Solving the Unsteady Stokes Equations using the Parareal Algorithm

Erica Madeleine Ligner
Master thesis
University of Oslo
Department of Informatics
`erical@ifi.uio.no / erica.ligner@c2i.net`

1st August 2005

Abstract

We have numerically tested the stability and convergence properties of the Parareal algorithm when it is run on the unsteady Stokes equations. The Parareal algorithm is a parallel–in–time scheme for solving time dependent differential equations. The unsteady Stokes equations from fluid mechanics is a PDE describing creeping flow. The motivation for wanting to use Parareal with the unsteady Stokes equations is to obtain faster computations in time.

We have tested stability and convergence by using variations of the θ–rule discretizations. The results were compared to similar numerical tests of the algorithm used with the heat equation. The Parareal algorithm is known to be stable and convergent for the heat equation from earlier analyses done of the Parareal algorithm. For stiff systems of ODEs the Parareal analysis states that the algorithm is stable when the coarse propagator uses $\theta \in [2/3, 1]$. The unsteady Stokes equations are parabolic PDEs, and when semi–discretized in space they become systems of stiff ODEs. We therefore believe that the Parareal algorithm will remain stable and convergent when run on this problem.

Our numerical results indicate that the Parareal algorithm is indeed stable for $[2/3, 1]$ when it is use to solve the unsteady Stokes equations, although some uncertainty on its convergence rate is experienced at $\theta = 2/3$.

A common way to estimate the error of the solution at time–step k in the Parareal algorithm is by basing it on the norm $\|\lambda_i^k - \lambda_i^{k-1}\|$. It is then assumed that this reflects the true error compared to the exact solution, which is given by the algorithm's equivalent serial solution. We have performed numerical tests that indicate that the ratio of the true error and the approximative error is constant, which suggests that this is indeed a good estimate of the error at iteration k.

The thesis was solved using a combination of Python and Diffpack where all governing code is written in Python. Through its successful usage in this project, the thesis implementation acts as a proof–of–concept to that such a combination is indeed possible for solving problems like the unsteady Stokes equations.

Table of Contents

Chapter 1

Introduction

In 2001, a new parallel scheme in time, called the Parareal algorithm, was proposed by Lions, Maday and Turinici [14], which decomposes the time domain to make parallel implementation possible.

The time dependent Stokes equations describe creeping flow, and are an important simplification of the more complex Navier–Stokes equations that are central in fluid dynamics. Finding a computational solution to the Stokes equation that will execute efficiently is therefore of interest, as they will need to be solved in many situations.

As shown in [2], the Parareal algorithm works well with most time discretizations of parabolic PDEs, and as the time dependent Stokes equations are parabolic, we believe that the Parareal algorithm will be suited for solving the Stokes equations such that the overall simulation time can be reduced.

The main goal of this master thesis is to test whether the Parareal algorithm can be used to solve the unsteady Stokes equations, and to determine possible restrictions as to when the algorithm may be used. Particularly, we will study whether the Parareal algorithm will follow the same stability and convergence properties as for other less complex and well tested parabolic PDEs. We will also evaluate the quality of one of the more commonly used stop criteria for the Parareal algorithm, as little literature is available on this topic.

Naturally, a thesis based on numerical mathematics will have a dual focus; the math associated with discretizing continuous model problems into discrete systems we can solve on a computer, and the technical details of implementation, with all the considerations that accompanies programming.

The heat and Stokes equations will be our model problems throughout this thesis. The heat equation is a well tested PDE with the Parareal algorithm, and its convergence and stability properties are known. We can use the heat equation both as a reference point for how we expect the algorithm to perform with the time dependent Stokes equations *and* as a means for testing the Parareal implementation itself.

It was decided that a flexible combination of Python and Diffpack would be used for our implementation. All governing code such as the Parareal algorithm, handling of time iterations and solving the linear system would be written in Python. Diffpack would merely be seen as a linear system generator and a *possible* source for iterative solvers. The Python library for sparse matrices, Pysparse, would be used as a second source for iterative solvers.

Based on personal preference, we made a decision to use Windows as the development environment for this thesis. The scientific community seems to favor Unix based platforms for their development. Thus at the start of the project there was no clear documentation for using the scientific tools required for the thesis on the Windows platform. Our colleagues and tutors felt uncomfortable using Windows, and we wished to create a guide for using Windows as en environment for scientific computing.

Chapter 2 gives an introduction to spatial and temporal time discretization techniques employed on our model problems. Temporal stability will be introduced for future use with the Parareal algorithm.

Chapter 3 and 4 introduce the heat and Stokes equations, respectively. Discretization of the equations, choice of iterative solvers and associated preconditioners is discussed. An overview of the respective implementations is also provided.

Chapter 5 covers the Parareal algorithm, the possible choices of stop–criteria, its convergence rate and its Python implementation.

Chapter 6 presents the numerical test results from running the Parareal algorithm on the heat and Stokes equations. We will test the stability and convergence properties when it runs with the Stokes equations, and compare this to the equivalent tests for the heat equation. We will also investigate the quality of the error estimate used in Parareal to determine convergence.

Chapter 7 provides the guide to working in the Windows environment.

Chapter 8 describe our reasons for using a Python–Diffpack combination and the experiences we had working with such a combination.

Chapter 9 presents the conclusions from our investigation, and proposes areas for future work.

All source code written for this thesis is available for download at `http://heim.ifi.uio.no/~erical/masterThesis/`

Chapter 2

Preliminaries

In the coming chapters we will make use of temporal and spatial discretization techniques Before proceeding we will in this preliminary chapter present some of the basic principles used in the subsequent chapters. It is not intended as a comprehensive study of the methods, and we thus opt for an engineering approach and only present the practical results which will be used in later chapters. This chapter will also to some degree cover the stability properties of the temporal discretization methods, as it relates to the stability of the Parareal algorithm, which is covered in greater detail in chapter 5 on page 54.

2.1 Spatial discretization with the Finite Element Method

For the discretization of the space–domain we will use the *finite element method* (FEM). During the work on this thesis the discretization of the spatial domain was merely a tool to achieve a fully discretized system, and so the finer points of the method will not be covered here. In this chapter we will present the outlines of the method to justify the discretizations of the heat and Stokes equations done in the following chapters. As *Diffpack* has extensive support for handling FEM, the goal here is merely to give the necessary tools for rendering an equation into a general finite element expression which can, with relatively little effort, be translated into program code using the Diffpack library. Diffpack hides the more complex parts of the finite element method, such as defining the so–called *basis functions* over each element, *integration* of the said basis functions, and assembling element matrices and vectors. We will therefore not discuss such topics in detail here.

The Finite Element Method (FEM) is a tool for solving partial differential equations (PDEs) *approximately*. Whereas the PDE is assumed to hold directly over a given region, Ω, it is a characteristic of FEM that the equation is only assumed to hold over a subregion of Ω (a *finite element*), and the approximation is carried out over each subregion. The global solution over Ω is then found by combining the solutions found over each element. Because of this element approach the method has flexible support of variations in the shape of the domain the problem is the defined over. As we shall see, FEM is well suited for solving initial-boundary value problems such as the time dependent heat and Stokes equations, because the boundary conditions are easily adapted into the discretization. The following discussion is based on [7, 8.1] and [8, 2.1 and 2.3].

To illustrate the principles, we introduce a simple boundary–value problem (BVP)

$$\nabla^2 u(x) = f(x), \quad x \text{ in } \Omega \tag{2.1}$$

$$\nabla u \cdot \eta = g(x), \quad x \text{ on } \partial\Omega_N \tag{2.2}$$

$$u(x) = h(x), \quad x \text{ on } \partial\Omega_E, \tag{2.3}$$

where $u(x)$ is the unknown function and $x \in \mathbb{R}^d$. The boundary $\partial\Omega = \partial\Omega_E \cup \partial\Omega_N$, $\partial\Omega_N \cap \partial\Omega_E = \emptyset$, has a Neumann condition on the outwards–pointing normal of the boundary, and one where Dirichlet boundary conditions are assumed to apply[1]. As shall become evident below, the Neumann condition fits naturally into the FEM scheme, and is often referred to as the *natural boundary condition*, whereas the Dirichlet conditions are often dubbed as the *essential boundary conditions* (thus the subscripts $\partial\Omega_N$ and $\partial\Omega_E$, respectively). Note that without the boundary conditions, an infinite number of candidate functions satisfy (2.1).

The continuous function$u(x)$ that satisfy the given initial-and boundary value conditions is very often hard to find. We therefore wish to approximate $u(x)$ in a *finite context* that can be solved using a computer. The starting point of the finite element method is seeking an approximation of $u(x)$ on the form

$$u(x) \approx \hat{u}(x) = N_0(x) + \sum_{j=1}^{n} u_j N_j(x), \quad x \in \Omega,$$

where the function N_0 is chosen so that it satisfies the essential boundary condition (2.3). The n remaining (basis) functions $N_j(x)$ are chosen so that they are *linearly independent* and *vanish* on the part of the boundary where essential boundary conditions are prescribed, i.e. they satisfy the zero Dirichlet boundary condition $N_j(x) = 0$, for x on $\partial\Omega_E$. The basis functions must also satisfy certain smoothness criterions (more on this later). The weights $\{u_j\}$ are real, albeit unknown, constants, and we can thus see the approximation $\hat{u}(x)$ as a weighted sum of basis functions. As we shall see in 2.1.2, due to the definition of the the basis functions, the essential boundary condition (2.3) can be incorporated into the sum such that we, without loss of information, can represent it by

$$u(x) \approx \hat{u}(x) = \sum_{j=1}^{n} u_j N_j(x), \quad x \in \Omega. \tag{2.4}$$

[1]A Dirichlet boundary condition dictates the values a solution is to take on the boundary of the domain, while a Neumann condition specifies the values the *derivative* of the solution is to take on the boundary [8].

2.1.1 The Galerkin equations

Clearly, if we insert $\hat{u}$ for u in (2.1), we will have a *residual* function, $r(x)$, given by

$$r(x) = \mathcal{L}(\hat{u}(x)) - f(x) \neq 0, \qquad \mathcal{L}(\hat{u}) = \sum_{j=1}^{n} u_j \mathcal{L}(N_j),$$

where $\mathcal{L}$ is the differential operator ∇^2. If $\hat{u} \equiv u$, the residual would of course be zero. Therefore, the closer the residual is to the zero function, the better our approximation will be. Without going into the details of functional analysis[2], we note that as the functions $\{N_j\}$ are defined to be linearly independent, they span the n–dimensional linear space

$$\mathbb{H}_n = \mathrm{Sp}\{N_1, N_2, \ldots, N_n\},$$

in which we can define the inner product $\langle \cdot, \cdot \rangle$ as

$$\langle q, w \rangle = \int_\Omega q(x) w(x) d\Omega.$$

The zero function in $\mathbb{H}_n$ is identified as being orthogonal to all members of said space, such the inner product of the zero function and an arbitrary member of $\mathbb{H}_n$ is zero. As $\mathbb{H}_n$ is, by definition, spanned by the basis functions $N_j(x)$, it is sufficient to demand that the zero function is orthogonal to the basis functions. Hence, we seek weights $\{u_j\}$ such that the residual r is (close to) the zero function, by demanding that

$$\langle r, N_i \rangle = \int_\Omega r(x) N_i(x)\, d\Omega = 0, \quad i = 1, \ldots, n. \tag{2.5}$$

Note that this reasoning is valid only if $\mathbb{H}_n$ is a reasonable approximation to the infinite–dimensional space to all the candidate solutions of (2.1). The equations that arise from the orthogonality conditions in (2.5) are called the *Galerkin equations*. Writing out the inner product $\langle r, N_i \rangle$ for (2.1), we have

$$\int_\Omega \left[\nabla^2 \hat{u} - f\right] N_i \, d\Omega \;=\; 0 \qquad\qquad i \;=\; 1, \ldots, n. \tag{2.6}$$

Integrating by parts using Lemma 2.1.1 on the next page, the equation reads

$$-\int_\Omega \nabla \hat{u} \cdot \nabla N_j \, d\Omega + \int_{\partial\Omega} N_i \frac{\partial \hat{u}}{\partial \eta} d\Gamma \;=\; \int_\Omega f N_i \, d\Omega \quad i \;=\; 1, \ldots, n,$$

and as the basis functions vanish on $\partial\Omega_E$ by design, this reduces to

$$-\int_\Omega \nabla \hat{u} \cdot \nabla N_j \, d\Omega + \int_{\partial\Omega_N} N_i \, g(x) \, d\Gamma \;=\; \int_\Omega f N_i \, d\Omega \quad i \;=\; 1, \ldots, n.$$

[2] [7, appendix A.2] is the basis for the little we use of functional analysis here

Substituting $\hat{u}$ with (2.4) and rearranging the terms, we have the rudiments of a linear system of n equations with n unknowns:

$$\sum_{j=1}^{n}\left(-\int_{\Omega}\nabla N_j\cdot\nabla N_i\,d\Omega\right)u_j=\int_{\Omega}fN_i\,d\Omega-\int_{\partial\Omega_N}N_i\,g(x)\,d\Gamma\quad i\;=\;1,\ldots,n. \tag{2.7}$$

This final version of the Galerkin equations is preferable to the linear system based on (2.6), because integration by parts reduces the degree of differentiability required by the basis functions, which again gives us greater flexibility in our choice of N_j. By solving the linear equation (2.7) we recover the coefficients $u_1,u_2,\ldots,u_n$ that render the best (in the sense of the underlying inner product) linear combination in $\hat{u}$. The linear system exists on a global level in the sense that each basis function is defined over the entire domain Ω, although they will eventually *vanish* over large parts of the domain.

Lemma 2.1.1 (Green's Lemma for integration by parts)**.**

$$-\int_{\Omega}\nabla\cdot[\kappa\nabla u]\,v\,d\Omega\;=\int_{\Omega}\kappa\nabla u\cdot\nabla v\,d\Omega-\int_{\partial\Omega}v\,\kappa\nabla u\cdot\eta\,d\Gamma \tag{2.8}$$

which implies

$$-\int_{\Omega}(\nabla\cdot u)\,v\,d\Omega\;=\int_{\Omega}u\cdot(\nabla v)\,d\Omega-\int_{\partial\Omega}v\,u\cdot\eta\,d\Gamma.$$

Alternatively, we can write $\nabla u\cdot\eta$ *as* $\dfrac{\partial u}{\partial\eta}$.

2.1.2 The Finite Elements

We now turn our attention to the choice of basis functions. As mentioned earlier, we need to construct the set of functions $\{N_i\}$ so that they are (approximately) linearly independent, but we would also like to keep the computational cost of evaluating the integrals at a minimum, which is true since we ensure that they disappear over large parts of the global domain and by choosing them so that one can construct efficient methods for numerically evaluating their definite integrals. Quoting [8, 2.3.1], we have that "*the finite element choice of N_i consists of three fundamental ideas:*

1. *divide the domain into non–overlapping* elements,
2. *let N_i be a simple polynomial over each element,*

3. *construct the global N_i as a piecewise polynomial that vanishes over most of the elements, except for a local patch of elements."*

Following the first principle, we divide Ω into m non–overlapping elements (sub domains), $\Omega_1, \ldots, \Omega_m$. To each element we assign a set of $p+1$ points (nodes) $x^{[i]},\ i = 1, \ldots, p+1$, which is enough to represent a polynomial of degree p. Globally we then have $mp+1$ nodes ($x^{[i]},\ i = 1, \ldots, mp+1$) distributed over the domain. To achieve the 2nd and 3rd principles, we define the basis functions to have the properties[3]

1. *N_i is a polynomial over each element, uniquely determined by its values at the nodes in the element.*

2. *$N_i(x^{[j]}) = \delta_{ij}$. δ_{ij} is the Kroenecker delta, which equals unity when $i = j$ and vanishes otherwise.*

This last property has the logical consequence that

$$\hat{u}(x^{[i]}) = \sum_{j=1}^{n} u_j N_j(x^{[i]}) = u_i,\ i = 1, \ldots, n,$$

which suggests that we can interpret u_i as the value of the function $\hat{u}$ at node i. As was mentioned earlier, $\hat{u}(x)$ is really approximated as

$$\hat{u}(x) = N_0(x) + \sum_{j=1}^{n} u_j N_j(x), \quad x \in (\Omega \cup \partial\Omega).$$

We have so far omitted to consider the N_0 term, though we will do so here. As some of the elements Ω_e will include the boundary of the model problem, $\partial\Omega = \partial\Omega_E \cup \partial\Omega_N$, a subset of the nodes $x^{[j]}$ will lie on $\partial\Omega_E$. As we have established, this subset $\{u_k\}$ of the weights $\{u_j\}$, is the solution to $\hat{u}$ on $\partial\Omega_E$. By the essential boundary condition (2.3), $\{u_k\}$ should be given by $h(x^{[k]})$, for $x^{[k]}$ on $\partial\Omega_E$. Ergo these weights are *not* unknown. Assuming k is the counter running over the nodes on $\partial\Omega_E$ and j runs over the nodes in $\Omega \cup \partial\Omega_N$, we have

$$\hat{u}(x) = \sum_k u_k N_k(x) + \sum_j u_j N_j(x).$$

In view of the finite element interpretation of the weights $\{u_j\}$, a clean way to handle the essential boundary condition is to simply merge the sums into one, and use the essential boundary condition to explicitly set $\{u_k\}$ in the resulting linear system (2.7). In other words, we continue to use $\hat{u}(x) = \sum_j u_j N_j(x)$, and replace the entry

$$\left(-\int_\Omega \nabla N_j \cdot \nabla N_i \, d\Omega \right) u_j \ = \ \int_\Omega f N_i \, d\Omega - \int_{\partial\Omega_N} N_i \, g(x) \, d\Gamma$$

[3]Also quoted from [8, 2.3.1]

with

$$u_j = h(x^{[j]})$$

for all $x^{[j]}$ on $\partial\Omega_E$.

To make FEM a flexible tool for handling complex shapes of Ω, it operates in an element–by–element fashion, and lets each element be mapped onto a standard (uniform) reference element with local coordinate system and node numbering. The uniform element shape is designed to make the calculation of the integral expressions *identical for each element*, and so be *independent* of any complexities in the global domain. This makes generic frameworks for finite elements such as Diffpack feasible. To achieve this, let $A^{(e)}$ denote a $n \times n$ matrix where all entries are zero, except a cluster of those representing the contributions from the integral expressions in element e, and let $b^{(e)}$ be interpreted in a similar fashion for the right hand side of the global system (2.7). The Galerkin equations in (2.7) can then be written as

$$Au = b \text{ where } A = \sum_{e=1}^{m} A^{(e)}, \quad b = \sum_{e=1}^{m} b^{(e)}.$$

Where elements share a node, contributions from both elements will be added to the appropriate entries of A and b corresponding to said node. We then collect the block of non–zero entries of $A^{(e)}$ and $b^{(e)}$ in element matrices and vectors, $\tilde{A}^{(e)}$ and $\tilde{b}^{(e)}$, and let the integrals be defined over a local coordinate system. Naturally, there must be some means to perform a mapping to/from the local/global coordinate system, in addition to mapping between local and global node numbering. Changing coordinate system will affect both the integral and the derivative expressions.

Diffpack hides the details of derivative transformation, the shape of the reference element in addition to concealing the mapping between local and global views and the assemblage of the global system based on element contributions. We write the expressions for the element matrices and vectors in Diffpack in accordance with (2.7) as

$$\sum_{j=1}^{n_e} \left(-\int_{\tilde{\Omega}} \nabla N_j \cdot \nabla N_i \det J \, d\xi \right) u_j = \int_{\tilde{\Omega}} f N_i \det J \, d\xi, \quad i = 1, \ldots, n_e,$$

assuming a Dirichlet condition $g(x) = 0$. Here, $\tilde{\Omega}$ is the reference element, the local coordinate system ξ is related to x through the mapping $x^{(e)}(\xi)$, $\det J$ is the determinant of the Jacobean matrix[4], and n_e is the number of nodes in the element. This is the Diffpack notation. In the “correct” mathematical expression you would also use the Jacobean to transform the derivatives. This concludes our discussion of discretization in the space domain, and we proceed to present a framework for discretizing the temporal domain.

[4] $J_{ij} = \partial x_j / \partial \xi_i$

2.2 Temporal discretization – the θ-rule

Temporal discretization of PDEs use the same techniques as those used to discretize ordinary differential equations (ODEs). An ODE is an equation that involves the derivatives of an unknown function of one variable, as opposed to PDEs which are equations that involve partial derivatives of an unknown function of several variables. A general initial value ODE has the form

$$\frac{\partial u}{\partial t} = f(t,u),\ t \geq 0,\ u(t_0) = u_0.$$

To use ODE techniques to handle temporal discretization of PDEs is not unreasonable. A PDE that is only discretized in space (i.e it is semi–discretized) become a set of ordinary differential equations, which we can show conceptually for the heat equation by

$$\frac{\partial u}{\partial t} = \nabla^2 u \Rightarrow \frac{\partial u}{\partial t} = f(t,u),\ \ f(t,u) = D_h u(t)$$

where D_h is a matrix representing the discretization of the partial differential operator ∇^2, and u is a vector where each entry is a time–dependent function. Semi–discretizing in space and then solving the resulting initial value problem (IVP) is often referred to as the *method of lines*, and the technique is typically suitable for *parabolic* PDEs [1]. As we will work with parabolic PDEs in this thesis, it is appropriate to discuss this technique here. D_h and u can be found by for example using the finite difference method, where we would approximate the unknown $u(x,t)$ by

$$u(x,t) \approx \hat{u}(x,t) = \sum_{j=1}^{n} u_j(t) N_j(x),$$

such that the vector u is comprised of the entries $u_j(t)$. D_h will then be the stiffness matrix, as we will see in chapter 3 on page 15.

Note that we could just as well have discretized the time derivative before handling the spatial domain, to create a set PDEs which could in turn be discretized using FEM – the fully discretized system at the end is independent of which domain is handled first. In the case where the time derivative is discretized prior to the spatial domain the right–hand side would be continuous ($\nabla^2 u(t,x)$) and not discrete ($D_h u(t)$). Indeed, this approach is what we will be use in the latter chapters, but we show the reverse order here because we want to utilize some of the analytic tools available for ODEs.

There are many different techniques for discretizing time derivatives on the form $\partial u/\partial t$ in ODE theory, although we shall only employ the simpler methods for this thesis, and consider methods such as *Runge–Kutta* and *multistep* schemes beyond the scope of this work. Common techniques for solving the time derivatives for PDEs seem to be the *forward*, *backward* and *centered* differences, which is often referred to as the *Explicit*/*Implicit Euler* and the *Crank-Nicolson* schemes, respectively. The θ-*rule* unifies these three discretization methods into a general discretization scheme, making it possible to easily alternate between the methods. This is particularly useful from an implementational point of view, as the same code can be used to generate the different discretization techniques by changing the value of a variable.

The idea is that we sample the derivative at the temporal point $t = t_{\ell-1+\theta}$, $\theta \in [0,1]$, and use[5]

$$\frac{\partial}{\partial t} u(x, t_{\ell-1+\theta}) \approx \frac{u^{\ell} - u^{\ell-1}}{\Delta t}.$$

The notation u^{ℓ} implies $u(t = t_{\ell})$, where $t_{\ell} = \ell \Delta t$ and Δt the length of the time–step. The right–hand side f at the time point $t_{\ell-1+\theta}$ is then approximated with a linear interpolation between the values of f at time points t_{ℓ} and at $t_{\ell-1}$. Thus the we have (2.9):

θ-rule.

$$\frac{u^{\ell} - u^{\ell-1}}{\Delta t} = \theta f^{\ell} + (1-\theta) f^{\ell-1}, \; \theta \in [0,1]. \tag{2.9}$$

For the semi–discretized heat equation in the previous example, we would have $f^{\ell} = f(t_{\ell}, u) = D_h u^{\ell}$, where $u^{\ell} = \{u_j(t_{\ell})\}$. Setting $\theta = 0$ we recapture the Explicit Euler scheme, $\theta = 1/2$ gives the Crank-Nicolson scheme, and $\theta = 1$ will produce the implicit Euler scheme.

2.2.1 Stiff ODEs and stability

The Explicit and Implicit Euler and Crank–Nicolson methods all show different stability traits, which suggests that stability is dependent on θ. *Stability* of a numerical discretization scheme refers to the numerical scheme's ability to mirror the qualitative properties of the true solution. A stable scheme will not introduce excessive amplification of components in the numerical solution that strongly deviates from the true solution, and thus turning the numerical results into nonsense. It will also recover the behaviour of the true solution of the ODE regardless of the size of Δt, though naturally the quality in terms of accuracy deteriorates as Δt increases.

The stability of the Parareal algorithm is closely linked with the stability of the numerical scheme(s) used to discretize the time derivative. The algorithm operates with a coarse and a fine discretization, and though they do not have to be based on the same scheme, i.e. use the same θ–value, they must both be stable for their chosen time steps in order for the algorithm itself to be stable. In addition the coarse coarse discretization must introduce damping. During testing it is likely practical if both the fine and coarse solvers are unconditionally stable – ergo be invariant of the time step – as we can then use arbitrary time step sizes without having to consider possible instabilities in the discretizations. For further details on the Parareal algorithm, see chapter 5 on page 54.

In the following, we will briefly study the cause for instabilities in the numerical schemes, which will lead to the notion of *stiff* ODEs and *stability functions*. From this we will be able

[5]The θ–rule as presented here is based on sections 1.7.6 and 2.2.2 in [8]

to see how we should choose θ in order to (theoretically) avoid instabilities in the discretization, and therefore also in the Parareal algorithm when it is used to solve the heat and Stokes equations. The following discussion is based on [7, chapters 4.1–4.2], unless otherwise stated.

A model problem and the unique solution

For our discussion of the concepts of stability and what stiff ODEs are, we will use a simple model problem and present an expression for its unique solution. The unique solution (or at least the knowledge on how the unique solution behaves) is essential for showing how well a numerical method approximates the true solution. We will study the simple model problem

$$\frac{du}{dt} = Au, \quad u(0) = u^0, \qquad \text{where } A = \begin{bmatrix} -100 & 1 \\ 0 & -\frac{1}{10} \end{bmatrix}, \tag{2.10}$$

which is originally presented in [7, chapter 4.1]. The parallel to the semi–discrete heat equation presented earlier in this chapter should be clear. By Proposition 2.2.1 the unique

Proposition 2.2.1. *Let v_j be an eigenvector of the $n \times n$ matrix C with eigenvalue λ_j, and α_j a scalar constant determined by the initial condition x_0. Then*

$$\begin{aligned} x(t) = \sum_{j=1}^{n} v_j e^{\lambda_j t}\alpha_j = Ve^{Dt}\alpha, \quad & V = (v_1 \mid v_2 \mid \cdots \mid v_n) \\ & D = diag(\lambda_1, \lambda_2, \ldots, \lambda_n) \end{aligned} \tag{2.11}$$

is a general *solution to the system of differential equations* $\frac{dx}{dt} = Cx.$

[6, Theorem 4.7.3]
[6, Principle of superposition,pg.155]

The vector α is determined by the initial condition

$$x(0) = V\alpha \quad \Rightarrow \quad \alpha = V^{-1}x^0,$$

such that, with [6, Theorem 6.3.1],

$$x(t) = Ve^{Dt}V^{-1}x^0 = e^{tC}x^0 \tag{2.12}$$

is the unique *solution to the system* $\frac{dx}{dt} = Cx,\ x(0) = x^0.$

solution to (2.10) is

$$u(t) = e^{At}u^0 = Ve^{Dt}V^{-1}u^0 \quad \text{or, alternatively,} \quad u(t) = \sum_{j=1}^{n} e^{\lambda_j t} y_j = e^{Dt} y_j, \tag{2.13}$$

where y_j is a vector dependent on α_j and v_j [6]. As long as $\mathrm{Re}\lambda_j < 0$, it is clear that the solution of (2.10) behaves in an asymptotic manner. Solutions where $\mathrm{Re}\lambda_j > 0$ is of limited interest since the solution would rapidly become very large. We will assume that we have eigenvalues where the real components are negative; $\mathrm{Re}\lambda_j < 0$, because we only discuss stability for a stable solution. For our model problem we have the following pairs of eigenvalues and eigenvectors:

$$\lambda_1 = -100,\ v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \lambda_2 = -\frac{1}{10},\ v_2 = \begin{bmatrix} 1 \\ \frac{999}{10} \end{bmatrix}$$

Stiffness

The aim is to produce numerical methods that recapture the asymptotic behaviour of the exact solution. Discretizing (2.10) by the θ–rule, we have[7]

$$\begin{aligned} u^\ell - \Delta t\theta A u^\ell &= u^{\ell-1} + \Delta t(1-\theta)Au^{\ell-1} \\ u^\ell &= \left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)u^{\ell-1} \end{aligned}$$

which can be written as

$$\begin{aligned} u^1 &= \left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)u^0 \\ u^2 &= \left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)u^1 = \left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)^2 u^0 \\ &\vdots \\ u^\ell &= \left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)^\ell u^0, \end{aligned} \tag{2.14}$$

where I is the identity matrix. Since we want (2.14) to approximate (2.13) at $t = t_\ell = \ell\Delta t$, we should have

$$\left(\frac{I+\Delta t(1-\theta)A}{I-\Delta t\theta A}\right)^\ell \approx e^{At_\ell}.$$

[6]For our discussion we assume that the diagonalization $A = VDV^{-1}$ is always possible.

[7]This should really be written as $\left((I-\Delta t\theta A)^{-1}(I+\Delta t(1-\theta)A)\right)$, but for the sake of a more compact expression we follow the notation used in [7]

A reasonable conjecture would then be that, based on (2.13), we can write the above as

$$\begin{gathered}\left(\frac{I+\Delta t(1-\theta)D}{I-\Delta t\theta D}\right)^{\ell} \approx e^{Dt_\ell}\\ \Downarrow \\ \left(\frac{1+\Delta t(1-\theta)\lambda_j}{1-\Delta t\theta\lambda_j}\right)^{\ell} \approx e^{\lambda_j t_\ell}\end{gathered}$$

such that (2.14) reads

$$u^\ell = V\left(\frac{I+\Delta t(1-\theta)D}{I-\Delta t\theta D}\right)^{\ell} V^{-1}u^0 = \sum_{j=1}^{n}\left(\frac{1+\Delta t(1-\theta)\lambda_j}{1-\Delta t\theta\lambda_j}\right)^{\ell} y_j.$$

In order for (2.14) to have the asymptotic behaviour of the exact solution we must have

$$\left|\frac{1+\Delta t(1-\theta)\lambda_j}{1-\Delta t\theta\lambda_j}\right| < 1,$$

or else the solution will rapidly blow up and become very large, i.e it becomes unstable.

Let us look at how this influences the choice of suitable θ–values. Setting $\theta = 0$ causes the discretization method to be explicit euler, and unless $\left|1+\Delta t\lambda_j\right| < 1$ for all eigenvalues, the solution will be unstable (it becomes arbitrarily large). As $\lambda_1 = -100$ and $\lambda_2 = -1/10$, we see that λ_1 places severe restraints on Δt compared to λ_2 (λ_1 force $\Delta t < 1/50$). If, on the other hand, we choose $\theta = 1$, we have an implicit euler scheme and the inequality that must be satisfied for all eigenvalues become

$$\left|\frac{1}{1-\Delta t\lambda_j}\right| < 1,$$

which is unconditionally true because $0 < \Delta t$ and $\mathrm{Re}\lambda_j < 0$, rendering $\left|1-\Delta t\lambda_j\right| > 1$, such that we can choose any non–negative Δt as our time–step.

For $\theta = 0$ we pointed out how λ_1 enforced a much smaller Δt compared to the demands set by λ_2 in order for the system to remain stable: widely different scales on the eigenvectors forced us to crawl when we could leap toward the solution. Our model problem is therefore an example of a so–called *stiff* ODE. [7, pg. 56] defines an ODE as stiff if "... *its numerical solution by some methods requires a significant depression of the step–size to avoid instability.*". The mechanism that usually induces stiffness is systems that have eigenvalues with very different scales, which implies that the "lifetime" of the different $v_j e^{\lambda_j t}\alpha_j$ components of the solution varies greatly.

Stability

As earlier stated, we want methods that are unconditionally stable in order for the Parareal algorithm to function properly under testing where it would be desirable to take larger time steps in order to reduce the execution time. In our preceding discussion of stability we pointed out that, for the θ–rule discretization of our model problem, the factor

$$\frac{I + \Delta t(1-\theta)A}{I - \Delta t\theta A}$$

was the crux for establishing stability. We call this the *stability function* for the θ–rule, and write

$$R(z) = \frac{I + (1-\theta)z}{I - \theta z}, \quad z = \Delta t A.$$

We can also write the stability function in terms of the eigenvalues

$$R(z_i) = \frac{1 + (1-\theta)z_i}{1 - \theta z_i}, \quad z_i = \Delta t \lambda_i, \; \forall\, \lambda_i \in A$$

The stability domain D of a method is the set of all z_i for which the asymptotic behaviour of the ODEs is recovered, provided that the latter equation is stable ($\mathrm{Re}\lambda_i < 0$). *Strong A–stability* is a way to restrain $R(z)$ to prevent the numerical method to blow up and produce nonsense solutions. A method that satisfies

$$\lim_{z \to -\infty} R(z) = a, \qquad \text{where } 0 < |a| < 1$$

is said to be strong A–stable. See Definition 4 in [20]. By way of *L'Hôpital's rule* one finds that the θ–method is strong A–stable for $\frac{1}{2} < \theta < 1$. In [20, Theorem 3] it is stated that if, for stiff ODEs, the stability function of the coarse propagator of the Parareal algorithm satisfies

$$\lim_{z \to -\infty} |R(z)| \leq \frac{1}{2},$$

then the predictor–corrector scheme on which the Parareal algorithm is built will be stable. This implies that the Parareal algorithm will be stable for $\frac{2}{3} \leq \theta \leq 1$ – provided that the underlying ODEs are stiff. Semi–discretization of *parabolic* PDEs give rise to a system of stiff ODEs, regardless of the elected method of discretization for the spatial domain [5]. As both the heat and the Stokes equations are parabolic PDEs we will be using Parareal to solve stiff systems, which justifies our discussion of stability for such systems.

This concludes our preliminary chapter. The next two chapters uses the spatial and temporal discretization techniques discussed here on the heat and Stokes equations, respectively, to arrive at fully discretized systems. Their implementation will also be discussed, before we move on to the Parareal equation. The stability concepts we have established will play a key role in studying how Parareal functions on the Stokes equations, which is covered in chapter 6 on page 70.

Chapter 3

The Heat Equation

We will in the following give an introduction to the heat equation, and briefly describe the methods applied for discretization in space and time. The heat equation is also referred to as the diffusion equation, since it models how heat propagates through a medium over time. As a matter of fact, you can use the heat (diffusion) equation to describe any unit that spreads out over a defined region in space and time in a manner consistent with heat dispersion.

The heat equation will be used extensively throughout this thesis. From a PDE point of view it is a simple and well studied parabolic equation, and as such it is an excellent test problem for this group of PDEs. Here, we will use the equation to establish discretization principles before moving on to the more complex Stokes equations. In addition, the Parareal algorithm is known to be well behaved when it is used to solve the heat equation – it was for instance one of the model problems applied in the study of the algorithm's convergence and stability properties in [20]. This implies that we can use the heat equation both as a reference point for how we expect the algorithm to perform with the time dependent Stokes equations *and* as a means for testing the Parareal implementation itself. Assuming that the model problem is correctly implemented, any deviations from the expected behaviour will be due to errors in the Parareal source code. "Correct" Parareal behaviour will be covered in later chapters.

3.1 Mathematical Model

The heat equation describes the variation of temperature in a given region over time, i.e how heat diffuses in a conductor. For heat propagation in a homogeneous medium our initial-boundary value problem (IBVP) reads

$$
\begin{aligned}
\frac{\partial u}{\partial t} &= \nabla \cdot (\kappa \nabla u) + f && \text{in } \Omega, \quad 0 < t, && (3.1)\\
u(x,t) &= g(x,t) && \text{on } \Omega_E,\, 0 < t, && (3.2)\\
\nabla u \cdot \eta &= 0 && \text{on } \Omega_N,\, 0 < t, && (3.3)\\
u(x,t) &= u_0 && \text{in } \Omega, \quad t = 0, && (3.4)
\end{aligned}
$$

where $u(x,t)$ is the unknown temperature, κ is a material-specific constant describing the conductive properties of the medium and $f(x)$ is a (constant in time) heat source. u_0 is the initial state of the system, $g(x,t)$ is Dirichlet type boundary conditions and $x \in \mathbb{R}^d$.

3.1.1 Temporal discretization

In chapter 2.2 on page 9 we established the θ–rule as a discretization method for the time domain, and we will employ it here to semi–discretize the heat equation. When previously discussing the θ–rule, we merely presented the technique, and briefly discussed its general stability properties. We pointed out how stability influence the range of possible θ values when discretizing a stiff problem that is to be used with Parareal in a meaningful way. This restricted θ such that we must have $\theta \in [2/3, 1]$. The stability and convergence of the Parareal algorithm were studied in [2] and [20], and it was found that as long as the time discretizations are stable, the algorithm shows distinct, exponential convergence toward the serial computation based on the fine propagator[1]. Naturally, the algorithm will no longer show the characteristic exponential decay in error if $\theta \notin [2/3, 1]$. By using the θ–method to discretize, adjusting the θ is easily done, and it is simple to verify that the convergence of the algorithm is behaving as expected [for the heat equation]. As the heat equation was one of the model problems in [20], we know that Parareal will behave just as expected when solving the initial–boundary value problem (3.1)-(3.4) – which incidentally also makes it an excellent test case for the Parareal implementation itself.

The overall goal of this thesis is to study the behaviour of Parareal when run on the unsteady Stokes equations compared to running it on our blue–print model, the heat equation. It it is therefore of interest to discretize the Stokes equations such that it is possible to study the performance of Parareal under the different temporal discretization techniques induced by varying θ. By doing θ–discretization of the heat equation, we have a reference for how the Parararreal algorithm should behave, and also the means to establish the principles of θ–discretization on a simpler problem.

[1]Central properties and typical convergence rate of the algorithm are reviewed in chapters 5.1.1 on page 58 and 5.1.3 on page 61

Applying (2.9) to (3.1) we have

$$\begin{aligned} \frac{u^\ell - u^{\ell-1}}{\Delta t} &= \theta\left(\nabla\cdot(\kappa\nabla u^\ell) + f\right) + (1-\theta)\left(\nabla\cdot(\kappa\nabla u^{\ell-1}) + f\right) \\ u^\ell - \Delta t\ \nabla\cdot(\kappa\nabla u^\ell) &= \Delta t\theta f + u^{\ell-1} + \Delta t(1-\theta)\nabla\cdot(\kappa\nabla u^{\ell-1}), \end{aligned} \tag{3.5}$$

which leaves us with a semi-discrete system of equations with initial and boundary conditions given by (3.2)–(3.4).

3.1.2 Spatial discretization

We will discretize the spatial domain of the the semi-discrete equations (3.5) by using the *Finite Element Method* (FEM) as described in chapter 2 on page 3. By approximating u as a sum over weighted basis functions,

$$u^\ell \approx \hat{u}^\ell = \sum_{j=1}^{n_u} u_j^\ell N_j(x),$$

and integrating to minimize the residual, we have

$$\begin{aligned} \int_\Omega \left[\hat{u}^\ell - \Delta t\theta\nabla\cdot(\kappa\nabla\hat{u}^\ell)\right] N_i\, d\Omega = \int_\Omega \Big[\ \Delta t\theta f + \hat{u}^{\ell-1} + \\ \Delta t(1-\theta)\nabla\cdot(\kappa\nabla\hat{u}^{\ell-1})\Big] N_i\, d\Omega, \end{aligned} \tag{3.6}$$

for $i = 1, \ldots, n_u$. Using Green's lemma to integrate by parts (2.1.1 on page 6), and the natural boundary condition (3.3) to eliminate the integral over the boundary $\partial\Omega_N$, we get

$$\begin{aligned} \int_\Omega \left[\hat{u}^\ell N_i + \Delta t\theta\kappa\nabla\hat{u}^\ell\cdot\nabla N_i\right]\, d\Omega &= c_i^\ell, \quad i = 1, \ldots, n_u \\ \sum_{j=1}^{n_u} u_j^\ell \left(\int_\Omega \left[N_j N_i + \Delta t\theta\kappa\nabla N_j\cdot\nabla N_i\right]\, d\Omega\right) &= c_i^\ell, \quad i = 1, \ldots, n_u, \end{aligned}$$

where c_i^ℓ is the right–hand side of (3.6). We must also do integration by parts on the $\nabla\cdot(\kappa\nabla\hat{u}^{\ell-1})$ term in c_i^ℓ, or else we will still demand that the basis functions are twice differentiable. As was pointed out in chapter 2.1.1 on page 5, this is not desirable since it restricts the pool of possible basis functions. Applying the same procedure as above on c_i^ℓ, we arrive at the linear system

$$\sum_{j=1}^{n_u} (M_{ij} + K_{ij}) u_j^\ell = c_i^\ell, \quad i = 1, \ldots, n_u, \tag{3.7}$$

where

$$\begin{aligned}
M_{ij} &= \int_\Omega N_j N_i \, d\Omega \\
K_{ij} &= \int_\Omega \Delta t \theta \kappa \nabla N_j \cdot \nabla N_i \, d\Omega \\
c_i^\ell &= \int_\Omega \left[\left(\Delta t \theta f + \hat{u}^{\ell-1} \right) N_i - \Delta t (1-\theta) \kappa \, \nabla \hat{u}^{\ell-1} \cdot \nabla N_i \right] d\Omega .
\end{aligned}$$

M is often referred to as the *mass matrix* whereas *K* is usually called the *stiffness matrix*. Mapping the integral expressions for each element to the local coordinate system, we have

$$\begin{aligned}
\tilde{M}_{ij}^{(e)} &= \int_{\tilde{\Omega}} N_i N_j \, \det J \, d\xi_1 \cdots d\xi_d \\
\tilde{K}_{ij}^{(e)} &= \int_{\tilde{\Omega}} \Delta t \theta \kappa J^{-1} \nabla N_i \cdot J^{-1} \nabla N_j \det J \, d\xi_1 \cdots d\xi_d \\
\tilde{c}_i^{(e)\ell} &= \int_{\tilde{\Omega}} \left[\left(\Delta t \theta f + \hat{u}^{\ell-1} \right) N_i - \Delta t (1-\theta) \kappa \, \nabla \hat{u}^{\ell-1} \cdot \nabla N_i \right] \det J \, d\xi_1 \cdots d\xi_d
\end{aligned}$$

where $\xi = (\xi_1 \cdots \xi_d)$ are the local coordinates in reference element $\tilde{\Omega}$. The parallel to Diffpack should be easy to perceive; the element expressions can be found in Listing 3.1 on page 26.

3.2 Iterative solver

By the discretizations done in the previous section, we now have a linear system that must be solved in order to create a numeric solution to our model problem. The next step is therefore to choose an appropriate *linear solver*. In the following discussion we will write the linear system (3.7) as

$$Mu + Ku = (M + K)u = c.$$

In choosing the iterative solver, the properties of the matrix $A = M + K$ play a central role. We will discuss the properties of M and K separately, and from this discussion draw conclusions on the overall properties of A. As we will show, both the mass and the stiffness matrices are *symmetric, positive–definite* (SPD) matrices, and therefore A is also SPD. This is significant for our choice of solver. The Conjugate Gradient (CG) method is one of the more prominent and well–known iterative methods for solving SPD systems of linear equations, and the *preconditioned* algorithm is an efficient de facto standard for such systems. Indeed, the preconditioned CG will be our choice of iterative solver for the heat equation.

For our study of M and K we will only consider real vectors, i.e $x \in \mathbb{R}^n$, as the solution to (3.1)–(3.4) will be real. The object of the current section is not to present why the matrix must be SPD in order to use CG as the iterative solver, but merely show that it is, and that CG can indeed be used to solve the current problem. For a good introduction to the conjugate gradient method and why it requires symmetric, positive–definite matrices, see [17] and [19].

A matrix is *positive–definite* if it satisfies

$$x^T Ax > 0,\ \forall\, x \neq 0,$$

and *symmetric* if it is its own transpose, i.e $A = A^T$, which means that although the diagonal itself is arbitrary, all other entries occur in pairs on opposite sides of the main diagonal. A symmetric matrix is necessarily square. For further details, see [13]. We also have that if A and B are positive–definite, then the matrices $A+B$ and AB will be positive–definite.

We first consider the mass matrix

$$M = \int_\Omega NN^T\, d\Omega,$$

where $N = [N_1\ N_2 \cdots N_{n_u}]$. As each component M_{ij} in M relies on N_jN_i, it is clear that M must be symmetric; the entries will repeat themselves across the diagonal since N_iN_j is commutative ($N_iN_j \equiv N_jN_i$). To determine whether M is positive–definite we need to consider the matrix $B = NN^T$. If B is positive–definite, M will clearly also be positive–definite. We have

$$\begin{aligned} u^T Bu = u^T NN^T u &= (u_1N_1 + \cdots + u_{n_u}N_{n_u})(u_1N_1 + \cdots + u_{n_u}N_{n_u}) \\ &= \left(\sum_{j=1}^{n_u} u_jN_j\right)^2, \end{aligned}$$

which implies

$$u^T NN^T u > 0\ \forall\, u \neq 0,$$

and hence M is symmetric, positive–definite.

We then turn our attention to the stiffness matrix, K. The following discussion is based on [16]. Let

$$B = \nabla N = \begin{bmatrix} \frac{\partial N_1}{\partial x_1} & \cdots & \frac{\partial N_{nu}}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial N_1}{\partial x_d} & \cdots & \frac{\partial N_{nu}}{\partial x_d} \end{bmatrix},$$

such that we can write K as

$$K = \int_\Omega \Delta t\theta\kappa\, B^T B\, d\Omega.$$

Also, we may express entry (i, j) in the matrix product $B^T B$ as

$$\sum_{r=1}^{d} B_{ri}B_{rj}.$$

Following the same argument as for M, $B^T B$ is clearly symmetric because the product $B_{ri}B_{rj}$ is commutative, and thus K must be symmetric. To ascertain whether K is positive–definite, we consider the product $u^T Ku$. Recall that the solution to the heat equation is approximated as $\hat{u}(x) = uN(x)$, and thus $uB = u\nabla N = \nabla\hat{u}$, and we can write $u^T Ku$ as[2]

$$u^T Ku = \int_\Omega \Delta t\theta\kappa\, u^T B^T Bu\, d\Omega = \int_\Omega \Delta t\theta\kappa\nabla\hat{u}\cdot\nabla\hat{u}\, d\Omega \geq 0, \tag{3.8}$$

because $\nabla\hat{u}\cdot\nabla\hat{u} \geq 0$. This implies that K is at the very least positive–*semi*definite. However, from (3.8) we see that $u^T Ku = 0$ if and only if the temperature gradient $\nabla\hat{u}$ is zero for all $x \in \Omega$, which would also imply that all components (nodal values) in u are equal. To study this prerequisite, we split u into two logical blocks,

$$u = \begin{bmatrix} u_E \\ \tilde{u} \end{bmatrix}.$$

u_E denotes all the nodes on the essential boundary $\partial\Omega_E$ determined by the Dirichlet boundary condition $u(x = a,t) = g(a,t)$, a on $\partial\Omega_E$, and $\tilde{u}$ is the vector of remaining nodes in $\Omega \cup \partial\Omega_N$. As u_E is prescribed, all components in $\tilde{u}$ would also have to be given by $g(a,t)$ in order for $\nabla\hat{u}$ to be zero. However, x is obviously not restricted to $\partial\Omega_E$, and so $g(x = a,t)$ is clearly not a solution to (3.1). We therefore have $\nabla\hat{u} \neq 0$ when the solution must satisfy Dirichlet boundary conditions. Thus

$$u^T Ku > 0 \;\forall\; u \neq 0,$$

and K is symmetric, positive–definite.

We conclude that $M + K$ is SPD, and we can use the conjugate gradient as an iterative solver for the system.

3.2.1 Preconditioning

We will take a brief look at preconditioning of the linear system, as it is a necessary tool for ensuring convergence within a limited number of iterations, and so guarantees that a solution is found within a reasonable amount of computation time. Preconditioners for the Conjugate Gradient method is available as library routines in both Diffpack and Pysparse, and only limited consideration of its basic shape is necessary here. Preconditioning the linear system arising from discretizing the Stokes equations is (naturally) more complex. Neither Diffpack nor Pysparse have any immediate functionality to precondition the system, and so we must construct it from more general library preconditioning tools. As a natural consequence, the details of the preconditioner will receive more attention in the Stokes chapter, whereas we

[2] Here, u is the vector representing the weights $\{u_j\}$ from chapter 3.1.2 on page 17, not to be confused with the exact solution $u(x,t)$.

present more general concepts of preconditioning here. Also, by introducing preconditioners here, we once again underline the similarities between solving the (in PDE terms) simple heat equation and the more complex Stokes equations.

Preconditioning is a way to manipulate a linear system $\mathcal{A}x = b$ before applying an iterative solver, with the aim of making the system better suited for the particular solver. By making the linear system better suited we mean that by applying the preconditioner before solving the linear system, the number of iterations before the iterative solver converges is significantly reduced. In other words, we increase the efficiency, as long as applying the preconditioner is significantly cheaper than the total cost of the avoided iterations.

The efficiency of the conjugate gradient method depend on well conditioned matrices. This is for that matter true for all the methods of the same family – the *Krylov subspace methods*. The condition number of $\mathcal{A}$, $\kappa(\mathcal{A})$, is a measure of how well-posed the system is. For a singular matrix $\kappa \to \infty$, whereas the identity matrix has $\kappa = 1$. This indicates that the smaller the condition number, the more amenable the linear system will be to digital computation. We would like to find a preconditioner $\mathcal{B}$, such that $\kappa(\mathcal{BA}) << \kappa(\mathcal{A})$, with the logical consequence that $\mathcal{B}$ should in some way be close to $\mathcal{A}^{-1}$;

$$\mathcal{BA}x = \mathcal{B}b \quad \approx \quad Ix = \mathcal{A}^{-1}b.$$

To use with the conjugate gradient method, $\mathcal{B}$ must be symmetric, positive–definite, in order for $\mathcal{BA}$ to be SPD. Of course, $\mathcal{B}$ must also be much cheaper to find than $\mathcal{A}^{-1}$ – if not we could just as well have run the iterative solver without preconditioning. For the preconditioned conjugate gradient method we remark that one does not actually build $\mathcal{B}$, but one must be able to compute the effect of applying $\mathcal{B}$ to a given vector r. For preconditioning to be cost–effective, the improvement of the convergence rate must outweigh the cost of computing $\mathcal{B}r$ once per CG iteration. Further information on the CG algorithm and how it is preconditioned can, as for other details on the method, be found in [8], [17] and [19].

The preconditioner options listed in the aforementioned literature classifies naturally into two categories; those preconditioners based on *classical iterative methods* or those based on *incomplete factorization*. We will not detail all the different methods that fall into each category, only point out the possible choices, based on our implementation platform (Diffpack or Pysparse).

Symmetric Successive Over–Relaxation (SSOR)

Of the classic iterative methods we may only use Jacobi iterations or Symmetric Successive Over–Relaxation (SSOR) iterations, as they both ensure that the matrix of the preconditioned system continues to be symmetric positive–definite. The merits of these two methods as preconditioners are briefly discussed in [8, C.3.2], and based on this we see SSOR as the preferred classic iterative preconditioner. When the classic iterative methods are used as preconditioners, one usually does exactly one (SSOR) iteration and use the resulting matrix as the preconditioner $\mathcal{B}$. The `precon` module in Pysparse only offer classic iterative methods as preconditioners, so SSOR will be used when solving the equation in a Pysparse setting (see chapter 3.4 on the facing page).

Relaxed Incomplete LU Factorization (RILU)

Of the incomplete factorization methods, the conclusion in [8, C.3.3] lean toward *Relaxed Incomplete LU Factorization* (RILU) as the preferred preconditioner for the conjugate gradient method. RILU is not available in the Pysparse module, so this preconditioner will only be used when Diffpack also provides the iterative solver. The basic idea is that we set $\mathcal{B} = \mathcal{A}$ and use Gaussian elimination to split $\mathcal{B}$ into an upper and lower triangle. To avoid destroying the sparsity pattern in $\mathcal{A}$ by fill–ins from the factorization, we multiply all candidate fill-ins with a relaxation parameter $\omega \in [0,1]$, and add them to the main diagonal instead. RILU usually performs well as a preconditioner, although adjustments in the choice of ω must probably be done for each initial-boundary value problem.

3.3 The heat equation and Parareal algorithm

The heat equation is a *parabolic* PDE, and the Parareal algorithm is unconditionally stable for most time discretizations of parabolic equations [2]. It is also a stiff problem as was discussed in chapter 2.2.1 on page 10, and hence we have from [20] that the algorithm should be stable for any choice of coarse propagator with the property

$$\lim_{z\to-\infty} R(z) = \frac{1+\theta z}{1-(1-\theta)z} \leq \frac{1}{2},$$

where $R(z)$ is the stability function. As previously mentioned the above criteria leads to $\theta \in [2/3, 1]$. Needless to say, the algorithm will not be stable if the the discretization for the fine propagator is unstable. Based on chapter 2.2 on page 9 we find that $z = \Delta t K$, where K is the stiffness matrix defined in chapter 3.1.2 on page 17.

Figure 3.1 and Figure 3.2 show how the error compared to the solution from a serial computation using the fine propagator develops as the number of Parareal iterations increase. The significance of the plots will become clearer after chapter 5 on page 54. In Figure 3.1 the Crank–Nicolson scheme ($\theta = 1/2$) is used for the coarse solver, and in Figure 3.2 $\theta = 2/3$. The fine solvers in both plots use Implicit Euler, which implies that they will be stable for any chosen time step. The Crank–Nicolson scheme clearly generates a form of instability, as is expected from the analysis. We remark that solving the heat equation using Crank–Nicolson and the coarse propagator (i.e. not using Parareal) is stable. Therefore it is not any instability in the coarse solver itself that prevents the algorithm from converging like it should.

You may also note that even though we claim Figure 3.1 to be unstable, it does actually converge as long as you run the maximum number of iterations. This is a property of the

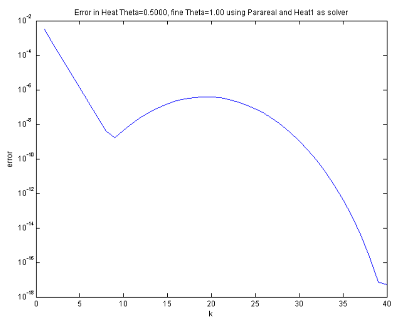

Figure 3.1: Parareal is not stable when Crank–Nicolson is used for the coarse solver. The fine solver is stable, as it uses Implicit Euler.

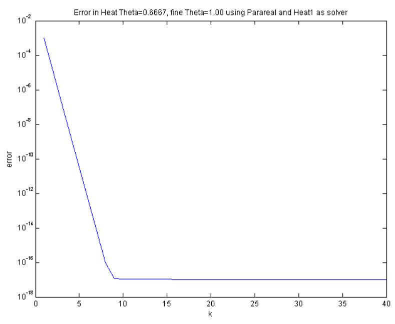

Figure 3.2: Parareal shows exponential convergence toward machine precision at $\theta = 2/3$. The fine solver is Implicit Euler.

instability in the Parareal algorithm that deviates from the traditional definition of instability, where taking one more time step only causes the numerical solution to be further removed from the analytical solution. The instability properties of Parareal will be discussed further in chapter 5 on page 54 and chapter 6 on page 70.

3.4 Implementation

Following our implementation strategy, a combination of Python and Diffpack programming was used to implement the heat equation, in order to use Parareal to solve the heat equation. Our reasons for implementing Parareal in Python will be discussed in chapter 8.1 on page 98.

The implementation phase of the heat equation served as a testing ground for the later implementation of Stokes equations, and as such the usage of the heat equation as a test model permeates most aspects of the work on this thesis. The heat equation was central during the initial scrutiny of the possibility to smoothly combine *Microsoft Visual Studio*, Python and *SWIG* during development, though naturally new techniques were discovered during the whole development phase. The results are described in chapter 7 on page 87. The heat equation also played a role during the development of the *Pysparse*↔*Diffpack* filter, which is discussed in appendix A on page 107. In the current section we will present a class hierarchy that support the interface requirements set by the Parareal implementation (see chapter 5.2 on page 63 for details). To summarize, the implementation of the heat equation served several purposes, by being a basis for

- testing Windows development tools
- finding and testing techniques that allow Diffpack to be a pure matrix generator, letting Python handle control of program flow.

- testing the Parareal algorithm implementation

3.4.1 Defining the model problem

For this thesis implementation we solved the heat equation over the domain $\Omega \in [0,1] \times [0,1]$, $x \in \mathbb{R}^d$, $d = 2$, and used the following boundary and initial conditions

$$\begin{aligned} g(x,t) &= 0, & x \text{ on } \partial\Omega_E \\ u_0 &= \prod_{r=1}^{d} \sin(\pi x_r), & x \text{ in } (\Omega \cup \partial\Omega) \end{aligned}$$

and set the forcing term $f(x)$ to

$$f(x) = 0.$$

The exact solution of (3.1) is then

$$u(x,t) = e^{-d\pi^2\kappa t} \prod_{r=1}^{d} \sin(\pi x_r), \quad x \in \mathbb{R}^d,\ d = 2, \tag{3.9}$$

where the subscript r indicates space dimension r of x, and κ is a variable that can be changed by program input. By knowing the exact solution to your particular test problem during implementation obviously allows you to measure the total *error* of your discrete solution. You can then use this to verify that the discrete solution is not unreasonable, and that the error is within the bounds suggested by error analysis. These factors will then indicate whether the implementation is correct. We will not discuss error analysis here, but suffice it to say that any discretization method introduces a certain error which is dependent on the spatial grid resolution and the time–step Δt, and by studying this error one can find how the error depends on these parameters. Error analysis is discussed in for example [8, A.4].

3.4.2 The heat class hierarchy

The heat class hierarchy is split into two groups; simulator classes that inherit from the fundamental Diffpack heat implementation, and a set of *factory* objects that conform to a interface requirements set by the Parareal Python implementation. The hierarchy is outlined in Figure 3.3, though we will discuss the hierarchy to some extent in the following.

The Diffpack `Heat1` implementation and Python interface class

The core of the heat equation hierarchy is based on the existing `Heat1` implementation that ships with Diffpack. The class is a standard (simple) time dependent Diffpack finite

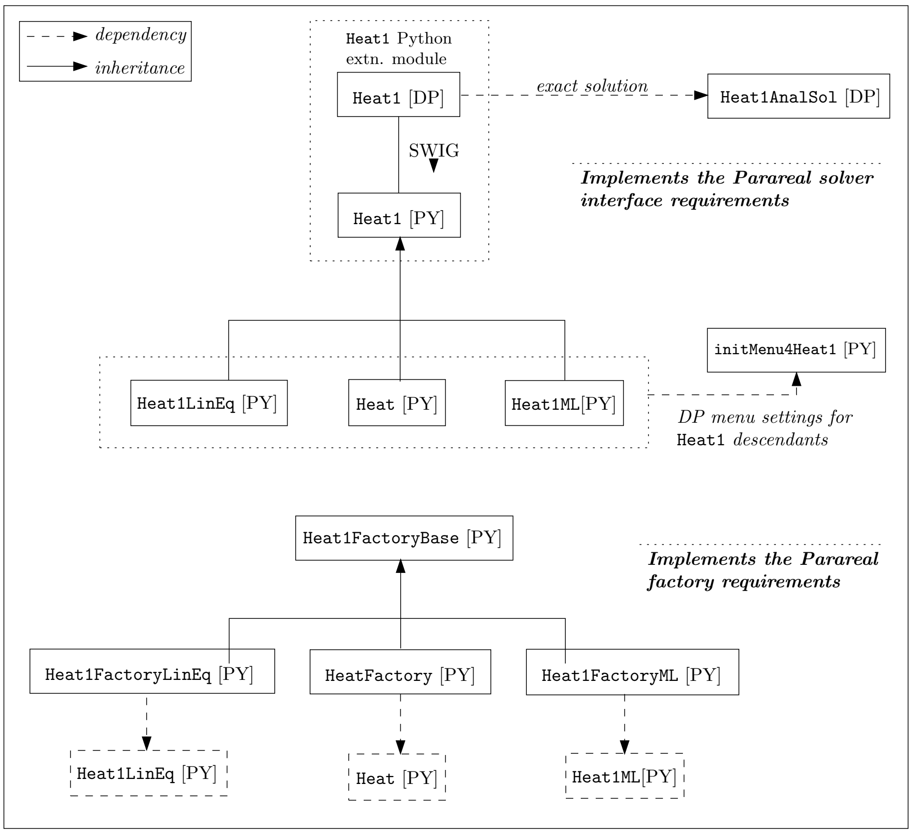

Figure 3.3: The heat class hierarchy ([DP] and [PY] refer to implementation in Diffpack or Python, respectively).

element simulator, though a fairly detailed documentation can be found in [8, chapter 3.10]. `Heat1AnalSol` is a functor class that implements the exact solution in (3.9), and `Heat1` uses it to set the initial condition plus to produce error estimates if necessary. In [11], a SWIG interface file[3] for `Heat1` was fashioned to create the `Heat1` *Python extension module* of the simulator. We therefore have two `Heat1` implementations; one done by use of Diffpack and one is a Python interface class. For clarity we will use `Heat1`[DP] and `Heat1`[PY] to distinguish between which level of abstraction we are discussing.

The main change to the `Heat1`[DP] is adding support of the θ–rule to the `integrands()` function as shown in Listing 3.1 on the following page. It is mainly the construction of the right hand side of the linear equation – `elmat.b(i)+=`$\cdots$ – that has seen any noteworthy changes. This is natural, as it is the right hand side that is most affected when one goes from implicit euler ($\theta = 1$), which was the original implementation, to adding support for $\theta \leq 1$. That the right hand side is markedly changed when $\theta \neq 1$ should be clear by studying

[3]See chapter 7.3.2 on page 94 for further details on SWIG and interface files

the final discrete equations at the end of chapter 3.1.2 on page 17. A very minor change that deviates somewhat from the Diffpack coding standard promoted by [8], which will simplify Parareal adaption, is moving the call to `setIC()` from the `timeLoop()` function to `solveProblem()`. This is merely to accommodate the extensions done for Parareal (chapter 5.2.1 on page 67), since one must then be able to provide a separate “initial condition” depending on which sub domain in time the solver is called for.

Listing 3.1: The `Heat1::integrands()` function with θ-rule discretization

```
void Heat1::integrands (ElmMatVec& elmat,
                        const FiniteElement& fe){
  const real detJxW      = fe.detJxW();
  const int  nsd         = fe.getNoSpaceDim();
  const int  nbf         = fe.getNoBasisFunc();
  const real dt          = tip->Delta();
  const real t           = tip->time();
  real gradNi_gradNj     = 0.0;
  real gradNi_gradup     = 0.0;
  const real dtThetaK    = dt*theta*k(fe, t);
  const real dtThetaMk= dt*(1.0-theta)*k(fe, t);
  const real fDtTheta    = f( fe, t ) * dt * theta;

  real up_pt = u_prev->valueFEM(fe); //u_prev at this itg.pt
  Ptv(NUMT) gradup_pt(nsd);          //grad u_prev
  u_prev->derivativeFEM( gradup_pt, fe );

  for(int i = 1; i <= nbf; i++) {
    for(int j = 1; j <= nbf; j++) {
      gradNi_gradNj =0.0;
      for(int s = 1; s <= nsd; s++)
        gradNi_gradNj += fe.dN(i,s)*fe.dN(j,s);

      elmat.A(i,j) += (fe.N(i)*fe.N(j) +\
                      dtThetaK*gradNi_gradNj)*detJxW;
    }
    gradNi_gradup = 0.0;
    for( int s = 1; s <= nsd; s++ )
      gradNi_gradup += dtThetaMk*gradup_pt(s)*fe.dN(i,s);

    elmat.b(i)+= fe.N(i)*(fDtTheta + up_pt)*detJxW -\
                 gradNi_gradup*detJxW;
  }
}
```

The SWIG interface file was extended so that `Heat1`[PY] would support the simulator requirements set by the Parareal Python implementation. It was also extended in such a way that `Heat1`[PY] would be equipped with the means to be a *linear system provider* at any given time point. Our implementation strategy for the Stokes equations is to use the finite element aspect of the Diffpack library to build the linear system at each time step, and by filtering the Diffpack data into Python, use Python modules to solve the (sparse) linear system arising from the discretization. More specifically, we want to filter the Diffpack classes into the sparse matrix environment provided by Pysparse (a Python package), and use the linear solvers in Pysparse to solve the ensuing system. The extensions to the Python version of the `Heat1` class interface will be a means of piloting this strategy. We will not detail all the extensions done to the class through SWIG here, only those relevant to the linear system provider strategy. Extensions done to match Parareal requirements are covered in chapter 5.2.1 on page 67. The central extensions for providing the linear system is sketched in Listing 3.2 on the following page.

The `getMatrix()` function need only be called once, as the matrix is static for all time points. The returned matrix is the global matrix assembled from all the `elmat.A(i,j)` entries in the `Heat1::integrands()` function. The matrix is returned as a `Matrix_double` pointer. It can be filtered to Pysparse using the `Dp2Pysparse` extension module. `getRHS()` returns the right hand side vector of the linear equation system, and filtering can be done using the `Dp2Numeric` extension module. The Pysparse package only offers its own matrix format, and uses the `array` object in the well–known Numeric (NumPy) module[4] to handle vectors. Both `Dp2Pysparse` and `Dp2Numeric` are described in appendix A on page 107.

As the right hand side is, through the `integrands()` function in `Heat1`[DP], dependent on the current and previous solutions, the SWIG interface must offer a mechanism for updating these with new values. If the current and previous solutions do not change, we practically have a steady–sate problem as the linear system will no longer change over time. Hence we have `setSolution()` which accepts a standard Diffpack vector object. Said vector object can be created out of a Numeric `array` object through the filter options in `Dp2Numeric`. The current solution in the Diffpack solver can be copied into the object holding the solution at previous time step using `shiftSolution()`. Typically you would, in Python, first call `shiftSolution()` and then `setSolution()`.

To trigger the creation of the linear system at a particular time step you would call `makeSystem()`. The boolean is a flag indicating whether you want the system matrix to be built as well as creating a new right hand side. You should only need to build the matrix once, before looping over the time steps.

[4]Numeric and other relevant open–source Python modules are discussed in chapter 7.1 on page 88

Listing 3.2: Excerpts from SWIG interface file for `Heat1`

```
%module Heat1
%{
#include "Heat1.h"
%}

class Heat1 : public FEM{
    /*
    SWIG definition of Heat1 is identical to Diffpack
    definition, but all Handle macros must be expanded,
    i.e. we have variables on the form
    Handle_FieldFE u;
    */
};

%extend Heat1{

    //Matrix generator for linear system provider
    Matrix_double* getMatrix();

    //RHS generator for linear system provider
    Vec_double& getRHS();

    //Sets current solution to newSol
    void setSolution( Vec_double& newSol );

    //Shifts u int u_prev
    void shiftSolution();

    //Builds lin.sys. compute_A = false -> only build RHS
    void makeSystem( bool compute_A );
}
```

The Python subclasses

In the process of developing and testing possibilities, three subclasses of `Heat1`[PY] were created, although only one follows the scheme wanted for the Stokes implementation. All three subclasses are depicted in the class hierarchy in Figure 3.3 on page 25. When creating these classes it was not only a goal to be able to solve the heat equation through Python, but also to have the classes implement the interface demands set on any solver to be used by the Parareal algorithm.

`Heat1LinEq` was the first solver adapted for Parareal, and was actually developed more for testing the Parareal implementation itself. As most of the adaptation is done at SWIG interface file level, it offers a very minimal extension to `Heat1`[PY]. The class name refers to how the Diffpack class `LinEqAdmFE` handles the iterative solver used to solve the linear system – which is set to the preconditioned conjugate gradient method via `initMenu4Heat1` as discussed below. Since the skeleton of the SWIG interface file was available at the very beginning, this class existed early in the development phase, and as such it was also at one point a matrix provider for testing the `Diffpack2Pysparse` filters and any other C++–to–Diffpack related topic presented in chapter 7 on page 87. Because of its role in testing the Parareal equation this class is quite adequately documented in chapter 5.2.1 on page 67. The reasoning behind the constructor is however not covered in 5.2.1, as it is not relevant for the discussion there. As the same strategy concerning the `MenuSystem` object has been used for all Python simulators in this thesis, both forheatt and Stokes, we will provide some of the background for the chosen solution here. Different options in a Diffpack simulator are set through a `MenuSystem` object. Every Diffpack simulator has a `define()` function that associates the `MenuSystem` object to the simulator and determines the possible menu options for the simulator. `scan()` is used to set the chosen options for the simulator. `define()` can be called only once per `MenuSystem` instance, whereas `scan()` can be called an arbitrary number of times. This implies that several instances of the same simulator may share a `MenuSystem` object, but instances of *different* simulators may not. To change a particular default option set by `define()` in a menu object through program code one uses the `set()` method. `initMenu4Heat1` is a Python class provided as a means to change the default options and "groom" the `Heat1`[DP] solver to function with the Python classes. `initMenu4Heat1` is a callable object that accepts a Python wrapped[5] `MenuSystem` instance and sets the desired options in the menu.

A clean way to handle the `MenuSystem` would be to equip each solver instance with a unique `MenuSystem` object, and then use `initMenu4Heat1` to set the options. However, there is some trouble when creating and using new `MenuSystem` objects that cause the Diffpack engine to shut down. The `global_menu` object is however available, and can be used instead. As it was not overly important to make the `MenuSystem` issue work in an ideal

[5]See appendix A on page 107

Listing 3.3: The Heat1LinEq constructor

```
import Heat1
import Diffpack2Python

class Heat1LinEq(Heat1.Heat1):
    """
    Class adjusts the interface to fit the Parareal alg.
    Uses the Diffpack LinEqAdmFE to solve the linear system.
    """
    def __init__( self, menuInitializer,
                  verbose=False, save=False ):
        """
        menuInitializer is callable; accepts MenuSystem
        and uses it to set desired (non-default) options.
        """
        Heat1.__init__( self )
        self.menu = Heat1.global_menu
        if not isinstance(self.menu, Diffpack2Python.MenuSystem):
            #wrap the object
            self.menu = Diffpack2Python.MenuSystemPtr(self.menu)

        #if not menu.defined:
        if menu.empty():
            self.define( self.menu )
        else: self.attachMenu( self.menu )
        menuInitializer( self.menu )
        self.scan()
```

way in order to study the interaction between Parareal/Heat and Parareal/Stokes, we opted for the less satisfactory solution shown in the constructor in Listing 3.3 on the preceding page. All objects in the `Heat1` extension module can use the same `MenuSystem` object, and this structure will therefore not create any conflicts, as long as you do not pass the `Heat1` global menu to objects outside the extension module. The function `attachMenu()` called as an alternative to `define()` is a SWIG created extension similar to those in Listing 3.2 on page 28 that associates the `MenuSystem` object to the simulator without defining menu items.

`Heat1ML` had a single purpose, which was to test the Python version of the ML algebraic multigrid (AMG) package, and make sure that we were able to use it on Windows. As such, the class is a “proof of concept” in that the module could indeed be used to solve a linear system in the given Python setting. The `ml` extension module is not officially available[6], and had not been tested on Windows, so the class was designed to test the module and fix possible Unix related issues. It was also a means to study and document the Python interface of the module. The results from this testing is documented in appendix C on page 115. `Heat1ML` uses algebraic multigrid to solve the linear system at each time step, which, unless you have a large number of nodes, will be very inefficient compared to the solvers employed by `Heat1LinEq` or `Heat`.

The Python ML module uses the matrix formats in the Pysparse package, and in `Heat1ML` Diffpack functions as a pure linear system provider: a Python module independent of Diffpack solves the linear system after the Diffpack data has been filtered into Python objects. Multigrid is well suited to solving *elliptic* problems, and does so efficiently – provided that the linear system is large enough. This is actually the reason for our interest in the module, though we have only discussed parabolic PDEs so far. As will become clear in chapter 4.2.1 on page 42, the Stokes preconditioner is composed of blocks of the *inverse of elliptic operators*, with the consequence that one must apply a linear solver *in order to find the preconditioner*. Based on the ML module’s close link with Pysparse and its proficiency in handling elliptic operators it is a natural tool to employ for calculating each block in the preconditioner. The implementation of `Heat1ML` is too large to show in its entirety here, but the class interface is displayed in Listing 3.4 on the next page to give an inkling of how it has been done.

`Heat` is the full test example of the Diffpack$\leftrightarrow$Python implementation of Stokes problem. Like `Hea1tML` it takes full use of the linear system providing functions outlined in Listing 3.2 on page 28 to create or refresh the linear system for each time step, but it uses the iterative solvers in `itsolvers` module of the Pysparse package to solve the system. One of the iterative methods accompanying the module is the preconditioned conjugate gradient

[6]The author, Roman Geus, wrote the module for his own work, and has of yet no immediate plans of making it publicly available.

Listing 3.4: The Heat1ML class interface

```
from Heat1 import *
import Diffpack2Python, ml
from NumWrap import NumWrap

class Heat1ML(Heat1):
    """
    Class adjusts the interface to fit the Parareal alg.
    Uses ML and Pysparse to solve the linear system.
    """

    def __init__( self, menuInitializer, MLTolerance=1.0e-8,
                  verbose=False, save=False ):
    def _fillEssBC( self ):
    def _initLinearSystem( self ):
    def _solveTimeDomain( self, startSolution ):
    def getSolution(self):
    def getIC( self ):

    def solveTimeDomain( self, *args ):
        """
        Solves the Heat1 equation using current timestep
        and startSolution as initial cond. 3 overloads:
            solveTimeDomain( startSolution )
               Solves Heat equation in current timedomain

            solveTimeDomain( tStart, tStop, startSolution )
               Solves Heat equation in [tStart, tStop].

            solveTimeDomain( tStart, tStop )
               Solves in [tStart,tStop] using the system IC.
        """

    def _solveAtThisTimeStep( self ):
        """
        Like solveAtThisTimeStep() in Diffpack, but uses ML
        to solve the linear equation system.
        """
```

method, which is, as concluded in chapter 3.2.1 on page 20, our preferred iterative solver for this particular problem. The Pysparse package also has a precondition module `precon` with preconditioners that merge seamlessly with functions from `itsolvers`. The module only implements the classic iterative methods Jacobi and SSOR, and based on the earlier conclusions we use SSOR to precondition the system in `Heat`. The class interface is identical to that of `Heat1ML` as listed in Listing 3.4 on the facing page, and it involves the same filtering back and forth between Python and Diffpack at each time step to update the linear system.

The factory classes

The Parareal module uses *factory classes* to present a generic interface for creating new coarse and fine solver instances which it will use when running the algorithm. We once again refer to the Parareal chapter for further details. The factory hierarchy presented in Figure 3.3 on page 25 is very simple; we have a factory base class that implements the expected public interface, whereas the subclasses specify the class type of the returned solver instance, i.e. `Heat1LinEq`, `Heat1ML` or `Heat`.

The Parareal algorithm opens for using different simulators for the fine and coarse solvers, and the factory classes accommodates this by giving you the option of using distinct θ–values for the fine and coarse solvers. The class interface for the base class and one subclass (the subclasses are all similar) is displayed in Listing 3.5 on the following page. This concludes the chapter on the heat equation. The following chapter uses the same basic techniques as we have discussed here, but just as the Stokes equations are more complex than the heat equation, there are an increased number of factors to consider when discretizing and implementing them. As the implementation for the heat equation has been covered in such detail here, the Stokes chapter will merely give an overview over the elements that strongly differ from the strategies used for hte heat implementation. It is also assumed that with the details of this chapter it will be relatively straight forward to explore the actual source code.

Listing 3.5: The Heat1FactoryBase and HeatFactory interfaces

```
class Heat1FactoryBase:
      """
      *gridResolution is the number of unknowns in each
      spatial direction, x[,y,z].
      """
    def getFineSolver( self, tStart, tStop,
                       timeStep, *gridResolution ):

    def getCoarseSolver( self, tStart, tStop,
                         timeStep, *gridResolution ):

# ====================================================== #

class HeatFactory(Heat1FactoryBase):
    """
    Factory class for the Heat1 simulator, using Pysparse
    modules as solver. *gridResolution is assumed to be
    [xNodes, yNodes].
    """
    def __init__( self, menuInitializer, theta,
            fineTheta=None, verbose=False, save=False ):

    def _getFineSolver( self, *args ):
    def _getCoarseSolver( self, *args ):

    def _getSolver( self, tStart, tStop, timeStep,
                    xNodes, yNodes ):
      """
      Resets menuInitializer settings according to input
      variables and uses it to create a Heat() instance.
      """
```

Chapter 4

The Time Dependent Stokes Equations

The following chapter introduces the time dependent Stokes equations with the associated discretization and implementation. It is largely based on the material on the steady–state Stokes problem discussed in [9] and [12]. Particularly, we will build the implementation of the unsteady Stokes equation on the steady–state class hierarchy developed for [9]. Note that we often will skip the "unsteady" and "time–dependent" identifiers for brevity when discussing the equations in this chapter, and will rather take care to use the terminology "steady–state" when needed.

To accentuate the similarities and differences between the techniques employed for the heat equation and those used for the Stokes equations, we will to the extent possible use the same structure as chapter 3. Due to the more complex nature of the Stokes equations compared to the heat equation, the topics introduced in the previous chapter will be somewhat expanded as we progress. Increased complexity introduces new issues that must be handled. The principal intent of this chapter is to arrive at a solver that can be used with the Parareal algorithm, just as we did for the heat equation.

4.1 Mathematical Model

The unsteady Stokes equations describe creeping flow, typically very viscous liquids or liquids with low velocity. The equations are a subset of the renown Navier–Stokes equations for incompressible fluid flow,

$$\frac{\partial v}{\partial t} + v \cdot \nabla v - \nu \nabla^2 v + \frac{1}{\rho} \nabla p \;=\; f \tag{4.1}$$

$$\nabla \cdot v \;=\; 0. \tag{4.2}$$

Here the velocity, $v(x,t)$, and the pressure, $p(x,t)$, are the unknowns. Due to high viscidity or very low velocity, the convection term $v \cdot \nabla v$ is close to zero, and we may ignore it. We are then left with (after scaling) the time–dependent Stokes equations (4.3)–(4.4). The full initial–boundary value problem can be formulated as

$$\begin{aligned}
\frac{\partial v}{\partial t} - \nabla^2 v + \nabla p &= f, \quad \text{in } \Omega, \qquad 0 < t && (4.3)\\
\nabla \cdot v &= 0, \quad \text{in } \Omega, \qquad 0 < t && (4.4)\\
v &= h, \quad \text{on } \partial\Omega_E, \quad 0 \leq t && (4.5)\\
-\nabla v \cdot \eta + p\eta &= 0, \quad \text{on } \partial\Omega_N, \quad 0 \leq t && (4.6)\\
v(x,0) &= v_0, \quad \text{in } \Omega, \qquad t = 0 && (4.7)\\
p(x,0) &= p_0, \quad \text{in } \Omega, \qquad t = 0, && (4.8)
\end{aligned}$$

where $x \in \mathbb{R}^d$ and d is the number of dimensions. $f(x,t)$ represent body forces and η is the unit normal vector pointing out of Ω. The boundary $\partial\Omega = \partial\Omega_N \cup \partial\Omega_E$ is made up of the natural and essential boundary conditions ((4.6)–(4.5)), respectively, and the initial state of the system is given by v_0 and p_0.

The *equation of motion* (4.3), is a *vector* equation of d dimensions, whereas (4.4), which governs *mass conservation*, is a *scalar* equation. (4.4) is also often referred to as the incompressibility or continuity constraint. Consequently, the Stokes equations are a set of $d+1$ equations with $d+1$ unknowns, as opposed to the heat equation which is a single, scalar equation with a single unknown – the temperature u. For clarity, we can write the equations in their component form. For two dimensions this reads

$$\begin{aligned}
\frac{\partial v_x}{\partial t} - \left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2}\right) + \frac{\partial p}{\partial x} &= f_x\\
\frac{\partial v_y}{\partial t} - \left(\frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2}\right) + \frac{\partial p}{\partial y} &= f_y\\
\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} &= 0.
\end{aligned} \tag{4.9}$$

4.1.1 Temporal discretization

As was discussed to some length for the heat equation, discretization by use of the θ–rule makes it trivial to change the the temporal discretization technique by adjusting θ in the range $[0,1]$. By doing so, we may study how the stability properties of Parareal respond to the different θ values to give us an indication of how Parareal performs on this type of problem. In practice, on would typically only use either one of the Euler methods, or the Crank-Nicolson scheme, since these are the only "proper" methods in the θ–rule scheme, but for our purposes it makes sense to provide the whole range of θ–values. This is particularly true, since the analysis of Parareal suggests that it is well behaved (stable) when solving parabolic equations – as long s $\theta \in [2/3,1]$. The interesting question is whether this θ–range is valid for the Stokes equations, or if ODE aspects that are not present for the heat equation will force extra considerations into the stability analysis. A possible new factor in the stability analysis could be that the Stokes equations are a set of Differential Algebraic Equations (DAE)[1] when they are semi–discretized in space.

We apply the θ–rule to (4.3)–(4.4), and have

$$\begin{aligned} v^\ell - \Delta t\theta\left(\nabla^2 v^\ell - \nabla p^\ell\right) = \;& \Delta t\theta f^\ell + v^{\ell-1} + \Delta t(1-\theta) f^{\ell-1} + \\ & \Delta t(1-\theta)\left(\nabla^2 v^{\ell-1} - \nabla p^{\ell-1}\right) \end{aligned} \tag{4.10}$$

$$\nabla\cdot v^\ell = \; 0. \tag{4.11}$$

As for the heat equation, we are now in possession of a system of semi–discretized equations that have the same boundary and initial conditions as the continuous system.

To make the upcoming spatial discretization expressions clearer, we will do a slight modification to the notation of the semi–discrete equation (4.10). Since v^ℓ and f^ℓ are vectors, we can express them as

$$v^\ell = \sum_{r=1}^{d} v^{\ell,r}, \quad f^\ell = \sum_{r=1}^{d} f^{\ell,r},$$

where the superscript r denotes the r-th component in v^ℓ and f^ℓ. Note that it will also indicate the r-th component of the vector equation (4.3)[2], and we can then write (4.10) as

$$\begin{aligned} v^{\ell,r} - \Delta t\theta\left(\nabla^2 v^{\ell,r} - \frac{\partial p^\ell}{\partial x^r}\right) = \;& \Delta t\theta f^{\ell,r} + v^{\ell-1,r} + \Delta t(1-\theta) f^{\ell-1,r} + \\ & \Delta t(1-\theta)\left(\nabla^2 v^{\ell-1,r} - \frac{\partial p^{\ell-1}}{\partial x^r}\right) \end{aligned} \tag{4.12}$$

$$\nabla\cdot v^\ell = \; 0, \tag{4.13}$$

for $r = 1,\ldots,d$. With this notation, it is easy to separate the vector equations from the scalar equation, and the parallel to (4.9) should be clear. In the following sections we will use v^ℓ when referring to the unknown velocity in general, and $v^{\ell,r}$ when discussing operations on the separate components of v^ℓ. Note that we can write (4.13) as

$$\nabla\cdot v^\ell = \sum_{r=1}^{d} \partial v^r/\partial x^r,$$

which corresponds to (4.9).

[1]Differential Algebraic Equations will be discussed in chapter 4.3 on page 46

[2]Comma in the superscripts merely separate indices and are not related to Einstein's summation convention or compact notation for differentiation ([8, A.3]).

4.1.2 Spatial discretization

Once more, we will use the finite element method described in Chapter 2 to discretize the spatial domain. Using the straightforward approach to spatial discretization that was used for the heat equation will however give non-physical oscillations in pressure, and one must look for strategies to overcome this problem. The details of why the solution produces oscillations are beyond the scope of this thesis, but it is related to how the pressure is not uniquely determined, which gives ambiguity to the system. In addition, it is only determined up to a constant, c.f. [9], [12].

A technique to correct the oscillation problem is to use so called *mixed elements*, where different basis functions are used for the different unknowns. One has, for example, the Taylor-Hood element, which uses quadratic basis functions for the velocity and linear basis functions for pressure. Mixed elements were used in the simulators developed for [9], which we shall use to form the base classes for our time dependent implementation. We will therefore use use mixed elements in our spatial discretization, in order to use the predefined steady–state solvers, and consequently this section is principally based on [9].

Another group of common techniques for handling the pressure oscillations employs perturbations of (4.4), such that it reads $\nabla \cdot v = z(p)$, where $z \neq 0$ and dependent on p, say via $z = \nabla^2 p$. Naturally, by using a different strategy for handling the pressure oscillations, one would employ discretization strategies that differ from the one we present below, but the basic shape of the resulting linear system is nevertheless the same. In fact, they will all be saddle point problems. When formulating the finite element expressions for the Stokes equations, we will end up with a saddle point problem due to the continuity constraint [3, ch. 3.5]. A quick introduction to the different techniques as used on the Navier–Stokes and steady Stokes equations can be found in [12] and [9], respectively.

To proceed, we approximate the unknowns as sums over weighted basis functions:

$$\begin{aligned} v^\ell &\approx \hat{v}^\ell = \sum_{r=1}^{d} \hat{v}^{\ell,r} = \sum_{r=1}^{d}\sum_{j=1}^{n_v} v_j^{\ell,r} N_j \\ p^\ell &\approx \hat{p}^\ell = \sum_{j=1}^{n_p} p_j^\ell L_j. \end{aligned}$$

For our mixed element formulation, we have used N_i and L_i as the basis functions for the discrete formulations of (4.12) and (4.13), respectively. The unknowns $\{v_j^{\ell,r}\}$ and $\{p_j^\ell\}$ are represented as vectors,

$$\begin{aligned} \overline{v}^\ell &= \{\overline{v}^{\ell,r}\} = \left[v_1^{\ell,1}, v_2^{\ell,1}, \ldots, v_{n_v}^{\ell,1}, v_1^{\ell,2}, \ldots, v_{n_v}^{\ell,2}, \ldots, v_1^{\ell,d}, \ldots, v_{n_v}^{\ell,d}\right]^T \\ \overline{p}^\ell &= \left[p_1^\ell, p_2^\ell, \ldots, p_{n_p}^\ell\right]^T. \end{aligned} \tag{4.14}$$

Note the distinction between $\overline{v}^\ell$, which represents a "vector of vectors" and $\overline{p}^\ell$, which is a standard vector.

The Galerkin equations for (4.10)–(4.11) are given by

$$\int_\Omega \left[\hat{v}^{\ell,r} - \Delta t\theta\left(\nabla^2\hat{v}^{\ell,r} - \frac{\partial\hat{p}^\ell}{\partial x^r}\right)\right] N_i \, d\Omega = \int_\Omega \hat{g}_i^{\ell,r} \, d\Omega, \qquad i = 1,\dots,n_v$$

$$\int_\Omega \nabla\cdot\hat{v}^\ell L_i \, d\Omega = 0, \qquad i = 1,\dots,n_p,$$

for $r = 1,\dots,d$. To make the expressions more compact, we have used

$$\hat{g}_i^{\ell,r} = \left[\Delta t\theta f^{\ell,r} + \hat{v}^{\ell-1,r} + \Delta t(1-\theta)\left(f^{\ell-1,r} + \nabla^2\hat{v}^{\ell-1,r} - \frac{\partial\hat{p}^{\ell-1}}{\partial x^r}\right)\right] N_i.$$

Using Green's Lemma as defined in (2.1.1) on the integral arising from the equation of motion, we lessen the smoothness criterions on the basis functions associated with $\hat{v}^\ell$ and $\hat{p}^\ell$. The resulting integral expression reduces to

$$\int_\Omega \left[\hat{v}^{\ell,r}N_i + \Delta t\theta\nabla\hat{v}^{\ell,r}\cdot\nabla N_i - \Delta t\theta\,\hat{p}^\ell\,\frac{\partial N_i}{\partial x^r}\right] d\Omega = \int_\Omega \hat{g}_i^{\ell,r}\, d\Omega,$$

since the integrands over the boundary cancel each other due to the open boundary condition (4.6):

$$\int_{\partial\Omega} \Delta t\theta\left(-\nabla\hat{v}^{\ell,r}\cdot\eta^r + \hat{p}^\ell\eta^r\right) N_i \, d\Gamma = 0.$$

A similar treatment of $\hat{g}_i^\ell$ reduces $\nabla^2 v^{\ell-1}$ and $\nabla p^{\ell-1}$ in a correspondingly, such that the components of $\hat{g}_i^\ell$ read

$$\begin{aligned}\hat{g}_i^{\ell,r} = \Big[&\left(\Delta t\theta f^{\ell,r} + \hat{v}^{\ell-1,r}\right)N_i + \\ &\Delta t(1-\theta)\left(f^{\ell-1,r}N_i - \nabla\hat{v}^{\ell-1,r}\cdot\nabla N_i + \hat{p}^{\ell-1}\frac{\partial N_i}{\partial x^r}\right)\Big].\end{aligned} \tag{4.15}$$

We see that N is now only required to be once differentiable, whereas L does not need to be differentiable at all. Note that we can multiply (4.13) with a factor $-\Delta t\theta$ without changing the overall solution to the system, such that we have

$$\int_\Omega \nabla\cdot\hat{v}^\ell L_i\, d\Omega = -\int_\Omega \Delta t\theta\nabla\cdot\hat{v}^\ell L_i\, d\Omega = -\sum_{r=1}^{d}\int_\Omega \Delta t\theta\nabla\cdot\hat{v}^{\ell,r}L_i\, d\Omega = 0.$$

Replacing $\hat{v}^\ell$ and $\hat{p}^\ell$ with their respective weighted sums we have a linear system:

$$\sum_{j=1}^{n_v}\left(M_{ij}+K_{ij}\right)v_j^{\ell,r} + \sum_{j=1}^{n_p} A_{ij}^r\, p_j^\ell = c_i^{\ell,r}, \quad i = 1,\dots,n_v,\ r = 1,\dots,d, \tag{4.16}$$

$$\sum_{r=1}^{d}\sum_{j=1}^{n_v} B_{ij}^r\, v_j^{\ell,r} = 0, \quad i = 1,\dots,n_p, \tag{4.17}$$

where

$$
\begin{aligned}
M_{ij} &= \int_\Omega N_j N_i \, d\Omega \\
K_{ij} &= \int_\Omega \Delta t \theta \nabla N_j \cdot \nabla N_i \, d\Omega = \int_\Omega \Delta t \theta \left(\sum_{k=1}^{d} \frac{\partial N_j}{\partial x^k} \frac{\partial N_i}{\partial x^k} \right) d\Omega \\
A^r_{ij} &= -\int_\Omega \Delta t \theta \frac{\partial N_i}{\partial x^r} L_j \, d\Omega \\
B^r_{ij} &= -\int_\Omega \Delta t \theta \frac{\partial N_j}{\partial x^r} L_i \, d\Omega \\
c^{r,\ell}_i &= \int_\Omega \hat{g}^{\ell,r}_i \, d\Omega
\end{aligned}
$$

M and K are the mass and stiffness matrices for velocity, respectively, and $\hat{g}^{\ell,r}_i$ is defined by (4.15). Observe that $A^r_{ij} = \left(B^r_{ij}\right)^T = B^r_{ji}$. The integrands at element level are trivial to express, and follow the exact same pattern as those given for the heat equation.

As for the heat equation we can write the linear system (4.16)–(4.17) as a matrix equation, but it takes a block form

$$
\mathcal{A} \begin{bmatrix} \bar{v}^\ell \\ \bar{p}^\ell \end{bmatrix} = \begin{bmatrix} (M+K) & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \bar{v}^\ell \\ \bar{p}^\ell \end{bmatrix} = \begin{bmatrix} c^\ell \\ 0 \end{bmatrix} \tag{4.18}
$$

where $(M+K)$ is a $dn_v \times dn_v$ matrix and B is $n_p \times dn_v$, which makes $\mathcal{A}$ a $(dn_v + n_p) \times (dn_v + n_p)$ matrix. This is a saddle point problem.

Because this is a saddle point problem, we must ensure that the solution exists and is unique. This boils down to choosing the right combination of elements, which are those elements that satisfy the *Babuška–Brezzi* (BB) condition. Further details on the BB condition is beyond the scope of this thesis, but suffice it to say that of the two conditions contained in the BB condition, only the $\inf\sup$ criteria is critical for fluid mechanics [3]. As an alternative to enforcing the Babuška–Brezzi condition on the mixed elements, one can do different perturbations of (4.4), as mentioned earlier in this chapter, c.f. [12]. Stability issues pertaining to the spatial discretization is otherwise beyond the scope of this thesis, and it is merely mentioned here because it is a crucial part of successfully solving the equations. We refer to for instance [3] for discussion of stability of saddle point problems. For the Parareal algorithm we only need the spatial discretization to be stable for the chosen time discretization, and we will just assume that good choices are made for the spatial discretization at each time step, such as using valid mixed elements.

4.2 Iterative solver

Like we did for the heat equation we will consider the properties of $\mathcal{A}$ to decide on the type of iterative solver we should use to solve the linear block system in (4.18). Our starting point is once again the symmetric properties of $\mathcal{A}$, i.e we wish to determine whether $\mathcal{A} = \mathcal{A}^T$. We have

$$\mathcal{A}^T = \begin{bmatrix} (M+K)^T & \left(B^T\right)^T \\ B^T & 0 \end{bmatrix}^T = \begin{bmatrix} (M+K)^T & B^T \\ B & 0 \end{bmatrix},$$

and so $\mathcal{A}$ is clearly symmetric if $M+K$ is symmetric. $M+K$ is the matrix arising from the discretization of the heat equation, and in 3.2 on page 18 we showed that this matrix is both symmetric and positive–definite. Thus $\mathcal{A}$ must be symmetric. As we did for the heat equation, we turn our attention to the quadratic form of $\mathcal{A}$, which reads

$$\begin{aligned} x^T \mathcal{A} x &= \begin{bmatrix} \bar{v}^T & \bar{p}^T \end{bmatrix} \begin{bmatrix} (M+K) & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \bar{v} \\ \bar{p} \end{bmatrix} \\ &= \bar{v}^T (M+K)\bar{v} + \bar{p}^T B\bar{v} + \bar{v}^T B^T \bar{p}, \end{aligned} \tag{4.19}$$

where $\bar{v} = \{\bar{v}^r\} = \{v_j^r\}$ and $\bar{p} = \{p_j\}$ are the solution vectors from (4.14) at some arbitrary point in time. We have all ready determined that $(M+K)$ is positive–definite, and we turn our attention to $\bar{p}^T B\hat{v}$ and $\bar{v}^T B^T \bar{p}$. For clarity we consider the components $\bar{v}^r$ separately, and we omit the factor $-\Delta t\theta$. We see that

$$\bar{p}^T B^r \bar{v}^r = \int_\Omega \bar{p}^T L \frac{\partial N^T}{\partial x^r} \bar{v}^r d\Omega, \quad r = 1, \cdots, d,$$

where N and L are vector representations of the basis functions,

$$N = [N_1, \ldots, N_{n_v}]^T, \quad L = [L_1, \ldots, L_{n_p}]^T.$$

Writing out the vector products in the integral expression we have

$$\left(\bar{p}^T L\right)\left(\frac{\partial N^T}{\partial x^r}\bar{v}^r\right) = \left(\sum_{k=1}^{n_p} p_k L_k\right)\left(\sum_{l=1}^{n_v} \frac{\partial N_l}{\partial x^r} v_l^r\right). \tag{4.20}$$

Recall that we approximate each component of the unknown velocity as a vector, $\bar{v}^r = \{v_j^r\}$, such that we have

$$\begin{aligned} \bar{v} &= \sum_{r=1}^{d} \bar{v}^r = \sum_{r=1}^{d}\sum_{j=1}^{n_v} v_j^r N_j \\ \nabla\cdot\bar{v} &= \sum_{r=1}^{d} \frac{\partial \bar{v}^r}{\partial x^r} = \sum_{r=1}^{d}\sum_{j=1}^{n_v} v_j^r \frac{\partial N_j}{\partial x^r}, \end{aligned}$$

and we see from (4.20) that

$$\bar{p}^T B\bar{v} = \int_\Omega \left(\sum_{k=1}^{n_p} p_k L_k \right) \nabla \cdot \bar{v}\, d\Omega.$$

Following the same pattern for $\bar{v}^T B^T \bar{p}$ we end up with

$$\bar{v}^T B^T \bar{p} = \int_\Omega \bar{v}^T \frac{\partial N}{\partial x} L^T \bar{p}\, d\Omega = \int_\Omega \nabla \cdot \bar{v} \left(\sum_{k=1}^{n_p} L_k p_k \right) d\Omega.$$

From the expanded expressions of $\bar{p}^T B\hat{v}$ and $\bar{v}^T B^T \hat{p}$ we see that even though $(M+K)$ is positive–definite, we can make no such assumption for the quadratic form of $\mathcal{A}$ in (4.19) when $x \neq 0$. In fact, we cannot even determine whether

$$x^T \mathcal{A} x \geq 0 \quad \text{or} \quad x^T \mathcal{A} x \leq 0,$$

and so $\mathcal{A}$ is *symmetric* but *indefinite*. The iterative solver called *Minimal Residual Method* (MINRES) is very suitable for such problems, and in the Python–Diffpack implementation (4.18) is solved using MINRES at each time step. Actually, the Python implementation uses MINRES whereas the pure Diffpack implementation uses symmetric MINRES, which is not available in the `Pysparse` package.

4.2.1 Block preconditioning

The saddle point problem for the Stokes equations is computationally intensive to solve, and it is essential to precondition the system beforehand to speed up the convergence rate. This section covers both preconditioning of the steady state and the unsteady Stokes equations. Since the implementation of the unsteady Stokes equations is based on the steady state implementation done for [9] and [10], the time dependent preconditioner will be built on existing preconditioners. By presenting both types of preconditioners, we motivate the extensions done to the existing steady state code, as discussed in 4.4 on page 46. A more thorough discussion of block preconditioners for the steady Stokes problem can be found in [10] and preconditioning of block systems for unsteady Stokes are handled in [15]. This section is based on both of these papers, though only the results that are of immediate interest are presented. In other words, we only present the final shape of the preconditioner as this is what we must implement, and are not overly concerned with the detailed reasoning behind each preconditioner.

The discrete representations of both the steady and unsteady Stokes equations produce a block structure on the form

$$\mathcal{A} \begin{bmatrix} \bar{v} \\ \bar{p} \end{bmatrix} = \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \bar{v} \\ \bar{p} \end{bmatrix} = \begin{bmatrix} c \\ 0 \end{bmatrix}, \tag{4.21}$$

where

$$\begin{aligned}
A = K &= \int_\Omega \nabla N_j \cdot \nabla N_i \, d\Omega, && \{i,j\} = 1,\cdots,n_v \\
B &= \int_\Omega \nabla \cdot N_j \, L_i \, d\Omega, && \{i,j\} = 1,\cdots,n_v \\
c &= \int_\Omega f N_i \, d\Omega, && i = 1,\cdots,n_v
\end{aligned}$$

for the steady state problem, and

$$\begin{aligned}
A = M + K &= \int_\Omega N_j N_i \, d\Omega + \Delta t \theta \int_\Omega \nabla N_j \cdot \nabla N_i \, d\Omega, && \{i,j\} = 1,\cdots,n_v \\
B &= \Delta t \theta \int_\Omega \nabla \cdot N_j \, L_i \, d\Omega, && \{i,j\} = 1,\cdots,n_v \\
c &= \int_\Omega \hat{g}_i^\ell \, d\Omega, && i = 1,\cdots,n_v
\end{aligned}$$

for the time dependent equations. The components of $\hat{g}_i^\ell$ are given by (4.15). Depending on whether the appear in the steady or unsteady equations, the vectors $\overline{v}$ and $\overline{p}$ may or may not be time dependent. Spatially, they will still be determined by (4.14).

The steady state preconditioner

As remarked earlier, (4.21) is a saddle point problem. As stated in [10], the best such problem we can hope to solve in terms of efficiency is a saddle point problem with a block matrix, $\mathscr{A}$, on the form

$$\mathscr{A} = \begin{bmatrix} I & Q^T \\ Q & 0 \end{bmatrix}, \quad \text{where} \quad QQ^T \sim I.$$

A preconditioner that will cause $\mathcal{A}$ to be similar to $\mathscr{A}$ is on the form $\mathcal{B} = diag(C,D)$, such that we have

$$\mathcal{B}\mathcal{A} = \begin{bmatrix} C & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} CA & CB^T \\ DB & 0 \end{bmatrix},$$

where

$$\begin{aligned} CA &\sim I \\ DBCB^T &\sim I. \end{aligned}$$

Here, I is the identity matrix. For the current block system, this means choosing C as close to A^{-1} as possible; $C = \tilde{A}^{-1}$, which leads to $DBCB^T = DB\tilde{A}^{-1}B^T$. $B\tilde{A}^{-1}B^T$ is close to the pressure Schur complement of the system, $BA^{-1}B^T$, which is already well conditioned – and thus similar to I – due to the Babuška–Brezzi condition. Therefore we do not need D to further ensure that we have $QQ^T \sim I$, and we set D similar to I. As we can see in the discussion of time dependent preconditioner in the subsequent section, the identity matrix of the continuous system will be represented by the mass matrix in a finite element context. We therefore use the mass matrix associated with pressure, M_p, to set D. Thus we have, for the steady state problem, a preconditioner given by

$$\mathcal{B} = \begin{bmatrix} \tilde{K}_v^{-1} & 0 \\ 0 & \tilde{M}_p^{-1} \end{bmatrix},$$

where the subscripts v and p indicate whether the matrices use N or L as basis functions. This is the form of the preconditioner used by `StokesBlocks`, which was developed as part of [10], and used as a base class for the time dependent Stokes implementation. Note that K is the discrete representation of ∇^2, which is a so–called *elliptic operator*, and one can therefore use for example multigrid to efficiently find $\tilde{K}_v^{-1}$. An efficient way to find $\tilde{M}_p^{-1}$ is to lump the mass matrix[3] to create a diagonal matrix for which one can easily and efficiently find the exact inverse.

The time dependent preconditioner

As previously stated, the time dependent preconditioner is based on the uniform preconditioner presented in [15]. We will here establish the results as they relate to our needs, i.e. the resulting shape of the preconditioner. The goal of the aforementioned paper is to create a well–conditioned linear system, where the condition number of the preconditioned matrix is uniformly bounded by $\Delta t\theta$ and the spatial discretization parameter h. The derived preconditioner is motivated by one constructed for the continuous (in space) equation system. The reasoning behind the continuous preconditioner is beyond the scope of this text, and we will only present the conclusions.

From the temporal discretization of the stokes equations we have a semi–discrete system, given by (4.10)–(4.11), that is continuous in space. We can rewrite this as

$$\begin{aligned} \left(I - \varepsilon^2\nabla^2\right) v^\ell + \varepsilon^2 \nabla p^\ell = \varepsilon^2 f^\ell + v^{\ell-1} + \beta^2 f^{\ell-1} + \\ \beta^2 \left(\nabla^2 v^{\ell-1} - \nabla p^{\ell-1}\right) \end{aligned} \tag{4.22}$$

$$-\varepsilon^2 \nabla \cdot v^\ell = 0 \tag{4.23}$$

where I is the identity matrix (vector), $\varepsilon = \sqrt{\Delta t\theta}$ and $\beta = \sqrt{\Delta t(1-\theta)}$. Alternatively we can write this as

$$\mathcal{A}_\varepsilon \begin{bmatrix} v^\ell \\ p^\ell \end{bmatrix} = \begin{bmatrix} \left(I - \varepsilon^2\nabla^2\right) & \mathbf{grad} \\ \text{div} & 0 \end{bmatrix} \begin{bmatrix} v^\ell \\ p^\ell \end{bmatrix} = \begin{bmatrix} g^\ell \\ 0 \end{bmatrix},$$

[3]Two possible lumping strategies are presented in chapter 2.4.2 in [8]

where g^ℓ is the right hand side of (4.22). From the discretization, it should be clear that the discrete version of $\mathcal{A}_\varepsilon$, $\mathcal{A}_{\varepsilon h}$, is given by (4.18), such that

$$\mathcal{A}_{\varepsilon h} = \begin{bmatrix} M_v + \varepsilon^2 K_v & \varepsilon^2 B^T \\ \varepsilon^2 B & 0 \end{bmatrix}.$$

Since $(I - \varepsilon^2\nabla^2)$ will discretize as $(M_v + \varepsilon_v^K)$, we see how the mass matrix is analogous to the identity matrix in a finite element context, which we used for the steady state preconditioner.

As is stated in [15], we will assume that $\varepsilon \in (0,1]$. This is not an unreasonable assumption. From the definition of the θ–rule we have that $\theta \in [0,1]$ and $\Delta t \neq 0$. In addition, Δt would always be positive, and for most practical uses one would choose $\Delta t < 1$. Further, it is pointed out in [15] that ε influences the properties of the equation system (4.22)–(4.23), such that when ε goes to zero we have a system that approaches the mixed Poisson problem, whereas when ε increases we move toward a steady–state Stokes problem. We will not explore these equation systems here, except to note that they do require different preconditioners, since they are two distinct problems. This implies that the preconditioner presented for the steady–state equations will not unconditionally be a good choice of preconditioner for the unsteady Stokes equations. A uniform preconditioner is therefore found for the problem that is independent of both ε and the spatial discretization step h. The continuous preconditioner is presented as

$$\mathcal{B}_\varepsilon = \begin{bmatrix} \left(I - \varepsilon^2\nabla^2\right)^{-1} & 0 \\ 0 & \varepsilon^2 I^{-1} + \left(\varepsilon^2\nabla^2\right)^{-1} \end{bmatrix},$$

which has its discrete equivalent given by

$$\mathcal{B}_{\varepsilon h} = \begin{bmatrix} (M_v + \varepsilon^2 K_v)^{-1} & 0 \\ 0 & \varepsilon^2 M_p^{-1} + \left(\varepsilon^2 K_p\right)^{-1} \end{bmatrix}.$$

We see that the preconditioner for the time dependent equations is similar to the steady state preconditioner, and only the expression for D has changed, by introducing the stiffness matrix for pressure as a result of the ∇^2 operator in the continuous preconditioner. In terms of implementation, we then only need to do slight modifications for the pressure component in the existing steady–state preconditioner, whereas the velocity component may remain unchanged. As for the steady–state preconditioner, each block in $\mathcal{B}_\varepsilon$ is is composed of an elliptic operator, such that one can use for example multigrid to find an approximate to the inverse. This is done for our Python implementation presented in chapter 4.4 on the following page, where the algebraic multigrid module `ML` was used to find approximates to $(M_v + \varepsilon^2 K_v)^{-1}$, $\varepsilon^2 M_p^{-1}$ and $\left(\varepsilon^2 K_p\right)^{-1}$. The Diffpack version uses the exact inverse of the lumped mass matrix and one SSOR iteration to approximate K_p

4.3 Stokes equations and Parareal algorithm

In chapter 4.1.1 on page 36 we argued for using the θ–rule to discretize the time derivate. The gist of the argument was that it gives one the opportunity to study the stability range for the Parareal algorithm when used on the Stokes equations compared to using it on the heat equation. We expect the Parareal algorithm to be able to handle Stokes, as it is a parabolic PDE [4, 1.7.4]. The interesting question is whether aspects of the Stokes equations that are not present in the heat equation will influence the range of θ–values for which Parareal is stable. For the heat equation, the analysis done in [2] and [20], holds, and the algorithm is stable in the range $\theta \in [2/3, 1]$.

When semi–discretized in space, the Stokes equations become a set of *differential algebraic equations* (DAE), as opposed to semi–discretizing the heat equation, which only transforms into a common ODE. Writing out the semi–discretized system of (4.3)–(4.4), we have

$$\begin{aligned}\frac{\partial v(t)}{\partial t} &= K\overline{v}(t) - B^T\,\overline{p}(t) + c(t)\\ 0 &= B\overline{v},\end{aligned}$$

where the vector notation from (4.14) is used on v and p, and

$$\begin{aligned}K &= \int_\Omega \nabla N_j \cdot \nabla N_i \, d\Omega\\ B &= \int_\Omega \nabla \cdot N_j L_i \, d\Omega\\ c &= f N_i.\end{aligned}$$

We see that the system is a DAE, since the first equation is a differential equation, whereas the second is a pure algebraic constraint. DAE may influence the ODE stability assumptions the Parareal analysis is based on. We postpone further discussion of DAE, as it will only be interesting to explore the DAE theory if convergence and stability of the algorithm deviates strongly from the blue–print set by the heat equation.

4.4 Implementation

Based on the discretizations done the previous chapter, the Stokes equations are now ready to be implemented. As for the heat equation, a mix of Python and Diffpack was used to create a class hierarchy that can be processed by the Parareal algorithm. We will commence by defining our model problem, before moving on to outlining the class hierarchies developed for the Stokes problem.

4.4.1 Defining the model problem

For the Stokes equations in this thesis we constructed an initial–boundary value problem for which the analytical solution is known. The model problem will likely be removed form any real–life scenario, but as we only need to have some arbitrary Stokes IVBP to solve with the Parareal algorithm, it is more practical to have an analytical solution. Access to an analytical solution during development is always useful, as you always have a reference to the solution your simulator program *should* produce. This gives you the opportunity to verify your code as you develop it. As a starting point we therefore need to construct an initial–boundary value problem for which the analytical solution is known.

Constructing an IBVP with known analytical solution

We define our domain identically to the one used for the heat equation, and set

$$\Omega \in [0,1][0,1] \text{ for } x \in \mathbb{R}^d,\ d = 2.$$

To construct an analytical solution to a IVBP, we simply decide on the functions $v(x,t)$ and $p(x,t)$ that will construct our analytic solution, and then base our initial condition, boundary values and source term on the chosen functions. Naturally, it is important that one chooses the functions such that they actually satisfy (4.3)–(4.4). For this thesis we solved a 2D problem, and used

$$\begin{aligned} v_x(x,t) &= v_x(x,y,t) &&= -\cos(xyt)x \\ v_y(x,t) &= v_y(x,y,t) &&= \cos(xyt)y \\ p(x,t) &= p(x,y,t) &&= \sin(xy) - at, \end{aligned}$$

where a is a constant. For our implementation, a is hard coded to $a = 0.9460830704$. Note that $v(x,t)$ also satisfy the continuity constraint (4.4), and not only (4.3). The initial condition for our system will then be given by

$$\begin{aligned} v_0 &= \begin{cases} v_1(x,y,0) = x \\ v_2(x,y,0) = y \end{cases}, \quad (x,y) \in [0,1], \\ p_0 &= p(x,y,0) = \sin(xy), \quad (x,y) \in [0,1], \end{aligned}$$

and the essential boundary condition for v on component form become

$$\begin{aligned} v(0,y,t) &= \begin{cases} v_x = 0 \\ v_y = y \end{cases}, \quad v(1,y,t) = \begin{cases} v_x = -\cos(yt) \\ v_y = \cos(yt)y \end{cases}, \quad y \in [0,1] \\ v(x,0,t) &= \begin{cases} v_x = -x \\ v_y = 0 \end{cases}, \quad v(x,1,t) = \begin{cases} v_x = -\cos(xt)x \\ v_y = \cos(xt) \end{cases}, \quad x \in [0,1]. \end{aligned}$$

By plugging the analytical solutions into (4.3)–(4.4), we obtain an expression for the source term f. A simple and quick way of doing this is using *Maple*, which will produce the symbolic solution rather than the numerical. You can also set it to produce the equivalent C code for your mathematical expressions, which is highly practical. By using Maple, we end up with a source term f where each component is given by

$$\begin{aligned} f_x &= -\cos(xyt)y^2t^2x - 2\sin(xyt)yt - \cos(xyt)x^3t + \cos(xy)y + \sin(xyt)x^2y \\ f_y &= \cos(xyt)y^3t + \cos(xyt)x^2t^2y + 2\sin(xyt)xt + \cos(xy)x - \sin(xyt)xy^2. \end{aligned}$$

We now have a complete initial–boundary value problem which we can implement using the discretizations from the previous chapters and Diffpack. On top of the Diffpack code we may place Python classes, that utilize our Diffpack implementation to solve the Stokes equations using Parareal.

4.4.2 The Stokes class hierarchy

The Stokes class hierarchy is split into two logical blocks: the class hierarchy in Diffpack and the class hierarchy in Python. We will in the following aim to give an overview of the Diffpack and Python classes. It is not intended as a full documentation of the implementation, but only as a complement to the source code and chapter 3.4 on page 23, which covers the implementation of the heat equation. As the Stokes classes follow the same structure as the heat equation classes, though the class hierarchy itself is larger. 3.4 together with the following section, should give a good, basic understanding of the Stokes implementation.

The Diffpack hierarchy

The core implementation of the Stokes equations, where mixed finite elements are used to cerate the linear system, is done through use of the Diffpack library. The Diffpack simulators are also designed to solve the linear system if requested, although the idea is that we will use Python to solve the linear system produced by Diffpack. The Diffpack implementation of the unsteady Stokes equations is based on the class hierarchy written for the Mixed Finite Elements paper [9]. Originally, this class hierarchy consisted of a Stokes solver that used a single merged linear system to solve the equations, and then a block solver was built on this general solver. The class hierarchy has been further expanded here by adding time dependency. The full Diffpack class hierarchy is presented in Figure 4.1. All classes marked `Time` were added to the hierarchy for this thesis, though some minor functionality in the base classes were adjusted in order to make extension to time dependent code cleaner and simpler.

The hierarchy divides into two groups; the functor classes that are elements used to fill the block structure and finding the for the implement the analytical solution, and the classes in the governing solver hierarchy. Focusing on the time–dependent elements of the hierarchy, `*AnalTime` classes are the Diffpack functor classes implementing the analytical

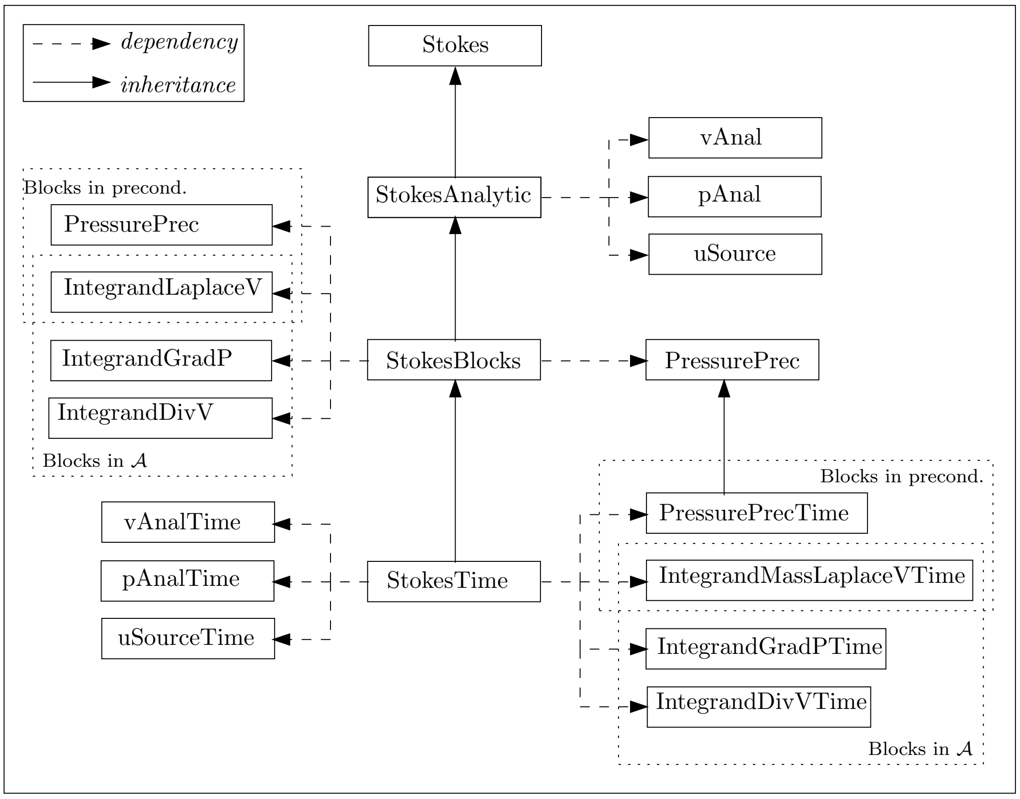

Figure 4.1: The Diffpack class hierarchy for steady and unsteady Stokes.

solution and source terms defined above. A second group of functor classes are those used to construct the separate blocks in the linear system matrix $\mathcal{A}$; `IntegrandMassLaplaceVTime`, `IntegrandGradPTime` and `IntegrandDivVTime`. As their names indicate, they respectively implement the blocks $(M+K)$, B^T and B. `IntegrandMassLaplaceVTime` is also used to construct the velocity component of the block preconditioner, $(M_v+\varepsilon^2 K_v)^{-1}$. `PressurePrecTime` creates the pressure block of the preconditioner, and must necessarily solve the components $\varepsilon^2 M_p^{-1}$ and $\left(\varepsilon^2 K_p\right)^{-1}$. To do so it uses another set of functor classes which were not added to the schematic class diagram, namely `IntegrandMassP` and `IntegrandLaplaceP`, respectively.

`StokesTime` expands `Stokes` to support time integration, but uses the functionality in its base class to build and manage the block linear system. There is really very little extra for the `StokesTime` class itself to do in terms of key functionality, except manage the time derivative. It must necessarily ensure that the time dependent version of the functor classes is used, but it is the base class and functor classes that handle the bulk of the logic. For instance will `IntegrandMassLaplaceVTime` not only build $M+K$, but it will also construct the dynamic right–hand side of the linear system. The integrands function that does this is shown in Listing 4.1 on the next page, although the actual construction of the right–hand side was omitted as it is fairly extensive. The elements of the integrands in (4.16) should be relatively easy to perceive.

Listing 4.1: The `IntegrandMassLaplaceVTime::integrandsMx()` function with θ-rule discretization

```
void IntegrandMassLaplaceVTime::integrandsMx( ElmMatVec& elmat,
                                        const MxFiniteElement& mfe ){
    const real detJxW = mfe.detJxW();
    const int nsd     = mfe.getNoSpaceDim();
    const int nvxbf   = mfe(1).getNoBasisFunc();

    real dtTheta   = data->tip->Delta() * data->theta;
    real stiffness;
    real mass;

    for( int d = 1; d <= nsd; d++ ){
        for( int i = 1; i <= nvxbf; i++ ){
            int ig = (d-1)*nvxbf+i;

            //fill elmat.A
            for( int j = 1; j <= nvxbf; j++ ){
                int jg = (d-1)*nvxbf+j;
                stiffness = 0;
                mass = mfe(d).N(i)*mfe(d).N(j);
                for( int k = 1; k <= nsd; k++ )
                    stiffness += mfe(d).dN(i,k) * mfe(d).dN(j,k);
                elmat.A(ig,jg) += (mass + stiffness*dtTheta)*detJxW;
            }
        }
    }
    fillRHS( elmat, mfe );
}
```

The Python hierarchy

A Python class of Stokes solvers hierarchy that supports the Parareal Python implementation interface requirements was constructed to utilize the Diffpack classes. For the Python implementation, there are are two principal hierarchies. One that inherits directly from the Diffpack class and only adds the minimal required Parareal interface – `StokesPureDp`, and a slightly more extensive hierarchy based on `StokesDpProvider` that use Diffpack primarily as a linear system generator. See Figure 4.2. `StokesDpProvider` handles all logic in obtaining and updating the linear system provided by Diffpack, whereas its subclasses implement different strategies for solving the linear system. Specifically, `StokesDpItSolver` expects the underlying Diffpack class to provide a means to solve the linear system, whereas `StokesPysparseItSolver` uses the iterative solvers in the `Pysparse` package. As was mentioned at the conclusion of chapter 4.2 on page 41, the Diffpack iterative solver will be Symmetric MINRES, whereas the `Pysparse` version must use the `itsolvers` module which only has support for MINRES.

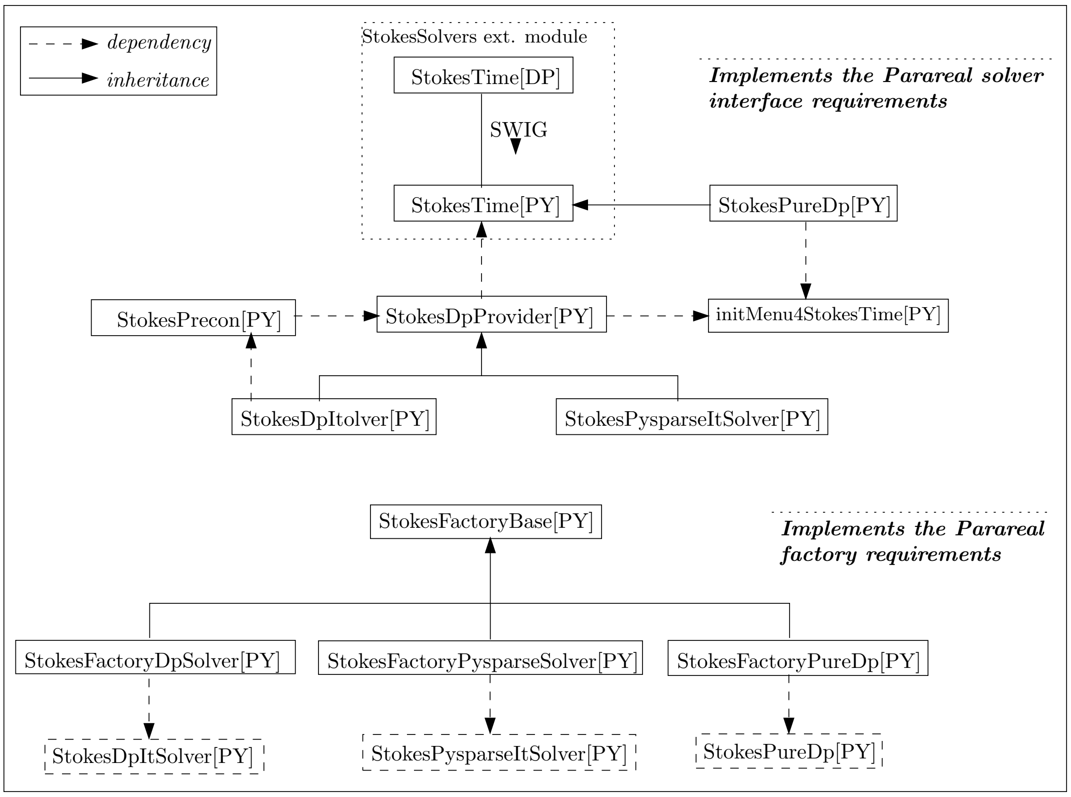
dependency
inheritance
StokesSolvers ext. module
StokesTime[DP]
SWIG
StokesTime[PY]
Implements the Parareal solver interface requirements
StokesPureDp[PY]
StokesPrecon[PY]
StokesDpProvider[PY]
initMenu4StokesTime[PY]
StokesDpItolver[PY]
StokesPysparseItSolver[PY]
StokesFactoryBase[PY]
Implements the Parareal factory requirements
StokesFactoryDpSolver[PY]
StokesFactoryPysparseSolver[PY]
StokesFactoryPureDp[PY]
StokesDpItSolver[PY]
StokesPysparseItSolver[PY]
StokesPureDp[PY]

Figure 4.2: The Python class hierarchy for unsteady Stokes ([DP] and [PY] refer to implementation in Diffpack or Python, respectively).

The class interfaces and structure follow the same pattern as was done for the heat implementation, and we will therefore not focus as much on their inner structure as we did in the heat implementation chapter. Our aim here is to highlight the different aspects of the implementation to give the reader an overview of the structure. For further details we refer to the source code. The similarities to the heat implementation are particularly true for the factory classes, and we will not discuss them further.

Preconditioning

When Diffpack is used to solve the linear equation system, it will also handle its own preconditioner (as defined in Figure 4.1) and so we only need to consider the preconditioner for `StokesPysparseItSolver`. When Diffpack is used as a linear system provider, it is clear that this must also include providing the preconditioner for the system, as Diffpack has all the tools for creating a matrix from a integral expression. We therefore have `StokesPrecon`, which is intended to work as the connector between a Diffpack preconditioner and a preconditioner for the `itsolvers` module. It will do this simply by defining the preconditioner interface the `itsolvers` functions expect, and act as preconditioner. It will obtain the matrices on which the preconditioner is based on from Diffpack, and $(M_v + \varepsilon^2 K_v)$, $\varepsilon^2 M_p$ and $\left(\varepsilon^2 K_p\right)$ will be filtered into `Pysparse` matrices. `StokesPrecon` must define a function, **def** `precon( self , x, y ):`, which will calculate the effect of applying the preconditioner to `x` and store the result in `y`, in order to become a preconditioner. It must therefore calculate

$$
\begin{aligned}
y_v &= (M_v + \varepsilon^2 K_v)^{-1} x_v \\
y_p &= \varepsilon^2 M_p^{-1} x_p + \left(\varepsilon^2 K_p\right)^{-1} x_p,
\end{aligned}
$$

where the subscripts denote the v and p components of the solution. As has been previously mentioned, these matrices represent elliptic operators, and we can use *algebraic multigrid* through the `ml` module to solve the linear systems

$$
\begin{aligned}
(M_v + \varepsilon^2 K_v) y_v &= x_v \\
\left.\begin{aligned} \varepsilon^2 M_p y_{p1} &= x_p \\ \varepsilon^2 K_p y_{p2} &= x_p \end{aligned}\right\} & \; y_{p1} + y_{p2} = y_p,
\end{aligned}
$$

instead of finding the their inverse. We will merely employ multigrid as a black box solver that functions well on elliptic operators.

The block structure

The block structure in the Stokes classes are not supported by Python. Specifically, if the `Pysparse` and `ML` modules are to be used one must remove the block structure, as they expect to work with `ll_mat` objects and `Numeric` arrays. These do not directly support the block structure used in Diffpack. Therefore, the `StokesTime` object has been expanded with functionality to fold out the block structure into plain vectors and matrices, and to reduce plain vectors to block vectors, all in order for it to be a linear system provider for Python. This machinery creates quite an extension to the class interface that do not directly relate to solving the Stokes equations. The `StokesDpProvider` uses these classes to filter solutions between Python and Diffpack for each time step, such that both representations have valid values to work with when they either update the time dependent right–hand side of the linear system (Diffpack) or when they solve the linear system (Python). Incidentally this means that for the `StokesDpItSolver` class for solving the linear system, one creates a bit of unnecessary overhead as you filter vectors up and down that will only ever be used in Diffpack. It was nevertheless done this way to show that you can easily make Python module based and create relative easy support for switching the linear solver. Since we only had access to one truly external linear solver (`Pysparse`), we "created" one by having a base class trigger the linear solver stored in StokesTime.

Because there was substantial extra logic connected to turning `StokesTime` into a linear system provider, all the extra functionality needed was added directly in the class source code and not through small extension functions in the SWIG interface file like it was done for the heat implementation.

This concludes our discussion of the Stokes problem. In the next section we will explore the Parareal algorithm, and prepare the implementation that will be used to test the algorithm on the Stokes problem.

Chapter 5

The Parareal Algorithm

Parallel computation is a strategy to speed up the computational process in order to for example solve the problem over a larger time–space domain using the less computation time than a sequential process, or to use a finer discretization (and thereby reach a better estimate) without drastic increase of computation time. And it is of course a means to solve large systems within a feasible time period, such as weather–forecasting.

To parallelize the finite element method, for example, one has the *domain decomposition method* to create relatively decoupled subproblems of the finite element mesh that become more amenable to parallel computation. There are also methods to parallelize the iterative solvers which must run to solve the linear system *at each time step*. These methods focus on speeding up the spatial aspect of the problem discretization, and leaves a largely sequential discretization. This is clear in domain–decomposition methods whose sole focus is to decompose the spatial domain and make the problem more parallelizable. In the second case, where parallel computation is introduced with the linear solver and not at domain level, the statement still has merit. Even though time and space discretization is closely linked in the final discrete linear system, a the larger part of the system is due to spatial discretization (see the discretizations of heat and Stokes). Since the time stepping is still a serial procedure, one can view the parallel linear solver as more of a *spatial* parallel process[1].

The *Parareal algorithm* is a tool to overcome the sequential nature of time–stepping. This chapter introduces the algorithm which solve differential equations parallel in time, and also gives an overview of the accompanying Python implementation. Since the thesis focuses on solving the heat and Stokes equations, the algorithm is discussed in terms of solving partial differential equations. The aim is to give an easy description of how the algorithm operates. Chapter 5.1 on the next page is mainly based on the description given in [20].

[1]Observations based on Xing Cai's course INF5640 autumn 2004 at IFI

5.1 The Parareal Algorithm

The Parareal algorithm operates on time dependent (unsteady) PDEs that must be integrated over the time domain $t \in [t_0, T]$ in order to find a solution. Time is naturally sequential, which is likely to account for the scarcity of parallel-in-time schemes compared to the mature field of spatial–oriented parallel calculation. Regardless of the reason, the result is that the integration process of time tends to be time consuming in the overall computational scheme, and it is a bottleneck that can be hard to avoid. The essence of the Parareal algorithm is to try to remove the sequential obstacle, and construct a new problem with increased parallelism. Once increased parallelism is achieved, the next step is to utilize it and create a parallel-in-time scheme for solving the PDE in question.

Consider an initial–boundary value problem where we seek the unknown function $u(x,t)$ on the sequential time line. If we could divide it into subdomains in time, while somehow have an initial value for each subdomain, we would have a set of subproblems that mirror the structure of the global problem – and a problem that is very amenable for parallel computation.

Ideally, we would like to have a situation where a set of solutions, $\{u(x,t_i)\}$, of the PDE is known at $i = 1 \cdots, N$ points in time prior to calculation. This would allow us to create N independent initial-boundary value *sub*problems – see Figure 5.1 – instead of the typical IBVP defined over the time domain $t \in [t_0, T]$, with a single initial condition at $t = t_0$. This is for example the structure we have seen for the heat and Stokes equations. Because we can handle these independent subproblems in parallel, we could efficiently solve the PDE over the entire time domain using a much finer time step compared to what would be feasible in a single process situation, where such fine grained computation is likely to be too time consuming. This is particularly the case for real–time simulations, which was the type of simulations the Parareal algorithm was originally intended for – thus the name Para*real*. If you are not concerned about real–time, the increased efficiency gives the possibility to widen the time–domain without increasing the computation time.

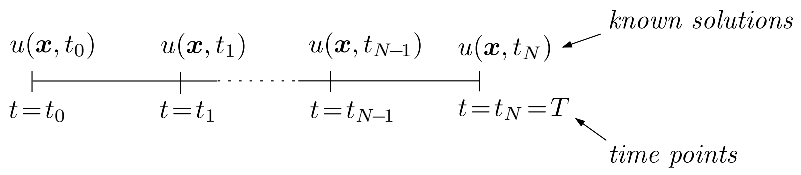

Figure 5.1: Ideal situation for increased parallelism

Decomposing the time line

To achieve a scenario approaching Figure 5.1 we begin by decomposing the time interval $t \in [t_0, T]$ into

$$t_0 = T_0 < T_1 < \cdots < T_i = i\Delta T < T_{i+1} < T_N = T,$$

where $\Delta T = T/N$ for some integer N. This gives rise to a natural division into N subdomains in time, $S_i = [T_{i-1}, T_i],\ i = 1, \ldots, N$, as shown in Figure 5.2. In a parallel setting, N will typically be the number of processes running in parallel, as it is natural to assign one subdomain per process.

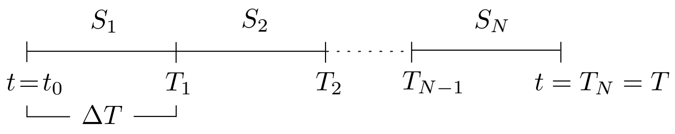

Figure 5.2: Time decomposition into S_N subdomains

As mentioned above, we would like the solution of the PDE to somehow be known at the start of each subdomain, i.e to have $u(x, T_{i-1}),\ i = 1, \ldots, N$ in advance. This is not the case for initial–boundary value problems, and therefore the Parareal algorithm tries to give a decent *prediction* of what the solution will be at these points in time. To achieve this we introduce a *coarse solver*, $\mathcal{G}$, suited to solve the underlying PDE with a coarse time discretization parameter, i.e suited to use ΔT as its time step. Using $\mathcal{G}$ we can find a solution for each time step in the decomposed time domain in Figure 5.2. We denote these solutions as $\lambda_i, i = 0, \ldots, N-1$, where λ_0 represent the initial condition of the original initial-boundary value problem. We can then use λ_0 to λ_{N-1} can be used as initial values for the subdomains S_1 to S_N.

A first parallel scheme

Finding the λ_i values is, as illustrated in Figure 5.3, just a standard serial pass over the time domain, using $\mathcal{G}$ as the solver and previously computed solutions as start value. As will become evident from the Parareal algorithm, this pass will be done several times. Since the overall goal of the Parareal algorithm is to save computation time, it is important that the coarse solver is *efficient*, to ensure that such a pass is relatively cheap to compute. Naturally,

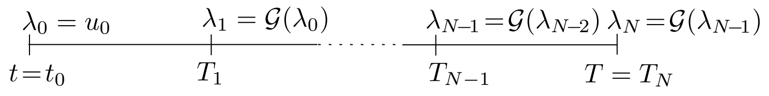
$\lambda_0 = u_0$
$\lambda_1 = \mathcal{G}(\lambda_0)$
$\lambda_{N-1} = \mathcal{G}(\lambda_{N-2})$
$\lambda_N = \mathcal{G}(\lambda_{N-1})$
$t = t_0$
T_1
T_{N-1}
$T = T_N$

Figure 5.3: Initializing the λ values

the set $\{\lambda_i\}$ also represent the numerical solution to the system at given time intervals, and in the algorithm we will use them both as initial values and solution objects.

By use of $\mathcal{G}$ we now have a set of problems approximating the scenario in Figure 5.1, and we can efficiently find a more accurate set of λ solutions by running *fine stepping solvers*, $\mathcal{F}(\lambda_{i-1})$, in parallel over each subdomain. Each fine solver will produce a solution for the end of its time domain, λ_i, which should be more accurate, and we can them to replace the previous λ-values. We will denote the original set of solution values as $\{\lambda_i^0\}$ and the new, improved set as $\{\lambda_i^1\}$.

Improving the scheme with Parareal

Since we only made a rough estimate of the values used as initial conditions for the fine solvers, we must expect the solution represented by $\{\lambda_i^1\}$ to have an inherent error. This will make it different to the solution we would have found if we had run a single fine solver over the global domain $[t_0, T_i]$[2]. In this light our scheme is inherently flawed – when introducing parallel computation we are obviously not interested in producing a result that is clearly inferior to the serial solution.

The Parareal algorithm tries to compensate for this by iteratively finding new coarse solutions using the most recent λ–values (in a similar fashion to Figure 5.3), and then applying a corrector to adjust the estimates. The corrector is based on the difference between the coarse and fine solutions from the previous iterations. This is is not a wholly unreasonable approach. The corrective term should give an indication of how far removed the previous coarse estimate was to the exact (serial) solution. The reasoning is then that this is likely to be similar to how far removed the current coarse estimate is to the corresponding fine estimate. The idea is summed up in (5.1).

$$\begin{aligned}
\lambda_0^k &= u_0 \\
\lambda_i^0 &= \mathcal{G}\left(\lambda_{i-1}^0\right), && i = 1,\dots,N \\
\lambda_i^k &= \mathcal{G}\left(\lambda_{i-1}^k\right) + \underbrace{\mathcal{F}\left(\lambda_{i-1}^{k-1}\right) - \mathcal{G}\left(\lambda_{i-1}^{k-1}\right)}_{\delta\mathcal{G}\left(\lambda_{i-1}^{k-1}\right)\ \text{(corrector)}}, && i = 1,\dots,N,\ 0 < k,
\end{aligned} \tag{5.1}$$

where k is the iteration number. The equivalent Parareal pseudo code is presented in Algorithm 1 on the following page, which might give a more intuitive understanding of the algorithm. One continues to iterate until some convergence criteria has been reached. The common practice is to choose some reasonable way to compare $\{\lambda_i^{k-1}\}$ and $\{\lambda_i^k\}$, say by computing the difference norm. Further discussion of handling the convergence criteria follows in section 5.1.3.

[2]In a Parareal setting this is referred to as the exact solution. It is the best possible solution a parallel computation can hope to find – it will never be more accurate than its equivalent serial estimate

Algorithm 1 Parareal Algorithm

```
$\lambda_0^0 = u_0$                    //*Set $\lambda_0$ to the initial condition*
for $i = 1$ to $N$ do
  $\lambda_i^0 = \mathcal{G}\left(\lambda_{i-1}^0\right)$    //*initialize using coarse time step solution*
end for
Solve subdomains $\mathcal{F}\left(\lambda_{i-1}^0\right)$ in parallel on $i = 1 \ldots N$ processes

k=1
while true do
  for $i = 1$ to $N$ do
    solve $\mathcal{G}\left(\lambda_i^{k+1}\right)$
    $\lambda_i^k = \mathcal{G}\left(\lambda_{i-1}^k\right) + \mathcal{F}\left(\lambda_{i-1}^{k-1}\right) - \mathcal{G}\left(\lambda_{i-1}^{k-1}\right)$
  end for
  if convergence then
    break loop
  end if
  Solve subdomains $\mathcal{F}_i\left(\lambda_{i-1}^k\right)$ in parallel on $i = 1 \ldots N$ processes
  k = k+1
end while
```

As $\mathcal{F}$ always operate on the previously known values λ_{i-1}^{k-1}, the fine solvers can be executed in parallel. $\mathcal{G}\left(\lambda_{i-1}^k\right)$ is, on the other hand, strictly sequential, since we constantly use λ–values from the current iteration. Thus we see that $\mathcal{G}$ must be efficient, because we need to run it once for each calculation.

5.1.1 Noteworthy properties of the algorithm

There are some properties one should be aware of when working with the Parareal algorithm. A more complete list can be found in [20].

Strictly parallel. Our first, simple observation is that the algorithm is strictly parallel, meaning that if you run a serial computation (i.e $N = 1$) it will involve more work, and thus run slower, than if you had done one, fine serial pass over the entire time domain.

The number of processes. The number of processes, N, must be chosen such that Δt is a factor of ΔT.

Exact solution. The exact solution is the solution found by doing a serial pass over the entire time domain, with a solver using the same temporal and spatial discretization parameters as $\mathcal{F}$. Therefore,

$$\begin{aligned} \lambda_0 &= u_0 \\ \lambda_i &= \mathcal{F}(\lambda_{i-1}),\ i = 1, \ldots, N, \end{aligned}$$

is identical to the solution we would find if we did a serial pass over the entire time domain using the fine solver. Note that a lack of iteration counter k indicates the exact solution. It is a fact that all values in the time domain $[t_0, k\Delta T]$ will be exact, and all λ_i^k for $i \leq k$ will be equal to the serial solution. This implies that

$$\begin{aligned} \lambda_0^k &= u_0 \\ \lambda_i^k &= \mathcal{G}\left(\lambda_{i-1}^k\right) + \mathcal{F}\left(\lambda_{i-1}^{k-1}\right) - \mathcal{G}\left(\lambda_{i-1}^{k-1}\right), && i = 1k+1, \ldots, N,\ 0 < k \\ \lambda_i^k &\equiv \lambda_i = \mathcal{F}(\lambda_{i-1}), && i = 1, \ldots, k \end{aligned}$$

Therefore, the algorithm will converge toward the exact solution, because the direct influence of λ_0 will propagate further up through the λ–values with each iteration. At $k = N$ we have

$$\begin{aligned} \lambda_0 &= u_0 \\ \lambda_i^N &= \mathcal{G}\left(\lambda_{i-1}^N\right) + \mathcal{F}\left(\lambda_{i-1}^{N-1}\right) - \mathcal{G}\left(\lambda_{i-1}^{N-1}\right),\ i = k+1, \ldots, N \\ \lambda_i^N &\equiv \mathcal{F}(\lambda_{i-1}),\ i = 1, \ldots, k. \end{aligned}$$

This implies that the solution will never be more accurate than the serial computation, even though the *algorithm* itself converges toward machine accuracy: the change between λ^k and λ^{k-1} is as small as can possibly be measured with machine accuracy.

Maximum number of iterations and speedup. In th exact serial solution, one expects a certain error compared to the analytical solution. When the same order of error is found in the parallel computation as is expected in the exact solution, further predictor-corrector passes will be superfluous. When determining convergence one should try to take this into account. It also means that there exists some maximum number of iterations before the algorithm starts doing unnecessary iterations, which is naturally unwanted when you are trying to achieve as much increase of computation time as possible.

If we let $t_\mathcal{G}$ denote the cost of computing $\mathcal{G}(\lambda_{i-1})$ for $i = 1, \ldots, N$, and $t_\mathcal{F}$ the time to compute $\mathcal{F}(\lambda_{i-1})$ for one subdomain, we can set bounds on the maximum number of iterations we can possibly have before the algorithm converges and still achieve speedup compared to a serial computation. Each pass through the main loop of the algorithm has a cost of $t_\mathcal{G} + t_\mathcal{F}$. We do not need to calculate $\mathcal{G}(\lambda^{k-1})$ or $\mathcal{F}(\lambda^{k-1})$ since they were found in the previous iteration. The initialization phase of the algorithm has the same cost as one iteration. To achieve a speedup over serial computation we must therefore have

$$(k+1)(t_\mathcal{G} + t_\mathcal{F}) < Nt_\mathcal{F},$$

which indicates that convergence must be reached for

$$k < \frac{Nt_{\mathcal{F}}}{t_{\mathcal{G}} + t_{\mathcal{F}}} - 1, \quad 0 < k$$

in order to achieve speedup. If we assume that $t_{\mathcal{G}} = t_{\mathcal{F}}$ we must reach convergence while $k < \frac{N}{2} - 1$. Looking at the speedup possibilities we see that the algorithm is obviously not useful for schemes with less than 3 processes running in parallel. $N = 3$ would require immediate convergence, whereas 10 processes running in parallel would, for example, require convergence while $k \leq 4$ to achieve speedup.

Flexible choice of $\mathcal{G}$ and $\mathcal{F}$. Note that the algorithm does not require $\mathcal{G}$ and $\mathcal{F}$ to solve the same equation, nor use the same spatial grid size or discretization scheme, as long as the sum of the returned solutions can be done in a consistent way. This means that you are free to let $\mathcal{G}$ solve some perturbed equation that could, for example, remove highly oscillating terms that are under sampled because of the coarse time step, or use a coarser spatial discretization to better match the coarse temporal discretization. This last feature can be useful as there is little point in doing a very fine discretization in space if the total numerical error dominated by the error from your coarse time discretization.

5.1.2 Strategies for determining convergence

The Parareal algorithm does not explicitly dictate how you should handle convergence. Two immediate strategies would be to either look for convergence by comparing a selection of entries in λ^k and λ^{k-1}, or you could use some suitable norm of the difference (i.e. a difference norm) between λ^k and λ^{k-1}. The latter would be more time consuming, but can potentially give a better average estimate of the global convergence of the λ values.

In the former alternative one would do a point wise check for convergence between λ_i^k and λ_i^{k-1}, but one must then make sure that $k < i$. Remember that the values λ_j^k, $1 \leq j < k$ are as accurate as they can be, and comparing old and new values at index j is meaningless as the conclusion will always be that convergence has been reached. Based on the shrinking pool of λ–values to use for comparison as the number of iterations increase, one might decide to compare the values at $i = N$ only, but by choosing a single point of comparison you could, however, risk that under the right circumstances the estimate λ_i^k mimics the exact serial solution and thus cause premature convergence.

For this thesis we opted for using the norm between the old and new λ–values in a global space–time norm,

$$e = \sqrt{\sum_{i=k}^{N} \|\lambda_i^k - \lambda_i^{k-1}\|^2 \Delta t},$$

as a measurement of the error. The algorithm is then terminated when $e < \varepsilon$. We have not found any literature that verifies whether this is a good reflection of the error in the current solution set, and we will therefore do numerical tests of the quality of this stop criteria in chapter 6 on page 70.

Once convergence has been determined, a returning issue is the final step of the algorithm. What is the last step? Should one say that the λ^k values used to determine convergence is the solution, or should one say that they are the best possible set of initial conditions and use λ^k to do a last "sweep" with the fine solver? Algorithm 1 on page 58 uses the former option. As long as you do not run the maximum number of iterations, $k = N$, λ_{i-1}^k will always be more accurate than λ_i^k, due to the way influence by λ_0 moves over the set of λ values. By doing a last fine sweep we would push this "accuracy boundary" one step further to the right – at the cost of increased an computation time $t_{\mathcal{F}}$. The alternative scenario for wrapping up the algorithm could be done as shown in Algorithm 2. Which strategy one should choose really depends on the stop criteria ε – if the former termination scheme is chosen, a more severe stop criteria must be passed to the algorithm in order to achieve the same degree of error.

Algorithm 2 Alternative termination of the algorithm

```
⋮
Solve subdomains $\mathcal{F}_i\left(\lambda_{i-1}^k\right)$ in parallel on $i = 1 \dots N$ processes
if convergence then
    Solve subdomains $\mathcal{F}_i\left(\lambda_{i-1}^k\right)$ in parallel on $i = 1 \dots N$ processes
    for $i = 1$ to $N$ do
        $\lambda_i^k = \mathcal{F}_i\left(\lambda_{i-1}^k\right)$
    end for
    break outer loop
end if
⋮
```

In the Python implementation outlined in chapter 5.2 on page 63, the termination strategy from algorithm Algorithm 1 on page 58 is used, and the global difference norm e is used to determine convergence.

5.1.3 Stability and typical convergence rate

In [2] it was found that the Parareal algorithm was unconditionally stable for most time discretizations of parabolic PDEs, though it is not so for hyperbolic equations. Both the heat equation and the Stokes equations are parabolic, so we should expect the Parareal algorithm to perform well for these problems. This is naturally connected to the preliminary section on stability on page 14, which concludes with how Parareal works well with *stiff* problems and how we, by semi–discretizing parabolic PDEs, end up with systems of stiff ordinary differential equations. That Parareal can be used to solve the heat equation was demonstrated in [20], but the test results from attempting to run Parareal on the Stokes equations are described in chapter 6 on page 70.

By [2, Theorem 3], the convergence rate of the Parareal algorithm converges exponentially when the number of iterations increase, such that the difference norm between the exact solution and the solution found by Parareal has an exponential decay toward machine accuracy. The exact solution is here the serial solution found using the same solver and discretization in time (and space) as $\mathcal{F}$. Naturally, convergence will only occur as long as the underlying solvers are stable for the respective fine and coarse time steps. Actually, Theorem 3 describes the convergence of $\|\lambda_j - \lambda_j^k\|_\alpha$, where λ_j is the exact solution at time step j and λ_j^k the solution from iteration k at j. $\|\cdot\|_\alpha$ is the norm of the Hilbert space $H^\alpha(\mathbb{R})$, though we shall not discuss this further here.

The Parareal stability function for stiff problems,

$$|\lambda_i^k| = \left|(-1)^k \binom{i-1}{k} (R(z))^i \lambda_0\right|,$$

was found in [20, Theorem 3]. Here, $R(z)$ is the stability function for the coarse propagator. To bound the error, we must not only demand $|R(z)| \leq 1$ as $z \to -\infty$ to bound $(R(z))^i$, but we must also ensure that the binomial coefficient is restricted. As both i and k are determined by the number of subdomains and iterations, the coarse stability function is the only remaining degree of freedom. One must therefore choose $R(z)$ so that it also binds the binomial coefficient. Based on the growth of the binomial coefficient, it was found that the stability function of the coarse propagator must fulfill

$$\lim_{z \to -\infty} |R(x)| \leq \frac{1}{2}.$$

The restrictions on $R(z)$ was previously stated in chapter 2 on page 3, where we used it to determine the bound on θ,

$$\theta \in [2/3, 1],$$

that has received so much attention in this thesis. The authors in [20] points out that this is a conservative estimate. For $k \ll N$, the demands on the damping properties of the coarse solver are milder. Consequently, we may use smaller θ values in practice than what the analysis states. Thus the Crank–Nicolson method is not wholly inappropriate to use for the coarse solver, as long as care is taken to ensure that convergence is determined while k is still small compared to N. Ergo, a smaller θ value will only restrict the number of iterations, in a similar fashion to how a large k bounds θ.

5.2 Implementation

In this section an outline of the Parareal implementation is presented. Refer to the source code for the full implementational details. The algorithm was not written to actually run in parallel, and the part of the algorithm that should have run in parallel is realized through use of constructs like “`for i in range( noFineSolvers ):`”. This was mainly done because it is more efficient to develop serial code, and for our testing purposes it was not really necessary to have a fully parallel implementation running.

5.2.1 Implementational Overview

We here give a brief overview of the Python implementation and sketch the interface requirements on the fine and coarse solvers grain to make it easy to adjust solvers to future use with the algorithm.

The Parareal implementation done for this thesis is a sequential version – one process loops over all the fine subdomains instead of distributing them out to separate processes. This does of course result in a highly inefficient program, but it is, as previously mentioned, efficient from an implementational point of view. Parallel programming tends to be time consuming, even though the high level Python MPI interface hopefully would make it simpler. As the goal was to study how well the Parareal algorithm performs on a time dependent Stokes problem, it was not essential to have an efficient, parallel implementation, but rather have a functioning Parareal simulator up and running as quick as possible. If the serial algorithm performs well on the time dependent Stokes problem, it will obviously show the same traits when ported to a true parallel implementation.

To use the Parareal implementation you need to provide a set of classes implementing a small set of interfaces. The implementation does not require you to inherit from a provided set of base classes, or implement particular interface objects, such as one would do in other object oriented languages. It merely assumes that the required functionality is present. For our purposes, this is one of the nice features of Python – there is no need to create an extensive set of solver base classes, unless you would like to organize your code in this way. The goal of the implementation was to keep it as generic as possible, giving flexibility in the choice of fine and coarse solvers. It should also be easy to adapt previously developed solvers with only minor modifications.

The `SolverFactory` class

To instantiate an instance of the `Parareal` class you must provide a *solver factory class*, which is a class used to produce instances of coarse and fine solvers. Any Python class implementing the interface requirements shown below can be used as a solver factory.

```
class SolverFactory:
```

```
def getFineSolver  ( self, tStart, tStop, timeStep,
                     *noUnknowns )
def getCoarseSolver( self, tStart, tStop, timeStep,
                     *noUnknowns )
```

The functions should return an instance of a coarse or fine solver, respectively, designed to solve the PDE in the time domain [tStart, tStop] using the given time step. The anonymous parameters `*noUnknowns` are assumed to set the number of unknowns (number of nodes) in each spatial dimension. The Parareal implementation will pass `*noUnknowns` untouched to the `SolverFactory` (see the description of the Parareal implementation), and you are therefore free to set the format of the parameters. You could for example use this to specify different spatial resolution for fine and coarse solvers. As previously mentioned, the returned coarse and fine solvers can be either instances of the same class, or represent an all together different implementational scheme.

The `Solver` classes

The `CoarseSolver` returned by `getCoarseSolver`$(\cdots)$ should implement the following interface

```
class CoarseSolver:
    solveTimeDomain(self, tStart, tStop, startSolution)
    getIC(self)
    getSolution(self)
```

The `FineSolver` interface is of similar nature:

```
class FineSolver:
    solveTimeDomain(self, startSolution)
    getSolution(self)
```

The `solveTimeDomain()` functions solve the PDE in the given time domain, and must accept an instance of a *solution object* – `startSolution` – representing the initial condition for the subdomain. The coarse version should also reset the size of the time domain using `tStart` and `tStop` before solving the domain.

This last feature is due to a small optimization done in the Parareal algorithm to speed up the serial computation. We know that all λ_i^k values for $i < k$ are as accurate as they can be, since the initial condition λ_0^k has directly influenced all fine solvers up to $i = k$ (cf. previous discussion on the exact solution). To calculate λ_i^k for $i < k$ is therefore superfluous workload in the algorithm, and we can use "**for** $i = k$ to N" on the inner loop of the algorithm instead of running it from $i = 1$. For the efficiency to have any effect, the coarse solver must then be able to reset its time domain. The algorithm also utilizes this functionality to only calculate small sections of the time domain and then saving the solution to the appropriate λ, before proceeding to the next time domain. For a parallel computation optimization may not be necessary unless you have other work for the processes freed by this change in the algorithm.

getSolution() is expected to return the most recently calculated solution of the time domain in the solver (i.e the solution at tStop) and getIC() in CoarseSolver should return a solution object representing the the initial condition for the time dependent PDE.

The **Solution** class

Considering the Parareal algorithm in Algorithm 1 on page 58, it should be fairly clear what the interface to the Solution returned by getSolution() and getIC() object should be:

```
class Solution:
    __add__(self, other)
    __sub__(self, other)
    __copy__(self)
    norm()
```

The __add__ and __sub__ functions must of course return a new object instance containing the result of self$\pm$other. Even though the Parareal algorithm works well with coarse and fine solvers that use different discretization techniques or solvers working on different perturbations of the original model problem (see chapter 5.1.1 on page 58), we must still expect the objects representing the spatial solution at any time point to be able to interact with other solution objects, regardless of their origin is a coarse or fine solver. This is after all the basis for creating the new λ values. You therefore need to make sure that addition and subtraction with a mix of fine and coarse solutions is meaningful, such that Python expressions resulting in function calls of type coarseSol.__add__(fineSol) or fineSol.__sub__(coarseSol) are valid.

__copy__ must be present to make the objects work well with Python's copy module. Parareal uses this function to keep copies of the old λ–values. norm() should return some suitable norm of the solution object. This is used during the convergence check. Actually, what the Parareal algorithm does is to compute the difference norm between solution objects, and use this to estimate the error as a global space–time norm. This was discussed in chapter 5.1.2 on page 60. As the difference between to solution objects is a new solution object, these objects must be able to produce a norm.

The **Parareal** class

We now turn our attention to the Parareal interface: To create a Parareal instance you

Listing 5.1: Parareal Interface

```
class Parareal:
    def __init__(self, noProcs, tStart, tStop, dt,
                 convTol, solverFactory, *noUnknowns)

    def run( self, convTol=1.0e-8 )
    def runErrEstimate( self )
    def runConvTest( self )

    def getLambdaValues(self)
    def getFinalNorm()
```

must pass a `SolverFactory` instance as described above. It also expects the number of processes and the global time domain parameters in order to initialize the solvers. `dt` is the fine time step passed to the $\mathcal{F}$ solvers; the coarse time step is calculated by the algorithm. There is also a possibility to send in anonymous parameters intended to describe the number of unknowns in each spatial dimension. `*noUnknowns` is passed untouched to the `SolverFactory` functions, so they can describe an arbitrary discretization format. In fact, you can use it to send any extra information to your factory class, as the Parareal algorithm does not use it in any way, though its assumed intention is reflected in the variable name.

`run()` executes the body of the Parareal algorithm, and terminates when convergence is reached based on the input parameter. `runErrEstimate()` and `runConvTest()` are functions that support the extra calculations needed to perform the tests described in chapter 6 on page 70. They both run the algorithm with the convergence tolerance set to exactly zero, which forces it to run maximum number of iterations. They respectively return arrays with the error and convergence quality estimates for each λ. For `runErrEstimate()`, an exact solution using the fine solver (see chapter 5.1.1 on page 58) will be calculated prior to the run of the main iterative loop. Then, for each update of λ^k, the error compared to the exact solution is calculated and stored in a list that will be returned after the algorithm has executed. This functionality is useful when checking the convergence rate of the algorithm for a particular PDE, or when testing the algorithm implementation itself. For example, using the heat equation the algorithm should display the distinct exponential decay in error as the number of iterations increase, and the heat equation solvers were used as a test to check our algorithm implementation.

Note that the objects in the error estimate list are determined by the type returned from the `norm()` function of the `Solution` objects.

Example: Adapting Heat1 to use with the Python implementation of Parareal

As has been explained to some length in previous chapters, particularly, the heat equation has been the test model problem for most of the work done on this thesis. As was shown in [20], the Parareal algorithm behave beautifully when solving the heat equation as long as the solvers are stable for the particular time step the solver operates under.

As the solver is ultimately based on the `Heat1` solver that ships with Diffpack, it is reasonably safe to assume that our solver is correct and bug–free. This makes it ideal for testing the Parareal implementation, because if the algorithm does not show the proper convergence traits it is most likely due to the actual algorithmic implementation and not the PDE solver. To show how you could adapt a Diffpack solver for use with the Parareal implementation, the necessary modifications to the Diffpack `Heat1` implementation are shown here. The following adaptation of the solver to match the interface requirements set above was created early in the development process in order to test the Parareal implementation. It does not necessarily show the final heat implementation. The full heat class hierarchy is described in chapter 3.4.2 on page 24.

In chapter 7.3.2 on page 94, we will show how to create SWIG interface files for a Diffpack simulator. Once you have the basic interface file for the `Heat1` implementation, it is straight forward to extend it to fit the Parareal algorithm. We will here extend `Heat1` so that it can be used as both fine and coarse solver. The necessary extensions to the SWIG interface file are shown in 5.2 on the next page. The `getSolution()` returns a reference to a Diffpack `Vec_double` object. Python will only interpret this as a C/C++ reference and not a Python object instance, and since it is not a valid object instance you cannot treat it as such from your Python code. This will cause all operations done on the solution objects in `Parareal.run()` to fail, regardless of whether the Diffpack object implements the interface requirements or not. A Python interface to `Vec_double` is located in the `Diffpack2Python` module developed for this thesis[3]. The interface `Diffpack2Python.Vec_double` has been extended to meet the interface reuirements of the `Solution` objects, and you can combine the extension code from Listing 5.2 with Python logic for wrapping the returned vector references to give you a fairly clean solution. You could do this by inheriting from `Heat1`, and then shadow the `getSolution()` function from the base class as shown in Listing 5.3 on page 69. Incidentally, 5.3 also shows the class interface to `Heat1LinEq`, which is part of the Heat class hierarchy discussed in chapter 3.4.2 on page 24. The `Heat1LinEq` constructor is also discussed there, and is not shown in greater detail here. This concludes the introductory chapters on model problems, discretization techniques, algorithms and source code documentation. The following chapter is dedicated to the test results from funning the Parareal algorithm on the heat and Stokes solvers developed through chapters 3–5.

[3]`Diffpack2Python` is documented in appendix A on page 107.

Listing 5.2: Parareal extensions to the `Heat1.i` SWIG interface

```
%extend Heat1{
    // Make Heat1 support interface for Parareal solvers

    void solveTimeDomain( Vec_double& startSolution ){
        self->dof->vec2field(startSolution, *(self->u));
        self->dof->vec2field(startSolution, *(self->u_prev));
        self->fillEssBC();
        self->timeLoop();
    }

    void solveTimeDomain( double tStart, double tStop,
                          Vec_double& startSolution {
        self->tip->scan( oform( "dt=%f, t in [%f,%f]\0",
                         self->tip->Delta(), tStart, tStop));

        Heat1_solveTimeDomain__SWIG_0(self, startSolution);
    }

    Vec_double& getSolution(){
        Vec_double* vec = new Vec_double();
        self->dof->field2vec(*(self->u), *vec);
        return *vec;
    }

    //getIC() will be done in Python

}//end %extend Heat1
```

Listing 5.3: Heat1LinEq class interface: Parareal extensions to Heat1

```
from Heat1 import *
import Diffpack2Python

class Heat1LinEq(Heat1):
    def __init__( self, menuInitializer,
                  verbose=False, save=False ):

    def getSolution(self):
        """Return Python wrapped Diffpack pointer"""
        dpVecClass = Diffpack2Python.Vec_doublePtr
        return dpVecClass( Heat1.getSolution(self) )

    def getIC( self ):
        self.setIC()
        return self.getSolution()
```

Chapter 6

Running Parareal on the Stokes equations: Test results

Using the results from all the preceding chapters and the technical details of programming covered in subsequent chapters and appendices, we are in a position where we can examine the some of the possible similarities and differences between running the Parareal algorithm on the heat equation versus using it to solve the Stokes equations. This chapter is divided into two main sections: in the first we will study the behaviour of the algorithm when θ varies in the Parareal stability range $\theta \in [2/3, 1]$. This test will be based on how the error compared to the *serial* solution (i.e. the Parareal exact solution) develops as the number of iterations increases. In the second section we will try to determine if the convergence strategy we have chosen to use in Parareal mirrors the true error compared to the serial solution, i.e. see if the convergence strategy seems valid. The second test will be particularly interesting, as we have not seen any previous tests on whether the norm over the λ–values actually is a good reflection of the error. All tests will be done numerically.

6.1 Experimenting with θ–stability

Throughout this report, we have advocated Parareal stability if the coarse solver is based on a discretization where $\theta \in [2/3, 1]$, and where the θ–value for the fine solver is chosen such that the solver is stable for the given fine time step Δt. The coarse solver must of course also be stable for its time step, ΔT. We remark, as has been stated earlier, that the exact solution for Parareal is the solution found by the *serial computation based on the fine propagator over the entire time domain*. Convergence and error of of the algorithm is measured against this serial solution.

For our (parabolic) equations, we expect the Parareal algorithm to have an exponential decay in error as the number of iterations, k, increase. This was established in chapter 5.1.3 on page 61. As the error is an exponential function of k, we would expect the *logarithm* of the error to decrease in a linear fashion as k goes to N (number of processes), until it reaches machine accuracy, a_m. It will then become become asymptotic with a_m. A typical example of this behaviour is shown in Figure 6.1

Before we proceed to study the stability function for our model problems, we need to look a bit closer at the slightly paradoxical nature of Parareal stability, which separates it from the traditional notion of stability. Usually, when one speaks of instability, performing one more step in whatever algorithm one is using will only worsen the situation, and cause your numerical solution to deviate further form the true solution. Not so for the Parareal algorithm. The number of iterations is bounded by N, and the algorithm will *always* converge to its exact solution given that we have infinite precision. This does not conform to the typical notion of instability. On the other hand, if it uses N iterations to converge, the algorithm itself becomes pointless; you have then in essence performed a serial computation with the added factor of communication overhead. Note that the error in in the algorithm may still become arbitrarily large before dying out at $k = N$, and by that it shows typical signs of instability.

6.1.1 The test parameters

We will look at the effect on the Parareal convergence rate on our two model problems when θ is varied around the stability threshold $\theta = 2/3$. We will run the maximum number of iterations, and compute the global space–time error of λ^k compared to the serial solution, such that we have an error function

$$e(k) = \sqrt{\sum_{i=1}^{N} \|\lambda_i - \lambda_i^k\|^2 \Delta t}.$$

From the analysis we know that the turning point in terms of stability should occur at $\theta = 2/3$, and the logarithm of $e(k)$ should show a linear plot. As the properties of the Parareal algorithm are known to be evident when used to solve the heat equation we will present the the plots of the error function from the heat experiments as blue prints to the expected behaviour of the Stokes error.

For the Stokes equations we solve for two unknowns in each Parareal iteration, and we must decide on how to handle the definition of the norms used in the error function. Because we only solve the pressure up to a constant whereas velocity is solved exact, the norms used to determine error and convergence on the Stokes problem will be based on velocity only.

In the following tests, common settings has been used for Parareal and the time discretization to ensure that the comparisons are as valid as possible. The number of processes, N, has been set to 40, such that we are certain that the error plots have a fairly high resolution and will show as much of their respective properties as possible. The algorithm will be forced to run the maximum number of iterations as the stop criteria is set to exactly zero, which will only happen when $k = N$, as $\lambda_N^N - \lambda_N^{N-1} = 0$. The time domain will be

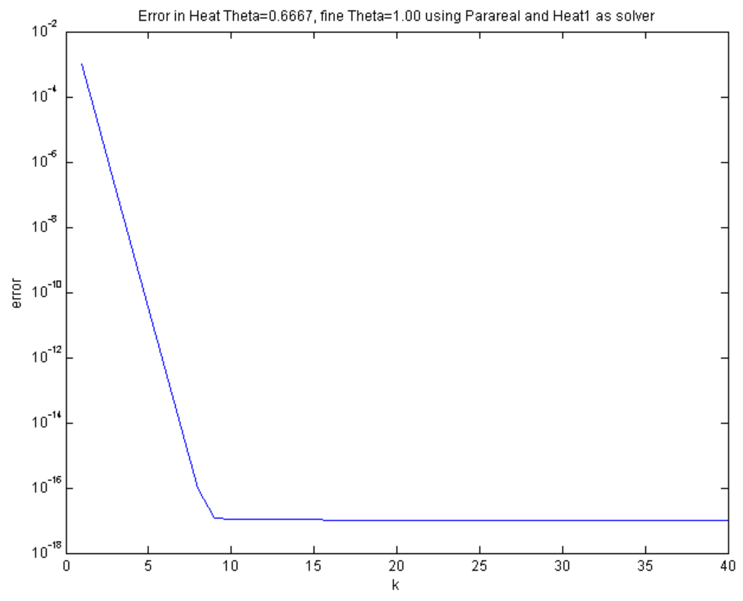

Figure 6.1: The logarithmic development of the Parareal error as k goes to $N = 40$ for the heat equation. $\theta = 2/3$ for the coarse solver and $\theta = 1.0$ for the fine solver.

$t \in [0.0, 3.6]$, and the fine time step set to $\Delta t = 0.01$. These settings will give us $\Delta T = 0.09$, such that we have 9 fine time steps per subdomain in time. The algorithm will be run for $\theta = \{2/3 - 1/10,\ 2/3,\ 2/3 + 1/10,\ 1\}$.

6.1.2 Parareal error for the heat equation: "blue–prints"

As depicted in Figure 6.1, the algorithm is clearly stable using $\theta = 2/3$ for the coarse solver; the plot shows a consistent convergence as the number of iterations increase. In Figure 6.1, the discretization of the fine solvers use $\theta = 1.0$, which is Implicit Euler. As the only criteria for the fine solver is that it should be stable, we could use other values as well, for example have fine solvers with $\theta = 1/2$ or $\theta = 2/3$, without actually affecting the stability. We can see this property in Figure 6.2 and Figure 6.3, where the fine theta value is alternately set to equal the coarse solver's θ–value and to $1/2$. We see that the choice of fine θ value have very little influence on how many iterations you need before the algorithm converges, which is in agreement with the convergence analysis. In the latter case the discretization method of the time derivate is Crank–Nicolson, which is a second–order in time discretization method, although that has no influence on the *Parareal* error, only on the error compared to the *analytical* solution.

You should also be able to run the fine solver with Explicit Euler ($\theta = 0$), as long as you make sure that the time step is small enough. For the tests done here we will not use Explicit

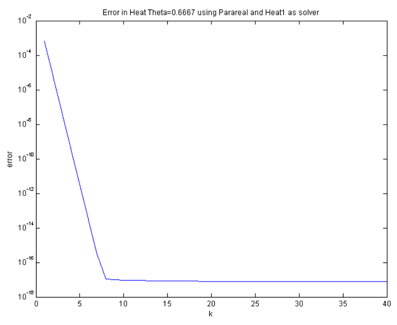

Figure 6.2: Error for heat as $k \to N$ using $\theta = 2/3$ for both coarse and fine solvers.

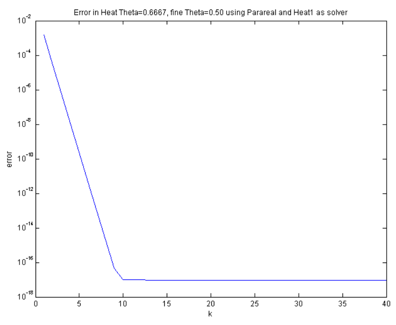

Figure 6.3: Error for heat as $k \to N$ using $\theta = 2/3$ and $\theta = 1/2$ for coarse and fine solvers, resp..

Euler, simply because it is too inefficient due to the time step restrictions. As the Parareal implementation is actually not implemented in parallel, it has a far longer execution time than the equivalent serial solver. To be forced to further reduce the fine time step would cause the tests to use far too much computation time compared to the relative little new insight the results would add to the existing tests. The computation time is particularly important for the Stokes equations, which, due to the increased complexity, has in itself a much longer execution time than the heat equation.

For the coming tests we will mostly let the fine and coarse solvers use equivalent θ values, as the fine solver will always be stable if the coarse solver is stable. This choice is merely for convenience, as it slightly reduces the number of parameters we must handle to fully describe the test cases. We will only introduce separate θ values if tests show a sudden influence from the fine θ value.

6.1.3 Unstable θ values

We begin our stability tests by setting $\theta = (2/3 - 1/10)$. As expected, the algorithm show classic Parareal instability for both the heat and the Stokes equations, where the error begin to increase after a certain number of iterations, though neither model problem show instabilities of any drastic scale under these particular circumstances. The respective error functions are shown in Figure 6.4 and Figure 6.5.

Having noted that there is indeed an instability for the heat equation here, we must point out that for this particular case, the error increase does not happen until after roughly 10 iterations, when the error is below 10^{-12}. For the heat equation we can clearly see the conservativeness of the $\|R(z)\| \leq 1/2$ restriction, as the instability is not pronounced until *after* the algorithm has run enough iterations to make the error very small. Depending on your stop criteria, the algorithm will likely have converged before the instabilities occur. Neither does the error have time increase by much before the convergence properties of the

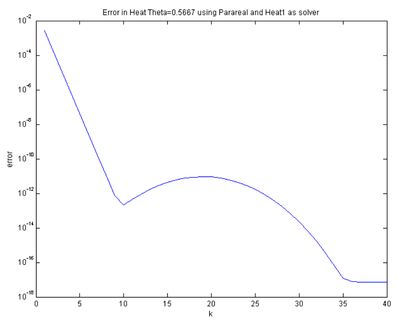

Figure 6.4: Error for heat as $k \to N$ using $\theta = 2/3 - 1/10$ for both coarse and fine solvers.

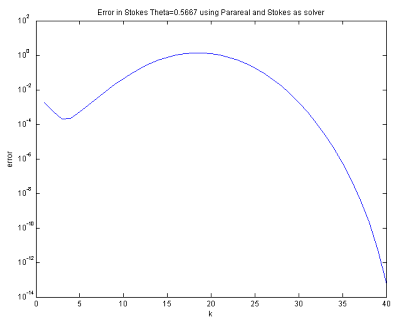

Figure 6.5: Error for Stokes as $k \to N$ using $\theta = 2/3 - 1/10$ for both coarse and fine solvers.

algorithm take effect, but it is no longer exponential when it resumes its descent at $k = 20$. If the algorithm did not converge at 10^{-12}, it is likely that it will not converge until $k \in \langle 30, 35 \rangle$, which implies that you have gained very little of the potential speedup the algorithm can give. We must also emphasize that by using a different discretization, changing the number of processes or otherwise changing the settings, this error could have become much larger, and it could also have occurred at a much earlier stage.

For the Stokes problem, the error show the same basic behaviour as for the heat equation, but here the instability has more effect as it appears long before the error has had any possibility of decreasing to an acceptable level. It clearly influences the number of iterations needed before the algorithm converges, which would probably not be before we were close to the maximum number of iterations. This does not necessarily imply that the algorithm cannot be used on the Stokes equations. For $\theta = 2/3$ we do after all expect the algorithm to be unstable, which it clearly is here. The instabilities actually make Parareal equations conform to the expected behaviour. We also note that the error becomes much larger than the equivalent run of the heat equation, and we will not reach machine accuracy at all before $k = N$. The differences we observe compared to the heat equation could be due to different eigenvalues, since the equations will discretize differently in the spatial domain. Different eigenvalues will again influence the stability function, c.f. chapter 2.2.1 on page 10, and thus account for the differences between the error functions. The differences in error could thus simply be due to how the two systems respond to the current settings, both spatial and temporal, rather than any potential trouble running the Parareal equation on the Stokes problem.

Most of the θ values used in this chapter are solely of interest in studying the Parareal error. In a real–life situation, you would normally choose between using Explicit or Implicit Euler, or Crank–Nicolson, to discretize your time derivatives. Merely to show that Crank–Nicolson is not an unconditionally good choice to use under Parareal, even though it is a second–order in time method, error plots for the heat and the Stokes model problem are

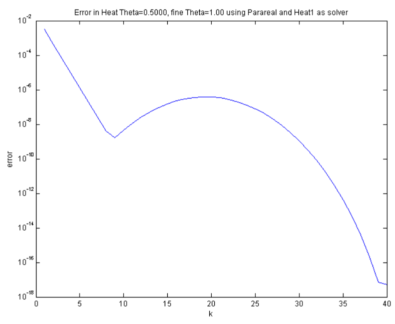

Figure 6.6: Error for heat as $k \to N$ using Crank–Nicolson for the coarse and Implicit Euler for the fine solvers.

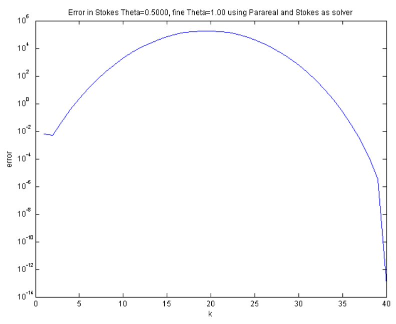

Figure 6.7: Error for Stokes as $k \to N$ using Crank–Nicolson for the coarse and Implicit Euler for the fine solvers.

shown in Figure 6.6 and Figure 6.7, respectively. The similarities to $\theta = (2/3 - 1/10)$ are clear, and the only difference for heat is that the instability makes itself known at a slightly earlier iteration, while the error grows larger and will not fully die out until $k = N$. The same observations apply to Stokes, but when you use Crank–Nicolson with the current variable settings on Stokes, the error grows very large compared to other tests we have seen.

With these preliminary observations done, it should also be clear that we for heat again see that the restrictions on θ are very cautious. Depending on how you set your error tolerance, you could very well use Crank–Nicolson as the coarse solver under these circumstances. It is stable up to about 10 iterations, when the error is reduced to about 10^{-9}. As long as the algorithm converges before this, nothing will seem amiss. The observations from chapter 5.1.3 on page 61 therefore holds true: as long as $k \ll N$, which in this case is equivalent to $10 \ll 40$, the algorithm will appear stable. For the Stokes equations however, the error is barely below 10^{-2} before the solution becomes unstable for this particular test case.

6.1.4 Stable θ values

The threshold of the stability domain: $\theta = \dfrac{2}{3}$

For $\theta = 2/3$ we expect the algorithm to be stable and exponentially convergent. By Figures 6.1–6.3 we already know this to be the case for the heat equation. The matching test for the Stokes equations is presented in Figure 6.1.4. The algorithm is stable in the global sense, as we have consistent convergence as k goes to N. The error never begin to increase as it would for the classic definition of stability. Neither does it show the smooth exponential decay in time like the algorithm shows for heat equation.

Remember that the plots of $\log(e(k))$ show the error over *all* λ–values compared to the values you would have from a serial computation. The "bump" in convergence shows shows echoes of the shape of the instability plots for $\theta = (2/3 - 1/10)$. It is likely due to small instabilities in *some* of the λ–values for a given iteration, which will be damped by the overall reduction of the global error $e(k)$. As the global error consistently decreases toward machine precision, if not in a completely exponential manner, one could see this small local instability more as a precision problem rather than an instability issue. In this view, a selection of your λ–values could be slightly further from the serial solution than others, depending on your convergence criteria.

We only do numerical tests here, so we can only propose conjectures. A possible influence to the local instability problem could be caused by Stokes classifying as a differential algebraic equation (DAE), as we have briefly mentioned earlier. DAE adds restraints to the type of temporal solver we may use and require so–called *stiffly accurate* methods to work, c.f. [1]. This could possibly influence the range of θ–values, as they must produce coarse solvers that have enough damping to keep the Parareal error bounded. We were not able to verify with full certainty that discretizations using $\theta \in [2/3, 1]$ are stiffly accurate, although both the Crank–Nicolson and Implicit Euler are stiffly accurate methods. We are fairly confident that using $\theta \in [1/2, 1]$ will give stiffly accurate methods, but this still leaves room for doubt as to wether $\theta = 2/3$ is an appropiate method for DAE problems.

Nevertheless, numerically, the algorithm appears to be stable, which is all we can conclude with here. The error does not grow, and you would be able to converge before $k = N$, although not as fast as for the heat equation.

Stability at $\frac{2}{3} < \theta$ and Implicit Euler

As there was clearly something influencing convergence for the Stokes equations when $\theta = 2/3$, we examine the error function as $\theta = (2/3 + 1/10)$. For a complete reference the test result for heat will also be presented, although it is, as expected, fully stable. We will also present the results from using Implicit Euler to discretize both our model problems.

From Figure 6.10, we see that the Stokes error show identical behaviour to the equivalent heat test; the numerical results show the algorithm as stable without any of the uncertainties from $\theta = 2/3$. It therefore seems as though whatever source to the instabilities disappear as θ moves away from $2/3$. The Stokes model problem has a slightly lower convergence rate than the heat equation which, analogous to the differences in instability that we pointed out for $\theta = (2/3 - 1/10)$, probably only indicates that the factor governing convergence is problem dependent, relying on for example the eigenvalues. As a last comment, further experiments indicated that all traces of the local instability/precision problem disappears between $\theta = (2/3 + 0.05)$ and $\theta = (2/3 + 0.06)$.

Moving on to using Implicit Euler, i.e. $\theta = 1.0$, we see by figures 6.11 and 6.12 that we have fully convergent error functions for both the heat and the Stokes equations. We can note that for the heat equation, the convergence rate increases as we move away from

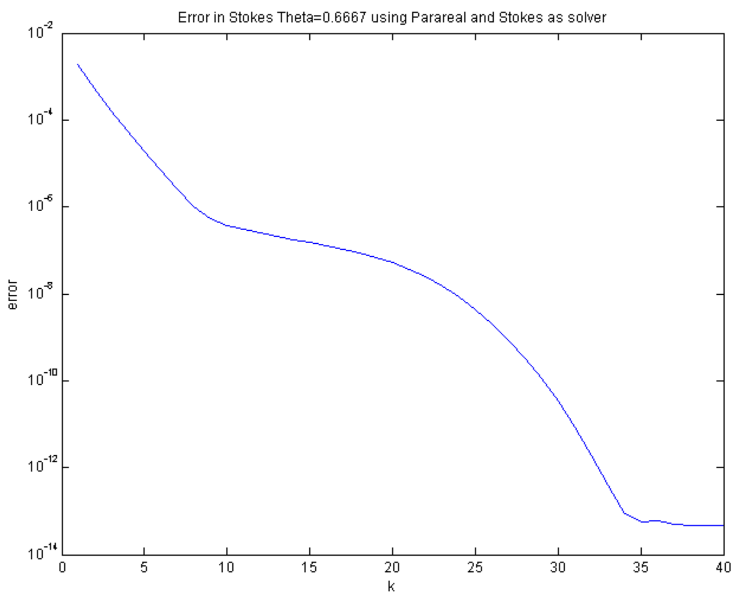

Figure 6.8: Error for Stokes as $k \longrightarrow N$ using $\theta = 2/3$ for both coarse and fine solvers.

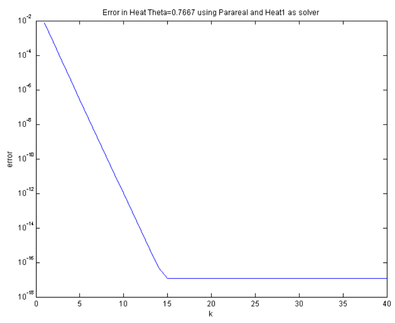

Figure 6.9: Error for heat as $k \longrightarrow N$ using $\theta = 2/3 + 1/10$ for both coarse and fine solvers.

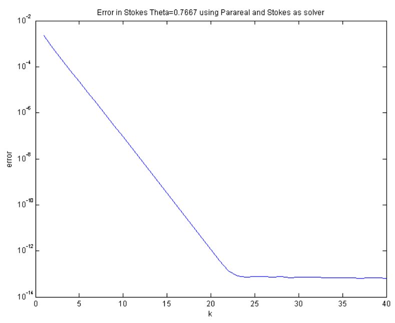

Figure 6.10: Error for Stokes as $k \longrightarrow N$ using $\theta = 2/3 + 1/10$ for both coarse and fine solvers.

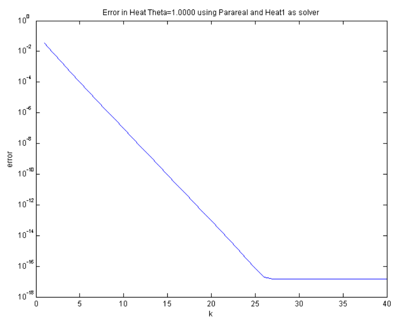

Figure 6.11: Error for heat as $k \rightarrow N$ using $\theta = 1.0$ for both coarse and fine solvers.

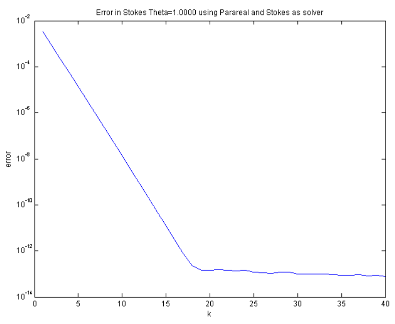

Figure 6.12: Error for Stokes as $k \rightarrow N$ using $\theta = 1.0$ for both coarse and fine solvers.

$\theta = 2/3$ and toward Implicit Euler. This could possibly be because the closer θ is to $2/3$, the closer the coarse propagator is to a Crank–Nicolson scheme, which is the only second–order in time scheme of all the possible choices of θ. For the Crank–Nicolson scheme, the second–order error constant is zero, leaving the third–order constant to describe the error. As θ goes to $1/2$, the second–order error constant approaches zero accordingly. A scheme using $\theta = 2/3$ will thus be closer to a second–order in time scheme than schemes where θ goes to 1. Therefore, an increased convergence rate could be caused by moving further and further away from $\theta = 1/2$, such that the coarse estimates of the solution for each time step will not be of as good quality as those found by a solver closer to the second order Crank–Nicolson. Once again, this is simply conjecture.

Conversely, the opposite is true for the Stokes equations: the closer you are to using Implicit Euler, the better the convergence rate will be. We will not try to explicate the the reasons for the separate behaviours on this point here, and merely point this out as an interesting numeric fact that should be studied further. It can also be used as a tentative *practical* guideline showing that for the Stokes equations one should use Implicit Euler for best possible convergence, whereas $\theta = 2/3$ (or $\theta = 1/2$) will be most efficient for the heat equation.

6.1.5 Stability for the Stokes equations

To conclude, we see that Parareal seem to be stable for the Stokes equations based on our numerical experiments, but caution that there are finer points that must be delved into before one can state this with certainty. One should particularly examine the cause for traces of instability for $\theta = 2/3$.

6.2 Convergence tolerance quality

In this chapter we will try to determine whether the stop criteria we opted for in chapter 5.1.3 on page 61 was a good choice. Ideally, one would like the stop criteria to reflect the true error, i.e. the error compared to the exact solution for Parareal. We have not found any references to testing this stop–criteria in the Parareal literature, and it seems that it is just generally assumed to appropriately reflect the true error in the algorithm at iteration k. Of course, the best convergence check we could possibly have would be to compare it to the exact serial solution, but if we had the exact serial solution there would naturally be no point in running Parareal. In choosing to base the stop criteria on the space–time norm

$$e_p(k) = \sqrt{\sum_{i=k}^{N} \|\lambda_i^k - \lambda_i^{k-1}\|^2 \Delta t},$$

we therefore hoping that it will reflect the corresponding difference norm to the serial solution,

$$e_s(k) = \sqrt{\sum_{i=k}^{N} \|\lambda_i - \lambda_i^{k-1}\|^2 \Delta t},$$

where λ_i is the exact solution at time step i. To terminate the algorithm you would do the comparison

$$e_p(k) < \varepsilon_p,$$

which is a substitute for what we actually want to express: the error introduced by the Parareal algorithm that we are willing to accept, i.e.

$$e_s(k) < \varepsilon_s.$$

As it stands, we have no knowledge on how we should choose our stop criteria ε_p such that we actually end up with ε_s as the global error of our solution compared to the serial solution.

If $e_p(k)$ mirrors $e_s(k)$, the ratio

$$\frac{e_p(k)}{e_s(k)} = a,$$

where a is a constant, should exist. Numerically, we will assume that all values close to a really is a. If a exists it would give us an idea of how we must set e_p in order for our chosen stop–criteria to reflect the actual error, as we could just use

$$\varepsilon_p = a\varepsilon_s.$$

For this test we have used the same settings as for the stability tests, but instead of letting θ be the main variable, we change N for each test. We will force the algorithm to run maximum number of iterations to make sure that we do not miss information due to premature convergence. As we will compare $e_p(k)/e_s(k)$, we must expect this fraction to produce nonsense when the Parareal algorithm reaches machine precision, as we will then have division between very small numbers. This is usually not a good idea, but as will not be interested in what happens after we reach machine precision we will ignore any irregularities occurring after the algorithm would clearly have converged.

We will combine the following parameters in order to try to numerically verify our stop criterion

$$\theta = \begin{cases} 2/3 \\ 1.0 \end{cases}, \quad N = \begin{cases} 9 \\ 18 \\ 24 \\ 40 \end{cases}$$

6.2.1 Convergence test results

Running Parareal on heat and Stokes with the specified settings, we end up with a series of of ratios that we can represent by plots to visualize the (hopefully) constant parts of the ratio. The tables 6.1–6.4 show the approximate ratio a for each combination of N and θ. The plots on which the data in these tables are based can be found in figures 6.13–6.28 on page 82. As we can see from the summaries in each table, we are able to find an approximate a for all our test runs, which indicates that there is indeed possible to adjust ε_s through ε_p. As expected, the plots show irregularities once they go beyond a certain k, due to division by very small numbers, but this is of no consequence, as one would reach and determine convergence before this. Some of the plots also show jumps between the initial iterations, but they stabilize quickly at a.

There will always be uncertainties connected to such numerical results, and for some of our test runs the listed a is perhaps a bit more dubious than others. All in all, the plots show a consistent behaviour in that there is always possible to find a, which is far better than expected. The area where the constant a is defined seems to follow the convergence plots fairly well, and there is usually a match between when machine precision is reached and when a is no longer present. As an example one can use Figure 6.24 where a disappears at $k \approx 17$, and compare it to the convergence plot in Figure 6.12 which shows convergence at roughly the same iteration. This connection does by no means seem to be guaranteed; in the equivalent situation for Heat convergence is reached at about $k = 25$, whereas a is only visible while $k < 15$. Admittedly, the error displayed in the convergence plot is very small at $k = 15$. In future work it would be interesting to determine when and why a exist – provided that these numerical tests are correct. Our tests merely indicate that there is a possible connection between e_s and e_p, and further work that analyze how this stop–criteria operates under Parareal should be done before one can confirm our numerical results with any certainty.

The ratios listed in the tables vary, although they seem be fairly invariant to the number of processes used, and if anything, they seem to rather cluster around the combination of model problem and θ value. The very obvious exception to this behaviour is shown for the heat tests where $\theta = 2/3$. Here, a vary between $a \approx 2$ and $a \approx 18$. Why this happens, and whether it is of any practical importance, should also be studied further.

Since the size of a does not seem to follow any discernible pattern at this point, one should probably adapt the same strategy as one uses for linear solvers: provide a good measure for variations in the ratio by choosing, for example, $a = 26$, as our largest value is 18. a would then hopefully stay inside this bound as one adjusts N, θ, Δt etc. etc.

$N = 9$	Constant ratio for $k < 6$	Ratio: $a \approx 9$
$N = 18$	Constant ratio for $k < 14$	Ratio: $a \approx 5.6$
$N = 24$	Constant ratio for $k < 17$	Ratio: $a \approx 5$
$N = 40$	Constant ratio for $k < 25$	Ratio: $a \approx 5$

Table 6.1: Convergence ratio for when running heat with $\theta = 1.0$

$N = 9$	Constant ratio for $k < 5$	Ratio: $a \approx 3.8$
$N = 18$	Constant ratio for $k < 15$	Ratio: $a \approx 9$
$N = 24$	Constant ratio for $k < 8$	Ratio: ≈ 18.3
$N = 40$	Constant ratio for $k < 15$	Ratio: $a \approx 2$

Table 6.2: Convergence ratio for when running heat with $\theta = 2/3$

$N = 9$	Constant ratio for $k < 4$	Ratio: $a \approx 5$
$N = 18$	Constant ratio for $k < 11$	Ratio: $a \approx 4$
$N = 24$	Constant ratio for $k < 15$	Ratio: $a \approx 3.5$
$N = 40$	Constant ratio for $k < 17$	Ratio: $a \approx 3$

Table 6.3: Convergence ratio for when running Stokes with $\theta = 1.0$

$N = 9$	Constant ratio for $k < 7$	Ratio: $a \approx 2.5$
$N = 18$	Constant ratio for $k < 8$	Ratio: $a \approx 2$
$N = 24$	Constant ratio for $k < 10$	Ratio: $a \approx 2.5$
$N = 40$	Constant ratio for $k < 20$	Ratio: $a \approx 2$

Table 6.4: Convergence ratio for when running Stokes with $\theta = 2/3$

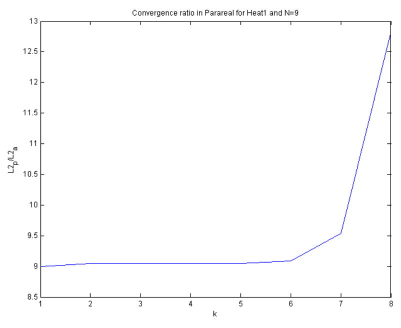

Figure 6.13: Convergence ratio for heat using $N = 9$ and $\theta = 1.0$.

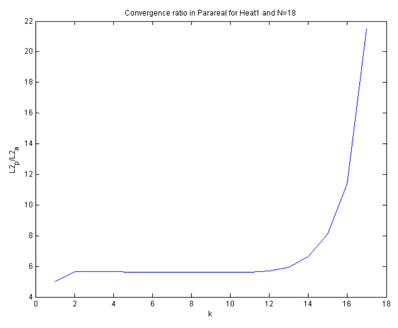

Figure 6.14: Convergence ratio for heat using $N = 18$ and $\theta = 1.0$.

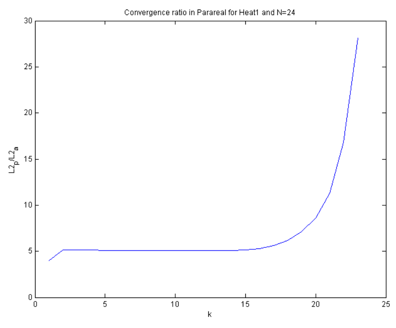

Figure 6.15: Convergence ratio for heat using $N = 24$ and $\theta = 1.0$.

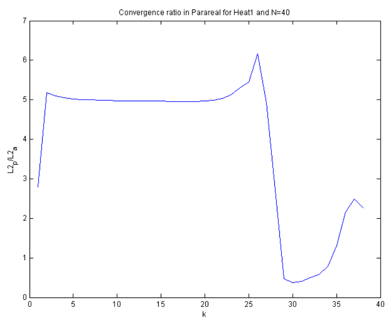

Figure 6.16: Convergence ratio for heat using $N = 40$ and $\theta = 1.0$.

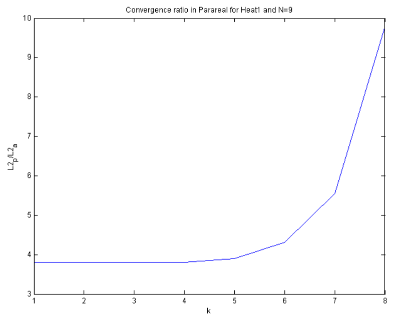

Figure 6.17: Convergence ratio for heat using $N = 9$ and $\theta = 2/3$.

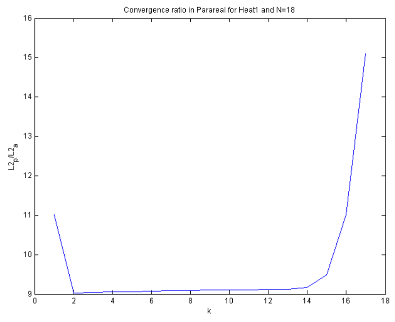

Figure 6.18: Convergence ratio for heat using $N = 18$ and $\theta = 2/3$.

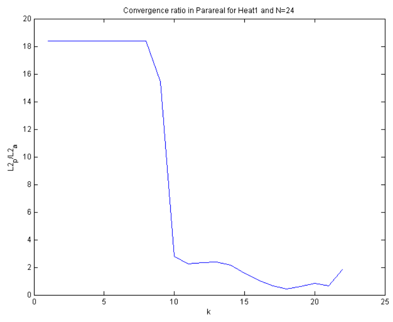

Figure 6.19: Convergence ratio for heat using $N = 24$ and $\theta = 2/3$.

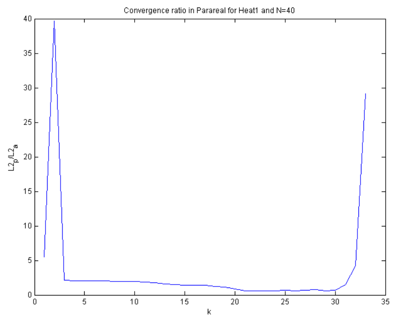

Figure 6.20: Convergence ratio for heat using $N = 40$ and $\theta = 2/3$.

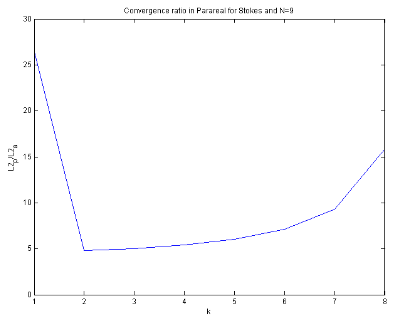

Figure 6.21: Convergence ratio for Stokes using $N = 9$ and $\theta = 1.0$.

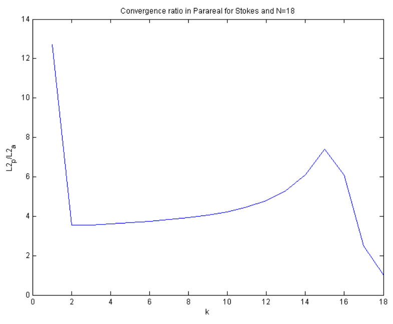

Figure 6.22: Convergence ratio for Stokes using $N = 18$ and $\theta = 1.0$.

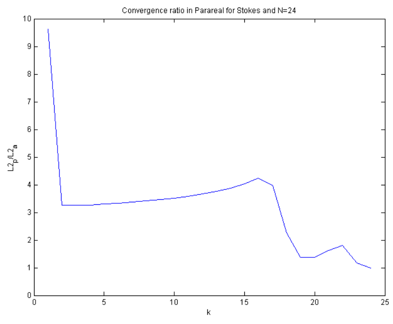

Figure 6.23: Convergence ratio for Stokes using $N = 24$ and $\theta = 1.0$.

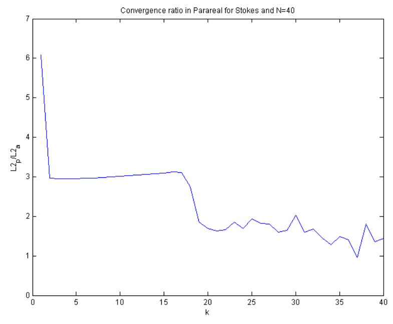

Figure 6.24: Convergence ratio for Stokes using $N = 40$ and $\theta = 1.0$.

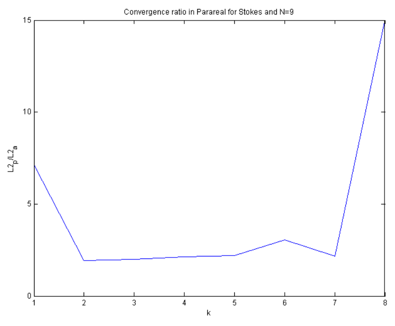

Figure 6.25: Convergence ratio for Stokes using $N = 9$ and $\theta = 2/3$.

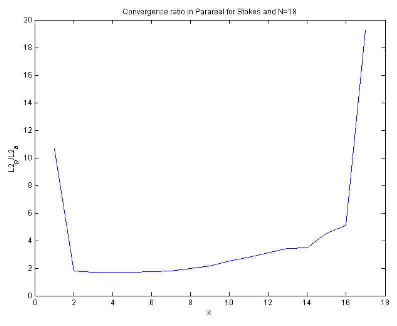

Figure 6.26: Convergence ratio for Stokes using $N = 18$ and $\theta = 2/3$.

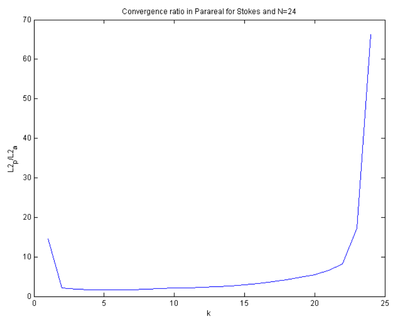

Figure 6.27: Convergence ratio for Stokes using $N = 24$ and $\theta = 2/3$.

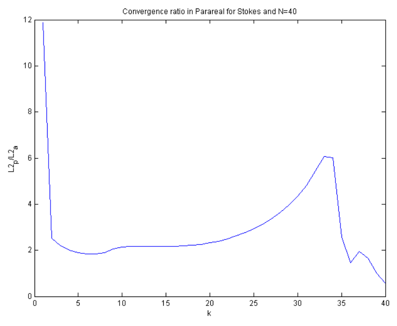

Figure 6.28: Convergence ratio for Stokes using $N = 40$ and $\theta = 2/3$.

6.3 Summary

The numerical tests indicate that the Stokes equations generally operate well under Parareal. There are some uncertain factors that must be resolved, which we have pinned as possibly relating to the DAE aspect of the Stokes equations. This is particularly the case in the lower bound of θ. As $\theta \rightarrow 1.0$ the algorithm seems to be as stable and convergent for the Stokes equations as it is for the Heat equation. This is a numerical result only, and should be verified with a thorough analysis.

Our initial tests of the convergence criteria indicate that

$$e_p = \sqrt{\sum_{i=k}^{N} \|\lambda_i^k - \lambda_i^{k-1}\|^{\Delta} t}$$

is a good estimate of the actual error in λ^k, as it seems to be only removed from the true error by a constant a. So far, this verifies that the stop criteria e_p is indeed a good choice, which seems to have only been assumed in the literature on Parareal.

Chapter 7

Working in the Microsoft Windows Environment

The scientific community seems to favor Unix based platforms, but most of the tools found in the Linux world has an equivalent on a Windows system. The Python interpreter has, of course, a Windows build, and tools like SWIG for automatically creating the wrapper code between C/C++ and Python use the same interface as the Linux version. Python modules such as Numeric and Pysparse work just as well in Windows. Diffpack also ships with a Windows version.

When we began our work, there was no clear documentation for using the scientific tools required for the thesis on the Windows platform. Our colleagues and tutors felt uncomfortable using Windows, and we therefore wished to create a guide for using Windows as en environment for scientific computing. This chapter is meant as a practical guide on how to employ the central tools used under this thesis, and is primarily based on product documentation and personal experiences. Hopefully this will be of use for later students and researchers that need to use the same set of tools in a Windows based environment. The main topics are how to set Python up on a Windows system, how to install new Python modules, and how to use Diffpack and SWIG through Visual Studio 6.0. The idea is that one can combine Diffpack, Python module code and SWIG in Visual Studio projects, which can be compiled and linked in one operation. This renders development of Python extension modules a smooth and efficient operation.

This chapter assumes that you are not a novice Windows user, and that you have knowledge of the Python environment and how you would write Python extension modules in C/C++, albeit not necessarily from a Windows viewpoint. Some familiarity with programming C/C++ in Visual Studio 6.0 is expected, so you should, for example, know how to create new projects and compile and link your source code. It is not assumed that you have written extension modules for Python through Visual Studio, or that you have used either Diffpack or SWIG in this setting.

The descriptions found here are based on the tools used when working on this thesis.

Our setup was as following:

Micrososft Windows XP Professional version 2002, w/Service Pack 2.0

Visual C++ 6.0 with Service Pack 6.0 installed for Visual Studio 6.0

ActiveState Komodo Professional version 3.1.0

ActivePython version 2.3.5 (#62, Feb 9 2005, 16:17:08), build 236

SWIG version 1.3.25 for Windows (`swigwin-1.3.25.zip`)

Numeric (Python module) version 23.8, often referred to as NumPy

Pysparse (Python module) version 0.33.029 and 0.34.032

Intel Math Kernel Library (MKL) version 7.0

Diffpack Mini Distribution for Windows version 4.0

7.1 Python in Windows

This section provides a brief overview of how to set Python up on a Windows system, and how to make both the release and debug versions available for development purposes. We will start with the one of the standard distributions of Python for Windows developed by Active State, and then look at installing Numeric and Pysparse.

7.1.1 The Active State Python distribution

The Python distribution used on Windows will usually be either the one provided for download from `www.python.org` (open source distribution), or ActivePython developed by Active State. Although it is free to download and use, ActivePython is a commercial product. On the download site at `www.python.org`, you can find Python Windows installers that do all the necessary work for setting up Python on your Windows machine, and Active State's home pages give easy access to their Python distribution installer. We prefer to use the ActivePython distribution, mostly because they also give easy access to, and installation of, the debug build of Python, which we could not find for the open source version. Active State also develops a nice IDE called *Komodo* for dynamic languages that support several scripting languages, like Python, Perl and PHP. Komodo runs on both Windows and Linux, but is a commercial product, and you will need to purchase a lisence to download the software.

Both Python distributions are straight forward to install once the installers are downloaded. Note that all Python distributions must be installed directly on the root of your drive at `C:\PythonXX\`, and not in "Program Files" or some other folder. Once installed, you should be able to use `python.exe` from the *Command Prompt* (`cmd.exe`), just like you would in a Linux environment.

For development purposes it is practical to have access to a debug build of the Python libraries, to use with debug builds of your extension modules. We have not found a pre–built debug version of the open source version of Python, and it only seems to be available through building it yourself from the source code, which, although probably not difficult, is more work than the simple and easy procedure for installing the Active State debug version. The Active State debug release of ActivePython is available for download through their FTP site, `ftp.activestate.com/ActivePython/etc/`. You should be able to find a `*.debug.zip` matching your Python release there. For this thesis `ActivePython-2.3.5-236-win32-ix86.debug.zip` was used to match the `2.3.5-236-win32-ix86` installation. To install the debug version of ActivePython, simply unzip to a temporary directory, and run the `install.py` script. To use the debug version of your extension module in some Python script, use `python_d.exe` and not `python.exe` as the interpreter of your Python scripts. The Python environment will load the correct build of modules, provided that they follow the naming convention `moduleName_d.dll` for debug modules. More details on creating and naming release and debug versions your extension modules are given in chapter 7.3.1 on page 93.

7.1.2 Downloading and Installing new Python Modules

Each release of a Python module is usually provided for download on `www.sourceforge.net` in several different *types* (software bundles/packages) to satisfy different needs and platforms. In general, the package types divide into two categories: bundles with or without the source code included. In other words, you either will or will not have to take compiling and linking into consideration. The `.rpm` and `src.rpm` types are intended for use with *RPM Package Manager* (RPM), which is a tool for installing, verifying, querying and updating software packages. RPM is a Linux tool and the `*.rpm` types need not be considered further. The remaining package types are usually the `.win32-pyX.X.exe` type, which is a 32-bit Windows installer, and the `.tar.gz` and `.zip` packages, which include the source code. `X.X` refers to the Python version.

Python is open source, but the Windows version is built (bizarrely) using Visual C++ 6.0, which is a commercial product with fairly expensive software licenses. When building or installing extension modules for Python in Windows using the source code packages. You are therefore principally required to use the Visual Studio 6.0 C/C++ compiler, because extension modules must be created with the same compiler and linker that was used to build the Python environment. Ergo, to compile and install a free, open source product, you are actually required to buy an expensive license.

However, most Python packages and modules for Windows come as a self-installing binary `.exe` build, which turns installation into a black-box procedure. Using the self-installers, you are not required to have Visual Studio installed to handle building of possible C/C++ code. The binary installers also give Windows users a nice, conform way to install extra Python packages, as the package is installed like any other software on a Windows machine. The `.exe` build is provided for most extension modules. Some smaller modules do not provide the self-installing build for Windows, and avoiding purchase of a (possibly) expensive lisence for Visual Studio to in order to use free software might be a lucrative alternative. There are internet sites describing how to work around the Visual Studio obstacle if you need to build the extension module manually, by using free, open source compilers. You could for example use *MinGW*, which is a variant of the *gcc* compiler. `http://sebsauvage.net/python/mingw.html` seems to offer a good description of such an alternative.

The disadvantage of using the `.exe` build of a module is that it only modifies the *release* build of your Python library installation. The result is that you cannot use the new extension module with a debug version of your own extension module during your development phase. If the `.exe` build is not provided, or you need access to the debug version of the module, you will need to use the Source `tar.gz` or Source `.zip` distributions. You then run the `setup.py` script to build and install the package, like you would in a Linux environment. Again, this requires a compiler installed on your system. See chapter 7.1.3 for further details on building and installing Python modules through `setup.py`.

If you are using the Visual Studio C++ compiler as the default on your system when building and installing Source distributions, you must make sure that the that the Visual Studio IDE is run on your system *at least once* after installation before you run `setup.py`. This is because certain entries in the Registry are not set until the IDE is run, your system is prevented from finding the compiler and linker, and thus cause `setup.py` to fail.

7.1.3 The Numeric (NumPy) module

Numerical Python (Numpy) adds a fast multidimensional array facility to Python, and allows Python programmers to efficiently manipulate large sets of objects organized in grid-like fashion [From The Numeric Manual].

The NumPy module comes in several different builds, which can be downloaded from `www.sourceforge.net`. However, if you want to have the debug build available on your system, you must build and install the package through the provided `setup.py` script. For Windows builds you can use either the `.zip` or `.tar.gz` bundles. The `README`s provided with the releases claims Window users should use `.zip` sources when installing on a Windows platform. However, it does not seem that `.zip` builds are released quite so frequently as the `.exe` and `.tar.gz` types. We have successfully built and installed Source `.tar.gz` releases on Windows, and there seems to be no problem, nor any discernible difference in contents to the `.zip` distributions. The reason the `.zip` type is recommended for Windows could be because `.zip` archives are more widely used on Windows systems compared to the `.tar.gz` format. Windows has built–in support for the zip–format, whereas programs for handling `tar.gz` packaged files must be downloaded separately . However, shareware programs like *WinRAR* handles `.tar.gz` just as well as `.zip` archives.

To install the debug version of Numeric together with the standard release version, unzip the Source to some temporary directory, and through `cmd` run `setup.py`, first with arguments `build -g` and then `install`, i.e enter `setup.py build -g | setup.py install` on the command line. The script will provide more information on how to run the setup by using command line argument `-h`.

7.1.4 The Python Sparse Matrix (Pysparse) package

The Pysparse module extends Python by a set of sparse matrix types, iterative solvers for systems of linear equations, standard preconditioners and an interface to a direct solver for sparse linear systems of equations (SuperLU). As with Numeric, you can download Pysparse from `www.sourceforge.net`. The author of Pysparse also provides a site describing the module at `http://people.web.psi.ch/geus/pyfemax/index.html`. The module is only offered as a Source `.tar.gz` type.

Pysparse depends on Numeric in addition to the LINPACK and BLAS libraries, so you must install these prior to installing Pysparse. How to install Numeric was described in 7.1.3 on the preceding page. The installation notes for Pysparse suggests using the LINPACK and BLAS implementation that accompanies the Intel Math Kernel Library (MKL), which is what the author of Pysparse used under testing of the package on the Windows platform. MKL is a commercial product, and is offered as a free, non-commercial download for the Linux platform only. They do however have a evaluation copy available for download for Windows users. Once you have installed Numeric and MKL you must modify the `setup.py` script to match the setup on your system. Use the setup under "Rivendell" as a blueprint for a Windows system. You must also make sure that the path to the MKL DLL directory, i.e `C:\...\MKL\ia32\bin`, is in your PATH environment variable to avoid runtime errors when importing Pysparse modules. The MKL installer offers to do this, which saves you from editing it yourself.

Originally, version 0.33.029 of Pysparse was used for this thesis. In this early setup script, the gcc math library "`m`" was included in the list of libraries the module should link with. This library does not exist on Windows and running the script as it will give linking errors due to non-existent library "`m.lib`". The necessary math library is however included by default by Visual C++ compiler, so it is safe to remove this reference in the setup script. In later releases of Pysparse, like version 0.34.032, the reference to the math library has been removed. If this is not the case with the version you are using, you you can add the following code after the default settings: will give linking errors due to non-existent library "`m.lib`". The necessary math library is however included by default by Visual C++ compiler, so it is safe to remove this reference in the setup script. In later releases of Pysparse, like version 0.34.032, the reference to the math library has been removed. If this is not the case with the version you are using, you you can add the following code after the default settings:
or just remove the reference all together if you are not concerned about generality.

```
# m.lib does not exist on Windows,
# and math is automatically included
if sys.platform in ['win32']:
    if 'm' in umfpack_libraries:
        del umfpack_libraries[umfpack_libraries.index('m')]
```

7.2 Diffpack in Visual Studio 6.0 projects

Diffpack for Windows sets up all necessary environment variables during installation, assuming you have installed Visual Studio 6.0 prior to installing Diffpack. If this is not the case, you can run the `DpSetup.bat` file located in your Diffpack folder after you have installed Visual Studio 6.0. The `ReadMe.html` in the root directory of your Diffpack installation, i.e the path pointed to by the `NOR` system variable, together with the section *Diffpack Topics* in [8, B.2.2], describe the appropriate settings for projects of type *Win32 Console Application* or *Win32 Application*. This section briefly summarizes the settings that apply when using Diffpack in a *Win32 Dynamic-Link Library* project (DLL project), which is what you will be working with when creating Python extensions, as described in chapter 7.3 on the facing page.

After you have created an empty DLL project, you must modify the project settings. As you will not be interested in producing a graphical user interface (GUI), the modifications largely follow the settings for *Win32 Console Application*s. If you right-click on your Diffpack project and choose settings, or go to *Project→Settings* (hot keys Alt+F7), you open the settings dialogue for the projects in the current Workspace. Select *All Configurations* in the *Settings for:* drop down menu, and under the *C/C++* tab, add the preprocessor directive `NUMT=double` to indicate that the Diffpack type `real` should be interpreted as a `double` in your program. To set up your debugging environment, select *Win32 Debug* under *Settings for:*, choose category *Code Generation* under the *C/C++* tab, and select *Debug Multithreaded DLL* as the run-time library. For the *Win32 Release* version you should use *Multithreaded DLL*.

In addition to this, you must include the header file `LibsDP.h` in your source code to give instructions for the linking process of the project. The documentation for *Console Application* setup states that it is sufficient to only include `LibsDP.h` in the source file containing the `main`(···) function. When creating Python extensions the equivalent would be to include it in the source file implementing the module's `init` function. If you are using SWIG, it would make sense to let the interface file that defines the initialization block (`%init{`···`}`) include `LibsDP.h`.

Diffpack Mini does not have a debug build. You will therefore have linking trouble when `_DEBUG` is defined. A quick fix is just to temporarily turn off the flag while including `LibsDP.h`. To make a slightly more general solution, and make the source code more portable, a header file, `Win32DpLink.h` was written that handles the `LibsDP.h` logic under

Windows. If Diffpack Mini is used, all linking trouble is resolved by defining the preprocessor macro `DP_MINI=1` and including `Win32DpLink.h`.

The steps described in this section is essentially all you need to set up a Diffpack project from scratch, as long as you are not interested in creating GUI interfaces. All other preprocessor macros mentioned in the documentation are automatically set when you create your project, except the `NUMT` macro, which we have dealt with here.

7.3 Creating Python Extension Modules in Windows using Visual Studio 6.0

The common approach when writing extension modules for Python is to compile your C/C++ code into a *Dynamic Linked Library* (DLL) file that can be loaded into Python as a module. There are two common approaches to achieve this: you can either manually write the necessary *wrapper code* between your C/C++ code and Python by using the *Python C API*, or you can use *SWIG* to automatically create the wrapper code through processing interface files. Either way you will have to create a *Win32 Dynamic-Link Library* project. This chapter does not aim to describe the details of wrapper code generation for Python extension modules, only how to handle it through Visual Studio 6.0.

If you are planning to create several projects accessing the Python C API, you might want to add the Python include and library paths permanently to your `include` and `lib` environment variables, so you do not have to set them specifically for every project you create. You can either do this the standard way by manipulating your environment variables through the *System Properties* dialogue, or you can do it through the Visual Studio IDE. In Visual Studio, you go to *Tools* → *Options* → *Directories* and you add the Python include path and library path under *Include files* and *Library files*, respectively. Typically, the paths will be something like `C:\Python23\include` and `C:\Python23\libs`. If you do not want to modify your include and library paths, you must specify these paths as extra include and library paths through your Project dialogue for *every project* you use the Python C API. To do this you choose category *Preprocessor* under the *C/C++* tab in your project settings and enter the extra include directory. Under the *Link* tab choose the *Input* category and enter the additional library path. See chapter 7.2 on the preceding page for details on adjusting your project settings.

7.3.1 Visual Studio and manually created wrapper code

When you create the wrapper code for your extension module manually, there are only a few settings that must be added to your *DLL* project in order to produce a working Python extension module. As when setting up a Diffpack DLL project, you must set the *Code Generation:* to *Debug Multithreaded DLL* and *Multithreaded DLL* under the settings for

Win32 Debug and *Win32 Release* mode, respectively. See chapter 7.2 on page 92 for further details. Under the *Link* tab in your project settings dialogue, set the *Output file name:* to `moduleName_d.dll` in Debug mode, and `moduleName.dll` in Release mode. The naming convention is necessary in order for `python` or `python_d` to find and load your module. Assuming that you have your wrapper code as part of your project files, add a new project file to your project and call it `moduleName.def`, i.e the same base name as the DLL output file. The only entry you need in this file is "`EXPORTS initmymoduleName`", which states that the `init` function of your module should be made accessible outside the module. This is necessary in order for the Python interpreter to have access to it when you import it into the Python interpreter.

7.3.2 SWIG in Visual Studio 6.0 projects

The Simple Wrapper and Interface Generator (SWIG) is a development tool for building scripting language interfaces to C/C++ programs. As a Python extension module developer you only need to write a simple interface file specifying how your C/C++ code should interact with Python instead of a large amount of tedious wrapper code. SWIG will interpret the interface file, and use it to automatically generate the wrapper code. You have to include the interface files as part of your Visual Studio project. By setting up your project properly, you can make Visual Studio call SWIG with the appropriate arguments to generate the wrapper code, include the newly generated wrapper code in the compilation process, and link your source code and the wrapper code into a library file in one smooth operation. Thus, with a single command, *Build*, you generate the wrapper code, compile, link and build your module, instead of having to go through several steps manually every time you need to build your module for testing in Python.

The first step is to make sure SWIG is in your path of executable programs. The process is equivalent to registering the include and library paths of your Python installation, described in chapter 7.3 on the previous page. As before, you can register SWIG as an executable through either the *System Properties* dialogue or using the Visual Studio IDE. In the IDE you enter the path to the SWIG executable in the list under *Executable files*, causing the system variable `PATH` to be modified.

The manual that ships with SWIG is extensive and will probably cover everything you need to know when using SWIG. Chapter 26, *SWIG and Python*, is particularly useful when working with SWIG to create the wrapper files for a Python extension module. The current section is largely based on chapter *26.2.9 Building Python Extensions under Windows*. The information is included here to give a complete reference to the tools you are most likely work with.

As mentioned earlier, you must add SWIG interface files to your project. You should also add a reference in your project to a (possibly non exiting) file in the root folder of your project. You should follow SWIG's naming convention in order for this to work, i.e add a reference to `myModule_wrap.c` if you are using C, or `myModule_wrap.cxx` if your module is written in C++. The file will probably not exist at this point, but Visual Studio will still keep a reference to it after issuing a warning. After SWIG has executed, the file will contain the generated wrapper code, and Visual Studio will include it in the build process. In the Project Settings dialogue, select the interface file SWIG should use, and open the *Custom Build* tab. Choose *All Configurations* to fill in for both the *Win32 Debug* and *Win32 Release* modes in one step. In the *Description* field enter "SWIG". Depending on whether you are using C or C++, enter the following in the *Commands* and *Outputs* fields:

Field: *Commands*

```
swig -python -o $(ProjDir)$(InputName)_wrap.c $(InputPath)
swig -python -c++ -o $(ProjDir)$(InputName)_wrap.cxx $(InputPath)
```

Field: *Outputs*

```
$(ProjDir)$(InputName)_wrap.c
$(ProjDir)$(InputName)_wrap.cxx
```

There are also some settings for the project itself that must be set for SWIG to work properly. The output file under *Link* must be set to combine the name of your Python module and the naming conventions SWIG expects of the DLL output files, i.e `_moduleName_d.dll` for the Debug build and `_moduleName.dll` for the Release build. These must of course match the module name given in your SWIG interface file. Note that the output files should *not* go int the Debug and Release directories. You must also add the `__WIN32__` preprocessor definition under the *C/C++* tab. Depending on whether you have registered the Python include and library paths in your environment variables or not, you might also need to set these for your project before you can successfully build your solution. See chapter 7.3 on page 93 for further details.

Diffpack ships with a set of Perl scripts for facilitating the creation of SWIG interfaces for Diffpack code – `MkDpSWIGMakefile` and `MkDpSWIGInterface`. See [11] for details on creating Python SWIG interfaces for Diffpack programs. These scripts are based on a GNU environment relying on GNUMake, GCC and the typical project structure of a Diffpack project on a Linux environment to function properly. Microsoft users will either have to download something like MinGW (Minimalist GNU for Windows) or just transfer the source files to a Linux host in order to run the scripts.

You should now be able to combine Python, Diffpack and SWIG in order to more efficiently develop Diffpack extension modules for Python.

7.3.3 Debugging Python Module Extensions

When you are working with Python extensions, your source code is located in a DLL library and called from some Python script. This prevents you from using the standard debugging procedure of your C/C++ code in Visual Studio (i.e by inserting a breakpoint and run the debugger).

To debug your extension code, you would typically use a debug version of the Python interpreter, like the `python_d.exe` mentioned in chapter 7.1.1. You will need to attach the `python_d` process to a debugging environment. Here you have two options: either attach `python_d` to the debugger *before* running the script, or enable *Just-in-time debugging* (JITD) and let the process attach itself once a breakpoint is reached. In the latter case, `python_d.exe` will report that it has encountered a problem or a fatal error when it reaches a breakpoint in the code. If you choose the debug option in the ensuing warning dialogue, it activates the debugger and positions you at the breakpoint that triggered the "fatal error". JITD is a nice feature to enable even though you have not inserted any breakpoints, because you can then choose to debug whenever your module crashes, which will hopefully giving you an idea as to where the program fails and why. To enable JITD, go to *Tools* → *Options* → *Debug* in Visual Studio, and tick *Just-in-time debugging*.

If you would rather debug using the first option, you start `python_d.exe`, and before you execute your script, in Visual Studio go to *Build* → *Start Debug* → *Attach to Process* and select `python_d`. When you run your script, the debugger will halt execution at the first breakpoint.

When debugging, you usually insert a breakpoint in your code by either hitting *F9* or using the menu in the IDE. When debugging through an attached process like `python_d` or by JITD, these breakpoints are no longer visible (to the debugger). You can hard code a breakpoint in your code that will still be visible however, by entering `_asm int 3;` wherever you would like your code to have a breakpoint. Note that this breakpoint will *always* trigger, even if you are running in release mode, though in release mode it will only cause the program to break in assembly code. `_asm int 3` causes the program to break in the source code rather than the kernel code (on Intel platforms), which is what we want it to do.

Hard coding breakpoints can be tedious, and if you forget to remove them, your program will seem to crash. This is particularly true if you are running a release build and cannot see that it was merely a forgotten breakpoint that caused your program to fail. However, once you have attached the `python_d` process to a debugger, you can open the source file you with to debug in the Visual Studio environment and use F9 to debug as usual.

If you use Komodo for developing Python code, you can choose which Python interpreter it should run as default in the *Preferences* dialogue. Setting it to `python_d.exe` you can do a full, continuous debug session of both your Python code and C/C++ code. The same principles for attaching `python_d` to the Visual Debugger apply as described above. In other words, you use `python_d.exe` as the interpreter in Komodo, and run your script to some breakpoint *in your script* set by Komodo. When you reach the breakpoint, you can attach the `python_d` process to the Visual Studio debugger. Now, breakpoints in your C/C++ level code will trigger as you step through your script.

7.4 Summary

In this chapter we have described all the principal tools used to develop the extension modules necessary to turn Diffpack into a linear system generator for Python. The chapter was the outcome of deciding to combine Python and Diffpack to produce simulators that could be used in combination with a Python implementation of the Parareal algorithm. In the following chapter we will present the reasons for using such a cross–language combination, and outline the results.

All the tools and techniques described in this chapter were used during the development phase of this thesis. By succeeding in creating a complete Python distribution module, we have illustrated that Windows may very well be used for scientific computing.

Chapter 8

Combining Python, SWIG and Diffpack

In this thesis we opted for using a combination of Python and Diffpack, which also introduced SWIG. All governing code such as the Parareal algorithm, handling of time iterations and solving the linear system would be written in Python. Diffpack would merely be seen as a linear system generator and a *possible* source for iterative solvers. In this section we will present our arguments for using such a combination, and our experiences from structuring the solution strategy to our problem in this way.

8.1 Why use Python and Diffpack?

The decision to use a combination of Python and Diffpack for this thesis originates in several factors. It is particularly linked to the nature of the Parareal algorithm, and the encouragement of the supervisors of this thesis.

In itself, the Parareal algorithm is completely generic. It can work with fine and coarse solvers that employ different temporal discretization techniques, the coarse propagator could use a coarser or different spatial grid, or one could let it solve a different (simpler) PDE altogether. Neither does it matter how the λ–values are defined. The only request from the algorithm is that the objects it uses satisfies a minimal interface, like requiring that the sum over the solution objects exist. Such flexibility is naturally reflected in Python, which is an object oriented, typeless scripting language. As you do not have to make the algorithm conscious of any kind of variable typing, it will also be possible to keep the implementation as clean and as close to its definition as desired. In Python, you would not have to construct a hierarchy of formal interfaces and descendants to reflect the generality of the algorithm, which you would have to do for a strongly typed language like C++. The Python version would just assume that certain properties were fulfilled by the objects involved, and let the caller ensure that all necessary operations are well defined. There would be no need to inherit from predefined interfaces.

Due to the generality of Python, one would be free to utilize Python, C/C++ or Fortran libraries, depending on ones preferences, to implement the actual solver for each time step.

The algorithm is designed to be a *governing* algorithm. Its essence is to initiate a coarse estimate of the λ–values, pass these values out as initial conditions to N fine solvers working in parallel, and then handle the results. The governing algorithm is not compelled to be efficient – it is the spatial solvers in $\mathcal{G}$ and $\mathcal{F}$ that require the carefully considered algorithms in terms of speed, accuracy and memory utilization. This structure coincides with one of the more traditional uses for scripting languages, which is to create governing programs that call other, more complex modules in order to perform a given task. Speed is usually not a crucial factor for scripting languages, as the efficiency of a governing process is not important in the overall computation time. As Python is an interpreted language its far less efficient than compiled languages like C or C++.

Yet another reason for using Python was the possibility that the algorithm would be ported to a true parallel platform, although this was by no means a certainty at the time when we decided to use Python. A quick study indicated that Python has a nice, clean interface to the MPI (message passing interface) standard for parallel programming. It was hoped that at a possible, future porting to MPI, it would be an advantage to have the algorithm written in Python and not in some comparatively low–level language like C++.

In addition, one has all the usual advantages of using Python compared to C++. There is the automatic garbage collection, you have increased expressiveness per statement, and a rich default library. Python a popular choice in the scientific community, and several scientific libraries provide Python interfaces, such as the AMG (algebraic multigrid) and Pysparse modules used in this thesis. This could give a richer pool of possible solvers to seamlessly plug into the algorithm.

8.2 Initial implementation

After initial research and once we had gained a good understanding of the various modules and tools suggested for implementing the framework around the Parareal algorithm, we managed to develop a successful implementation combining Parareal and the heat and Stokes solvers. Several successful tests were run using the initial implementation of the algorithm. Toward the end, there was a need to extend the implementation to support further tests. During this process a bug was introduced that prevented us from completing the remaining tests until it was resolved. The bug would become evident through a series of seemingly mysterious and fatal runtime errors within Diffpack.

Somewhere in the process of letting Python administrate the tests and calling our extension modules, which again use the Diffpack library, the simulation would randomly crash. There were situations where small, seemingly Diffpack unrelated changes in the Python script would cause previously functioning Diffpack code to break within its library. The bug would always become evident through calls to Diffpack.

The randomness of when the program would crash, and how adding or removing code at unrelated places would affect the error, could indicate that memory was being corrupted, either by being overwritten or through resource management conflicts with Python.

In hindsight another cause for the error could be how each of the three separate extension modules developed (Diffpack2Python, Heat1, StokesSolvers) all initialize Diffpack separately. Although no documentation was found that discourages this practice, it could induce communication between modules to fail, depending on how Diffpack is implemented.

8.3 Debugging

Debugging across programming languages made locating the bug difficult. The many programming languages (C++, Python), libraries (Diffpack, Pysparse, ML) and tools (SWIG) required to develop the initial implementation strategy made it impossible within the available time to gain expert knowledge within any of these areas. An example is the SWIG layer between the Python code and our Diffpack simulator that introduces uncertainty when one does not have complete understanding of how SWIG generates the the wrapper code.

None of our debugging tools were sufficient to resolve the type of bug we were experiencing. The fact that Diffpack was only available to us as a closed library prevented our tools from tracing the bug back to its origin. One has to take into consideration that the error could originate in any of the many layers of languages and libraries used. This made it impossible to systematically track and eliminate the error.

8.4 Resolving the problem

The initial implementation strategy and open design is complex, and makes debugging the kind of error we were experiencing extremely challenging. We eventually decided that no more time would be spent on investigating the source of the error, and that focus should be kept on Parareal and Stokes. As a consequence, the decision was made to simplify the approach used to implement Parareal, in order for testing to proceed. The decision was made for two reasons:

- The remaining project schedule did not allow for extensive time to debug and correct the bug. It was concluded that the remaining test results were of greater importance to the completion of this thesis than fulfilling the project with the initial implementation strategy. In terms of studying the performance of Parareal on the Stokes equations, the choice of implementation strategy is invariant.

- The initial implementation strategy had already proved successful with the first set of tests. We were able to verify that one can indeed use Python as the central solver and only see Diffpack generated matrices and vectors as modules that merely plug into the overall solution. It was therefore not deemed necessary to complete the project with

> the current implementation strategy. There is no reason why the initial implementation should not work with the final set of tests, it is merely a question of development time.

All existing code was thus ported to a pure C++ solution, which would simplify and reduce the possibilities for errors to occur. Once the implementation was successfully ported, the bug no longer appeared.

The knowledge gained from developing the initial solution made rewriting and simplifying the implementation to a single–language solution quick and easy. The process of rewriting the code to C++ was of markedly shorter duration than the time already spent on trying to locate the bug. Both implementations are available for download at the author's website as described in chapter 1 on page 1.

Chapter 9

Conclusions and future work

We will draw some conclusions primarily based on the Parareal test results in chapter 6 on page 70 and on the experience from using a cross–language implementation strategy to implement the framework around using Parareal. We will also point out some unresolved issues that should receive further attention.

9.1 Parareal

All conclusions for the Parareal algorithm are of numerical nature only, and can merely be viewed as guidelines to the behaviour of the algorithm. For the Parareal algorithm, we investigated its stability and the quality of the chosen stop criteria.

9.1.1 Stability

The original instigator to this thesis was to investigate the relation between the unsteady Stokes equations and the Parareal algorithm. It was believed that the algorithm would be able to handle such a problem, but there was a sufficient amount of uncertainty associated with the assumption to justify a numeric investigation. It was found that a model problem shows its amenability to computation by Parareal through its influence on the convergence rate and stability of the Parareal algorithm.

Particularly, if Parareal is used to solve a system of stiff ODEs, the stability function of the Parareal algorithm can be expressed through a dependency on the stability function for the *coarse* propagator. This was discussed in section 5.1.3. Because the Stokes and heat equations are parabolic PDEs, they will transform into systems of stiff ODEs when semi–discretized in space. The bounds on the stability function for the coarse solver in order to maintain Parareal stability was discussed in sections 2.2.1 and 5.1.3, and we concluded that if θ–*rule* discretization is employed, the θ–parameter for the coarse propagator is bounded by

$$\theta \in [2/3, 1] .$$

Tests were performed in section 6.1 on page 70 that aimed to give an indication of the appropriateness of using Parareal to solve the Stokes equations. The convergence and stability traits displayed for Stokes were compared to the corresponding convergence and stability properties exhibited when the algorithm was used to solve the heat equation. Earlier studies of the algorithm verified that the heat equation is suitable for use with Parareal, and it would therefore give an appropriate *blue–print* of the expected behaviour. The tests were conducted by varying θ for the coarse propagator both inside and outside the bound set by the stability function.

Based on the behaviour displayed by the algorithm when run with the heat equation, it was found that the Parareal appears to follow the same stability pattern as we move outside $\theta \in [2/3, 1]$ as the heat equation. We therefore concluded that the algorithm was stable when run with the Stokes equations.

However, the numerical tests also indicate that the algorithm deviates slightly from its expected behaviour at the lower end of the bound on θ. For the employed test cases, it enters a fully exponential convergence rate somewhere in the range $\langle 2/3,\ 2/3+1/10]$. We proposed that the reason for this slight deviation in behaviour *may* be due to the Stokes equations classifying as a *differential algebraic equation* (DAE) when semi–discretized in space. This could possibly cause some of the stability restrictions on the coarse propagator to change. As the numerical oddities only appear at the lower bound of θ and only influence the convergence rate, we concluded that the numerical tests indicate that the algorithm is stable and convergent for the Stokes equations, but that further numerical analysis is necessary to determine the full impact of the Stokes equations on the algorithm.

We also observed that for the Stokes equations the convergence seemed to improve as $\theta \to 1$, which is the opposite behaviour of the heat equation, which shows increased convergence rate as $\theta \to 2/3$.

9.1.2 The algorithmic stop criteria

In section 5.1.2, a strategy for determining convergence based on the global space–time norm

$$e_p = \sqrt{\sum_{i=k}^{N} ||\lambda_i^k - \lambda_i^{k-1}||^2 \Delta t}$$

was proposed, as it is generally assumed to give a good reflection of the true error in the algorithm. In section 6.2 a numerical test was suggested to give an indication of the actual quality of e_p. a was defined as the ratio between e_p and the equivalent space–time norm, e_s, based on $||\lambda_i - \lambda_i^k||$. λ_i is the exact, serial solution. With this ratio it is possible to give an estimate of how one must scale the the stop–criteria, ε_p, used by the algorithm, to properly reflect the error compared to the exact solution one is willing to accept.

The numerical tests conducted indicates that for a particular execution of the algorithm, the ratio e_p/e_s appear constant until machine accuracy is reached. The conclusion based on these results is therefore that e_p accurately mirrors e_s, such that one can use $\varepsilon_p = a\varepsilon_s$ to have an error of ε_s in the solution produced by the algorithm. Further research should be done to verify that a does indeed exist for most or all model problems, and also to determine a good value for a. The tests done for this thesis produced a–values in the range $[2, 18]$.

9.2 Implementation strategy

The implementation strategy in this thesis was originally structured around using Diffpack, SWIG, Python and the Python module Pysparse in combination to solve the PDEs in question. This behaved as expected and can thus be seen as a proof–of–concept: one can indeed use Python as the solver and just see Diffpack as another module that can be plugged into the solver. In our case, we used Python, Diffpack, SWIG and Pysparse to build, precondition and solve the linear system at each time iteration.

9.3 Future work

9.3.1 Verifying the numerical tests

As has been pointed out, a deeper insight to the numerical tests performed on Parareal is in order. The minor stability issues around $\theta = 2/3$ for the Stokes equations must be investigated and analyzed. It would also be interesting to see whether the *increase* in the convergence rate as $\theta \to 1$ for Stokes has an analytical explanation, as it is the exact opposite behaviour to the heat equation.

The possibilities surrounding the algorithmic stop criteria should also be studied in order to be able to give better estimates of how one should set ε_p to reflect the true error in the solution.

9.3.2 Parallel implementation

The Parareal algorithm was implemented as a serial (and therefore highly inefficient) solver. The topmost priority was to check the Stokes equations' amenability to parallel computation through Parareal. To achieve this it is not necessary to actually implement the algorithm in parallel. By omitting the parallel factor, we would be able to keep the focus on the Stokes–Parareal combination and not on time consuming technicalities arising from a fully parallel implementation.

However, based on the conclusions in 9.1.1, it would be appropriate to view parallel implementation as the the next logical step in order to fully utilize the speedup possibilities. As the uncertainties at $\theta = 2/3$ for Stokes is likely to cause further testing, it will be an obvious advantage to be able to do the tests in parallel. The serial version is very slow to execute, which comes on top of the computationally intensive Stokes discretization. This turns testing into a very slow process.

Appendix A

The Python-Diffpack Interface

When using Diffpack from Python as a provider of services, such as creating linear systems, it is practical to have access to both a system for filtering from native Diffpack vectors and matrices to their Python equivalent, as well as Python interfaces to common Diffpack administrative classes, such as the MenuSystem class. `Diffpack2Pythn` is a Python extension module that was written in order facilitate the process of using Diffpack as a matrix and linear system provider. Its origin is based on the `DP` module written for [11]. The module offers two filter classes – one for filtering vectors and one for matrices – in addition to a small set of classes that provide a Python interface to some of the most commonly encountered classes when Diffpack is accessed from Python. The module is not constructed to handle the smart pointers in Diffpack (`Handle(Type)`). Python is based on references and it would be unnatural from a Python point of view to use such handles. The module contains the following classes:

Dp2Numeric	Diffpack to Numeric to Diffpack filter for vectors
Dp2Pysparse	Diffpack to Pysparse to Diffpack filter for matrices
MenuSystem	wraps the Diffpack class `MenuSystem`
Vec_double	wraps the Diffpack class `Vec_double`
TimePrm	wraps the Diffpack class `TimePrm` (handling time integration)
LinEqAdm	wraps the Diffpack class `LinEqAdm` (linear solvers access)
LinEqAdmFe	wraps the Diffpack class `LinEqAdmFe` (linear solvers access)

A.1 The `Dp2Numeric` filter class

`Dp2Numeric` is a filter class for converting between references to Diffpack `Vec_double` instances and one-dimensional instances of the Numeric class `array`. The class interface is simple and only has two groups of functions; one for converting from Diffpack to Python, and one for the reverse action.

py2dp(dpVec_double, numpyArray) fills the entries of the one-dimensional `Numeric` array `numpyArray` into `dpVec_double`, which should be a reference to a Diffpack `Vec_double` object. `Diffpack2Python` provides a wrapper class for the `Vec_double` which can be used to manipulate the Diffpack objects from Python.

py2dp(numpyArray) as `py2dp(dpVec_double, numpyArray)`, but will create and return a *new* `Vec_double` instance and return the C++ pointer to Python. The wrapper code does not keep track of this newly created object, and you should probably use the functionality offered by SWIG for flagging objects as the responsibility of the Python garbage collection. See chapter A.3 on the facing page.

dp2py(dpVec_double) creates and returns an instance of a one-dimensional `Numeric array` filled with the values from `dpVec_double` which should be an instance of `Vec_double`.

A.2 The `Dp2Pysparse` filter class

The `Dp2Pysparse` does conversion between Diffpack and Pysparse, with much of the same interface as `Dp2Numeric`. It accepts and returns `LLMatType` objects, the basic matrix in the `spmatrix` module from the `Pysparse` package. Even though the filter interface accepts the base class `Matrix_double`, the it only supports the subclasses `MatSparse_double`, `MatBand_double` and `MatDiag_double` when filtering from Diffpack to Python. This is due to the design of Diffpack, where the subclasses of `Matrix_double` have different schemes for efficiently accessing its non-zero values. The matrix implementations will issue warnings when non-zero values are read unless the “`--nowarnings`” directive is applied to Diffpack. The Diffpack simulator will terminate if too many warnings are raised. To avoid this one must provide separate filter functions in the C++ implementation of `Dp2Pysparse` for each type of matrix. Therefore, the filter class does not support any arbitrary subclass of `Matrix_double`. The class interface accepts the common base class `Matrix_double`, and will automatically determine the type of subclass passed. Extending the filter to support other subclasses of `Matrix_double` in the future should thus be easy, and it will merge seamlessly with existing code. The Python interface has one function for translating from Diffpack to Pysparse, and two functions for the reverse action.

py2dp(dpMatrix_double, llMatObject) Accepts a reference to an existing `Matrix_double` instance , clears the previous contents, and fills it with the the non-zero entries of `llMatObject`. `llMatObject` should be of type `LLMatType` defined in `spmatrix`. Because the same sparsity pattern is assumed in `dpMatrix_double` and `llMatObject`, any subclass of `Matrix_double` *should* work as long as this assumption holds true. If the sparsity pattern between the entries in `llMatObject` and `dpMatrix_double` does not match, Diffpack will issue warnings for attempting to write to entries outside the pattern, and will eventually shut down.

py2dp(llMatObject) Creates a *new* `MatSparse_double` object, which is returned as a C++ pointer. The matrix is filled with the non-zero values of `llMatObject`. The same caution as was mentioned for the equivalent `Diffpack2Python.py2dp` must be shown here.

dp2py(dpMatrix_double) Creates and returns an object of type `LLMatType` filled with the non-zero values in the input matrix. The input matrix should be a reference to one of the subclasses `MatSparse_double`, `MatBand_double` or `MatDiag_double`.

A.3 The Python wrapped Diffpack classes

The module also provides Python wraps of several Diffpack classes. As all of these classes have been wrapped using SWIG, each class has a an associated wrapper class for C++ pointers, which has identical identical interface to the standard class. For each class you have a `classNamePtr` class whose constructor accepts a C++ pointer to an instance of the native C++ class. If, for example, a `Vec_double` pointer was returned from Diffpack, it could be accessed as any normal object in Python by wrapping it in `Vec_doublePtr`.

Diffpack2Python module itself offers access to the the global variables for standard in, out and error in Diffpack, i.e `s_i`, `s_o` and `s_e`, plus access to the `global_menu` instance. The standard input/output/error objects can be accessed as usual through `s_i`, `s_o` and `s_e`, and hold pure C++ references that can, for example, be sent to all the native Diffpack print functions that require `Is` or `Os`. `global_menu` is pre-wrapped in a Python `MenuSystem` interface.

MenuSystem Python interface to Diffpack `MenuSystem` class. The class is provided to give an easy way to manipulate the initialization phase of your Diffpack simulators interfaced from Python, and gives you the possibility to utilize Diffpack's menu system through your Python script. The interface was for example used to manipulate the settings for the simulators used during this thesis. The manual for the class can be found in the Diffpack documentation.

Vec_double Python interface to Diffpack `Vec_double` class. `__add__(self,other)`, `__sub__(self,other)` and `__copy__(self)` has been added to match the interface requirements of the Parareal implementation. See 5.2 on page 63.

TimePrm wraps the equivalent Diffpack class, which offers functionality for administrating time integration. For any unsteady solver this interface should be useful as it gives easy access to all the current time information in the simulator.

LinEqAdm and LinEqAdmFe wraps the Diffpack linear system managers to provide easy access to the different linear solvers defined in Diffpack through manipulating its menu settings.

The public methods and attributes of the underlying Diffpack class are available for all wrapped classes, and refer to the Diffpack manual for documentation of these functions.

As the classes in this extension module are SWIG generated, they all have `thisown` attribute that can be used to flag whether the object should be garbage collected by Python or not. Setting the flag to true will cause Python's resource manager to remove the object when no further Python objects refer to it. This is useful when using the filter class functions that create new Diffpack instances, but care must be taken to avoid resource conflicts with Diffpack. Note that the `ClassNamePtr` classes set the `thisown` flag to false by default, and you must specify if Python is to be responsible for the new object. More information on this can be found in the SWIG manual for Python.

We conclude by remarking that using statements like **from** Diffpack2Python **import**$*$ seem to cause Diffpack to fail, whereas **import** Diffpack2Python does not seem to cause these errors. This is possibly related to the problems described in chapter 8 on page 98.

Appendix B

The pararealStokes package

The `pararealStokes` is a Python distribution packed with `distutils`. It contains the extension modules with the Diffpack simulators for the heat and Stokes equations, Pure Python modules and packages that utilize or inherit from the classes in the extension modules. The package also provides some utility scripts for administering the execution of the Parareal algorithm with either heat or Stokes and a set of scripts for plotting test results returned from the Parareal algorithm.

Due to the trouble in combining Diffpack and Python as described in chapter 8 on page 98, the `pararealStokes` package is incomplete with regards to the tests done in chapter 6 on page 70. All the tests done on combining Parareal and Stokes was done in the pure C++ implementation of the Python scripts and modules in this distribution. The distribution is available for download at `http://heim.ifi.uio.no/~erical/masterThesis/`, together with its equivalent pure C++ version. A short review of the central classes will be provided here, and we refer to the documentation accompanying the package for further details.

The only test case from chapter 6 supported by the Python scripts is the error convergence test – for any other test result refer to the C++ implementation. When running the installation, the extension modules, Pure Python modules and packages will be installed to your default installation directory for Python modules, or to another specified directory. The scripts contained in the Script directory will, however, not be installed.

B.1 Python extension modules

`pararealStokes` provides three extension modules:

- `Diffpack2Numeric`
- `Heat1`
- `StokesSolvers`

Diffpack2Numeric implements filtering and wrapping functionality between objects native to Diffpack and to Python. The documentation of the module can be found in chapter A on page 107. The remaining modules will be discussed here.

B.1.1 Heat1

Heat1 implements the Diffpack simulator for heat in accordance with the discretizations presented in chapter 3 on page 15. The only class made available through the module is Heat1, although there is also a functor class for providing the exact solution. The class supports the Parareal interface requirements.

B.1.2 StokesSolvers

The extension module encompasses the classes listed in Figure 4.1 on page 49. All the Stokes solver classes and preconditioners are available in Python, as they have corresponding SWIG interface files. The central classes used by the Pure Python modules for this thesis are StokesTime implemented in StokesTime.cpp and StokesTime.hpp, and Pressure-PrecTime implemented in PressurePrec.cpp and PressurePrec.hpp. These two classes have been extended such that they can be used as a linear system provider and preconditioner for the pressure component.

The PressurePrecTime class merely makes its mass and stiffness matrices available, whereas the StokesTime class has a fairly extended interface compared to standard Diffpack simulators. The extension to the class is done directly in the class definition and not through the SWIG interface file, which was the principal way of extending the class interface for the heat equation. It was not done for the Stokes equations, due to the greater complexity of the internal block structuring. As Python has no concept of the Diffpack block structure, the class must be structured in such a way as makes the block structure invisible to the callee. The block structure must be stretched out before it can be returned, and in a likewise manner, it must be mapped onto a block structure when a continuous vector is sent into the simulator. The interface StokesTime offers as a linear system provider is detailed in Listing B.1.

Listing B.1: Excerpt from StokesTime covering the functions offered as a linear system provider

```
    bool                        linSysInitialized();
    bool                        linPrecInitialized();
    void                        buildLinearSystem();
    void                        refreshLinearSystem();
    void                        buildPreconditioners();
    const  Matrix(NUMT)&        getMatrix();
    const  Vec(NUMT)&           getRHS();
    const  Vec(NUMT)&           getSolution();
```

```
TimePrm&                getTimePrm();
void                    setCurrSolFromVec(const Vec(NUMT)& vec);
void                    shiftSolution();
void                    fillEssBC(Vec(NUMT)& contVec);
PressurePrecTime*       getPressurePrec();
Matrix(NUMT)*           getVelocityPrec();

void assmebleBlockVector( LinEqVector& blockVec,
                          Vec(NUMT)& vec );

Vec(NUMT)*          getVelocitySolution();
Vec(NUMT)*          getPressureSolution();
```

B.2 Pure Python packages and modules

The pure Python implementation can be found in the `pyModules` directory. The Python classes will either inherit directly from the classes in the extension modules, or use them as linear system providers. The Python implementation of the heat and Stokes solvers in Python were covered in chapter 3.4 on page 23 and and in 4.4 on page 46. The python modules their associated factory–and menu classes have been gathered in two separate package structures; the `Heat` package and the `Stokes` package. In addition, the modules `Parareal.py`, `VecWraps.py` and `NumWrap.py` are defined. The `Parareal` class is described in chapter 5.2 on page 63. The remaining modules, `VecWraps.py` and `NumWrap.py`, are helper classes for handling the transition from Diffpack vectors to Python representation.

`VecWraps.py` contains two alternative wrapper classes for the `Vector_double` class: `VecWrap` and `VecExposeWrap`. Both classes support the Parareal interface requirements. Their main intent is to facilitate Python garbage collection, by properly setting the “thisown” flag on newly created objects when for example summation or copy operations are performed. `vecExposeWrap` is only used with the Stokes equations, and creates support for only calculating the norm of either the velocity or the pressure component of the solution when the `norm()` function is called from Parareal.

`NumWrap.py` contains a wrapper class `NumWrap` that works around the fact that it is not possible to inherit from Numeric arrays. As the existing interface of Numeric arrays do not conform to our Parareal implementation, we handle this by creating a wrapper class whose sole member is a Numeric `array` object, and define the necessary functions, such as `__add__`, `__copy__` etc. Numeric arrays are used as the vector class in the Pysparse package, and by using this wrapper code one easily facilitates a combination of Parareal and Pysparse, where Diffpack is purely a linear system generator.

B.3 Python scripts

pararealStokes contains several scripts, which, for instance, run the Parareal algorithm on either the heat equation or on the Stokes equations. The scripts will not be installed to your Python package depository, but they do assume that the extension and pure Python modules have been installed to your Python module installation directory (i.e. that they exist in one of the directories pointed to by the PYTHONPATH environment variable).

There are two main groups of scripts in the pararealStokes distribution: those that run the Parareal algorithm on either the heat or the Stokes equations, and those who plot the results produced by running the Parareal algorithm. runPararealHeat1.py and runPararealStokes.py have a rich command line interface that states the possibilities of the scripts, but they are mainly intended to run the Parareal algorithm to produce error convergence data that will be written to data files. One can then run plotPararealError.py to get logarithmic plots of the error. This is parallel to what was done during testing of the stability of the Parareal when it was run on the Stokes equations. The plotting scripts require Matlab and the Python Matlab extension module, pymat, to be installed. For further requirements, we refer to the source code distribution.

The command line functionality of runPararealHeat1 and runPararealStokes is maintained by the module cmdLineParsers. Strictly speaking, it is not a script and should reside in the pyModules directory, but as its sole purpose is to handle the command line options for the respective run–scripts it was not deemed necessary to include it with the installed modules.

Appendix C

The Python interface to ML

For the work on this thesis the author of the `pysparse` package for Python, Mr. Roman Geus, kindly made his multigrid extension module `ml` available for use with `pysparse`. `ml` is a Python extension that wraps the *ML* package distributed through *Sandia National Laboratories*. *ML* is a C library that provides multigrid preconditioning for linear solvers. The package can also be used as an independent solver of linear systems on the form $Ax = b$. It uses Algebraic Multigrid (AMG), which makes it more efficient on large distributed systems than multigrid solvers that must construct a grid hierarchy before running the algorithm. *ML* is intended for use on large sparse linear systems arising from PDE discretizations, and does not perform well on small grids. *ML* naturally works well on systems that are well suited for use with multigrid, such as those based on *elliptic* PDEs [18].

The Python extension of *ML* is intended for use with the `pysparse` package. It expects the input matrix A to be a `ll_mat` object defined in the `spmatrix` module, and it implements the preconditioner interface expected by the iterative solvers in `itsolvers`. The following documentation is extracted from the *doc strings* associated with the different objects and functions in the `ml` module, and is included here mainly to gather the information in one single document.

The basic object in the module is the AMG solver object `AMGObject`. `ml.setup(···)` creates an `AMGObject` instance, and all further interaction with *ML* is done through this object. The moule had only been tested and used on Linux by the author, and minor modifications were required in order for it to run on Windows. The `changes.txt` file sent to Roman Geus to notify him of the changes are available with the rest of the source code for this project, at `http://heim.ifi.uio.no/~erical/masterThesis/`

Creating an AMGObject The following is reported by the `ml.setup` doc string on creating a new `AMGObject`:

```
Kamg = ml.setup( A,
                 maxLevels=25, maxCoarseSize = 32,
                 maxCycles = 1000, tol=1e-8,
                 useSuperLU=0, printLevel=10,
                 printFrequency=1, smoother="GaussSeidel",
                 smootherSteps=1, smootherOmega=1.0)

Parameters:
  A:              input matrix as spmatrix.ll_mat object
  maxLevels:      maximum number of Multigrid levels to be
                  built
  maxCoarseSize:  maximum size of the system matrix on the
                  coarsest level
  maxCycles:      maximum number of multigrid V-cycles
                  performed, when the solve(), the cycle()
                  or the iterate() method is called
  tol:            tolerance used in the stopping criterion of
                  the solve(), the cycle() or the iterate()
                  methods
```

Of the smoothing options listed in [18], “Jacobi”, “GaussSeidel” and “SymGaussSeidel” seem to be supported.

AMGObject interface The `AMGObject` has the following properties and functions:

```
Properties:     { complexity, debugLevel, maxCoarseSize,
                  maxCoarseSize, maxCycles, maxLevels,
                  numLevels, printFrequency, printLevel
                  shape, smoother, smootherOmega, tol
                  smootherSteps, totalCycles, useSuperLU }
Methods:
    precon(b, x)  solves a linear system using a MG iteration
                  starting with a zero initial guess.
                  The right-hand side is stored in b. The
                  solution is stored in x.
                  x and b are 1D NumPy array of appropriate
                  shape and type. The precon method is
                  identical to the solve method.

    solve(b, x)   solves a linear system using a MG iteration
                  starting with a zero initial guess.
                  The right-hand side is stored in b. The
```

```
                 solution is stored in x.
                 x and b are 1D NumPy array of appropriate
                 shape and type. The solve method is
                 identical to the precon method.

cycle(x, y)      performs one MGV cycle on x and stores
                 result in y.

iterate(b, x)    solves a linear system using a MG iteration
                 On input, b contains the right-hand side
                 and x contains the initial guess. On output
                 the solution is stored in x.
                 x and b are 1D NumPy array of appropriate
                 shape and type.

iterate(b, x)    solves a linear system using a MG iteration
                 The iterate method is identical to the
                 precon method.
```

Appendix D

The Pysparse C API

The Pysparse module ships with no explicit documentation of the Pysparse C API, and this chapter endavours to present the most essential of the API for future users. It is based on our exploration of the source and header files for version 0.33.029, and helpful discussions with its author, Roman Geus. API The Pysparse C API is largely based on the API provided with the Numeric module and we have found it useful to search for solution to related problems and fixes for `Numeric`. There is an abundance of information and discussion groups to be found on the internet written by programmers working with the Numeric module.

After installing the Pysparse module the necessary include files should be located in your Python include path. See 7.3 on page 93. In version 0.33.029 of the Pysparse package most of the functionality is accessed through `pysparse/spmatrix_api.h`. Other possible header files of interest, depending on which objects you are working with, are `pysparse/ll_mat.h`, `pysparse/csr_mat.h` and `pysparse/sss_mat.h` giving the declaration for `LLMatObject`, `CSRMatObject` and `SSSMatObject`, respectively.

D.1 `pysparse/spmatrix_api.h`

In the manner of Numeric, `pysparse/spmatrix_api.h` mainly holds a collection of macro definitions declaring function pointers to other sections of the module implementation. One of the effects of this is that one must keep in mind that

```
SpMatrix_LLMatSetItem( this->llMat, i-1, j-1, entry );
```

would, for example, not necessarily produce the same results as

```
SpMatrix_LLMatSetItem( this->llMat, (i-1), (j-1), entry );.
```

A fair share of the descriptions here are gathered from comments in the source code, but we have attempted to make it all available in one document. The documentation also lists the source file each function is defined in to make it easier to study the restrictions the functions operate under. The functions appear here in the same order as they are listed in `pysparse/spmatrix_api.h`

SpMatrix_ParseVecOpArgs

```
int SpMatrix_ParseVecOpArgs( PyObject *args, double **x_data,
                             double **y_data, int n )
```

Parses the arguments of Python functions that expect two one-dimensional Py_Array objects. The return value is 0 if the operation was successful, or -1 after setting the Python error indicator if an error occurred. According to the documentation in the source code, use of this function is not recommended as it does not free the created objects. The documentation suggests instead the use of the macro `SPMATRIX_PARSE_ARGS_ARR_ARR` defined in `pysparse/spmatrix.h`.

Source file: `spmatrixmodule.c`

SpMatrix_GetShape

```
int SpMatrix_GetShape( PyObject *op, int dim[] )
```

where `dim[]` has minimum length 2. Queries the shape attribute of an object and stores the result in `dim`. It should be used to get the number of rows and columns in a sparse matrix or a preconditioner pointed to by `op`. Returns 0 if the operation was successful, or -1 after setting the Python error indicator if an exception occurred.

Source file: `spmatrixmodule.c`

SpMatrix_GetOrder

```
int SpMatrix_GetOrder( PyObject *op, int *n )
```

Has much of the same functionality as `SpMatrix_GetShape`, but it will fail if the shape of the sparse matrix or preconditioner is not square. Assumes that `n` points to allocated memory of minimum size `sizeof(int)*2`. Like `SpMatrix_GetShape`, the function returns 0 if the operation was successful, or -1 after setting the Python error indicator if an exception occurred.

Source file: `spmatrixmodule.c`

SpMatrix_GetItem

```
double SpMatrix_GetItem( PyObject *op, int i, int j )
```

Accesses matrix entry `(i,j)` and returns the entry `op[i,j]` as a double.

Source file: `spmatrixmodule.c`

SpMatrix_Matvec

```
int SpMatrix_Matvec( PyObject *matrix, int nx,
                     double *x, int ny, double *y )
```

Invokes matrix-vector multiplication, by calling the `matvec`-method associated with `matrix` to compute `y = matrix*x`. The vectors `x` and `y` are given as arrays of double. Function returns 0 if the operation was successful, or -1 if an error occurred. Depending on the type of `matrix`, i.e whether it is a `LLMatObject`, `CSRMatObject` or a `CorrEqObject`, the `matvec` method called will be either `LLMat_matvec`(···) defined in `ll_mat.c`, `CSRMat_matvec`(···) defined in `csr_mat.c` or `CorrEq_matvec`(···) defined in `correq.c`.

Source file: `spmatrixmodule.c`

SpMatrix_Precon

```
int SpMatrix_Precon( PyObject *prec, int n,
                     double *x, double *y )
```

Applies the preconditioner `prec` on the vector `x` and stores the result in vector `y`. This is done by calling the `precon` method of `prec`. The vectors `x` and `y` are given as arrays of double with length `n`. The return value is 0 if the operation was successful, or -1 if an error occurred. Like in `SpMatrix_Matvec`, the type of `prec` governs which `precon` method will be called. The called preconditioner will be either `Jacobi_precon`(···) or `SSOR_precon`(···), both defined in `preconmodule.c`, or `CorrEq_precon`(···) defined in `correq.c`, depending on whether `prec` is a `JacobiObject`, `SSORObject` or a `CorrEqObject`.

Source file: `spmatrixmodule.c`

SpMatrix_NewLLMatObject

```
PyObject* SpMatrix_NewLLMatObject( int dim[], int sym,
                                   int sizeHint )
```

Creates a Python object of type `LLMatObject` representing a `dim[0]`×`dim[1]` sparse matrix. `sym` indicates whether the matrix is symmetric or not (i.e. should evaluate to `true` or `false`), and `sizeHint` give an approximate to the number of non-zeros in the matrix. Sets the Python error indicator and returns `NULL` on failure.

Source file: `ll_mat.c`

SpMatrix_LLMatGetItem

```
double SpMatrix_LLMatGetItem( LLMatObject *a, int i, int j )
```

Returns the matrix entry `a[i,j]` as a double value.

When you do try to use `SpMatrix_LLMatGetItem` or the subsequent macros, you might have compilation errors on the form `syntax error : ')'`. `SpMatrix_LLMatGetItem` is the first function that accepts a `LLMatObject` as a function parameter. You must either modify `spmatrix_api.h` to include `ll_mat.h` *or* make sure you include `ll_mat.h` in your own source file to make the definition of `LLMatObject` available to prevent the macro expansion from failing.

Source file: `ll_mat.c`

SpMatrix_LLMatSetItem

```
int SpMatrix_LLMatSetItem( LLMatObject *a, int i, int j,
                           double x )
```

Sets the matrix entry `a[i,j] = x`. Returns 0 if the operation was successful, or -1 after setting the Python error indicator if an exception occurred.

Source file: `ll_mat.c`

SpMatrix_LLMatUpdateItemAdd

```
int SpMatrix_LLMatUpdateItemAdd( LLMatObject *a, int i, int j,
                                 double x )
```

Performs the operation `a[i,j] += x` on the matrix entry. Returns 0 if the operation was successful, or -1 after setting the Python error indicator if an error occurred.

Source file: `ll_mat.c`

SpMatrix_LLMatBuildColIndex

```
int SpMatrix_LLMatBuildColIndex( struct llColIndex **idx,
                                 LLMatObject *self,
                                 int includeDiagonal )
```

Builds data structure for column-wise traversal. Builds a linked-list data structure, which links the entries of each column in the object pointed to by `self`. If `includeDiagonal` is zero the diagonal elements of `self` are not included in the linked-list data structure. Returns 0 if the operation was successful, or -1 after setting the Python error indicator if an exception occurred. `llColIndex` is defined in `ll_mat.h`, and is a list data structure which links the entries of each column of a LLMat matrix.

Source file: `ll_mat.c`

SpMatrix_LLMatDestroyColIndex

```
void SpMatrix_LLMatDestroyColIndex( struct llColIndex **idx )
```

Frees the memory allocated by calling `SpMatrix_LLMatBuildColIndex`.

Source file: `ll_mat.c`

ItSolvers_Solve

```
int ItSolvers_Solve( PyObject *linsolver, PyObject *A, int n,
                     double *b, double *x, double tol,
                     int itmax, PyObject *K, int *info,
                     int *iter, double *relres )
```

Invokes the iterative linear solver `linsolver` (callable Python object), to (approximately) solve the linear system

$$A * x = b$$

to an accuracy of `tol`. The maximum number of iteration steps taken is `itmax`. The vectors `x` and `y` are given as arrays of double and have length `n`. The function returns 0 if the operation was successful, or -1 if an error occurred.

Source file: `spmatrixmodule.c`

import_spmatrix

```
import_spmatrix()
```

Similarly to creating extension modules using the Numeric C API, `import_spmatrix()` must be called in the initialization function of your module. This macro assures that the `spmatrix` module is loaded and that the pointer array `SpMatrix_API` is initialized. If `SpMatrix_API` is not initialized the macros in `pysparse/spmatrix_api.h` described above break down and will give runtime errors in your module.

If the extension module spans several source files make sure that
`SPMATRIX_UNIQUE_SYMBOL uniqueLabel_spmatrix` is defined in each file to make
`spmatrix` handle the `SpMatrix_API` function pointer array correctly. If you do not do this the array will be declared multiple times, because `spmatrix_api.h` will be included in several source files and you will have compilation and/or linking problems. This means that for every source file where you include the API header, you must declare

One fix we did to the Pysparse C API when working on this thesis was to borrow a feature from Numeric's C API. If this is the case for your installation, you should also define `NO_IMPORT_SPMATRIX` together with `SPMATRIX_UNIQUE_SYMBOL` in *every other file* than the file where the module initialization function is located (i.e. in every file where import_spmatrix is not called). If this does not seem to help you must probably add support for SPMATRIX_UNIQUE_SYMBOL to your spmatrix module yourself by modifying your pysparse/spmatrix_api.h header file. Replace the existing code around SPMATRIX_UNIQUE_SYMBOL with the following:

```
/* C API address pointer */
#if defined(NO_IMPORT_SPMATRIX) /*Mimic Numeric fix*/
extern void **SpMatrix_API;
#else
#if defined(SPMATRIX_UNIQUE_SYMBOL)
void **SpMatrix_API;
#else
static void **SpMatrix_API;
#endif
#endif
```

As we can see, by using NO_IMPORT_SPMATRIX we declare SpMatrix_API to be extern in all other files than the file where import_spmatrix() is called, and thus fixing the compilation/linking errors. This fix was reported to the author, and it is included from versoin 0.34.031.

Source file: spmatrix_api.h

Bibliography

[1] Uri M. Ascher and inda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential–Algebraic Equations*. Society for Industrial and Applied Mathematics (SIAM), July 1998.

[2] Guillaume Bal. On the Convergence and the Stability of the Parareal Algorithm to solve Partial Differential Equations. In Ralf Kornhuber, Ronald Hoppe, Jaques Périaux, Olivier Pironneau, Olof Widlund, and Jinchao Xu, editors, *Domain Decomposition Methods in Science and Engineering*, volume 40 of *Lecture Notes in Computational Science and Engineering*, pages 425–432. Springer Verlag, October 2004. Proceedings of the 15th International Conference on Domain Decomposition Methods in Berlin.

[3] Dietrich Braess. *Finite Elements. Theory, fast solvers, and applications in aolid mechanics*. Cambridge University Press, 1997.

[4] M.O Deville, P.F Fischer, and E.H Mund. *High-Order Methods for Incompressible Fluid Flow*. Number 9 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, first edition, August 2002.

[5] Lawrence C. Evans. *Partial Differential Equations*. Number 19. American Mathematical Society, 1998.

[6] Martin Golubitsky and Michael Dellnitz. *Linear Algebra and Differential Equations Using MATLAB*. Brooks Cole, first edition, January 1999.

[7] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, January 1996.

[8] Hans Petter Langtangen. *Computational Partial Differential Equations. Numerical Methods and Diffpack Programming*. Number 1 in Texts in Computational Science and Engineering. Springer Verlag, second edition, March 2003.

[9] Hans-Petter Langtangen and Kent-Andre Mardal. Mixed Finite Elements. In Hans Petter Langtangen and Aslak Tveito, editors, *Advanced Topics in Computational Partial*

Differential Equations, volume 33 of *Lecture Notes in Computational Science and Engineering*, chapter 4, pages 153–198. Springer Verlag, November 2003.

[10] Hans-Petter Langtangen and Kent-Andre Mardal. Systems of PDEs and Block Preconditioning. In Hans Petter Langtangen and Aslak Tveito, editors, *Advanced Topics in Computational Partial Differential Equations*, volume 33 of *Lecture Notes in Computational Science and Engineering*, chapter 5, pages 199–236. Springer Verlag, November 2003.

[11] Hans-Petter Langtangen and Kent-Andre Mardal. Using Diffpack from Python Scripts. In Hans Petter Langtangen and Aslak Tveito, editors, *Advanced Topics in Computational Partial Differential Equations*, volume 33 of *Lecture Notes in Computational Science and Engineering*, chapter 8, pages 321–359. Springer Verlag, November 2003.

[12] Hans-Petter Langtangen, Kent-Andre Mardal, and Ragnar Winther. Numerical Methods for Incompressible Viscous Flow. In C.T. Miller, M.B. Parlange, and S.M. Hassanizadeh, editors, *25 Years of Advances in Water Resources*, pages 265–286. Elsevier Science & Technology Books, May 2003. Originally published in *Advances in Water Resources*, Vol.25, Issue 8, August 2002, pp.1125-1146.

[13] David C. Lay. *Linear algebra and its applications*. Addison–Wesley World Student Series. Addison–Wesley, second edition, October 1999. International Edition.

[14] Jaques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'EDP par un schéma en temps "pararéel". *Comptes Rendus de l'Académie des Sciences. Série I - Comptes Rendus Mathematique*, 332(7):661–668, April 2001.

[15] Kent-Andre Mardal and Ragnar Winther. Uniform preconditioners for the time dependent Stokes problem. *Numerische Mathematik*, 98(2):305–327, April 2004. Springer Verlag Berlin.

[16] Niels Saabye Ottosen and Hans Petersson. *Introduction to the Finite Element Method*. Prentice Hall, first edition, June 1992.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. The PWS Series in Computer Science. PWS Publishing Company, first edition, July 1996.

[18] Marizio Sala, Jonathan J. Hu, and Ray S. Tuminaro. *ML 3.1 Smoothed Aggregation User's Guide*. Computational Math & Algorithms, Sandia National Laboratories, 3.1 edition, September 2004. Sandia report number SAND2004-4821.

[19] Jonathan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. `http://www-2.cs.cmu.edu/~jrs/jrspapers.html`, August 1994.

[20] Gunnar Andreas Staff. Convergence and Stability of the Parareal Algorithm. Master's thesis, Norwegian University of Science and Technology (NTNU), June 2003. Faculty of Information Technology, Mathematics and Electrical Engineering.