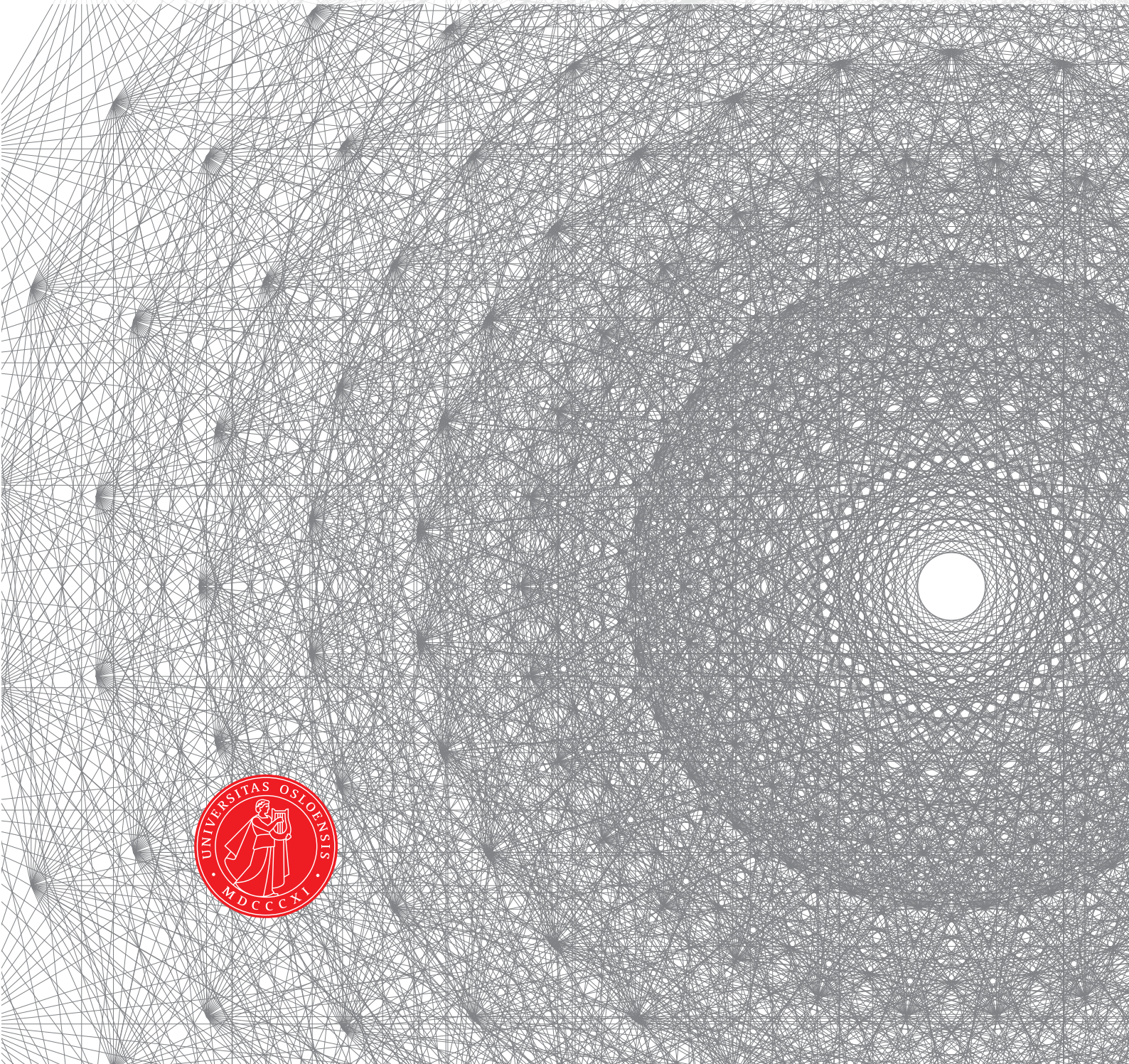
UiO : Department of Mathematics
University of Oslo

A Parallel in Time Method for Optimal Control

Parareal-Based Preconditioner for the BFGS Algorithm

Andreas Thune

Master's Thesis, Spring 2017



This master's thesis is submitted under the master's programme *Computational Science and Engineering*, with programme option *Computational Science*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group E_8, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.

Abstract

In this thesis we propose a parallel in time method for reducible optimal control problems with differential equation constraints. Our method uses a BFGS algorithm with a Parareal-based preconditioner to optimize a series of unconstrained and time-parallelizable subproblems that depend on a penalty parameter μ. For large μ's, the solution of the subproblems will approach the numerical solution of the sequential algorithm.

We derive and implement the method for an ODE constrained example problem, and explore the consistency of the method both through theory and experiments. We present the already known Parareal-based preconditioner and derive some of its properties. We also show that it is applicable to the BFGS optimization algorithm.

The performance of our proposed algorithm is tested on the ODE constrained example problem using both simulated and actual parallelism. We observe that our preconditioned method seems to be independent of the number of decompositions of the time interval, and we were able to achieve modest speedup results between 9 and 23.5 on 48 to 120 cores. We also observe that our method experiences significant loss of potential speedup for large penalty parameters μ.

Acknowledgements

I want to start by offering my thanks to the Department of Mathematics at the University of Oslo (UiO) for giving me the opportunity to carry out this master thesis. A special thanks goes to Simula Research Laboratory for making it possible for me to write my master thesis in an excellent research environment, and for providing localities for studying as well as generous benefits.

I would particularity like to thank my supervisor Simon Wolfgang Funke for his invaluable input and direction, and for our constructive and informative weekly meetings. Without your help, guidance and positive attitude, the completion of this thesis would not have been possible. I appreciate the remarks and comments offered by my secondary supervisor Kent Andre Mardal, especially for his contributions at the beginning of this project.

I thank UNIK at UiO for permission to use The Abel computer cluster, which has been a valuable tool in the work on this thesis.

Andreas Thune
Oslo, 29. Mai 2017

Contents

Chapter 1

Introduction

In today's world, high performance computing is an essential tool for scientists in many fields such as engineering, computational physics and chemistry, bioinformatics or weather forecasting. Many problems that arise in these areas are so computationally costly, that they can not be solved efficiently or at all on a single processor. Instead we solve or accelerate the solution of such problems by running them on large-scale clusters of multiple processes in parallel. One of the main issues with parallel computing is that many numerical solvers are sequentially formulated, and the work of translating these algorithms into a parallel framework can often be time and effort intensive.

One class of large-scale problems suited for parallelization, that frequently occurs both in science and engineering, are time-dependent partial differential equations (PDEs). The traditional approach to implementing parallel solvers for such problems is to restrict the parallel computations to operations in spatial dimension at each time step, while the time-integration is done sequentially. Letting the implementation be serial in temporal direction is the most intuitive way of parallelizing time-dependent PDEs, since evolving an equation in time is a naturally sequential process. A lot of work has also been done on parallel solvers for spatially dicretized problems, meaning that methods and strategies for parallelization in space already are developed and tested [1]. However, for a fixed discretization, the achievable speedup through spatial parallelization is limited when the number of cores are high. Therefore, introducing parallelism in temporal direction is a way of increasing the speedup beyond this bound. It is therefore desirable to develop solvers for time-dependent PDEs that are parallel in time.

There exists multiple methods for parallel in time solvers of evolution equations. The most famous and most developed of these methods is the so called Parareal method introduced in [2]. The parallelism of Parareal is restricted to the temporal

dimension, and can therefore be used to parallelize both time-dependent PDEs and ordinary differential equations (ODEs). It shares this feature with the related multiple shooting methods [3, 4], while waveform relaxation methods [5, 6] and multigrid methods [7–9] achieves parallelism in time by parallelizing in both space and time simultaneously. In this thesis we will restrict ourself to Parareal, and the other methods will not be touched upon any further.

Parallel differential equation (DE) solvers are well studied. In this thesis we will focus on a similarly important set of problems. Optimization problems constrained by time-dependent DEs. These occur for example in: Optimal control ,variational data assimilation and optimal design. Optimization with differential equation constraints are minimization problems of the following form:

$$\min_{y,v} J(y,v) \quad \text{subject to } E(y,v) = 0. \tag{1.1}$$

The functional J that we want to minimize is usually referred to as the objective function. $E(y,v) = 0$ represents the differential equation constraint, and is called the state equation. The state equation is solved for the state y, while the v variable called the control represents parameters of the equation. The goal of the control problem (1.1) is to find a pairing $(\bar{y}, \bar{v})$ that minimizes the objective function, but also satisfies the constraints set up by the state equation $E(\bar{y}, \bar{v}) = 0$. For further details on optimization we refer to [10].

Different strategies for numerically solving the optimal control problem (1.1) exists. One alternative is to set up and solve the optimality system stemming from the Lagrangian function associated with problem (1.1). In this thesis we will not use this approach, but instead reduce the constrained problem (1.1) into an unconstrained problem of type (1.2), and then solve this new problem using techniques from unconstrained optimization.

$$\min_{v} J(y(v), v). \tag{1.2}$$

Choosing this approach will obviously limit us to optimization problems where this reduction is possible, which is the case for many practical problems. The process of finding the minimizer of problem (1.2) involves repeatedly solving differential equations. The computational cost of solving these equations will dominate the overall computational cost of any optimal control solver based on the reduction approach. Parallelization of problems of type (1.2) are therefore connected to the parallelization of differential equations. In this thesis we will develop and investigate a Parareal-based parallel in time framework for optimal control problems with time-dependent DE constraints. To achieve this, we will use the same strategy as

in [11], meaning that we enforce the dependency between decomposed intervals by altering the objective function. The Parareal algorithm is then applied as a preconditioner for the optimization algorithm.

In [11] the parallelization of time-dependent optimal control problems is done by applying the Parareal preconditioner to the steepest descent method. We will instead propose and implement a parallel in time algorithm using the same Parareal preconditioner in combination with the BFGS method. We will also derive properties of the Parareal-based preconditioner showing that it is compatible with BFGS. The consistency of our method is discussed both through theory and experiments. The algorithm that we present in this thesis is applicable to optimal control problems with ODE and PDE constraints, and for PDE constraints it can also be combined with spatial parallelization. For simplicity we will restrict the example problems to ordinary differential equation constraints.

Our algorithm is in many ways similar to the parallel in time algorithm for 4d variational data assimilation introduced in [12], since that algorithm also is based on the BFGS method. The main difference between our algorithmic framework and the one found in [12], is that we include the Parareal-based preconditioner from [11]. Experiments conducted in [12] showed that their algorithm experienced low to no speedup at all, and that the problems grew when more CPUs were added. Our algorithm does not share these scaling problems. In fact our experiments show that the algorithmic framework introduced in this thesis works better for higher numbers of processes (16+) than it does when we use a smaller number of processes.

1.1 Summary

The overall goal of this thesis is to establish a parallel in time algorithm for solving optimal control problems with time-dependent DE constraints. The structure of the work done can roughly divided into two parts:

1. Background and presentation of the algorithms
2. Verification and experiments

The bulk of the thesis is found in the first part, where we present and motivate a parallel framework for parallelization of control problems. In chapter 2 we give a short literature review of previous work done on the Parareal algorithm, its theory and its application, emphasizing possible extensions to optimal control. In chapter 3 we look at general theory for optimal control with DE constraints. Among other things the adjoint approach to gradient evaluation is presented. Chapter 3

also includes one section on optimization algorithms and one on finite difference discretizations of ODEs. A more detailed, but still shallow presentation of the Parareal algorithm is found in chapter 4. Of special importance in this presentation, is the section on the alternative algebraic formulation of Parareal, since this formulation is used later in chapter 5 to derive the Parareal based preconditioner.

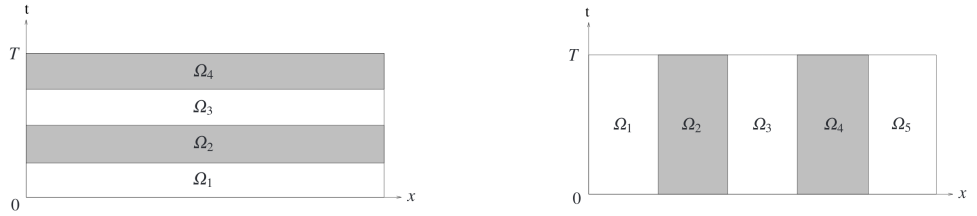
How we translate the solution process of time-dependent optimal control problems into a parallel framework, is detailed in 5. Here we explain how to decompose the time domain of control problems, and how we can use the penalty method to enforce the continuity constraints that arises when decomposing in time. The rest of chapter 5 is dedicated to the presentation and derivation of the Parearal preconditioner. Since we want to use this preconditioner in combination with the BFGS optimization algorithm, we also need to check if the proposed preconditioner possesses the necessary mathematical properties for this to be possible.

The second part of the thesis deals with implementation, testing and verification of the algorithm. In chapter 6 we explain how we discretize the decomposed time domain and the non-penalized and penalized objective function for an example problem. In chapter 7 the discetized objective function and its gradient from chapter 6 is verified using the Taylor test. In the second part of chapter 6 we also explain how we implement the objective function and gradient evaluation in parallel using the message passing interface (MPI), with special attention on communication between processes and how this communication affects the speedup. The speedup of our implementation for function and gradient evaluation is also verified against the theoretical optimal speedup in chapter 7. Chapter 7 also demonstrates how the solution of the discretized control problem converges to the exact solution, and the consistency of the penalty framework presented in chapter 5. Chapter 8 contains the results of experiments conducted using the method proposed in chapter 5. The main focus of these results are the speedup this parallel algorithm produces. We measure speedup both in wall clock time and in a measure representing ideal speedup. This ideal speedup is based on the number of objective function and gradient evaluations done by the sequential and parallel algorithm.

Chapter 2

Literature Review

The Parareal algorithm is not the first attempt to parallelize the solution of time-dependent differential equation in temporal direction, since Nievergelt already in 1964 proposed a procedure in [3] that eventually led to the so called multiple shooting methods. In [13] the authors relate the Parareal algorithm to the multiple shooting methods, and also explains why Parareal can be thought of as a multigrid method. A historic overview of the development of parallel in time algorithms can be found in [14]. Here the author also present different strategies for parallelizing time-dependent differential equations. One such strategy are the already mentioned multiple shooting methods [3, 4], which also include the Parareal algorithm. What characterizes such methods is that they only decompose the time domain. This separates the multiple shooting methods from waveform relaxation methods [5,6], where the spatial domain is decomposed through time. The difference in these decomposition techniques is illustrated in figure 2.1. Other strategies presented are multigrid [7–9] methods and direct solvers in space-time [15–17].



(a) Multiple shooting decomposition (b) Waveform relaxation decomposition

Figure 2.1: Different decomposition techniques for parallel in time algorithms. (a) shows the strictly temporal decomposition of multiple shooting methods. In (b) the decomposition is done spatially through time. Image source: [14].

The Parareal algorithm was introduced by Lions, Maday and Turinici in [2] as a

way to solve differential evolution equations $f(y(t), t) = 0$ with solution $y(t)$ in parallel. The idea is to combine a coarse (but fast) and fine (but slow) numerical scheme for discretization in time. To introduce parallelism we first decompose the time domain $I = [0, T]$ into N subintervals $I_i = [T_{i-1}, T_i]$. This gives us N equations $f_i(y_i(t), t) = 0$ defined on each interval I_i.

The first step of the Parareal algorithm is to solve $f(y(t), t) = 0$ sequentially on the entire interval using the coarse scheme. This gives us a (coarse) solution $Y(t)$ defined on the entire interval. We can then use $\{\lambda_i^0 = Y(T_i)\}_{i=1}^{N-1}$ as initial conditions for the decomposed equations $f_{i+1}(y_{i+1}^0(t), t) = 0$. The second step is then to solve these equations in parallel using the fine scheme, which will result in one solution $y_i(t)$ on each interval I_i. The idea then, is to utilize the difference $S_i^0 = y_i^0(T_i) - \lambda_i^0$ between coarse and fine solution to repeat this process in an iteration. This is done by propagating the differences S_i^0 with the coarse solver, to update the initial conditions for each decomposed equation. These new initial conditions λ_i^1 can then be used to solve the decomposed equations $f_{i+1}(y_{i+1}^1(t), t) = 0$ in parallel with the fine solver. We can then define updated differences $S_i^1 = y_i^1(T_i) - \lambda_i^1$ and repeat the iteration until we are satisfied with the solution. A schematic presentation of the above described procedure can be viewed in figure 2.2. The version of Parareal presented in [2] is most practical for use on linear equations. An alternative version of Parareal algorithm is found in [18], which is equivalent to the one in [2] for linear equations, but is easier applied to non-linear equations.

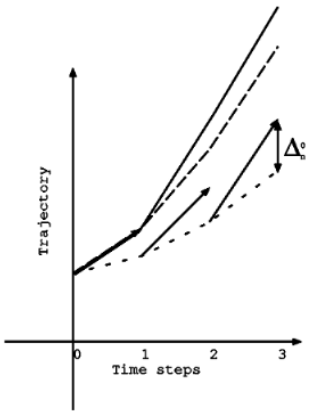


Figure 2.2: Schematic presentation of Parareal step. A first trajectory is generated using the coarse scehme (dashed line). Then starting from all points of this trajectory, we advance the equation with the fine scheme (arrow). The error is measured by $S_i^0 = \Delta_i^0$. We generate the corrected trajectory (long dashed line) by propagating S_i^0 with the coarse scheme. This results in a trajectory closer to the exact trajectory (solid line). Notice that the Parareal trajectory is exact for the first time step. Source of image and caption: [18].

A lot of the work on the Parareal algorithm has been focused on establishing its stability and convergence properties. The stability results are found in [19], [20] and [21]. In [19,20] sufficient conditions for the stability of Parareal of autonomous

differential equations (2.1) is derived:

$$\frac{\partial y}{\partial t} = \rho y, \quad y(0) = y_0, \quad \rho < 0 \tag{2.1}$$

while [21] presents more general stability results for parabolic equations. The stability of Parareal applied to hyperbolic equations is a more difficult question [22]. The convergence of Parareal is studied in [2], [21], [23] and [13]. In [2] Lions, Maday and Turinici show that k iterations of the Parareal algorithm applied to equation (2.1) gives $\mathcal{O}(\Delta T^{k+1})$ order of accuracy if we use a coarse solver with order one accuracy and coarse time step ΔT. This result is extended in [21] to more general equations, and the order of accuracy is shown to be improved to $\mathcal{O}(\Delta T^{p(k+1)})$ when the coarse solver has order p. [13, 23] return to analysis of equation (2.1). Instead of looking at a fixed number of iterations k, Gander and Vandewalle show convergence properties for the Parareal algorithm as the iteration count increases. They derived superlinear convergence for bounded time intervals and linear convergence for unbounded time intervals.

The Parareal algorithm has been applied to different equations, including on the Navier-Stokes equations [24], to molecular-dynamics simulations [18], to stochastic ordinary differential equations [25], to reservoir simulations [26], to fluid, structure and fluid-structure problems [27], or on the American put [28]. The success of applying the Parareal algorithm varies between the different problems. For example in [28] a simulated speedup of 6.25 is achieved on 50 decompositions, which translates to an efficiency of 12.5%. In [27], speedups between 4.0 and 8.2 are achieved on twenty cores for an unsteady flow model. This corresponds to an efficiency of $20\% - 41\%$. The parallel in time algorithm was less successful when applied to structure and fluid-structure dynamics, since the authors of [27] here experienced difficulties with stability. For certain problem parameters, stability issues are encountered in [24], however for other parameters the algorithm is stable, and a speedup between 6.0 and 19.7 for 32 cores is estimated. This estimation, that assumes zero parallel overhead, would yield efficiency between 18.75% and 61.56%.

Since the Parareal algorithm is an iterative procedure, a stopping criteria for when to terminate the iteration is required. This is studied in [29], where an error control mechanism for the Parareal algorithm is introduced to limit the number of Parareal iterations. The stopping criteria that the authors propose stops the algorithm when the difference between coarse and fine solution at the subinterval boundaries T_i are similar to the expected global error of the fine solver. One challenge associated with parallel computing is partition and load bearing. This issue also arises in the Parareal algorithm, where the difficulties originates from the following observation: After k iterations of the Parareal algorithm, the solution in

the k first subintervals is equal to the fine solution, see figure 2.2. This means that after k iterations, the the k-th process becomes idle. How to tackle this issue is described in [30], where the authors also present a practical implementation of the Parareal algorithm.

The Parareal algorithm parallelizes the solution process of time-dependent differential equations. In [11] Maday and Turinici extend Parareal for optimal control problems with time-dependent differential equation constraints. In particular the problem looked at in [11], is:

$$\min_{y,u} J(y,u) = \frac{1}{2}\int_0^T ||u(t)||_U^2 dt + \frac{\alpha}{2}||y(T) - y^T||^2,$$
$$\begin{cases} \frac{\partial y}{\partial t} + Ay = Bu \\ y(0) = y_0 \end{cases}$$

The authors introduce parallelism in the same ways as for the differential equation case, by decomposing the time domain and equation. The continuity of the state equation between subintervals is enforced by adding a penalty term to the objective function J, that penalizes jumps in the solution of the state equation. This is based on the penalty method for constrained optimization described in [31]. In [11] they use quadratic penalty terms, which leads to the following modified objective function:

$$J_\mu(y, u, \lambda_1, .., \lambda_{N-1}) = J(y,u) + \frac{\mu}{2}\sum_{i=1}^{N-1}(y_i(T_i) - \lambda_i)^2 \tag{2.2}$$

The λ_i variables are called the virtual controls and are the initial conditions of the decomposed state equations $f(y_{i+1}(t), t) = 0$. Solving both the original and modified optimal control problems require us to repeatedly evaluate the objective function and its gradient. Every time we do this we need to solve either the state equation, or the state equation and its adjoint. Decomposing the time interval allows us to solve these equations in parallel, and if we solve the modified problem with a sufficiently large penalty μ, we will end up with the solution of the original problem. One does not necessarily need a coarse level to make this parallel framework produce a speedup. This is illustrated in [12], where the authors create a time-parallel algorithm for 4d variational data assimilation. The penalization of the objective function was done using the augmented Lagrangian approach, which is a variation of the penalization done in (2.2). However, the experiments conducted in [12] yielded limited success. Some speedup was achieved, however, the speedup was only attainable when using a parallel/sequential hybrid method, that first solved the penalized problem in parallel, for small penalty terms, and then

used the parallel solution as an initial guess for the sequential algorithm.

In [11] the Parareal algorithm is reformulated as a preconditioner for the algebraic system that arises when we set $\lambda_i = y_i(T_i)$. Using this formulation the authors derive a preconditioner for the optimization algorithm that solves the penalized optimal control problem. The preconditioner they propose involves both a backward and a forward solve of the linearised state equation with a coarse solver, and it is to be applied to the λ part of the gradient of J_μ. The motivation is that this Parareal based preconditioner could decrease the number of function and gradient evaluations needed for the optimization algorithm to converge, and the results in [11] do indeed look promising. In an experiment with 100 cores, the authors report a theoretical speedup of around 400, which is superlinear. They do however believe that this result is due to properties of the example they chose, and do not expect superlinerar speedup as a general rule.

The optimal control setting can also be used to modify the original Parareal algorithm. One example is [32], where the preconditioner for the optimal control problem from [11] is used in a modified Parareal algorithm to stabilize it for hyperbolic equations. The adjoint based Parareal algorithm is proposed in [33]. In this paper the authors address the bottleneck for speedup produced by having to repeatedly apply the coarse solver. This especially becomes a problem when the number of decompositions in time grows, while the problem size stays constant. The solution proposed in [33] is to only use the coarse solver once to get an initial guess for the intermediate initial conditions, and thereafter improve this initial guess by minimizing a functional of type (2.2) using an optimization algorithm. The optimization steps can be done completely in parallel, and the scalability of the adjoint based Parareal algorithm is therefore a lot better than the original.

In [34, 35] the authors derive a way to couple the Parareal algorithm with an optimization procedure for control of quantum systems. Like in [11] a penalty term is added to the objective function to handle the continuity constraints, but the optimization of the penalized functional is done in a slightly different way than in [11]. The approach taken in [35] is to minimize the penalized objective function using an alternating direction decent method. This means that the minimization of the functional of type 2.2 is done in two steps. First we minimize it for the virtual control $\{\lambda_i\}_{i=1}^{N-1}$, and then for the original control v. A Parareal step is incorporated into the minimization of the penalized objective function with respect to the virtual control variables.

We will in this thesis handle the DE constraints by moving them into the ob-

jective function, and therefore reducing the constrained optimization problems to unconstrained ones. An alternative to this strategy is the Lagrangian approach, where one first defines the Lagrangian function (2.3), and then derive the optimality system using the KKT-conditions.

$$\mathcal{L}(y, v, \lambda) = J(y, v) + \lambda E(y, v) \tag{2.3}$$

Some work has been done on trying to apply the Parareal algorithm to the solution process of the optimality system. For reference see: [36–39].

Chapter 3

Optimal Control with ODE Constraints

In this chapter we present the basic mathematical background that the rest of the thesis will be based on. The chapter covers three different subjects. The first subject is on general theory of optimal control problems with DE constraints. The second subject is on finite difference discretization of differential equations and numerical integration, and the last subject deals with optimization algorithms. In addition to the general theory, we present an example optimal control problem with ODE constraints, that will be used throughout the rest of the thesis.

3.1 General Optimal Control Problem

In this thesis we consider reducible optimization problems with time-dependent differential equation constraints. This problem is a special case of the more general optimization problem, which we will state in definition 3.1. Here we also define what it means for an optimization problem to be reducible, by introducing the reducibility condition (3.3).

Definition 3.1 (Optimization with DE constraints)**.** *Let Y, V, Z be Banach spaces, where Y, V also are reflexive. Given an objective function $J : Y \times V \to \mathbb{R}$ and an operator $E : Y \times V \to Z$, optimization with DE constraints then refers to minimization problems on the following form:*

$$\min_{y \in Y, v \in V} J(y, v), \tag{3.1}$$

$$\text{Subject to: } E(y, v) = 0. \tag{3.2}$$

The differential equation $E(y, v) = 0$ is called the state equation, while the variables y and v are respectively known as the state and the control. If the following

condition holds:

$$\forall v \in V,\ \exists! y \in Y \ s.t.\ E(y, v) = 0, \tag{3.3}$$

we say the the optimization problem is reducible.

An alternative way of expressing the reducibility condition (3.3), is that for all controls $v \in V$ the differential equation $E(y, v) = 0$ is well posed. Optimization problems with ill posed state equations can both be solvable and interesting, but the methods introduced in this thesis are designed around reducible problems. Therefore we will from this point always assume that the optimization problems we look at are reducible. When the reducibility condition (3.3) holds the state can be written as a function $y(v)$ implicitly defined through the state equation. Using $y(v)$ we are able to define the reduced optimization problem:

Definition 3.2 (Reduced problem)**.** *Consider the optimization problem from definition 3.1, and assume that reducibility condition (3.3) holds. We can then define the reduced objective function $\hat{J} : V \to \mathbb{R}$ as:*

$$\hat{J}(v) = J(y(v), v). \tag{3.4}$$

The reduced optimization problem is then defined as the unconstrained minimization problem:

$$\min_{v \in V} \hat{J}(v). \tag{3.5}$$

Problem (3.5) is called the reduced problem because we have moved the differential equation constraints into the functional. By doing this, we have transformed the constrained problem (3.1-3.2) into an unconstrained one (3.5), and we can therefore solve the reduced problem using tools from unconstrained optimization. Let us therefore briefly discuss the fundamental theory that algorithms for unconstrained optimization are based on. We start by defining what it means to be a minimizer of a functional $J : V \to \mathbb{R}$.

Definition 3.3 (Global and local minimizer)**.** *A point $\bar{v} \in V$ is called a global minimizer of the functional $J : V \to \mathbb{R}$, if:*

$$\forall v \in V \quad J(\bar{v}) \leq J(v). \tag{3.6}$$

A point $\bar{v} \in V$ is called a local minimizer of J if there exists a neighbourhood $\mathcal{N} \subset V$ of $\bar{v}$ such that:

$$\forall v \in \mathcal{N} \quad J(\bar{v}) \leq J(v). \tag{3.7}$$

If the inequality of (3.7) is strict we say that $\bar{v}$ is a strict local minimizer.

Investigating whether a point $\bar{v} \in V$ is a local minimizer of a functional J using definition (3.7), would require us to check if $J(\bar{v}) \leq J(v)$ for all v's in the vicinity of $\bar{v}$. This is not a viable strategy, and for sufficiently smooth functionals more efficient and practical ways for identifying minimizers exist. Typically, as we soon will see, we can check whether $\bar{v}$ is a minimum of J, by examining the gradient $\nabla J(\bar{v})$ and the Hessian $\nabla^2 J(\bar{v})$. In theorem 3.1 from [31] we present sufficient conditions on $\nabla J(\bar{v})$ and $\nabla^2 J(\bar{v})$, which guarantee that $\bar{v}$ is a strict local minimizer of J, for the case when $V = \mathbb{R}^n$. Generalizations of theorem 3.1 for when V is a reflexive Banach space will not be considered here. For further details on this topic see [10].

Theorem 3.1. *Let $J : \mathbb{R}^n \to \mathbb{R}$ be a functional, and assume its Hessian is twice continuously differentiable in a neighbourhood $\mathcal{N}$ of a point $\bar{v} \in \mathbb{R}$. If $\nabla J(\bar{v}) = 0$ and if $\nabla^2 J(\bar{v})$ is positive definite, $\bar{v}$ is a strict local minimizer of J.*

Proof. See [31] □

The conditions of theorem 3.1 are sufficient for $\bar{v}$ to be a local minima, but the condition on the Hessian of J is not always necessary. In some cases, in particular when J is convex, $\nabla J(\bar{v}) = 0$ is sufficient for determining if $\bar{v}$ is a local minimizer. When J is convex, any local minimizer $\bar{v}$ will additionally be a global minimizer. The foundation for unconstrained optimization algorithms is based around theorem 3.1, since such algorithms always seek a point where the gradient of J is zero. To be able to use tools from unconstrained optimization on problem (3.5), we will therefore need a way to evaluate the gradient of the reduced objective function $\hat{J}$. There are several ways of doing this, but we will focus on the so called adjoint approach, which turns out to be the most computationally effective way to evaluate $\hat{J}'(v)$.

Before we explain the adjoint approach to gradient evaluation, we investigate under what conditions problem (3.1-3.2) even have a solution. To answer this question, we will write up a result from [10] concerning the existence and uniqueness of solution for linear-quadratic optimization problems. This class of problems is less general than the problems from definition 3.1, and a more general existence result exist. To state this result however, would require the introduction of concepts from functional analysis that go beyond the scope of this thesis. In addition the example problem that we will introduce in section 3.1.1 belongs to the linear-quadratic class of optimization problems.

Theorem 3.2. *Assume that H, V are Hilbert spaces and that Y, Z are Banach spaces. Given vectors $q \in H$ and $g \in Z$, and bounded linear operators $A : Y \to Z$,*

$B : V \to Z$ and $Q : Y \to H$, we can define the linear-quadratic optimization problems as follows:

$$\min_{y \in Y, v \in V} J(y, v) = \frac{1}{2}||Qy - q||_H^2 + \frac{\alpha}{2}||v||_V^2,$$
$$\textit{Subject to: } Ay + Bv = g.$$

If $\alpha > 0$, the above linear-quadratic optimization problem has a unique solution pair $(y, v) \in Y \times V$.

Proof. See [10]. □

3.1.1 Example Problem

To better understand the adjoint approach to gradient evaluation of the reduced objective function, we define a simple optimal control problem with ODE constraints, so that we later can derive its adjoint equation and gradient. The problem will also be used to test and verify the implementation of our method in chapter 7 and 8. In our example both the state $y \in C^1$ and the control $v \in C$ will be functions on an interval $[0, T]$. The specific objective function we consider is:

$$J(y, v) = \frac{1}{2}\int_0^T v(t)^2 dt + \frac{\alpha}{2}(y(T) - y^T)^2 \tag{3.8}$$

The state equation $E(y, v) = 0$ is a linear, first order equation with the control as a source term:

$$\begin{cases} y'(t) = ay(t) + v(t) \quad \text{for} \quad t \in (0, T), \\ y(0) = y_0. \end{cases} \tag{3.9}$$

The state equation of our optimal control problem is uniquely solvable for all continuous controls v, and has solution:

$$y(t) = e^{at}(C(y_0) + \int_0^t e^{-a\tau} v(\tau) d\tau)$$

This means that our example problem (3.8-3.9) is reducible. Since the state equation is linear, and since all terms in the objective function are quadratic, (3.8-3.9) is also an example of a linear-quadratic optimization problem. Theorem 3.2 therefore guaranties a unique minimizer of problem (3.8-3.9).

3.2 The Adjoint Equation and the Gradient

The usual way of finding the minimum (or maximum) value of a function $\hat{J}$, is to solve the equation $\hat{J}'(v) = 0$. Solving this equation usually requires us to be able to evaluate, or have an expression for the derivative of $\hat{J}$. There are different ways to evaluate the gradient of the reduced objective function $\hat{J}(v)$. We will here take the adjoint approach. This strategy leads to an expression for the gradient of the reduced objective function (3.4), which we state in proposition 3.1. The reason it is called the adjoint approach, is that the gradient $\hat{J}'(v)$ depends on the so called adjoint equation. The definition of this equation is found in Proposition 3.1. Proposition 3.1 also include conditions on the operators J and E from definition 3.1, which are necessary for the existence of the gradient of $\hat{J}$. These conditions involve the notion of Fréchet differentiability, which is a generalization of directional derivatives for operators on Banach spaces. For a more precise definition of Fréchet differentiability, we refer to [10].

Proposition 3.1 (Gradient and adjoint of the reduced objective function)**.** *Let $\hat{J}$ be the reduced objective function from definition 3.2, and assume that the state equation operator E and the objective function J are Fréchet differentiable. Assume also that the partial derivative $E_y(y, v) : Y \to Z$ of E with respect to y is a linear and continuously invertible operator. Then the gradient of $\hat{J}$ with respect to the control v is:*

$$\hat{J}'(v) = -E_v(y, v)^* p + J_v(y, v), \tag{3.10}$$

where p is the solution of the adjoint equation:

$$E_y(y, v)^* p = J_y(y, v). \tag{3.11}$$

Proof. If J and E are Fréchet differentiable and if E_y is continuously invertible, the implicit function theorem ensures that $y(v)$ is continuously differentiable. For a more detailed discussion on the implicit function theorem, see [10]. To differentiate $\hat{J}(v) = J(y(v), v)$, we take the total derivative with respect to v D_v of the unreduced objective function:

$$\hat{J}'(v) = D_v J(y(v), v) = y'(v)^* J_y(y, v) + J_v(y, v).$$

The problematic term in the above expression, is $y'(v)^*$, since the function $y(v)$ is implicitly defined through E. We can however find an equation for $y'(v)^*$ if we take the the total derivative of the state equation with respect to v.

$$\begin{aligned} D_v E(y(v), v) = 0 &\Rightarrow E_y(y, v) y'(v) = -E_v(y, v) \\ &\Rightarrow y'(v) = -E_y(y, v)^{-1} E_v(y, v) \\ &\Rightarrow y'(v)^* = -E_v(y, v)^* E_y(y, v)^{-*}. \end{aligned}$$

Instead of inserting $y'(v)^* = -E_v(y,v)^* E_y(y,v)^{-*}$ into our gradient expression, we define the adjoint equation as:

$$E_y(y,v)^* p = J_y(y,v).$$

This now allows us to write up the gradient as follows:

$$\begin{aligned}\hat{J}'(v) &= y'(v)^* J_y(y,v) + J_v(y,v) \\ &= -E_v(y,v)^* E_y(y,v)^{-*} J_y(y,v) + J_v(y,v) \\ &= -E_v(y,v)^* p + J_v(y,v).\end{aligned}$$

□

Expression (3.10) gives us a recipe for evaluating the reduced objective function for a control variable $v \in V$. Typically this evaluation requires us to solve both the state and adjoint equation, and then inserting the adjoint into expression (3.10). To better illustrate how gradient evaluation works let us derive the adjoint equation and the gradient of the problem introduced in section 3.1.1.

3.2.1 Adjoint of the Example Problem

We want to derive the gradient of problem (3.8-3.9). However, before we state the gradient, we write up and derive the adjoint equation.

Proposition 3.2. *The adjoint equation of the problem (3.8-3.9) is:*

$$-p'(t) = ap(t) \tag{3.12}$$

$$p(T) = \alpha(y(T) - y^T) \tag{3.13}$$

Proof. From proposition 3.1 we know that the adjoint equation is $E_y(y,v)^* p = J_y(y,v)$. To find the adjoint equation we therefore need expressions for $E_y(y,v)^*$ and $J_y(y,v)$. In the derivation of these terms we will use the weak formulation of the state equation (3.9). Let $(\cdot,\cdot)$ be the L^2 inner product over $(0,T)$, and then define the operator δ_τ to represent function evaluation at time $t = \tau$ in a L^2-inner product setting. We can then write up the weak formulation of the state equation (3.9) by multiplying it with a test function $\phi(t)$, integrating the result over $(0,T)$ and then moving the derivative from y to ϕ by doing partial integration. The weak formulation of the state equation is then:

$$\text{Find } y \in L^2(0,T) \text{ such that}$$

$$\mathcal{E}[y,\phi] = (y, -(\frac{\partial}{\partial t} + a - \delta_T)\phi) - (y_0\delta_0 + v, \phi) = 0 \quad \forall\ \phi \in C^\infty((0,T)).$$

Instead of finding E_y, we will linearise and adjoint $\mathcal{E}$. We can then derive the weak formulation of the adjoint equation and use this to reconstruct the strong formulation. We linearise $\mathcal{E}$ by differentiating it with respect to y. This yields:

$$\mathcal{E}_y[\cdot, \phi] = (\cdot, (-\frac{\partial}{\partial t} - a + \delta_T)\phi).$$

To find the adjoint of $\mathcal{E}_y$, we need to find a bilinear form $\mathcal{E}_y^*$, such that $\forall v, w \in L^2(0,T)$ the following holds:

$$\mathcal{E}_y[v, w] = \mathcal{E}_y^*[w, v].$$

We achieve this through partial integration:

$$\begin{aligned}\mathcal{E}_y[v, w] = (v, (-\frac{\partial}{\partial t} - a + \delta_T)w) &= \int_0^T -v(t)(w'(t) + aw(t))dt + v(T)w(T) \\ &= \int_0^T w(t)(v'(t) - av(t))dt + v(0)w(0) \\ &= (w, (\frac{\partial}{\partial t} - a + \delta_0)v) =: \mathcal{E}_y^*[w, v].\end{aligned}$$

We now have the left hand side of the adjoint equation. We get the right hand side by differentiating the objective function:

$$\begin{aligned}J_y(y, v) &= \frac{\partial}{\partial y}(\frac{1}{2}\int_0^T v^2 dt + \frac{\alpha}{2}(y(T) - y^T)^2) \\ &= \alpha\delta_T(y(T) - y^T).\end{aligned}$$

The weak formulation of the adjoint equation then is: Find p such that $\mathcal{E}_y^*[p, \psi] = (J_y(y, v), \psi)$, $\forall \psi \in C^\infty((0,T))$. Writing out $\mathcal{E}_y^*[p, \psi] = (J_y(y, v), \psi)$ yields:

$$\int_0^T p(t)\psi'(t) - ap(t)\psi(t)dt + p(0)\psi(0) = \alpha(y(T) - y^T)\psi(T)$$

If we then do partial integration, the equation reads: Find p such that:

$$\int_0^T (-p'(t) - ap(t))\psi(t)dt + p(T)\psi(T) = \alpha(y(T) - y^T)\psi(T) \quad \forall\ \psi \in C^\infty((0,T))$$

Since we can vary ψ arbitrarily, we get the strong formulation:

$$\begin{cases} -p'(t) = ap(t) \\ p(T) = \alpha(y(T) - y^T) \end{cases}$$

□

With the adjoint we can find the gradient of $\hat{J}$. Let us state the result first.

Proposition 3.3. *The gradient of the reduced objective function $\hat{J}$ with respect to v is*

$$\hat{J}'(v) = v + p. \tag{3.14}$$

Proof. Expression (3.10) in proposition 3.1 states that the gradient of the reduced objective function is $\hat{J}'(v) = -E_v(y,v)^* p + J_v(y,v)$. To find the gradient of our example problem we therefore need formulas for $E_v(y,v)$ and $J_v(y,v)$. These terms can be shown to be:

$$\begin{aligned} J_v(y,v) &= v \\ \mathcal{E}_v[\cdot,\phi] &= -(\cdot,\phi). \end{aligned}$$

Here $\mathcal{E}$ is the bilinear form defined in the proof of proposition 3.2. Since $\mathcal{E}_v[\cdot,\phi]$ is symmetric, $\mathcal{E}_v^* = \mathcal{E}_v$, and its strong formulation is $E_v(y,v)^* = -1$. By inserting relevant terms into (3.10), we get the gradient:

$$\hat{J}'(v) = -E_v(y,v)^* p + J_v(y,v) \tag{3.15}$$

$$= p + v. \tag{3.16}$$

□

Evaluating the gradient of our example problem can now be boiled down to the following three steps:

1. Solve the state equation (3.9) for y.
2. Use y to solve the adjoint equation (3.12) for p.
3. Insert p and control v into gradient formula (3.14).

To see why the above procedure is computationally effective, let us compare it with the finite difference approach to evaluating the gradient. Using finite difference we can find an approximation of the directional derivative $(\hat{J}'(v), h)_V$ in direction $h \in V$, by choosing a small $\epsilon > 0$ and setting:

$$(\hat{J}'(v), h)_V \approx \frac{\hat{J}(v + \epsilon h) - \hat{J}(v)}{\epsilon} \tag{3.17}$$

To calculate the above expression, we need to evaluate the objective function at $v + \epsilon h$ and v. Since objective function evaluation requires the solution of the state equation, finding the directional derivative of $\hat{J}$ in a direction h involves solving two ODEs. We are however interested in the gradient of $\hat{J}$, not its directional

derivatives. To find $\hat{J}'(v)$ we calculate (3.17) for all unit vectors in V. This assumes that V is a finite space, which is always true in the discrete case. If we now look at the discrete case and assume that $V = \mathbb{R}^n$ we can write up a recipe for finding $\hat{J}'(v)$ using finite difference. Let e_i denote the i-th unit vector of $\mathbb{R}^n$. $\hat{J}'(v)$ can then be found in the following way:

1. Evaluate $\hat{J}(v)$.
2. Evaluate $\hat{J}(v + \epsilon e_i)$ for $i = 1, ..., n$.
3. Set the i-th component of $\hat{J}'(v)$ to be $\dfrac{\hat{J}(v + \epsilon e_i) - \hat{J}(v)}{\epsilon}$.

To execute the above steps, we need to solve the state equation for $n + 1$ different control variables. In comparison finding $\hat{J}'(v)$ using the adjoint approach only requires us to solve the state and adjoint equations once, independently of the dimension of V. For finite difference the computational cost of one gradient evaluation therefore depends linearly on the number of components in the control variable v, while the computational cost of the adjoint approach is independent of the size of v. In addition, since the the adjoint of the linearised state equation is linear, the adjoint equation is linear. This means that the adjoint equation often is computationally cheaper to solve than the state equation, especially if the state equation is non-linear.

3.2.2 Exact Solution of the Example Problem

It turns out that we can find the exact solution of problem (3.8-3.8) by utilizing the adjoint equation (3.12-3.13) and the gradient of the reduced objective function (3.16). Finding an exact solution to our example problem will be useful in chapter 7 and 8, where we will be testing and verifying different aspects of our algorithm. The derivation of the solution is based on two key observations. The first observation is a relation between the optimal control $\bar{v}$ and the adjoint p, which is a result from the fact that $\hat{J}'(\bar{v}) = 0$ is a necessary condition for $\bar{v}$ being a minimizer of $\hat{J}$. Inserting expression (3.16) into $\hat{J}'(\bar{v}) = 0$ yields:

$$\bar{v}(t) = -p(t). \tag{3.18}$$

The second observation concerns the solution of the adjoint equation (3.12-3.13). Given a state $y(t)$, the solution of the adjoint equation is:

$$p(t) = \alpha(y(T) - y^T)e^{a(T-t)} = \omega e^{-at}. \tag{3.19}$$

Combining observation (3.18) with observation (3.19) suggests that a minimizer $\bar{v}$ of $\hat{J}$ should be on the form:

$$\bar{v}(t) = C_0 e^{-at}. \tag{3.20}$$

It turns out that plugging anstatz (3.20) into the state equation, and then using the resulting state to solve the adjoint equation makes us able to find the solution of our example problem. The solution is stated in proposition 3.4 followed by its derivation.

Proposition 3.4. *Assume $a \neq 0$ and $\alpha > 0$. Then the solution of optimal control problem (3.8-3.8) is:*

$$\bar{v}(t) = \alpha \frac{e^{aT}(y^T - e^{aT}y_0)}{1 + \frac{\alpha e^{aT}}{2a}(e^{aT} - e^{-aT})} e^{-at} \tag{3.21}$$

Proof. We start the proof by writing up the state equation (3.9) with (3.20) as source term:

$$\begin{cases} y'(t) = ay(t) + C_0 e^{-at} \quad \text{for} \quad t \in (0, T), \\ y(0) = y_0. \end{cases}$$

This is a first order linear ODE with solution:

$$y(t) = y_0 e^{at} + \frac{C_0}{2a}(e^{at} - e^{-at}) \tag{3.22}$$

If we insert the state (3.22) into the formula for the adjoint (3.19), we can express the adjoint $p(t)$ in terms of the constant C_0:

$$p(t) = \alpha(y(T) - y^T)e^{a(T-t)} \tag{3.23}$$

$$= \alpha e^{aT}(y_0 e^{aT} + \frac{C_0}{2a}(e^{aT} - e^{-aT}) - y^T)e^{-at} \tag{3.24}$$

The last step is to plug $v(t) = C_0 e^{-at}$ and $p(t)$ from (3.24) into observation (3.18) and then solve for C_0:

$$\begin{aligned} v(t) = -p(t) &\iff C_0 e^{-at} = -\alpha e^{aT}(y_0 e^{aT} + \frac{C_0}{2a}(e^{aT} - e^{-aT}) - y^T)e^{-at} \\ &\iff C_0(1 + \frac{\alpha e^{aT}}{2a}(e^{aT} - e^{-aT})) = \alpha e^{aT}(y^T - y_0 e^{aT}) \\ &\iff C_0 = \alpha \frac{e^{aT}(y^T - e^{aT}y_0)}{1 + \frac{\alpha e^{aT}}{2a}(e^{aT} - e^{-aT})} \end{aligned}$$

Division by $(1+\frac{\alpha e^{aT}}{2a}(e^{aT}-e^{-aT}))$ is always allowed, since $\frac{1}{a}(e^{aT}-e^{-aT}) > 0, \forall a \neq 0$ and $\forall T > 0$. □

3.3 Numerical Solution

To be able to solve optimal control problems numerically, we need to discretize the objective function and the state and adjoint equations. We are mainly interested in time-dependent equations, and one way of discretizing ODEs or PDEs in temporal direction, is to use a finite difference method. Since the objective function includes an integral term, we also need methods for numerical integration. In this section we will only look at first order equations on the following form:

$$\begin{cases} \frac{\partial}{\partial t}y(t) = F(y(t), t), \quad t \in I = [0, T] \\ y(0) = y_0 \end{cases} \tag{3.25}$$

Both the state and adjoint equation of our example problem can be formulated as an equation on form (3.25), and understanding the numerics of (3.25) is therefore sufficient for the purposes of this thesis. Before we introduce numerical methods for solving ODEs and evaluating integrals, we need to explain how we discretize the time domain $I = [0, T]$. We do this by dividing I into n parts of length $\Delta t = \frac{T}{n}$, and then setting $t_k = k\Delta t$. This gives us a sequence $I_{\Delta t} = \{t_k\}_{k=0}^{n}$ as a discrete representation of the interval I. Numerically solving a differential equation for y on $I_{\Delta t}$ means that we try to find $y(t_k)$ for $k = 0, ..., n + 1$. For the rest of this section we let the notation y_k denote evaluating the function y at time t_k.

3.3.1 Discretizing ODEs Using Finite Difference

Finite difference is a tool for approximating derivatives of functions. When we have a discretized domain $I_{\Delta t}$ with time step Δt, the derivative of a function y at point t_k is approximated by:

$$\frac{\partial}{\partial t}y(t_k) \approx \frac{y_k - y_{k-1}}{\Delta t}. \tag{3.26}$$

By exploiting approximation (3.26) we can create methods for solving ODEs. This is done by relating y_k to neighbouring values y_j, $j \neq k$ through the ODE. The most simplistic examples of such finite difference methods are the explicit and implicit Euler methods. We write up these methods applied to (3.25) in definition 3.4 below.

Definition 3.4. *Explicit Euler applied to equation (3.25) means that for $k = 1, ..., n$ the value of y_k is determined by the following formula:*

$$y_k = y_{k-1} + \Delta t F(y_{k-1}, t_{k-1}). \tag{3.27}$$

If one instead uses implicit Euler the expression for y_k is:

$$y_k = y_{k-1} + \Delta t F(y_k, t_k). \tag{3.28}$$

By looking at expression (3.27) and (3.28) we see the origin of the names of the Euler methods. In the formula for implicit Euler, y_k appears on both sides of the equal sign, and is therefore implicitly defined. For the explicit Euler scheme y_k only appears on the left-hand side of expression (3.27), which means y_k is defined explicitly, and hence the name explicit Euler. Another thing to notice about the finite difference schemes in definition 3.4, is that they solve the equation forwardly. This means that given y at time t_K, we can use (3.27) and (3.28) to find y_j for $j > K$. The adjoint equation of optimal control problem with time-dependent DE constraints is however solved backwards in time. We therefore need finite difference schemes for solving ODEs backwards. This is easily achieved by rearranging expression (3.27) and (3.28) in definition 3.4. A backwards solving explicit Euler scheme is found by adjusting the forward solving implicit Euler scheme, while a backwards implicit Euler method is derived by rearranging the forward explicit Euler formula. These modified backwards solving schemes are written up in definition 3.5.

Definition 3.5. *An explicit Euler finite difference scheme for equation (3.25) with initial condition at $t = T$ instead of $t = 0$ yields the following formula for y_k:*

$$y_k = y_{k+1} - \Delta t F(y_{k-1}, t_{k-1}). \tag{3.29}$$

If one instead uses implicit Euler the expression for y_k is:

$$y_k = y_{k+1} - \Delta t F(y_k, t_k). \tag{3.30}$$

We say that both the explicit and implicit Euler methods have an accuracy of order one. To explain what we mean by this, let us assume that we know that the function $\hat{y}$ solves equation (3.25) for a given F, and that $\hat{y}$ is sufficiently smooth. If we then use method (3.27) or (3.28) with some Δt to solve (3.25) numerically, there exists a constant C such that the following error bound between $\hat{y}$ and numerical solution y holds:

$$\max_{k=0,\dots,n} |y_k - \hat{y}(t_k)| \leq C\Delta t \tag{3.31}$$

A more accurate but still simple alternative to the explicit and implicit Euler finite difference methods, is the so called Crank-Nicolson method [40]. We write up this method in a definition:

Definition 3.6. *The Crank-Nicolson finite difference scheme applied to equation (3.25) produces the following formula for y_k:*

$$y_k = y_{k-1} + \frac{\Delta t}{2}(F(y_k, t_k) + F(y_{k-1}, t_{k-1})). \tag{3.32}$$

In a setting where we are solving (3.25) backwards in time, the expression for y_k is changed to:

$$y_k = y_{k+1} - \frac{\Delta t}{2}(F(y_k, t_k) + F(y_{k-1}, t_{k-1})). \tag{3.33}$$

When comparing (3.32) with (3.27) and (3.28) we notice that the formula for y_k in the Crank-Nicolson method is simply the average between the formulas for y_k in the explicit and implicit Euler methods. We improve the accuracy by one order, that is quadratic convergence order, if we use Crank-Nicolson instead of the Euler methods. This means that the bound stated in (3.31) is improved to:

$$\max_{k=0,\ldots,n} |y_k - \hat{y}(t_k)| \leq C\Delta t^2 \tag{3.34}$$

Other more accurate finite difference schemes exist, in particular Runge-Kutta methods, but in this thesis we restrict the usage of finite difference methods to the ones presented in this section.

3.3.2 Numerical Integration

In this subsection we present three simple methods for numerical integration. We need such methods since the objective function in our example problem (3.8) includes an integral. The methods that we present in definition 3.7 are called the left-hand rectangle rule, the right-hand rectangle rule and the trapezoid rule. Their names stem from the geometrical objects used to estimate the area under the function we want to integrate.

Definition 3.7. *We want to estimate the integral $S = \int_0^T v(t)dt$ numerically with a discretized time domain $I_{\Delta t} = \{t_k\}_{k=0}^n$. The left-hand rectangle rule approximates S using the following formula:*

$$S_l = \Delta t \sum_{k=0}^{n-1} v_k \tag{3.35}$$

A slightly different approach to estimating S is the right-hand rectangle rule, defined by a formula similar to (3.35):

$$S_r = \Delta t \sum_{k=1}^{n} v_k \tag{3.36}$$

A third way of approximating S is the trapezoid rule:

$$S_{trap} = \Delta t \frac{v_0 + v_n}{2} + \Delta t \sum_{k=1}^{n-1} v_k \tag{3.37}$$

The rectangle methods in definition 3.7 are of accuracy order one, while the trapezoid rule is of second order. It turns out that the above presented numerical methods are analogue to the three finite difference schemes stated in section 3.3.1. The left- and right-hand rectangle methods are related to the explicit and implicit Euler schemes, while the trapezoid rule is connected with Crank-Nicolson. When making numerical solvers for optimal control problems it therefore makes sense to discretize the differential equation and integral evaluation using methods of the same convergence order.

3.4 Optimization Algorithms

Deriving and solving the adjoint equation gives us a way of evaluating the gradient of optimal control problems with ODE constraints. With the gradient we can solve optimal control problems numerically by using an optimization algorithm. There exists many different optimization algorithms, but here we will only look at line search methods that are useful to us in this thesis. The methods we present are the steepest descent method and the related BFGS and L-BFGS methods.

3.4.1 Line Search Methods and Steepest Descent

Line search methods are algorithms used to solve problems of the type:

$$\min_x f(x), \ f : \mathbb{R}^n \longrightarrow \mathbb{R}$$

All line search methods are iterative methods that starts at an initial guess x^0 and generate a sequence $\{x^k\}$ that hopefully will converge to a solution. The k-th iteration in the algorithm can be described in the following way:

1. *Choose downhill direction* $p_k \in \mathbb{R}^n$
2. *Choose step length* $\alpha_k \in \mathbb{R}$
3. *Set* $x^{k+1} = x^k + \alpha_k p_k$

If f is differentiable, a necessary condition for a point $x^* \in \mathbb{R}^n$ to be a minimizer of f, is that $\nabla f(x^*) = 0$. This optimality condition is used to create a stopping criteria for line search methods in the following way: Given a tolerance $\tau > 0$ and a norm $||\cdot||$ stop the line search iteration when

$$||\nabla f(x^k)|| < \tau. \tag{3.38}$$

We summarize line search methods in algorithm 3.1, which can be used to solve unconstrained optimization problems like the reduced optimal control problem

(3.5). When we in later chapters introduce parallel methods for solving reducible optimization problems, we will use algorithm 3.1 to measure the performance of these methods.

Algorithm 3.1: The line search method

Data: Choose an initial guess x^0 and a tolerance τ
while $||\nabla f(x^k)|| \geq \tau$ **do**
  $x^{k+1} \leftarrow x^k - \alpha_k p_k$;
end

To apply algorithm 3.1, we of course need strategies for obtaining good search directions p_k and step lengths α_k. How one chooses p_k and α_k is what separates different line search methods. Let us start with how to choose a good step length. There are several ways of doing this, but for our purposes the so called Wolfe conditions will suffice. The Wolfe conditions consists of two conditions on f, presented below:

$$f(x^k + \alpha_k p_k) \leq f(x^k) + c_1 \alpha_k \nabla f(x^k) \cdot p_k \tag{3.39}$$

$$\nabla f(x^k + \alpha_k p_k) \cdot p_k \geq c_2 \nabla f(x^k) \cdot p_k \tag{3.40}$$

Here we use constants $0 < c_1 < c_2 < 1$. The first Wolfe condition ensures that the decrease in function value of one steepest descent iteration is proportional to both step length and direction. The second condition is that the gradient of f at $x^k + \alpha_k p_k$, should be less steep than at x^k, and therefore closer to fulfilling the optimality condition (3.38). If we can find a step length that satisfies these conditions we will use it. How to actually find a step length that satisfies the Wolfe conditions is quite involved, and we will therefore not go into this topic any further. For more information on the Wolfe conditions, see: [41,42]. We now look into a couple of line search methods that will be used later in the thesis, starting with steepest descent.

The steepest descent method is a very simple line search method, where the step length p_k is set to the negative gradient direction at point x^k, i.e $p_k = -\nabla f(x^k)$. This gives us the following update for each iteration:

$$x^{k+1} = x^k - \alpha_k \nabla f(x^k) \tag{3.41}$$

The problem with steepest descent is that it converges quite slowly. To understand why let us first write up a definition that characterizes convergence rates.

Definition 3.8. *We say that a sequence $\{x^k\}$ converges linearly to a limit L, if there exists $\epsilon \in (0, 1)$ such that*

$$\lim_{k \to \infty} \frac{||x^{k+1} - L||}{||x^k - L||} = \epsilon.$$

If $\epsilon = 0$ we say that $\{x^k\}$ converges superlinearly to L, while $\epsilon = 1$ is characterized as sublinear convergence. Lastly we say that $\{x^k\}$ converges quadratically towards L, if

$$\lim_{k\to\infty} \frac{||x^{k+1} - L||}{||x^k - L||^2} = \epsilon.$$

With definition 3.8 in mind let us state a theorem from [31] that specifies the convergence rate of the steepest descent method.

Theorem 3.3. *Assume that $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is twice continuously differentiable, that the steepest decent method converge to a point x^*, and that the Hessian of f at this point, $\nabla^2 f(x^*)$ is positive definite. Then the following holds:*

$$f(x^{k+1}) - f(x^*) \leq (\frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1})^2 (f(x^k) - f(x^*))$$

Here $\lambda_1 \leq \cdots \leq \lambda_n$ denotes the eigenvalues of $\nabla^2 f(x^*)$.

Proof. See [31]. □

The bound for $f(x^{k+1}) - f(x^*)$ given in theorem 3.3 corresponds to a linear convergence with $\epsilon = (\frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1})^2$. For badly conditioned Hessians $\nabla^2 f(x^*)$, meaning $\lambda_n >> \lambda_1$, ϵ will approach one, and the convergence rate becomes almost sublinear. In general the linear convergence rate of steepest descent is considered poor, and we need improved algorithms to get faster convergence.

3.4.2 BFGS and L-BFGS

Since steepest descent has slow convergence, one usually uses faster line search methods to solve numerical optimization problems. One alternative is Newtons method. In Newtons method the search direction p_k is found by multiplying the inverse Hessian with the the negative gradient at x^k. This results in the following iteration:

$$x^{k+1} = x^k - \nabla^2 f(x^k)^{-1} \nabla f(x^k) \tag{3.42}$$

As we will see in theorem 3.4, the convergence of the Newton method relies on quite strict conditions on the Hessian $\nabla^2 f(x^k)$, which are not always satisfied. An alternative to the Newton method is so called quasi-Newton methods. Instead of applying $\nabla^2 f(x^k)^{-1}$ to the negative gradient direction, such methods apply approximations of the inverse Hessian to $-\nabla f(x^k)$. The approximate Hessians are constructed for each x^k, using information from previous iterates. One well known

quasi-Newton method is the BFGS method [43–46]. In BFGS the inverse Hessian approximation is calculated by the following recursive formula:

$$H^{k+1} = (\mathbb{1} - \rho_k S_k \cdot Y_k) H^k (\mathbb{1} - \rho_k Y_k \cdot S_k) + S_k \cdot S_k, \tag{3.43}$$

$$S_k = x^{k+1} - x^k, \tag{3.44}$$

$$Y_k = \nabla f(x^{k+1}) - \nabla f(x^k), \tag{3.45}$$

$$\rho_k = \frac{1}{Y_k \cdot S_k}, \tag{3.46}$$

$$H^0 = \beta \mathbb{1}. \tag{3.47}$$

The above formula is designed in such a way, that H^k is symmetric positive definite. This gives us a requirement for the initial inverted Hessian approximation H^0, namely that it also needs to be symmetric positive definite. The usual choice however, is just identity or a multiple β of the identity, where the multiple reflects the scaling of the variables. Strategies of how to chose a scaling factor β is detailed in [47] and [48]. Each line search iteration for BFGS looks like:

$$x^{k+1} = x^k - \alpha H^k \nabla f(x^k) \tag{3.48}$$

In the BFGS method information from all previous iterations is used to create the inverse Hessian approximation for the new iteration. An alternative to this is to limit the number of iterations the recursive formula remembers to only the latest iterations. This variation of the BFGS method is called L-BFGS [49]. The length of the memory need to be chosen in advance, and the typical choice is 10. Two advantages L-BFGS has over BFGS is firstly that it requires less memory storage than BFGS. The second advantage is that limiting the memory of the inverse Hessian approximation accelerates the convergence of BFGS. This is demonstrated in [47] for several different optimization problems. The reason for the improved convergence, is that more recent iterates possess more relevant information for the current Hessian, and by emphasizing the more relevant information, we improve the approximation of the Hessian.

Convergence results for Newton and quasi-Newton methods

Both Newton and quasi-Newton methods converge faster than the steepest descent method. To show this we will include a couple of theorems from [31] concerning this topic. We start with a result on the convergence rate of Newtons method.

Theorem 3.4. *Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable, and that the Hessian $\nabla^2 f(x)$ is Lipschitz continuous in the neighbourhood of a solution x^**

that satisfies $\nabla f(x^*) = 0$ *and that* $\nabla^2 f(x^*)$ *is positive definite. Then the following holds for the Newton iteration 3.42:*

1. *If* x^0 *is suffichently close to* x^**, the sequence of iterates converge to* x^**.*
2. *The rate of convergence of* $\{x^k\}$ *is quadratic*
3. *The sequence of gradient norms* $\{||\nabla f(x^k)||\}$ *converges towards zero quadratically*

Proof. See [31]. □

The quadratic convergence of the Newton iteration is a big improvement in comparison with steepest descent, however theorem 3.4 also highlights one of the problems with the method. Since we need to invert $\nabla^2 f(x^k)$ to find the search direction at x^k, we need an initial x^0 sufficiently close to the actual solution for the iteration to even work. This problem does not arise in BFGS and L-BFGS, since the Hessian approximation is designed to be invertible. Unfortunately though, these quasi-Newton methods does not have the convergence properties of Newtons method, as the next result shows.

Theorem 3.5. *Assume* $f : \mathbb{R}^n \to \mathbb{R}$ *is three times differentiable. Consider then the quasi-Newton iteration* $x^{k+1} = x^k - \alpha_k B_k^{-1} \nabla f(x^k)$*, where* B_k *is an approximation of the Hessian along the search direction* $p_k = -B_k^{-1} \nabla f(x^k)$*, satisfying the condition:*

$$\lim_{K \to \infty} \frac{||(B_k - \nabla^2 f(x^k))p_k||}{||p_k||} = 0$$

If the sequence $\{x^k\}$ *originating from the quasi-Newton iteration converges to a point* x^**, where* $\nabla f(x^*) = 0$ *and* $\nabla^2 f(x^*)$ *is positive definite, the convergence is superlinear.*

Proof. See [31]. □

Even though quasi-Newton methods do not posses the quadratic convergence of the Newton method, superlinear convergence is still better than the linear convergence of steepest descent.

Chapter 4

Parallel in Time ODE Solver Methods

The process of solving time-dependent differential equations in the temporal direction, is an exercise which one would intuitively think is unsuited for parallelization. This is due to the fact that the solution of such equations at every time T depends on the solution at times $t < T$, and it is therefore difficult to partition the solution process into independent tasks that can be solved in parallel. However the Parareal scheme introduced by Lions, Maday and Turinici in [2]is an approach to overcome this limitation. We will however not introduce Parareal as it is described in [2], but rather present an alternative formulation of the algorithm given in [18]. Before we state the Parareal algorithm, let us first explain how we decompose the time domain, and an example equation defined on it.

4.1 Decomposing the Time Interval

The Parareal scheme is used to parallelize differential equations in temporal direction, by decomposing the time interval $I = [0, T]$. An example of a time-dependent differential equation that on this interval is:

$$\begin{cases} \frac{\partial u}{\partial t} + Au = f & \text{for } t \in I \\ u(0) = u_0 \end{cases} \tag{4.1}$$

Decomposing the interval I means dividing the interval into N subintervals $\{I_i = [T_{i-1}, T_i]\}_{i=1}^{N}$, with length $\Delta T = T/N$. We define new equations for each interval:

$$\begin{cases} \frac{\partial u^i}{\partial t} + Au^i = f & \text{for } t \in I^i \\ u^i(T_i) = \lambda_{i-1} \end{cases} \tag{4.2}$$

Here $\lambda_0 = u_0$, while $\{\lambda_i\}_{i=1}^{N-1}$ are virtual intermediate initial conditions. If $\Lambda = (\lambda_0, .., \lambda_{N-1})$ are known values, we can solve the equations independently on each interval. The problem is that the λ's depend on the solution from previous intervals, and need to be calculated by solving the equation. The Parareal scheme is a way of getting around this.

4.2 Parareal

We see that when we decompose the time domain, the original initial value problem (4.1) brakes down to a set of N initial value problems on the form (4.2). The idea of [18] is then first to define a fine solution operator $\mathbf{F}_{\Delta T}$, which when given an initial condition λ_{i-1} at time T_{i-1}, evolves λ_i, using a fine scheme applied to the i-th equation (4.2), from time T_i to T_{i+1}. Meaning:

$$\hat{\lambda}_i = u^i(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1})$$

We name $\mathbf{F}_{\Delta T}$ the fine propagator, and note that letting $\hat{\lambda}_1 = \mathbf{F}_{\Delta T}(u_0)$, and then applying $\mathbf{F}_{\Delta T}$ sequentially to $\hat{\lambda}_i$, is the same as solving (4.1), using the underlying numerical method of the fine propagator. However, we intend to use $\mathbf{F}_{\Delta T}$ simultaneously on a given set of initial values $\Lambda = (\lambda_0 = u_0, \lambda_1, ..., \lambda_{N-1})$, and not sequentially. Since we also want $\hat{\lambda}_i$ to be as close as possible to λ_i for $i = 1, ..., N-1$, we define a coarse propagator $\mathbf{G}_{\Delta T}$, and use this operator to predict the Λ values. The predictions are made by sequentially applying the coarse propagator to the system (4.2). This means:

$$\lambda_i^0 = \mathbf{G}_{\Delta T}(\lambda_{i-1}^0), \quad i = 1, ..., N-1, \tag{4.3}$$
$$\lambda_0^0 = u_0, \tag{4.4}$$

where the superscript denotes the Parareal iteration. Once we have these predicted initial values, we can apply the fine propagator on all N equations (4.2) simultaneously, and then use the difference between our fine solution and coarse solution $\delta_{i-1}^0 = \mathbf{F}_{\Delta T}(\lambda_{i-1}^0) - \mathbf{G}_{\Delta T}(\lambda_{i-1}^0)$ at time T_i to correct λ_i^0. The correction for time T_i, is done by using the coarse propagator on the already corrected λ_{i-1}^1, and then add the difference δ_{i-1}^0 to $\mathbf{G}_{\Delta T}(\lambda_{i-1}^1)$. When this sequential process is done, we have a new set of initial conditions λ_i^1, $i = 1, ..., N-1$, which means that we can redo the correction procedure in an iterative fashion. The prediction-correction formulation of Parareal can then be written up as the following iteration:

$$\lambda_i^{k+1} = \mathbf{G}_{\Delta T}(\lambda_{i-1}^{k+1}) + \mathbf{F}_{\Delta T}(\lambda_{i-1}^k) - \mathbf{G}_{\Delta T}(\lambda_{i-1}^k), \quad i = 1, ..., N-1 \tag{4.5}$$
$$\lambda_0^k = u_0 \tag{4.6}$$

Updating our initial conditions Λ^k from iteration k to iteration $k+1$, requires N fine propagations, which we can do in parallel, and N coarse propagations, that we need to do sequentially. We can now write up a simple algorithm for doing K steps of Parareal.

Algorithm 4.1: K steps of the Parareal algorithm

$\lambda_0^0 \leftarrow u_0$;
for $i = 1, ..., N-1$ **do**
  $\lambda_i^0 \leftarrow \mathbf{G}_{\Delta T}(\lambda_{i-1}^0)$;
end
for $k = 1, ..., K$ **do**
  $\lambda_0^k \leftarrow u_0$;
  $\hat{\lambda_i^k} \leftarrow \mathbf{F}_{\Delta T}(\lambda_{i-1}^{k-1})$// `In parallel`;
  for $i = 1, ..., N-1$ **do**
    $\lambda_i^k \leftarrow \mathbf{G}_{\Delta T}(\lambda_{i-1}^k) + \hat{\lambda_i^k} - \lambda_i^{k-1}$;
  end
end

In algorithm 4.1 we do K iterations of the Parareal algorithm, where K is a prechosen number. If one wanted to construct an actual Parareal algorithm, the iteration should instead terminate, when a certain stopping criteria is met. After N iterations the Parareal algorithm produces the same solution as the fine sequential solver. We therefore need a stopping criteria that ensures that the Parareal solution is sufficiently accurate, while also terminating before N iterations are done. A discussion on suitable stopping criteria for the Parareal algorithm can be found in [29].

Another important component of Parareal, that affects its performance, is the coarse propagator $\mathbf{G}_{\Delta T}$. If $\mathbf{F}_{\Delta T}$ is based on a finite difference scheme with time step Δt, one natural choice for $\mathbf{G}_{\Delta T}$, would be to use the same scheme as the fine propagator with bigger time step. One must be careful though, since such schemes might be unstable for big time steps. To give an example of a coarse propagator, let us consider a $\mathbf{G}_{\Delta T}$ that is based on the implicit Euler scheme with time step ΔT. Using this scheme for our coarse propagator in the context of problem (4.1), would mean that we find $\mathbf{G}_{\Delta T}(\lambda_i)$ by solving (4.7) for $\mathbf{G}_{\Delta T}(\lambda_i)$.

$$\frac{\mathbf{G}_{\Delta T}(\lambda_i) - \lambda_i}{\Delta T} + A\mathbf{G}_{\Delta T}(\lambda_i) = f(T_i) \tag{4.7}$$

In the above example we just used a coarse discretization of our problem (4.1) to define the coarse propagator. There are however a lot of other ways to construct $\mathbf{G}_{\Delta T}$. In [18] for example, they create the coarse propagator by simplifying the

physics of the problem the authors are trying to model. The underlying numerical method of the coarse propagator should in any case be chosen so that the computational cost of $\mathbf{G}_{\Delta T}$ is negligible in comparison to the cost of $\mathbf{F}_{\Delta T}$.

4.3 Algebraic Formulation

In [11] an algebraic reformulation of (4.5) is presented. The setting in [11] is slightly different than the one we had in section 4.2, since they are trying to solve an optimal control problem with differential equation constraints, rather than to just solve a differential equation. Luckily for us the problem they are looking at is very much connected to that of solving the time decomposed differential equation system. The problem they solve follows below:

$$\begin{aligned} &\min_{\Lambda} \hat{J}(\Lambda) = \sum_{i=1}^{N-1} ||u^i(T_i) - \lambda_i||^2 \\ &\text{Subject to } \; u^i(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1}) \; i = 1, ..., N \end{aligned}$$

In the above optimal control problem the $\mathbf{F}_{\Delta T}$ is the fine propagator from the previous section, and u and Λ is also as defined in section 4.2. What we immediately notice, is that we can find the solution of the above problem by setting $J(\Lambda) = 0$, which gives us the solution $\lambda_i = u^i(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1})$. The authors of [11] then write this system on matrix form as:

$$\begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{F}_{\Delta T} & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{F}_{\Delta T} & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{F}_{\Delta T} & \mathbb{1} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \cdots \\ \lambda_{N-1} \end{bmatrix} = \begin{bmatrix} u_0 \\ 0 \\ \cdots \\ 0 \end{bmatrix} \tag{4.8}$$

Or with notation:

$$M \; \Lambda \; = \; H. \quad \text{With } M \in \mathbb{R}^{N \times N}, H \in \mathbb{R}^N \text{ given by (4.8).} \tag{4.9}$$

We can solve system (4.8) by sequentially applying the fine propagator, but we again want to use the coarse propagator, so that we can run the fine propagator in parallel. We first define the coarse equivalent to M as:

$$\bar{M} = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{G}_{\Delta T} & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{G}_{\Delta T} & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{G}_{\Delta T} & \mathbb{1} \end{bmatrix} \tag{4.10}$$

Using $\bar{M}$, we can write up what turns out to be the Parareal iteration (4.5) in Matrix notation:

$$\Lambda^{k+1} = \Lambda^k + \bar{M}^{-1}(H - M\Lambda^k) \tag{4.11}$$

Looking at the (4.11), we recognise the Parareal iteration as a preconditioned fix point iteration, where $\bar{M}^{-1}$ is the preconditioner.

4.4 Convergence of Parareal

In this section we look at some of the convergence properties of the Parareal algorithm given in the literature. The first publication on Parareal [2] studied the convergence in context of the following equation:

$$\frac{\partial}{\partial t}y(t) = ay(t), \quad t \in [0, T], \quad y(0) = y_0 \tag{4.12}$$

We state their findings in the proposition below:

Proposition 4.1. *Let us decompose $I = [0, T]$ into N subintervals of length $\Delta T = \frac{T}{N}$, and then let $\mathbf{F}_{\Delta T}$ and $\mathbf{G}_{\Delta T}$ be the fine and coarse propagators for equation (4.12). If $\mathbf{G}_{\Delta T}(\omega)$ is evaluated using the implicit Euler scheme (4.7), there exist for all integers k a constant c_k such that the error between the k-th iterate of the Parareal algorithm (4.5) and the exact solution of (4.12) y is bounded in the following way:*

$$\forall\ i, 0 \leq i \leq N-1 \quad |\lambda_i^k - y(T_i)| + \max_{t \in [T_i, T_{i+1}]} |y_{i+1}^k(t) - y(t)| \leq c_k \Delta T^{k+1} \tag{4.13}$$

It is important to note that the error bound given in proposition 4.1 only holds for fixed ks, since the constant c_k grows with k. This means that if we do k iterations of Parareal, the algorithm converges to the fine numerical solution, when ΔT goes to zero at a rate of $\mathcal{O}(\Delta T^{k+1})$. We can therefore say that k iterations behaves like a numerical method of order $k+1$. In [2], the authors used a first order implicit Euler scheme for their coarse propagator. It turns out that if one instead uses a coarse scheme of order p, the convergence bound (4.13) after k iterations is improved to $\mathcal{O}(\Delta T^{p(k+1)})$. This was shown in [21], where the bounds were derived for more general equations.

The case where we let ΔT be fixed, and look at convergence when we increase k, is analysed in [23]. Here the authors again investigate the convergence of the equation (4.12). They found that the convergence was superlinear in k for bounded time intervals $[0, T]$, and linear for unbounded time interval.

To demonstrate the Parareal algorithm, we will try to verify proposition 4.1 for the following linear ODE:

$$\frac{\partial}{\partial t}y(t) = \cos(2\pi t)y(t), \quad t \in [0,4], y(0) = 3.52 \tag{4.14}$$

This is a simple separable equation with solution $y_e(t) = y_0 e^{-\frac{\sin(2\pi t)}{2\pi}}$. To test the Parareal algorithm we choose a fine solver that discretizes (4.14) using the second order Crank-Nicolson finite difference scheme [40], while we base the coarse solver on a first order implicit Euler scheme. The experimental setup, is to do "zero", one, two and three iterations of Parareal on different time decompositions, and then check if we get the convergence rate proposed in proposition 4.14. The error between the exact solution y_e and the solution y we get from k Parareal iterations is measured in the max-norm, and we use $\Delta t = 10^{-6}$ as small time step for the fine discretization. We calculate the convergence rate by comparing the error at different coarse time step sizes $\Delta T_1 > \Delta T_2$ using formula:

$$\text{rate} = \frac{\log(\frac{||y_{\Delta T_2} - y_e||_{l_\infty}}{||y_{\Delta T_1} - y_e||_{l_\infty}})}{\log(\frac{\Delta T_2}{\Delta T_2})}. \tag{4.15}$$

The results can be found in tables 4.1 to 4.4. Plots of the results of one Parareal iteration applied to large ΔT values are also added in Figure 4.1. In table 4.1 we observe a convergence rate of one, which is in line with proposition 4.1. For table 4.2 and 4.3, we see that when the coarse time step ΔT approaches zero, the convergence rate goes to two and three. This is again what we expect in light of proposition 4.1. In the last table the results are obtained by applying three iterations of Parareal. Proposition 4.1 then suggests a convergence rate of $\mathcal{O}(\Delta T^4)$. We do however not observe this in table 4.4, since the error decreases only at a rate of 2.9911, between $N = 1000$ and $N = 2000$. The likely cause of this, is that we when doing three iterations of Parareal using $N = 1000$ and $N = 2000$ approach the underlying error of the fine propagator, and the Parareal algorithm can not outperform the error of the fine scheme.

To illustrate how the Parareal algorithm works, we did a second experiment on equation 4.14. The results of this experiment is presented in figure 4.1. This figure shows the result of applying one iteration of Parareal to our example problem, using $N = 6, 12, 24$ decompositions of the time interval. We observe that the solution improves when N is increased and consequentially ΔT is decreased. However, for all decompositions the results are not good, evident by the noticeable jumps between subintervals, and more iterations of Parareal are required to get

a satisfactory solution. Another observation about the plots in figure 4.1, is that the numerical solution is exact (in the sense of the fine scheme) for the two first decomposed intervals. This aspect is especially noticeable for $N = 6$. The solution being exact for the $k+1$ first subintervals after k iterations of Parareal is a known property of the algorithm. This property also complicates the implementation of Parareal, since the $k+1$ first processes will become idle after k iterations. Algorithm 4.1, though simple and usable, is not an optimal implementation of Parareal, in part due to the above discussed property.

Table 4.1: Results for initial coarse and fine solver applied to equation 4.14. The first column (N) represents the number of decomposed subintervals, and the second column is the corresponding coarse time step size $\Delta T = \frac{T}{N}$. The third column measures the maximal absolute difference between the exact solution y_e and the numerical solution y from "zero" steps of Parareal. Using these errors we can find the convergence rate with formula (4.15). We observe a convergence rate of 1, which is as expected for an implicit Euler scheme.

N	ΔT	$\|\|y - y_e\|\|_{l_\infty}$	rate
40	0.100	0.802	–
50	0.090	0.628	1.09
100	0.040	0.298	1.07
200	0.020	0.145	1.04
500	0.008	0.057	1.01
1000	0.004	0.028	1.00
2000	0.002	0.014	1.00

Table 4.2: Convergence results for one iteration of Parareal. The columns are the same as in table 4.1. We see a quadratic convergence rate, which is consistent with proposition 4.1.

N	ΔT	$\|\|y - y_e\|\|_{l_\infty}$	rate
40	0.100	3.65e-2	–
50	0.080	2.79e-2	1.20
100	0.040	8.87e-3	1.65
200	0.020	2.42e-3	1.87
500	0.008	4.06e-4	1.95
1000	0.004	1.03e-4	1.97
2000	0.002	2.6e-5	1.99

Table 4.3: Convergence results for two iterations of Parareal. From proposition 4.1 we expect a convergence rate of three, and we see that the rate of convergence approaches 3 when ΔT goes to zero.

N	ΔT	$\|\|y - y_e\|\|_{l_\infty}$	rate
40	0.100	3.75e-03	–
50	0.080	1.38e-03	4.478
100	0.040	1.09e-04	3.66
200	0.020	2.34e-05	2.21
500	0.008	1.85e-06	2.77
1000	0.004	2.44e-07	2.91
2000	0.002	3.07e-08	2.99

Table 4.4: Convergence results for three iterations of Parareal. Doing three iterations of Parareal using a first order coarse scheme, we expect that the rate of convergence converges to four. We see that the convergence rate does not behave as we would expect from proposition 4.1. The most likely explanation for this is that the error of the Parareal algorithm approaches the numerical error of the fine scheme.

N	ΔT	$\|\|y - y_e\|\|_{l_\infty}$	rate
40	0.100	2.57e-04	–
50	0.080	7.81e-05	5.34
100	0.040	1.48e-06	5.71
200	0.020	1.34e-07	3.46
500	0.008	6.04e-09	3.38
1000	0.004	4.43e-10	3.76
2000	0.002	5.57e-11	2.99

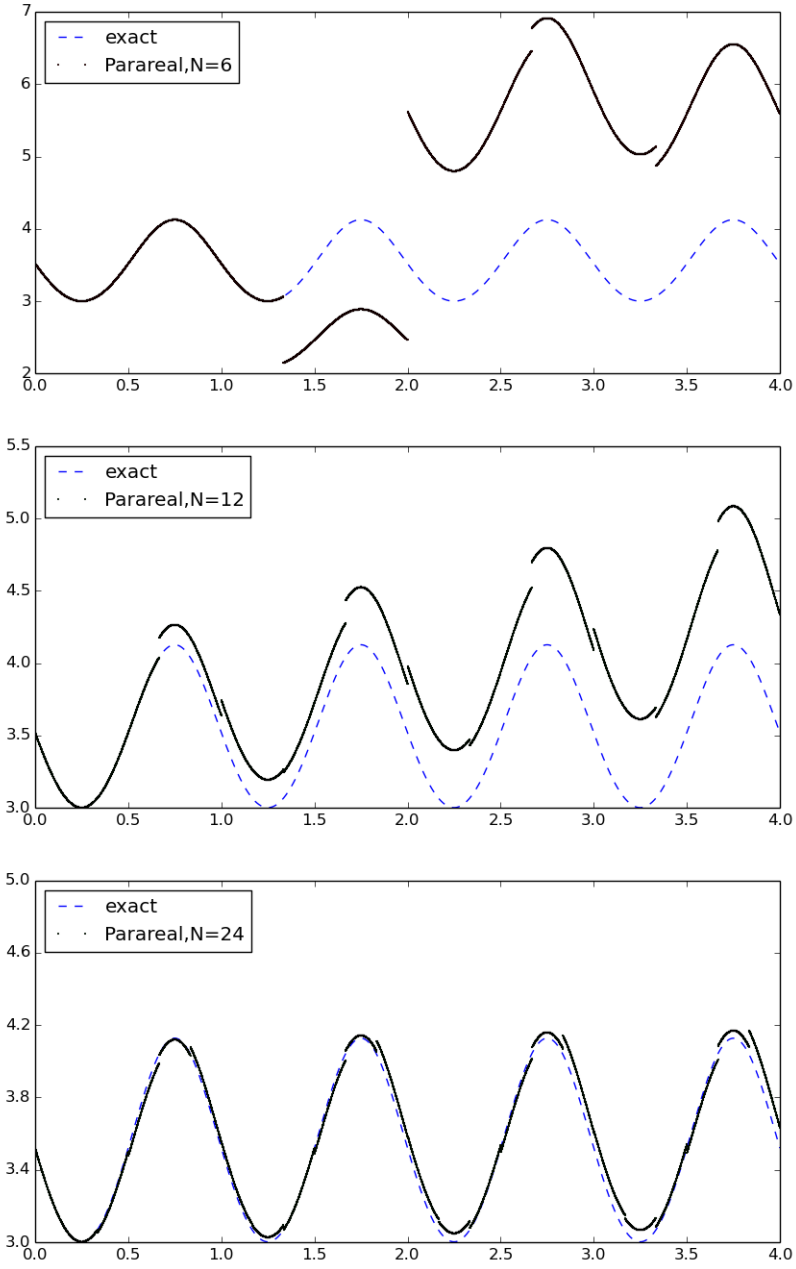


Figure 4.1: The result of 1 iteration of the Parareal algorithm on equation (4.14). The equation is solved using a fine propagator based on the Crank-Nicholsen with resolution $\Delta t = \frac{4}{10^4}$ and a coarse propagator based on implicit Euler. We solve (4.14) using three different time decompositions, $N = 6, 12, 24$, which translates to coarse time steps $\Delta T = \frac{2}{3}, \frac{1}{3}, \frac{1}{6}$ used for the coarse propagator.

Chapter 5

Parareal-Based BFGS Preconditioner

In the previous chapter we saw that the Parareal scheme allows us to parallelize time-dependent differential equations in their temporal direction. In this chapter we will present a parallel in time method for optimal control problems with time-dependent differential equation constraints.

This chapter consists of four sections. In the first section we decompose the time domain as we did in section 4.1, only now in the context of control problems with time-dependent DE constraints. Decomposing the time interval leads to a reformulation of the control problem that includes extra constraints on the state equation. How to handle these new constraints are dealt with in section 5.2. To solve this constrained problem, we use the same approach as [11] namely the penalty method. The penalty method is a simplified version of the augmented Lagrangian approach used in [12] for parallel in time 4d variational data assimilation. We demonstrate the use of the penalty method by revisiting the example problem from section 3.1.1.

In the third section a Parareal based preconditioner to be used in the optimization algorithms solving the optimal control problems is presented. This preconditioner originally proposed in [11] is derived using ideas from subsection 4.3 and we will in chapter 8 see that it is crucial for the parallel in time algorithm to obtain any meaningful speedup. In the fourth and last section we propose a parallel in time method based on the penalty framework of section 5.2 and the BFGS optimization algorithm. The Parareal-based preconditioner from section 5.3 is used as an initial inverted Hessian approximation in the BFGS algorithm.

5.1 Optimal Control with Time-Dependent ODE Constraints on a Decomposed Time Interval

We want to solve reducible optimization problems of type (3.1-3.2), where the state equation constraint $E(y(t), v, y_0) = 0$ is time-dependent and solved on the interval $I = [0, T]$, with initial condition y_0. To introduce parallelism to our optimal control problem, we need to decompose the time domain and the state equation as we did in chapter 4.

Definition 5.1 (Decomposed state equation)**.** *Let $0 = T_0 < T_1 < \cdots < T_{N-1} < T_N = T$ and define the i-th decomposed subinterval to be $I_i = [T_{i-1}, T_i]$. We then introduce $N-1$ intermediate initial conditions $\Lambda = (\lambda_1, ..., \lambda_{N-1})$, and set $\lambda_0 = y_0$. Using these intermediate initial conditions we are able to define N decomposed state equations:*

$$E^i(y_i(t), v, \lambda_{i-1}) = 0 \quad t \in I_i. \tag{5.1}$$

Solving the state equation $E(y(t), v, \Lambda) = 0$ on the entire time domain, then means solving the N decomposed equations (5.1) for y_i, and setting the state $y(t)$ to be:

$$y(t) = \begin{cases} y_1(t) & t \in [T_0, T_1] \\ y_2(t) & t \in (T_1, T_2] \\ \cdots & \cdots \\ y_N(t) & t \in (T_{N-1}, T_N] \end{cases} \tag{5.2}$$

Using the decomposed time interval, state and state equation of definition 5.1, we can define the decomposed optimal control problem. Since we want to solve the decomposed state equations simultaneously, the intermediate initial conditions Λ will be added to the optimization problem as additional control variables. Because these variables are artificially introduced by us, we refer to Λ as the virtual control, while we call the original control v the real control. We also want the state to be continuous, so we need to introduce new constraints on the problem that enforces the continuity of $y(t)$. These new constraints are written up along with the decomposed reformulation of the optimal control problem in definition 5.2.

Definition 5.2 (Decomposed optimal control problem)**.** *Let Y, V, Z be defined as in definition 3.1. The decomposed optimal control problem with time-dependent DE constraint is the following minimization problem:*

$$\min_{y \in Y, v \in V, \Lambda} J(y(t), v, \Lambda), \tag{5.3}$$

$$\text{subject to: } E(y(t), v, \Lambda) = 0, \quad t \in [0, T]. \tag{5.4}$$

To enforce the continuity of $y(t)$ between subintervals, we introduce extra constraints:

$$y_i(T_i) = y_{i+1}(T_i) = \lambda_i \quad i = 1, .., N-1. \tag{5.5}$$

If all the decomposed state equations $E^i(y_i, v, \lambda_{i-1}) = 0$ are uniquely solvable for all control variables $v \in V$, we can reduce the decomposed optimization problem from definition 5.2. We write up the reduced version of problem (5.3-5.5) in the next definition.

Definition 5.3 (Decomposed and reduced optimal control problem)**.** *Consider problem (5.3-5.5). We assume that this problem is reducible, and can therefore define $\hat{J} : V \to \mathbb{R}$ as:*

$$\hat{J}(v, \Lambda) = J(y(v, \Lambda)(t), v, \Lambda)$$

The decomposed and reduced optimal control problem with time-dependent differential equation constraints is then the following constrained minimization problem:

$$\min_{v \in V, \Lambda} \hat{J}(v, \Lambda), \tag{5.6}$$

$$y_i(T_i) = \lambda_i, \quad i = 1, ..., N-1. \tag{5.7}$$

Unlike the undecomposed case, the reduced and decomposed optimal control problem is not unconstrained. A strategy for handling the extra constraints (5.7) is discussed in the next section.

5.2 The Penalty Method

To solve the constrained problem (5.6-5.7), we will use the penalty method [31], which transforms constrained problems into a series of unconstrained problems. This is done by moving the constraints into the objective function $J(v, \Lambda)$. For each constraint a term is added to J, which is positive for variables that does not satisfy the constraint, but zero if it does. The penalization of the constraints can be done in different ways, but we will restrict ourself to the quadratic penalty method, where the the terms penalizing the constraints are quadratic.

Definition 5.4 (Quadratic penalty method)**.** *Consider the constrained optimization problem:*

$$\min_x f(x) \quad \textit{subject to: } c_i(x) = 0, \quad i = 1, ..., N, \tag{5.8}$$

Given a penalty parameter $\mu > 0$, the quadratic penalty method defines an altered functional $f_\mu : X \to \mathbb{R}$ related to the functional of problem (5.8).

$$f_\mu(x) = f(x) + \frac{\mu}{2} \sum_{i=1}^{N} c_i(x)^2 \tag{5.9}$$

Minimizing f_μ is an unconstrained optimization problem. If we now instead consider our decomposed optimization problem (5.6-5.7), we can write up its penalized objective function $\hat{J}_\mu$ as:

$$\hat{J}_\mu(v, \Lambda) = \hat{J}(v) + \frac{\mu}{2} \sum_{i=1}^{N-1} (y_i(T_i) - \lambda_i)^2. \tag{5.10}$$

The idea of the penalty method is that the minimizer of (5.9) should approach a feasible minimizer of (5.8) when we increase the penalty parameter μ. Since the penalized problem can be difficult to solve for large μ values, the usual approach for solving constrained problems with the penalty method, is to minimize the penalized objective function for an increasing sequence of penalty parameters μ. We write up the general algorithmic framework of the penalty method applied to problem (5.6-5.7) in algorithm 5.1.

Algorithm 5.1: Penalty method

Data: Choose $\mu_0, \tau_0 > 0$, and some initial control (v^0, Λ^0)
for $k = 1, 2, ...$ **do**
  Find (v^k, Λ^k) s.t. $\| \nabla \hat{J}_{\mu_{k-1}}(v^k, \Lambda^k) \| < \tau_{k-1}$;
  if *STOP CRITERION satisfied* **then**
    Stop algorithm;
  else
    Choose new $\tau_k \in (0, \tau_{k-1})$ and $\mu_k \in (\mu_{k-1}, \infty)$;
  end
end

If we want to use the penalty method, we need to know if the framework presented in algorithm 5.1 is consistent. We say that the penalty method is consistent, if for a given global minimizer (v, Λ) of $\hat{J}$, the iterates (v^k, Λ^k) produced by framework 5.1 converges to (v, Λ), meaning:

$$\lim_{k \to \infty} (v^k, \Lambda^k) = (v, \Lambda).$$

From [31] we get a result that deals with this:

Theorem 5.1. *Assume that $\forall k$, (v^k, Λ^k) is the exact global minimizer of J_{μ_k} in context of the framework in algorithm 5.1. Then each limit point of the sequence $\{(v^k, \Lambda^k)\}$ is a solution of the problem (5.6-5.7).*

Proof. [31] □

Theorem 5.1 tells us that the penalty method of algorithm 5.1 is consistent, if we for all k can find an exact global minimizer of J_{μ_k}. In practice, finding an exact minimizer of the penalized objective function is not always achievable. Another theorem from [31] indicates what to expect if we are unable to find a global minimizer of J_{μ_k} for all k's.

Theorem 5.2. *If the tolerance τ_k and the penalty parameter μ_k of the method in algorithm 5.1 satisfy*

$$\lim_{k\to\infty} \tau_k = 0, \quad \textit{and} \quad \lim_{k\to\infty} \mu_k = \infty.$$

the limit point of the the sequence $\{(v^k, \Lambda^k)\}$ will be a feasible point. In context of problem (5.6-5.7), this means that:

$$\forall i = 1, ..., N-1 \quad \lim_{k\to\infty} (y_i^k(T_i) - \lambda_i^k) = 0$$

Proof. [31] □

Theorem 5.1 provides the theoretical consistency of the method in algorithm 5.1, this consistency is based on the assumption that we are able to minimize the penalized objective function for ever increasing penalty parameters μ. Theorem 5.2 shows that the quadratic penalty method at least will produce a feasible control solution. In section 7.4 we will try to verify the consistency of algorithm 5.1 for the example problem, and we will see that theorem 5.1 and 5.2 are crucial for understanding the results we get.

We will now look at the most important part of the framework in algorithm 5.1, namely the optimization of the penalized objective function $\hat{J}_\mu$. To be able to optimize $\hat{J}_\mu$, we need its gradient, and we will therefore in the next section derive the gradient of the general penalized objective function (5.10) using the adjoint approach. We will also find an expression for $\hat{J}'_\mu(v, \Lambda)$ for the example problem (3.8-3.9). In section 5.3 we will present a Parareal-based preconditioner, that we will use to improve the optimization of $\hat{J}_\mu$.

5.2.1 The Gradient of the Penalized Objective Function

We have introduced the penalized objective function (5.10), that depends on both the real and virtual control, and we now want to evaluate its gradient. We again take the adjoint approach as we did in section 3.2, and the expression for $\hat{J}'_\mu(v,\Lambda)$ belonging to the general optimization problem (5.3-5.4) is given in proposition 5.1.

Proposition 5.1 (Gradient of the penalized objective function)**.** *Let $\hat{J}_\mu$ be the penalized objective function (5.10). With similar assumptions as in proposition 3.1, the gradient of $\hat{J}_\mu$ is as follows:*

$$\hat{J}'_\mu(v,\Lambda) = -(E_v(y(t),v,\Lambda)^* + E_\Lambda(y(t),v,\Lambda)^*)p(t) + (\frac{\partial}{\partial v} + \frac{\partial}{\partial \Lambda})J_\mu(y,v,\Lambda). \tag{5.11}$$

The decomposed adjoint $p(t)$ is defined on $I = [0,T]$ as:

$$p(t) = \begin{cases} p_1(t) & t \in [T_0, T_1] \\ p_2(t) & t \in (T_1, T_2] \\ \cdots & \cdots \\ p_N(t) & t \in (T_{N-1}, T_N] \end{cases} \tag{5.12}$$

where the p_i's are the solutions of the decomposed adjoint equations:

$$E^i_{y_i}(y_i(t),v,\Lambda)^* p_i(t) = \frac{\partial}{\partial y_i} J_\mu(y_i,v,\Lambda), \quad t \in [T_{i-1}, T_i]. \tag{5.13}$$

Proof. Same reasoning as in proposition 3.1. □

Notice that the state equation $E(y(t),v,\Lambda) = 0$ consists of several equations defined separately on each of the decomposed subintervals. The result is that the adjoint equation also consists of several equations defined on each interval. To see this clearly we will derive the adjoint and the gradient for the example problem (3.8-3.9).

5.2.2 Deriving the Adjoint for the Example Problem

Before we derive the adjoint equation of the decomposed example problem (3.8-3.9) we need to write up the decomposed state equation and the penalized objective function. We start by decomposing the interval $[0,T]$ into N subintervals $\{[T_{i-1}, T_i]\}_{i=1}^N$. We can then define the decomposed state equation on each interval:

$$\begin{cases} \frac{\partial}{\partial t} y_i(t) = a y_i(t) + v(t) & t \in (T_{i-1}, T_i) \\ y_i(T_{i-1}) = \lambda_{i-1} \end{cases} \tag{5.14}$$

We get the reduced penalized objective function by adding the the penalty terms to the unpenalized objective function (3.8):

$$\hat{J}_\mu(v,\Lambda) = \frac{1}{2}\int_0^T v(t)^2 dt + \frac{\alpha}{2}(y(T)-y^T)^2 + \frac{\mu}{2}\sum_{i=1}^{N-1}(y_i(T_i)-\lambda_i)^2 \tag{5.15}$$

Having formulated the penalized objective function, we are now ready to write up its gradient. The gradient of (5.15) is given in proposition 5.3, but since the gradient depends on the decomposed adjoint equations, we write up these first.

Proposition 5.2. *The decomposed adjoint equation of problem (3.8-3.9) on interval* $[T_{N-1}, T_N]$ *is:*

$$\begin{cases} -\frac{\partial}{\partial t}p_N = ap_N \\ p_N(T_N) = \alpha(y_N(T_N)-y_T) \end{cases} \tag{5.16}$$

On $[T_{i-1}, T_i]$ *the decomposed adjoint equations are:*

$$\begin{cases} -\frac{\partial}{\partial t}p_i = ap_i \\ p_i(T_i) = \mu(y_i(T_i)-\lambda_i) \end{cases} \tag{5.17}$$

Proof. The decomposed adjoint equation on interval $I_i = [T_{i-1}, T_i]$ is defined by the equation $E^i_{y_i}(y_i, v, \Lambda)^* p_i = \frac{\partial}{\partial y_i} J_\mu(y_i, v, \Lambda)$. This means that to derive it, we need expressions for $E^i_{y_i}(y_i, v, \Lambda)^*$ and $\frac{\partial}{\partial y_i} J_\mu(y_i, v, \Lambda)$. We will use the same approach as in the proof of proposition 3.2, meaning that we will use the weak formulation of the decomposed state equations to derive the adjoint. If we let $(\cdot,\cdot)_i$ denote the L^2 inner product on (T_{i-1}, T_i), we can define a bilinear form $\mathcal{E}^i$ as:

$$\mathcal{E}^i[y_i, \phi] = (y_i, (-\frac{\partial}{\partial t} - a + \delta_{T_i})\phi)_i - (v + \delta_{T_{i-1}}\lambda_{i-1}, \phi)_i$$

The weak formulation of the i-th state equation then reads:

$$\text{Find } y_i \text{ s.t.} \quad \mathcal{E}^i[y_i, \phi] = 0 \quad \forall \phi \in C^\infty((T_{i-1}, T_i)).$$

Arguing similarly as we did in the proof of proposition 3.2, we find the linearised adjoint of $\mathcal{E}^i$ to be:

$$\mathcal{E}^i_{y_i}[\cdot, \psi]^* = (\cdot, (\frac{\partial}{\partial t} - a + \delta_{T_{i-1}})\psi)_i$$

The weak formulation of the i-th adjoint equation is then: Find p_i such that $\mathcal{E}^i_{y_i}[p_i, \psi]^* = (\frac{\partial}{\partial y_i} J_\mu(y_i, v, \Lambda), \psi)_i$, $\forall \psi \in C^\infty$. If we can find an expression for $\frac{\partial}{\partial y_i} J_\mu$,

we will have the weak adjoint equation. It turns out that we are able to decompose the penalized objective function into N functions J_μ^i defined as:

$$J_\mu^i(y_i, v, \Lambda) = \int_{T_{i-1}}^{T_i} v(t)^2 dt + \frac{\mu}{2}(y_i(T_i) - \lambda_i)^2, \quad \text{for } i = 1, ..., N-1, \text{ and}$$
$$J_\mu^N(y_N, v, \Lambda) = \int_{T_{N-1}}^{T_N} v(t)^2 dt + \frac{\alpha}{2}(y_N(T_N) - y^T)^2.$$

We notice that the the sum of these decomposed objective functions equals the penalized objective function (5.15). What we also see is that J_μ^i only depends on the i-th state equation. This means that $\frac{\partial}{\partial y_i} J_\mu = \frac{\partial}{\partial y_i} J_\mu^i$.

$$\frac{\partial}{\partial y} J_\mu^i(y_i, v, \Lambda) = \mu \delta_{T_i}(y_i(T_i) - \lambda_i), \quad i = 1, .., N-1, and$$
$$\frac{\partial}{\partial y} J_\mu^N(y_N, v, \Lambda) = \delta_{T_N} \alpha(y_N(T_N) - y^T)$$

For $i = 1, ..., N-1$, the weak formulation of the decomposed adjoint equations will look like:

$$\text{Find } p_i \text{ s.t.} \quad (p_i, (\frac{\partial}{\partial t} - a + \delta_{T_{i-1}})\psi)_i = (\mu \delta_{T_i}(y_i(T_i) - \lambda_i), \psi)_i \quad \forall \psi \in C^\infty((T_{i-1}, T_i)).$$

For $i = N$ the adjoint equation is almost identical to the above expression, with exception of the $(\mu \delta_{T_i}(y_i(T_i) - \lambda_i), \psi)_i$ term, which instead is replaced by $(\delta_{T_N} \alpha(y_N(T_N) - y^T), \psi)_i$. Using partial integration we can move the differentiation from ψ to p_i. The adjoint equation on for $i = 1, ..., N-1$, is then: Find p_i such that

$$\int_{T_{i-1}}^{T_i} (-p_i'(t) - a p_i(t))\psi(t) dt + p_i(T_i)\psi(T_i) = \mu(y_i(T_i) - \lambda_i)\psi(T_i) \quad \forall \psi \in C^\infty.$$

Since we can vary ψ arbitrarily, we recover the strong formulation stated in (5.17). □

With the adjoint equations we can derive the gradient.

Proposition 5.3. *The gradient of (5.15), $\hat{J}_\mu'$, with respect to the control (v, Λ) is:*

$$\hat{J}_\mu'(v, \Lambda) = (v + p, p_2(T_1) - p_1(T_1), ..., p_N(T_{N-1}) - p_N(T_{N-1})) \tag{5.18}$$

Proof. Proposition 5.1 states the gradient of the penalized objective function for a general decomposed problem in (5.11). To derive an expression for the gradient

of our example problem, we need to differntiate the decomposed state equations and the penalized objective function with respect to the real and virtual control. We will again use the weak formulation of the state equation given in the proof of proposition 5.2 to find the different terms. The weak formulation of the i-th state equation is based on the bilinear form $\mathcal{E}^i[v,\phi] = (y_i, (-\frac{\partial}{\partial t} - a + \delta_{T_i})\phi)_i - (v + \delta_{T_{i-1}}\lambda_{i-1}, \phi)_i$. Differentiating $\mathcal{E}^i$ with respect to the real and virtual control yields:

$$\begin{aligned}\mathcal{E}^i_v[\cdot,\phi] &= -(\cdot,\phi)_i, \quad i = 1,...,N,\\ \mathcal{E}^i_{\lambda_{i-1}}[\cdot,\phi] &= -(\cdot,\delta_{T_{i-1}}\phi)_i, \quad i = 2,...,N.\end{aligned}$$

Notice that both of these forms are symmetric, and we therefore do not need to do more work to find their adjoints. The strong interpretation of $\mathcal{E}^i_v$ and $\mathcal{E}^i_{\lambda_{i-1}}$, is that $\mathcal{E}^i_v$ is multiplication by minus one, while $\mathcal{E}^i_{\lambda_{i-1}}$ is multiplication by minus one and evaluation at $t = T_{i-1}$. Next we want to differentiate the decomposed objective functions J^i_μ also defined in the proof of proposition 5.2.

$$\begin{aligned}\frac{\partial}{\partial v} J^i_\mu(y,v,\Lambda) &= v, \quad i = 1,...,N,\\ \frac{\partial}{\partial \lambda_i} J^i_\mu(y,v,\Lambda) &= -\mu(y_i(T_i) - \lambda_i), \quad i = 1,...,N-1.\end{aligned}$$

The last step of the proof is to insert the above derived expressions into formula (5.11). We separate the gradient into two parts, where the first part is the gradient with respect to the real control, while the second part are the components that depends on the virtual control. We start by stating $\frac{\partial}{\partial v}\hat{J}_\mu$:

$$\begin{aligned}\frac{\partial}{\partial v}\hat{J}_\mu(v,\Lambda) &= -E^*_v p + \sum_{i=1}^{N} \frac{\partial}{\partial v} J^i_\mu(y_i,v,\Lambda)\\ &= p + v\end{aligned}$$

We then find the component of the gradient related to λ_i. Only the $i+1$-th state equation and the i-th decomposed objective function depends on λ_i. This yields:

$$\begin{aligned}\frac{\partial}{\partial \lambda_i}\hat{J}_\mu(v,\Lambda) &= -E^{i+1}_{\lambda_i}(y_{i+1},v,\Lambda)^* p_{i+1} + \frac{\partial}{\partial \lambda_i} J^i_\mu(y_i,v,\Lambda)\\ &= p_{i+1}(T_i) - \mu(y_i(T_i) - \lambda_i)\\ &= p_{i+1}(T_i) - p_i(T_i)\end{aligned}$$

Here we made use of $E^{i+1}_{\lambda_i}(y_{i+1},v,\Lambda)^* = -1$ and $p_i(T_i) = \mu(y_i(T_i) - \lambda_i)$ from (5.17). Combining $\frac{\partial}{\partial v}\hat{J}_\mu$ and $\frac{\partial}{\partial \lambda_i}$ for $i = 1,..,N-1$ gives us the gradient (5.18). □

5.3 Parareal Preconditioner

Parallelizing the solution process of optimal control problems with time-dependent differential equation constraints comes down to minimizing a series of penalized objective functions. Since we have derived the gradient of these penalized objective functions for a specific example, we can now solve the control problem numerically using an optimization algorithm. We can for example use the steepest descent method (3.41), which would create the following iteration for each penalized control problem:

$$(v^{k+1}, \Lambda^{k+1}) = (v^k, \Lambda^k) - \rho_k \nabla \hat{J}_\mu(v^k, \Lambda^k) \tag{5.19}$$

Alternatively we could use a BFGS iteration (3.48), which would result in the following update:

$$(v^{k+1}, \Lambda^{k+1}) = (v^k, \Lambda^k) - \rho_k H^k \nabla \hat{J}_\mu(v^k, \Lambda^k) \tag{5.20}$$

Where H^k is the inverse Hessian approximation defined in (3.43). To improve convergence of the unconstrained optimization solvers, we include the Parareal-based preconditioner, proposed in [11], in our optimization algorithms. Assuming that $v \in \mathbb{R}^{n_v}$, the preconditioner Q will be on the form:

$$Q = \begin{bmatrix} \mathbb{1} & 0 \\ 0 & Q_\Lambda \end{bmatrix} \in \mathbb{R}^{n_v+N-1 \times n_v+N-1}, \quad Q_\Lambda \in \mathbb{R}^{N-1 \times N-1} \tag{5.21}$$

We see that Q only affects the $N-1$ last components of the gradient, which is the part connected with the virtual control Λ. The real control v is therefore not directly affected by Q. For steepest descent, we apply Q, by modifying (5.19) in the following way:

$$(v^{k+1}, \Lambda^{k+1}) = (v^k, \Lambda^k) - \rho_k Q \nabla \hat{J}_\mu(v^k, \Lambda^k) \tag{5.22}$$

For us to expect any improvement in convergence for the preconditioned steepest descent, Q would have to resemble the Hessian of $\hat{J}_\mu$, at least for the virtual part of the control. We also need Q to be cheaply computable. Applying Q to the BFGS iteration, is done by setting the initial Hessian approximation $H^0 = Q$. To be able to do this, we need Q to be symmetric positive definite, since that is a requirement on H^0.

We derive Q by looking at a constructed optimal control problem that we call the virtual problem. The virtual problem is a control problem decomposed as detailed in section 5.1, but its objective function $\mathbf{J}$ is set to be the penalty terms, which only depends on the virtual control Λ. We already stated this problem in section 4.3, and by utilizing the algebraic Parareal formulation, we will try to represent the equation $\hat{\mathbf{J}}'(\Lambda) = 0$ with a system $\mathcal{A}\Lambda = R$, and then base Q_Λ on an approximation of $\mathcal{A}^{-1}$.

5.3.1 Virtual Problem

The Parareal-based preconditioner only affects the part of the gradient connected to the virtual control Λ. To motivate and derive Q, we therefore consider an optimal control problem where the real control v is removed, and the objective function only depends on Λ. We have already presented this problem in section 4.3, but we restate it here for future reference. However, before we do this let us first properly define the fine and coarse propagators.

Definition 5.5 (Fine and coarse propagator)**.** *Let $f(y(t), t) = 0$ be a time-dependent differential equation. Given $\Delta T = \frac{T}{N}$ and an initial condition ω, let y_f and y_c be a fine and a coarse numerical solution of the initial value problem:*

$$\begin{cases} f(y(t), t) = 0 & \text{for } t \in (0, \Delta T) \\ y(0) = \omega \end{cases} \tag{5.23}$$

We then define the fine propagator as $\mathbf{F}_{\Delta T}(\omega) = y_f(\Delta T)$ and the coarse propagator as $\mathbf{G}_{\Delta T}(\omega) = y_c(\Delta T)$. We also define the lower triangular matrices $M, \bar{M} \in \mathbb{R}^{N-1 \times N-1}$ as:

$$M = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{F}_{\Delta T} & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{F}_{\Delta T} & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{F}_{\Delta T} & \mathbb{1} \end{bmatrix}, \bar{M} = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{G}_{\Delta T} & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{G}_{\Delta T} & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{G}_{\Delta T} & \mathbb{1} \end{bmatrix}.$$

We then use the fine propagator $\mathbf{F}_{\Delta T}(\omega)$ to define the virtual problem.

Definition 5.6 (Virtual problem)**.** *Given a fine propagator $\mathbf{F}_{\Delta T}$, that solves a time-dependent differential equation $f(y(t), t) = 0$, an initial condition $\lambda_0 = y_0$ and the control variable $\Lambda = (\lambda_1, ..., \lambda_{N-1})$, the virtual control problem is:*

$$\min_{\Lambda} \mathbf{J}(\Lambda, y) = \frac{1}{2} \sum_{i=1}^{N-1} (y_{i-1}(T_i) - \lambda_i)^2, \tag{5.24}$$

$$\text{subject to} \quad y_{i-1}(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1}) \quad \text{for } i = 1, ..., N-1 \tag{5.25}$$

We also recognize function (5.24) as a least squares function, which we can express more compactly as:

$$\mathbf{J}(y, \Lambda) = \frac{1}{2} x(\Lambda)^T x(\Lambda), \tag{5.26}$$

where the vector function $x : \mathbb{R}^{N-1} \to \mathbb{R}^{N-1}$ *is:*

$$x(\Lambda) = \begin{pmatrix} \lambda_1 - \mathbf{F}_{\Delta T}(\lambda_0) \\ \lambda_2 - \mathbf{F}_{\Delta T}(\lambda_1) \\ \cdots \\ \lambda_{N-1} - \mathbf{F}_{\Delta T}(\lambda_{N-1}) \end{pmatrix}. \tag{5.27}$$

In chapter 4 we explained how the virtual problem could be solved by setting $\lambda_i = \mathbf{F}_{\Delta T}(\lambda_{i-1})$, which is the same as solving $\mathbf{J}(\Lambda, y) = 0$. This equation can be written up on matrix form as:

$$M\ \Lambda = H. \tag{5.28}$$

The H on the right hand side of the above equation is the propagator applied to the initial condition:

$$H = \begin{bmatrix} \mathbf{F}_{\Delta T}(y_0) \\ 0 \\ \cdots \\ 0 \end{bmatrix}.$$

In section 4.3 we mentioned that the Parareal algorithm could be reformulated as a preconditioned fix point iteration solving equation (5.28). This can be expressed in matrix form as follows:

$$\Lambda^{k+1} = \Lambda^k + \bar{M}^{-1}(H - M\Lambda^k), \tag{5.29}$$

where $\bar{M}$ is the coarse version of the matrix M stated in definition 5.5. When we are solving the original optimal control problem we do not try to find a triple (v, Λ, y) that solves $J_\mu(v, \Lambda, y) = 0$. Instead we try to solve $\hat{J}'_\mu(v, \Lambda) = 0$. To find the Parareal-based preconditioner, we therefore try to find a similar expression to (5.28) for $\hat{\mathbf{J}}'(\Lambda) = 0$. To be able to find this expression, we first need to define the coarse and fine adjoint propagators.

Definition 5.7 (Fine and coarse adjoint propagator)**.** *Let* $f(y(t), t) = 0$ *be a time-dependent differential equation. Given* ΔT*, a state* $y(t)$ *and an initial condition* ω*, let* p_f *and* p_c *be a fine and a coarse numerical solution of the initial value problem:*

$$\begin{cases} f'(y(t), t)^* p(t) = 0 \quad \text{for } t \in (0, \Delta T) \\ p(\Delta T) = \omega \end{cases} \tag{5.30}$$

We then define the fine adjoint propagator as $\mathbf{F}^*_{\Delta T}(\omega) = p_f(0)$ *and the coarse adjoint propagator as* $\mathbf{G}^*_{\Delta T}(\omega) = p_c(0)$. *We also define adjoint versions of the matrices* M *and* $\bar{M}$ *as:*

$$M^* = \begin{bmatrix} \mathbb{1} & -\mathbf{F}^*_{\Delta T} & 0 & 0 \\ 0 & \mathbb{1} & -\mathbf{F}^*_{\Delta T} & \cdots \\ \cdots & 0 & \mathbb{1} & -\mathbf{F}^*_{\Delta T} \\ 0 & \cdots & \cdots & \mathbb{1} \end{bmatrix}, \bar{M}^* = \begin{bmatrix} \mathbb{1} & -\mathbf{G}^*_{\Delta T} & 0 & 0 \\ 0 & \mathbb{1} & -\mathbf{G}^*_{\Delta T} & \cdots \\ \cdots & 0 & \mathbb{1} & -\mathbf{G}^*_{\Delta T} \\ 0 & \cdots & \cdots & \mathbb{1} \end{bmatrix}.$$

Using the matrices from definition 5.7 we can write up the following proposition concerning the gradient of the reduced objective function of the virtual problem.

Proposition 5.4. *The reduced objective function of the virtual problem (5.24-5.25) is:*

$$\hat{\mathbf{J}}(\Lambda) = \frac{1}{2} \sum_{i=1}^{N-1} (\mathbf{F}_{\Delta T}(\lambda_{i-1}) - \lambda_i)^2. \tag{5.31}$$

Solving $\hat{\mathbf{J}}'(\Lambda) = 0$ *is equivalent to resolving the system:*

$$M^* \; M \; \Lambda \; = \; M^* \; H. \tag{5.32}$$

A preconditioned fix point iteration for equation (5.32) inspired by the Parareal formulation (5.29) is therefore:

$$\Lambda^{k+1} = \Lambda^k + \bar{M}^{-1}\bar{M}^{-*}(M^*H - M^*M\Lambda^k). \tag{5.33}$$

Proof. We have already derived the gradient of $\hat{\mathbf{J}}$ in (5.18). There we stated the gradient for the penalized version of the example problem (3.8-3.9). If we ignore the part of this gradient related to the real control v, we get the following expression for $\hat{\mathbf{J}}'$:

$$\hat{\mathbf{J}}'(\Lambda) = \{p_{i+1}(T_i) - p_i(T_i)\}_{i=1}^{N-1}.$$

Here p_i refers to the decomposed adjoint equation on interval $[T_{i-1}, T_i]$. We now want to show that setting $p_{i+1}(T_i) - p_i(T_i) = 0$ for $i = 1, ..., N-1$ is equivalent to equation 5.32. To do this we will simply write out the expression $M^*(M\Lambda - H)$ and show that it equals $\hat{\mathbf{J}}'(\Lambda)$. We start with $M\Lambda - H$.

$$M \; \Lambda - H = \begin{pmatrix} \lambda_1 - \mathbf{F}_{\Delta T}(\lambda_0) \\ \lambda_2 - \mathbf{F}_{\Delta T}(\lambda_1) \\ \cdots \\ \lambda_{N-1} - \mathbf{F}_{\Delta T}(\lambda_{N-1}) \end{pmatrix}.$$

Notice that $\mathbf{F}_{\Delta T}(\lambda_{i-1}) - \lambda_i$ is the initial condition of i-th adjoint equation, i.e. $p_i(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1}) - \lambda_i$. By exploiting this, and multiplying $M\Lambda - H$ with M^* we get:

$$M^*(M\ \Lambda - H) = \begin{pmatrix} \mathbf{F}^*_{\Delta T}(p_2(T_2)) - p_1(T_1) \\ \mathbf{F}^*_{\Delta T}(p_3(T_3)) - p_2(T_2) \\ \cdots \\ -p_{N-1}(T_{N-1}) \end{pmatrix} \tag{5.34}$$

$$= \begin{pmatrix} p_2(T_1) - p_1(T_1) \\ p_3(T_2) - p_2(T_2) \\ \cdots \\ p_{N-1}(T_{N-2}) - p_{N-2}(T_{N-2}) \\ -p_{N-1}(T_{N-1}) \end{pmatrix}. \tag{5.35}$$

The last step is done by using $p_i(T_{i-1}) = -F^*_{\Delta T}(-p_i(T_i))$, and this is possible since the adjoint equation is linear. We see that the i-th component of $M^*(M\Lambda - H)$ is equal to $p_{i+1}(T_i) - p_i(T_i)$ for $i \neq N-1$. The last component of $M^*(M\Lambda - H)$ is $-p_{N-1}(T_{N-1})$, and we are therefore missing $p_N(T_{N-1})$. This is however unproblematic since in context of the the virtual problem $p_N(T_{N-1}) = 0$. This shows us that $\hat{\mathbf{J}}'(\Lambda) = M^*(M\Lambda - H)$, which means that $\hat{\mathbf{J}}'(\Lambda) = 0 \iff M^*M\Lambda = M^*H$. Since $\bar{M}$ and $\bar{M}^*$ approximates M and M^*, $\bar{M}^{-1}\bar{M}^{-*}$ would be a natural preconditioner for a fix point iteration solving $M^*M\Lambda = M^*H$. □

Proposition 5.4 motivates $Q_\Lambda = \bar{M}^{-1}\bar{M}^{-*}$ as a preconditioner for solvers of decomposed and penalized optimal control problems, and this is actually the Parareal-based preconditioner proposed in [11]. Inserting Q_Λ into Q yields the following:

$$Q = \begin{bmatrix} \mathbb{1} & 0 \\ 0 & \bar{M}^{-1}\bar{M}^{-*} \end{bmatrix}. \tag{5.36}$$

In [11] Q is proposed as a preconditioner for a steepest descent method. Other than to motivate Q the authors of [11] do not explore or derive any properties of the Parareal-based preconditioner. We are however interested in using Q in combination with the BFGS algorithm, and to be able to do this we need to know that Q is positive definite and that it is related to the Hessian of the objective function. We are also interested in the computational cost of Q. We will investigate the properties of the preconditioner by looking at the least squares formulation (5.26) of problem (5.31).

5.3.2 Properties of the Parareal-Based Preconditioner

We want to investigate the properties of $Q_\Lambda = \bar{M}^{-1}\bar{M}^{-*}$, and to do this we will show that $\bar{M}^1\bar{M}^*$ is an approximation to the Hessian of $\hat{\mathbf{J}}(\Lambda)$. To calculate the Hessian we use the least squares formulation (5.26) of the virtual objective function.

Proposition 5.5 (Virtual Hessian)**.** *The Hessian of function (5.26) is*

$$\begin{aligned}\nabla^2\hat{\mathbf{J}}(\Lambda) &= \nabla x^T\nabla x + \sum_{i=1}^{N-1}\nabla^2 x_i(\Lambda)x_i(\Lambda)\\ &= M(\Lambda)^T M(\Lambda) + D(\Lambda)\end{aligned}$$

Here $D(\Lambda)$ is a diagonal matrix with diagonal entries

$$D_i = -\mathbf{F}''_{\Delta T}(\lambda_i)(\lambda_{i+1} - \mathbf{F}_{\Delta T}(\lambda_i)) \quad i = 1, ..., N-1,$$

while $M(\Lambda)$ is the linearised forward model:

$$M(\Lambda) = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{F}'_{\Delta T}(\lambda_1) & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{F}'_{\Delta T}(\lambda_2) & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{F}'_{\Delta T}(\lambda_{N-1}) & \mathbb{1} \end{bmatrix}$$

Proof. We start by differentiating $\hat{\mathbf{J}}$:

$$\begin{aligned}\nabla\hat{\mathbf{J}}(\Lambda) &= \nabla x(\Lambda)^T x(\Lambda)\\ &= \sum_{i=1}^{N-1}\nabla x_i(\Lambda)x_i(\Lambda)\end{aligned}$$

If we now differentiate $\nabla\hat{\mathbf{J}}$, we get:

$$\nabla^2\hat{\mathbf{J}}(\Lambda) = \nabla x^T\nabla x + \sum_{i=1}^{N-1}\nabla^2 x_i(\Lambda)x_i(\Lambda)$$

We see that $\nabla x(\Lambda) = M(\Lambda)$, by looking at $\frac{\partial x_i}{\partial\lambda_j}$

$$\frac{\partial x_i}{\partial\lambda_j} = \begin{cases} 1 & i = j \\ -\mathbf{F}'_{\Delta T}(\lambda_j) & i > 1 \wedge j = i-1 \\ 0 & i \neq j \vee j \neq i-1 \end{cases}$$

We can similarly find $\nabla^2 x_i$ by differentiating x twice:

$$\frac{\partial^2 x_i}{\partial \lambda_j \partial \lambda_k} = \begin{cases} -\mathbf{F}''_{\Delta T}(\lambda_j) & i > 1 \wedge j = k = i-1 \\ 0 & \text{in all other cases} \end{cases}$$

Summing up the terms $\nabla^2 x_i(\Lambda) x_i(\Lambda)$ yields the diagonal matrix $D(\Lambda)$. □

The first term of $\nabla^2 \hat{\mathbf{J}}(\Lambda) = M(\Lambda)^T M(\Lambda) + D(\Lambda)$ resembles $M^* M$ from the previous section, while the second term $D(\Lambda)$ is new. $D(\Lambda)$ is a diagonal matrix where the diagonal entries consists of products between the second derivative of $\mathbf{F}_{\Delta T}$ and the residuals $\lambda_{i+1} - \mathbf{F}_{\Delta T}(\lambda_i)$. If the governing equation of the propagator $\mathbf{F}_{\Delta T}$ is linear, $\mathbf{F}''_{\Delta T}(\lambda_i) = 0$. This would again mean that $D(\Lambda) = 0$ and that $\nabla^2 \hat{\mathbf{J}}(\Lambda) = M(\Lambda)^T M(\Lambda)$. We will therefore split our discussion of the Hessian of $\hat{\mathbf{J}}$ into two cases. In the first we assume the state equation is linear, while in the second case we discuss problems with non-linear state equations.

Linear State Equations

Assuming that the state equation is linear means that $\nabla^2 \hat{\mathbf{J}}(\Lambda) = M(\Lambda)^T M(\Lambda)$. Differentiating the propagator $\mathbf{F}_{\Delta T}$ is the same as linearising its governing equation. When the governing equation is itself linear and homogeneous, linearising it does not change the equation, and $\mathbf{F}'_{\Delta T}(\lambda_i)\lambda_i = \mathbf{F}_{\Delta T}(\lambda_i)$. This means that the M matrix from section 5.3.1 is equal to $M(\Lambda)$. The same is true for M^* and $M(\Lambda)^T$. Since $\nabla^2 \hat{\mathbf{J}}(\Lambda) = M^* M$ we see that the Parareal-based preconditioner proposed in [11] is in fact related to the inverse Hessian of the reduced penalized objective function. Furthermore if we can show that $\bar{M}^* \bar{M}$ is a positive definite matrix, we can use Q as an initial approximation of the inverse Hessian in the BFGS optimization algorithm. This is as we will see in the following proposition indeed the case.

Proposition 5.6. *If $\mathbf{G}_{\Delta T}$ and $\mathbf{G}^*_{\Delta T}$ are based on consistent numerical methods, that is $\bar{M}^* = \bar{M}^T$, then the matrix $\bar{M}^* \bar{M}$ is positive definite.*

Proof. If $\mathbf{G}_{\Delta T}$ and $\mathbf{G}^*_{\Delta T}$ are based on consistent numerical methods equal, meaning that $\mathbf{G}_{\Delta T}(\omega) = \mathbf{G}^*_{\Delta T}(\omega)$. When inserting this into the matrices $\bar{M}$ and $\bar{M}^*$ from definition 5.5 and 5.7, we clearly see that $\bar{M}^* = \bar{M}^T$. For $M^* M$ to be positive definite, the following two conditions must hold:

1. $x^T \bar{M}^* \bar{M} x \geq 0 \quad \forall x \in \mathbb{R}^{N-1}$
2. $x^T \bar{M}^* \bar{M} x = 0 \iff x = 0$

The first condition hold due to $\bar{M}^* = \bar{M}^T$:

$$x^T \bar{M}^* \bar{M} x = (\bar{M} x)^T \bar{M} x = ||Mx||^2 \geq 0.$$

The second condition hold if $\bar{M}$ is invertible. This is true because $\bar{M}$ is a triangular matrix, with identity on its diagonal, and therefore has a determinant equal to 1. The determinant of a matrix being unequal to zero is equivalent with it being invertible, which means that $\bar{M}$ is invertible. This also means that M^*M is positive definite, since both requirements for positive definiteness are satisfied. □

Proposition 5.6 shows that the $\bar{M}^*\bar{M}$ matrix stemming from the virtual problem is positive definite. We can therefore use it as an initial Hessian approximation in the BFGS algorithm, at least as long as $\mathbf{G}_{\Delta T}$ and $\mathbf{G}^*_{\Delta T}$ are consistent. Now let us take a look at the case where the governing equation of $\mathbf{F}_{\Delta T}$ is non-linear.

Non-Linear State Equations

Unlike the Hessian of the linear problem the Hessian of the non-linear problem consists of two parts. One is the linearised forward model multiplied with its adjoint, while the second part is a diagonal matrix related to the second derivative of the propagator $\mathbf{F}_{\Delta T}$, and the residuals $\lambda_i - \mathbf{F}_{\Delta T}$. The first part of $\nabla^2\hat{\mathbf{J}}$ is analogue to the Hessian of the linear problem. It is symmetric positive definite, and taking its inverse corresponds to first applying the backwards model, and then the forward model. What makes the Hessian of the non-linear problematic is therefore its second term. The first issue with the diagonal matrix $D(\Lambda)$, is how to calculate $\mathbf{F}''_{\Delta T}$. Another issue is that we can not guarantee that the sum of $M(\Lambda)^T M(\Lambda)$ and $D(\Lambda)$ is a positive matrix, and the same problem would arise in a coarse approximation of $\nabla^2\hat{\mathbf{J}}$. The lack of positivity is a problem since we want to use the coarse approximation as an initial inverted Hessian approximation in the BFGS-algorithm.

One way to get around the $D(\Lambda)$ term in the Hessian for a non-linearly constrained problem, is simply to ignore it. This leaves us with the $M(\Lambda)^T M(\Lambda)$ term, which we know how to deal with. Ignoring the term depending on the second derivative and the residual is actually a known strategy, called the Gauss-Newton method, for solving non-linear least square problems. Details on this method can be found in [31]. A justification for this approach, is that at least in instances where we are close to a solution, the $\lambda_i - \mathbf{F}_{\Delta T}$ terms will be close to zero, and the $M(\Lambda)^T M(\Lambda)$ term will therefore dominate the Hessian. Ignoring the $D(\Lambda)$ term means that we can define an inverse Hessian approximation based on a coarse propagator $\mathbf{G}_{\Delta T}$ in the same way as we did for the problem with linear state equation constraints.

This means that we define a matrix $\bar{M}(\Lambda)$:

$$\bar{M}(\Lambda) = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{G}'_{\Delta T}(\lambda_1) & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{G}'_{\Delta T}(\lambda_2) & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{G}'_{\Delta T}(\lambda_{N-1}) & \mathbb{1} \end{bmatrix} \tag{5.37}$$

The term $\bar{M}(\Lambda)^{-1}\bar{M}(\Lambda)^{-*}$ can then be used in an approximation of the inverse Hessian, as detailed in section 5.3.1.

5.3.3 Parareal-Based Precoditioner for the Example Problem

To illustrate what Q actually will look like we write up $\bar{M}^*\bar{M}$ for our example problem (3.8-3.9). The state equation of this problem is:

$$f(y(t), t) = y'(t) - ay(t) - v(t) = 0, \tag{5.38}$$

If we linearise f, the source term v disappears, and the state equation becomes homogeneous. We then base the coarse propagators $\mathbf{G}_{\Delta T}$ and $\mathbf{G}^*_{\Delta T}$ on the linearised state equation and its adjoint:

$$y'(t) = ay(t), \tag{5.39}$$
$$p'(t) = -ap(t). \tag{5.40}$$

Let us now try to write out $\bar{M}^*\bar{M}$ for our example problem, when we have decomposed the time interval into N subintervals. We first need to choose a numerical method to discretize the linearised state equation (5.39) and the adjoint equation (5.40). In this example we will use the implicit Euler scheme from section 3.3.1, with $\Delta T = \frac{T}{N}$. We can then write up $\mathbf{G}_{\Delta T}(\omega)$ and $\mathbf{G}^*_{\Delta T}(\omega)$:

$$\frac{\mathbf{G}_{\Delta T}(\omega) - \omega}{\Delta T} = a\mathbf{G}_{\Delta T}(\omega) \tag{5.41}$$
$$\Rightarrow \mathbf{G}_{\Delta T}(\omega) = \frac{\omega}{1 - a\Delta T} \tag{5.42}$$
$$\frac{\omega - \mathbf{G}^*_{\Delta T}(\omega)}{\Delta T} = -a\Delta T\mathbf{G}^*_{\Delta T}(\omega) \tag{5.43}$$
$$\Rightarrow \mathbf{G}^*_{\Delta T}(\omega) = \frac{\omega}{1 - a\Delta T} \tag{5.44}$$

Since $\mathbf{G}_{\Delta T}(\omega) = \mathbf{G}^*_{\Delta T}(\omega)$, using implicit Euler both forwards and backwards produce consistent coarse propagators. We can now write up an exact expression for

$\bar{M} \in \mathbb{R}^{N-1 \times N-1}$.

$$\bar{M} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -\frac{1}{1-a\Delta T} & 1 & 0 & \cdots \\ 0 & -\frac{1}{1-a\Delta T} & 1 & \cdots \\ 0 & \cdots & -\frac{1}{1-a\Delta T} & 1 \end{bmatrix}.$$

By traversing $\bar{M}$ we get $\bar{M}^*$. When we apply Q, we are not using $\bar{M}^*\bar{M}$, but instead its inverse. Let us illustrate how to apply the inverse of $\bar{M}^*\bar{M}$ to the virtual gradient through an example, wehere we set $N = 4$. We first decompose $I = [0, T]$ into four sub-intervals $[T_0, T_1], [T_1, T_2], [T_2, T_3]$ and $[T_3, T_4]$. If we then evaluate the discrete gradient for a real control variable $v \in \mathbb{R}^{n+1}$ and a virtual control $\Lambda = (\lambda_1, \lambda_2, \lambda_3)$, the result is $\hat{J}_\mu(v, \Lambda) \in \mathbb{R}^{N+n}$. Multiplying Q with $\hat{J}_\mu(v, \Lambda)$ will only affect its three last components, which we name $J_{\lambda_1}, J_{\lambda_2}$ and J_{λ_3}. Applying Q to $\hat{J}_\mu$ is done in two steps. We first multiply with $\bar{M}^{-*}$ based on on the propagator $\mathbf{G}^*_{\Delta T} = -\frac{1}{1-a\Delta T}$

$$\begin{aligned} \bar{J_{\lambda_1}} &= J_{\lambda_1} - \frac{1}{1-a\Delta T}(J_{\lambda_2} - \frac{1}{1-a\Delta T} J_{\lambda_3}) \\ \bar{J_{\lambda_2}} &= J_{\lambda_2} - \frac{1}{1-a\Delta T} J_{\lambda_3} \\ \bar{J_{\lambda_3}} &= J_{\lambda_3} \end{aligned}$$

The second step is then to apply the forward system based on the coarse propagator $\mathbf{G}_{\Delta T} = -\frac{1}{1-a\Delta T}$:

$$\begin{aligned} \bar{\bar{J}}_{\lambda_1} &= \bar{J_{\lambda_1}} \\ \bar{\bar{J}}_{\lambda_2} &= \bar{J_{\lambda_2}} - \frac{1}{1-a\Delta T} \bar{J_{\lambda_1}} \\ \bar{\bar{J}}_{\lambda_3} &= \bar{J_{\lambda_3}} - \frac{1}{1-a\Delta T}(\bar{J_{\lambda_2}} - \frac{1}{1-a\Delta T} \bar{J_{\lambda_1}}) \end{aligned}$$

The result of multiplying Q with the discrete penalized gradient is that the three last components of $\hat{J}_\mu(v, \Lambda)$ is changed to $\bar{\bar{J}}_{\lambda_1}, \bar{\bar{J}}_{\lambda_2}$ and $\bar{\bar{J}}_{\lambda_3}$.

an important special case of the Parareal-based preconditioner is the case when $N = 2$. If we decompose the time domain into $N = 2$ subdomains, both $\bar{M}$ and $\bar{M}^*$ becomes the identity matrix. This means that for $N = 2$, $Q = \mathbb{1}$, and therefore Q has no effect. Since the preconditioner has no effect for $N = 2$, we might also expect that for "small" N the impact of applying Q to the penalized gradient is only modest, and that the usefulness of Q only materializes for higher values of decomposed subintervals N.

Computational Cost of Parareal-Based Preconditioner

The last aspect of the preconditioner Q (5.36), that we have yet not discussed is its computational cost. For Q to be an effective preconditioner it needs to be cheap to compute. As the above example for $N = 4$ shows, applying Q_Λ to the virtual part of the gradient $\hat{J}'_\mu(v, \Lambda)$, comes down to first solving the linearised backward model, and then the linearised forward model on mesh of size N. This will translate to a computational cost of $\mathcal{O}(N)$. Here we have of course assumed that the state and adjoint equations are ODEs, and that the coarse propagator $\mathbf{G}_{\Delta T}$ is based on a finite difference scheme as in (5.44). If we we instead were solving a PDE, the cost of applying Q would also include computations done in spatial direction. If the spatial discretization has size $\mathcal{M}$, the cost of Q would instead be $\mathcal{O}(\mathcal{M}N)$, but this could again be made cheaper by using a coarse resolution in space for $\mathbf{G}_{\Delta T}$.

For $\mathcal{O}(N)$ to be considered a cheap operation, we require $N << n$, where n is the number of fine time steps. Since N is the number of decomposed subintervals, N does also equal the maximal number of processes that can be used to parallelize in time. If we want to increase the number of processes, we also need to increase N. This creates an upper limit for the scalability of our algorithm, when we use the Parareal-based preconditioner. For a fixed problem size n, the absolute upper limit of processes that can be used is $N = n$, but since we need a cheap to compute Q, this limit is in practice lower.

5.4 Summary and Presentation of Algorithm

In the previous section we presented and derived properties of the Parareal-based preconditioner Q introduced in [11]. We showed that Q is symmetric positive definite, and that it approximates the inverse Hessian of $\hat{J}_\mu$, at least for the part connected to the virtual control. We can therefore use Q as an initial inverted Hessian approximation in the BFGS or L-BFGS optimization algorithms for minimization of the penalized objective function $\hat{J}_\mu(v, \Lambda)$ (5.6). Combining the preconditioned BFGS solver with the quadratic penalty method of algorithm 5.1 makes us able to propose algorithm 5.2 as a parallel in time method for solving optimization problems with time-dependent DE constraints.

In algorithm 5.2 we use the Parareal-based preconditioner Q, but an unpreconditioned version of algorithm 5.2 would also be able to solve problem (5.6). When we investigate the performance of our Parareal-preconditioned method we will compare it with the unpreconditioned algorithm.

Algorithm 5.2: Quadratic penalty method with preconditioned BFGS optimization

Data: Choose $\mu_0, \tau_0 > 0$, and some initial control (v^0, Λ^0)
for $k = 1, 2, ...$ **do**
  $(v_0^k, \Lambda_0^k) \leftarrow (v^{k-1}, \Lambda^{k-1})$;
  $H^0 \leftarrow Q(\Lambda_0^k)$;
  while $||\hat{J}'_{\mu_{k-1}}(v_j^k, \Lambda_j^k)|| \geq \tau_{k-1}$ **do**
    $(v_{j+1}^k, \Lambda_{j+1}^k) \leftarrow (v_j^k, \Lambda_j^k) - \rho^j H^j \hat{J}'_{\mu_{k-1}}(v_j^k, \Lambda_j^k)$// `In parallel`;
    Update H^{j+1};
    $H^0 \leftarrow Q(\Lambda_{j+1}^k)$;
  end
  $(v^k, \Lambda^k) \leftarrow (v_j^k, \Lambda_j^k)$;
  if *STOP CRITERION on* (v^k, Λ^k) *satisfied* **then**
    Stop algorithm;
  else
    Choose new $\tau_k \in (0, \tau_{k-1})$ and $\mu_k \in (\mu_{k-1}, \infty)$;
  end
end

What separates algorithm 5.2 from the general quadratic penalty method in algorithm 5.1 is that the optimization step is done using BFGS (or L-BFGS) with $H^0 = Q$. The most computationally costly part of algorithm 5.2, is the optimization step of the BFGS algorithm:

$$(v_{j+1}, \Lambda_{j+1}) = (v_j, \Lambda_j) - \rho^j H^j \hat{J}'_\mu(v_j, \Lambda_j) \tag{5.45}$$

In section 3.4 we explained how general line search methods are applied, and also how one updates the inverse Hessian approximation in the BFGS and L-BFGS algorithms. Let us however briefly discuss how the update (5.45) is done in context of the minimization of our decomposed and penalized objective function (5.9). Executing update (5.45) is done in four steps:

1. *Evaluate* $\hat{J}'_\mu(v_j, \Lambda_j)$.
2. *Apply* H^j *to* $\hat{J}'_\mu(v_j, \Lambda_j)$.
3. *Find step length* ρ^j
4. *Set* $(v_{j+1}, \Lambda_{j+1}) = (v_j, \Lambda_j) - \rho^j H^j \hat{J}'_\mu(v_j, \Lambda_j)$

The first step of the above procedure, is to evaluate the gradient of $\hat{J}_\mu$. We know from proposition 5.1, that this requires us to first solve the decomposed state equations, and then the decomposed adjoint equations. We can solve the

decomposed state equations and then the decomposed adjoint equations in parallel, since they are defined independently of each other on the decomposed subintervals. Applying H^j to $\hat{J}'_\mu(v_j, \Lambda_j)$ is done using the recursive formula defining the H^j update:

$$H^j \hat{J}_\mu = (\mathbb{1} - \rho_{j-1} S_{j-1} \cdot Y_{j-1}) H^{j-1} (\mathbb{1} - \rho_{j-1} Y_{j-1} \cdot S_{j-1}) \hat{J}_\mu + S_{j-1} \cdot S_{j-1} \hat{J}_\mu \quad (5.46)$$

$Y, S \in \mathbb{R}^{n+N}$ are vectors based on previous iterates, that we defined in section 3.4. The point of evaluating $H^j \hat{J}_\mu$ recursively as in (5.46), is that we do not need to build the full matrix H^j. An important thing about formula (5.46), is that it is solely made up of dot products, vector subtraction, vector addition and scalar products, all of which are perfectly parallelizable operations [50]. With the exception of the initial inverted Hessian approximation $H^0 = Q$, which we assume is computationally cheap relative to a state equation solve, applying H^j to the gradient of $\hat{J}_\mu$ can be done completely in parallel.

The third step of the BFGS update is to find a step length ρ^j, that satisfies the Wolfe conditions (3.39-3.40). We will not explain how to find ρ^j here, however, what we can say, is that calculating the step length requires at least one evaluation of $\hat{J}_\mu$ and one of $\hat{J}'_\mu$. This means that finding ρ^j is a computationally costly procedure, but since evaluating $\hat{J}_\mu$ and $\hat{J}'_\mu$ boils down to solving the state and adjoint equations, finding ρ^j is also a perfectly parallelizable process. The fourth and last step of the BFGS update , is to update (v_{j+1}, Λ_{j+1}) using formula (5.45). This is a very simple step involving only scalar multiplication and vector subtraction, and can of course be executed in parallel.

How to update the penalty parameter μ_k and tolerance τ_k, as well as how to choose an adequate stopping criteria, are all aspects of algorithm 5.2, that require consideration. We will however not look into these questions in this thesis, and when we test out the method in algorithm 5.2 in chapter 8, we will in all experiments use one penalty iteration with a large penalty parameter μ. We found that this strategy worked reasonably well for the example problem, while also being sufficient for demonstrating the method. Strategies for updating the μ and τ variables can be found in [31], but there does not seem to be a general approach that fits every type of problem.

Chapter 6

Discretization and Parallelization of the Penalized Objective Function

In the previous chapters we derived the adjoint equation and the gradient for our example optimal control problem with ODE constraints. We also explained how we can parallelize the solving of the state and adjoint equations using the penalty method, and we introduced a preconditioner for our optimization algorithm based on the Parareal scheme. Before we can start to test our Parallel algorithm, we need to discretize the time domain, the equations, the objective function and its gradient.

We discretize the time interval $I = [0, T]$ by dividing it into n parts of length $\Delta t = \frac{T}{n}$, and set $t_k = k\Delta t$. This gives us a sequence $I_{\Delta t} = \{t_k\}_{k=0}^{n}$ as a discrete representation of the interval I. Using $I_{\Delta t}$ we can start to discretize our example problem.

6.1 Discretizing the Non-Penalized Example Problem

We restate our example state equation (3.9) and objective function (3.8) for future reference.

$$\begin{cases} y'(t) = ay(t) + v(t), \ t \in (0, T) \\ y(0) = y_0 \end{cases} \tag{6.1}$$

$$J(y,v) = \frac{1}{2}\int_0^T v(t)^2 dt + \frac{\alpha}{2}(y(T) - y^T)^2 \tag{6.2}$$

The reduced gradient of (6.2) is:

$$\nabla \hat{J}(v) = v(t) + p(t), \tag{6.3}$$

where p is the solution of the adjoint equation:

$$\begin{cases} -p'(t) = p(t) \\ p(T) = \alpha(y(T) - y^T) \end{cases} \tag{6.4}$$

We now want to discretize (6.1-6.4), so we can solve the problem numerically. What we particularly want, is an expression for the gradient.

6.1.1 Finite Difference Schemes for the State and Adjoint Equations

To evaluate the gradient of our example problem numerically, we need to discretize its state (6.1) and adjoint (6.4) equation. We do this by applying the finite difference schemes introduced in section 3.3.1. We denote the discrete state as $y_{\Delta t} = \{y_k\}_{k=0}^n$ and the discrete adjoint as $p_{\Delta t} = \{p_k\}_{k=0}^n$. With explicit Euler, implicit Euler and Crank-Nicholson we get three different expressions for y_{k+1} and p_{k-1}, and with these expressions we can solve (6.1) and (6.4) numerically. We start with the explicit Euler scheme (3.27):

$$y_{k+1} = (1 + \Delta t a)y_k + \Delta t v_k \tag{6.5}$$

$$p_{k-1} = p_k(1 + \Delta t a) \tag{6.6}$$

Applying the implicit Euler scheme to (6.1) and (6.4) yields:

$$y_{k+1} = \frac{y_k + \Delta t v_{k+1}}{1 - a\Delta t} \tag{6.7}$$

$$p_{k-1} = \frac{p_k}{1 - \Delta t a} \tag{6.8}$$

When we use Crank-Nicolson the expressions for y^{k+1} and p^{k-1} are:

$$y_{k+1} = \frac{(1 + \frac{\Delta t a}{2})y_k + \frac{\Delta t}{2}(v_{k+1} + v_k)}{1 - \frac{\Delta t a}{2}} \tag{6.9}$$

$$p_{k-1} = \frac{1 + \frac{\Delta t a}{2}}{1 - \frac{\Delta t a}{2}} p_k \tag{6.10}$$

The expressions for the state y_{k+1} stems from the forward solving schemes (3.27), (3.28) and (3.32), while p_{k-1} were found using (3.29), (3.30) and (3.33). One issue that becomes apparent when looking at the finite difference scheme formulas above is the question of stability. For all the schemes certain combinations of Δt and a will result in division by zero, or unnatural oscillations. These numerical artefacts can be removed by decreasing Δt. We summarize the different stability requirements of the three schemes in table 6.1, where we for each scheme have written up the stable values of Δt for positive and negative a values. We notice

Table 6.1: Stability domains for finite difference schemes

	$a < 0$	$a > 0$
Explicit Euler	$0 < \Delta t < -\frac{1}{a}$	$\Delta t > 0$
Implicit Euler	$\Delta t > 0$	$0 < \Delta t < \frac{1}{a}$
Crank-Nicolson	$0 < \Delta t < -\frac{2}{a}$	$0 < \Delta t < \frac{2}{a}$

that the implicit Euler scheme is stable for all Δt values when $a < 0$, and that the same holds true for explicit Euler in the case where $a > 0$. This makes these schemes attractive candidates for use in coarse propagators in the context of the Parareal algorithm or preconditioner.

6.1.2 Numerical Gradient

We have discretized both the domain and the equations, but we also need to evaluate the objective function (6.2) numerically. Since integration is involved in (6.2), we have to choose a numerical integration rule. In section 3.3.2 we introduced three different methods for numerical integration, namely the left- and right-hand rectangle rule, as well as the trapezoid rule. Which of the methods we use in our discrete objective function depends on which finite difference scheme we used to discretize the ODEs. For explicit Euler we use the left-hand rule, for implicit Euler we use the right-hand rule, and for Crank-Nicholson we use the trapezoid rule. If we for example used Crank-Nicholson and the trapezoid rule to discretize problem (6.2), the discretized objective function would look like the following:

$$\hat{J}_{\Delta t}(v_{\Delta t}) = \frac{1}{2} trapz(v_{\Delta t}^2) + \frac{\alpha}{2}(y_n - y^T)^2 \tag{6.11}$$

$$= \Delta t \frac{v_0^2 + v_n^2}{4} + \frac{1}{2}\sum_{i=1}^{n-1} \Delta t v_i^2 + \frac{\alpha}{2}(y_n - y^T)^2 \tag{6.12}$$

We now want to find the gradient of the discrete objective function for the different combinations of finite difference schemes and integration rules, so that we can

minimize (6.1-6.2) numerically. The gradients for the different discretizations are stated in terms of the discrete control $v_{\Delta t}$ and discrete adjoint $p_{\Delta t}$ in theorem 6.1 below.

Theorem 6.1. *If the implicit Euler finite difference scheme together with the right-hand rectangle rule is used to evaluate the numerical objective function, the gradient $\nabla \hat{J}_{\Delta t}$ of (6.12) will be given as:*

$$\nabla \hat{J}_{\Delta t}(v_{\Delta t}) = M_0 v_{\Delta t} + B p_{\Delta t} \tag{6.13}$$

where M_θ and B are the matrices:

$$M_\theta = \begin{bmatrix} \theta\Delta t & 0 & \cdots & 0 \\ 0 & \Delta t & 0 & \cdots \\ 0 & 0 & \Delta t & \cdots \\ 0 & \cdots & 0 & (1-\theta)\Delta t \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \Delta t & 0 & 0 & \cdots \\ 0 & \Delta t & 0 & \cdots \\ 0 & \cdots & \Delta t & 0 \end{bmatrix}$$

If one instead uses the explicit Euler finite difference scheme on the differential equations and the left-hand rectangle rule for integration, the gradient will instead be:

$$\nabla \hat{J}_{\Delta t}(v_{\Delta t}) = M_1 v_{\Delta t} + B^T p_{\Delta t}$$

Lastly if the state and adjoint equation of problem (6.1-6.2) is discretized using the Crank-Nicholson scheme, while numerical integration is done using the trapezoid rule, the numerical gradient is:

$$\nabla \hat{J}_{\Delta t}(v_{\Delta t}) = M_{\frac{1}{2}} v_{\Delta t} + \frac{1}{2}\left(\frac{1}{1+\frac{\Delta t a}{2}} B + \frac{1}{1-\frac{\Delta t a}{2}} B^T\right) p_{\Delta t}$$

Proof. Let us start with the $M_\theta v$ terms of the gradients. These terms comes from the integral $\int_0^T v(t)^2 dt$, which we approximate using the numerical integration rules stated in section 3.3.2. It turns out that we can define the three integration rules applied to $v_{\Delta t}^2$ using the matrix M_θ:

$$\int_0^T v(t)^2 dt \approx \Delta t(\theta v_0 + (1-\theta)v_n) + \sum_{i=1}^{n-1} \Delta t v_i^2 = v_{\Delta t}^T M_\theta v_{\Delta t}$$

The function $f(v) = \frac{1}{2} v^T M_\theta v$ obviously has $M_\theta v$ as gradient. The second term of the gradient comes from the second term of the functional, namely $g(v) = \frac{\alpha}{2}(y^n - y^T)^2$. This term needs to be handled separately for each finite difference discretization of the ODEs. We start with case where implicit Euler was used. To

differentiate g with respect to the i-th component of v, we will apply the chain rule multiple times. Let us first demonstrate by calculating $\frac{\partial g}{\partial v_n}$:

$$\begin{aligned}\frac{\partial g(v)}{\partial v_n} &= \frac{\partial g(v)}{\partial y_n}\frac{\partial y_n}{\partial v_n} = \alpha(y_n - y^T)\frac{\partial y_n}{\partial v_n} \\ &= \alpha(y_n - y^T)\frac{\Delta t}{1 - a\Delta t}\end{aligned}$$

To get to the second line we used the implicit Euler formula (6.7). If we then look at the scheme (6.8) for the adjoint equation, we see that:

$$\alpha(y_n - y^T)\frac{\Delta t}{1 - a\Delta t} = \Delta t\frac{p_n}{1 - a\Delta t} = \Delta t p_{n-1}$$

Using the same approach, we can find an expression for $\frac{\partial g(v)}{\partial v_i}$:

$$\begin{aligned}\frac{\partial g(v)}{\partial v_i} &= \alpha(y_n - y^T)(\prod_{k=i+1}^{n}\frac{\partial y_k}{\partial y_{k-1}})\frac{\partial y_i}{\partial v_i} = \frac{p_n}{(1 - a\Delta t)^{n-i}}\frac{\Delta t}{1 - a\Delta t} \\ &= \frac{p_n \Delta t}{(1 - a\Delta t)^{n-i+1}} = \Delta t p_{i-1}\end{aligned}$$

since v_0 is not part of the scheme, $\frac{\partial g(v)}{\partial v_0} = 0$. If we now write up the gradient of $g(v)$ on matrix form, you get $\nabla g(v) = Bp$. The expression for the gradient in the case where we use the explicit Euler scheme can be found in a similar fashion. In the case we where we are using the Crank-Nicholson scheme for ODE discretization, the algebra of differentiating g, gets slightly more complicated. Utilizing the expressions for y_{k+1} and p_{k-1} in (6.9) and (6.10), that we get from applying Crank-Nicholson to the state and adjoint equation, we are able to derive $\frac{\partial g(v)}{\partial v_i}$:

$$\begin{aligned}\frac{\partial g(v)}{\partial v_i} &= \alpha(y_n - y^T)(\frac{\partial y_i}{\partial v_i}\prod_{k=i+1}^{n}\frac{\partial y_k}{\partial y_{k-1}} + \frac{\partial y_{i+1}}{\partial v_i}\prod_{k=i+2}^{n}\frac{\partial y_k}{\partial y_{k-1}}) \\ &= p_n(\frac{\partial y_i}{\partial v_i}(\frac{1 + \frac{\Delta ta}{2}}{1 - \frac{\Delta ta}{2}})^{n-i} + \frac{\partial y_{i+1}}{\partial v_i}(\frac{1 + \frac{\Delta ta}{2}}{1 - \frac{\Delta ta}{2}})^{n-i+1}) \\ &= \frac{\Delta t}{2(1 - \frac{\Delta ta}{2})}(p_i + p_{i+1}) = \frac{\Delta t}{2}(\frac{p_{i-1}}{1 + \frac{\Delta ta}{2}} + \frac{p_{i+1}}{1 - \frac{\Delta ta}{2}})\end{aligned}$$

For $i = 1, ..., n-1$, the last expression of the above calculation is equal to the i-th component of $\frac{1}{2}(\frac{1}{1+\frac{\Delta ta}{2}}B + \frac{1}{1-\frac{\Delta ta}{2}}B^T)p_{\Delta t}$, which is what we wanted to show. By doing similar calculations we see that the Crank-Nicholson gradient stated in theorem 6.1 is also correct for $i = 0$ and $i = n$. □

6.2 Discretizing the Penalized Example Problem

In the previous section we discretized the objective function, state equation and adjoint equation of the example problem (3.8-3.9). We also derived an expression for the gradient of J. Let us now do the same for the decomposed problem (5.14-5.15). We start by restating the decomposed example ODE, and the penalized objective function.

$$\begin{cases} \frac{\partial}{\partial t} y_i(t) + a y_i(t) = v(t) \ t \in (T_{i-1}, T_i) \\ y^i(T_{i-1}) = \lambda_{i-1} \end{cases} \tag{6.14}$$

$$\hat{J}_\mu(v, \Lambda) = \frac{1}{2} \int_0^T v(t)^2 dt + \frac{\alpha}{2}(y_N(T) - y^T)^2 + \frac{\mu}{2} \sum_{i=1}^{N-1} (y_i(T_i) - \lambda_i)^2 \tag{6.15}$$

Let us also remember the gradient of (6.15) stated in (5.18):

$$\hat{J}'_\mu(v, \lambda) = (v + p, p_2(T_1) - p_1(T_1), ..., p_N(T_{N-1}) - p_N(T_{N-1})). \tag{6.16}$$

Before we can discretize the penalized objective function (6.15) and its gradient (6.16), we need to decompose the discrete time domain $I_{\Delta t} = \{t_k\}_{k=0}^n$. We do this by choosing a subsequence $\{t_{k_i}\}_{i=0}^N \subset I_{\Delta t}$, such that $t_{k_i} = T_i$. Using this subsequence we can define N decomposed discrete subintervals $I^i_{\Delta t} = \{t_{k_{i-1}}, t_{k_{i-1}+1}, ..., t_{k_i}\}$. The discrete subintervals $I^i_{\Delta t}$ contain n_i points, and we choose the subsequence $\{t_{k_i}\}$ so that n_i stays roughly the same for all i. Discretizing the decomposed ODEs is straight forward, however the solution of the state and adjoint equations now consists of independent solutions $y^i_{\Delta t}$ and $p^i_{\Delta t}$ on each subinterval $I^i_{\Delta t}$, where

$$\begin{aligned} y^i_{\Delta t} &= (y^i_{k_{i-1}}, y^i_{k_{i-1}+1}, ..., y^i_{k_i}) \text{ and} \\ p^i_{\Delta t} &= (p^i_{k_{i-1}}, p^i_{k_{i-1}+1}, ..., p^i_{k_i}), \ i = 1, ..., N. \end{aligned}$$

One problem with $y^i_{\Delta t}$ and $p^i_{\Delta t}$ existing independently on each interval, is that we get an overlap on all the subinterval boundaries, which have the potential of complicating the evaluation of the penalized numerical objective function and of its gradient. It turns out that for our example problem this problem only arises in the gradient evaluation. We can therefore quite simply write up the penalized

numerical objective function:

$$\hat{J}_{\mu,\Delta t}(v_{\Delta t},\Lambda) = \frac{1}{2}v_{\Delta t}^T M_\theta v_{\Delta t} + \frac{\alpha}{2}(y_n^N - y^T)^2 + \frac{\mu}{2}\sum_{i=1}^{N-1}(y_{k_i}^i - \lambda_i)^2 \tag{6.17}$$

$$= \Delta t\frac{\theta v_0^2 + (\theta-1)v_n^2}{2} + \frac{\Delta t}{2}\sum_{i=1}^{n-1} v_i^2 + \frac{\alpha}{2}(y_n^N - y^T)^2 + \frac{\mu}{2}\sum_{i=1}^{N-1}(y_{k_i}^i - \lambda_i)^2. \tag{6.18}$$

We now write up the gradient of the discretized objective function (6.18) in theorem 6.2 expressed in terms of the discrete adjoint $p_{\Delta t}$.

Theorem 6.2. *The gradient of (6.18), $\hat{J}_{\mu,\Delta t} : \mathbb{R}^{N+m} \to \mathbb{R}$ consists of two parts. The second part $\nabla\hat{J}_{\mu,\Delta t}(\Lambda) \in \mathbb{R}^{N-1}$ related to the virtual control is independent of the choice of finite difference scheme, and is given by:*

$$\nabla\hat{J}_{\mu,\Delta t}(\Lambda) = (p_{k_1}^2 - p_{k_1}^1, p_{k_2}^3 - p_{k_2}^2, ..., p_N^{k_{N-1}} - p_{N-1}^{k_{N-1}}). \tag{6.19}$$

The first part $\nabla\hat{J}_{\mu,\Delta t}(v_{\Delta t}) \in \mathbb{R}^{m+1}$, which is connected to the real control variable $v_{\Delta t}$, depends on the finite difference scheme used to discretize the adjoint and state equations. If we use the implicit Euler scheme to evaluate (6.18), the $v_{\Delta t}$ part of the gradient will be:

$$\nabla\hat{J}_{\mu,\Delta t}(v_{\Delta t}) = M_0 v_{\Delta t} + (B^1 p_{\Delta t}^1, B^2 p_{\Delta t}^2, ..., B^N p_{\Delta t}^N), \tag{6.20}$$

where $M_\theta \in \mathbb{R}^{(n+1)\times(n+1)}$ is the matrix defined in theorem 6.1, and $B^i \in \mathbb{R}^{n^i\times(n^i-1)}$, for $i > 1$ and $B^1 \in \mathbb{R}^{n^i\times(n^i)}$ are the matrices defined below. $n^i = k_i - k_{i-1}$ here means the length of vector $p_{\Delta t}^i$.

$$B^1 = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \Delta t & 0 & 0 & \cdots \\ 0 & \Delta t & 0 & \cdots \\ 0 & \cdots & \Delta t & 0 \end{bmatrix}, B^i = \begin{bmatrix} \Delta t & 0 & \cdots & 0 \\ 0 & \Delta t & 0 & \cdots \\ 0 & \cdots & \Delta t & 0 \end{bmatrix}.$$

If one instead uses the explicit Euler finite difference scheme on the differential equations, the gradient will instead look like:

$$\nabla\hat{J}_{\mu,\Delta t}(v_{\Delta t}) = M_1 v_{\Delta t} + (\bar{B}^1 p_{\Delta t}^1, \bar{B}^2 p_{\Delta t}^2, ..., \bar{B}^N p_{\Delta t}^N), \tag{6.21}$$

where $\bar{B}^i \in \mathbb{R}^{n^i\times(n^i-1)}$ for $i < N$, and $\bar{B}^1 \in \mathbb{R}^{n^i\times(n^i)}$ are defined as:

$$\bar{B}^i = \begin{bmatrix} 0 & \Delta t & 0 & \cdots \\ 0 & 0 & \Delta t & \cdots \\ 0 & \cdots & 0 & \Delta t \end{bmatrix}, \bar{B}^N = \begin{bmatrix} 0 & \Delta t & \cdots & 0 \\ 0 & 0 & \Delta t & \cdots \\ 0 & 0 & 0 & \Delta t \\ 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Finally the gradient of the discrete objective function, in the case where we use Crank-Nicholson to dicretize the ODEs is:

$$\nabla \hat{J}_{\mu,\Delta t}(v_{\Delta t}) = M_{\frac{1}{2}} v_{\Delta t} + \frac{1}{2}(\frac{1}{1+\Delta t a} B p_{\Delta t} + \frac{1}{1-\Delta t a}\bar{B} p_{\Delta t}).$$

Here $B, \bar{B} \in \mathbb{R}^{n+N \times n+1}$ are matrices, which we can define using block notation:

$$B = \begin{bmatrix} B^1 & 0 & \cdots & 0 \\ 0 & B^2 & 0 & \cdots \\ 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & B^N \end{bmatrix}, \bar{B} = \begin{bmatrix} \bar{B}^1 & 0 & \cdots & 0 \\ 0 & \bar{B}^2 & 0 & \cdots \\ 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \bar{B}^N \end{bmatrix}.$$

By $p_{\Delta t} \in \mathbb{R}^{n+N}$ we mean the vector $p_{\Delta t} = (p^1_{\Delta t}, p^2_{\Delta t}, ..., p^N_{\Delta t})$

Proof. Let us begin with the Λ part of the gradient. We find each component by differentiating $\hat{J}_{\mu,\Delta t}$ with respect to λ_i, for $i = 1, ..., N-1$. It turns out there are two cases, namely $i = N-1$ and $i \neq N-1$, these cases are however quite similar, so we will only do the $i \neq N-1$ case. For each $i = 1, ..., N-2$, there are only two terms in $\hat{J}_{\mu,\Delta t}$ that depend on λ_i, and these are λ_i itself and $y^{i+1}_{k_{i+1}}$. With this in mind let us start to differentiate $\hat{J}_{\mu,\Delta t}$.

$$\begin{aligned} \frac{\partial \hat{J}_{\mu,\Delta t}}{\partial \lambda_i}(v_{\Delta t}, \Lambda) &= -\mu(y^i_{k_i} - \lambda_i) + \mu(y^{i+1}_{k_{i+1}} - \lambda_{i+1}) \frac{\partial y^{i+1}_{k_{i+1}}}{\partial \lambda_i} \\ &= \mu(y^{i+1}_{k_{i+1}} - \lambda_{i+1})(\frac{1}{1-a\Delta t})^{k_{i+1}-k_i} - \mu(y^i_{k_i} - \lambda_i). \end{aligned}$$

To get the $(\frac{1}{1-a\Delta t})^{k_{i+1}-k_i}$ term we used the chain rule on $\frac{\partial y^{i+1}_{k_{i+1}}}{\partial \lambda_i}$ and the implicit Euler scheme for our particular equation given in (6.7). The next step is done by noticing that the terms $\mu(y^i_{k_i} - \lambda_i)$ and $\mu(y^{i+1}_{k_{i+1}} - \lambda_{i+1})$ are the initial conditions of the i-th and $i+1$-th adjoint equations, which means that $\mu(y^i_{k_i} - \lambda_i) = p^i_{k_i}$ and $\mu(y^{i+1}_{k_{i+1}} - \lambda_{i+1}) = p^{i+1}_{k_{i+1}}$. Inserting this we get:

$$\begin{aligned} \frac{\partial \hat{J}_{\mu,\Delta t}}{\partial \lambda_i}(v_{\Delta t}, \Lambda) &= p^{i+1}_{k_{i+1}}(\frac{1}{1-a\Delta t})^{k_{i+1}-k_i} - p^i_{k_i} \\ &= p^{i+1}_{k_i} - p^i_{k_i}. \end{aligned}$$

The last step is done by utilizing the implicit Euler scheme for our adjoint equation (6.8).

The $v_{\Delta t}$ part of the gradient is almost equal to the non-penalized gradient, the

only difference being that the adjoint now is defined separately on each subinterval and not on the entire time interval $[0, T]$. We can again divide the functional (6.18) into two parts, the integral over $v_{\Delta t}$, $f(v_{\Delta t}) = \frac{1}{2} v_{\Delta t}^* M_\theta v_{\Delta t}$ and

$$g(v_{\Delta t}) = \frac{\alpha}{2}(y_n^N - y^T)^2 + \frac{\mu}{2}\sum_{i=1}^{N}(y_{k_i}^i - \lambda_i)^2.$$

As for the non-penalized gradient, the derivative of the f term is quite easily seen to be $M_\theta v_{\Delta t}$, the problems start when we want to differentiate g with respect to a specific component v_k in $v_{\Delta t}$. If we are using the implicit Euler scheme to dicretize the state and adjoint equations, the k-th component of $v_{\Delta t}$ only affects the solution of one of the n state equations. If $k \in \{k_{i-1}+1, k_{i-1}+2, ..., k_i\}$, v_k is used to find $y_{\Delta t}^i$, which means that the only term in g, that depend on v_k, is $\frac{\mu}{2}(y_{k_i}^i - \lambda_i)^2$ if $i \neq N$, or $\frac{1}{2}(y_n^N - y^T)^2$ if $i = N$. If we now assume that $i \neq N$ and $k \in \{k_{i-1}+1, k_{i-1}+2, ..., k_i\}$, we can differentiate g with respect to v_k:

$$\begin{aligned}\frac{\partial g}{\partial v_k} &= \mu(y_{k_i}^i - \lambda_i)\Big(\prod_{l=k+1}^{k_{i+1}} \frac{\partial y_l}{\partial y_{l-1}}\Big)\frac{\partial y_k}{\partial v_k} = \frac{p_{k_i}^i}{(1-a\Delta t)^{k_i-k}}\frac{\Delta t}{1-a\Delta t} \\ &= \frac{p_{k_i}^i \Delta t}{(1-a\Delta t)^{k_i-k+1}} = \Delta t p_{k-1}^i.\end{aligned}$$

The numerical gradient restricted to $\{k_{i-1}+1, k_{i-1}+2, ..., k_i\}$, will then be $B^i p_{\Delta t}^i$, which exactly what we claimed. □

6.3 Parallelization of Function and Gradient Evaluation

The most computationally costly part of algorithm 5.2 is evaluating the penalized objective function and its gradient. These evaluations are needed to find the search direction and step length in the BFGS line search method. In this section we present parallel algorithms for the evaluation of the penalized objective function and its gradient, in a setting where we assume no shared memory between the processes. We will in particular focus on the communication that takes place between the processes, since the communication steps are important for understanding the performance of the algorithms. The function evaluation requires us to solve the state equation, while the calculation of the gradient needs both the solution of the state and adjoint equation. The algorithms for function and gradient evaluation are obviously different, however, they both share the same starting point, which we explain below.

Let us assume that we have N processes, which we name $\{P_i\}_{i=0}^{N-1}$. Then assume that each process P_i only knows the parts of the control that are required for the process to solve the state equation and to locally evaluate the objective function. This also includes the the virtual control variables $\{\lambda_i\}_{i=1}^{N-1}$. To make it simple let us also assume that there is no overlap in the real control between the processes, which is the case for explicit and implicit Euler discretizations of the state and adjoint equations, but not for Crank-Nicolson discretizations. After each process P_i has solved their part of the state equation, they all have the following data stored locally:

$$\begin{aligned}
&\textit{Control variable:} \quad v_{i+1} \\
&\textit{Penalty control variable:} \quad \lambda_i \\
&\textit{Solution to local state equation:} \quad y^{i+1} = \{y_j^{i+1}\}_{j=k_i}^{k_{i+1}}
\end{aligned}$$

Using this data we should be able to evaluate the penalized objective function, or to calculate its gradient.

6.3.1 Parallel Algorithm for Objective Function Evaluation

The penalized objective function consists of two parts:

$$\hat{J}_\mu(v, \lambda) = \hat{J}(y(v), v) + \frac{\mu}{2} \sum_{j=1}^{N-1} (y^j(T_j) - \lambda_j)^2.$$

Let us begin with the penalty term. Each process P_i only have λ_i and $y^{i+1}(T_{i+1})$ stored locally. This means that to calculate all penalty terms the processes will have to send either λ_i or $y^{i+1}(T_{i+1})$ to one of its neighbours. For example P_i could send λ_i to P_{i-1} for $i = 1, ..., N-1$:

$$P_0 \xleftarrow{\lambda_1} P_1 \xleftarrow{\lambda_2} P_2 \xleftarrow{\lambda_3} \cdots \xleftarrow{\lambda_{N-2}} P_{N-2} \xleftarrow{\lambda_{N-1}} P_{N-1}$$

For the evaluation of $\hat{J}(y(v), v)$, let us assume that there exists functions $\hat{J}^{i+1}(y^{i+1}(v_{i+1}), v_{i+1})$, such that:

$$\hat{J}(y(v), v) = \sum_{j=1}^{N} \hat{J}^j(y^j(v_j), v_j).$$

If this is the case we can evaluate each part of the objective function locally, and then get the global $\hat{J}_\mu$ by doing one summation reduction. The penalized objective

function evaluation algorithm is:

Algorithm 6.1: Parallel objective function evaluation

Data: Partitioned control variable (v_{i+1}, λ_i) given as input to each process P_i for $i = 0, ..., N-1$.

begin

Process P_i solve state equation y^{i+1} using (v_{i+1}, λ_i)`// In parallel`;

for $i = 1, ..., N-1$ **do**

$P_{i-1} \xleftarrow{\lambda_i} P_i$;

end

`// Evaluate local objective function` $\hat{J}_\mu^i$ `in parallel`;

if $i == N-1$ **then**

$\hat{J}_\mu^N(y^N(v_N), v_N) \leftarrow \hat{J}^N(y^N(v_N), v_N)$;

else

$\hat{J}_\mu^{i+1}(y^{i+1}(v_{i+1}), v_{i+1}) \leftarrow \hat{J}^{i+1}(y^{i+1}(v_{i+1}), v_{i+1}) + \frac{\mu}{2}(y^{i+1}(T_{i+1}) - \lambda_{i+1})^2$

end

$\hat{J}_\mu(y(u), u) \leftarrow$ **MPI_Reduce**$(\hat{J}_\mu^{i+1}, +)$

end

6.3.2 Parallel Algorithm for Gradient Evaluation

The gradient of the penalized optimal control problem looks like the following:

$$\nabla \hat{J}_\mu(v, \lambda) = (J_v(y(v), v) - B^* p, \{p_{i+1}(T_i) - p_i(T_i)\}_{i=1}^{N-1}).$$

p is here the solution to the adjoint equation, which has to be calculated before we can evaluate the gradient, and $B = E_y(y, v, \Lambda)$. For processes P_i, $i < N-1$, the initial condition of the adjoint equation is $p^{i+1}(T_{i+1}) = \mu(y^{i+1}(T_{i+1} - \lambda_{i+1})$. This means that the first step after solving the state equations for gradient evaluation, is the same as for function evaluation, i.e. we have to send λ_i from P_i to P_{i-1}:

$$P_0 \xleftarrow{\lambda_1} P_1 \xleftarrow{\lambda_2} P_2 \xleftarrow{\lambda_3} \cdots \xleftarrow{\lambda_{N-2}} P_{N-2} \xleftarrow{\lambda_{N-1}} P_{N-1}$$

Each process can now solve its adjoint equation locally, and we can start to actually evaluate the gradient. The first step, would be to send $p_{i+1}(T_i)$ from P_i to P_{i-1} so that we can find the penalty part of the gradient. Each process should also be able to calculate their own part of the gradient as $\nabla \hat{J}^{i+1} = (J_v(y^{i+1}(v_{i+1}), v_{i+1}) - B_{i+1}^* p^{i+1})$. The final step is now to gather all the local parts of the gradient to the form the actual gradient. In summation we get the following algorithm for gradient evaluation:

Algorithm 6.2: Parallel gradient evaluation

Data: Partitioned control variable (v_{i+1}, λ_i) given as input to each process P_i for $i = 0, ..., N-1$.

begin

Process P_i solve state equation y^{i+1} using (v_{i+1}, λ_i)`// In parallel`;

for $i = 1, ..., N-1$ **do**

$P_{i-1} \xleftarrow{\lambda_i} P_i$;

end

Process P_i solve adjoint equation p^{i+1} using (y_{i+1}, λ_{i+1})`// In parallel`;

for $i = 1, ..., N-1$ **do**

$P_{i-1} \xleftarrow{p^{i+1}(T_i)} P_i$;

end

`// Evaluate local gradient` $\nabla \hat{J}_\mu^i$ `in parallel`;

$\nabla \hat{J}_{v_{i+1}}^{i+1} \leftarrow J_v(y^{i+1}(v_{i+1}), v_{i+1}) - B_{i+1}^* p^{i+1}$;

if $i \neq N-1$ **then**

$\nabla \hat{J}_{\lambda_{i+1}}^{i+1} \leftarrow p_{i+2}(T_{i+1}) - p_{i+1}(T_{i+1})$;

end

$\nabla \hat{J}_\mu \leftarrow$ **MPI_Gather**$(\nabla \hat{J}^{i+1}, p_{i+1}(T_i) - p_i(T_i))$;

end

6.4 Analysing Theoretical Parallel Performance

Now that we know what type of communication is involved in objective function evaluation and gradient computation, we can try to model the expected performance of the two algorithms. One way to measure performance of algorithms is to look at their execution times. Therefore let us define T_s as execution time of the sequential algorithm, and T_p as parallel algorithm execution time. Let us also define the speedup $S = \frac{T_s}{T_p}$. Since we for now are only modelling performance we do not actually calculate the execution times, but we do know that the run time of the algorithms are related to the size of the problem, meaning the number of time steps n. The final thing we need before we start our performance analysis, is a way to model communication between two processes. One way of modelling the communication time T_c for sending a message of size m between to processes, is proposed in [50] as:

$$T_c = T_l + mT_w$$

Here T_l is a constant representing latency or startup time, while T_w is a constant representing the per message-unit transfer time. With these tools, we can now start analysing the performance of our algorithms.

6.4.1 Objective Function Evaluation Speedup

To evaluate the objective function, we first need to solve the state equation. If we have discretized the state equation using $n+1$ time steps, evolving the state equation requires $\mathcal{O}(n)$ operations. The next step is then to apply the functional on the control and the state, which we assume at most requires $\mathcal{O}(n)$ operations. The sequential objective function evaluation execution time is therefore:

$$T_s = \mathcal{O}(n).$$

In our parallel algorithm we also solve the state equation and apply the functional, but since we divide the time steps equally between all processes, solving the state equation and applying the functional only requires $\mathcal{O}(\frac{n}{N})$ operations. Since we also have penalty terms in our functional we get additional $\mathcal{O}(N)$ operations. Now for the communication. We are doing two communication steps one is sharing the λs between process neighbours, and the other is reducing the local function values into one global function value. The send and receive time is given by $T_c = T_l + dim(\lambda_i)T_w$, which requires $\mathcal{O}(1)$ operations, while the reduction time T_{red} can be modelled as:

$$\begin{aligned} T_{red} &= \log N(T_l + T_w) \\ &= \mathcal{O}(\log N). \end{aligned}$$

Here we assume that the parallel architecture is made in a certain way, and that the local functional value is a floating point. This results in parallel function evaluation execution time:

$$\begin{aligned} T_p &= \mathcal{O}(\frac{n}{N}) + \mathcal{O}(N) + \mathcal{O}(\log N) + \mathcal{O}(1) \\ &= \mathcal{O}(\frac{n}{N}) + \mathcal{O}(N). \end{aligned}$$

The speedup is then:

$$\begin{aligned} S = \frac{T_s}{T_p} &= \frac{\mathcal{O}(n)}{\mathcal{O}(\frac{n}{N}) + \mathcal{O}(N)} \\ &= \mathcal{O}(N). \end{aligned}$$

This is an optimal speedup.

6.4.2 Gradient Evaluation Speedup

When we calculate the objective function gradient sequentially, we solve both the state and adjoint equations. The required operations are however still in the order of number of time steps, i.e:

$$T_s = \mathcal{O}(n).$$

For the parallel algorithm the operations required to solve the local state and adjoint equations are $\mathcal{O}(\frac{n}{N})$. We then have two $\mathcal{O}(1)$ send-receive communications similar to the send and receive for function evaluation. Lastly we need to model the gathering of the gradient. First define L to be the length of the gradient. The run time of the gather T_{gather}, can then be modelled as:

$$\begin{aligned} T_{gather} &= T_l \log N + \frac{L}{N} T_w (N-1) \\ &= \mathcal{O}(\log N) + \mathcal{O}(L). \end{aligned}$$

The execution time of the parallel algorithm is therefore:

$$T_p = \mathcal{O}(\frac{n}{N}) + \mathcal{O}(\log N) + \mathcal{O}(L).$$

Again we find the speedup by dividing T_s by T_p:

$$\begin{aligned} S = \frac{T_s}{T_p} &= \frac{\mathcal{O}(n)}{\mathcal{O}(\frac{n}{N}) + \mathcal{O}(\log N) + \mathcal{O}(L)} \\ &= \frac{1}{\frac{1}{N} + \frac{\log N}{n} + \frac{L}{n}} = \frac{1}{\frac{1}{N} + \frac{L}{n}}. \end{aligned}$$

If L is independent of n, the speedup for gradient evaluation is $\mathcal{O}(N)$, like it is for function evaluation, however if L is dependent on n, this is not the case, and we would instead get speedup $S = \mathcal{O}(\frac{n}{L(n)})$. In a case where the control for example is the source term in the state equation, we would actually get $S = \mathcal{O}(1)$, which is really bad, and we would not expect any improvement when using parallel, at least for large n values. There is however a way to get around this problem, which is to store both the gradient and the control locally, which means that you never have to do a gather call. If this is done, and if a solution spread between all processes is accepted, the speedup for gradient evaluation will also be $\mathcal{O}(N)$.

Chapter 7

Verification

In this chapter we will verify implementations of the algorithm presented in chapter 5 using the discretization detailed in chapter 6. All implementations are done in the Python programming language, and the numerics are done using the NumPy [51] library. Plots are created using the matplotlib [52] package, tables are auto generated using Pandas [53] and the parallel parts are implemented using the mpi4py [54] library. We test our algorithm using the example problem (3.8-3.9), with the following parameters:

$$J(y, v) = \frac{1}{2}\int_0^1 v(t)^2 dt + \frac{1}{2}(y(1) - 11.5)^2 \tag{7.1}$$

$$\begin{cases} y'(t) = -3.9y(t) + v(t) \; t \in (0, 1) \\ y(0) = 3.2 \end{cases} \tag{7.2}$$

Using this problem we will first test the numerical gradients stated in section 6.1.2 and 6.2, and then investigate if the minimizer of the discretized objective function converges to the exact minimizer derived in section 3.2.2. We also check if the theoretical speedup for objective function and gradient evaluation suggested in 6.4 is in line with actual measurements. The last test done is on the consistency of the penalty framework.

7.1 Taylor Test

The Taylor test is a good way to test the correctness of the gradient of a function. The test is as its name implies connected with Taylor expansions of a function, or more precisely the following two observations:

$$|J(v + \epsilon w) - J(v)| = \mathcal{O}(\epsilon)$$
$$|J(v + \epsilon w) - J(v) - \epsilon \nabla J(v) \cdot w| = \mathcal{O}(\epsilon^2)$$

Here w is a vector in the same space as v, while $\epsilon > 0$ is a constant. The test is carried out by evaluating $D = |J(v + \epsilon w) - J(v) - \epsilon \nabla J(v) \cdot w|$ for decreasing ϵ's, and if D approaches 0 at a 2nd order rate, we consider the test as passed.

7.1.1 Verifying the Numerical Gradient Using the Taylor Test

We will now use the Taylor test on the discrete gradient stemming from problem (7.1-7.2). We discretize this problem using the Crank-Nicolson scheme for the state and adjoint equation, and the trapezoid rule for numerical integration, as suggested in chapter 6. We let the time step be $\Delta t = \frac{1}{100}$, and evaluate the objective function and its gradient using the control variable $v = 1$. To apply the Taylor test, we need a direction $w \in \mathbb{R}^{101}$, which we set to be a vector with components randomly chosen from numbers between 0 and 100. To make table 7.1 more readable we define the following measures:

$$D_1(\epsilon) = |J(v + \epsilon w) - J(v)| \tag{7.3}$$
$$D_2(\epsilon) = |J(v + \epsilon w) - J(v) - \epsilon \nabla J(v) \cdot w| \tag{7.4}$$

We evaluate $D_1(\epsilon)$ and $D_2(\epsilon)$ for decreasing ϵ's, and list the results in table 7.1.

Table 7.1: Results from Taylor test conducted on a non-penalized discrete objective function. ϵ is the decreasing parameter used in $D_1(\epsilon)$ (7.3) and $D_2(\epsilon)$ (7.4). The last two columns of the table show the rate at which $D_1(\epsilon)$ and $D_2(\epsilon)$ approach zero for decreasing ϵ. If $D_2(\epsilon)$ converges to zero at second order rate the test is passed, and this is indeed observed in the results of the table.

ϵ	D_1	D_2	$\|\|\epsilon w\|\|_{l_\infty}$	$\log(\frac{D_1(\epsilon)}{D_1(10\epsilon)})$	$\log(\frac{D_2(\epsilon)}{D_2(10\epsilon)})$
1e+00	5.95e+3	5.24e+03	9.99e+1	–	–
1e-01	1.23e+2	5.24e+01	9.99e+0	-1.7	-2
1e-02	7.64e+0	5.24e-01	9.99e-1	-1.2	-2
1e-03	7.17e-1	5.24e-03	9.99e-2	-1.0	-2
1e-04	7.12e-2	5.24e-05	9.99e-3	-1.0	-2
1e-05	7.12e-3	5.24e-07	1.00e-3	-1.0	-2
1e-06	7.12e-4	5.24e-09	1.00e-4	-1.0	-2
1e-07	7.10e-5	5.25e-11	1.00e-5	-1.0	-2

Table 7.1 clearly shows that $|J(v + \epsilon w) - J(v) - \epsilon \nabla J(v) \cdot w|$ converges to zero at a second order rate. This means that the numerical gradient of our test problem passes the Taylor test. This again indicates that both the numerical gradient and the implementation of it are correct. We include a plot of the gradient used in the Taylor test together with a finite difference approximation of $\hat{J}'(v)$ in figure 7.1. We see that the discrete gradient form section 6.1 is practically identical to the finite difference approximation. Notice also how the discrete gradient have jumps in its first and last component. These jumps come from the discretization method used on the integral, the state equation and adjoint equation. Next we check whether the gradient of the discrete and penalized objective function passes the Taylor test.

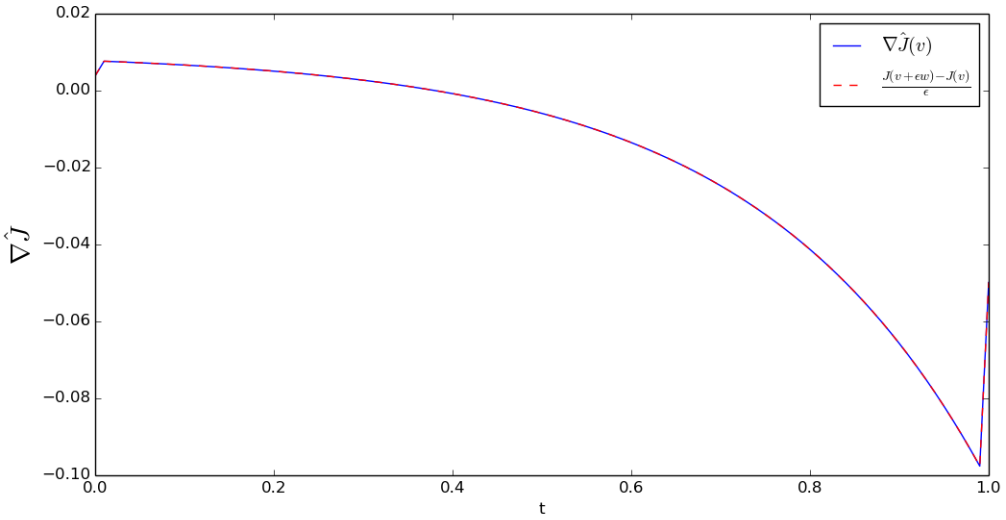


Figure 7.1: Gradient of non-penalized objective function of problem (7.1-7.2) calculated using expression from section 6.1.2, and a finite difference approximation. Notice that these methods produce almost identical results. The jumps at $t = 0$ and $t = 1$ originates from the Crank-Nicolson and trapezoid rule discretization of the objective function.

7.1.2 Verifying the Penalized Numerical Gradient Using the Taylor Test

We will now use the Taylor test on the penalized numerical gradient (6.19-6.20) that we get when decomposing $I = [0, T]$ into $N = 10$ subintervals while solving the same problem as in the test for the gradient of the non-penalized objective function (7.1-7.2). We then discretize in time using $\Delta t = \frac{1}{100}$. The control variable is now a vector $v \in \mathbb{R}^{N+m}$ and we set $v_k = 0 \ \forall k = 0, ..., N + n - 1$, while

the w_ks are chosen randomly from numbers between 0 and 100. The results of applying the Taylor test to this problem are given in table 7.2. Here D_1 and D_2 are again defined as in (7.3-7.4).

Table 7.2: Taylor test for penalized discrete objective function. We are again interested in at what rate $D_1(\epsilon)$ and $D_2(\epsilon)$ converge to zero, when we decrease ϵ. We observe that $D_1(\epsilon)$ approaches zero at a first order rate, while $D_2(\epsilon)$ converges at a second order rate. Since $D_2(\epsilon)$ vanishes at second order rate, the Taylor test is passed.

ϵ	D_1	D_2	$\|\|\epsilon w\|\|_{l_\infty}$	$\log(\frac{D_1(\epsilon)}{D_1(10\epsilon)})$	$\log(\frac{D_2(\epsilon)}{D_2(10\epsilon)})$
1e+00	1.08e+04	1.07e+04	9.77e+01	–	–
1e-01	1.11e+02	1.07e+02	9.77e+00	-1.98	-2
1e-02	1.43e+00	1.07e+00	9.77e-01	-1.88	-2
1e-03	4.68e-02	1.07e-02	9.77e-02	-1.48	-2
1e-04	3.71e-03	1.07e-04	9.77e-03	-1.10	-2
1e-05	3.61e-04	1.07e-06	9.77e-04	-1.01	-2
1e-06	3.60e-05	1.07e-08	9.77e-05	-1.00	-2
1e-07	3.60e-06	1.07e-10	9.77e-06	-1.00	-2
1e-08	3.60e-07	1.08e-12	9.77e-07	-1.00	-2

Again we see that $|J(v + \epsilon w) - J(v) - \epsilon \nabla J(v) \cdot w|$ converges to zero at a second order rate, meaning that the penalized numerical gradient also passes the Taylor test. In figure 7.2 we see how the penalized and decomposed gradient tested with the Taylor test looks like. Notice that there are jumps between the decomposed subintervals, which is expected, since the intermediate initial conditions were set equal to one. We observe that the real part of the gradient in figure 7.2 looks nothing like the gradient in figure 7.1. This is because the adjoint equation of the decomposed problem is defined separately on each interval.

7.2 Convergence Rate of the Sequential Algorithm

In section 7.1 we demonstrated that our implementation of the gradients for different discretizations of the objective function introduced in theorem 6.1 and 6.2 satisfy the Taylor test. Since the discretized objective function $\hat{J}_{\Delta t}$ and its gradient

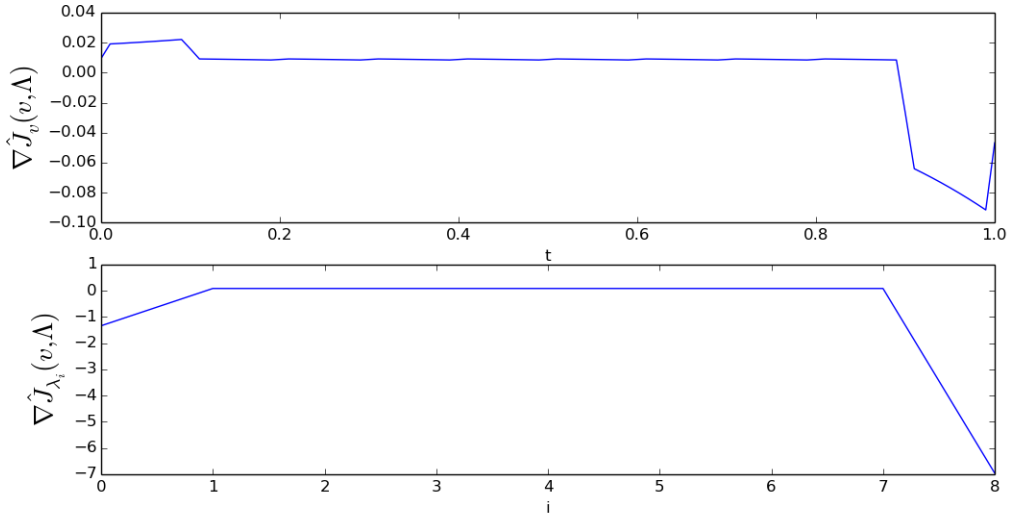


Figure 7.2: Plots showing the real and virtual part of the numerical gradient of the decomposed and penalized objective function of problem (7.1-7.1) found using formula (6.19-6.20). Notice that the real part of the gradient has jumps between subintervals.

pass the Taylor test, we expect that we can find the minimizer $\bar{v}$ of $\hat{J}_{\Delta t}$ by using algorithm 3.1 from chapter 3. What we now want to investigate, is whether the minimizer of the discrete objective function converges towards the exact minimizer derived in section 3.2.2. We test this by solving optimal control problem (7.1-7.2) using algorithm 3.1 with both a Crank-Nicolson and an implicit Euler discretization of the state and adjoint equations. To measure the difference between exact optimal control v_e and the numerical optimal control v we look at the relative maximal difference between v_e and v for $t \in (0, T)$, meaning

$$||v|| = \max_{k=1,\ldots,n-1} |v_k| \tag{7.5}$$

We also look at the relative difference in objective function value between the controls. For both these measures, we calculate at what rate they converge to zero for decreasing Δt values. The results for the implicit Euler discretization is found in table 7.3, while Crank-Nicolson results are given in table 7.4.

Notice that the convergence rate of the norm difference in table 7.3 approaches one when Δt tends to zero. This is consistent with what we would expect for a finite difference scheme of first order. We also notice that the difference in function value converges an order of one faster towards zero than the control difference.

Table 7.3: Convergence of algorithm 3.1 using an implicit Euler discretization. For each Δt we find a numerical solution v to problem (7.1-7.2) using the L-BFGS optimization algorithm. We measure the difference between v and the exact solution v_e by using the norm (7.5). We also look at the difference in objective function value between v and v_e. In the columns (norm rate) and (functional rate), we report at which rate these error measures approach zero. We observe that the norm approaches zero at a rate of one, while the difference in functional value converge towards zero at a second order rate.

Δt	$\frac{\|\|v_e - v\|\|}{\|\|v\|\|}$	$\frac{\hat{J}(v_e) - \hat{J}(v)}{\hat{J}(v_e)}$	norm rate	functional rate
0.02000	0.2126	1.70e-02	–	–
0.01000	0.1360	4.50e-03	-0.64	-1.92
0.00100	0.0174	4.70e-05	-0.89	-1.98
0.00010	0.0018	4.72e-07	-0.99	-1.99
0.00001	0.0002	4.72e-09	-1.00	-1.99

Table 7.4: Convergence of algorithm 3.1 using a Crank-Nicolson discretization. The columns mean the same as in table 7.3. For Crank-Nicolson we observe a second and a third order rate of convergence for the norm and functional difference when Δt goes to zero.

Δt	$\frac{\|\|v_e - v\|\|}{\|\|v\|\|}$	$\frac{\hat{J}(v_e) - \hat{J}(v)}{\hat{J}(v_e)}$	norm rate	functional rate
0.02000	4.17e-02	2.30e-03	–	–
0.01000	1.10e-02	3.38e-04	-1.91	-2.77
0.00100	1.18e-04	3.93e-07	-1.96	-2.93
0.00010	1.42e-06	3.99e-10	-1.92	-2.99
0.00001	1.48e-08	3.97e-13	-1.98	-3.00

Table 7.4 present results similar to the ones in table 7.3, however the convergence rates using a Crank-Nicolson scheme to discretize the ODEs are one order higher than the rates we got using implicit Euler. This is again expected since the Crank-Nicolson scheme is of order two. In both tables we observe that $\frac{\hat{J}(v_e) - \hat{J}(v)}{\hat{J}(v)}$ is always positive, which means that $\hat{J}(v_e) > \hat{J}(v)$. This makes sense, since $\hat{J}$ here means the discrete objective function, and v is the minimum of this function, while v_e is the minimum of the continuous objective function.

7.3 Verifying Function and Gradient Evaluation Speedups

In 6.4 we derived the theoretical speedup for numerical gradient and objective function evaluation when decomposing the time-interval. It would now be interesting to check if the implementation achieves the theoretical speedup for our example problem (7.1-7.2). Now let us explain the experimental setting. A computer with 6 cores was used to verify the results of section 6.4. Having 6 cores means that we can do gradient and function evaluation for $N = 1, 2, ..., 6$ decompositions with different time step sizes Δt. For each combination of N and Δt, we will run the function and gradient evaluations ten times, and then choose the the smallest execution time produced by the ten runs. The speedup is then calculated by dividing the sequential execution time by the parallel execution time. Tables 7.5-7.8 below present runtime and speedup for both gradient and function evaluation for different Δts and Ns. All evaluations are done with control input $v = 1$ and $\lambda_i = 1$.

Since the parallel algorithm has some overhead, we do not expect any improvements for small problems. This is reflected in table 7.5 and 7.6, where we for in table 7.5 observe an increased execution time when running function and gradient evaluation in parallel. In table 7.6 we see only a modest speedup, that is significantly lower than the expected speedup from section 6.4. In table 7.7 and 7.8 however, where $\Delta t \leq 10^{-5}$, we observe speedup results in line with what we expect from the theory. In these tables the speedups for function and gradient evaluation are close to N. We do however observe, that the speedups in particular for function evaluation is slightly lower than expected. This can be caused by parallel overhead or a suboptimal implementation.

Table 7.5: Measuring the performance of algorithm 6.1 and 6.2, for $N = 2, ..., 6$ and $\Delta t = 10^{-2}$.

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.00019	0.00021	1.000	1.000
2	0.00020	0.00024	0.946	0.875
3	0.00025	0.00028	0.780	0.753
4	0.00030	0.00034	0.642	0.632
5	0.00036	0.00039	0.544	0.547
6	0.00045	0.00045	0.427	0.480

Table 7.6: Same as table 7.5, but now $\Delta t = 10^{-4}$.

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.0088	0.0150	1.000	1.000
2	0.0044	0.0077	1.983	1.946
3	0.0031	0.0053	2.838	2.816
4	0.0024	0.0040	3.582	3.677
5	0.0020	0.0033	4.267	4.457
6	0.0019	0.0030	4.519	4.978

Table 7.7: Same as table 7.5, but now$\Delta t = 10^{-5}$.

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.0874	0.1548	1.000	1.000
2	0.0435	0.0756	2.006	2.046
3	0.0302	0.0521	2.888	2.971
4	0.0223	0.0386	3.913	4.003
5	0.0180	0.0314	4.848	4.921
6	0.0161	0.0269	5.425	5.755

Table 7.8: Same as table 7.5, but now$\Delta t = 10^{-7}$.

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	8.350	14.930	1.000	1.000
2	4.200	7.233	1.987	2.064
3	2.932	5.033	2.847	2.966
4	2.190	3.861	3.812	3.866
5	1.796	3.089	4.647	4.833
6	1.524	2.599	5.479	5.744

Anther interesting observation about tables 7.5-7.8, is that the gradient evaluation time is consistently less than two times greater than a function evaluation. Since evaluating the gradient requires us to solve both the state and adjoint equation, while function evaluation only requires a state equation solve, one might expect

that the execution time of a gradient evaluation would be exactly two times greater then the execution time of a function evaluation. This is however not the case, and the reason is that even though both the state and adjoint equations are linear, the state equation is still more computationally costly since it unlike the adjoint equation is non-homogeneous.

7.4 Consistency of the Penalty Method

When we introduced the penalty method in section 5.2, we also presented a result showing that the iterates $\{v^k\}$ stemming from the penalty algorithmic framework converged towards the solution of the non-penalized problem v if (v^k, Λ^k) is the exact global minimizer of $\hat{J}_{\mu_k}$ for all k. We can write this up as:

$$\lim_{k\to\infty} v^k = v$$

An alternative way of looking at this, is to let v^μ be the minimizer of $\hat{J}_\mu$, and instead write the above limit as:

$$\lim_{\mu\to\infty} v^\mu = v \tag{7.6}$$

The interpretation of the above limit, is that solving the penalized problem with an ever increasing penalty parameter μ should result in a solution that is getting closer and closer to the solution of the non-penalized problem. This means that the penalty algorithm is consistent, since it produces the same solution as the ordinary non-decomposed problem. It is therefore worth checking if the implementation of the penalized problem actually has the property (7.6). We investigate this by comparing the numerical solution we get by applying algorithm 5.2 on the decomposed problem (7.1 - 7.2) with the numerical solution we get by solving the undecomposed example problem with the sequential algorithm 3.1.

We discretize (7.1-7.2) using Crank-Nicolson and the trapezoid rule for two different time steps. First we let $\Delta t = 10^{-2}$ and apply the penalty method for $N = 2$ and $N = 10$ decompositions, we then let $\Delta t = 10^{-3}$ and test the penalty method on $N = 2$ and $N = 7$ decompositions. We use different metrics to compare the non-penalized and penalized solutions, so that we better see how the solution of the penalized problem behaves when we solve it for an increasing sequence of μ

values. We define the metrics as follows:

$$\text{Realtive objective function difference:} \quad \Delta\hat{J} = \frac{\hat{J}(v_\mu) - \hat{J}(v)}{\hat{J}(v)}. \tag{7.7}$$

$$\text{Realtive penalized objective function difference:} \quad \Delta\hat{J}_\mu = \frac{\hat{J}_\mu(v) - \hat{J}_\mu(v_\mu)}{\hat{J}_\mu(v)}. \tag{7.8}$$

$$\text{Relative control } L^2\text{-norm difference:} \quad \Delta v = \frac{||v_\mu - v||_{L^2}}{||v||_{L^2}}. \tag{7.9}$$

$$\text{Maximal jump in decomposed state equation:} \quad \Delta y = \sup_i \{y^i_{j_i} - y^{i+1}_{j_i}\}. \tag{7.10}$$

Notice that both $\Delta\hat{J}$ and $\Delta\hat{J}_\mu$ should be grater than 0, since v and v_μ are the minimizers of $\hat{J}$ and $\hat{J}_\mu$. The measure of jumps in the state equation Δy is added to check that the penalty solution approaches a feasible solution in context of the continuity constraints (5.5). The results of the above detailed experiment are presented through logarithmic plots in figure 7.3 and 7.4. In addition to the already mentioned measures, these plots include the error between the exact solution v_e and the sequential error v as a reference.

The plots in figure 7.3 and 7.4 all show a similar picture, and we observe that all measures decrease when the penalty parameter is increased. Still there are several parts of the plots worthy of note. The measure $\Delta\hat{J}$ related to the unpenalized objective function is the value that converges to zero the fastest. If we look at the values of $\Delta\hat{J}$ before the machine precision is reached we see that $\Delta\hat{J}$ is proportional to $\frac{1}{\mu^2}$. The convergence rate of $\Delta\hat{J}$ for $\Delta t = 10^{-2}$ and $N = 2$ is shown in table 7.9 together with the convergence rate of Δv. Δv and the other measures converge to zero at a rate of one, however we see that the relative error Δv between the controls v and v_μ stops to decrease long before the machine precision is reached. It seems that this barrier is hit around the same time as $\Delta\hat{J}$ approaches machine precision. The reason for this probably is that small changes in the control v_μ no longer registers in $\hat{J}_\mu$. It is therefore difficult to find an appropriate step length in the line search method, and this again means that we no longer are capable of finding the exact global minimizer of $\hat{J}_\mu$. If we remember theorem 5.1 from section 5.2 the sequence $\{v^k\}$ only converge toward v if for all k (v^k, Λ^k) is the global minimizer of $\hat{J}_\mu$. The results of figure 7.3 and 7.4 is therefore in line with the claim of theorem 5.1.

Unlike Δv, $\Delta\hat{J}_\mu$ and Δy continue to decrease steadily towards zero, even after $\Delta\hat{J}$ has hit machine precision. The $\Delta\hat{J}_\mu$ and Δy metrics are both related to the $\frac{\mu}{2}\sum_{i=1}^{N-1}(y^i(T_i) - \lambda_i)^2$ term, which is the part that enforces the continuity constraints (5.5). This result is again in line with the theory presented in section

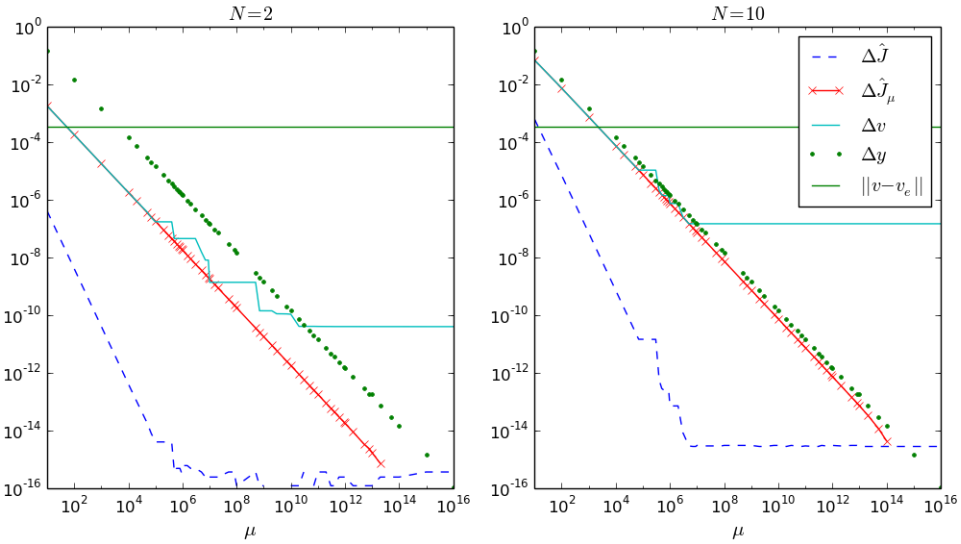


Figure 7.3: Logarithmic plot showing how the minimizer of $\hat{J}_\mu$ develops in comparison to the minimizer of $\hat{J}$ for increasing penalty parameter μ, when solving problem (7.1-7.2) with $\Delta t = 10^{-2}$. The measures plotted are defined in (7.7-7.10). The horizontal line $||v - v_e||$ is the accuracy of the unpenalized sequential solution. The left plot shows results of using $N = 2$ decompositions of the time interval, while on the right we have used $N = 10$ decompositions.

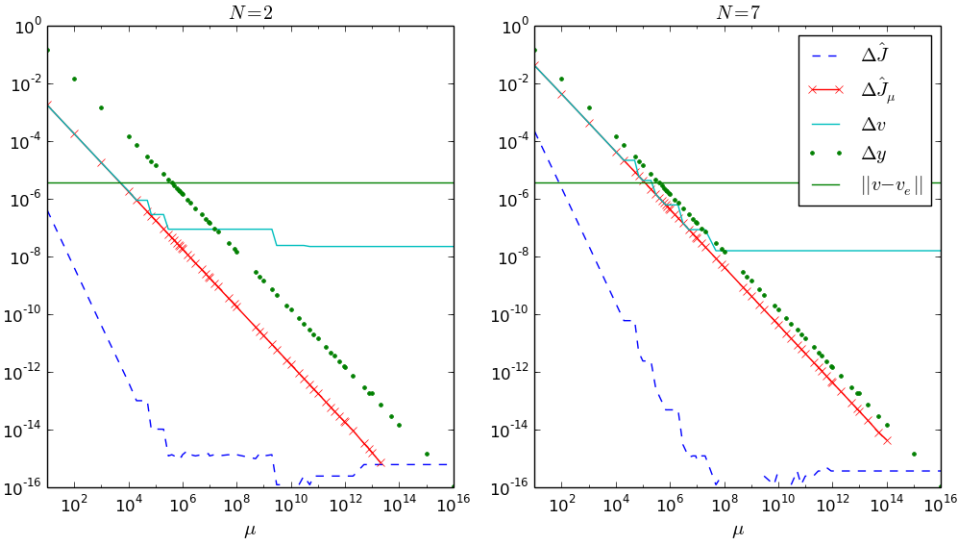


Figure 7.4: Same as figure 7.3, only now $\Delta t = 10^{-3}$.

5.2. There we stated theorem 5.2, which guaranteed that the iterates $\{(v^k, \Lambda^k)\}$ of the penalty method would converge to a feasible point when μ_k tended to infinity. In context of the reduced and decomposed problem (5.6-5.7), (v, Λ) being feasible means that the continuity constraints in (5.5): $y^i_{j_i} - \lambda_i = 0$ are satisfied for all $i = 1, ..., N-1$. In all the plots of figure 7.3 and 7.4 $\Delta y = \sup_i\{y^i_{j_i} - y^{i+1}_{j_i}\}$ approach machine precision for large μ's. The iterates $\{(v^k, \Lambda^k)\}$ obtained by our method applied to problem (7.1-7.2) therefore clearly converge towards a feasible point.

In the examples we have present here, the difference Δv between the numerical solutions v^μ and v obtained by our method and the serial algorithm goes below the difference $||v_e - v||$ between the exact solution v_e and v before it stops to converge. Our method is therefore as accurate as the sequential solution for the tested problems. If Δt is lowered, we expect that $||v_e - v||$ also will decline. When this happens it is not guaranteed that our method will be as accurate as the sequential algorithm. The conclusion of this section on the consistency of our method must be, that our method is consistent if we can minimize the penalized objective function for all penalty parameters μ. In theory this should be possible, but as we have seen the optimization of $\hat{J}_\mu$ gets difficult when μ becomes large. We theorize that this is caused by the fact ΔJ for large μ's hit machine precision, which then complicates the line search step in the optimization algorithm.

Table 7.9: Convergence rates for $\Delta t = 10^{-2}$ and $N = 2$. The second and third column show how difference in objective function value and norm between the numerical solution of the penalized problem v^μ and the solution of the unpenalized problem v develop for increasing μ values. The two last columns show at what rate these differences approach zero. We observe that $\Delta\hat{J}$ is proportional to μ^{-2}, while Δv is proportional to μ^{-1}. Notice how Δv stops to decrease at around the same time as $\Delta\hat{J}$ hits machine precision.

μ	$\Delta\hat{J}$	Δv	$\Delta\hat{J}$ rate	Δv rate
1e+01	4.10e-07	1.79e-03	–	–
1e+02	4.11e-09	1.79e-04	-1.99	-0.999
1e+03	4.12e-11	1.79e-05	-1.99	-1.000
1e+04	4.13e-13	1.79e-06	-1.99	-0.999
1e+05	4.04e-15	1.73e-07	-1.99	-0.985
4e+05	3.80e-15	1.71e-07	-0.11	-0.008
5e+05	3.67e-16	4.54e-08	-10.4	-5.96

Chapter 8

Experiments

In this chapter we will, through experiments, investigate what speedup one obtains by using our proposed algorithm 5.2 for parallelizing optimal control problems in temporal direction. Unlike the parallel performance of gradient and objective function evaluation, the parallel performance of our overall algorithm is difficult to model. The reason for this is that it is difficult to say a-priori how many gradient and function evaluations are needed for the optimization algorithms to terminate. We are therefore unable to derive any theoretical expected speedup.

In section 6.4 we explained that a good way of measuring performance of a parallel algorithm is to compare its execution time to the sequential execution time of the best sequential algorithm. We will assume that algorithm 3.1 is the best sequential algorithm for solving reducible optimal control problems. When solving optimal control problems with DE constraints, the runtime of our solution algorithm will depend on how many times we have to evaluate the objective function and its gradient, since these evaluations require either the solution of the state equation or the state and adjoint equations. We know from theory in section 6.4 and verification in section 7.3, that the speedup of parallel gradient and function evaluation depends linearly on the number of processes we use. An alternative way of measuring parallel performance is therefore to compare the sum of gradient and function evaluations in the sequential and parallel algorithms. Let us give these numbers a name:

$$L_s = \textit{Number of function and gradient evaluations for sequantial algorithm}$$
$$L_{p_N} = \textit{Number of function and gradient evaluations for parallel algorithm using } N \textit{ processes}$$

Using these definitions we define the ideal speedup $\hat{S}$, as the speedup one would expect based on L_s and L_{p_N} and the speedup results we have for function and

gradient evaluations:

$$\hat{S} = \frac{NL_s}{L_{p_N}} \tag{8.1}$$

With $\hat{S}$, it is possible to say something about the performance of the parallel algorithm without having to time it, or actually run it in parallel. It will also be useful to compare the ideal speedup with the measured speedup, as a way to check if the parallel implementation is implemented efficiently.

8.1 Testing the Parareal-Based Preconditioner on the Example Problem

In this section we will test the parallel framework introduced in chapter 5 on our example problem (3.8-3.9). To be able to do this, we need to define a specific objective function and state equation. The problem we will look at in this section is the following:

$$J(y, v) = \frac{1}{2}\int_0^T v(t)^2 dt + \frac{1}{2}(y(T) - 11.5)^2, \quad T = 100, \tag{8.2}$$

$$\begin{cases} y'(t) = -0.097y(t) + v(t) \quad t \in (0, T) \\ y(0) = 3.2 \end{cases} \tag{8.3}$$

We motivate the choice of a large end time $T = 100$ with the findings of section 7.4. There we observed that the penalty method ran into trouble when the time steps became too small, because the error in objective function value then hit machine precision. To be able to test the problem for a large number of time steps, we therefore need a large T.

We want to test how algorithm 5.2 performs when we vary the number of decompositions N, the penalty parameter μ and the number of fine time steps n. We also want to investigate how our preconditioned algorithm performs in comparison to a version of algorithm 5.2 where we do not use the Parareal-based preconditioner. Most of our tests are conducted using simulated parallelism, but we also include an experiment where we test out a parallel implementation of algorithm 5.2 on multiple cores.

In all our experiments we will use the L-BFGS algorithm [49] with memory length 10 to optimize the penalized objective function. We briefly explained how this algorithm works in section 3.4. Everywhere where the Parareal-based preconditioner

is used, it is constructed using a coarse propagator $\mathbf{G}_{\Delta T}$ based on the same numerical scheme as the fine solver. This means that when we use a Crank-Nicolson scheme to discretize the state and adjoint equations, $\mathbf{G}_{\Delta T}$ will also be based on Crank-Nicolson. In the cases where we have used an implicit Euler dicretization $\mathbf{G}_{\Delta T}$ is also based on implicit Euler.

8.1.1 Comparing Unpreconditioned and Preconditioned Penalty Method

In section 5.3 we introduced the Parareal preconditioner, as an approximation to the inverse Hessian. When using this preconditioner in our L-BFGS solver, we hope that the number of gradient and function evaluations needed in our algorithm will be smaller than if we do not use it. The experiment is conducted by first solving problem (8.2-8.3) without decomposing the time interval, and then solving the decomposed problem using $N = 2, 4, 8, 16, 32, 64, 128$ decompositions. For all minimizations of the penalized objective function, we used penalty parameter $\mu = 40000$. This means that we will only use one penalty iteration, as we have found this to be the most effective way to solve the decomposed example problem. To discretize the equations we have used the implicit Euler scheme with $\Delta t = \frac{T}{10^5} = 10^{-3}$. For both the penalized and non-penalized problems we use L-BFGS with stop criteria:

$$||\nabla J||_{L^2} < 10^{-5}$$

Since the point of this test is to compare the effectiveness of the Parareal-based preconditioner, we solve the decomposed problems with and without it. In table 8.6 we have included the total number of gradient and function evaluations for the two cases as "pc L" and "non-pc L". We also measured the relative L^2-norm difference between the exact solution v_e and all the penalized control solutions. The ideal speedup (8.1) is calculated for preconditioned and unpreconditioned solvers. The results of the sequential solver is presented in the first row named $N = 1$, and we let $L_S = L_{p_1}$.

There are several things of note about the results in table 8.1. First off we see that the normed difference in control between exact and parallel solution lies in the range from 10^{-5} to 10^{-3}. Another observation about the norm difference, is that for each N, the preconditioned and unpreconditioned solvers seems to produce roughly the same error.

When we look at the total number of gradient and functional evaluations for the preconditioned and unpreconditioned solvers, we see that there are differences.

Table 8.1: Comparing unpreconditioned and preconditioned solver for test problem (8.2-8.3) using N decompositions in time. Here v_e denotes the exact control solution, v_{pc} the preconditioned solver control solution and v the unpreconditioned solver solution. L_{p_N} represents total number of gradient and function evaluations used in each optimization. The ideal speedup $\hat{S}$ is based on this L_{p_N}. Notice that the preconditioned ideal speedup is significantly larger than the unpreconditioned ideal speedup for large N.

N	pc L_{p_N}	non-pc L_{p_N}	$\|\|v_e - v_{pc}\|\|$	$\|\|v_e - v\|\|$	pc $\hat{S}$	non-pc $\hat{S}$
1	13	13	0.000040	0.000040	1.00	1.00
2	15	15	0.000064	0.000064	1.73	1.73
4	29	29	0.000451	0.000472	1.79	1.79
8	53	53	0.000483	0.001725	1.96	1.96
16	109	175	0.001612	0.004105	1.90	1.18
32	97	361	0.001267	0.008545	4.28	1.15
64	43	469	0.001621	0.017026	19.34	1.77
128	43	799	0.002712	0.033097	38.69	2.08

While it seems to be little to no benefit to use the preconditioner for $N = 2, ..., 8$, it becomes very important for the bigger N values, where number of gradient and functional evaluations seems to explode for the unpreconditioned solver. If one accepts the above solutions as good enough, we see that we for the preconditioned solver get speedup for all decompositions, and that the ideal speedup seems to increase when we increase N. We do however see that the ideal speedup for each N is considerably less then optimal for all N. Another thing that we notice when looking at the sum of gradient and function evaluations for the preconditioned solver, is that it increases steadily up to $N = 16$, and then starts to decline again for higher Ns. The reason for this is that when we increase the number of decomposed subintervals, we also make the coarse solver in the Parareal preconditioner finer. This means that the preconditioner becomes a better approximation of the Hessian, which makes the L-BFGS iteration converge faster.

8.1.2 The Impact of μ on our Parareal-Based Preconditioner

In the previous subsection we observed that the performance of our Parareal preconditioned algorithm was independent of the number decompositions N. In this subsection we will investigate how our method performs, when we let N and n be constant, but vary the penalty parameter μ. The experiment will be conducted

by solving problem (8.2-8.3) using a Crank-Nicolson finite difference scheme. We use $n = 10^3$ and $\Delta t = 0.1$. The test will be executed sequentially, and potential speedup is measured using the ideal speedup $\hat{S}$ defined in (8.1). To be able to calculate $\hat{S}$, we first need to solve (8.2-8.3) using the serial algorithm. We present the accuracy and total number of function and gradient evaluations L_S of the serial solution in table 8.2.

Table 8.2: Number of gradient and function evaluations (L_s) and relative L^2 error ($||v_e - v||$) between exact solution v_e and solution of sequential algorithm v.

	$\|\|v_e - v\|\|$	L_S
serial result	3.88e-05	23

We then test our method on problem (8.2-8.3) for $N = 16$ and $N = 64$ decompositions of the time interval. The penalty parameter will vary between 10 and 10^{10}. We again want to see how the preconditioned solver performs in comparison with the unpreconditioned solver, and we therefore solve our example problem using both solvers. We compare the solutions v_{pc} and v we get using these solvers with exact solution v_e. We measure the performance of the solvers by counting the total number of function and gradient evaluations L_{p_N} done for each solve, and compare this number with L_S from table 8.2. Using L_{p_N} and L_S, we can calculate the ideal speedup $\hat{S}$. The results of the above detailed experiment is presented in table 8.3 and 8.4.

The results of table 8.3 and 8.4 again show that the preconditioned method performs better than the unpreconditioned method. This is observed in all three measures L_{p_N}, $\hat{S}$ and $||v_e - v||$ presented in the tables. For both $N = 64$ and $N = 16$, the total number of function and gradient evaluations L_{p_N} does not differ too much when we change the penalty parameter $\mu < 10^8$. This again leads to stable ideal speedup values. We also notice that the measured difference between the exact and numerical control solutions is proportional to $\frac{1}{\mu}$ until it reaches the error of the sequential solution. This is in line with what we observed in section 7.4 about the consistency of our method. When we do not use the preconditioner, we see that we for most of μ values fail to get a speedup larger than one, and we do not get an error of the same order as the error from the sequential algorithm. Though the speedup results of table 8.4 are better than the ones shown in table 8.3, the results of both tables demonstrate that our preconditioned method can achieve speedup even for penalty parameters $\mu > 10^8$.

Table 8.3: Results from solving the decomposed problem (8.2-8.3) with $N = 16$. Both a preconditioned and unpreconditioned version of our method were used. The penalized objective function is minimized for increasing penalty parameters μ. The columns show total function and gradient evaluations ($L_{p_{16}}$), ideal speedup $\hat{S}$ and relative L^2 error ($||v_e - v_{pc}||$). These three values are presented for both the preconditioned and unpreconditioned solver. The preconditioned parallel solver obtains an error roughly the same as the sequential solver when $\mu \geq 1000$. It also achieves a modest ideal speedup between 2 and 3, when $\mu < 10^8$. For $\mu \geq 10^8$ we no longer observe a speedup greater than one for our preconditioned algorithm.

μ	pc $L_{p_{16}}$	non-pc $L_{p_{16}}$	pc $\hat{S}$	non-pc $\hat{S}$	$\|\|v_e - v_{pc}\|\|$	$\|\|v_e - v\|\|$
1e+1	173	211	2.12	1.74	6.8e-3	6.8e-3
1e+2	99	357	3.71	1.03	6.5e-4	6.5e-4
1e+3	153	885	2.40	0.41	3.1e-5	1.1e-4
1e+4	123	433	2.99	0.84	3.2e-5	7.8e-5
1e+5	155	379	2.37	0.97	3.9e-5	9.2e-5
1e+08	410	3007	0.89	0.12	3.9e-5	4.3-2
1e+10	388	3007	0.94	0.12	3.9e-5	4.8-2

Table 8.4: Same as table 8.3, but now with $N = 64$. We observe better ideal speedup than in table 8.3 for the preconditioned solver, and the error is still roughly the same as the one measured for the sequential solver. The results achieved when solving the decomposed problem (8.2-8.3) without the Parareal-based preconditioner is worse when we use $N = 64$ instead of $N = 16$. We again observe significant drop in speedup when $\mu > 10^5$.

μ	pc $L_{p_{64}}$	non-pc $L_{p_{64}}$	pc $\hat{S}$	non-pc $\hat{S}$	$\|\|v_e - v_{pc}\|\|$	$\|\|v_e - v\|\|$
1e+1	101	1141	14.5	1.2	4.5e-2	4.5e-2
1e+2	141	1583	10.4	0.9	4.7e-3	4.7e-3
1e+3	169	1693	8.7	0.8	4.3-4	4.4e-4
1e+4	125	2437	11.7	0.6	1.0e-5	1.8e-4
1e+5	121	1807	12.1	0.8	3.4e-5	1.8e-4
1e+08	439	3007	3.3	0.4	3.9e-5	6.7e-4
1e+10	582	3007	2.5	0.4	3.9e-5	2.3e-1

When we increase μ above 10^5, we see for both $N = 16$ and $N = 64$, that L_{p_N} increases. For $N = 16$ we even observe speedup values less than one for the pre-

conditioned algorithm. We can however improve the performance of our method for large μ by doing multiple penalty steps. We illustrate this with the results presented in table 8.5. Here we first solved our example problem with $\mu = 10^5$ and $N = 16$, and then used this solution as an initial guess to the optimization of the penalized objective function with $\mu = 10^8$. We see that when we use this approach, solving the problem with $\mu = 10^8$ only requires 25 extra function and gradient evaluations. Adding these to $L_{p16} = 155$ from the $\mu = 10^5$ solve yields a total of 180 function and gradient evaluations, which for our example means an ideal speedup of $\hat{S} = 2.04$. Similar results can be obtained for $N = 16$.

Table 8.5: Result of first solving problem (8.2-8.3) using $\mu = 10^5$, for $N = 16$, and then using this solution as an initial guess for the solution of the same problem using $\mu = 10^8$. The two first rows present L_{p16}, $\hat{S}$ and normed control error ($||v_e - v_{pc}||$) between exact solution v_e and numerical solution v_{pc} for each penalty iteration. The last row show the total result of both solves. Notice that going from $\mu = 10^5$ to $\mu = 10^8$ only requires 25 extra function and gradient evaluations. In comparison solving problem (8.2-8.3) with $\mu = 10^8$, but without a good initial guess requires $L_{p16} = 410$.

μ	L_{p16}	$\hat{S}$	$\|\|v_e - v_{pc}\|\|$
1e+5	155	2.37	3.85e-5
1e+8	25	14.72	3.87e-5
Overall result	180	2.04	3.87e-5

8.1.3 Speedup Results for a High Number of Decompositions

To properly test algorithm 5.2, we have tested its use on the example problem on the Abel computer cluster. Using Abel, we are able to test our algorithm for a large number of CPUs. For all experiments the execution time of the sequential and parallel algorithms is measured by timing the solvers ten times, and choosing the lowest execution time. All our tests are done using an implicit Euler discretization. We run the test for three different problem sizes $n = 6 \cdot 10^5, 12 \cdot 10^5, 24 \cdot 10^5$ using an increasing number of processes N. Each process gets its own decomposed subinterval. The results for selected values of N and $n = 24 \cdot 10^5$ are found in table 8.6, while the remaining results are presented in figure 8.1.

The results of table 8.6 demonstrates that our parallel method can achieve actual

Table 8.6: Results gained from solving problem (8.2-8.3) for $n = 24 \cdot 10^5$ on N processes. The first two columns shows the error in control and objective function value. L_{p_N} is the total number of gradient and function evaluations, while $\hat{S}$ is the ideal speedup. The execution time for each N, and the corresponding speedup and efficiency are given in the last three columns.

N	$\frac{\|\|v-v_e\|\|_{L^2}}{\|\|v_e\|\|_{L^2}}$	$\frac{\hat{J}(v)-\hat{J}(v_e)}{\hat{J}(v_e)}$	L_{p_N}	$\hat{S}$	time (s)	speedup	efficiency
1	0.000002	–	19	1.00	63.37	1.00	1.000
4	0.000018	2.76e-10	37	2.05	40.12	1.57	0.394
16	0.000061	3.16e-09	97	3.13	28.40	2.23	0.139
32	0.000044	1.65e-09	85	7.15	12.68	4.99	0.156
48	0.000031	8.28e-10	73	12.49	7.05	8.98	0.187
72	0.000021	3.70e-10	88	15.54	6.24	10.14	0.140
96	0.000015	2.19e-10	61	29.90	3.63	17.45	0.181
120	0.000012	1.72e-10	61	37.37	2.69	23.55	0.196

speedup. The achieved speedup is however quite modest, since we for 120 cores only get a speedup of 23.5. We also notice that the speedup is smaller then the ideal speedup $\hat{S}$. This is as expected, since $\hat{S}$ assumes zero parallel overhead. There are three factors that cause the parallel overhead. The first is the overhead caused by communication and hardware. It is difficult to diminish these effects, but when the problem size increase the impact of the built in overhead should decrease. The second factor is our implementation. Our code is not optimized, and this has a bigger effect on the more complicated parallel algorithm than the sequential one. The effects of a suboptimal implementation does not necessarily diminish when we increase the problem size, but we might be able to remove these effects by improving our code. The last factor that impacts the parallel overhead, is our sequential Parareal-based preconditioner. The Parareal-based preconditioner is applied to the gradient through a backward and a forward solve on a coarse mesh of size N. This means that the effect of the preconditioner on the parallel overhead increases when we increase the numbers of processes. If $N << n$ these effects will however be very small. The results of table 8.6 are obtained by solving a problem of size $n = 24 \cdot 10^5$, while the largest N value is 120, which means that N is 20000 times smaller than n. It is therefore unlikely that it is the sequential Parareal-based preconditioner that is the main cause of the gap between ideal and measured speedup in table 8.6. Instead the disparity in ideal and measured speedup is probably caused by a combination of built in overhead and a suboptimal implementation.

Another interesting observation about table 8.6, is that the solution seems to improve when we increase N, which we see by looking at how the numerical control solution compares to the exact solution when we increase N. We compare exact and numerical solution by looking at normed difference in control and difference in function value. For $N \leq 16$ these measures increase, but for $N > 16$ they start to decrease again. We see the same type of pattern for L_{p_N}, which represents the total number of objective function and gradient evaluations done in each optimization. One interpretation of this, is that the Parareal-based preconditioner improves when the coarse decompositions become finer. The results of figure 8.1 paints a similar picture.

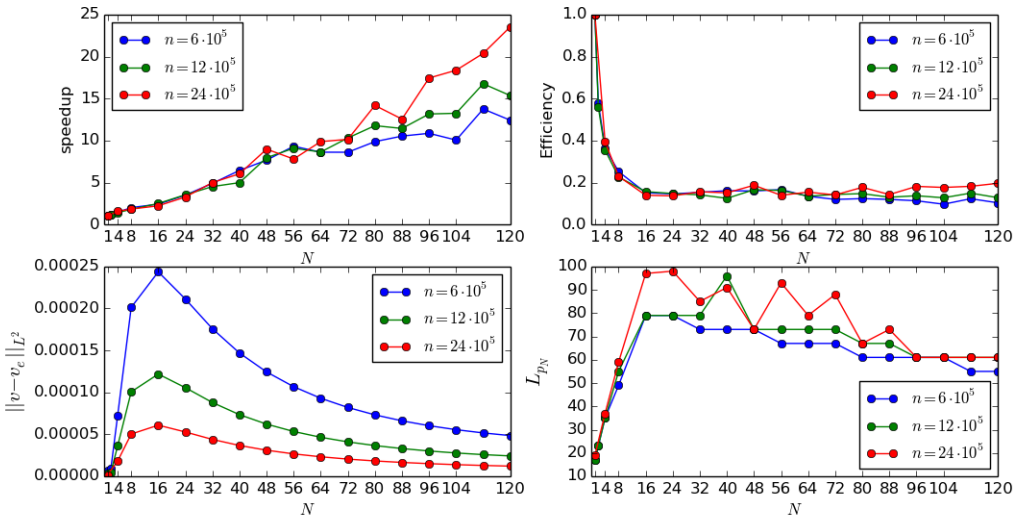


Figure 8.1: Speedup, efficiency, relative norm error and total number of objective function and gradient evaluations (L_{p_N}) for problem (8.2-8.3) using $n = 6\cdot10^5, 12\cdot10^5, 24\cdot10^5$ time steps on N cores.

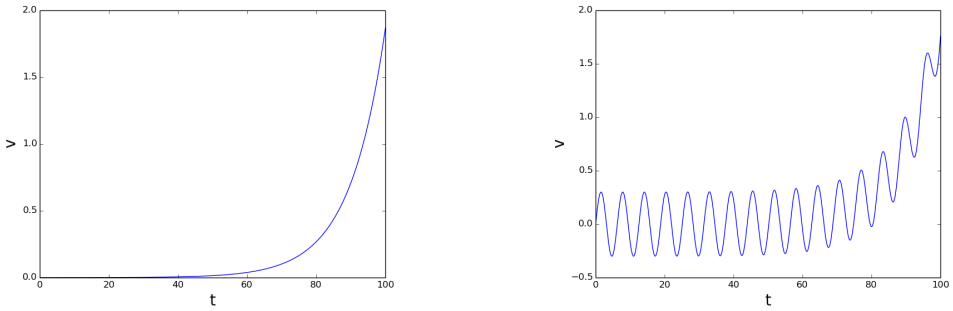
By looking at figure 8.1, we see that our algorithm preformed the best, at least in the sense of speedup, for $n = 24 \cdot 10^5$. We do however also observe the same type of behaviour for all values of n. We see that the error between exact and numerical control solution for all n first increases up till around $N = 16$, and then decreases and flattens out. The total number of gradient and function evaluations L_{p_N} becomes larger for higher N's when $N \leq 16$, but for $N > 16$ L_{p_N} starts to decrease slightly. We observe that for $N > 16$, the effectiveness $E = \frac{S}{N}$ of our method stabilizes around 0.17. The reason for this is that the total number of function and gradient evaluations decrease for $N > 16$.

8.1.4 Tests on a Less Smooth Problem

As we see by looking at figure 8.2a, the control solution of our example problem (8.2-8.3), is very smooth. It is therefore interesting to see if our algorithm can produce good results for problems with more uneven solutions. A simple way of slightly complicating our example problem is to add a sine function to the integral in the objective function. To produce the control solution pictured in 8.2b, we alter J in the following way:

$$J(y,v) = \frac{1}{2}\int_0^T (v(t) - 0.3\sin(t))^2 dt + \frac{1}{2}(y(T) - 11.5)^2, \quad T = 100. \tag{8.4}$$

We will now try to minimize the altered objective function (8.4) coupled with the same state equation constraints as before. Since we in section 8.1.3 showed that our algorithm is capable of producing actual speedup, we are now only looking at ideal speedup and not wall clock speedup. In table 8.7 we present results gained by applying our algorithmic framework to a Crank-Nicholson discretized minimization of the altered objective function (8.4) using $\Delta t = 10^{-2}$. For all decomposition sizes, the problem was solved using one penalty iteration. In an attempt to produce good results, we let the penalty parameter μ and tolerance τ vary between $10^4 - 10^5$ and $10^{-6} - 10^{-5}$.



(a) Minimizer of objective function (8.2) (b) Minimizer of objective function (8.4)

Figure 8.2: Optimal control for the unaltered and altered example problem (8.2-8.3). Notice the smoothness and simplicity of figure 8.2a.

It is interesting to contrast the findings of table 8.7 with the results from table 8.1. We notice that the total number of gradient and function evaluations (L_{p_N}) is consistently higher in table 8.7, but since this is also the case for the sequential solver, we actually observe better ideal speedup in table 8.7 than in table 8.1. This might indicate that our method has a higher potential for success on more

complicated problems, where the serial solver requires a higher number of function and gradient evaluations.

Table 8.7: Results of applying our algorithmic framework to optimization of (8.4) for different decomposition sizes N. The columns display error ($\frac{||v_e - v||}{||v||}$), total number of gradient and function evaluations (L_{p_N}) and ideal speedup ($\hat{S}$). The error is measured between the exact solution and the numerical solution for each N. For all $N > 1$, we observe the same order of accuracy as the serial algorithm. We also notice encouraging ideal speedup results, that are in part caused by an expensive serial solve, i.e. $L_S = L_{p_1} = 53$.

N	$\frac{\|v_e - v\|}{\|v\|}$	L_{p_N}	$\hat{S}$
1	0.000015	53	1.000
2	0.000021	75	1.413
8	0.000012	129	3.286
16	0.000010	159	5.333
32	0.000010	131	12.946
64	0.000010	157	21.605
128	0.000036	161	42.136

Chapter 9

Summary and Conclusions

The topic of this thesis is the parallelization of optimization problems with time-dependent differential equation constraints. The method we have proposed is developed around reducible problems, where the state equation constraint is well posed. Our method has been explained through a simple ODE constrained example problem, and we have also used this example for verification and testing. Even though we only considered an ODE example, we believe that our method also is applicable to time-dependent PDE constrained problems. Examples of such constraints can be PDEs on form (9.1).

$$\begin{cases} u_t(x,t) + Au(x,t) = f \quad \text{for} \quad (x,t) \in U \times (0,T), \\ u(x,0) = u_0(x). \end{cases} \tag{9.1}$$

Here A is a differential operator. The reason for making this claim, is that after the spatial discretization is taken care of, evolving equations of type (9.1) through time is done in the same way as for ODEs.

The main contribution of this thesis is the analysis of the Parareal-based preconditioner Q (5.36) proposed in [11] and the introduction of algorithm 5.2 based on the quadratic penalty method from section 5.2 and a preconditioned BFGS algorithm. The analysis of the Parareal-based preconditioner showed that Q is positive definite and an approximation of the inverse Hessian of the penalized objective function. This made us able to use the preconditioner from [11] in the BFGS algorithm as an initial inverse Hessian approximation. The Parareal-preconditioned BFGS algorithm is the central part of algorithm 5.2. We use the preconditioned BFGS method to minimize the penalized objective function $\hat{J}_\mu$ for increasing penalty parameters μ. The idea is that when μ gets sufficiently large, the minimizer of $\hat{J}_\mu$ will also approximate the minimizer of $\hat{J}$.

An important aspect of our method is the evaluation of the reduced and penalized objective function $\hat{J}_\mu(v, \Lambda)$ and its gradient. In chapter 6 we explained how to discretize $\hat{J}_\mu$ in context of the example problem, and we also explained how we can parallelize the evaluation of $\hat{J}_\mu$ and $\hat{J}'_\mu$. In chapter 7 we verified different features of our implementations of both the sequential and parallel algorithms for the example problem. In particular, we looked at the consistency of our method. We observed that the solution obtained by algorithm 5.2 approached the numerical solution of the sequential algorithm when we increased the penalty parameter μ. However, we also noticed, that when the difference between the function values of the sequential and parallel algorithm became close to machine precision, the control solution of the parallel algorithm stopped converging towards the sequential solution. This observation underlines a limitation of our algorithm. This limitation is that we can not always guarantee the consistency of our method, especially when the time steps used to dicretize the state equation become small. What we however can expect from the iterates $\{(v^k, \Lambda^k)\}$ obtained by algorithm 5.2 is that they will converge to a feasible point.

In chapter 8 we tested the performance of algorithm 5.2 on an example problem. We measured the performance in both accuracy and potential speedup. What we were particularity interested in was investigating whether the performance of the preconditioned algorithm is independent of the parameters N, n and μ, representing number of decompositions, number of fine time steps and penalty parameter. What we found was that increasing the first two of these parameters did not significantly worsen the performance of algorithm 5.2. In the case of number of processes and decomposed intervals N, we even observed improved results when N was increased. The most likely cause of this, is that when we increase N, the Parareal-based preconditioner Q becomes an improved approximation of the inverse Hessian of $\hat{J}_\mu$. For the penalty parameter μ, the picture became more nuanced. We observed that the computational cost of minimizing $\hat{J}_\mu$ increased sharply when μ became vary large. We did however also see that this effect could be diminished by doing multiple iterations of algorithm 5.2. In chapter 8 we also measured actual wall clock speedup for as many as 120 cores. This experiment was conducted on the Abel computer cluster, and we were able to achieve actual speedup. We for example obtained a speedup of 23.5 when using 120 cores.

9.1 Future Work

The algorithm proposed in this thesis experiences trouble, when the penalty parameter gets large. Several strategies for improving the method and the Parareal preconditioner could be taken. One example is to replace the quadratic penalty

method for removing the virtual constraints with the more advanced augmented Lagrangian method used in [12]. Instead of using the BFGS method for minimizing the penalized objective function, other optimization algorithms and techniques could be considered. In [34, 35] for example, the authors used an alternating direction decent method to minimize the penalized functional of an optimal control problem. Another potential improvement can perhaps be to find a preconditioner Q_v that approximates the inverted Hessian of $\hat{J}(v)$, and then alter the Parareal-based preconditioner in the following way:

$$Q = \begin{bmatrix} Q_v & 0 \\ 0 & Q_\Lambda \end{bmatrix}$$

Different approaches to parallelization of optimal control in temporal direction might also be considered. One could for example try to restrict the parallelization to the differential equations. By this we mean solving the optimal control problem in the traditional serial way, but when we need to solve the state and adjoint equations, we solve these using for example the Parareal algorithm. Using this strategy to parallelize optimal control problems, would simplify the optimization, but also make solving the state and adjoint equations more involved. Load bearing, which is simple for the method we have proposed would also become a more complicated issue if this alternative strategy is chosen.

In this thesis we have tested algorithm 5.2 for only one problem. It would be interesting to investigate how our proposed algorithm performs for other more complex problems. Since the problem we used was so easily solved, it was difficult to compute with the execution time of the serial algorithm. If a more challenging problem were solved, the potential for higher speedups might therefore be higher. Another issue that we have not considered, is how to choose the coarse propagator $\mathbf{G}_{\Delta T}$, that we use to construct the preconditioner. This becomes more important when we are solving PDE constrained problems, since we then also have a spatial discretization. One could then consider a Parareal-based preconditioner defined by a propagator $\mathbf{G}_{\Delta T}$ that uses a coarse dicretization in both space and time.

Bibliography

[1] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*. Oxford University Press (UK), 2014.

[2] J.-L. Lions, Y. Maday, and G. Turinici, "Résolution d'edp par un schéma en temps «pararéel»," *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, vol. 332, no. 7, pp. 661–668, 2001.

[3] J. Nievergelt, "Parallel methods for integrating ordinary differential equations," *Communications of the ACM*, vol. 7, no. 12, pp. 731–733, 1964.

[4] A. Bellen and M. Zennaro, "Parallel algorithms for initial-value problems for difference and differential equations," *Journal of Computational and applied mathematics*, vol. 25, no. 3, pp. 341–350, 1989.

[5] E. Lelarasmee, *The waveform relaxation method for time domain analysis of large scale integrated circuits: Theory and applications*. Electronics Research Laboratory, College of Engineering, University of California, 1982.

[6] M. J. Gander, "Overlapping schwarz for linear and nonlinear parabolic problems," 1996.

[7] W. Hackbusch, "Parabolic multi-grid methods," in *Proc. of the sixth int'l. symposium on Computing methods in applied sciences and engineering, VI*, pp. 189–197, North-Holland Publishing Co., 1985.

[8] C. Lubich and A. Ostermann, "Multi-grid dynamic iteration for parabolic equations," *BIT Numerical Mathematics*, vol. 27, no. 2, pp. 216–234, 1987.

[9] G. Horton and S. Vandewalle, "A space-time multigrid method for parabolic partial differential equations," *SIAM Journal on Scientific Computing*, vol. 16, no. 4, pp. 848–864, 1995.

[10] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich, *Optimization with PDE constraints*, vol. 23. Springer Science & Business Media, 2008.

[11] Y. Maday and G. Turinici, "A parareal in time procedure for the control of partial differential equations," *Comptes Rendus Mathematique*, vol. 335, no. 4, pp. 387–392, 2002.

[12] V. Rao and A. Sandu, "A time-parallel approach to strong-constraint four-dimensional variational data assimilation," *Journal of Computational Physics*, vol. 313, pp. 583–593, 2016.

[13] M. J. Gander and S. Vandewalle, "On the superlinear and linear convergence of the parareal algorithm," in *Domain decomposition methods in science and engineering XVI*, pp. 291–298, Springer, 2007.

[14] M. J. Gander, "50 years of time parallel time integration," in *Multiple Shooting and Time Domain Decomposition Methods*, pp. 69–113, Springer, 2015.

[15] W. L. Miranker and W. Liniger, "Parallel methods for the numerical integration of ordinary differential equations," *Mathematics of Computation*, vol. 21, no. 99, pp. 303–320, 1967.

[16] Y. Maday and E. M. Rønquist, "Parallelization in time through tensor-product space–time solvers," *Comptes Rendus Mathematique*, vol. 346, no. 1-2, pp. 113–118, 2008.

[17] S. Güttel, "A parallel overlapping time-domain decomposition method for odes," in *Domain decomposition methods in science and engineering XX*, pp. 459–466, Springer, 2013.

[18] L. Baffico, S. Bernard, Y. Maday, G. Turinici, and G. Zérah, "Parallel-in-time molecular-dynamics simulations," *Physical Review E*, vol. 66, no. 5, p. 057701, 2002.

[19] G. A. Staff and E. M. Rønquist, "Stability of the parareal algorithm," in *Domain decomposition methods in science and engineering*, pp. 449–456, Springer, 2005.

[20] Y. Maday, E. Rønquist, and G. A. Staff, "The parareal-in-time algorithm: Basics, stability analysis, and more," *Preprint*, pp. 1–20, 2007.

[21] G. Bal, "On the convergence and the stability of the parareal algorithm to solve partial differential equations," in *Domain decomposition methods in science and engineering*, pp. 425–432, Springer, 2005.

[22] X. Dai and Y. Maday, "Stable parareal in time method for first-and second-order hyperbolic systems," *SIAM Journal on Scientific Computing*, vol. 35, no. 1, pp. A52–A78, 2013.

[23] M. J. Gander and S. Vandewalle, "Analysis of the parareal time-parallel time-integration method," *SIAM Journal on Scientific Computing*, vol. 29, no. 2, pp. 556–578, 2007.

[24] P. F. Fischer, F. Hecht, and Y. Maday, "A parareal in time semi-implicit approximation of the navier-stokes equations," in *Domain decomposition methods in science and engineering*, pp. 433–440, Springer, 2005.

[25] G. Bal, "Parallelization in time of (stochastic) ordinary differential equations," *Math. Meth. Anal. Num.(submitted)*, 2003.

[26] I. Garrido, M. S. Espedal, and G. E. Fladmark, "A convergent algorithm for time parallelization applied to reservoir simulation," in *Domain Decomposition Methods in Science and Engineering*, pp. 469–476, Springer, 2005.

[27] C. Farhat and M. Chandesris, "Time-decomposed parallel time-integrators: theory and feasibility studies for uid, structure, and fluid-structure applications," *International Journal for Numerical Methods in Engineering*, vol. 58, no. 9, pp. 1397–1434, 2003.

[28] G. Bal and Y. Maday, "A "parareal" time discretization for non-linear pde's with application to the pricing of an american put," in *Recent developments in domain decomposition methods*, pp. 189–202, Springer, 2002.

[29] B. Lepsa and A. Sandu, "An efficient error control mechanism for the adaptive'parareal'time discretization algorithm," in *Proceedings of the 2010 Spring Simulation Multiconference*, p. 87, Society for Computer Simulation International, 2010.

[30] E. Aubanel, "Scheduling of tasks in the parareal algorithm," *Parallel Computing*, vol. 37, no. 3, pp. 172–182, 2011.

[31] J. Nocedal and S. J. Wright, "Numerical optimization 2nd," 2006.

[32] F. Chen, J. S. Hesthaven, Y. Maday, and A. S. Nielsen, "An adjoint approach for stabilizing the parareal method," tech. rep., 2015.

[33] V. Rao and A. Sandu, "An adjoint-based scalable algorithm for time-parallel integration," *Journal of Computational Science*, vol. 5, no. 2, pp. 76–84, 2014.

[34] Y. Maday and G. Turinici, "Parallel in time algorithms for quantum control: Parareal time discretization scheme," *International journal of quantum chemistry*, vol. 93, no. 3, pp. 223–228, 2003.

[35] Y. Maday, J. Salomon, and G. Turinici, "Monotonic parareal control for quantum systems," *SIAM Journal on Numerical Analysis*, vol. 45, no. 6, pp. 2468–2482, 2007.

[36] S. Ulbrich, "Preconditioners based on "parareal" time-domain decomposition for time-dependent pde-constrained optimization," in *Multiple Shooting and Time Domain Decomposition Methods*, pp. 203–232, Springer, 2015.

[37] T. P. Mathew, M. Sarkis, and C. E. Schaerer, "Analysis of block parareal preconditioners for parabolic optimal control problems," *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1180–1200, 2010.

[38] T. Carraro, M. Geiger, and R. Rannacher, "Indirect multiple shooting for nonlinear parabolic optimal control problems with control constraints," *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. A452–A481, 2014.

[39] J. W. Pearson, M. Stoll, and A. J. Wathen, "Regularization-robust preconditioners for time-dependent pde-constrained optimization problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 33, no. 4, pp. 1126–1152, 2012.

[40] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 43, pp. 50–67, Cambridge Univ Press, 1947.

[41] P. Wolfe, "Convergence conditions for ascent methods," *SIAM review*, vol. 11, no. 2, pp. 226–235, 1969.

[42] P. Wolfe, "Convergence conditions for ascent methods. ii: Some corrections," *SIAM review*, vol. 13, no. 2, pp. 185–188, 1971.

[43] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms 2. the new algorithm," *IMA Journal of Applied Mathematics*, vol. 6, no. 3, pp. 222–231, 1970.

[44] R. Fletcher, "A new approach to variable metric algorithms," *The computer journal*, vol. 13, no. 3, pp. 317–322, 1970.

[45] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of computation*, vol. 24, no. 109, pp. 23–26, 1970.

[46] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of computation*, vol. 24, no. 111, pp. 647–656, 1970.

[47] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1, pp. 503–528, 1989.

[48] J. C. Gilbert and C. Lemaréchal, "Some numerical experiments with variable-storage quasi-newton algorithms," *Mathematical programming*, vol. 45, no. 1, pp. 407–435, 1989.

[49] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

[50] A. Grama, *Introduction to parallel computing*. Pearson Education, 2003.

[51] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[52] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[53] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, pp. 51–56, van der Voort S, Millman J, 2010.

[54] L. Dalcin, "mpi4py," 2007.