Contents

Chapter 1

Introduction

In today's world high performance computing is an essential tool for scientists in many fields such as engineering, comutational physics and chemistry, bioinformatics, weather forecasting and so on. 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 most 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 pops up both in science and elsewhere, 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 spacial dimension within 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. Decomposing in space on the other hand allows us to partition the problem into independent tasks. It is usually more efficient to limit the parallel computing to the spacial domain, however in cases where the number of available cores are high, introducing parallelism in temporal direction can reduce the overall solution time. We therefore want 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 parallel in time methods is the so called Parareal method introduced in [30]. 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 [6, 40], while waveform relaxation methods [18, 28] and multigrid methods [24, 26, 32] achieves parallelism in time by parallelizing in both space and time simultaneously. In this thesis we will restrict our self to Parareal, and the other methods will not be touched upon any further.

The topic of this thesis is a subject closely related to time dependent differential equations(DE) namely optimal control with time dependent DE constraints. Optimal control with DE constraints are minimization problems on 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, while the variables y and v are respectively named the state and the control. The goal of the control problem (1.1) is to find a pairing $(\bar{y}, \bar{v})$ that minimize the objective function, but also satisfy the constraints set up by the sate equation $E(\bar{y}, \bar{v}) = 0$. Optimal control problems with PDE constraints have many applications. Some examples are: Variational data assimilation, sensitivity analysis, goal-based error estimation and more.

Numerically solving optimal control problems with DE constraints involves solving multiple differential equations. The computational cost of solving these equations will dominate the overall computational cost of any optimal control solver. Parallelization of problems of type (1.1) are therefore connected to the parallelization of differential equations. In this thesis we will investigate a Parareal-based parallel in time framework for optimal control with time dependent DE constraints. To achieve this, we will use the same strategy as in [36], 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 [36] the parallelization of time dependent optimal control problems is done by applying the Parareal preconditioner to the steepest descent method. We will instead test out an implementation where the same Paraeal preconditioner is used in combination with the BFGS method. The algorithm that we present in this thesis is in theory applicable to optimal control problems with DE constraints. We will however in this thesis restrict our self to example problems with ordinary differential equation(ODE) constraints.

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, and the structure of the work done can roughly divided into two parts:

1. Backround and presentation of algorithm
2. Verification and experiments

The bulk of the thesis is found in the first part, where we present and motivate a parallel framework for parallization of the control problem. 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. 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 Paraeal based preconditioner. 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. Also included in chapter 3 is one section on optimization algorithms and one on finite difference discretizations of ODEs.

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 it possesses the necessary properties for this to be possible.

The second part of the thesis deals with 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 speedup. The theoretical speedup for function and gradient evaluation is also verified in 7. Chapter 7 also contains a section showing how the solution of the discretized control problem converges to the exact solution, and a section that demonstrates the consistency of the penalty framework presented in chapter 5. Chapter 8 contains the results of experiments done using the method from 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 sequaential and parallel algorithm.

Chapter 2

Literature review

The Parareal algorithm was introduced by Lions, Maday and Turinici in [30] as a way to solve differential evolution equations $f(y(t), t) = 0$ in parallel. This is done by combining a coarse and fine 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 solution $Y(t)$ defined on the entire interval, and 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. The version of Parareal presented in [30] is most practical for use on linear equations. An alternative version of Parareal is found in [2]. The formulation given here is equivalent to the one in [30] for linear equations, but is easier applied to non-linear equations.

A lot of the work done on the Parareal algorithm, has been focused on establishing its stability and convergence properties. The stability results are found in [46], [33] and [4], where Staff in [46] derives sufficient conditions for the stability of Parareal for autonomous differential equations:

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

while [4] presents more general stability results for parabolic equations. The stability of Parareal applied to hyperbolic equations is a more difficult question as is explained in [10]. The convergence of Parareal is studied in [30], [4], [16] and [17]. In [30] 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 [4] 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. [16, 17] 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.

Different applications of the algorithm exists for example on the Navier-Stokes problem [13], for molecular-dynamics simulations [2], on stochastic ordinary differential equations [3], reservoir simulations [19], fluid, structure, fluid-structure [12], or on the American put [5]. The success of applying the Parareal algorithm varies between the different problems. In [5] a simulated speedup of 6.25 is achieved on 50 decompositions, which translates to an efficiency of 12.5%. In [12] speedup between 4 and 8.2 is 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 it here experienced difficulties with stability. For certain problem parameters, stability issues are encountered in [13], however for other parameters the algorithm is stable, and a speedup between 6 and 19.7 for 32 cores is estimated.

Since the Parareal algorithm is an iterative procedure, a stopping criteria for when to terminate the iteration is required. This is done in [29], where an error control mechanism for the Parareal algorithm to limit the number of Parareal iterations is introduced. The stopping criteria that they propose, is to stop 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 of many challenges associated with parallel computing is partitioning 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 2.1. This means that after k iterations, the the k-th process becomes idle. How to tackle this issue is described in [1], where the authors also present a practical implementation of the

Parareal algorithm.

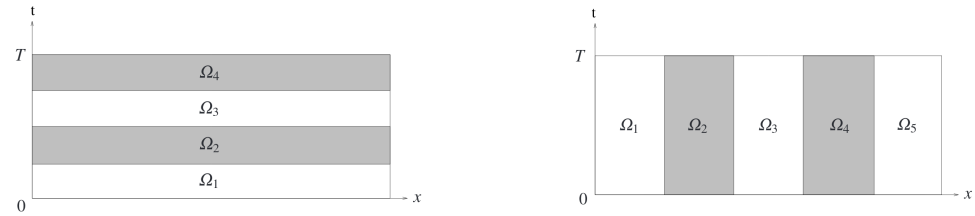
$$
\begin{array}{cccc}
 & & & \lambda_3^3 \\
 & & \nearrow & \uparrow \\
 & & \lambda_2^2 & \lambda_3^2 \\
 & \nearrow & \uparrow\nearrow & \uparrow \\
 & \lambda_1^1 & \lambda_2^1 & \lambda_3^1 \\
\nearrow & \uparrow\nearrow & \uparrow\nearrow & \uparrow \\
\lambda_0^0 & \lambda_1^0 & \lambda_2^0 & \lambda_3^0
\end{array}
$$

Figure 2.1: We see how the fine solution move from the initial condition at λ_0 to λ_3 through three iterations of the Parareal algorithm

As mentioned in [17] 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 [40] that eventually led to the so called multiple shooting methods. In [17] the authors also explain why the Parareal algorithm can be characterized as both a multiple shooting method and a multigrid method. A historic overview of the development of parallel in time algorithms can be found in [15]. Here we can also find presentations for the different strategies for parallelizing time dependent differential equations. One such strategy are the already mentioned multiple shooting methods [6, 40], 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 [18, 28], where the spatial domain is decomposed through time. The difference in these decomposition techniques is illustrated in figure 2.2. Other strategies presented are multigrid [24, 26, 32] methods and direct solvers in space-time [23, 34, 39].

The Parareal algorithm parallelizes the solution process of time dependent differential equations. In [36] Maday and Turinici extends Parareal, so that it can be used on optimal control problems with time dependent differential equation constraints. The problem looked at in [36], is:

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



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

Figure 2.2: Different decomposition techniques for parallel in time algorithms. Both figures lifted from [15].

We introduce parallelism in the same ways as we did 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 [42]. In [36] they use quadratic penalty terms, which leads to the following modified objective function:

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

We call the λ_is the virtual control, and they 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, 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 [44], 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). The experiments conducted in [44] yielded limited success, since 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 [36] 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 hope is that this Parareal based preconditioner will decrease the number of function and gradient evaluations needed for the optimization algorithm to converge, and the results in [36] 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 [8], where the preconditioner for the optimal control problem from [36] is used in a modified Parareal algorithm to stabilize it for hyperbolic equations. The adjoint based Parareal algorithm is proposed in [43]. 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 [43] 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 [35, 37] the authors derive a way to couple the Parareal algorithm with an optimization procedure for control of quantum systems. Like in [36] 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 [36]. 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.

As mentioned the Parareal preconditioner presented in [36] only affects the penalty terms. A more advanced preconditioner is derived in [47], where the Parareal preconditioner is incorporated into a preconditioner for a control problem with time-dependent partial differential equation constraints and inequality constraints.

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 are only looking at optimal control problems with time dependent differential equation(DE) constraints. This problem is only a part of the more general control problem, which can be formulated as:

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

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

Here $J : Y \times V \to \mathbb{R}$ is the objective function that we want to minimize, and the spaces Y, V are reflexive Banach spaces. The operator $E : Y \times V \to Z$, where Z is a Banach spaces, is called the state equation when we set it equal to zero. The state equation should have the property that $\forall v \in V$, $\exists! y(v) \in Y$ that satisfies:

$$E(y(v), v) = 0.$$

The first question we need to ask ourself, is under what conditions does the above problem even have a solution. This depends on the operators J, E and the spaces

Y, V, Z, and we will therefore come back to this question when we have defined our example problem. Instead let us assume for now that a solution exists, and try to derive a way to find it. If we replace y with $y(v)$ we can define the reduced problem:

$$\min_{v \in V} \hat{J}(v) = \min_{v \in V} J(y(v), v) \tag{3.3}$$

The problem (3.3) is called the reduced problem because we have moved the differential equation constraints into the functional, and by doing so transformed the constrained problem into an unconstrained one. Since the aim is to find the minimum of a function, it is natural, that to solve (3.3), we need to be able to differentiate the objective function $\hat{J}$ with respect to the control v. There are different ways of doing that, but we will focus on the so called adjoint approach, which is the most computational effective way of calculating the gradient of $\hat{J}$.

3.1.1 Example problem

To better understand the adjoint approach to gradient evaluation of the reduced objective function, we will 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 in chapter 7 and 8. In our example both the state y and the control v will be functions on an interval $[0, T]$. This allows us to define the objective function:

$$J(y, v) = \frac{1}{2} \int_0^T v(t)^2 dt + \frac{\alpha}{2} (y(T) - y^T)^2 \tag{3.4}$$

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

$$\begin{cases} y'(t) = ay(t) + v(t) \text{ for } & t \in (0, T), \\ y(0) = y_0. \end{cases} \tag{3.5}$$

Now that we actually have an optimal control problem, we can return to the question of solvability. To achieve this, we will write up a general result from [25] about linear quadratic problems, which also includes problem (3.4-3.5). Linear quadratic problems are a quite simple class of problems, and more general theorems for problems with non-linear equations and non-quadratic objective functions do however exist, and can be found in [25].

Theorem 1. *The linear quadratic optimization problems, that can be written on* *the form:*

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

Where H, V are Hilbert spaces, Y,Z are Banach spaces, $q \in H$, $g \in Z$ and $A : Y \to Z$, $B : V \to Z$ and $Q : Y \to H$ are bounded linear operators. If $\alpha > 0$, the above written problem has a unique solution pair $(y, v) \in Y \times V$.

Proof. See [25] □

It is obvious, that the objective function (3.4) of our example problem is a quadratic function that fits into the setting of the above theorem, and since the equation (3.5) is linear, with solution $y(t) = e^{at}(C(y_0) + \int_0^t e^{-a\tau} v(\tau) d\tau)$, which is unique for every integrable v, it is clear that problem (3.4-3.5) is a linear quadratic optimization problem.

3.2 The adjoint equation and the gradient

The usual way of finding the minimum(or maximum) value of a function J, is to solve the equation $J'(x) = 0$. Solving this equation usually requires us to be able to evaluate, or have an expression for the derivative of J. With this in mind let us try to find the gradient $\hat{J}'(v)$ of the reduced problem (3.3).

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

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 differentiate the state equation with respect to v, and assume that the operator $E_y(y(v), v) : Y \to Z$ is continuously invertible.

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

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

$$E_y(y(v), v)^* p = J_y(v). \tag{3.6}$$

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

$$\hat{J}'(v) = y'(v)^* J_y(y(v), v) + J_v(y(v), v) \tag{3.7}$$
$$= -E_v(y(v), v)^* E_y(y(v), v)^{-*} J_y(y(v), v) + J_v(y(v), v) \tag{3.8}$$
$$= -E_v(y(v), v)^* p + J_v(y(v), v). \tag{3.9}$$

Evaluating the gradient for a control variable $v \in V$ typically requires solving both the state and adjoint equation, and then inserting the solutions into the expression for the gradient (3.9). 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 Example adjoint

We want to derive the gradient of problem (3.4-3.5), and in order to do so, we need the adjoint equation of the problem, which we now state in the theorem below, followed by its derivation.

Theorem 2. *The adjoint equation of the problem (3.4-3.5) is:*

$$-p'(t) = ap(t) \tag{3.10}$$
$$p(T) = \alpha(y(T) - y^T) \tag{3.11}$$

Proof. Before we calculate the different terms used to derive the adjoint equation, we want to fit our ODE into an expression E. We do this by writing up the weak formulation of the equation:

$$y \in L^2(0,T) \text{ such that}$$
$$L[y,\phi] = \int_0^T -y(t)\phi'(t) - ay(t)\phi(t)dt - y_0\phi(0) + y(T)\phi(T) - \int_0^T v(t)\phi(t) = 0$$
$$\forall \ \phi \in C^\infty((0,T))$$

To derive the adjoint we need E_y and J_y. For E_y we define $(\cdot,\cdot)$ to be the L^2 inner product over $(0,T)$. Since the weak formulation includes evaluation at $t=0$ and $t=T$, we define an operator δ_τ that represent function evaluation in an L^2 inner product setting. We do this in the following way: Let $\tau \in [0,T]$ then:

$$(v, \delta_\tau w) = (\delta_\tau v, w) = \int_0^T v(t)\delta_\tau w(t)dt = v(\tau)w(\tau)$$

Using the above notation, we can write E_y quite compactly as:

$$E_y = L_y[\cdot,\phi] = (\cdot, (-\frac{\partial}{\partial t} - a + \delta_T)\phi)$$

Let us be more thorough with J_y, which is the right hand side of the adjoint equation.

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

We have $E_y = (\cdot, (-\frac{\partial}{\partial t} - a + \delta_T)\phi)$, but we want to find its adjoint E_y^*. Therefore let us explain what is ment by an adjoint on the context of the L^2 inner product $(\cdot,\cdot)$. Let $B : L^2(0,T) \to L^2(0,T)$ be a linear operator. Then the adjoint of B, B^* is an operator on $L^2(0,T)$, such that $\forall v, w \in L^2(0,T)$:

$$(Bv, w) = (v, B^* w).$$

The adjoint of the bilinear form $E_y = L_y$, would therefore be a bilinear form $L_y^* = E_y^*$ such that $\forall v, w \in L^2(0,T)$:

$$L_y[v,w] = L_y^*[w,v].$$

Therefore to derive the adjoint of E_y, we will insert two functions v and w into $L_y[v,w]$, and try to change the places of v and w.

$$\begin{aligned}
E_y = L_y[v,w] &= \int_0^T -v(t)(w'(t) + aw(t))dt + v(T)w(T) \\
&= \int_0^T w(t)(v'(t) - av(t))dt + v(T)w(T) - v(T)w(T) + v(0)w(0) \\
&= \int_0^T w(t)(v'(t) - av(t))dt + v(0)w(0) \\
&= L_y^*[w,v] = E_y^*
\end{aligned}$$

If we multiply J_y with a test function $\psi \in C^\infty((0,T))$ and set $L_y^*[p,\psi] = (J_y,\psi)$, we get the following equation: Find p such that:

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

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

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

Using this 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 $\hat{J}$. Lets state the result first.

Theorem 3. *The gradient of the reduced objective function $\hat{J}$ with respect to v is*

$$\hat{J}'(v) = v + p. \tag{3.12}$$

Proof. Firstly we need J_v and E_v^*:

$$\begin{aligned} J_v &= v \\ E_v &= L_v[\cdot, \phi] = -(\cdot, \phi). \end{aligned}$$

Since $L_v[\cdot, \phi]$ is symmetric, $E_v^* = E_v$, and strongly formulated, $E_v = -1$. By inserting relevant terms into (3.9), we get the gradient:

$$\begin{aligned} \hat{J}'(v) &= -E_v^* p + J_v && (3.13) \\ &= p + v. && (3.14) \end{aligned}$$

□

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

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

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:

$$(\hat{J}'(v), h)_V \approx \frac{\hat{J}(v + \epsilon h) - \hat{J}(v)}{\epsilon} \tag{3.15}$$

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 (3.15) 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 therfore 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.

3.2.2 Exact solution of the example problem

It turns out that we can find the exact solution of problem (3.4-3.4) by utilizing the adjoint equation (3.10-3.11) and the gradient of the reduced objective function (3.14). Finding an exact solution to our example problem will be useful for us in chapter 7, 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 trivial fact that $\hat{J}'(\bar{v}) = 0$ is a necessary condition for $\bar{v}$ being a minimizer of $\hat{J}$. Inserting expression (3.14) into $\hat{J}'(\bar{v}) = 0$ yields:

$$\bar{v}(t) = -p(t). \tag{3.16}$$

The second observation concerns the solution of the adjoint equation (3.10-3.11). Given a state $y(t)$, the solution of the adjoint equation is:

$$p(t) = \alpha(y(T) - y^T)e^{a(T-t)} = \omega e^{-at}. \tag{3.17}$$

Combining observation (3.16) with observation (3.17) suggests that a minimizer $\bar{v}$ of $\hat{J}$ should be on the form:

$$\bar{v}(t) = C_0 e^{-at}. \tag{3.18}$$

It turns out that plugging anstatz (3.18) 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 1 followed by its derivation.

Proposition 1. *Assume $a \neq 0$ and $\alpha > 0$. Then the solution of optimal control problem(3.4-3.4) is the following function:*

$$\bar{v}(t) = \alpha \frac{e^{aT}(y^T - e^{aT} y_0)}{1 + \frac{\alpha e^{aT}}{2a}(e^{aT} - e^{-aT})} e^{-at} \tag{3.19}$$

Proof. We start the proof by writing up the state equation (3.5) with (3.18) as source term:

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

This is a first order linear ODE with solution:

$$y(t) = y_0 e^{at} + \frac{C_0}{2a}(e^{at} - e^{-at}) \tag{3.20}$$

If we insert the state (3.20) into the formula for the adjoint (3.17), we can express the adjoint $p(t)$ in terms of the constant C_0:

$$\begin{aligned} p(t) &= \alpha(y(T) - y^T) e^{a(T-t)} && (3.21) \\ &= \alpha e^{aT}(y_0 e^{aT} + \frac{C_0}{2a}(e^{aT} - e^{-aT}) - y^T) e^{-at} && (3.22) \end{aligned}$$

The last step is to plug $v(t) = C_0 e^{-at}$ and $p(t)$ from (3.22) into observation (3.16) 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 $\forall T > 0$. $\square$

3.3 Finite difference

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 the standard 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:

$$\begin{cases} \frac{\partial}{\partial t}y(t) = F(y(t),t), & t \in I \\ y(0) = y_0 \end{cases} \tag{3.23}$$

Both the state and adjoint equation of our example problem can be formulated as an equation on form (3.23), and understanding the numerics of (3.23) 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 discritize 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:

$$\frac{\partial}{\partial t}y(t_k) \approx \frac{y_k - y_{k-1}}{\Delta t}. \tag{3.24}$$

By exploiting approximation (3.24) 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 (3.23) in definition 1 below.

Definition 1. *Explicit Euler applied to equation (3.23) means that for $k = 1,...,n$ the value of y_k is determined by the following formula:*

$$y_k = y_{k-1} + \Delta t F(y_{k-1}, t_{k-1}). \tag{3.25}$$

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

$$y_k = y_{k-1} + \Delta t F(y_k, t_k). \tag{3.26}$$

By looking at expression (3.25) and (3.26) 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 (3.25), which means y_k is defined explicitly, and hence the name explicit Euler. Another thing to notice about the finite difference schemes in definition 1, is that they solve the equation forwardly. This means that given y at time t_K, we can use (3.25) and (3.26) 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 (3.25) and (3.26) in definition 1. 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 2.

Definition 2. *An explicit Euler finite difference scheme for equation (3.23) with initial condition at $t = T$ instead of $t = 0$ yields the following formula for y_k:*

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

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

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

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 (3.23) for a given F, and that $\hat{y}$ is sufficiently smooth. If we then use method (3.25) or (3.26) with some Δt to solve (3.23) numerically, there exists a constant C such that the following error bound between $\hat{y}$ and numerical solution y holds:

$$\max_{k=0,\ldots,n} |y_k - \hat{y}(t_k)| \leq C\Delta t \tag{3.29}$$

A more accurate but still simple alternative to the Explicit and implicit Euler finite difference methods, is the so called Crank-Nicolson method [9]. We write up this method in a definition:

Definition 3. *The Crank-Nicolson finite difference scheme applied to equation (3.23) produces the following formula for y_k:*

$$y_k = y_{k-1} + \frac{\Delta t}{2}(F(y_k, t_k) + F(y_{k-1}, t_{k-1})). \tag{3.30}$$

In a setting where we are solving (3.23) backwards in time, the expression for y_k is changed to:

$$y_k = y_{k+1} - \frac{\Delta t}{2}(F(y_k, t_k) + F(y_{k-1}, t_{k-1})). \tag{3.31}$$

When comparing (3.30) with (3.25) and (3.26) 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 if we use Crank-Nicolson instead of the Euler methods. This means that the bound stated in (3.29) is improved to:

$$\max_{k=0,\ldots,n} |y_k - \hat{y}(t_k)| \leq C\Delta t^2 \tag{3.32}$$

Other more accurate finite difference schemes exist, 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 (3.4) includes an integral. The methods that we present in definition 4 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 4. *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:*

$$S_l = \Delta t \sum_{k=0}^{n-1} v_k \tag{3.33}$$

A slightly different approach to estimating S is the right-hand rectangle rule, defined by a formula similar to (3.33):

$$S_r = \Delta t \sum_{k=1}^{n} v_k \tag{3.34}$$

A third way of approximating S is the trapezoid rule:

$$S_{trap} = \Delta t \frac{v_0 + v_n}{2} + \Delta t \sum_{k=1}^{n-1} v_k \tag{3.35}$$

The rectangle methods in definition 4 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 analogue methods.

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 our 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:

$$\begin{aligned}
&1.\textit{Choose direction } p_k \in \mathbb{R}^n \\
&2.\textit{Choose step length } \alpha_k \in \mathbb{R} \\
&3.\textit{Set } x^{k+1} = x^k + \alpha_k p_k
\end{aligned}$$

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

$$||\nabla f(x^k)|| < \tau \tag{3.36}$$

What separates different line search methods, is how one chooses descent direction p_k and step length α_k. 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:

$$f(x^k + \alpha_k p_k) \leq f(x^k) + c_1 \alpha_k \nabla f(x^k) \cdot p_k$$
$$\nabla f(x^k + \alpha_k p_k) \cdot p_k \geq c_2 \nabla f(x^k) \cdot p_k$$

Here we use constants $0 < c_1 < c_2 < 1$. The first Wolfe condition ensures that the decrease in function value 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 (3.36). 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. Instead let us 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:

$$x^{k+1} = x^k - \alpha_k \nabla f(x^k) \tag{3.37}$$

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 5. *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 5 in mind let us state a theorem from [42] that specifies the convergence rate of the steepest descent method.

Theorem 4. *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. [42] □

The bound for $f(x^{k+1}) - f(x^*)$ given in theorem 4 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:

$$x^{k+1} = x^k - \nabla^2 f(x^k)^{-1} \nabla f(x^k) \tag{3.38}$$

As we will see in theorem 5, 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 [7, 14, 21, 45]. In BFGS the inverse Hessian approximation is calculated by the following recursive formula:

$$\begin{align} H^{k+1} &= (\mathbb{1} - \rho_k S_k \cdot Y_k) H^k (\mathbb{1} - \rho_k Y_k \cdot S_k) + S_k \cdot S_k, \tag{3.39}\\ S_k &= x^{k+1} - x^k, \tag{3.40}\\ Y_k &= \nabla f(x^{k+1}) - \nabla f(x^k), \tag{3.41}\\ \rho_k &= \frac{1}{Y_k \cdot S_k}, \tag{3.42}\\ H^0 &= \beta \mathbb{1}. \tag{3.43} \end{align}$$

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 [31] and [20]. Each line search iteration for BFGS looks like:

$$x^{k+1} = x^k - \alpha H^k \nabla f(x^k) \tag{3.44}$$

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 [41]. 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 [31] 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 [42] concerning this topic. We start with a result on the convergence rate of Newtons method.

Theorem 5. *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 3.38:*

1. *If* x^0 *is sufficiently 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. [42] □

The quadratic convergence of the Newton iteration is a big improvement in comparison with steepest descent, however theorem 5 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 6. *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. [42] □

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

Chapter 4

Parallel in time ODE solver methods

The process of resolving time dependent differential equation in temporal direction, is an exercise which one would intuitively think is unsuited for parallelization. This is due to the simple fact that the solution of such equations at every time T depends on the solution at times $t < T$, and it is therefore very difficult to partition the solution process into independent tasks that can be run in parallel. However the so called Parareal scheme introduced by Lions, Maday and Turinici in [30] makes it possible to parallelize the numerical solution of differential equations in time. We will however not introduce Parareal as it is described in [30], but rather present an alternative formulation of the algorithm given in [2]. 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:

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

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 λs 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 [2] 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:

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

$$\lambda_0^0 = u_0 \tag{4.4}$$

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 1: K steps of 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 1 we do K iterations of the Parareal algorithm, where K is a pre-chosen number. If one wanted to construct an actual Parareal algorithm, the iteration should instead terminate, when a certain stopping criteria is met. In general we want the iteration to stop when the Parareal solution is sufficiently close to the sequential solution, but we also want to avoid finding the sequential solution. A good stopping criteria for the Parareal algorithms can be found in [29].

Another important topic, that affects the performance of Parareal, that we have yet to mention, is how to choose a good coarse propagator $\mathbf{G}_{\Delta T}$. If $\mathbf{F}_{\Delta T}$ is based on a finite difference scheme method with time step Δt, one obvious 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. One simple and safe way of choosing $\mathbf{G}_{\Delta T}$, is to use implicit Euler 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:

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

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 [2] for example, they create the coarse propagator by simplifying the physics of the problem they 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 [36] an algebraic reformulation of (4.5) is presented. The setting in [36] 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:

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

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 [36] then write this system on matrix form as:

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

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 prconditioner.

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 [30] 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 2. *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 propagator solving 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 2 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 exact solution(or rather 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 [30], they used a first order implicit Euler scheme for their coarse propagator. It turns out that if one instead uses a 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 [4], 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 [16]. 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 2 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 [9], 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_k 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. 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.

Table 4.1: Results for initial coarse and fine solver applied to equation 4.14. We observe a convergence rate of 1, which is as expected for an implicit Euler scheme.

N	ΔT	err	rate
40	0.0250	0.802648	–
50	0.0200	0.628177	1.09837
100	0.0100	0.298689	1.07253
200	0.0050	0.145239	1.04021
500	0.0020	0.057075	1.01934
1000	0.0010	0.028365	1.00875
2000	0.0005	0.014139	1.00441

Table 4.2: Convergence results for one iteration of Parareal. We see a rate of convergence consistent with proposition 2.

N	ΔT	err	rate
40	0.0250	0.036593	–
50	0.0200	0.027970	1.20431
100	0.0100	0.008871	1.65668
200	0.0050	0.002426	1.87042
500	0.0020	0.000406	1.95102
1000	0.0010	0.000103	1.97996
2000	0.0005	0.000026	1.99027

Table 4.3: Convergence results for two iterations of Parareal. We see that the rate of convergence approaches 3

N	ΔT	err	rate
40	0.0250	3.759869e-03	–
50	0.0200	1.384116e-03	4.47838
100	0.0100	1.090653e-04	3.6657
200	0.0050	2.345744e-05	2.21707
500	0.0020	1.852693e-06	2.77046
1000	0.0010	2.448745e-07	2.91951
2000	0.0005	3.072620e-08	2.9945

Table 4.4: Convergence results for three iterations of Parareal. We see that the convergence rate does not behave as we would expect from proposition 2. This most likely explanation for this is that the error of the Parareal algorithm approaches the numerical error of the fine scheme.

N	ΔT	err	rate
40	0.0250	2.574574e-04	–
50	0.0200	7.818525e-05	5.34084
100	0.0100	1.487138e-06	5.71629
200	0.0050	1.345036e-07	3.46682
500	0.0020	6.044837e-09	3.38581
1000	0.0010	4.431655e-10	3.76979
2000	0.0005	5.573852e-11	2.9911

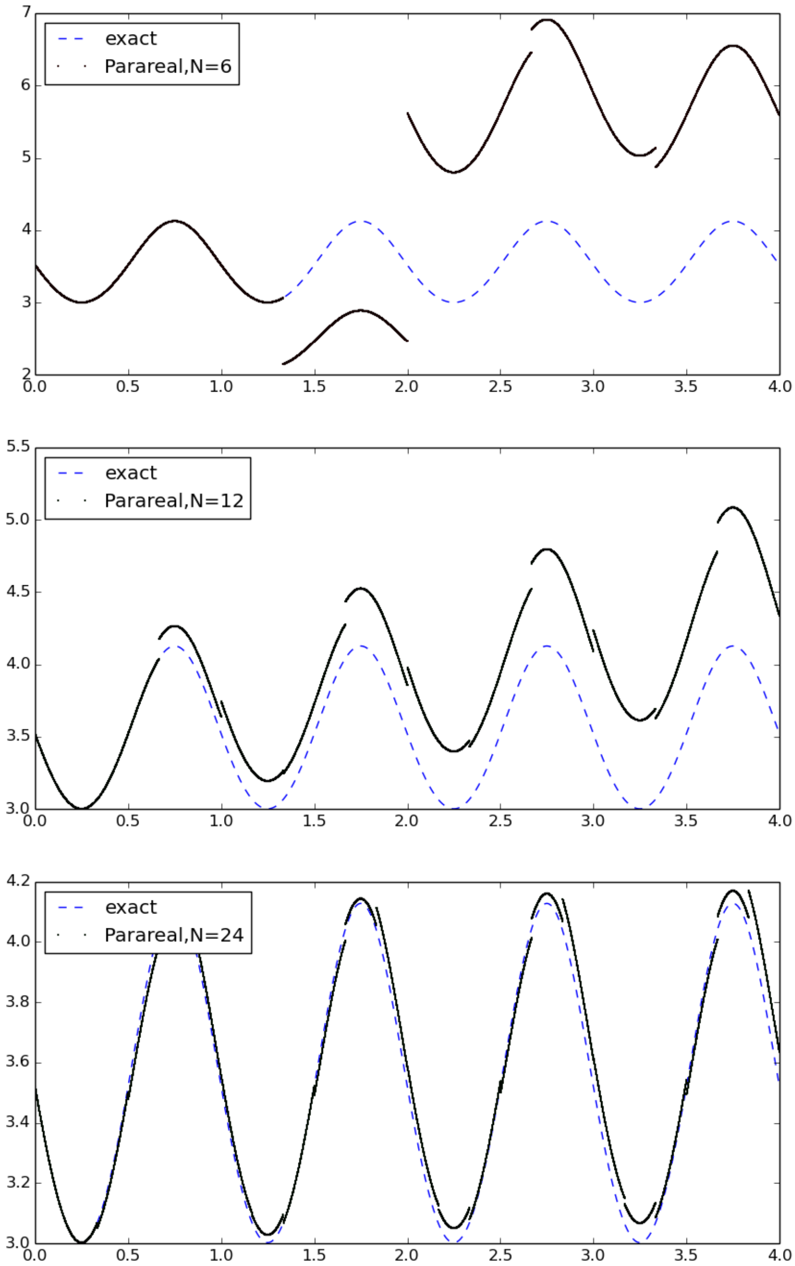


Figure 4.1: The result of 1 iteration of the Parareal algorithm on equation (4.14), for three different time decompositions, $N = 6, 12, 24$.

Chapter 5

Parareal 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 look at how to parallelize optimal control problems with time dependent differential equation constraints in temporal direction.

This chapter consists of three 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. We use the same approach as [36] namely the penalty method. This is a simplified version of the augmented Lagrangian approach used in [44] 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 last section a Parareal based preconditioner to be used in the optimization algorithms solving the optimal control problems is presented. This preconditioner originally proposed in [36] is derived using ides 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.

5.1 Optimal control problem with time-dependent DE constraints on a decomposed time interval

Before we start to decompose the time interval, let us again state the general problem that we want to solve:

$$\min_{y\in Y, v\in V} J(y(t),v) \tag{5.1}$$

$$\text{Subject to: } E(y(t),v)=0 \quad t\in[0,T] \tag{5.2}$$

To introduce parallelism, we decompose $I=[0,T]$ into N subintervals $I_i=[T_{i-1},T_i]$, with $T_0=0$ and $T_N=T$. To be able to solve the differential equation E on each interval I_i, we introduce intermediate initial conditions $y(T_i)=\lambda_i$ for $i=1,...,N-1$. This means that instead of finding y by solving E on the entire time domain I, we can now find y by solving E separately on on each subinterval I_i. The problem (5.2) now reads:

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

$$\text{Subject to: } E^i(y^i(t),v)=0 \quad t\in[T_{i-1},T_i] \quad \forall i \tag{5.4}$$

Since we want the state y to be continuous, we also need the following conditions:

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

Both the problems (5.1-5.2) and (5.3-5.5) are constrained problems, which we solve by reducing them to unconstrained problems. In the original setting this can easily be done if we assume that each control variable v corresponds to a unique solution y of the state equation E. We can then define a reduced objective function $\hat{J}(v)$, and minimize it with respect to v, i.e solve the unconstrained problem:

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

Assuming that the decomposed state equations also can be uniquely resolved $\forall v$, we can again define a reduced objective function $\hat{J}$. However because of the extra conditions (5.5) the reduction of (5.4) still produces a constrained problem:

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

$$y^{i-1}(T_i)=\lambda_i \quad \forall i \tag{5.7}$$

5.2 The penalty method

To solve the constrained problem (5.6-5.7), we will use the penalty method [42], which transforms constrained problems into a series of unconstrained problems by incorporating the constraints into the functional. Incorporating the constraints means penalizing not satisfying the constraints. To use the penalty method on (5.6-5.7) we first introduce the initial conditions to the decomposed state equations as variables $\Lambda = (\lambda_1, .., \lambda_{N-1})^T$, and then define the penalized objective function $\hat{J}_\mu$:

$$\hat{J}_\mu(v, \Lambda) = \hat{J}(v) + \frac{\mu}{2} \sum_{i=1}^{N-1} (y^{i-1}(T_i) - \lambda_i)^2 \tag{5.8}$$

With $\mu > 0$. Since the new control variables are created by us, and do not exist in the original control problem, we call Λ the virtual control. The control v of the original problem will form now on be referred to as the real control. If we now minimize $\hat{J}_\mu$ with respect to (v, Λ), while letting μ tend to infinity, we hope that the solution satisfies the conditions (5.5), while also minimizing the actual problem (5.6-5.7). The algorithmic framework of this reads:

Algorithm 2: Penalty framework

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. $\parallel \nabla \hat{J}_{\mu_{k-1}}(v^k, \Lambda^k) \parallel < \tau_{k-1}$;
    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

Assuming that we have a solution v minimizing $\hat{J}$, one would hope that the iterates $\{v^k\}$ from the penalty method converges to the solution of the non-penalized problem, that is:

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

From [42] we get a result that deals with this:

Theorem 7. *Assume v^k is the exact global minimizer of J_{μ_k}, then each limit point of the sequence $\{v^k\}$ is a solution of the problem (5.2).*

Proof. [42] □

The above result shows that the penalty algorithmic framework actually produces a solution to the original problem. There are however parts of the above framework, that still needs special attention, namely how to find (v^k, Λ^k) in each iteration, how to update μ_k and τ_k and how to choose an adequate stopping criteria. Finding the optimal control for each iteration is done by applying an optimization method for unconstrained problems, that is dependent on the gradient at (v^k, Λ^k). Let us therefore differentiate the penalized objective function.

5.2.1 The gradient of the penalized objective function

To find the gradient of 5.8 we start by differentiating $\hat{J}_\mu(v, \Lambda)$:

$$\hat{J}'_\mu(v,\Lambda) = DJ_\mu(y(v,\Lambda), v, \Lambda) \tag{5.9}$$

$$= y'(v,\Lambda)^* \frac{\partial}{\partial y} J_\mu(y(v,\Lambda), v, \Lambda) + (\frac{\partial}{\partial v} + \frac{\partial}{\partial \Lambda}) J_\mu(y(v,\Lambda), v, \Lambda) \tag{5.10}$$

To find an expression for $y'(v,\Lambda)^*$ we differentiate the state equation E:

$$\begin{aligned} DE(y(v,\Lambda), v, \Lambda) = 0 &\Rightarrow E_y(y(v,\Lambda), v, \Lambda) y'(v,\Lambda) = -E_v(y(v,\Lambda), v, \Lambda) - E_\Lambda(y(v,\Lambda), v, \Lambda) \\ &\Rightarrow y'(v) = -E_y(y(v,\Lambda), v, \Lambda)^{-1}((E_v(y(v,\Lambda), v, \Lambda) + E_\Lambda(y(v,\Lambda), v, \Lambda)) \\ &\Rightarrow y'(v,\Lambda)^* = -(E_v(y(v,\Lambda), v, \Lambda)^* + E_\Lambda(y(v,\Lambda), v, \Lambda)^*) E_y(y(v,\Lambda), v, \Lambda)^{-*} \end{aligned}$$

Inserting the above expression for $y'(v,\Lambda)^*$ into the gradient yields:

$$\hat{J}'_\mu(v,\Lambda) = -(E_v^* + E_\Lambda^*) E_y^{-*} \frac{\partial}{\partial y} J_\mu + (\frac{\partial}{\partial v} + \frac{\partial}{\partial \Lambda}) J_\mu \tag{5.11}$$

$$= -(E_v^* + E_\Lambda^*) p + (\frac{\partial}{\partial v} + \frac{\partial}{\partial \Lambda}) J_\mu \tag{5.12}$$

Where p is the solution of the adjoint equation:

$$E_y^* p = \frac{\partial}{\partial y} J_\mu$$

Notice that the state equation E 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 (3.4-3.5).

5.2.2 Deriving the adjoint for the example problem

Let us first recall the example optimal control problem with ODE constraints:

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

$$\begin{cases} \frac{\partial}{\partial t} y(t) = ay(t) + v(t) \quad & t \in (0,T) \\ y(0) = y_0 \end{cases}$$

We can now decompose the interval $[0,T]$ into N subintervals $\{[T_{i-1},T_i]\}_{i=1}^N$, and then define the above state equation on each interval, which forces us to penalize the objective function. The decomposed state equations will look like:

$$\begin{cases} \frac{\partial}{\partial t} y^i(t) = ay^i(t) + v(t) \quad & t \in (T_{i-1},T_i) \\ y^i(T_{i-1}) = \lambda_{i-1} \end{cases} \tag{5.13}$$

The reduced penalized objective function will be given as:

$$\hat{J}_\mu(v,\Lambda) = \frac{1}{2}\int_0^T v(t)^2 dt + \frac{\alpha}{2}(y(T) - y^T)^2 + \frac{\mu}{2}\sum_{i=1}^{N-1}(y^{i-1}(T_i) - \lambda_i)^2 \tag{5.14}$$

Theorem 8. *The adjoint equation of problem (3.4-3.5) on interval* $[T_{N-1},T_N]$ *is:*

$$\begin{cases} -\frac{\partial}{\partial t} p_N = ap_N \\ p_N(T_N) = \alpha(y_N(T_N) - y_T) \end{cases} \tag{5.15}$$

On $[T_{i-1},T_i]$ *the adjoint equation is:*

$$\begin{cases} -\frac{\partial}{\partial t} p_i = ap_i \\ p_i(T_i) = \mu(y_i(T_i) - \lambda_i) \end{cases} \tag{5.16}$$

Proof. Lets begin as we did for the non-penalty approach, by writing up the weak formulation of the state equations:

$$\text{Find } y_i \in L^2(T_{i-1},T_i) \text{ such that}$$

$$L^i[y_i,\phi] = \int_{T_{i-1}}^{T_i} -y_i(t)(\phi'(t) + a\phi(t)) + v(t)\phi(t)dt - \lambda_{i-1}\phi(T_{i-1}) + y_i(T_i)\phi(T_i) = 0$$

$$\forall\ \phi \in C^\infty((T_{i-1},T_i))$$

To find the adjoint equations we differentiate the E^is and the functional $\hat{J}_\mu$ with respect to y. To simplify notation, let $(\cdot,\cdot)_i$ be the L^2 inner product of the interval $[T_{i-1},T_i]$.

$$E_y^i = L_y^i[\cdot,\phi] = (\cdot, -(\frac{\partial}{\partial t} + a - \delta_{T_i})\phi)_i$$

Lets differentiate $\hat{J}_\mu$:

$$\frac{\partial}{\partial y}\hat{J}_\mu = \alpha\delta_{T_N}(y_n(T_N) - y_T) + \mu\sum_{i=1}^{N-1}\delta_{T_i}(y_i(T_i) - \lambda_i)$$

Since y really is a collection of functions, we can differentiate $\hat{J}_\mu$ with respect to y_i. This gives us:

$$\frac{\partial}{\partial y_N}\hat{J}_\mu = \alpha\delta_{T_N}(y_n(T_N) - y_T)$$
$$\frac{\partial}{\partial y_i}\hat{J}_\mu = \mu\delta_{T_i}(y_i(T_i) - \lambda_i)\ i \neq N$$

We will now find the adjoint equations, by finding the adjoint of the E_y^is. This is done as above, by inserting two functions v, w into $L_y^i[v,w]$, and then moving the derivative form w to v.

$$\begin{aligned}
E_y^i = L_y^i[v,w] &= \int_{T_{i-1}}^{T_i} -v(t)(w'(t) + aw(t))dt + v(T_i)w(T_i) \\
&= \int_{T_{i-1}}^{T_i} w(t)(v'(t) - av(t))dt + v(T_i)w(T_i) - v(T_i)w(T_i) + v(T_{i-1})w(T_{i-1}) \\
&= \int_{T_{i-1}}^{T_i} w(t)(v'(t) - av(t))dt + v(T_{i-1})w(T_{i-1}) \\
&= (L_y^i)^*[w,v]
\end{aligned}$$

this means that $(E_y^i)^* = (L_y^i)^*[\cdot,\psi]$. The weak form of the adjoint equations is then found, by setting setting $(L_y^i)^*[p,\psi] = (J_{y_i},\psi)_i$. This gives two cases:

$i = N$ case:

Find $p_N \in L^2(T_{N-1},T_N)$ such that $\forall\ \psi \in C^\infty((T_{N-1},T_N))$
$$\int_{T_N-1}^{T_N} p_N(t)\psi'(t) - ap_N(t)\psi(t)dt + p_N(T_{N-1})\psi(T_{N-1}) = \alpha(y(T_N) - y^T)\psi(T_N)$$

$i \neq N$ cases:

Find $p_i \in L^2(T_{i-1},T_i)$ such that $\forall\ \psi \in C^\infty((T_{i-1},T_i))$
$$\int_{T_i-1}^{T_i} p_i(t)\psi'(t) - ap_i(t)\psi(t)dt + p_i(T_{i-1})\psi(T_{i-1}) = \mu(y_i(T_i) - \lambda_i)\psi(T_i)$$

The strong formulation, is obtained by partial integration:

$i = N$ case:

$$\text{Find } p_N \in L^2(T_{N-1}, T_N) \text{ such that } \forall\ \psi \in C^\infty((T_{N-1}, T_N))$$
$$\int_{T_N-1}^{T_N} -p_N'(t)\psi(t) - ap_N(t)\psi(t)dt + p_N(T_N)\psi(T_N) = \alpha(y(T_N) - y^T)\psi(T_N)$$

$i \neq N$ cases:

$$\text{Find } p_i \in L^2(T_{i-1}, T_i) \text{ such that } \forall\ \psi \in C^\infty((T_{i-1}, T_i))$$
$$\int_{T_i-1}^{T_i} -p_i'(t)\psi(t) - ap_i(t)\psi(t)dt + p_i(T_i)\psi(T_i) = \mu(y_i(T_i) - \lambda_i)\psi(T_i)$$

This gives us the ODEs we wanted. □

With the adjont equations we can find the gradient.

Theorem 9. *The gradient of (5.14), $\hat{J}_\mu'$, with respect to the control (v, Λ) is:*

$$\hat{J}_\mu'(v, \Lambda) = (v + p, p_2(T_1) - p_1(T_1), ..., p_N(T_{N-1}) - p_N(T_{N-1})) \tag{5.17}$$

Proof. If we first find E_v^*, E_λ^*, J_v and J_λ we can derive the gradient by simply inserting these expression into (5.12). We begin with the E terms:

$$E_v = L_v[\cdot, \phi] = -(\cdot, \phi)$$
$$E_{\lambda_{i-1}}^i = L_{y_{i-1}}^i[\cdot, \phi] = -(\cdot, \delta_{T_{i-1}}\phi)_i$$

Notice that both of these forms are symmetric, and we therefore do not need to do more work to find their adjoints, they are however derived from the weak formulation, and it might therefore be easier to translate these forms to their strong counterpart:

$$E_v = -1$$
$$E_{\lambda_{i-1}}^i = -\delta_{T_{i-1}}$$

Then lets differentiate J_μ:

$$\frac{\partial}{\partial v}J_\mu = v$$
$$\frac{\partial}{\partial \lambda_i}J_\mu = -\mu(y_i(T_i) - \lambda_i)$$

We can insert the above derived expressions into formula (5.12) to find the gradient. To make it simple we separate the v and Λ terms. First we look at the gradient associated with the source term v:

$$\begin{aligned}\frac{\partial}{\partial v}\hat{J}_\mu(v,\Lambda) &= -E_v p + \frac{\partial}{\partial v}J_\mu \\ &= p + v\end{aligned}$$

We then find the components of the gradient related to λ_i.

$$\begin{aligned}\frac{\partial}{\partial \lambda_i}\hat{J}_\mu(v,\Lambda) &= -E_{\lambda_i} p + \frac{\partial}{\partial \lambda_i}J_\mu \\ &= p_{i+1}(T_i) - \mu(y_i(T_i) - \lambda_i) \\ &= p_{i+1}(T_i) - p_i(T_i)\end{aligned}$$

Combining $\frac{\partial}{\partial v}\hat{J}_\mu$ and $\frac{\partial}{\partial \lambda_i}$ for $i = 1, .., N-1$ gives us the gradient 5.17. $\square$

5.3 Parareal preconditioner

Parallelizing the solution process of optimal control problems with time dependent differential equation constraints comes down to solving a series of penalized control problems. Since we have derived the gradient of these penalized problems 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 (3.37), which would create the following iteration for each penalized control problem:

$$(v^{k+1}, \Lambda^{k+1}) = (v^k, \Lambda^k) - \rho_k \nabla \hat{J}_\mu(v^k, \Lambda^k) \tag{5.18}$$

Alternatively we could use a BFGS iteration (3.44), which would result in the following update:

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

Where H^k is the inverse Hessian approximation defined in (3.39). To improve convergence of the unconstrained optimization solvers, we include the Parareal-based preconditioner, proposed in [36], in our optimization algorithms. The preconditioner Q will be on the form:

$$Q = \begin{bmatrix} \mathbb{1} & 0 \\ 0 & Q_\Lambda \end{bmatrix} \in \mathbb{R}^{n+N\times n+N}, \quad Q_\Lambda \in \mathbb{R}^{N-1\times N-1} \tag{5.20}$$

We see that Q only affects the $N-1$ last components of the gradient, which is the part connected with the Λ part of the control. The original control v is therefore not directly affected by Q. For steepest descent, we apply Q, by modifying (5.18) in the following way:

$$(v^{k+1}, \Lambda^{k+1}) = (v^k, \Lambda^k) - \rho_k Q \nabla \hat{J}_\mu(v^k, \Lambda^k) \tag{5.21}$$

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 Λ part of the control. 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 term, 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 find a good candidate for Q_Λ.

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 6 (Fine and coarse propagator)**.** *Let $f(y(t), t) = 0$ be a time dependent differential equation without a source term. 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:*

$$\begin{cases} f(y(t), t) = 0 & \textit{For } t \in (0, \Delta T) \\ y(0) = \omega \end{cases} \tag{5.22}$$

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 triangonal 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 7 (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 defined as follows:*

$$\min_{\Lambda} \mathbf{J}(\Lambda, y) = \sum_{i=1}^{N-1} (y_{i-1}(T_i) - \lambda_i)^2 \tag{5.23}$$

$$\textit{Subject to} \;\; y_{i-1}(T_i) = \mathbf{F}_{\Delta T}(\lambda_{i-1}) \; i = 1, ..., N-1 \tag{5.24}$$

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 could be written up on matrix form as:

$$M \; \Lambda = H. \tag{5.25}$$

The H on 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 explained how the Parareal algorithm could be reformulated as a preconditioned fix point iteration solving equation (5.25), expressed as follows:

$$\Lambda^{k+1} = \Lambda^k + \bar{M}^{-1}(H - M\Lambda^k) \tag{5.26}$$

Where $\bar{M}$ is the coarse version of the matrix M stated in definition 6. 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 (5.25) 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 8 (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:*

$$\begin{cases} f'(y(t),t)^* p(t) = 0 & \textit{For } t \in (0, \Delta T) \\ p(\Delta T) = \omega \end{cases} \tag{5.27}$$

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 8 we can write up the following proposition concerning the gradient of the reduced objective function of the virtual problem.

Proposition 3. *The reduced objective function of the virtual problem (5.23-5.24) is:*

$$\hat{\mathbf{J}}(\Lambda) = \sum_{i=1}^{N-1} (\mathbf{F}_{\Delta T}(\lambda_{i-1}) - \lambda_i)^2. \tag{5.28}$$

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

$$M^* \ M \ \Lambda \ = \ M^* \ H. \tag{5.29}$$

A preconditioned fix point iteration for equation (5.29) inspired by the Parareal formulation (5.26) is therefore:

$$\Lambda^{k+1} = \Lambda^k + \bar{M}^{-1}\bar{M}^{-*}(M^*H - M^*M\Lambda^k). \tag{5.30}$$

Proof. Luckily for us we have already derived the gradient of $\hat{\mathbf{J}}$ in (5.17). There we stated the gradient for the penalized version of the example problem (3.4-3.5). 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 5.29. 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:

$$M^*(M\ \Lambda - H) = \begin{pmatrix} \mathbf{F}^*_{\Delta T}(p_2(T_2)) - p_1(T_1) \\ \mathbf{F}^*_{\Delta T}(p_3(T_3)) - p_2(T_2) \\ \cdots \\ -p_{N-1}(T_{N-1}) \end{pmatrix} \tag{5.31}$$

$$= \begin{pmatrix} p_2(T_1) - p_1(T_1) \\ p_3(T_2) - p_2(T_2) \\ \cdots \\ p_{N-1}(T_{N-2}) - p_{N-2}(T_{N-2}) \\ -p_{N-1}(T_{N-1}) \end{pmatrix}. \tag{5.32}$$

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 always 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 $M^*M\Lambda = M^*H$. $\square$

Proposition 3 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 [36]. Inserting Q_Λ into Q yields the following:

$$Q = \begin{bmatrix} \mathbb{1} & 0 \\ 0 & \bar{M}^{-1}\bar{M}^{-*} \end{bmatrix}. \tag{5.33}$$

In [36] Q is proposed as a preconditioner for a steepest descent method. We do however not know if Q is positive definite, or if it is in any shape or form related to the Hessian of the objective function. We will investigate these questions further by reformulating the reduced objective function (5.28) for the virtual problem to a least squares problem.

5.3.2 Virtual least squares problem

Looking at the equation $M^*M\Lambda = M^*H$ we recognize the normal equation, which is connected to linear least squares problems. We therefore suspect that the virtual problem can be reformulated as a least squares problem. It turns out that this is indeed the case. We write up the new formulation in definition 9.

Definition 9 (Virtual least squares problem)**.** *Given a propagator $\mathbf{F}_{\Delta T}$ as defined in definition 6 and an initial condition $\lambda_0 = y_0$ for the state equation, the least squares formulation of the virtual optimal control problem (5.23-5.24) reads as follows:*

$$\min_{\Lambda \in \mathbb{R}^{N-1}} \hat{\mathbf{J}}(\Lambda) = x(\Lambda)^T x(\Lambda), \tag{5.34}$$

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

$$x(\Lambda) = \begin{pmatrix} \lambda_1 - \mathbf{F}_{\Delta T}(\lambda_0) \\ \lambda_2 - \mathbf{F}_{\Delta T}(\lambda_1) \\ \cdots \\ \lambda_{N-1} - \mathbf{F}_{\Delta T}(\lambda_{N-1}) \end{pmatrix}. \tag{5.35}$$

We are now interested in finding the Hessian of $\hat{\mathbf{J}}(\Lambda)$, which we hope to relate to the Parareal-based preconditioner.

Proposition 4. *The Hessian of function (5.34) is*

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

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 $\hat{\mathbf{J}}$:

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

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

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

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:

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

Now summing up the terms $\nabla^2 x_i(\Lambda) x_i(\Lambda)$ would yield the diagonal matrix $D(\Lambda)$ described in proposition 4. □

The first term of $\nabla^2\hat{\mathbf{J}}(\Lambda) = 2M(\Lambda)^T M(\Lambda) + 2D(\Lambda)$ resembles M^*M from the previous section, while the second term $2D(\Lambda)$ 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 $\nabla^2\hat{\mathbf{J}}(\Lambda) = 2M(\Lambda)^T M(\Lambda)$. We will therefore split our discussion of the Hessian of $\hat{\mathbf{J}}$ into two cases. In the first we assume the sate 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 $\nabla^2\hat{\mathbf{J}}(\Lambda) = 2M(\Lambda)^T M(\Lambda)$. Differentiating the propagator $\mathbf{F}_{\Delta T}$ is the same as linearising its governing equation. When the governing equation is it self linear, linearising it does not change the equation. Therefore $\mathbf{F}'_{\Delta T}(\lambda_i)\lambda_i = \mathbf{F}_{\Delta T}(\lambda_i)$. 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 $\nabla^2\hat{\mathbf{J}}(\Lambda) = 2M^*M$ we see that the Parareal-based preconditioner proposed in [36] is in fact related to the inverse Hessian of the reduced penalized objective function. 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 however quite simple to do, as we will see in the proof of the following proposition.

Proposition 5. *If $\mathbf{G}_{\Delta T}$ and $\mathbf{G}^*_{\Delta T}$ are based on consistent numerical methods, $\bar{M}^* = \bar{M}^T$, and 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, $\mathbf{G}_{\Delta T}(\omega) = \mathbf{G}^*_{\Delta T}(\omega)$. When inserting this into the matrices $\bar{M}$ and $\bar{M}^*$ from definition 6 and 8, we clearly see that $\bar{M}^* = \bar{M}^T$. For M^*M to be positive definite, the following two conditions must hold:

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

The first conditions 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 our matrix $\bar{M}$ is invertible. This also means that M^*M is positive definite, since both requirements for positive definiteness are satisfied. □

Proposition 5 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 $M(\Lambda)^TM(\Lambda)$ 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.

A way to get around the $D(\Lambda)$ term in the Hessian for 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 for for solving non-linear least square problems. Details can be found in [42]. 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 $\bar{M}(\Lambda)$:

$$\bar{M}(\Lambda) = \begin{bmatrix} \mathbb{1} & 0 & \cdots & 0 \\ -\mathbf{G}'_{\Delta T}(\lambda_1) & \mathbb{1} & 0 & \cdots \\ 0 & -\mathbf{G}'_{\Delta T}(\lambda_2) & \mathbb{1} & \cdots \\ 0 & \cdots & -\mathbf{G}'_{\Delta T}(\lambda_{N-1}) & \mathbb{1} \end{bmatrix} \tag{5.36}$$

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 example problem

To illustrate what Q actually will look like we write up $\bar{M}^*\bar{M}$ for our example problem (3.4-3.5). The state and adjoint equation of this problem is:

$$y'(t) = ay(t) + v(t), \tag{5.37}$$
$$p'(t) = -ap(t). \tag{5.38}$$

The state equation includes a source term, which will not be included in the governing equation of the propagators, since the propagators are based on the virtual sourceless problem. This means that the governing equation of $\mathbf{G}_{\Delta T}$ is $y'(t) = ay(t)$. Alternatively we could let (5.37) govern $\mathbf{G}_{\Delta T}$, but instead use $\bar{M}(\Lambda)$ from (5.36) in our preconditioner, which would produce the same result.

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 state and adjoint. 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 $\mathbf{G}^*_{\Delta T}(\omega)$:

$$\begin{aligned}
\frac{\mathbf{G}_{\Delta T}(\omega) - \omega}{\Delta T} &= a\mathbf{G}_{\Delta T}(\omega) \\
&\Rightarrow \mathbf{G}_{\Delta T}(\omega) = \frac{\omega}{1 - a\Delta T} \\
\frac{\omega - \mathbf{G}^*_{\Delta T}(\omega)}{\Delta T} &= -a\Delta T \mathbf{G}^*_{\Delta T}(\omega) \\
&\Rightarrow \mathbf{G}^*_{\Delta T}(\omega) = \frac{\omega}{1 - a\Delta T}
\end{aligned}$$

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 this is done for our example problem, when $N = 4$. 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 $\bar{\bar{J}}_{\lambda_3}$.

We end the section on the Parareal-based preconditioner with an important note about what happens 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 have no effect. Since Q 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.

Chapter 6

Discretization and MPI communication

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 prconditioner 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

Let us remember our example state equation (3.5) and objective function (3.4)

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

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

We know that 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 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(3.25):

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

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

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 (3.25), (3.26) and (3.30), while p_{k-1} were found using (3.27), (3.28) and (3.31). 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 course 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 10 below.

Theorem 10. *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 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 look like:

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

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. Lets 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 sate and adjoint equation, we are able to derive $\frac{\partial g(v)}{\partial v_i}$:

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

For $i = 1, ..., n-1$, the last expression of the above calculation is equal to the i-th component of $\frac{1}{2}(\frac{1}{1+\frac{\Delta t a}{2}}B + \frac{1}{1-\frac{\Delta t a}{2}}B^T)p_{\Delta t}$, which is what we wanted to show. By doing similar calculations we see that the Crank-Nicholson gradient stated in theorem 10 is also correct for $i = 0$ and $i = n$. □

6.2 Discretizing the decomposed time-domain

Decomposing the time interval $I = [0, T]$ into N equally sized subintervals $I_i = [T_i, T_{i+1}]$, and solving the state and adjoint equations separately on each subinterval, is what allows our algorithm to be run in parallel. Decomposing I is simple in the continuous case, however we are solving these equations numerically, and in the discrete case, partitioning I is more involved. To explain how we decompose I in the discrete case, lets look at how to do it for a general differential equation F:

$$\begin{cases} F(y(t), v(t)) = 0 & \text{For } t \in [0, T] \\ y(0) = y_0 \end{cases}$$

We then decompose I, and assume that we have $N-1$ intermediate initial conditions $\{\lambda_i\}_{i=1}^{N-1}$, such that we get a solvable equation on each subinterval:

$$\begin{cases} F^i(y^i(t), v(t)) = 0 & \text{For } t \in [T_i, T_{i+1}] \\ y^i(T_i) = \lambda_i \end{cases}$$

Now lets look at what happens when we discretize I. Lets divide I 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 some finite difference scheme, we can transform the differential equation F into a difference equation $F_{\Delta t}$:

$$\begin{cases} F_{\Delta t}(y(t_k), v(t_k)) = 0 & \text{For } k = 1, ..., n \\ y(t_0) = y_0 \end{cases}$$

The next step is to decompose the discrete interval $I_{\Delta t}$ into N discrete subintervals. This is simply done by extracting a subsequence $\{t_{k_i}\}_{i=0}^{N} \subset I_{\Delta t}$ where $t_{k_0} = t_0$ and $t_{k_N} = t_n$. This results in N sequences on the form $I_{\Delta t}^i = \{t_{k_i}, t_{k_i+1}, ..., t_{k_{i+1}}\}$, and if we assume, as we did in the continuous case, that we have some intermediate initial conditions $\{\lambda_i\}_{i=1}^{N-1}$, we can solve $F_{\Delta t}$ separately on each $I_{\Delta t}^i$:

$$\begin{cases} F_{\Delta t}^i(y^i(t_k), v(t_k)) = 0 & \text{For } k \in \{k_i, k_i+1, ..., k_{i+1}\} \\ y^i(t_{k_i}) = \lambda_i \end{cases}$$

There is one minor issue with decomposing $I_{\Delta t}$, which we did not mention above, and that has to do with the choice of the subsequence $\{t_{k_i}\}_{i=0}^{N}$. In theory, one could of course freely chose any subsequence of $I_{\Delta t}$, but we generally want the difference $t_{k_i} - t_{k_{i+1}}$ to be constant for all i. This is however not always possible, since there is no guarantee that n is divisible by N.

6.2.1 Partitioning

The general rule for partitioning a task between N processes, is to distribute the work of the task as evenly as possible. The task in the above setting is solving $F_{\Delta t}$, and the work to be distributed, is the computations required to move the solution from one time step to the next for all time steps. Since there are n time steps, we should be able to say that the main task of solving $F_{\Delta t}$ can be divided into n subtasks, and it is these n tasks that we want to distribute between the N processes. Now deciding how many subtasks each process should get is quite simple. Start with defining the numbers:

$$\begin{aligned} q &= \lfloor \frac{n}{N} \rfloor \\ r &= N \mod n \end{aligned}$$

Then we give each process q tasks, and then add one task to r processes. To which processes you give the extra task does not really matter, but the most straightforward way of doing it is just to give the first r processes the extra task. We however chose to look at the distributing problem slightly different. Instead of trying to distribute the n tasks, we try to evenly divide the $n+1$ points $\{t_k\}_{k=0}^n$ among the N processes. We redefine q and r as:

$$\begin{aligned} q &= \lfloor \frac{n+1}{N} \rfloor \\ r &= N \mod n+1 \end{aligned}$$

Every process now gets q points, but due to overlap every process gets an extra point excluding the first process. Then the remaining r points are given to the first r processes. This allows us to define the sequence $\{k_i\}_{i=0}^N$ recursively as follows:

$$k_{i+1} = k_i + q + \delta_{i\neq 0} + \delta_{i<r}$$

Here the δs are conditional functions defined as:

$$\delta_S = \begin{cases} 1 \ S = \text{True} \\ 0 \ S = \text{False} \end{cases}$$

We now have a way of decomposing the discrete time interval, and therefore also a way of partitioning the finite difference solver of the state equation in temporal direction. However both when we want try to find the gradient in a penalized optimal control problem, or when we just want to parallelize solving a differential equation using the parareal scheme, some communication between the processes is required.

6.2.2 Numerical gradient of the penalized example problem

Let us restate the decomposed example ODE, and the penalized objective function defined in (5.13-5.14)

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

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

Let us also remember the gradient of (6.15) stated in (5.17)

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

We now want to discretize the objective function (6.15) and find its discrete gradient for the different finite difference schemes and integration rules, as we did for the non-penalized problem. Discretizing the decomposed ODEs is straight forward, however the solution of the state and adjoint equations now consists of independent solution $y_{\Delta t}^i$ and $p_{\Delta t}^i$ on each subinterval $I_{\Delta t}^i$, where

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

One problem with $y_{\Delta t}^i$ and $p_{\Delta t}^i$ existing independently on each interval, is that we get an overlap on all the subinterval boundaries, which have the potential of complicating the penalized numerical objective function and its gradient. It turns out that for our example problem this problem only arises in the gradient. 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 discitized objective function (6.18) in theorem 11 expressed in terms of the discrete adjoint $p_{\Delta t}$.

Theorem 11. *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:*

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

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:

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

where $M_\theta \in \mathbb{R}^{(n+1)\times(n+1)}$ is the matrix defined in theorem 10, 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 $p^i_{\Delta t}$.

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

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

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

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 it self and $y^{i+1}_{k_{i+1}}$. With this in mind lets 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

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

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 $y^i_{\Delta t}$, which means that the only term in g, that depend on v_k, is

$\frac{\mu}{2}(y^i_{k_i} - \lambda_i)^2$ 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:

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

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

6.3 Communication without shared memory

If we assume that we are solving our optimal control problem in parallel on processes that do not share any memory, there will have to be communication between the processes. The parts of the algorithm that we are parallelizing, is the evaluation of the objective function and its gradient for a given control variable v. 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. Since the required communication between the processes are different for function and gradient evaluation, we describe them separately. However they both share the same starting point, which is explained 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 control variables $\{\lambda_i\}_{i=1}^{N-1}$ that originates from the penalty terms in the functional. 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 P_i have solved their part of the state equation, they all have the following data stored locally:

Control variable: v_i

Penalty control variable: λ_i

Solution to local state equation: $y^{i+1} = \{y^i_j\}_{j=k_{i-1}}^{k_i}$

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

6.3.1 Communication in functional 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. Since 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 $\hat{J}^i(y^{i+1}(v_i),v_i)$, such that:

$$\hat{J}(y(v),v) = \sum_{j=0}^{N-1}\hat{J}^j(y^j(v_j),v_j)$$

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 3: Parallel objective function evaluation

Data: Partitioned control variable (v_i,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,λ_i)`// In parallel`;

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

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

  end

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

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

    $\hat{J}^{N-1}_\mu(y^N(v_{N-1}),v_{N-1}) \leftarrow \hat{J}^{N-1}(y^N(v_{N-1}),v_{N-1})$;

  else

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

  end

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

end

6.3.2 Communication in gradient computation

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. For processes P_i, $i < N - 1$, the initial condition of the adjoint equation is $p^i(T_i) = \mu(y^i(T_i - \lambda_i)$. 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 $\nabla \hat{J}^i = (J_v(y^{i+1}(v^i), v^i) - B_i^* p^{i+1})$. 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 4: Parallel gradient evaluation

Data: Partitioned control variable (v_i, 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, 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**

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

  end

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

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

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

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

  end

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

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 lets 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 [22] 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

The function evaluation consists of solving the state equation, which requires $\mathcal{O}(n)$ operations, and then applying the functional on the control and the state on all $n+1$ time points, which probably also requires $\mathcal{O}(n)$ operations, we can assume that the sequential objective function evaluation execution time is

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

For 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 float. 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 speedup

When we calculate the objective function gradient sequentially, we solve both the state and adjoint equations. Still the required operations are 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 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 discritization detailed in chapter 6. All implementations are done in the python programming language, and the numerics is done using the NumPy [48] library. Plots are created using the matplotlib [27] package, tables are auto generated using Pandas [38] and the parallel parts are implemented using the mpi4py [11] library. We test our algorithm using the example problem (3.4-3.5), 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 6.2.2, 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 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 ϵs, and list the results in table 7.1.

Table 7.1: Taylor test for non-penalized discrete objective function

ϵ	D_1	D_2	$\|\|\epsilon w\|\|_{l_\infty}$	$\log(\frac{D_1(10\epsilon)}{D_1(\epsilon)})$	$\log(\frac{D_2(10\epsilon)}{D_2(\epsilon)})$
1.000000e+00	5956.494584	5.244487e+03	99.987417	–	–
1.000000e-01	123.645671	5.244487e+01	9.998742	1.68281	2
1.000000e-02	7.644529	5.244487e-01	0.999874	1.20883	2
1.000000e-03	0.717253	5.244487e-03	0.099987	1.02768	2
1.000000e-04	0.071253	5.244487e-05	0.009999	1.00287	2
1.000000e-05	0.007121	5.244489e-07	0.001000	1.00029	2
1.000000e-06	0.000712	5.244760e-09	0.000100	1.00003	1.99998
1.000000e-07	0.000071	5.255194e-11	0.000010	1	1.99914

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. Let us then check if this is also the case for the penalized problem.

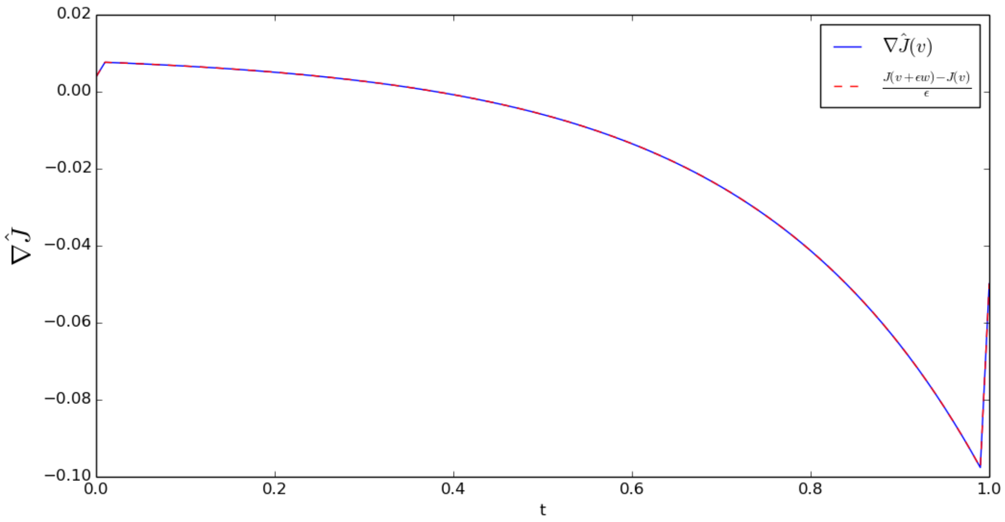


Figure 7.1: Gradient of non-penalized objective function calculated using expression from section 6.1.2, and a finite difference approximation.

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).

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.

Table 7.2: Taylor test for penalized discrete objective function

ϵ	D_1	D_2	$\|\|\epsilon w\|\|_{l_\infty}$	$\log(\frac{D_1(10\epsilon)}{D_1(\epsilon)})$	$\log(\frac{D_2(10\epsilon)}{D_2(\epsilon)})$
1.000000e+00	1.080513e+04	1.076907e+04	9.771288e+01	–	–
1.000000e-01	1.112972e+02	1.076907e+02	9.771288e+00	1.98715	2
1.000000e-02	1.437558e+00	1.076907e+00	9.771288e-01	1.88886	2
1.000000e-03	4.683423e-02	1.076907e-02	9.771288e-02	1.48706	2
1.000000e-04	3.714207e-03	1.076907e-04	9.771288e-03	1.1007	2
1.000000e-05	3.617285e-04	1.076907e-06	9.771288e-04	1.01148	2
1.000000e-06	3.607593e-05	1.076908e-08	9.771288e-05	1.00117	2
1.000000e-07	3.606624e-06	1.076979e-10	9.771288e-06	1.00012	1.99997
1.000000e-08	3.606527e-07	1.086074e-12	9.771288e-07	1.00001	1.99635

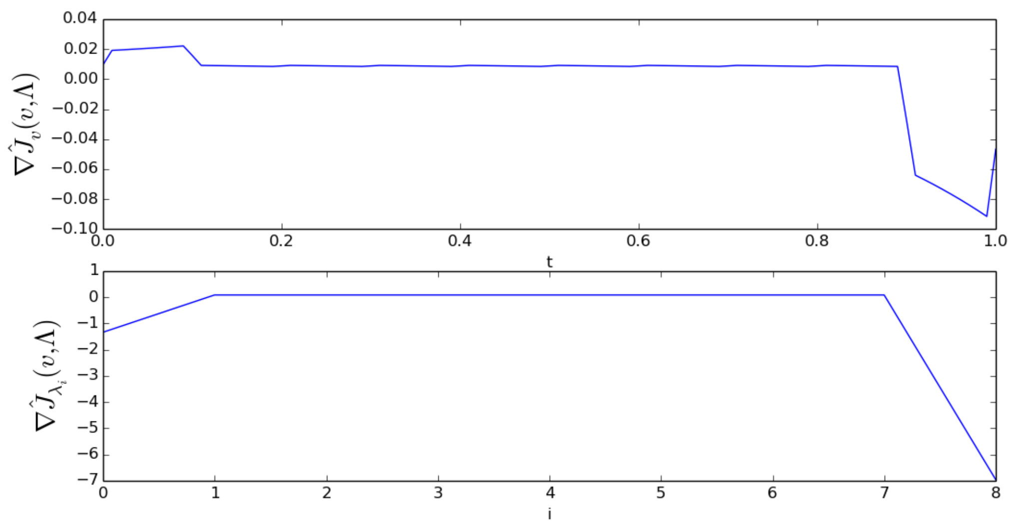


Figure 7.2: Plots showing the Λ and control part of the numerical gradient found using formula (6.19-6.20).

7.2 Convergence rate of solver for the non-penalized problem

In section 7.1 we demonstrated that our implementation of the gradients for different discretizations of the objective function introduced in theorem 10 and 11 satisfy the Taylor test. Since the discretized objective function $\hat{J}_{\Delta t}$ and its gradient pass the Taylor test, we expect that we can find the minimizer $\bar{v}$ of $\hat{J}_{\Delta t}$ by using an optimization algorithm. What we now want to find out, is if the minimizer of the discrete objective function converges towards the exact minimizer derived in section 3.2.2. We investigate this by solving optimal control problem (7.1-7.2) using both a Crank-Nicolson and an implicit Euler discretization. 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

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

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 discirization is found in table 7.3, while Crank-Nicolson results are given in table 7.4. Notice that the

Table 7.3: Convergence of implicit Euler numerical sequential solver of optimal control problem.

Δt	$\frac{\|\|v_e-v\|\|}{\|\|v\|\|}$	$\frac{\hat{J}(v_e)-\hat{J}(v)}{\hat{J}(v_e)}$	norm rate	functional rate
0.02000	0.212619	1.709101e-02	–	–
0.01000	0.136096	4.506561e-03	-0.643642	-1.92314
0.00100	0.017469	4.703915e-05	-0.891585	-1.98139
0.00010	0.001795	4.722900e-07	-0.9883	-1.99825
0.00001	0.000180	4.724790e-09	-0.99882	-1.99983

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. The results of table 7.4 show 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. One last remark concerns the choice of norm (7.5). This norm excludes the values of v and v_e at $t = 0$ and $t = T$. If these points are included, we do not see the convergence rates given in table 7.3 and 7.4.

Table 7.4: Convergence of Crank-Nicolson numerical sequential solver of optimal control problem.

Δt	$\frac{\|\|v_e-v\|\|}{\|\|v\|\|}$	$\frac{\hat{J}(v_e)-\hat{J}(v)}{\hat{J}(v_e)}$	norm r	val r
0.02000	4.177702e-02	2.309886e-03	–	–
0.01000	1.109500e-02	3.383020e-04	-1.9128	-2.77144
0.00100	1.189515e-04	3.931451e-07	-1.96976	-2.93475
0.00010	1.421834e-06	3.992950e-10	-1.92252	-2.99326
0.00001	1.480190e-08	3.978299e-13	-1.98253	-3.0016

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 shows runtime and speedup for both gradient and function evaluation for different Δts and Ns. All evaluations are done with control input $v = 1$ and $\lambda_i = 1$.

Table 7.5: $\Delta t = 10^{-2}$

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.000196	0.000217	1.000000	1.000000
2	0.000207	0.000248	0.946860	0.875000
3	0.000251	0.000288	0.780876	0.753472
4	0.000305	0.000343	0.642623	0.632653
5	0.000360	0.000396	0.544444	0.547980
6	0.000458	0.000452	0.427948	0.480088

Table 7.6: $\Delta t = 10^{-4}$

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.008877	0.015016	1.000000	1.000000
2	0.004475	0.007713	1.983687	1.946843
3	0.003127	0.005332	2.838823	2.816204
4	0.002478	0.004083	3.582324	3.677688
5	0.002080	0.003369	4.267788	4.457109
6	0.001964	0.003016	4.519857	4.978780

Table 7.7: $\Delta t = 10^{-5}$

N	functional time(s)	gradient time(s)	functional speedup	gradient speedup
1	0.087484	0.154841	1.000000	1.000000
2	0.043598	0.075660	2.006606	2.046537
3	0.030286	0.052114	2.888595	2.971198
4	0.022356	0.038681	3.913222	4.003025
5	0.018045	0.031463	4.848102	4.921368
6	0.016126	0.026905	5.425028	5.755101

Table 7.8: $\Delta t = 10^{-7}$

N	functinal time(s)	gradient time(s)	functional speedup	gradient speedup
1	8.350907	14.930247	1.000000	1.000000
2	4.200743	7.233497	1.987960	2.064043
3	2.932549	5.033368	2.847662	2.966254
4	2.190376	3.861509	3.812545	3.866428
5	1.796729	3.089178	4.647839	4.833081
6	1.524042	2.599027	5.479447	5.744552

Since the parallel algorithm has some overhead, we do not expect any improvements for small problems. This is reflected in the above results, where we for

$\Delta t = 10^{-2}$ see an increased execution time when running function and gradient evaluation in parallel. For $\Delta t = 10^{-4}$ we see only a modest speedup, that is significantly lower than the expected speedup from section 6.4. For $\Delta t \leq 10^{-5}$, however we see speedup results in line with what we expect from the theory.

7.4 Consistency

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. 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 solution we get by decomposing and then applying the penalty method on problem (7.1-7.2) with the solution we get by solving the undecomposed problem.

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:

$$
\begin{aligned}
\text{Realtive objective function differnce:} \quad & A = \frac{\hat{J}(v_\mu) - \hat{J}(v)}{\hat{J}(v)} \\
\text{Realtive penalized objective function differnce:} \quad & B = \frac{\hat{J}_\mu(v) - \hat{J}_\mu(v_\mu)}{\hat{J}_\mu(v)} \\
\text{Relative control } L^2\text{-norm difference:} \quad & C = \frac{||v_\mu - v||_{L^2}}{||v||_{L^2}} \\
\text{Maximal jump in decomposed state equation:} \quad & D = \sup_i \{y^i_{k_i} - y^{i+1}_{k_i}\}
\end{aligned}
$$

Notice that both A and B 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 D 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.

The plots in figure 7.3 and 7.4 all show a similar picture, and we observe that all

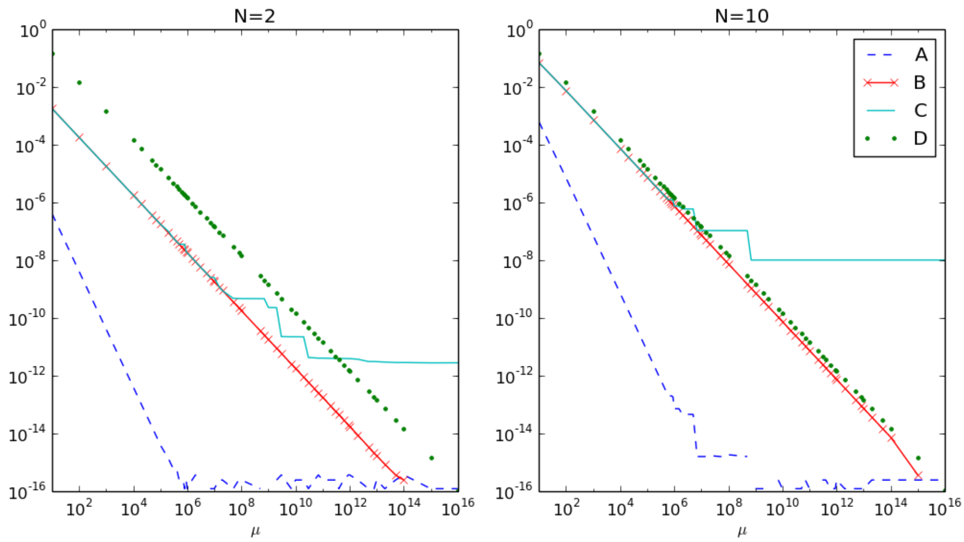


Figure 7.3: Logarithmic plot showing how solution of penalty method applied to problem (7.1-7.2) develops for $\Delta t = 10^{-2}$.

measures decrease when the penalty parameter is increased. Still there are several parts of the plots worthy of note. The measure A related to the unpenalised objective function is the value that converges to zero the fastest. If we look at the vales

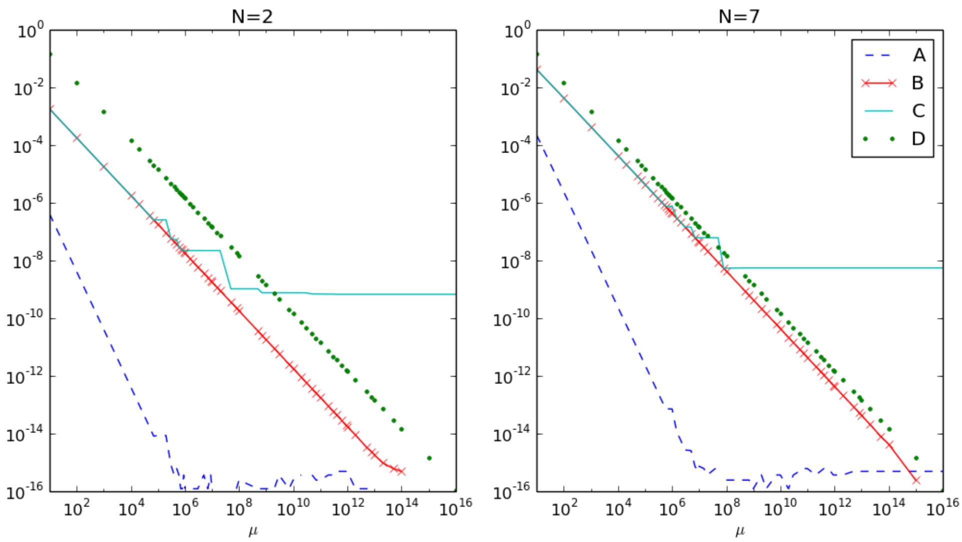


Figure 7.4: Same as figure 7.3, only now $\Delta t = 10^{-3}$.

of A before the machine precision is reached we see that A is proportional to $\frac{C}{\mu^2}$. The convergence rate of A for $\Delta t = 10^{-2}$ and $N = 2$ is shown in table 7.9 together with the rate convergence rate of C. C and the other measures converge to zero at a rate of one, however we see that the relative error between the controls v and v_μ C stops to decrease long before the machine precision is reached. It seems that this barrier is hit around the same time as A approaches machine precision. The reason for this probably is that small changes in the control v_μ no longer registers in $\hat{J}_\mu$, and it is therefore difficult to find an appropriate step length in the line search method.

B and D on the other hand continue to decrease steadily towards zero, even after A has hit machine precision. The B and C 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 means that after a certain point, the penalty method only improves the Λ part of the control, while v remains the same.

Table 7.9: Convergence rates for $\Delta t = 10^{-2}$ and $N = 2$. Notice how the $||v_\mu - v||$ stops to decrease at around the same time as $\frac{J(v_\mu)-J(v)}{J(v)}$ hits machine precision.

μ	$\frac{J(v_\mu)-J(v)}{J(v)}$	$\|\|v_\mu - v\|\|$	A rate	C rate
1.000000e+01	4.105697e-07	1.790231e-03	–	–
1.000000e+02	4.119052e-09	1.793140e-04	-1.998590	-0.9992948
1.000000e+03	4.120272e-11	1.793401e-05	-1.999871	-0.9999368
1.000000e+04	4.137632e-13	1.796793e-06	-1.998174	-0.9991795
2.000000e+04	1.058008e-13	9.076756e-07	-1.967455	-0.9851756
5.000000e+04	1.789909e-14	3.733858e-07	-1.939131	-0.9694245
7.000000e+04	7.968773e-15	2.444327e-07	-2.405011	-1.259160
1.000000e+05	4.045685e-15	1.730018e-07	-1.900553	-0.9690555
2.000000e+05	4.045685e-15	1.721746e-07	0.000000	-0.0069153
3.000000e+05	3.923088e-15	1.719606e-07	-0.007589	-0.0030671
4.000000e+05	3.800492e-15	1.718652e-07	-0.110360	-0.0019278
5.000000e+05	3.677895e-16	4.545076e-08	-10.46580	-5.960652e

Chapter 8

Experiments

In this chapter we will through experiments investigate what speedup one might get by using the algorithm for parallelizing optimal control problems with time dependent DE constraints in temporal direction, introduced in previous chapters. 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 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 the best way of measuring performance of a parallel algorithm is to compare its execution time to the sequential execution time of the best sequential algorithm. 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 this numbers a name:

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

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 implemented efficiently.

8.1 L-BFGS with and without parareal preconditioner

In section 5.3 we introduced the parareal preconditioner, as an approximation to the Hessian. 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. To test this, let us again write up an example problem to be solved:

$$J(y,v) = \frac{1}{2}\int_0^1 v(t)^2 dt + \frac{1}{2}(y(T) - 1.5)^2$$
$$\begin{cases} y'(t) = 0.9y(t) + v(t) \ t \in (0,1) \\ y(0) = 3.2 \end{cases}$$

We will first solve this problem without decomposing the time interval, and then solve the decomposed problem using $N = 2,3,4,5,6,8,16,32,64$ and penalty parameter $\mu = 10^4$. 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 problem for this specific problem. For both the penalized and non-penalized problems we use L-BFGS with stop criteria:

$$||\nabla J||_{L^2} < 10^{-6}$$

Since the point of this test is to compare the effectiveness of the parareal preconditioner, we solve the decomposed problems with and without it. In the result table below 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 non-penalized control solution and all the penalized control solutions. The ideal speedup (8.1) is calculated for preconditioned and unpreconditioned solvers.

Table 8.1: Comparing unpreconditioned and preconditioned solver

N	pc L	non-pc L	$\|\|v_{seq} - v_{pc-par}\|\|$	$\|\|v_{seq} - v_{par}\|\|$	pc $\hat{S}$	non-pc $\hat{S}$
1	31.0	31.0	–	–	1.000000	1.000000
2	37.0	37.0	0.00027766	0.00027766	1.675676	1.675676
3	55.0	55.0	0.000358143	0.000324047	1.690909	1.690909
4	61.0	65.0	0.000359398	0.000207032	2.032787	1.907692
5	65.0	71.0	0.000279559	0.000279684	2.384615	2.183099
6	63.0	77.0	0.000501682	0.000352778	2.952381	2.415584
8	137.0	133.0	0.000498978	0.000499069	1.810219	1.864662
16	77.0	779.0	0.00110429	0.00108333	6.441558	0.636714
32	95.0	1097.0	0.00225308	0.00224816	10.442105	0.904284
64	73.0	2021.0	0.00455414	0.00455295	27.178082	0.981692

There are several things of note about the results in table 8.1. First off we see that the normed difference in control between sequential and parallel solution lies in the range from $3 \cdot 10^{-4}$ to $4 \cdot 10^{-3}$, and that it gets bigger when we increase N. 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. While it seems to be little to no benefit to use the preconditioner for $N = 1, ..., 8$, 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 consistently less then half of 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 $N = 8$, 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.

8.2 Speedup results for $N = 1, ..., 6$

In the previous section we measured the effectiveness of the parareal preconditioner using the ideal speedup (8.1), for a quite large time step $\Delta t = 10^{-3}$, which for $T = 1$ translates to $n = 1000$ timesteps. From section 7.3, we know that we need more time steps than that to get any benefit from the parallelization. This means that if we want to get actual wall clock time speedup, we need to use smaller time steps, or solve the equation on larger time interval. Let us again write up the version of the example problem that we want to solve for this experiment:

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

$$\begin{cases} y'(t) = (t) + v(t) \ t \in (0,1) \\ y(0) = 1 \end{cases}$$

This time we only use the preconditioned solver, but now we also measure the execution time so that it is possible to calculate the speedup. The time measurement is done by running each solver ten times and choosing the lowest time. Also included in the results is the number of function and gradient evaluations L, the ideal speedup $\hat{S}$, the relative difference in function value and between sequential and parallel solutions and the relative l^2-norm difference in control between sequential and parallel solutions. We used the same strict end criteria for the L-BFGS iteration on all solvers:

$$||\nabla J|| < 10^{-10}$$

We solved the penalized problem using only one penalty iteration setting the penalty parameter $\mu = \frac{1}{\Delta t}$. Finally, we ran my experiments using $\Delta t = 10^{-5}, 0.5(10^{-5}), 10^{-6}$. The results follow in table 8.2-8.4.

When comparing the ideal speedup $\hat{S}$ with the actual achieved wall clock speedup in the results found in 8.2, we see that the ideal speedup is a lot higher. We do however notice that both ideal and actual speedup increase for larger N, which indicates that they might be somewhat related.

Table 8.2: $\Delta t = 10^{-5}$

N	L	$\hat{S}$	$\frac{J(v_1)-J(v_N)}{J(v_1)}$	$\frac{\|\|v_1-v_N\|\|_{l2}}{\|\|v_1\|\|_{l2}}$	speedup	time (s)
1:	43	1.000000	–	–	1.000000	4.948604
2:	49	1.755102	1.49099e-10	6e-06	1.349916	3.665862
3:	55	2.345455	6.32732e-10	1.1e-05	1.732804	2.855836
4:	61	2.819672	1.44678e-09	1.6e-05	2.083327	2.375337
5:	67	3.208955	2.62965e-09	2.1e-05	2.263943	2.185834
6:	61	4.229508	4.1827e-09	2.7e-05	2.877279	1.719890

Table 8.3: $\Delta t = 0.5(10^{-5})$

N	L	$\hat{S}$	$\frac{J(v_1)-J(v_N)}{J(v_1)}$	$\frac{\|\|v_1-v_N\|\|_{l2}}{\|\|v_1\|\|_{l2}}$	speedup	time (s)
1:	45	1.000000	–	–	1.000000	10.199495
2:	51	1.764706	3.76222e-11	3e-06	1.322927	7.709792
3:	61	2.213115	1.58185e-10	6e-06	1.580280	6.454233
4:	71	2.535211	3.61721e-10	8e-06	1.802585	5.658261
5:	73	3.082192	6.57328e-10	1.1e-05	2.147440	4.749607
6:	69	3.913043	1.04562e-09	1.3e-05	2.374901	4.294703

For $\Delta t = 0.5(10^{-5})$ and $\Delta t = 10^{-6}$, the results are a bit disappointing, however we do continue to see the a relation between ideal and actual speedup, which justifies the use of $\hat{S}$ as a measure of effectiveness for our algorithm. In the last table for $N = 3$, we notice that the error is high, and that the number of gradient evaluations is low. The reason for this is that the last line search iteration for this specific case failed to satisfy the Wolfe conditions due to numerical errors. The next natural step would then be to do a second penalty iteration with increased penalty parameter and or decreased tolerances, for line search and L-BFGS iterations. This was however not done in this case, since the set-up for the experiment was only to do one penalty iteration.

Table 8.4: $\Delta t = 10^{-6}$

N	L	$\hat{S}$	$\frac{J(v_1)-J(v_N)}{J(v_1)}$	$\frac{\|\|v_1-v_N\|\|_{l2}}{\|\|v_1\|\|_{l2}}$	speedup	time (s)
1:	51	1.000000	–	–	1.000000	56.968187
2:	97	1.051546	1.45061e-12	1e-06	0.871757	65.348693
3:	37	4.135135	1.19209e-07	0.00041	2.995382	19.018671
4:	95	2.147368	1.43499e-11	2e-06	1.538004	37.040329
5:	83	3.072289	4.19647e-11	5e-06	2.098120	27.152017
6:	115	2.660870	4.21117e-11	3e-06	1.780952	31.987489

Bibliography

[1] Eric Aubanel. Scheduling of tasks in the parareal algorithm. *Parallel Computing*, 37(3):172–182, 2011.

[2] Leonardo Baffico, Stephane Bernard, Yvon Maday, Gabriel Turinici, and Gilles Zérah. Parallel-in-time molecular-dynamics simulations. *Physical Review E*, 66(5):057701, 2002.

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

[4] Guillaume Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In *Domain decomposition methods in science and engineering*, pages 425–432. Springer, 2005.

[5] Guillaume Bal and Yvon 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*, pages 189–202. Springer, 2002.

[6] Alfredo Bellen and Marino Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. *Journal of Computational and applied mathematics*, 25(3):341–350, 1989.

[7] Charles G Broyden. The convergence of a class of double-rank minimization algorithms 2. the new algorithm. *IMA Journal of Applied Mathematics*, 6(3):222–231, 1970.

[8] Feng Chen, Jan S Hesthaven, Yvon Maday, and Allan Svejstrup Nielsen. An adjoint approach for stabilizing the parareal method. Technical report, 2015.

[9] John Crank and Phyllis 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*, volume 43, pages 50–67. Cambridge Univ Press, 1947.

[10] Xiaoying Dai and Yvon Maday. Stable parareal in time method for first-and second-order hyperbolic systems. *SIAM Journal on Scientific Computing*, 35(1):A52–A78, 2013.

[11] Lisandro Dalcin. mpi4py, 2007.

[12] Charbel Farhat and Marion Chandesris. Time-decomposed parallel time-integrators: theory and feasibility studies for uid, structure, and fluid-structure applications. *International Journal for Numerical Methods in Engineering*, 58(9):1397–1434, 2003.

[13] Paul F Fischer, Frédéric Hecht, and Yvon Maday. A parareal in time semi-implicit approximation of the navier-stokes equations. In *Domain decomposition methods in science and engineering*, pages 433–440. Springer, 2005.

[14] Roger Fletcher. A new approach to variable metric algorithms. *The computer journal*, 13(3):317–322, 1970.

[15] Martin J Gander. 50 years of time parallel time integration. In *Multiple Shooting and Time Domain Decomposition Methods*, pages 69–113. Springer, 2015.

[16] Martin J Gander and Stefan Vandewalle. Analysis of the parareal time-parallel time-integration method. *SIAM Journal on Scientific Computing*, 29(2):556–578, 2007.

[17] Martin J Gander and Stefan Vandewalle. On the superlinear and linear convergence of the parareal algorithm. In *Domain decomposition methods in science and engineering XVI*, pages 291–298. Springer, 2007.

[18] Martin Jakob Gander. Overlapping schwarz for linear and nonlinear parabolic problems. 1996.

[19] Izaskun Garrido, Magne S Espedal, and Gunnar E Fladmark. A convergent algorithm for time parallelization applied to reservoir simulation. In *Domain Decomposition Methods in Science and Engineering*, pages 469–476. Springer, 2005.

[20] Jean Charles Gilbert and Claude Lemaréchal. Some numerical experiments with variable-storage quasi-newton algorithms. *Mathematical programming*, 45(1):407–435, 1989.

[21] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26, 1970.

[22] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[23] Stefan Güttel. A parallel overlapping time-domain decomposition method for odes. In *Domain decomposition methods in science and engineering XX*, pages 459–466. Springer, 2013.

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

[25] Michael Hinze, René Pinnau, Michael Ulbrich, and Stefan Ulbrich. *Optimization with PDE constraints*, volume 23. Springer Science & Business Media, 2008.

[26] Graham Horton and Stefan Vandewalle. A space-time multigrid method for parabolic partial differential equations. *SIAM Journal on Scientific Computing*, 16(4):848–864, 1995.

[27] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[28] Ekachai 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.

[29] Bianca Lepsa and Adrian Sandu. An efficient error control mechanism for the adaptive'parareal'time discretization algorithm. In *Proceedings of the 2010 Spring Simulation Multiconference*, page 87. Society for Computer Simulation International, 2010.

[30] Jacques-Louis Lions, Yvon Maday, and Gabriel Turinici. Résolution d'edp par un schéma en temps «pararéel». *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 332(7):661–668, 2001.

[31] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[32] Ch Lubich and A Ostermann. Multi-grid dynamic iteration for parabolic equations. *BIT Numerical Mathematics*, 27(2):216–234, 1987.

[33] YVON Maday, E Rønquist, and GUNNAR ANDREAS Staff. The parareal-in-time algorithm: Basics, stability analysis, and more. *Preprint*, pages 1–20, 2007.

[34] Yvon Maday and Einar M Rønquist. Parallelization in time through tensor-product space–time solvers. *Comptes Rendus Mathematique*, 346(1-2):113–118, 2008.

[35] Yvon Maday, Julien Salomon, and Gabriel Turinici. Monotonic parareal control for quantum systems. *SIAM Journal on Numerical Analysis*, 45(6):2468–2482, 2007.

[36] Yvon Maday and Gabriel Turinici. A parareal in time procedure for the control of partial differential equations. *Comptes Rendus Mathematique*, 335(4):387–392, 2002.

[37] Yvon Maday and Gabriel Turinici. Parallel in time algorithms for quantum control: Parareal time discretization scheme. *International journal of quantum chemistry*, 93(3):223–228, 2003.

[38] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. van der Voort S, Millman J, 2010.

[39] Willard L Miranker and Werner Liniger. Parallel methods for the numerical integration of ordinary differential equations. *Mathematics of Computation*, 21(99):303–320, 1967.

[40] Jürg Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, 1964.

[41] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

[42] Jorge Nocedal and Stephen J Wright. Numerical optimization 2nd, 2006.

[43] Vishwas Rao and Adrian Sandu. An adjoint-based scalable algorithm for time-parallel integration. *Journal of Computational Science*, 5(2):76–84, 2014.

[44] Vishwas Rao and Adrian Sandu. A time-parallel approach to strong-constraint four-dimensional variational data assimilation. *Journal of Computational Physics*, 313:583–593, 2016.

[45] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[46] Gunnar Andreas Staff and Einar M Rønquist. Stability of the parareal algorithm. In *Domain decomposition methods in science and engineering*, pages 449–456. Springer, 2005.

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

[48] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.