_1^T \mathbf{Z}_i)^T \mathbf{u}_B, \quad i = 1, \ldots, k. \tag{6.22}$$

This indicate that instead of restricting each vector to a block, the same result is achieved by expanding $\mathbf{Z}_i$ to match the size of $\mathbf{u}_B$, padded appropriately with zeros, and scaled with $\mathbf{D}$. Collect all the null spaces $\mathbf{Z}_i$ by defining

$$\mathbf{Q} = \begin{bmatrix} \mathbf{D} \tilde{\mathbf{R}}_1^T \mathbf{Z}_1 & \cdots & \mathbf{D} \tilde{\mathbf{R}}_k^T \mathbf{Z}_k \end{bmatrix}, \tag{6.23}$$

in which case Equation (6.22) becomes $\mathbf{Q}^T \mathbf{u}_B = \mathbf{0}$. The pre-balancing step (6.21a) can be rewritten as

$$\mathbf{Q}^T (\mathbf{g} - \mathbf{S}\mathbf{Q}\lambda) = 0, \tag{6.24}$$

or

$$(\mathbf{Q}^T \mathbf{S} \mathbf{Q}) \lambda = \mathbf{Q}^T \mathbf{g}. \tag{6.25}$$

Similar for the post-balancing step (6.21e)

$$(\mathbf{Q}^T \mathbf{S} \mathbf{Q}) \mu = \mathbf{Q}^T (\mathbf{g} - \mathbf{S}\mathbf{u}_B). \tag{6.26}$$

The process of balancing a vector $\mathbf{v}$ (6.25) corresponds to solving a problem with the coarse grid operator $\mathbf{A}_0 = \mathbf{Q}^T \mathbf{S} \mathbf{Q}$ using the right hand side $\mathbf{Q}^T \mathbf{v}$. The post-balancing step (6.21e), or (6.26), can be interpreted as a coarse level step of a two level algorithm of Galerkin type: The residual on the fine grid is restricted to a coarse grid, where a coarse grid correction is computed, and the solution is updated with the coarse grid correction in Equation (6.21f).

If for a block i, $\mathbf{S}^{(i)}$ is nonsingular, then $\mathbf{Z}_i$, λ_i and μ_i are void and the ith part of the sum in the balancing equations (6.21a) and (6.21e) is left out. This especially implies that if all blocks are nonsingular, then BDD is the same as the Neumann-Neumann method, and there is no coarse grid correction to improve convergence.

In theory, if using a Krylov type iterative method, the initial residual should be balanced as in the post-balancing (6.21e) and (6.21f) using the new $\mathbf{z}$ as start guess for $\mathbf{u}_B$. Then the pre-balancing step (6.21a) can be omitted, since the residual produced by a Krylov type iterative method automatically is balanced. This can be shown by induction: Assume the residual is balanced at step i, meaning

$$\mathbf{Q}^T \mathbf{r}_i = \mathbf{0}. \tag{6.27}$$

The result from the preconditioner, which is the error estimate for the ith step, is balanced in the post-balancing step (6.21e) such that

$$\mathbf{Q}^T (\mathbf{r}_i - \mathbf{S}\mathbf{e}_i) = \mathbf{0}, \tag{6.28}$$

and the solution and residual are updated according to

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \mathbf{e}_i, \tag{6.29}$$

$$\mathbf{r}_{i+1} = \mathbf{g} - \mathbf{S}\mathbf{u}_{i+1}. \tag{6.30}$$

The new residual is balanced since

$$\begin{aligned}\mathbf{Q}^T\mathbf{r}_{i+1} &= \mathbf{Q}^T(\mathbf{g}-\mathbf{S}\mathbf{u}_{i+1}) = \mathbf{Q}^T(\mathbf{g}-\mathbf{S}\mathbf{u}_i-\mathbf{S}\mathbf{e}_i)\\ &= \mathbf{Q}^T(\mathbf{r}_i-\mathbf{S}\mathbf{e}_i)\end{aligned}, \tag{6.31}$$

which is zero due to the post-balancing (6.28).

In practice however, due to rounding errors, omitting the pre balancing step might prevent the method in achieving very accurate solutions, so care must be taken.[3]

According to [Smith96] it is possible for certain model problems to show that the condition number of the BDD preconditioned system grows like $O((1+\log(\frac{H}{h}))^2)$, h being the characteristic grid size and H the characteristic block size, in both 2 and 3 dimensions. Secondly by modifying the scaling $\mathbf{D}_i$, the condition number can be bound independently of jumps in the coefficients of the PDE between blocks, see also [Mand93a]. Furthermore, a strength of as well the Neumann-Neumann method as the BDD method is that knowledge of which nodes are on faces, edges or vertices, is not needed, as is not the case for many other Schur complement preconditioners.

The BDD method as presented here relies on the original system to be symmetric. If $\mathbf{A}$ is symmetric, then also all $\mathbf{A}^{(i)}$ are symmetric, and hence also all $\mathbf{S}^{(i)}$. The latter can be seen from the definition of the Schur complements (6.5).

The symmetry is required of the following reason [Bark92]: For a singular system to be consistent it is necessary for the right hand side $\mathbf{f}$ to belong to the range of $\mathbf{A}$, $\mathbf{f}\in R(\mathbf{A})$. Also

$$R(\mathbf{A}) = N\left(\mathbf{A}^T\right)^{\perp}. \tag{6.32}$$

If $\mathbf{A}$ is symmetric, then $R(\mathbf{A}) = N(\mathbf{A})^{\perp}$. To make a right hand side of a symmetric system consistent, it is therefor sufficient to remove any part of the right hand side that lies in the null space.

[3] For e.g. Example A.7 (in a 7 block setup) it was only possible for GMRES to reduce the residual by about a factor 10^{-6}

In the case where $\mathbf{A}$ is not symmetric, then instead any part of the right hand side that lies in the null space of the transposed system must be removed. Hence, if $\mathbf{A}$ is not symmetric then all the $\mathbf{S}^{(i)}$'s are not symmetric either, and to apply the BDD method it is in general necessary to know the null space of the transpose of $\mathbf{S}^{(i)}$ and instead create $\mathbf{Z}_i$ such that

$$N\left(\mathbf{S}^{(i)\,T}\right) \subset R(\mathbf{Z}_i). \tag{6.33}$$

6.3 BDD Method Using Shadow Variables

As well the Neumann-Neumann method as the BDD method is based on a finite element discretization, where the interfaces are represented by a number of nodes. In a finite volume discretization, no nodes can in the same manner represent the interfaces. The purpose of this section is to adapt the finite volume discretization to the BDD method.

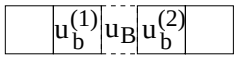


Figure 6.2: *Shadow layer using average*

A First Attempt. Let us make a set of artificial cells on the interfaces, their value defined by the average of cells on each side. These artificial cells $\mathbf{u}_B$ will be put into the shadow layer at each block, $\mathbf{u}_s^1$ and $\mathbf{u}_s^2$,

$$\mathbf{u}_B = \mathbf{u}_s^{(1)} = \mathbf{u}_s^{(2)} = \frac{1}{2}(\mathbf{u}_b^{(1)} + \mathbf{u}_b^{(2)}). \tag{6.34}$$

Add the new unknowns and equations to the system

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{0} \\ \tilde{\mathbf{I}}_1 & \tilde{\mathbf{I}}_2 & -2\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \\ \mathbf{0} \end{bmatrix}, \tag{6.35}$$

where $\tilde{\mathbf{I}}_i$ pick out the $\mathbf{u}_b^{(i)}$ of $\mathbf{u}^{(i)}$, $\mathbf{u}_b^{(i)} = \tilde{\mathbf{I}}_i \mathbf{u}^{(i)}$. This system will typically have a structure similar to Figure 6.3, i.e. for every nonzero column of $\mathbf{A}_{21}$ there will be an entry in $\tilde{\mathbf{I}}_1$. By the use of the shadow rows

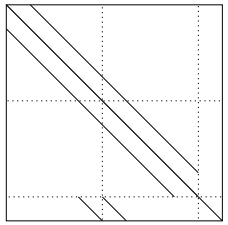


Figure 6.3: *Structure of matrix. Dotted lines indicate interfaces.*

it is possible to eliminate the non block diagonal parts $\mathbf{A}_{21}$ and $\mathbf{A}_{12}$,

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & 2\mathbf{C}_1 \\ \mathbf{0} & \mathbf{B}_2 & 2\mathbf{C}_2 \\ \tilde{\mathbf{I}}_1 & \tilde{\mathbf{I}}_2 & -2\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \\ \mathbf{0} \end{bmatrix}, \tag{6.36}$$

where $\mathbf{C}_1$ as previously is defined as $\mathbf{A}_{12}$ with all zero columns removed, and $\mathbf{B}_i = \mathbf{A}_{ii} - \mathbf{C}_i \tilde{\mathbf{I}}_i$.

Each block will get a system matrix of the form (6.3) as

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{B}_i & 2\mathbf{C}_i \\ \tilde{\mathbf{I}}_i & -\mathbf{I} \end{bmatrix}. \tag{6.37}$$

Now the system is set up exactly as is the case for the BDD method, though the finite volume discretization does not in general make a symmetric system. And anyway, if a regular grid is used and symmetry is obtained for the original matrix $\mathbf{A}$, then $\mathbf{C}_i = \tilde{\mathbf{I}}_i^T$, and due to the coefficient 2 in front of $\mathbf{C}_i$, $\mathbf{A}^{(i)}$ will never be symmetric.

Assume that block i is singular. The null space of $\mathbf{A}^{(i)}$ and hence also $\mathbf{S}^{(i)}$ is known, and as is always the case for the Poisson problem, it is the constant vector.

However, applying the BDD method to the above system turns out in general not to converge. The reason is that all of the singular cases

in Equation (6.21c) do no longer get a right hand side which have a solution, the local systems are not consistent. This is somewhat expected due to the lack of symmetry.

Let us take a step back, and look at some properties of the Poisson problem and a finite volume discretization.

It can be shown that a Poisson problem with pure Neumann boundary conditions is consistent, if the right hand side forcing, including the boundary conditions on the original boundary, sum up to zero. Said in another way: Sources, sinks and in- and outflow through boundaries must balance. Look at the Poisson problem, integrate over the domain

$$\int_\Omega f \, \mathrm{d}\Omega = \int_\Omega \nabla^2 u \, \mathrm{d}\Omega, \tag{6.38}$$

and apply the special case of Green's first identity (D.3) on the right hand side to get

$$\int_\Omega f \, \mathrm{d}\Omega - \int_{\partial\Omega} \frac{\partial u}{\partial n} \, \mathrm{d}S = 0, \tag{6.39}$$

which state exactly the condition to be satisfied. If the system is consistent, then the solution is unique up to a constant. This consistency property should be inherited by any discrete Poisson operator.

We have verified experimentally that a finite volume discretization of a pure Neumann problem, even though it does not produce a symmetric system, has the property that

$$R(\mathbf{A}) = N(\mathbf{A})^\perp, \tag{6.40}$$

in which case $N(\mathbf{A}) = N(\mathbf{A}^T)$. The property have been verified on examples from Appendix A having pure Neumann BC. It can probably be proven always to be the case, but no effort has been put into that here. An argument is that due to the conservative approach in the finite volume discretization, flux through a face between cell i and j, F_{ij}, is treated symmetrically in the sense that $F_{ij} = -F_{ji}$. The system $\mathbf{A}$ is build from these fluxes, hence some of the properties of this

symmetry might be inherited by $\mathbf{A}$. I have heard a saying stating that conservatism is a weak form of symmetry, a poor mans symmetry. These tests rectify the saying.

A Second Attempt. Let us return to the breakdown of the BDD method. A deeper analysis shows that the orthogonality condition (6.40) no longer holds for all $\mathbf{S}^{(i)}$. An example is presented in Example 6.1 in the end of this section. To make the BDD method work, the constant vector so far used in $\mathbf{Z}_i$ should be replaced by the null space of the transpose of $\mathbf{S}^{(i)}$, which is not known.

Remember that it is possible to make a finite volume discretization on the entire domain, which will produce an operator that fulfills the orthogonality condition (6.40). Thus, it should be possible also for the individual blocks by imposing an appropriate finite volume discretization on each block.

Therefor, let us consider the shadow cells as part of a usual conservative finite volume discretization by inserting the shadow cells between two blocks as in Figure 6.2, which is infinitesimally thin: The area of the faces facing other shadow cells, in Figure 6.2 the north and south cells, are infinitesimally and contributions from here to the operator are zero. Flux through and area of face east and west are alike and equals the flux over the interface. Introducing these new shadow variables and equations into the system will produce something very alike the former,

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{0} \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_1 & \hat{\mathbf{D}}\tilde{\mathbf{I}}_2 & -2\hat{\mathbf{D}}\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \\ \mathbf{0} \end{bmatrix}, \tag{6.41}$$

where $\hat{\mathbf{D}}$ is a diagonal scaling matrix, scaling the rows as to implement exactly an infinitesimally thin cell, $(\hat{\mathbf{D}})_{jj}$ matches the flux into $(\mathbf{u}_b^{(1)})_j$ through the interface. The structure is the same as in figure (6.3), and

eliminating the non block diagonal part is no different than before;

$$\begin{bmatrix} \mathbf{B}_1 & \mathbf{0} & 2\mathbf{C}_1 \\ \mathbf{0} & \mathbf{B}_2 & 2\mathbf{C}_2 \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_1 & \hat{\mathbf{D}}\tilde{\mathbf{I}}_2 & -2\hat{\mathbf{D}}\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{(1)} \\ \mathbf{u}^{(2)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \\ \mathbf{0} \end{bmatrix}. \tag{6.42}$$

This system shows experimentally to have property (6.40) as the usual finite volume discretization have. The block matrix for each block

$$\mathbf{A}^{(i)} = \begin{bmatrix} \mathbf{B}_i & 2\mathbf{C}_i \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_i & -\hat{\mathbf{D}}\mathbf{I} \end{bmatrix}. \tag{6.43}$$

This turns out to produce Schur complements with the necessary orthogonality property $R(\mathbf{S}^{(i)}) = N(\mathbf{S}^{(i)})^{\perp}$, consult e.g. again Example 6.1. Note that the condition has only been shown to hold experimentally.

The BDD method applied to the system with the new artificial interface cells works nicely.

A Correction to $\mathbf{A}^{(i)}$. An experimental analysis as Example 6.1 of $\mathbf{A}^{(i)}$ from the second attempt shows that $R(\mathbf{A}^{(i)}) \neq N(\mathbf{A}^{(i)})^{\perp}$, indicating problems. But even though $\mathbf{A}^{(i)}$ in equation (6.19) is used to solve the Schur system, $\mathbf{S}^{(i)}$, then the balancing of the right hand side $\mathbf{s}_i$ is sufficient to produce a vector $[\mathbf{0}\ \mathbf{s}_i]^T$ as right hand side to $\mathbf{A}^{(i)}$ which is consistent.

The "defect" of $\mathbf{A}^{(i)}$ can be fixed. The system (6.43) does not strictly correct implement a FV discretization on a single block: Following the setup in Figure 6.2, the flux through the east face of $\mathbf{u}_b^{(1)}$ is calculated using the gradient $\mathbf{u}_B - \mathbf{u}_b^{(1)}$, which 2nd order accurate in between the two cell centers. The face is unfortunately located at the center of $\mathbf{u}_B$.

Locally for each block, the center of $\mathbf{u}_B$ can be moved artificially away from the interface by multiplying $\mathbf{A}^{(i)}$ from the left with

$$\begin{bmatrix} \mathbf{I} & \mathbf{C}_i\hat{\mathbf{D}}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}, \tag{6.44}$$

producing a corrected $\tilde{\mathbf{A}}^{(i)}$

$$\tilde{\mathbf{A}}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{C}_i\hat{\mathbf{D}}^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{B}_i & 2\mathbf{C}_i \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_i & -\hat{\mathbf{D}}\mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{ii} & \mathbf{C}_i \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_i & -\hat{\mathbf{D}}\mathbf{I} \end{bmatrix}. \quad (6.45)$$

This will implement a boundary condition not on the interface, but somewhere else. It also changes the way to produce the solution of the Schur complement systems similarly,

$$\mathbf{S}^{(i)^{-1}}\mathbf{v} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{ii} & \mathbf{C}_i \\ \hat{\mathbf{D}}\tilde{\mathbf{I}}_i & -\hat{\mathbf{D}}\mathbf{I} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{C}_i\hat{\mathbf{D}}^{-1}\mathbf{v} \\ \mathbf{v} \end{bmatrix}. \quad (6.46)$$

The system matrix $\tilde{\mathbf{A}}^{(i)}$ shows to have the structure of a "correct" FV discretization, e.g. it shows experimentally to fulfill the orthogonality condition $R(\tilde{\mathbf{A}}^{(i)}) = N(\tilde{\mathbf{A}}^{(i)})^{\perp}$, even though it is not symmetric.

Example 6.1: Fulfilling the Orthogonality Condition The example is based on Example A.3 having Dirichlet conditions on the west boundary of the bottom left block and homogeneous Neumann elsewhere. All but the bottom left block yield singular Schur complements.

We will investigate the matrices $\mathbf{A}^{(i)}$ and $\mathbf{S}^{(i)}$ for the singular block in the bottom right corner.

The objective is to show which attempts in this section that make the matrices $\mathbf{A}^{(i)}$ and $\mathbf{S}^{(i)}$ fulfill the orthogonality condition (6.40). The three attempts that are tested, are:

- The first attempt using an average to create the interface variables.
- The second attempt creating the interface variables as a part of a finite volume discretization.
- The third attempt including a correction of $\mathbf{A}^{(i)}$ to implement a "correct" finite volume discretization on each block.

Each sub-figure in Figure 6.4 shows the projection of the range on the null space: All eigenvectors except the one corresponding to the zero eigenvalue are projected onto the constant vector. The projection is carried out for the three approaches mentioned above and for as well $\mathbf{A}^{(i)}$ as $\mathbf{S}^{(i)}$. If a matrix fulfills the orthogonality condition (6.40), then the projection should be zero for all eigenvectors spanning the range.

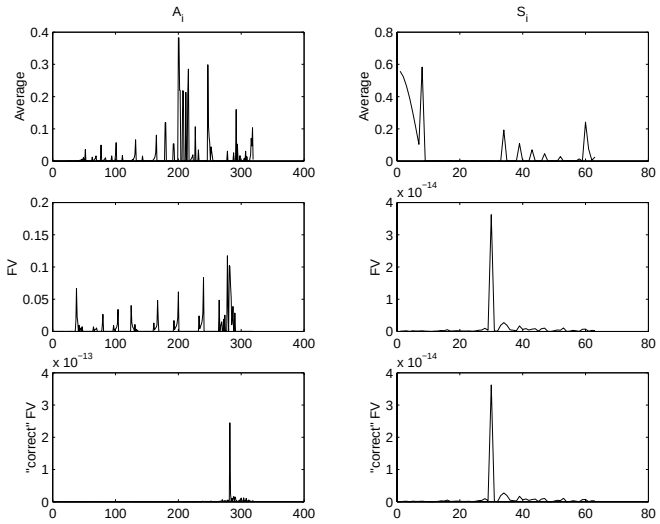


Figure 6.4: *Projection of the range at null space*

- In the first attempt neither $\mathbf{A}^{(i)}$ nor $\mathbf{S}^{(i)}$ fulfills the orthogonality condition, and as mentioned previously the method does not work.
- In the second attempt $\mathbf{S}^{(i)}$ but not $\mathbf{A}^{(i)}$ fulfills the condition. However the method works fine.
- When correcting $\mathbf{A}^{(i)}$, both fulfill the condition, and the method still works fine.

Whether $\tilde{\mathbf{A}}^{(i)}$ or $\mathbf{A}^{(i)}$ is used to solve the local Neumann problems does not make any difference.[4]

End of Example 6.1.

Note if there is no diagonal cell dependence over the interface, e.g.

[4] Apart from that Matlab produces far less warnings when using $\tilde{\mathbf{A}}^{(i)}$

when the grid is orthogonal, then

$$\mathbf{C}_i = \left(\hat{\mathbf{D}}\tilde{\mathbf{I}}_i\right)^T = \tilde{\mathbf{I}}_i^T\hat{\mathbf{D}}, \tag{6.47}$$

or

$$\mathbf{C}_i\hat{\mathbf{D}}^{-1} = \tilde{\mathbf{I}}_i^T \tag{6.48}$$

If the grid furthermore is sufficiently regular, e.g. equidistant then $\hat{\mathbf{D}} = \mathbf{I}$ and $\mathbf{C}_i = \tilde{\mathbf{I}}_i^T$.

6.4 Complexity of BDD method

The objective of this section is to outline the computational complexity of the BDD method. We do not include all details, but as a measure of complexity we count the number of algebraic systems that need to be solved. This will be counted as local block solves. We will distinguish between solves with Dirichlet and Neumann BC and call them local Dirichlet solve or local Neumann solve respectively. Also we will count the number of coarse grid solves.

The method consist of two basic steps. Assume the domain is decomposed into k blocks. The first basic step is to compute the matrix vector product $\mathbf{Sv}$ (6.12), which involves k local Dirichlet solves.

The second basic step is to apply the preconditioner to compute an approximate solution as described in Equations (6.21). It involves:

- A coarse grid solve (6.21a).
- A matrix vector product $\mathbf{S}\lambda$ including k local Dirichlet solves (6.21b).
- k local Neumann solves (6.21c).
- A second matrix vector product $\mathbf{Su}_B$ including k local Dirichlet solves (6.21e).
- A second coarse grid solve (6.21e).

In total $2k$ local Dirichlet solves, k local Neumann solves, and two coarse grid solves

Before a Krylov subspace method can be started, some initialization must be done:

- The right hand side g must be created as in Equation (6.9). In total k local Dirichlet solves.
- The coarse grid operator $\mathbf{A}_0 = \mathbf{Q}^T\mathbf{SQ}$ must be created. For each column in $\mathbf{Q}$ we must apply the matrix vector product (6.12), which involves k local Dirichlet solves each. Since there will usually be $O(k)$ columns in $\mathbf{Q}$, this will imply in total $O(k^2)$ local Dirichlet solves. However, most of the local solves will have a zero right hand side and thereby also a zero solution, which can be exploited. Then only $O(d_c k)$ local solves are necessary, d_c being the dimension of the decomposition.

In total $O((1+d_c)k)$ local Dirichlet solves for initialization.

A Krylov method usually for each iteration apply the matrix vector product once, and the preconditioner once, in total $3k$ local Dirichlet solves, k local Neumann solves, and two coarse grid solves per iteration.

Finally to obtain the solution on the interior, we must perform a back solve (6.11) involving k local Dirichlet solves.

Then the computations are done, the solution is found.

CHAPTER 7

Computational Results

The purpose of this chapter is to verify properties stated in previous chapters by numerical experiments

For the classical alternating Schwarz methods, no numerical experiments have been made, because the method does not fit appropriately into the NS3 context. It will be mentioned however, if a classical alternating Schwarz method arise as a special case of another method.

Computations have been carried out on the machine Newton in the G-databar at DTU. This is a Sun Enterprise 6500, a 24 processor machine each being 400 MHz/8MB cache Sun UltraSparc II CPU's, running Solaris SunOS 5.7.

The G-databar is shared among all students at DTU, and therefor timings depend on the average load of the machine.[1] Timings are subject to some degree of averaging and should be taken as guidelines only.

[1] This is the case even though the Matlab function `cputime` should return the time used by Matlab only.

7.1 Notes on Implementation

Implementation is done in Matlab, using Matlabs sparse matrix data structures. Matlab 6.0 (R12) is available, and timings have been made using Matlabs `cputime` function.

A suite of test examples are listed in Appendix A. All test examples originate from NS3: Grids and systems matrices are created in NS3. Routines to dump NS3 data to a file and Matlab routines to read the same data into Matlab have been provided by Stefan Mayer. Data from NS3 include nonzero coefficient of the system matrix, row and column indexes of these nonzero coefficients, a right hand side, and grid positions for cell centers in 2D.

Local block solutions, that is in the Schur method case; solutions to the local Dirichlet problems (6.12) and the local Neumann problems (6.19), are based on the backslash operator in Matlab. The name "backslash operator" originates from the Matlab syntax, where a system $\mathbf{Au} = \mathbf{f}$ is solved for $\mathbf{u}$ by typing

```
> u = A \ f;
```

The backslash operator is also called "left matrix divide". When the backslash operator fails to produce a solution, Matlabs Krylov method GMRES is applied instead.

The Krylov method used to solve the Schur complement system, is chosen to be Matlabs GMRES, using a restart of 10.

Emphasis in the implementation is on general structures rather than efficiency. The objective is to exploit properties of different methods, and it is an important property that new ideas are fast and easy to implement and test. Another reason to take timings as guidelines only.

Matlab implementation of the two basic parts of the BDD method is presented in Appendix F. The matrix vector product (6.12) is presented in F.1, while the BDD preconditioner is presented in F.2.

7.2 DN Method

We will consider the generalized DN method in as well the approximative as the exact formulation.

Results listed with the parameters α and β are produced by the approximative method, while results listed with γ are produced by the exact method.

We will state two results for each test. The first is the error reduction factor for the last 12 iterations, as to approximate the asymptotic error reduction factor. The second is the overall error reduction factor. Since the error is usually reduced faster in the first couple of iterations, the overall factor is usually better than the asymptotic.

We will first consider a two block setup to verify whether two iteration convergence can be achieved in the regular case, and to see how much an irregular grid and different sized blocks influence convergence.

Secondly we will address a 2×2 block setup to see performance for different iteration matrices.

7.2.1 Results for Two Block System in 2D

Test grids for the two block setup constitute the two lower blocks of the examples in Appendix A.1.

The tests have Dirichlet conditions on the original domain boundary, and the domain is discretized using a regular grid of 32×16 cells, if not otherwise specified.

The first tests will consider only the regular Example A.1.

Classical Alternating Schwarz. First we examine the performance of the classical alternating Schwarz method by using only Dirichlet steps in the DN method.

A variant with 1 cell overlap arises from the Dirichlet step (5.34) of the approximative DN method using $\alpha = 1$, while a $\frac{1}{2}$ cell overlap arises when setting $\gamma = -1$ in the DN method (5.38).

Results for the two cases are listed in Table 7.1. As expected, when

	D. par.	Error reduction factor	
1 cell overlap	$\alpha = 1.0$	0.826	0.768
$\frac{1}{2}$ cell overlap	$\gamma = -1.0$	0.903	0.841

Table 7.1: *Classical alternating Schwarz*

the overlap is increased from $\frac{1}{2}$ cell to 1 cell, convergence speed increase similarly.

Note that the setup here is the same as in Example 5.6: The asymptotic error reduction factor for the DN method of 0.903 match the largest absolute eigenvalue for the iteration matrix, found in the example to be 0.905.

Two Iteration Convergence of the DN Method. The next objective is to verify that two iteration convergence can be obtained by the DN method also in 2D, as was possible in 1D in Example 5.1. The direction matrix is

$$\mathbf{O} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}. \tag{7.1}$$

Several choices of parameters of the approximative method, α and β, have been explored, some are listed in Table 7.2. The DN method pro-

	D. par.	N. par.	Error reduction factor	
DN	$\alpha = 0.5$	$\beta = 0.5$	0.333	0.279
	$\alpha = 0.5$	$\beta = 0.6$	0.310	0.270
	$\gamma_1 = -1.0$	$\gamma_2 = 1.0$	2 it.	

Table 7.2: *DN method*

duces a solution in two iterations, however the approximative method does not. Best convergence speed is actually not even achieved for the natural parameters $\alpha = \beta = 0.5$, but they are close to optimal. Convergence is a lot better than with the classical alternating Schwarz method though.

Dependence on Grid Spacing. The two tests above is repeated with a finer discretization using 64×32 cells. That is grid spacing is halfed in both directions.

In the classical alternating Schwarz methods, the overlap is halfed, and so convergence speed should be halfed. Results are in Table 7.3, and behave as expected. Note especially that $1/2$ cell overlap in Table

	D. par.	Error reduction factor	
1 cell overlap	$\alpha = 1.0$	0.906	0.842
$\frac{1}{2}$ cell overlap	$\gamma = -1.0$	0.949	0.883

Table 7.3: *Classical alternating Schwarz with finer discretization*

7.1 match the one cell overlap in Table 7.3.

Using again the iteration matrix $\mathbf{O}$ (7.1) from above, the DN method produces Table 7.4.

	D. par.	N. par.	Error reduction factor	
64×32	$\alpha = 0.5$	$\beta = 0.5$	0.258	0.210
128×64	$\alpha = 0.5$	$\beta = 0.5$	0.156	0.129
64×32	$\gamma_1 = -1.0$	$\gamma_2 = 1.0$	2 it.	

Table 7.4: *DN method with finer discretization*

As expected, the DN method case gives two iteration convergence, i.e. it is independent of grid spacing. It might be a bit surprising though, that convergence improves for the approximative version. The argument is: As the grid spacing is decreased, the values communicated over the boundary are defined closer and closer to the interface. In the limit, values are defined at the interface, in which case the approximative and exact method are alike. Loosely speaking, the approximative method converges towards the exact method as the grid spacing is decreased, and is thereby a better and better approximation of the exact method, hence the name “approximative”.

Non Regular Grids. To complete the 2 block case, we finally consider non regular grids and different sized blocks, to give an idea of how the DN method works for more general examples. Table 7.5 lists results for Example A.3 where the blocks are not equally big, and Table 7.6 lists results for Example A.4 where also the grid is nonorthogonal.

	D. par.	N. par.	Error reduction factor	
classical	$\alpha = 1$		0.830	0.738
	$\gamma_1 = -1.0$		0.906	0.805
DN	$\alpha = 0.5$	$\beta = 0.5$	0.347	0.278
	$\alpha = 0.6$	$\beta = 0.5$	0.309	0.252
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.290	0.261

Table 7.5: *Blocks of different size, Example A.3*

	D. par.	N. par.	Error reduction factor	
classical	$\alpha = 1$		0.363	0.329
	$\gamma_1 = -1.0$		0.535	0.486
DN	$\alpha = 0.5$	$\beta = 0.5$	0.430	0.430
	$\alpha = 0.7$	$\beta = 0.5$	0.348	0.309
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.193	0.193

Table 7.6: *Blocks of different size and a irregular grid, Example A.4*

Notice the following which is the case for results in both tables:

- The classical methods using 1 cell overlap is about twice as fast as if using only $\frac{1}{2}$ cell overlap.
- The DN method does not any longer produce a solution in two iterations. However, the error reduction factor is small, and only two iterations is used to decrease the error by more than one decade.
- The DN method is superior to the approximative DN method.
- For the approximative DN method, the natural parameters are not necessarily the optimal. Here an trial and error approach have been used to find optimal parameters, but this might not always be possible. Choosing optimal parameters have some influence, as is most evident in Table 7.6.

Note that the γ parameters in the DN method are not optimizable. If other values than ± 1 are chosen, the method converges very slowly or not at all. This have been verified experimentally.

Making the same test on 3D examples produce somewhat similar results. It have not been tested extensively, so no results will be presented here.

7.2.2 Results for 2×2 Block System in 2D

Test grids in the 2×2 block setup are the examples in Appendix A.1. The domains have Dirichlet BCs on all boundaries, and the domains are discretized using 32×32 cells.

Tests have been made with the direction matrices from Section 5.5, which I will rewrite here. In addition the matrix $\mathbf{O}_0$ is created to exchange Dirichlet data only, implementing a classical alternating Schwarz method.

$$\mathbf{O}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}. \tag{7.2a}$$

$$\mathbf{O}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{7.2b}$$

$$\mathbf{O}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}. \tag{7.2c}$$

$$\mathbf{O}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{7.2d}$$

$$\mathbf{O}_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}. \tag{7.2e}$$

These direction matrices have been applied using both the approximative and the exact DN method to Example A.1, Example A.2, and

Example A.4.

Results are listed in Appendix A.4.1.

Note especially that convergence for the regular grid of Example A.1 was achieved in only 3 iterations using $\mathbf{O}_4$, and in 4 iterations using $\mathbf{O}_3$. This is a very strong result.

The following list will summarize results and conclusions from the tables in Appendix A.4.1 for each of the 5 iteration matrices:

- As have always been the case, the $\mathbf{O}_0$ case is twice as good for the approximative DN method compared to the DN method.
- The $\mathbf{O}_1$ case diverges for all examples using the DN method, as is also the case in Example 5.7.
  The approximative method diverges for many choices of parameters. Using sufficient underrelaxation on the Neumann parameter, $\beta < 0.5$, can make the method behave fairly well. Therefor some optimization is needed to find the best choices of the parameters.
  However, it should be possible to get fairly good results with the DN method, if underrelaxation is applied, i.e. the new result is a weighted sum of the Neumann result and the result from previous iteration. This has not been tried.
- The $\mathbf{O}_2$ is very stable. It is actually a bit to conservative, in the sense that fastest convergence is achieved using over-relaxation parameters, $\beta > 0.5$ and $\alpha > 0.5$, and still it is not very efficient. Some optimization is needed here to find the best choice of parameters.
- The $\mathbf{O}_3$ using the DN method gives 4 iteration convergence on the regular grid. Actually any combination of the 4 rows in $\mathbf{O}_3$ will accomplish this. The matrix furthermore gives fast convergence for the other grids. For the approximative DN method, it converges for most choices of α and β. The optimal parameters are close to the natural ($\alpha = \beta = 0.5$). If not, the natural parameters give close to optimal behavior.
- The $\mathbf{O}_4$ is by far the best. The DN method gives 3 iteration convergence on the regular grid and very good convergence on the others. Also the approximative DN method gives very fine results, having optimal parameters close to the natural, or again: The natural parameters give close to optimal behavior. The approximative DN method is not quite as fast as the DN method.

7.2.3 Performance in General

It has not been possible to produce fast convergence for a general setup.

I have especially tried to make results for a 3D setup of $2 \times 2 \times 2$ blocks, but without success. Convergence could no be improved significantly comparing with a classical alternating Schwarz method with a one cell overlap.

The failure to produce results in a general setup is a major drawback of the method.

7.3 Schur and BDD Methods

The tests we will go through here are designed to verify that properties stated in [Smith96] of almost independence of discretization, and for the BDD method almost independence of block decomposition, also apply to a finite volume discretization.

Since the Krylov method GMRES is used to iterate globally, the error in each iteration is not available as in the previous section. GMRES returns the norm of the residual, however the difference between the residual and the error might be several orders of magnitude depending on the system.

We will declare convergence when the original residual has been decreased by a factor 10^{-10}, and as a measure record the number of iterations used. If the residual is decreased by this factor, then the error will usually decrease similarly.

We will also list the time used by the GMRES method. Initialization and setup of the Schur system are not timed. It is the time used for a single processor, and should thereby give an idea of the efficiency of

the preconditioner. Remember that timings should be taken as guidelines only.

At last, the condition number of the matrix product $\mathbf{BS}$ is calculated, when possible, i.e. when all blocks are nonsingular. The product is a kind of iteration matrix for a Krylov type iterative method, and its condition number relates directly to convergence speed. Unfortunately if any block is singular, then $\mathbf{S}^{(i)}$ is singular and its inverse has a condition number which is infinite, which the preconditioner, $\mathbf{B} = \sum(\mathbf{S}^{(i)})^{-1}$, inherits. Hence the condition number does not provide any usable information when singular blocks are present.

The right hand side has added noise at a level corresponding to its norm, $\text{var}(noise) = ||\mathbf{f}||_2$. It is necessary to add noise to the system, otherwise in many of the examples the 3D case will behave like a 2D or 1D case: The BCs do usually not vary, hence there will be no variation in the 3rd dimension. Furthermore, the energy in white noise is equally distributed on all scales, hence if a method can converge fast with noise as the right hand side, then it can most likely converge fast with any right hand side.

7.3.1 Neumann-Neumann Method

Independence of Discretization. We will start by considering the examples that the DN method was exposed to in last section, namely the 2×2 block setup, in both 2D and 3D:

Example	Examples of Appendix A.1 in both 2D and 3D
BC	Dirichlet on W,E,N, and S. 3D version has homogeneous Neumann on F and B.
Forcing	Random
Discr.	Varying
Block dec.	2×2 ($\times 1$) blocks
Results	2D: Table 7.7. 3D: Table 7.8

The $\kappa(\mathbf{BS})$ denote the condition number of the matrix product $\mathbf{BS}$. The two tables show that in both 2D and 3D the following properties hold:

- Almost independence of the grid spacing. Table 7.7 reveals that a decrease in the grid spacing increase only slightly the condition number and the number of iterations used. It would have been best of course, if there was no increase. Theory state the the number of iterations grow like $O((1 + log(H/h))/H)$ [Smith96], so the increase seems to conform with theory.
- How the grid is constructed within each block has a somewhat subtle influence on convergence. Going from a regular grid to a non regular grid, from Example A.1 to A.2 or from Example A.3 to A.4, increases the condition number slightly, almost without affecting the iteration count though. The conclusion must be that the grids inside the blocks do not matter much.

Ex.	# cells	# it.	cpu time	$\kappa(\mathbf{BS})$
	32×32	5	2.7	4.48
A.1	64×64	5	17	6.35
	128×128	6	145	8.99
A.2	32×32	5	5.0	4.67
A.3	32×32	6	5.0	3.81
A.4	32×32	6	5.8	4.28

Table 7.7: *2D case*

Ex.	# cells	# it.	cpu time	$\kappa(\mathbf{BS})$
A.1	$10 \times 10 \times 10$	8	3.1	2.48
A.2	$10 \times 10 \times 10$	8	2.9	2.81
A.3	$10 \times 10 \times 10$	6	2.3	1.65
A.4	$10 \times 10 \times 10$	7	2.7	1.92

Table 7.8: *3D case*

Decreasing Performance as the Number of Block Grows. This test will show the implications of lacking a coarse grid correction, hence if increasing the number of blocks in any dimension, convergence will deteriorate. The example is based on the oblong Example A.5, and the following test show the deterioration of convergence as the number of

blocks in the horizontal direction increases.

Example	A.5.
BC	Pure Dirichlet.
Forcing	Random.
Discr.	32×32 cells, regular grid.
Block dec.	$k \times 1$ blocks
Results	Table 7.9

$k \times 1$	# it.	cpu time	$\kappa(\mathbf{BS})$
2×1	1	1.7	1.0
4×1	3	2.6	1.004
8×1	6	3.1	1.16
16×1	12	4.6	2.31

Table 7.9: *2D case*

Results in Table 7.9 verify that the iteration number grows like $O(1 + log(H/h)/H)$, which in this case is almost the same as $O(1/H)$. Also notice the growth in the condition number.

7.3.2 BDD Method

The BDD method adds a coarse grid correction, and should therefor be more or less independent of the number of blocks and the block size. For certain model problems is is possible to show, that the condition number grows like $O((1 + log(H/h))^2)$ [Smith96]. Hence iteration count is almost independent of as well the grid spacing h and the block size H.

Independence of Number of Blocks in 1D This first test shows how a coarse grid correction makes a difference. It is Example A.5 without noise, and decomposed in the horizontal direction only.

Example	A.5 2D with 1D properties.
BC	Homogeneous Dirichlet at east and west, homogeneous Neumann elsewhere.
Forcing	Constant.
Discr.	64×16 cells, regular grid.
Block dec.	$k \times 1$ blocks
Results	Table 7.10.

$k \times 1$	# it.	cpu time
2×1	1	1.07
4×1	1	1.27
8×1	1	0.86
16×1	1	0.64

Table 7.10: *2D case*

Since the BCs have no variation in the 2nd dimension and furthermore the forcing over the entire domain is constant (no noise), there will be no variation in the second dimension. It will behave as a 1D problem. Results in Table 7.10 show that for this 2D case with 1D properties, convergence is achieved in one iteration for any number of blocks.

Checkerboard Decomposition The next test will apply a checkerboard decomposition on Example A.1 in 2D.

Example	A.1 in 2D.
BC	Dirichlet at south, Neumann elsewhere.
Forcing	Random.
Discr.	48×48 cells, regular grid.
Block dec.	$k \times k$ blocks, checkerboard pattern.
Results	Table 7.11.

The results in Table 7.11 verify the independence of block number and block size. Even timings stay constant, or decrease, at least in the beginning. Timings is commented in more detail later.

$k \times k$	# it.	cpu time
2×2	5	7.4
3×3	15	15.0
4×4	15	13.3
6×6	17	12.8
8×8	17	14.2
12×12	16	20.5
16×16	15	36.6

Table 7.11: *2D case*

Decomposing Only in One Directions This test is alike the first test of the BDD method, apart from noise is added to the right hand side, giving 2D variation.

Example	A.5 in 2D.
BC	Dirichlet at east and west, homogeneous Neumann elsewhere.
Forcing	Random.
Discr.	96×24 cells, regular grid.
Block dec.	$k \times 1$ blocks
Results	Table 7.12.

$k \times 1$	# it.	cpu time
2×1	1	1.8
4×1	3	5.1
8×1	6	6.6
16×1	11	8.7
32×1	33	30
48×1	52	44

Table 7.12: *2D case*

Table 7.12 shows that it is not sufficient to decompose in just one direction.

However if we decompose evenly in both direction:

Example	A.5 in 2D.
BC	Dirichlet at east and west, homogeneous Neumann elsewhere.
Forcing	Random.
Discr.	96×24 cells, regular grid.
Block dec.	$4k \times k$ blocks
Results	Table 7.13.

$4k \times k$	# it.	cpu time
4×1	3	5.0
8×2	11	10.5
16×4	16	30
32×8	14	41
48×12	12	84

Table 7.13: *2D case*

Then Table 7.13 reveal that iteration count again is independent of the block decomposition, as for the checkerboard test earlier.

Note especially the cpu timings in Table 7.13. They are increasing quite heavily as the number of blocks grows. The same is the case for the checkerboard decomposition results in Table 7.11.

Consider the last test in Table 7.13: The domain is discretized in $96 \times 24 = 2304$ cells. These are decomposed into $48 \times 12 = 576$ blocks having each $2 \times 2 = 4$ cells. The decomposition produces in total 1092 block to block interfaces[2], which each add 2 shadow variables to the number of unknowns, in total 2184 extra unknowns. Hence there are about as many shadow variables as interior variables. The system has grown to double size, and the extra time needed to calculate the solution to these extra variables has become significant. Furthermore Matlab must take care of many small matrices, and many loops grows big, hence there is most likely some administrative overhead included in order to handle that many blocks.

[2] $47 \cdot 12 + 48 \cdot 11 = 1092$

Blocks With High Aspect Ratio Table 7.12 shows that it is not sufficient to decompose in just one direction, but what if only one block has a high aspect ratio?

Example	A.5 in 3D.
BC	Dirichlet at east and west, homogeneous Neumann elsewhere.
Forcing	Random.
Discr.	$128 \times 5 \times 3$ cells, regular grid.
Block dec.	$3 \times 1 \times 1$ blocks, the middle block width is decreased, its width compared to the total width is given as aspect ratio.
Results	Table 7.14.

Aspect ratio	# it.	cpu time
$1:3$	2	3.1
$1:5$	2	3.1
$1:7$	3	4.3
$1:13$	5	6.9
$1:21$	6	7.9
$1:31$	8	10.1
$1:65$	10	12.1
$1:128$	12	15.8

Table 7.14: *2D case*

Table 7.14 shows an increase in the iteration count as the aspect ratio gets worse. However the iteration count do not increase as heavily as in Table 7.12, where also the aspect ratio gets worse as more blocks are used.

Other Examples

The last two examples are provided to show that the method works for more general grids and decompositions than a rectangular grid using a 2×1 or 2×2 block decomposition.

The first example is the $2 \times 2 \times 2$ block decomposition, which we have not yet found a way to solve using the DN method.

Example	A.1 and A.2 in 3D.
BC	Dirichlet at front, homogeneous Neumann elsewhere.
Forcing	Random.
Discr.	$10 \times 10 \times 10$ cells, regular grid.
Block dec.	$2 \times 2 \times 2$ blocks
Results	Table 7.15.

Example	# it.	cpu time
A.1	13	9.7
A.2	15	9.0

Table 7.15: $2 \times 2 \times 2$ *block decomposition*

Table 7.15 shows that the iteration counts compare with any of the other tests so far, on as well a regular as a not so regular grid.

The last test is an example of how a grid is typically created close to a corner.

Example	A.7 in 3D.
BC	Dirichlet at east boundary of top right block, inhomogeneous Neumann on west boundary of top left, homogeneous Neumann elsewhere.
Forcing	Random.
Discr.	In 3 block setup: Top left block: $6 \times 18 \times 4$ cells, top right block: $18 \times 18 \times 4$ cells, bottom block $18 \times 6 \times 4$ cells, irregular orthogonal grid.
Block dec.	$k + 1 \times 1$ blocks
Results	Table 7.16.

Table 7.16 again compare with iteration count for the other tests. Note however that the block decomposition is only refined in one direction, hence the iteration count grows with the number of blocks.

k	# it.	cpu time
3	17	67.4
5	18	42.6
7	22	35.5

Table 7.16: *2D case*

CHAPTER 8

Summary

This project is in some sense far from over. It have been like a journey with a given destination, but without map showing the paths.

During the process many paths have been visited. Most have been turned down. Maybe because they ended blind, but usually we turned around before we knew were it lead; at the turning point it was evaluated as another dead end, or another long way round. Therefor a lot of paths are left relatively unexplored.

Also the destination has not yet been reached, NS3 is not yet parallel.

The objective of this chapter is to summarize what have been accomplished so far. I will relate it to articles on the subject if I have found any. Also I will outline subjects which need further investigation.

Subjects for further investigation split into two categories: The work to be done to implement a parallel Poisson solver in NS3, and work which from another point of view may be interesting. The former will be presented in a separate section, while the latter is addressed as a part of the specific subject.

8.1 Matlab

There have been several problems with Matlab, mainly with the Matlab backslash operator used to solve a system $\mathbf{Au} = \mathbf{f}$. The problems arise when the system matrix is singular, and the system is consistent. In this case, there is an infinity of solutions.

A singular matrix represented in Matlab is usually due to rounding errors only close to singular.

In the process of calculating the solution to the singular system, the backslash operator might encounter a zero, which is used in a division. The division by zero produces a `Inf` value, Matlabs representation of infinity. The `Inf` value, when encountered in other expressions, will produce again either `Inf` or `NaN`, "not a number", and the result returned from the operator is usually a vector of `NaN` values. The method has broken down. The break down of the method does not depend on whether the right hand side is consistent or not.

However, if not a strict zero is encountered due to the rounding errors, the backslash operator has no problem in finding one of the solutions to the system. This difference in behavior from a break down case to a near break down case seems not well justified.

In case of a consistent system, a Krylov subspace method can be used, and it will find a solution. However, for the systems considered here, the backslash operator usually out-competes a Krylov subspace method in the time used to find a solution, and is therefor preferred.

One solution to this dilemma is simply to add noise of very small magnitude (close to machine precision) to the diagonal of the system matrix, just enough to make sure that the backslash operator never encounters a strict zero. This will usually only alter the solution at the same magnitude, and will usually be acceptable. Another solution is to examine the result from the backslash operator, and if it contains `NaN` values, then try again with a Krylov subspace method. Both are not very nice brute force solutions, but they work.

As for singular inconsistent system, there exist no solution to the system. The backslash operator produces a result of order $1/eps$, eps being the machine precision of Matlabs double precision arithmetic.

A result which is unusable. It would have been better if Matlab had returned a solution in a least squares sense, as it does when the system matrix is under- or over-determined.[1] So, if a solution in a least square sense is sought, simply remove an arbitrary row from the system matrix, making the system under-determined, and then use the backslash operator to solve the system. This works well, but have not been used in testing. A Krylov subspace method for inconsistent systems will not converge at all.

For as well consistent as inconsistent singular system, another approach to solve the above problems is to use regularization methods [Hans98].

Apart from the problems about the singular system, Matlab have been a very effective tool in the development and test of the different methods. There have been more time to thoughts and development than if method and test should have been implemented in an ordinary programming language.

8.2 Classical Alternating Schwarz

According to [Smith96], classical alternating Schwarz methods in conjunction with multigrid have in practice the following properties: If the problem size and the number of computational nodes are both doubled, then it will take the same time to compute the solution. But it is not fully scalable: If increasing the number of computational nodes p to solve the same problem, the solution time will not be proportional to $1/p$.

Otherwise there is not much to say about the method. It does not fit into NS3, which is the main reason that it have not been explored more extensively here.

[1] According to Matlabs help text and Matlabs Function Reference for the backslash operator

8.3 Non Overlapping Domain Decomposition

This have lead to some interesting result in the 2×2 block case: 3 iteration convergence. It is not something I have seen addressed in any articles, where on contrary several seem to work with the case of 2 blocks [Hadj00], [Rice98a], [Rice98b], [Doug97]. As is stated in [Doug97], "the authors are still considering the case when interior vertices occur".

This, I believe, is just as worthy of an exploration as the 2 block case. It might lead to further insight into why examples with interior vertices sometimes behave very poor according to e.g. Example 5.7, and how this may be circumvented.

Otherwise the non overlapping method presented here has not performed satisfactory. It lacks several features before it is applicable to more general geometries and NS3:

- It is in general difficult, if not impossible, to decide which direction matrix to use. For simple cases experience may give an idea of how to construct it, but for cases with many blocks in a complex setup, it might be difficult to get even fair convergence.
- It is not straight forward how to accelerate the method by e.g. a Krylov method. Two consecutive steps are usually not alike, which is a requirement for a standard Krylov method. Therefor either a special Krylov method must be applied, or all the different steps should be grouped into one big step, which is then fed to a standard Krylov method. Since a combination of all steps may be very unstable, care must be taken.
- The method needs a coarse grid correction in order to get convergence independent of the number of blocks. How this coarse grid should be created is yet to be explored. Some initial attempts to use the same technique as the BDD method have failed, and arguments from multigrid theory can motivate why: The error is not smooth on the fine grid before it is restricted to the coarse grid. Most likely another approach is necessary.
- Finally, at present the method converge slowly or not at all, if some blocks are singular. Therefor procedures to handle the singular cases must be provided.

8.4 Schur Complement Methods

The data imported from NS3 do not include flux through each cell face. It has therefor not been possible to create the interface variables for general systems, since their creation is based on the flux over the interface. Though, if no diagonal cell dependencies exist over the interface, the flux can be retracted from the system matrix. Hence it has only been possible to test examples which do not include diagonal cell dependencies over the interface. This is the case for an orthogonal grid. The non orthogonal examples tested with the BDD method have been especially designed not to have diagonal cell dependencies. However, test to verify that it works in general should be applied.

In [Smith96] is given several algorithms for solving the Schur complement. Most behave asymptotically like the BDD method. However, a method called the Vertex Space Method or Copper Mountain Algorithm shows to have condition number and thereby convergence rate independent of mesh as well as block decomposition. It adds several local solvers associated with points near vertices in 2D, and near vertices and edges in 3D. It may be worth to investigate whether these extra solvers make a difference in practice compared to the BDD preconditioner and if so, whether the effort to adapt NS3 to implement these local solvers is worth the work.

8.4.1 Neumann-Neumann Method

What is the difference between the Neumann-Neumann method and the original DN method? Both methods solve a Dirichlet problem followed by a Neumann problem, however the Neumann-Neumann method performs well while the DN method does not converge in general.

The difference between the two can be found in their formulation: The Neumann-Neumann method is formulated as preconditioner designed to fit into an accelerated method like a Krylov method. If the Neumann-Neumann preconditioner is used instead in a stationary iterative method, then it will not converge in general but behave like the DN method. Hence the DN method will most likely perform just as well, if it could be formulated as a preconditioner to a Krylov method.

The Neumann-Neumann method will however still have an advantage: The global Krylov method works only on the interface variables. The number of interface variables is much less than the number of interior variables, hence the complexity of the Krylov method is small compared with a Krylov method on all variables.

The Neumann-Neumann methods has not been exposed to examples with singular blocks. Nothing special have been done to improve convergence when singular blocks are present, hence convergence at present is very bad and do not compare with other methods.

Thus the Neumann-Neumann method has not been tested on e.g. a checkerboard decomposition.

The Neumann-Neumann methods shows to perform fine for a smaller number of blocks. However as already mentioned the method lacks a coarse grid correction to improve performance when the number of blocks increase. The BDD method solves this, however many other methods exist which address this problem, see e.g. [Dryja93].

8.4.2 BDD Method

The BDD method shows to perform well for even a large number of blocks. Results indicate that performance of the method in a finite volume formulation as presented here corresponds with performance stated for a finite element approach in [Smith96] and [Mand93a].

However some things need to be kept in mind: It is important to decompose evenly in all directions. Whether it should be even in domain size or in number of cells in each block have not been addressed.

Further testing must show.

Note that the BDD preconditioner can be used as a preconditioner for a single processor machine as well, if considering the computing timings. Especially Table 7.10 show that computing times are almost halfed going from 1 to 16 blocks.

The BDD method allows geometries of very general shape, more general than NS3 can create.

Even though the BDD preconditioner is general, it is restricted to problems where the null space of the operator, or in unsymmetric cases the null space of the orthogonal operator, is known or computable. This is the case for the Poisson problem, but for other elliptic problems the null space may not be known in advance. If so, the null space can be computed: Assume $\mathbf{S}^{(i)}$ is singular, then the null space of its transpose can be found by solving

$$(\mathbf{S}^{(i)})^T \mathbf{v} = \mathbf{0} \tag{8.1}$$

at least as many times as the dimension of the null space, using a Krylov method and different random start vectors $\mathbf{v}$. All the solutions $\mathbf{v}$ should span the null space [Bark92].

As a remark, I will mention that the BDD have already been applied to the incompressible Stokes equation in [Pava96], where computational results conform with theory. [Pava96] is based on a discretization using mixed finite or spectral elements.

8.5 Conclusion

Let us return to the very first sentence of this thesis: How to parallelize a Poisson solver. There is no doubt for me that the BDD method can do this efficiently:

- The restrictions from NS3 are obeyed.

  - The method is based on local solvers.
  - The coarse grid dimension is minimal, only one variable per block.
- The method is based on an accelerated Krylov subspace method. The Krylov method iterate only on the interface variables, whos number is much smaller than the entire number of variables.
- Communication between blocks is only on the finest level.

The main motivation was to parallelize the NS3 package. However, there have not been enough time during this project to actually implement the BDD method in NS3, and there are still things to investigate before an actual implementation starts. Therefor it has not been possible to compare the method against the existing implementation on a single node machine.

How will NS3 respond to the BDD method? The biggest addition to NS3 is from my knowledge of the program the implementation of a Krylov subspace method. I do not believe that this is a very difficult thing to do. The local block solvers more or less already exist in the package. However, each block should be able to solve both a Dirichlet and a Neumann problem of slightly different size, which will introduce some changes in order to do this efficiently.

8.6 Further Work

Some areas still need to be investigated in order to implement the BDD method most effectively in NS3. Some will be mentioned here:

- Explore the different Krylov subspace methods, and find the one fitted the best for this Poisson problem.
  In NS3 the transpose is available. GMRES does not use this. Therefor there might be more efficient solvers, that take advantage of this. Furthermore, another method may simply perform better than GMRES in general for the Poisson problem.
- Parallelize the Krylov subspace method.
  One way to implement the BDD method is to have one computational node which handle the Krylov subspace algorithm on all the interface variables, and let this node send data to the remaining nodes to compute exact block solutions. This is just an ordinary sequential Krylov subspace method using some parallel subroutines to compute matrix vector products and to apply a preconditioner.
  However, it is most likely more efficient if a distributed version of a Krylov subspace method is implemented, having e.g. all inner products calculated locally and then apply some communication to find the global inner product.
  A subject for further investigation is to explore whether a Krylov subspace method can be parallelized effectively. In [Haase96] is described a parallel GMRES, another useful reference may be [Erhel95].
- NS3 can handle moving grids. Moving grids imply that the system matrix changes for each iteration. Strictly it is necessary to update the balancing procedures (coarse grid operator) each time the system has changed.
  However this might be computationally too expensive.
  Hence an examination of when it is possible / sufficient to reuse a no longer exact coarse grid operator, and how this influence convergence, may save valuable computing time.
- The effect of inexact block solvers.
  In this work, only exact block solvers have been applied. Computing only an approximate solution on each block will spare some computing time per iteration at the expense of an increase in the number of iterations of the global method.
  Whether the time spared in solving each block inexact compensates for the increase in iteration count is unknown in our case.
  The problem of inexact versus exact block solvers is addressed in [Brak98] (based on a finite volume staggered grid discretization decomposed in a Scwarz like manner with minimal overlap and without a coarse grid correction). [Brak98] conclude that inexact subdomain solvers can reduce computing time.
- The optimal decomposition, that is how to balance the discretization and the block decomposition.

In order to get more computing power, we would like to include more computational nodes and thereby we would also like a matching block decomposition.
However, the more blocks, the more extra interface variables we need to solve for. As argued previously, the number of interface variables should be so small, so that the extra time used to compute a solution to these extra variables must not be significant compared to solving for the original variables.
Furthermore, the more blocks and interface variables, the more communication is needed. Communication is usually expensive, and should not be a bottleneck. Hence, communication time must in some way match computation time.
In other words, there will be an optimal decomposition giving the smallest computation time. Guidelines of how to crate an optimal decomposition as a function of discretization and number of computational nodes can be valuable.
[Chan95] address a somewhat similar problem, finding the optimal coarse grid size for an additive Schwarz preconditioner to a Krylov subspace method.

- Consider a case where there are more blocks than computational nodes. Then blocks on the same node may use results from other blocks already calculated, e.i. for blocks on the same node the method should be multiplicative.
  This is of special interest in cases where only one or few computational nodes are available.
  The question is whether the BDD preconditioner exists in a semi or full multiplicative version. And if it does, whether it improves convergence significantly, especially in the one node case.

!

?

APPENDIX A

Test Suite and Results

The following presents the suite of test examples which is used for numerical experiments. Outer lines indicate domain boundaries and lines in the domain indicate block boundaries, interfaces. Dashed or dash-dotted lines indicate optional interfaces, i.e. when the domain can be split up into a different number of blocks. Markers (+) indicate the center of each finite volume cell, the position at which the mean value for the unknown in entire cell is determined.

A.1 2 by 2 Block Setup

All of the 2×2 block domains presented here have a 3D extension. The 3D version is a regular equidistant extension of the 2D version.

The examples come with different boundary conditions, so boundary conditions must be specified when the examples are used.

A 2×1 block setup can be achieved by decomposing only by e.g. the vertical interface, or by using the two bottom blocks only.

Figure A.1 and A.2 also exist in a cubic $2 \times 2 \times 2$ block setup, where all 8 blocks share a common corner point (vertex) located exactly in the center of the cube.

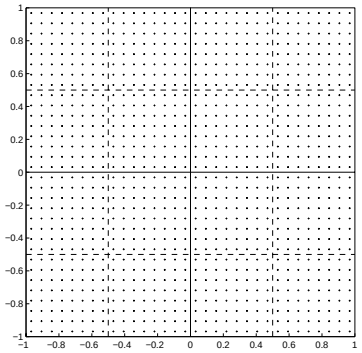


Figure A.1: *Regular grid*

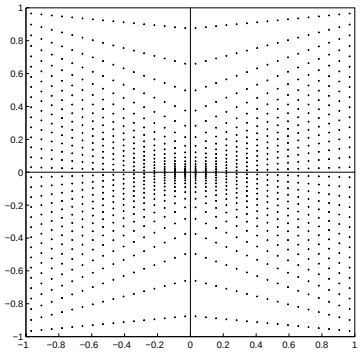


Figure A.2: *Irregular grid*

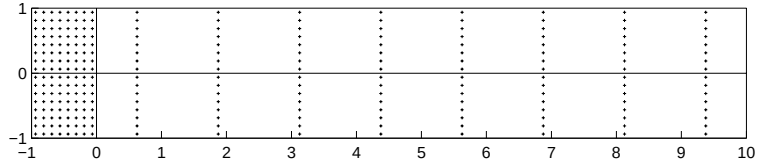


Figure A.3: *Blocks of different size with regular grid*

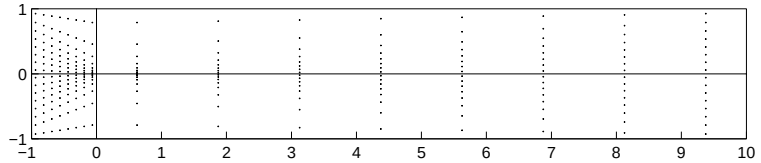


Figure A.4: *Blocks of different size with irregular grid*

A.2 4 by 1 Block Setup

The 4×1 block domains exist also both in 2D and 3D version, again the 3D versions being a regular equidistant extension of the 2D version.

If not otherwise specified, the west and east boundaries have Dirichlet conditions while north and south (and front and back in 3D) have homogeneous Neumann conditions.

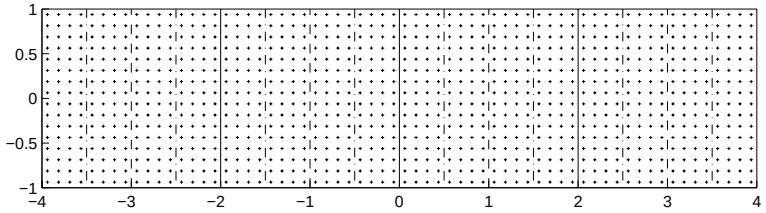


Figure A.5: *Oblong domain, regular grid*

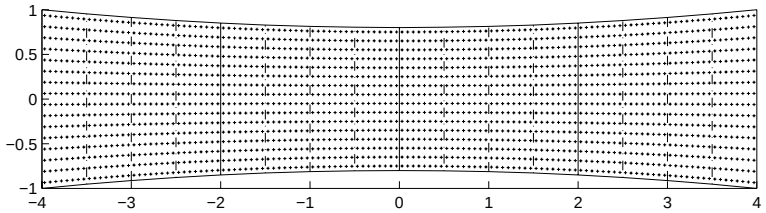


Figure A.6: *Oblong domain, irregular grid*

A.3 3 Blocks With a Corner

This example shows how a discretization may be created when a corner is present. The discretization is finer close to be able to include more details of how the fluid flows around the corner.

Only a 3D version have been explored. The west boundary of the top-left block has inhomogeneous Neumann condition while the east boundary of the top-right block has Dirichlet condition. The remaining boundaries have a homogeneous Neumann condition.

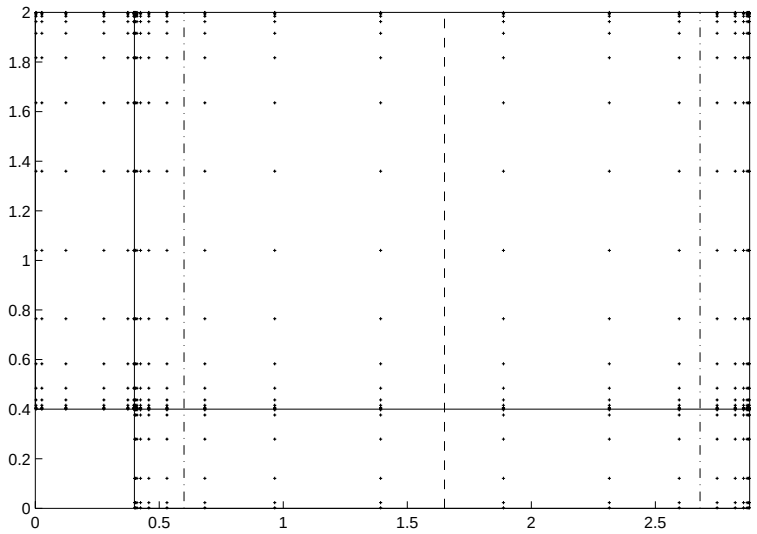


Figure A.7: *3 blocks with a corner*

A.4 Results

This appendix will present results which were to lengthy to put into the result section.

A.4.1 Non Overlapping Domain Decomposition

Tables presented here are results from Section 7.2.2 for the 3 different examples A.1, A.2, and A.4.

To recapitulate: Results listed with the parameters α and β are produced by the approximative DN method, while results listed with γ are produced by the exact DN method.

Example A.1 Regular grid.

	D. par.	Error reduction factor	
$\mathbf{O}_0$	$\alpha = 1$	0.929	0.874
	$\gamma_1 = -1.0$	0.950	0.904

Table A.1: *Classical alternating Schwarz*

	D. par.	N. step	Error reduction factor	
$\mathbf{O}_1$	$\alpha = 0.5$	$\beta = 0.3$	0.605	0.581
	$\gamma_1 = -1.0$	$\gamma_2 = 1.0$	diverges	

Table A.2: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_2$	$\alpha = 0.5$	$\beta = 0.5$	0.788	0.719
	$\alpha = 0.5$	$\beta = 0.7$	0.610	0.570
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.861	0.781

Table A.3: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_3$	$\alpha = 0.5$	$\beta = 0.5$	0.559	0.513
	$\alpha = 0.4$	$\beta = 0.5$	0.601	0.507
	$\gamma_1 = -1$	$\gamma_2 = 1$	4 it.	

Table A.4: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_4$	$\alpha = 0.5$	$\beta = 0.5$	0.608	0.472
	$\alpha = 0.6$	$\beta = 0.5$	0.481	0.441
	$\gamma_1 = -1$	$\gamma_2 = 1$	3 it.	

Table A.5: *DN method*

Example A.2 Irregular grid.

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_0$	$\alpha = 1$		0.965	0.917
	$\gamma_1 = -1$		0.973	0.934

Table A.6: *Classical alternating Schwarz*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_1$	$\alpha = 0.4$	$\beta = 0.3$	0.624	0.600
	$\gamma_1 = -1$	$\gamma_2 = 1$	divergent	

Table A.7: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_2$	$\alpha = 0.5$	$\beta = 0.5$	0.810	0.760
	$\alpha = 0.5$	$\beta = 0.7$	0.620	0.580
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.865	0.803

Table A.8: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_3$	$\alpha = 0.5$	$\beta = 0.5$	0.600	0.560
	$\alpha = 0.4$	$\beta = 0.5$	0.665	0.540
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.714	0.653

Table A.9: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_4$	$\alpha = 0.5$	$\beta = 0.5$	0.600	0.560
	$\alpha = 0.4$	$\beta = 0.5$	0.652	0.540
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.326	0.324

Table A.10: *DN method*

Example A.4 Blocks of different size and irregular grid.

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_0$	$\alpha = 1$		0.971	0.920
	$\gamma_1 = -1$		0.980	0.933

Table A.11: *Classical alternating Schwarz*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_1$	$\alpha = 0.4$	$\beta = 0.4$	0.640	0.576
	$\gamma_1 = -1$	$\gamma_2 = 1$	divergent	

Table A.12: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_2$	$\alpha = 0.5$	$\beta = 0.5$	0.781	0.673
	$\alpha = 0.7$	$\beta = 0.5$	0.617	0.613
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.743	0.675

Table A.13: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_3$	$\alpha = 0.5$	$\beta = 0.5$	0.630	0.556
	$\alpha = 0.5$	$\beta = 0.4$	0.598	0.555
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.510	0.471

Table A.14: *DN method*

	D. par.	N. par.	Error reduction factor	
$\mathbf{O}_4$	$\alpha = 0.5$	$\beta = 0.5$	0.672	0.554
	$\alpha = 0.6$	$\beta = 0.5$	0.471	0.484
	$\gamma_1 = -1$	$\gamma_2 = 1$	0.356	0.336

Table A.15: *DN method*

APPENDIX B

Preconditioners and Krylov Subspace Methods

B.1 Preconditioners

Consider a system of the form

$$\mathbf{A}\mathbf{u} = \mathbf{f}, \tag{B.1}$$

Denote the error for an approximate solution $\mathbf{u}^k$ by $\mathbf{e}^k = \mathbf{u} - \mathbf{u}^k$, then

$$\mathbf{A}\mathbf{e}^k = \mathbf{A}(\mathbf{u} - \mathbf{u}^k) = \mathbf{f} - \mathbf{A}\mathbf{u}^k = \mathbf{r}^k, \tag{B.2}$$

where $\mathbf{r}^k$ is called the residual. If having an approximate solver $\mathbf{B}$ for above error equation, then

$$\mathbf{e}^k \approx \mathbf{B}\mathbf{r}^k \tag{B.3}$$

can be used in an iterative correcting procedure of the form

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \mathbf{e}^k = \mathbf{u}^k + \mathbf{B}(\mathbf{f} - \mathbf{A}\mathbf{u}^k), \tag{B.4}$$

giving a new (and hopefully better) approximate solution. The matrix $\mathbf{B}$ (or sometimes its inverse) is called a preconditioner.

B.2 Krylov Subspace Method

Consider again the system (B.1). Given an initial vector $\mathbf{u}_0$, the corresponding residual is $\mathbf{r}_0 = \mathbf{f} - \mathbf{A}\mathbf{u}_0$. Then system (B.1) can be written as

$$\mathbf{A}(\mathbf{u} - \mathbf{u}_0) = \mathbf{r}_0. \tag{B.5}$$

The Krylov subspaces based on $\mathbf{A}$ and $\mathbf{u}_0$ is defined as

$$\mathbf{K}_k = \text{span}\left\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{k-1}\mathbf{r}_0\right\}. \tag{B.6}$$

Let the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ be a basis for $\mathbf{K}_k$. That is the vectors $\mathbf{v}_i$ are linearly independent and they span the kth Krylov subspace. Set $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$. We seek an approximation to the exact solution $\mathbf{u}$ on the form

$$\mathbf{u}_k = \mathbf{u}_0 + c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k = \mathbf{u}_0 + \mathbf{V}_k\mathbf{c}_k. \tag{B.7}$$

Unless by chance $\mathbf{u} - \mathbf{u}_0 \in \mathbf{K}_k$, no such approximation satisfies system (B.5), i.e. makes

$$\mathbf{A}(\mathbf{u}_k - \mathbf{u}_0) = \mathbf{r}_0, \tag{B.8}$$

since $\mathbf{u}_k$ is only a projection of the exact solution to a k dimensional subspace. The idea is now to project the system (B.8) into a kth dimensional subspace, where the system can be solved.

Therefor, define yet another subspace of dimension k having the basis $\mathbf{w}_1, \ldots, \mathbf{w}_k$, and set $\mathbf{W}_k = [\mathbf{w}_1, \ldots, \mathbf{w}_k]$. Project system (B.8) to the subspace spanned by $\mathbf{W}_k$ and use the definition of the approximation (B.7);

$$\mathbf{W}_k^T\mathbf{A}(\mathbf{V}_k\mathbf{c}_k) = \mathbf{W}_k^T\mathbf{r}_0. \tag{B.9}$$

Solve this system for $\mathbf{c}_k$ and set the approximation

$$\mathbf{u}_k = \mathbf{u}_0 + \mathbf{V}_k\mathbf{c}_k. \tag{B.10}$$

The projection approach above is only possible if $\mathbf{W}_k^T\mathbf{A}\mathbf{V}_k$ is nonsingular. Fortunately [Bark92] shows that it is not only nonsingular but also symmetric positive definite (SPD), if choosing

- $\mathbf{W}_k = \mathbf{V}_k$ when $\mathbf{A}$ is SPD,
- $\mathbf{W}_k = \mathbf{A}\mathbf{V}_k$ when $\mathbf{A}$ is nonsingular.

Suppose for now that

$$\mathbf{w}_i^T\mathbf{A}\mathbf{v}_j \begin{cases} = 0 & \text{if } i \neq j \\ \neq 0 & \text{if } i = j \end{cases}, \tag{B.11}$$

then the two sets $\mathbf{w}_i$ and $\mathbf{v}_i$ are said to be orthogonal with respect to $\mathbf{A}$. With this assumption the matrix $\mathbf{W}_k^T\mathbf{A}\mathbf{V}_k$ is a diagonal matrix, and the solution to the system (B.9) is given component wise as

$$c_i = \frac{\mathbf{w}_i^T\mathbf{r}_0}{\mathbf{w}_i^T\mathbf{A}\mathbf{v}_i}, \quad i = 1, \ldots, k. \tag{B.12}$$

If the two sets do not satisfy assumption (B.11), then just change to another basis. Simply compute two new basis's $\tilde{\mathbf{v}}_i$ and $\tilde{\mathbf{w}}_i$ spanning $\mathbf{V}_k$ and $\mathbf{W}_k$ respectively, which do satisfy the assumption. This is an orthogonalization procedure with assumption (B.11) as the orthogonal condition, and one way to build the new sets is by using a Gram Schmidt process.

This have brought us in a position to be able to sketch how a typical Krylov subspace method works: Provide an initial solution $\mathbf{u}_0$, and decide how $\mathbf{V}_k$ and $\mathbf{W}_k$ should relate to $\mathbf{K}_k$. While the approximative solution is not precise enough, do:

- Expand the Krylov subspace by one dimension.
- Create basis's for $\mathbf{V}_k$ and $\mathbf{W}_k$ which are orthogonal with respect to $\mathbf{A}$, using an orthogonalization process.
- Solve the diagonal system (B.9), and compute a new approximate solution (B.10).

Note especially that if n is the dimension of the system, then $\mathbf{K}_n$ spans the entire solution space, hence the the exact solution is guaranteed after at most n iterations.

We will now look at the work required to compute the next approximate solution.

- To expand the Krylov subspace, it is necessary to compute one matrix vector product, namely $\mathbf{A}(\mathbf{A}^{k-1}\mathbf{r_0})$.
- The orthogonalization procedure is based on inner products. The procedure needs to compute one inner product for each $\mathbf{v}_i$ and $\mathbf{w}_i$. Since the set of $\mathbf{v}_i$ and $\mathbf{w}_i$ grows with the dimension of the Krylov subspace, the work increases similarly. However, if $\mathbf{A}$ is SPD, then the orthogonalization of $\mathbf{K}_k$ can reuse computations from the orthogonalization of $\mathbf{K}_{k-1}$. Furthermore $\mathbf{v}_i = \mathbf{w}_i$, hence the procedure needs only to compute one inner product, and there is no need to save the $\mathbf{v}_i$ vectors [Bark92].
  If $\mathbf{A}$ is not SPD, the vectors $\mathbf{v}_i$ and $\mathbf{w}_i$ are usually created such that only one inner product for each pair is necessary, that is k inner products.
- The diagonal elements of the diagonal system must be computed. This can be done at the cost of one inner product per diagonal element, so again the work increases with the dimension of the Krylov subspace. If again $\mathbf{A}$ is SPD, the diagonal elements from the previous iteration can be reused, so only one new element must be calculated, at the cost of one inner product.
  Finally an update from the previous approximative solution to the new is needed.

A method that implement the procedure for a SPD system, is the Conjugate Gradient (CG) method. It uses for each iteration; one matrix-vector product, 2 inner products and 3 vector updates.

Implementations for non SPD systems usually suffer from the fact that the number of inner products grows with the dimension of the subspace. The method GMRES (Generalized Minimum RESidual) uses for each iteration; one matrix-vector product, $k+1$ scalar products, k saxpy operations,[1] a vector scaling, and a usually small dimensional least squares problem [Hanke00].

Most non SPD implementations exist in restarted or truncated versions. Restarted, meaning that at some point the approximative solution $\mathbf{u}_k$ is used as a new start guess $\mathbf{u}_0$ and the procedure is started

[1] A saxpy reads; "scalar a times $\mathbf{x}$ plus $\mathbf{y}$", $(\mathbf{w} = a\mathbf{x} + \mathbf{y})$

from scratch. Truncated, meaning that only a fixed number of the latest $\mathbf{v}_k$ is saved and processed. Convergence is usually no longer guaranteed for restarted and truncated versions, but they usually perform well.

Preconditioned Krylov Methods A Krylov subspace method can be applied in conjunction with a preconditioner. Consider what is called the preconditioned system

$$\mathbf{BAu} = \mathbf{Bf}, \tag{B.13}$$

which has the same solution as (B.1). Instead apply a Krylov subspace method to the preconditioned system. This can be done without explicitly forming the product $\mathbf{BA}$.

Using a preconditioner may improve convergence speed greatly, depending on the preconditioner used. Convergence speed is closely related to the condition number of the system.[2] If a system converges very slowly, the condition number is big. Now, $\mathbf{B}$ is an approximative solver, so $\mathbf{B} \approx \mathbf{A}^{-1}$, hence $\mathbf{BA}$ is "close" to identity. The identity matrix has unity eigenvalues and therefor a condition number of one. However, $\mathbf{BA}$ is usually not close to identity, but the preconditioning can make the condition number much smaller than that of the original system, and hence improve convergence.

Let us sum up: For a Krylov subspace method to work, it is required to provide procedures:

- to compute the product $\mathbf{v} = \mathbf{Au}$ to obtain the residual.
- to compute an approximate solution $\mathbf{v} = \mathbf{Bu}$

It is shown in [Bark92] that good preconditioners have the same properties as good splittings, $\mathbf{A} = \mathbf{M} + \mathbf{N}$, used in stationary iterative methods. Therefor, $\mathbf{B} = \mathbf{M}^{-1}$ is usually a good choice of a preconditioner.

[2] The factor between the smallest and the largest singular value of the system.

Krylov Type Iterative Methods There exist many Krylov type iterative methods, a haste look in the literature have produced the following list. Methods for general system include: GCR, GMRES, FOM, ORTHOMIN, CGNR, CGNE, CGLS, LSQR, BiCG, BiCGStab, QMR, TFQMR, CGS. For symmetric but not necessarily positive definite systems the methods MINRES, SYMMLQ, can be used, while for symmetric positive definite systems there seems to be only one choice, the CG method. All of the methods may be used in conjunction with a preconditioner. This is just methods mentioned in [Bark92] and [Hanke00], many more exist.

Why so many different methods? For efficiency, meaning the time it takes to obtain a satisfactory solution. The methods differ for example in their memory requirement, work per iteration, orthogonalization procedure, whether the transpose are available, and in their numerical stability. Having for example a matrix with almost nonorthogonal columns, rounding errors may cause one method to break down, while another method may be less sensitive and produce a satisfactory solution.

One should keep in mind that all methods have their strengths and drawbacks, so none is preferable to another in the general case - which method to use depends entirely on the problem to solve.

For references on Krylov subspace methods, see [Bark92] or [Hanke00].

APPENDIX C

DN Method Proofs

This appendix presents two examples, which proof that 2 iteration convergence can be achieved for the DN method. The first example is for the one dimensional case, while the second is in any dimension.

The examples are referenced from Section 5.4.

C.1 Proof for 1D

Example C.1: 2 Iteration Solution to 1D Poisson Problem The proof is based on finding eigenvalues of the iteration matrix for a double step including both a Dirichlet and a Neumann step.

First find the iteration matrix $\mathbf{G}$ for a single step, the inverse of $\mathbf{M}$ from Equation (5.38d) is needed,

$$\mathbf{M}^{-1} = \begin{bmatrix} \tilde{\mathbf{B}}_1^{-1} & \mathbf{0} & \tilde{\mathbf{B}}_1^{-1}\mathbf{C}_x & -\tilde{\mathbf{B}}_1^{-1}\gamma\tilde{\mathbf{I}}_x^T \\ \mathbf{0} & \tilde{\mathbf{B}}_2^{-1} & -\tilde{\mathbf{B}}_2^{-1}\gamma\tilde{\mathbf{I}}_y^T & \tilde{\mathbf{B}}_2^{-1}\mathbf{C}_y \\ \mathbf{0} & \mathbf{0} & -\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{bmatrix}. \tag{C.1}$$

This will produce the iteration matrix

$$\mathbf{G} = \mathbf{M}^{-1}\mathbf{N} = \begin{bmatrix} -\tilde{\mathbf{B}}_1^{-1}\gamma\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x & \tilde{\mathbf{B}}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y & 0 & 0 \\ \tilde{\mathbf{B}}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x & -\tilde{\mathbf{B}}_2^{-1}\gamma\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y & 0 & 0 \\ 0 & -\tilde{\mathbf{I}}_y & 0 & 0 \\ -\tilde{\mathbf{I}}_x & 0 & 0 & 0 \end{bmatrix}. \tag{C.2}$$

Let $\mathbf{G}_{\gamma_d}$ denote the iteration matrix for the odd iterations using γ_d, and $\mathbf{G}_{\gamma_n}$ denote the iteration matrix for the even iterations using γ_n. The iteration matrix for a double step is the product of the iteration matrices for the two individual steps,

$$\mathbf{G}_{\gamma_d}\mathbf{G}_{\gamma_n} = \begin{bmatrix} \mathbf{D}_1^{-1}\gamma_d\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x\mathbf{N}_1^{-1}\gamma_n\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x + \mathbf{D}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y\mathbf{N}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x & & & \\ -\mathbf{D}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x\mathbf{N}_1^{-1}\gamma_n\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x - \mathbf{D}_2^{-1}\gamma_n\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y\mathbf{N}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x & & & \\ \text{xx} & & & \\ \text{xx} & & & \end{bmatrix}$$
$$\begin{bmatrix} -\mathbf{D}_1^{-1}\gamma_d\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x\mathbf{N}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y - \mathbf{D}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y\mathbf{N}_2^{-1}\gamma_n\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y & 0 & 0 \\ \mathbf{D}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x\mathbf{N}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y^T + \mathbf{D}_2^{-1}\gamma_d\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y\mathbf{N}_2^{-1}\gamma_n\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y & 0 & 0 \\ \text{xx} & 0 & 0 \\ \text{xx} & 0 & 0 \end{bmatrix}, \tag{C.3}$$

where $\mathbf{D}_i^{-1} = \tilde{\mathbf{B}}_i^{-1}$ from $\mathbf{G}_{\gamma_d}$ and $\mathbf{N}_i^{-1} = \tilde{\mathbf{B}}_i^{-1}$ from $\mathbf{G}_{\gamma_n}$. The $\mathbf{C}_x$ and $\tilde{\mathbf{I}}_x$ are alike in both iteration matrices and do not need to be distinguished. Details about the last rows are omitted, since in an eigenvalue analysis they give no contribution due to the corresponding zero columns, see Appendix D.4 for proof.

The two step iteration matrix (C.3) is general, but to continue, some assumptions are needed: A 1D Poisson with Dirichlet boundary conditions, discretized using a regular grid, and decomposed in two equally big blocks. Because of the regular grid and symmetry around the interface, then the following holds: $\mathbf{C}_i = \tilde{\mathbf{I}}_i^T$. The $\mathbf{C}$ have only one column and $\tilde{\mathbf{I}}$ only one row both with one element of unity on either first or last position. The matrices $\tilde{\mathbf{B}}_1$ and $\tilde{\mathbf{B}}_2$ are opposite matrices,[1] and similar are the pairs $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{N}_1$, $\mathbf{N}_2$ and their inverse.

4 nonzero blocks of the two step iteration matrix (C.3) need to be calculated. Taken bit by bit:

[1] See definition of opposite matrices in Appendix D.3

- Each product of the form $\mathbf{D}^{-1}\mathbf{C}$ or $\mathbf{D}^{-1}\tilde{\mathbf{I}}^T$ produces a matrix with only one column by picking out a specific column of $\mathbf{D}^{-1}$.
- Then the product $(\mathbf{D}^{-1}\mathbf{C})\tilde{\mathbf{I}}$ places this column at a specific column in a new matrix of the same size as $\mathbf{D}^{-1}$, while all other columns are zero columns.
- Furthermore two matrices having only one column as described above are multiplied together,
- and finally added with a another matrix product.

The procedure for each of the 4 blocks are pinned out in Equations (C.5), where $\mathbf{d}_j^i$ is the j'th column of $\mathbf{D}_i^{-1}$, and $d_{j,k}^i$ is the k'th element of $\mathbf{d}_j^i$ and similar for $\mathbf{N}_i^{-1}$, $\mathbf{n}_j^i$ and $n_{j,k}^i$.

$$\mathbf{D}_1^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{d}_n^1\end{bmatrix} \quad \mathbf{N}_1^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{n}_n^1\end{bmatrix} \rightarrow \quad \gamma_d\gamma_n \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,n_{n,n}^1\mathbf{d}_n^1\end{bmatrix} \tag{C.4a}$$

$$\mathbf{D}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{d}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \quad \mathbf{N}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{n}_1^2\end{bmatrix} \rightarrow \quad \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,n_{1,1}^2\mathbf{d}_n^1\end{bmatrix} \tag{C.4b}$$

$$\mathbf{D}_1^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{d}_n^1\end{bmatrix} \quad \mathbf{N}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{n}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \rightarrow \quad -\gamma_d \begin{bmatrix}n_{n,n}^1\mathbf{d}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \tag{C.4c}$$

$$\mathbf{D}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{d}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \quad \mathbf{N}_2^{-1}\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y = \begin{bmatrix}n_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \rightarrow \quad -\gamma_n \begin{bmatrix}n_{1,1}^2\mathbf{d}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \tag{C.4d}$$

$$\mathbf{D}_2^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{d}_1^2\end{bmatrix} \quad \mathbf{N}_1^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{n}_n^1\end{bmatrix} \rightarrow \quad -\gamma_n \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,n_{n,n}^1\mathbf{d}_1^2\end{bmatrix} \tag{C.4e}$$

$$\mathbf{D}_2^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{d}_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \quad \mathbf{N}_2^{-1}\mathbf{C}_y\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{n}_1^2\end{bmatrix} \rightarrow \quad -\gamma_d \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,n_{1,1}^2\mathbf{d}_1^2\end{bmatrix} \tag{C.4f}$$

$$\mathbf{D}_2^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \begin{bmatrix}\mathbf{0}\cdots\mathbf{0}\,\mathbf{d}_1^2\end{bmatrix} \quad \mathbf{N}_1^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{n}_n^1\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \rightarrow \quad \begin{bmatrix}n_{n,n}^1\mathbf{d}_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \tag{C.4g}$$

$$\mathbf{D}_2^{-1}\mathbf{C}_x\tilde{\mathbf{I}}_y = \begin{bmatrix}\mathbf{d}_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \quad \mathbf{N}_2^{-1}\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y = \begin{bmatrix}n_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \rightarrow \quad \gamma_d\gamma_n \begin{bmatrix}n_{1,1}^2\mathbf{d}_1^2\,\mathbf{0}\cdots\mathbf{0}\end{bmatrix} \tag{C.4h}$$

Putting this together will produce the two stop iteration matrix

$$\mathbf{G}_{\gamma_d}\mathbf{G}_{\gamma_n} = \begin{bmatrix} \mathbf{0} & (\gamma_d\gamma_n n_{n,n}^1 + n_{1,1}^2)\mathbf{d}_n^1 & (-\gamma_d n_{n,n}^1 - \gamma_n n_{1,1}^2)\mathbf{d}_n^1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & (-\gamma_n n_{n,n}^1 - \gamma_d n_{1,1}^2)\mathbf{d}_1^2 & (n_{n,n}^1 + \gamma_d\gamma_n n_{1,1}^2)\mathbf{d}_1^2 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & x & x & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & x & x & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{C.5}$$

There are only nonzeros in the two columns corresponding to cells at the interface. All eigenvalues except two will therefor be zero. The last two eigenvalues will depend only on the 2×2 block around the diagonal,

$$\begin{bmatrix} (\gamma_d\gamma_n n_{n,n}^1 + n_{1,1}^2)d_{n,n}^1 & (-\gamma_d n_{n,n}^1 - \gamma_n n_{1,1}^2)d_{n,n}^1 \\ (-\gamma_n n_{n,n}^1 - \gamma_d n_{1,1}^2)d_{1,1}^2 & (n_{n,n}^1 + \gamma_d\gamma_n n_{1,1}^2)d_{1,1}^2 \end{bmatrix}. \tag{C.6}$$

The eigenvalues depends on one element of each of the $\mathbf{D}^i$ and $\mathbf{N}^i$ matrices: $n_{n,n}^1$, $n_{1,1}^2$, $d_{n,n}^1$ and $d_{1,1}^2$. Eigenvalues for a 2×2 system can be determined by the equation

$$0 = \lambda^2 - \tau\lambda + \Delta, \tag{C.7}$$

where τ is the trace and Δ the determinant of the 2×2 block. This will have zero eigenvalues if the trace and determinant are both zero. Remember that $\mathbf{D}^1$ and $\mathbf{D}^2$ are opposite and so also $\mathbf{N}^1$ and $\mathbf{N}^2$, hence $n_{n,n}^1 = n_{1,1}^2 = n$ and $d_{n,n}^1 = d_{1,1}^2 = d$. The trace and determinant can therefor be expressed as

$$\tau = 2(\gamma_d\gamma_n + 1)nd = 0 \tag{C.8}$$

$$\begin{aligned}\Delta &= (\gamma_d\gamma_n + 1)^2 n^2 d^2 - (-\gamma_d - \gamma_n)^2 n^2 d^2 \\ &= (\gamma_d\gamma_n - \gamma_d - \gamma_n + 1)^2 n^2 d^2 \\ &= (\gamma_d - 1)(\gamma_n - 1)n^2 d^2 = 0.\end{aligned} \tag{C.9}$$

This system has two zero solutions, namely for

$$\gamma_d = 1, \quad \gamma_n = -1 \qquad \text{or} \qquad \gamma_d = -1, \quad \gamma_n = 1. \tag{C.10}$$

This corresponds exactly to one Neumann step followed by one Dirichlet step. Whether the first step is the Neumann or the Dirichlet does not matter.

Note that in this 1D case, the only demand for this to work, is if $n_{n,n}^1 = n_{1,1}^2$ and $d_{n,n}^1 = d_{1,1}^2$. This might be accomplished with other assumptions than

those used here. E.g. since the solution on the interface does not depend on how the inner of each block is discretized, then the inner block discretization do not matter. We have experimentally verified this by stretching a grid on the first $\frac{1}{4}$ of the interval, and still only 2 iterations is needed.

If $n^1_{n,n} \neq n^2_{1,1}$ and $d^1_{n,n} \neq d^2_{1,1}$, then this ends up as a two parameter optimization problem, which in general is very difficult to solve. Especially since $d^1_{n,n}$ and $d^2_{1,1}$ depends on γ_d, and $n^1_{n,n}$ and $n^2_{1,1}$ depends on γ_n.

This proof is inspired by [Hadj00]. There are however some differences in the two proofs: In [Hadj00] the systems for the Dirichlet and Neumann step are created independently, and not from the same system as we have done. This make it possible for them to implement the exchange of data with a weighting between each block much like the updating (5.37). Furthermore the values of $d^1_{n,n}$, $d^2_{1,1}$, $n^1_{n,n}$, and $n^2_{1,1}$ are calculated explicitly as a function of two weighting parameters. Thereby it is possible to find the trace and determinant as a function of the weighting parameters, and minimize the eigenvalues, which they generalize to any dimension including blocks of different size. Similar weighting parameters are not included this formulation.

End of Example C.1.

C.2 Proof for Any Dimension

Example C.2: 2 Iteration Solution to Poisson Problem in Any Dimension Consider a Poisson operator with pure Dirichlet boundary conditions on a domain in any dimension of size $[0;1]^n$.

Assume the grid is regular, that is orthogonal with constant grid spacing in all directions. Number the cells in a natural way. The matrix of the resulting algebraic system will by symmetric due to the regular grid. Split the domain into two equally big blocks by halfing in one dimension, splitting the solution vector in two equally big vectors $\mathbf{x}$ and $\mathbf{y}$. This time we will number the cells such that cells at the interface are "in the middle"

$$\mathbf{u} = \begin{bmatrix} \mathbf{x}_i & \mathbf{x}_b & \mathbf{y}_b & \mathbf{y}_i \end{bmatrix}^T, \tag{C.11}$$

where subscript i indicate interior cells and b cells at the interface.

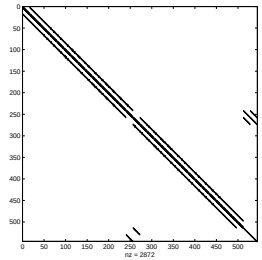


Figure C.1: *Spy of matrix*

Split up the system as in Equation (5.32) and proceed as in Section 5.4. The iteration matrix (5.38c), I will rewrite here:

$$\begin{bmatrix} \tilde{\mathbf{B}}_1 & \mathbf{0} & \mathbf{C}_x & -\gamma\tilde{\mathbf{I}}_x^T \\ \mathbf{0} & \tilde{\mathbf{B}}_2 & -\gamma\tilde{\mathbf{I}}_y^T & \mathbf{C}_y \\ \mathbf{0} & \tilde{\mathbf{I}}_y & -\mathbf{I} & \mathbf{0} \\ \tilde{\mathbf{I}}_x & \mathbf{0} & \mathbf{0} & -\mathbf{I} \end{bmatrix} \tag{5.38c}$$

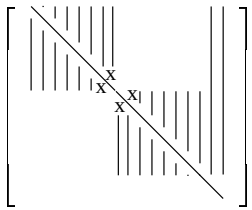
Due to the regular grid and symmetry around the interface the matrices have a lot of structure: $\mathbf{C}_x = \tilde{\mathbf{I}}_x^T$, $\tilde{\mathbf{B}}_1$ and $\tilde{\mathbf{B}}_2$ are opposite as defined in Section D.3 and symmetric. An example of a spy of the matrix (showing nonzero elements) can be seen in Figure C.1. The splitting into $\mathbf{M}$ and $\mathbf{N}$ will follow Section 5.4 and Equation (5.38d) and (5.38d). The inverse of $\mathbf{M}$ is presented in Equation (C.1). Let us take a look at its structure in this case.

$$\mathbf{M}^{-1} = \begin{bmatrix} \ddots & & & \\ & \mathbf{x} & \mathbf{y} & \\ & \mathbf{x} & \mathbf{y} & \\ & & & \ddots \end{bmatrix} \tag{C.12}$$

The two block matrices are opposite, and furthermore they have the structure

such that

$$\mathbf{M}^{-1} = \left[\begin{array}{c} \\ \\ \\ \end{array}\right] \tag{C.13}$$



This structure is a property of the regular grid and symmetry around the interface, and this is the vital assumption for this proof. To get the iteration matrix, $\mathbf{M}$ is multiplied by $\mathbf{N}$ to give the iteration matrix (C.2), which I will rewrite here, replacing $\mathbf{C}_x = \tilde{\mathbf{I}}_x^T$.

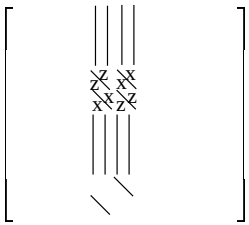
$$\mathbf{G} = \mathbf{M}^{-1}\mathbf{N} = \left[\begin{array}{cc} -\tilde{\mathbf{B}}_1^{-1}\gamma\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x & \tilde{\mathbf{B}}_1^{-1}\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_y \\ \tilde{\mathbf{B}}_2^{-1}\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_x & -\tilde{\mathbf{B}}_2^{-1}\gamma\tilde{\mathbf{I}}_y^T\tilde{\mathbf{I}}_y \\ & -\tilde{\mathbf{I}}_y \\ -\tilde{\mathbf{I}}_x & \end{array}\right] \tag{C.14}$$

Consider the products on the form $\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x$. Let us take a look at two of the combinations

$$\tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_x = \left[\begin{array}{cc} & \\ & \end{array}\right] \qquad \tilde{\mathbf{I}}_x^T\tilde{\mathbf{I}}_y = \left[\begin{array}{cc} & \\ & \end{array}\right], \tag{C.15}$$

all nonzero values have the value one. Such matrices pick out certain columns of $\tilde{\mathbf{B}}_i^{-1}$ and places these columns in another column. The result is an iteration matrix $\mathbf{G}$ of the form

$$\mathbf{G} = \left[\begin{array}{cc} & \\ & \\ & \end{array}\right]. \tag{C.16}$$



Why now the z's? The iteration matrix (C.14) include a γ parameter, so that $\mathbb{Z} = -\gamma\mathbb{X}$.

Now it is time to consider two consecutive steps using two different values of γ: The Dirichlet step uses $\gamma = -1$ and the Neumann $\gamma = 1$. Consider the middle part of $\mathbf{G}$. If setting $\mathbf{S} = \mathbb{X}$ in the Dirichlet step and similar $\mathbf{T} = \mathbb{X}$[2] in the Neumann step, then the center part of the iteration matrix for the two steps have the form

$$\hat{\mathbf{G}}_D = \left[\begin{array}{cc} \mathbf{S} & \mathbf{S} \\ \mathbf{S} & \mathbf{S} \end{array}\right] \qquad \hat{\mathbf{G}}_N = \left[\begin{array}{cc} -\mathbf{T} & \mathbf{T} \\ \mathbf{T} & -\mathbf{T} \end{array}\right] \tag{C.17}$$

Multiplying these two together gives exactly the zero matrix. So the product of the entire two will produce a two step iteration matrix on the form

$$\mathbf{G}_D\mathbf{G}_N = \left[\begin{array}{cc} & \\ & \\ & \end{array}\right]. \tag{C.18}$$

The matrix of this structure has both zero trace and zero determinant, and have therefor purely zero eigenvalues. Hence the exact solution is achieved after just these two iterations.

Note that this proof is based on the structure of the system matrix. This structure only appears when using a regular grid and there is symmetry around the interface. However, if that is the case, this proof holds for any dimension.

End of Example C.2.

[2] $\mathbf{S}$ and $\mathbf{T}$ are not identical since the $\tilde{\mathbf{B}}$'s depend on the γ values used.

APPENDIX D

Notes

D.1 Green's Identities

The basic tool for deriving Green's identities is the divergence theorem,

$$\int_\Omega \nabla \cdot \mathbf{F} \, \mathrm{d}\Omega = \int_{\partial\Omega} \mathbf{F} \cdot \mathbf{n} \, \mathrm{d}S, \tag{D.1}$$

where $\mathbf{n}$ is the unit outward pointing normal vector on $\partial\Omega$.

From the product rule $\nabla \cdot (v\nabla u) = \nabla v \cdot \nabla u + v\nabla^2 u$ and by use of the divergence theorem, Green's first identity (G1) arise

$$\int_{\partial\Omega} v\frac{\partial u}{\partial n} \, \mathrm{d}S, = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}\Omega + \int_\Omega v\nabla^2 u \, \mathrm{d}\Omega, \tag{D.2}$$

where $\frac{\partial u}{\partial n} = \mathbf{n} \cdot \nabla u$. This is valid for any solid region Ω and any pair of functions u and v. Set especially $v = 1$ to get

$$\int_{\partial\Omega} \frac{\partial u}{\partial n} \, \mathrm{d}S, = \int_\Omega \nabla^2 u \, \mathrm{d}\Omega. \tag{D.3}$$

The middle term in G1 does not change if u and v are interchanged. So by writing G1 for u and v and again for v and u, and subtract the two, Green's second identity (G2) arise

$$\int_\Omega u\nabla^2 v - v\nabla^2 u \,\mathrm{d}\Omega = \int_{\partial\Omega} u\frac{\partial v}{\partial n} - v\frac{\partial u}{\partial n}\,\mathrm{d}S, \tag{D.4}$$

which is valid under same circumstances as G1. See [Stra92].

D.2 About Inversion of Matrices

Lemma D.1 *The inverse of a symmetric matrix will again be symmetric.*

Proof: The definition of the inverse of a matrix $\mathbf{A}$ is a matrix $\mathbf{A}^{-1}$ that obeys

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}. \tag{D.5}$$

Transposing gives

$$\left(\mathbf{A}\mathbf{A}^{-1}\right)^T = \left(\mathbf{A}^{-1}\mathbf{A}\right)^T = \mathbf{I}, \tag{D.6}$$

$$\left(\mathbf{A}^{-1}\right)^T \mathbf{A}^T = \mathbf{A}^T \left(\mathbf{A}^{-1}\right)^T = \mathbf{I}, \tag{D.7}$$

which define the inverse of $\mathbf{A}^T$, so

$$\left(\mathbf{A}^T\right)^{-1} = \left(\mathbf{A}^{-1}\right)^T. \tag{D.8}$$

Therefore, if $\mathbf{A}$ is symmetric, $\mathbf{A} = \mathbf{A}^T$, then $\mathbf{A}^{-1} = (\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T$, and so the inverse is symmetric. □

Lemma D.2 *Permuting the rows of matrix* $\mathbf{A}$ *using a permutation matrix* $\mathbf{P}$ *corresponds to permuting the columns of* $\mathbf{A}^{-1}$ *with* $\mathbf{P}^T$.

Proof: Reordering the rows is done by multiplying $\mathbf{P}$ from the left.

$$\mathbf{B} = \mathbf{P}\mathbf{A}. \tag{D.9}$$

Use the definition of the inverse of $\mathbf{A}$, and insert the identity $\mathbf{I} = \mathbf{P}^{-1}\mathbf{P}$ in the middle.

$$\left(\mathbf{A}^{-1}\mathbf{P}^{-1}\right)(\mathbf{P}\mathbf{A}) = \mathbf{I}, \tag{D.10}$$

$$\left(\mathbf{A}^{-1}\mathbf{P}^{T}\right)(\mathbf{P}\mathbf{A}) = \mathbf{I}. \tag{D.11}$$

So the inverse of $\mathbf{B}$ is the inverse of $\mathbf{A}$ multiplied from the right with $\mathbf{P}^T$, which is a reordering of the columns:

$$\mathbf{B}^{-1} = \mathbf{A}^{-1}\mathbf{P}^T. \tag{D.12}$$

□

Consider a matrix on the form

$$\begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & -\mathbf{I} & & \\ \mathbf{A}_{3,1} & & -\mathbf{I} & \\ \mathbf{A}_{4,1} & & & -\mathbf{I} \end{bmatrix}. \tag{D.13}$$

Defining

$$\tilde{\mathbf{A}} = \mathbf{A}_{1,1} + \mathbf{A}_{1,2}\mathbf{A}_{2,1} + \mathbf{A}_{1,3}\mathbf{A}_{3,1} + \mathbf{A}_{1,4}\mathbf{A}_{4,1} \tag{D.14}$$

will give the inverse as

$$\begin{bmatrix} \tilde{\mathbf{A}}^{-1} & \tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,2} & \tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,3} & \tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,4} \\ \mathbf{A}_{2,1}\tilde{\mathbf{A}}^{-1} & -\mathbf{I} + \mathbf{A}_{2,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,2} & \mathbf{A}_{2,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,3} & \mathbf{A}_{2,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,4} \\ \mathbf{A}_{3,1}\tilde{\mathbf{A}}^{-1} & \mathbf{A}_{3,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,2} & -\mathbf{I} + \mathbf{A}_{3,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,3} & \mathbf{A}_{3,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,4} \\ \mathbf{A}_{4,1}\tilde{\mathbf{A}}^{-1} & \mathbf{A}_{4,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,2} & \mathbf{A}_{4,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,3} & -\mathbf{I} + \mathbf{A}_{4,1}\tilde{\mathbf{A}}^{-1}\mathbf{A}_{1,4} \end{bmatrix} \tag{D.15}$$

Proof: By insertion into the definition of the inverse (D.5) □

D.3 About "Opposite" Matrices

Definition D.3 (Opposite matrices) *Having a matrix*

$$\mathbf{A} = [\mathbf{a}_1 \cdots \mathbf{a}_n], \tag{D.16}$$

then the "opposite" matrix is defined as

$$\mathbf{B} = [\mathbf{b}_1 \cdots \mathbf{b}_n], \tag{D.17}$$

where $\mathbf{b}_1$ *is* $\mathbf{a}_n$ *with elements in opposite order and similar for* $\mathbf{b}_2$, $\mathbf{a}_{n-1}$ *and so on. On matrix form*

$$\mathbf{B} = \mathbf{P}_o\mathbf{A}\mathbf{P}_o, \qquad \mathbf{P}_o = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}. \tag{D.18}$$

Element wise if $\mathbf{A}$ *is of size* $n \times n$*, the matrices are opposite when*

$$a_{i,j} = b_{n-i+1,n-j+1}. \tag{D.19}$$

Lemma D.4 *If the two matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *are opposite and nonsingular, then the inverse of* $\mathbf{A}$ *will be opposite to the inverse of* $\mathbf{B}$.

Proof: Use Equation (D.18):

$$\begin{aligned} \mathbf{B}^{-1} &= (\mathbf{P}_o\mathbf{A}\mathbf{P}_o)^{-1} && \text{(D.20)} \\ &= \mathbf{P}_o^T\mathbf{A}^{-1}\mathbf{P}_o^T && \text{(D.21)} \\ &= \mathbf{P}_o\mathbf{A}^{-1}\mathbf{P}_o. && \text{(D.22)} \end{aligned}$$

□

Because of Lemma D.4, we get in Example C.1, that when $\mathbf{A}_1$ and $\mathbf{A}_2$ are opposite, then $(\mathbf{A}_2^{-1})_{1,1} = (\mathbf{A}_1^{-1})_{n,n}$, which is why it all works out fine in the example.

D.4 About Zero Columns and Eigenvalues

Lemma D.5 *If the* i*'th column (row) of a matrix* $\mathbf{A}$ *is a zero column (row), then the corresponding* i*'th row (column) does not influence the eigenvalues of the system.*

Proof: Consider a matrix with a zero column

$$\begin{bmatrix} \mathbf{A}_{11} & 0 & \mathbf{A}_{13} \\ \mathbf{a}_{21}^T & 0 & \mathbf{a}_{23}^T \\ \mathbf{A}_{31} & 0 & \mathbf{A}_{33} \end{bmatrix}. \tag{D.23}$$

If λ is an eigenvalue, then $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ or $(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$, which only have nonzero solutions when $\mathbf{A} - \lambda\mathbf{I}$ is singular. Singularity also implies the determinant $|\mathbf{A} - \lambda\mathbf{I}| = 0$. A standard procedure to find all eigenvalues is to subtract λ from the diagonal, set the determinant to zero and solve for λ,

$$\begin{vmatrix} \mathbf{A}_{11} - \lambda\mathbf{I} & 0 & \mathbf{A}_{13} \\ \mathbf{a}_{21}^T & -\lambda & \mathbf{a}_{23}^T \\ \mathbf{A}_{31} & 0 & \mathbf{A}_{33} - \lambda\mathbf{I} \end{vmatrix} = 0. \tag{D.24}$$

Following the rules of linear algebra then

$$\lambda \begin{vmatrix} \mathbf{A}_{11} - \lambda\mathbf{I} & \mathbf{A}_{13} \\ \mathbf{A}_{31} & \mathbf{A}_{33} - \lambda\mathbf{I} \end{vmatrix} = 0, \tag{D.25}$$

so every row corresponding to a zero column have no effect on any of the eigenvalues. The proof for a zero row follows same procedure. □

For reference, consult a linear algebra book, e.g. [Eisi93].

APPENDIX E

Elliptic PDE

This section will define and describe the different types of linear second order PDE's. It is mainly based on [Stra92]. We will use the notation $u_x = \partial_x u = \partial u / \partial x$.

Consider a second order PDE of the form

$$au_{xx} + bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0, \tag{E.1}$$

Consider now an equation substituting x for u_x and x^2 for u_{xx}, omitting lover order terms:

$$ax^2 + bxy + cy^2 + \cdots = 0. \tag{E.2}$$

Depending on a, b, and c, assuming they are not all zero, this will describe either an ellipse, a parabola or a hyperbola. In each of the three cases Equation (E.2) can be reduced by a coordinate transform to $x^2 + y^2 + \cdots = 0$, $x^2 + \cdots = 0$, or $x^2 - y^2 + \cdots = 0$ respectively. The same can be applied to a second order PDE, transforming the PDE into one of the canonical forms:

- $b^2 - 4ac < 0$: Elliptic, and (E.1) can be transformed to

$$u_{xx} + u_{yy} + \cdots = 0, \tag{E.3}$$

- $b^2 - 4ac = 0$: Parabolic, and (E.1) can be transformed to

$$u_{xx} + \cdots = 0, \tag{E.4}$$

- $b^2 - 4ac > 0$: Hyperbolic, and (E.1) can be transformed to

$$u_{xx} - u_{yy} + \cdots = 0, \tag{E.5}$$

This is the definition in 2D.

Let us consider the second order PDE (E.1) in a linear algebra context. The PDE using matrix notation can be written as

$$\begin{bmatrix} \partial_x & \partial_y \end{bmatrix} \begin{bmatrix} a & \frac{1}{2}b \\ \frac{1}{2}b & c \end{bmatrix} \begin{bmatrix} \partial_x \\ \partial_y \end{bmatrix} u = r(u), \tag{E.6}$$

where $r(u)$ contain all remaining lower order terms. To Transform the second order PDE into one of its canonical forms corresponds to make an eigenvalue decomposition of the coefficient matrix.

In general, a second order PDE in any dimension n can be written as

$$\begin{bmatrix} \partial_{x_1} & \cdots & \partial_{x_n} \end{bmatrix} \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} \partial_{x_1} \\ \vdots \\ \partial_{x_n} \end{bmatrix} u = r(u). \tag{E.7}$$

Call the matrix $\mathbf{A}$, the following defines the type of the PDE:

- The PDE is elliptic, if all eigenvalues of $\mathbf{A}$ have the same sign. Then $\mathbf{A}$ is positive (or negative) definite.
- The PDE is parabolic if one eigenvalue is zero, and the rest have the same sign.
- The PDE is hyperbolic if one eigenvalue have opposite sign of all the others, but none are zero. If there are at least two positive and two negative, the PDE is sometimes called ultra-hyperbolic.

If the coefficient matrix $\mathbf{A}$ depends on the position $\mathbf{x}$, then a PDE may be elliptic in some regions, parabolic in others, and hyperbolic somewhere else.

The different types of PDEs have quite different behavior, which can be described by their characteristic lines. The characteristic lines are said to limit the domain of influence: If the characteristic lines are drawn through a point (x_0, y_0), then the solution in that point can only influence the solution at other points within the characteristic lines, the gray region in Figure E.1. Often, the y-axis is the time axis. Note

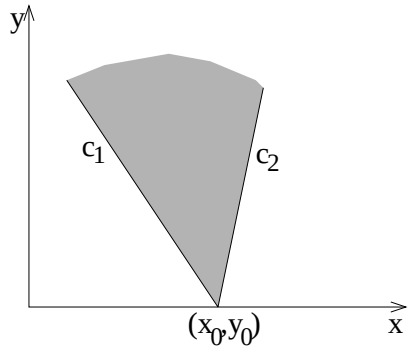


Figure E.1: *Domain of influence*

that the lines may be curves in some cases. The characteristic lines set the speed of propagation of the system.

Returning to the two dimensional case, then the characteristic lines are defined as

$$\frac{\partial y}{\partial x} = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{E.8}$$

- If now the PDE is hyperbolic, $b^2 - 4ac > 0$, then two characteristic lines exist, which is exactly the case in Figure E.1.
- If however the PDE is parabolic, $b^2 - 4ac = 0$, then there is only on characteristic line. For simplicity, assume that also $b = 0$ (otherwise transform first to one of the canonical forms), then the characteristic line is horizontal. The domain of influence have become the entire half plane above the line, the speed of propa-

gation has become infinite. Any forcing at (x_0, y_0) will influence the solution in any point (x, y) for $y > y_0$.

- If finally the PDE is elliptic, $b^2 - 4ac < 0$, there are no real characteristic lines. Still the speed of propagation is infinite, but now the domain of influence is the entire domain.

E.1 Examples of the Different PDE Types

Hyperbolic. The wave equation is hyperbolic. If we fire a canon, the sound will expand in spheres with a finite speed and reach the listener nearby almost immediately, while a listener (not too) far away will only hear the sound seconds later. The speed of propagation is limited by the speed of sound.

Parabolic. An example is the diffusion equation, which describes how e.g. heat is transported within some media.

If at a specific time t_0 at one point, a rod is heated at the middle, that will have an immediate effect on the temperature at the entire rod. The effect is only from t_0 and forward, not for time before t_0.

Elliptic. This is for example the Laplace or Poisson equation. It often is used to describe a steady state, for example the steady state form of an elastic surface fixed at the boundary, e.g. a soap film.

If at one point some extra weight is added to the surface, or some points at the fixed boundary is moved down, this will force down the surface at all points.

System including time as one of the dimensions are never elliptic, since we cannot change the past. Such systems can at most be parabolic.

APPENDIX F

Implementation

This appendix will list two Matlab functions, the two basic steps of the BDD method: The matrix vector product (6.12), and the BDD preconditioner in equations (6.21).

Also a list of Matlab files, that have been used throughout the project is given, with a short description of the use. This is mostly for my own reference.

F.1 Matrix vector product, Su

```
function Su=Sufun(u_B,A_II,A_IB,A_BI,A_BB,A_i,Rt,D,A0,Rt0,n_block,Rs)

Su = zeros(size(u_B));
for i=1:n_block
  Su = Su + Rt{i}*(A_BB{i}*(Rt{i}'*u_B));
  Su = Su - Rt{i}*(A_BI{i} * ( A_II{i} \ (A_IB{i}*(Rt{i}'*u_B)) ));
end
```

F.2 BDD preconditioner

```
function u_B=MSufun(g,A_II,A_IB,A_BI,A_BB,A_i,Rt,D,A0,Q,n_block,Rs)

u_B = zeros(size(g));
gamma_u_B = [];

s = g;
if size(A0,1)
  u01 = A0 \ ( Q'*s );
  gamma_u_B = full(Q * ( u01 ) );
  s = g - Sufun(gamma_u_B,A_II,A_IB,A_BI,A_BB,A_i,Rt,D,A0,Q,n_block);
end
s = D*s;
for i = 1:n_block
  rhs_i = (Rs{i}*([zeros(size(A_II{i},2),1) ; Rt{i}'*s]));
  u_i = A_i{i} \ rhs_i;
  u_B = u_B + Rt{i}*u_i( size(A_II{i},2)+1 : end );
end
u_B = D*u_B;

if size(A0,1)
  r = g - Sufun(u_B,A_II,A_IB,A_BI,A_BB,A_i,Rt,D,A0,Q,n_block);
  u02 = A0 \ ( Q'*r );
  u_B = u_B + Q * ( u02 );
end
```

F.3 Matlab files

This section give an overview over the files that have been made. They are ordered by importance to the project.

dumpread.m Script to read in files from NS3. This script decide which data to read in, and returns a right hand side and the system matrix. Uses function and scripts in the `readdumpdir` directory.

dn.m Script that implements the DN method, as well the approximative as the exact.

bdd.m Script that implements the BDD method.

MSufun.m Function that implement the BDD preconditioner. This function is given as argument to GMRES.

Sufun.m Function that implement the matrix-vector product $\mathbf{Su}$. This function is given as argument to GMRES.

balancing.m Function to balance a vector using a coarse grid operator $\mathbf{A}_0$ and a restriction operator $\mathbf{Q}$.

blk_ize.m Function to add extra variables representing shadow layer and create a block diagonal matrix for the DN method, as in Equation (5.26). Used by `dn.m`.

blk_ize_r.m Function to add extra variables representing shadow layer and create a block diagonal matrix for the DN method, as in Equation (5.27).

blk_ize_schur_av.m Function to add extra variables representing shadow layer and create Schur complement matrices, as presented in the section on Schur complement methods. Used by `bdd.m`

P_cr.m Function to permute from one 2D checkerboard decomposition to another. Used to produce different checkerboard sizes for the tests in Section 7.3.2.

checkerboard.m Function to create a 2D checkerboard permutation matrix: The matrix permute from a natural ordering to a $n \times m$ block setup. Used by `P_cr.m`.

plotgrid.m Script to plot the grid which is read from NS3. Plot is only 2D, projecting the last dimension.

blk_unroll.m Function to create a solution matrix from a solution vector. The solution matrix can be used in plot commands in Matlab, like `mesh`. Works only for quadrangular domains in 2D.

comm_celim.m Function to perform a special kind of Gaussian elimination of the $\begin{bmatrix} \mathbf{C} & -\mathbf{D} \end{bmatrix}^T$ part of Equation (5.21) to produce a kind of identity matrix on the $\mathbf{C}$ part.

comm_cbelim.m Function alike the comm_celim.m apart from it does only work on the neighbour shadow variables, not its own.

comm_rbelim.m Function to perform a special kind of Gaussian elimination of the $\begin{bmatrix} \mathbf{R} & -\mathbf{D} \end{bmatrix}^T$ part of Equation (5.21) to produce a identity matrix on the $\mathbf{D}$ part.

alt_schwarz.m Script that have produced results in Example 4.2. Shows importance of overlap for alternating Schwarz methods. Both additive / multiplicative, and Krylov subspace method GMRES(10) is tested.

DN_eigvals.m Script that compute the eigenvalues of iteration matrices for each step of the DN method. The script have produced

Figure 5.5.

conv_overlap_2blk.m Script to show how fast a 2 block domain decomposition solver communicating only Dirichlet data converges depending on the overlap of the domains. Results from this are not presented, since they are out-competed by the alt_schwarz.m.

schur_eigvec_proj.m Script to produce results for Example 6.1

Finally there are the directory `readdumpdir`, which contains files that read NS3 datafiles into Matlab arrays. Function and scripts in this directory are used by `dumpread.m`, and provided by Stefan Mayer.

The Gaussian elimination function are not presented anywhere in the report, since they have not been to much practical use. They are implemented using (optional) pivoting, and (optimal) norming of the diagonal element, with some warnings in singular or close to singular cases.

Bibliography

[Ande95] John D. Anderson, JR., *Computational Fluid Dynamics, The Basics with Applications*, McGraw-Hill Book Co, 1995.

[Axel84] O. Axelsson, V.A. Barker, *Finite Element Solution of Boundary Value Problems, theory and computation*, Academic Press Inc, 1984

[Bark92] V. A. Barker, *H. 62, Iterative Methods for Sparse Systems of Linear Equations*, IMM, Technical University of Denmark, 1992.

[Brak98] E. Brakkee, C. Vuik, P. Wesseling, *Domain Decomposition for the Incompressible Navier-Stokes Equations: Solving subdomain problems accurately and inaccurately*, Int. J. Numer. Meth. Fluids 26, p 1217-1237, 1998.

[Bren94] Susanne C. Brenner, L. Ridgway Scott, *The Mathematical Theory of Finite Element Methods*, Springer, 1994

[Brig87] William L. Briggs, *A Multigrid Tutorial*, SIAM, 1987.

[Chan94] Tony F. Chan, Tarek P. Mathew, *Domain Decomposition Algorithms*, Acta Numerica p. 61-143, 1994.

[Chan95] Tony F. Chan, Jian Ping Shao, *Parallel Complexity of Domain Decomposition Methods and Optimal Coarse Grid Size*, Parallel Computing vol. 21 p. 1033-1049, 1995.

[Doug97] J. Douglas JR, C.-S. Huang, *An Accelerated Domain Decomposition Procedure Based on Robin Transmission Conditions*, BIS 37:3, p. 678-686, 1997.

[Dryja93] Maksymilian Dryja, Olof B. Widlund, *Schwarz Methods of Neumann-Neumann Type for Three-Dimensional Elliptic Finite Element Problems*, Technical Report TR1993-626, Computer Sci-

ence Department, Courant Institute of Mathematical Sciences, New York University, 1993.

[Eisi93] Jens Eising, *Lineær Algebra*, Matematisk Institut, Technical University of Denmark, 1993

[Erhel95] J. ERHEL, *A Parallel GMRES Version for General Sparse Matrices*, Electronic Transactions on Numerical Analysis 3, p. 160-176, 1995.

[Ferzi97] Joel H. Ferziger, Milovan Peric, *Computational Methods for Fluid Dynamics*, Springer-Verlag Berlin Heidelberg, 1997.

[Flet01] C. A. J. Fletcher, *Computational Techniques for Fluid Dynamics 2*, Springer, 1991.

[Gods01] Bjørn Godske, *Test kan revolutionere vindmølleindustrien*, Ingeniøren, friday July 6th 2001.

[Hadj00] A. Hadjidimos, D. Noutsos, M. Tzoumas, *Nonoverlapping Domain Decomposition: A linear algebra viewpoint*, Mathematics and Computers in Simulation 51, p. 597-625, 2000.

[Hanke00] Michael Hanke, *Lecture Notes: Advanced Numerical Methods*, Royal Institute of Technology, Stockholm, Department of Numerical Analysis and Computer Science, February 28, 2000.

[Hans98] Per Christian Hansen, *Rank-Deficient and Discrete Ill-Posed Problems. Numerical Aspects of Linear Inversion.*, SIAM, 1998.

[Haase96] G. Haase, *An Incomplete Factorization Preconditioner Based on a Non-Overlapping Domain Decomposition Data Distribution*, Institusbericht Nr. 510, Institut für Mathematic, Johannes Kepler Universität Linz, 1996.

[Mayer98] Stefan Mayer, Antoine Garapon, Lars Sørensen, *A Fractional Step Method for Unsteady Free-Surface Flow with Applications to Non-Linear Wave Dynamics*, Int. Journ. for Num. Methods in Fluids, vol. 28, no. 2, p. 293-315, 1998.

[Mand93a] Jan Mandel, *Balancing Domain Decomposition*, Comm. on Applied Numercial Methods 9, p. 233-241, 1993.

[Mand93b] Jan Mandel, Marian Brezina, *Balancing Domain Decomposition: Theory and Performance in Two and Three Dimensions*, Technical Report, Center for Computational Mathematics, University of Colorado at Denver, 1993.

[McCor94] Stephen F. McCormick, *Multigrid Methods*, SIAM, second printing, 1994.

[Niel96] Hans Bruun Nielsen, *Numerisk Lineær Algebra*, IMM, Technical University of Denmark, 1996.

[Pava96] Luca F. Pavarino, Olof B. Widlund, *Balancing Neumann-Neumann Methods for Incompressible Stokes Equations*, SIAM J. Numer. Anal. Vol. 33, No. 4, p. 1303-1335, 1996.

[Rice98a] J. R. Rice, E. A. Vavalis, Daoqi Yang, *Analysis of a Non-Overlapping Domain Decomposition Method For Elliptic Partial Differential Equations*, J. Comput. Appl. Math. 87:11-19, 1998.

[Rice98b] J. R. Rice, P. Tsompanopoulou, E. Vavalis, *Fine Tuning Interface Relaxation Methods for Elliptic Differential Equations*, preprint submitted to Elsevier Science, 1998.

[Sege00] Roger SegelKen, *Bumblebees finally cleared for takeoff: Insect flight obeys aerodynamic rules, Cornell physicist proves*, `http://www.news.cornell.edu/releases/March00/APS_Wang.hrs.html`, 2000.
Or *Yes, Bumblebees Can Fly, Computer Modeling Proves*, `http://unisci.com/stories/20001/0321005.htm`, 2000.

[Smith96] Barry Smith, Petter Bjørstad, William Gropp *Domain Decomposition - Parallel multilevel methods for elliptic partial differential equations*, Cambrigde University Press, 1996.

[Stra92] Walter A. Strauss *Partial Differential Equations, an introduction*, John Wiley & Sons, Inc, 1992.

[Wang00] Z. Jane Wang, *Two Dimensional Mechanism for Insect Hovering*, Physical Review Letters, vol. 85, num. 10, 2000.