NTNU

Norwegian University of Science and Technology

Efficient optimization in Model Predictive Control

Ruben Køste Ringset

Master of Science in Engineering Cybernetics

Submission date:	July 2009
Supervisor:	Bjarne Anton Foss, ITK
Co-supervisor:	Lars Struen Imsland, Cybernetica AS

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Practical use of optimization, for MPC or parameter estimation, requires a large number of gradient calculations. These gradients are used to compute search directions, for instance in a SQP algorithm. Computing gradients is time-consuming and limits the use of for instance MPC to small and medium-sized systems.
Gradients are usually computed by finite difference methods. Alternatives include forward- and adjoint-based methods. The latter is efficient for problems with many decision variables and few outputs. Efficiency may however deteriorate in cases with output constraints which typically are present in MPC. In this project which continues earlier work the use of adjoints as a means to increase efficiency of optimization in MPC will be further studied.

Tasks:

1. Present adjoint-based methods and review central literature. The presentation shall focus on MPC which apply a sequential approach.

2. Study efficient ways to compute the impulse response matrix (including changes in the MPC formulation), in particular how adjoints can aid this.

3. Output constraints can be detrimental to the efficiency of adjoint-based methods. Hence, optimization algorithms where explicit output constraints can be removed, as for instance in barrier methods, is an interesting option to exploit the efficiency of adjoint-based methods. Propose methods in which the efficiency of adjoints can be exploited.

4. Evaluate the methods above by comparing them with forward methods. This should be done by identifying suitable examples as well as test scenarios for evaluating the methods.

The thesis report may include a paper to a selected conference with the main results of this work.

Assignment given: 26. January 2009
Supervisor: Bjarne Anton Foss, ITK

Abstract

This thesis is about calculation of derivative information as a means to increase efficiency of optimization algorithms in nonlinear model predictive control (NMPC). NMPC is briefly discussed to set up a framework and define notation. Some general optimization techniques are introduced. The next sections are about computation of derivative information in general and include finite difference techniques as well as the forward and the adjoint method. Next, these techniques are used to obtain derivative information for NMPC. Forward and adjoint techniques are compared, both theoretically and by applying them to suitable test examples. The discussion includes both discrete and continuous-discrete system models. It is well known that the adjoint method is efficient for obtaining sensitivity of a low dimensional function with respect to a large number of parameters. If the optimization problem is posed by single shooting and there are no constraints on the output variables, adjoints will be very efficient, as only the objective function gradient is needed. However, in most cases these constraints cannot be removed. Constraint lumping or optimization algorithms like for instance penalty and barrier methods, which include these constraints as penalty terms in the objective function, are interesting alternatives. These approaches allow for very efficient gradient calculation using adjoints, but comes at the expense of other difficulties.

Preface

This master thesis is done as a compulsory part of the study for the degree Master of Science in Engineering Cybernetics at the Norwegian University of Technology and Science (NTNU).

The thesis continues earlier work from a project thesis. The work has been done in cooperation with Cybernetica AS, a Norwegian company which provides model based control systems for the process industry. I would like to thank them for the opportunity to work with them.

I would also like to express my gratitude to both of my supervisors:

Professor Bjarne Anton Foss at the Department of Engineering Cybernetics, NTNU for introducing me to adjoints and for his inspiration and support. During the last months Bjarne has provided me with the opportunity to take part in several interesting activities and he has also put me in contact with people from other parts of the world doing research on the same topics.

Lars Struen Imsland at Cybernetica AS for giving of his time and providing excellent guidance. During the nice discussions we have had at the office, Lars has always given valuable advice.

Finally, I would also like to thank the most important people in my life, Monica and both my parents for always being there for me.

Ruben Ringset, July 2009

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Linear model predictive control (MPC) has gained a lot of popularity over the years. Nonlinear model predictive control (NMPC) has also received more attention in the last years, both by practitioners and theorists. Unfortunately, NMPC does not enjoy the well established theory for linear MPC, and this fact sometimes limits its applicability to small and medium-sized systems. The optimization for NMPC is for instance, much more computationally demanding which makes it harder to meet the real time requirements, especially for large scale models. Gradient based optimization techniques require derivative information in each iteration. Obtaining these gradients can be a very computationally demanding task as they typically involve a lot of simulations of the system model.

However, by using the adjoint method the number of simulations to obtain the objective function gradient can be dramatically reduced as this can be done by only 2 simulations - one in forward time and one in reverse time. Traditional techniques for obtaining the objective function gradient like finite differencing require $L+1$ simulations of the system model where L is the number of control variables on the prediction horizon. Through this thesis, different methods for obtaining derivatives will be evaluated and compared. As will be discussed more later, different optimization techniques require different types of derivative information. The advantages of choosing the best optimization algorithm and the best method for obtaining derivatives will often conflict. Through this thesis, some of these issues will be addressed by evaluating optimization techniques together with algorithms for obtaining derivatives.

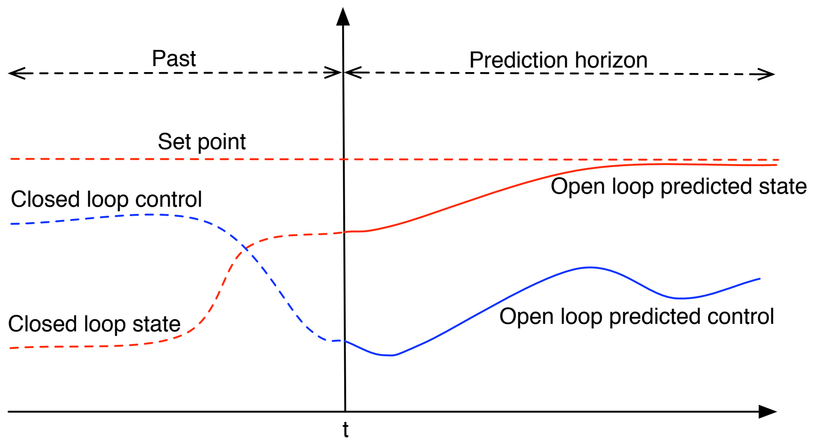
The structure of this thesis is as follows: Section 2 introduces MPC, sets up a framework and defines notation. Section 3 focus on how the NMPC optimization problem is posed and briefly introduces some optimization techniques like sequential quadratic programming (SQP), penalty methods, augmented Lagrangian, and interior point methods. Section 4 and 5 are about derivative calculation in general. The forward and reverse methods are compared, and some simple examples are given. The reader may find it instructive to understand this theory for more general problems before these techniques are applied to the NMPC problem. In section 6 the theory from section 4 and 5 is applied to the NMPC problem. The discussion includes calculation of the objective function gradient and the impulse response matrix which essentially is the output constraint gradient. Both discrete and continuous-discrete models are discussed and a number of different algorithms are given. Theoretical bounds for runtime of these algorithms are compared to actual simulation results using suitable test examples. In section 7 we look at constraint lumping and optimization techniques like barrier and penalty methods. These methods reduce the dimension of the functions of which the sensitivity with respect to the control variables is needed. This makes adjoint gradient calculation efficient, but comes at the expense of some other difficulties which need to be resolved. Section 8 adds some final remarks and conclusions.

2 Nonlinear Model Predictive Control

In this section a short introduction to model predictive control (MPC) and an outline of the MPC problem is given. Model predictive control is a methodology or class of advanced control algorithms which use a dynamic system model of the plant (for example an ODE or DAE model) to predict and optimize behavior of the plant into the future. Linear MPC has shown great success in applications, especially in the process industry and is spreading to other application areas [17]. MPC handles multivariable systems (MIMO), constraints on inputs and outputs and possibly states in a very transparent manner. One can argue that a model predictive controller operates in a similar way as an experienced human operator would operate the process, since also the operator of a plant will use knowledge of the plant dynamics and couplings to predict and optimize behavior in the future.

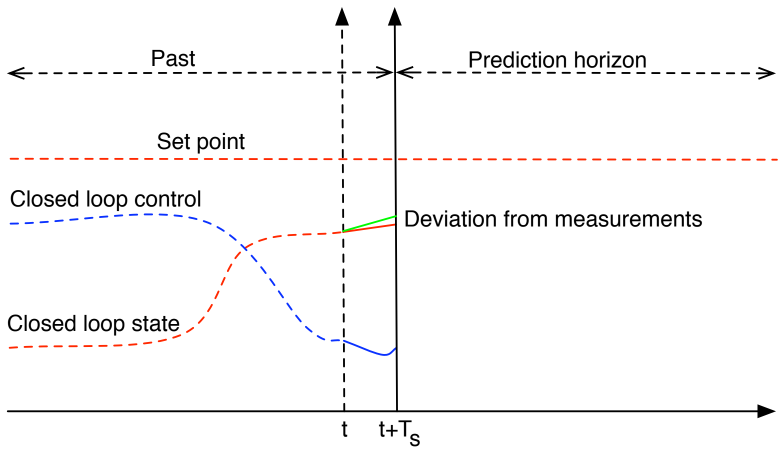
MPC algorithms are control algorithms based on solving an online optimization problem. The optimization algorithm minimizes some objective function which reflects the desired control performance subject to the model of the system and possibly constraints on inputs, states and outputs. The solution of the optimization problem is a set of controls into the future which will be optimal with respect to the specified objective function and the constraints on the prediction horizon. This principle is illustrated in figure 2.1.

Figure 2.1: MPC principle



For the nominal case where there is no model-plant mismatch, process noise or measurement noise, we could just optimize one time and then apply the optimized solution in the future. However this approach will not be robust due to modeling errors and noise. This is shown in figure 2.2 which illustrates that after some time T_s, the predicted trajectory may deviate from the real trajectory.

Figure 2.2: MPC principle - Model error



Since this scheme is not robust, feedback must be incorporated into the system. This is done by applying the control signal to the process until time T_s when the next measurement becomes available. Then, optimization and prediction is performed again on a receding horizon taking the new measurements into account.

2.1 Mathematical formulation of NMPC

In this section the NMPC optimization problem is formulated and notation is defined. Some common choices of objective functions for NMPC are presented. Properties like constraint handling and nominal stability of the MPC controller are briefly discussed.

Suppose the nonlinear model of the system is on the following form

$$x_{k+1} = f(x_k, u_k),\ z_k = g(x_k, u_k),\ x_0 = x(t_0),\ x_k \in \mathbb{R}^{N_x},\ u_k \in \mathbb{R}^{N_u},\ z_k \in \mathbb{R}^{N_z},$$

where x_k is the state vector, u_k the controlled inputs and z_k the controlled outputs. Define the vectors

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_N \end{pmatrix},\ u = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix},\ z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{N-1} \end{pmatrix}.$$

There are also constraints on the inputs and outputs given by some upper and lower bounds u_{min}, u_{max}, z_{min}, z_{max} and some objective function J we want to optimize. It is also possible to specify input and output constraints by more complicated functions, but for simplicity, we stick to simple box constraints. The objective function J together with these bounds specify the desired control performance.

Model predictive control is a control strategy that is often used on systems with many inputs and outputs, and it is very common that the MPC algorithm only controls set points for conventional controllers further down in the control hierarchy. In these situations objective functions may for instance be derived directly from economical considerations of the plant operation. One possibility is to penalize deviation from a precomputed reference trajectory by some norm which yields the following objective function

$$J = \sum_{k=0}^{Np} \|z_k - z_k^r\| + \sum_{k=0}^{Nc} \|u_k - u_k^r\| \,. \tag{2.1}$$

Here N_p and N_c is the prediction horizon and the control horizon, respectively . The control horizon is typically chosen shorter than the prediction horizon, i.e. input blocking. In the following, these horizon lengths are for simplicity assumed equal by setting $N = N_p = N_u$.

One very common objective function for model predictive control is the following quadratic function

$$J = \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N,\ Q = Q^T \geq 0,\ R = R^T \geq 0,\ P = P^T > 0,$$

where it is assumed that we want to regulate the system to the origin. Due to the positive (semi)definiteness of Q, R, and P, this function is convex. For linear system models and convex inequality constraints on inputs and outputs, the optimization problem will be a convex quadratic problem, which of global solutions can be found reliably. For NMPC, the system model which is posed as an equality constraint is nonlinear, resulting in a non-convex problem where global solutions can be hard to find.

The terminal cost term $\frac{1}{2}x_N^T P x_N$ is often included to penalize deviation from the desired state at the end of the horizon. For linear MPC this term can be chosen to satisfy

$$\frac{1}{2}x_N^T P x_N = \frac{1}{2}\sum_{k=N}^{\infty}[z_k^T Q z_k + u_k^T R u_k]$$

by solution of a Riccati equation [17]. Given that no output or input constraints are active for $k \geq N$, the solution will yield the linear quadratic regulator (LQR) for $k \geq N$, which will be optimal. We will not elaborate further on this, but we note that it allows the use of an objective function with infinite horizon by solving a finite dimensional optimization problem. This separation is often referred to as dual-mode MPC and was one of the keys in proving stability for linear MPC.

Another possibility for proving stability of linear MPC is to add a dead beat constraint at the end of the horizon, i.e. $x_N = 0$. This will result in stable closed loop when the optimization problem has a feasible solution. Applying a dead beat constraint will in fact also provide stability for nonlinear MPC [11]. However, adding a dead beat constraint is quite restrictive, and one might run into feasibility problems for short horizon lengths.

By using the quadratic objective function above, the NMPC optimization problem that must be solved at each sampling instant can be formulated as

$$\min\ J \quad = \quad \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N, \tag{2.2}$$

subject to

$$\begin{aligned} x_{k+1} &= f(x_k, u_k), & (2.3)\\ z_k &= g(x_k, u_k), & (2.4)\\ x_0 &= x(t_0), & (2.5)\\ u_{min} \leq\ u_k &\leq u_{max}, & (2.6)\\ z_{min} \leq\ z_k &\leq z_{max}. & (2.7) \end{aligned}$$

This will be used as the basic formulation throughout this thesis. The nonlinear program (NLP) (2.2)-(2.7) can be solved by a number of different techniques. This is discussed in more detail in the next section. There is however no guarantee that the NLP (2.2)-(2.7) will have a feasible solution, even though it is crucial that the NMPC algorithm calculates an input to the system at every sampling instant. When no feasible solution is found, we need a way to relax the NLP.

One way to relax the problem is to let the optimization algorithm deliberately break some of the constraints on inputs or outputs. In MPC it is possible to have both hard and soft constraints on inputs and outputs. Hard constraints should not be violated at any time, e.g. a valve cannot be more than closed, a tank cannot contain a negative volume of liquid etc. Soft constraints can for example be related to product quality and such. These should preferably not be violated, but they can be violated when it is necessary in order for the optimization problem to have a solution. A typical situation is that input constraints are hard since these often are related to physical limitations of actuators, while output constraints are soft since outputs often relate to product quality or other quantities that do not have absolute bounds.

The optimization problem with soft constraints can be formulated by adding extra slack variables to the objective function. For example, considering all the constraints on the outputs as soft and adding a quadratic penalty term for violating these constraints will yield the following optimization problem

$$\min\ J \quad = \quad \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N + \mu\,\|\epsilon\|_2^2\,, \tag{2.8}$$

subject to

$$\begin{aligned} x_{k+1} &= f(x_k, u_k), && (2.9)\\ z_k &= g(x_k, u_k), && (2.10)\\ x_0 &= x(t_0), && (2.11)\\ u_{min} \leq u_k &\leq u_{max}, && (2.12)\\ \begin{bmatrix} z_k - z_{max} \\ -z_k + z_{min} \end{bmatrix} &\leq \begin{bmatrix} \epsilon_{max} \\ \epsilon_{min} \end{bmatrix} = \epsilon, && (2.13)\\ 0 &\leq \begin{bmatrix} \epsilon_{max} \\ \epsilon_{min} \end{bmatrix} = \epsilon. && (2.14) \end{aligned}$$

The optimizer will have a strong motivation to keep ϵ zero whenever possible. μ is an extra design parameter which can be chosen in terms how much we would like to penalize violation of constraints. $\mu = 0$ yields the unconstrained problem and by choosing μ large, this becomes the hard constrained problem.

It is also possible to use other penalty functions than the quadratic function above. It is desirable to have an exact penalty function which has the property that under certain conditions on the penalty parameter μ related to the norm of the Lagrange multipliers, constraints are not violated unless there is no feasible solution to the hard constrained problem [20]. The multipliers are however not known a priori. Exact penalty functions in the context of NMPC will be discussed in more detail later.

3 Optimization algorithms

The NMPC optimization problem is a general nonlinear programming problem (NLP) to which there are many different approaches and algorithms available. A good overview and classification of efficient numerical optimization algorithms for solution of the NMPC problem can be found in [8]. It is assumed that the reader is somewhat familiar with the field of numerical optimization and is referred to [20] for a text on this topic which also parts of the most relevant basics introduced in the following section are based on. We first discuss different ways of posing the optimization problem. Then we briefly introduce sequential quadratic programming (SQP) which is an optimization technique widely used in MPC software. We also briefly introduce some other techniques like penalty methods, augmented Lagrangian and interior point methods. Common for these three latter methods is that the constraints are appended to the objective function as a penalty term. These methods will later prove themselves as useful formulations as the derivative information required by the optimizer may be very efficiently evaluated using the adjoint method. These kind of methods will be revisited in the context of NMPC with output constraints and adjoints in section 7.

3.1 Formulation of the NMPC optimization problem

The optimization problem that is solved at every time instant can be posed in different ways. Which method that will be suitable becomes a trade-off between the size of the variable space, structure in the problem and integration or separation between optimization and system simulation.

Sequential approach

This method is also referred to as 'single shooting' or 'reduced space'. Consider the nonlinear program (2.2)-(2.7). If the vector u is fixed, x and z will be uniquely determined. Existence and uniqueness for discrete time systems is simply guaranteed just by $f(x_k, u_k)$ being a function. For continuous time models this is a somewhat more complicated matter depending on additional conditions [15] which are assumed to hold throughout this thesis. It is therefore possible to address the optimization problem with just u as free optimization variables by considering $x(u)$ and $z(u)$ as implicit functions which can be found by simulation. In each step of the optimization algorithm, system simulation and optimization are performed sequentially. This approach will have a reduced variable space compared to the full nonlinear program (2.2)-(2.7). However a disadvantage is that the subproblems will have less structure and the linear algebra will typically involve dense matrices. Another disadvantage with this approach is that simulation and optimization are performed sequentially giving the optimization algorithm no control over the simulation. Thus, highly nonlinear or unstable systems can be challenging.

Simultaneous approach

This method is also referred to as 'full space'. This approach addresses the full nonlinear program (2.2)-(2.7) with both u, x and z as free optimization variables. Advantages are that the optimization has better control over the simulation as both optimization and simulation are performed simultaneously. The method is therefore better suited for unstable or highly nonlinear systems. The main disadvantage with this method is that the nonlinear program is addressed in its full variable space. However the underlying subproblems in this approach have much more structure and one can make use of linear algebra tools that takes the sparse banded structure into account [17].

Multiple shooting

This method can be viewed as a combination of the sequential and the simultaneous approach. Multiple shooting divides the horizon into many subhorizons. The sequential approach is then performed on each of these sub-horizons. Extra equality constraints where the state at the end of a subhorizon should match the state of the next subhorizon are added. This method allows combining the sequential and the simultaneous approach in the way that is suitable for the problem at hand. Compared to the sequential approach, multiple shooting gives more control over the simulation since the nonlinearities are spread out on smaller subhorizons rather than simulating the whole prediction horizon before running the optimization.

Throughout this thesis the sequential approach will be used as it allows for very efficient gradient calculation using adjoint techniques.

3.2 Sequential Quadratic Programming (SQP)

SQP algorithms generate search directions by solving a sequence of quadratic programs (QP) which can be used both with line search and trust region methods.

Consider the nonlinear program

$$\min\ f(x), \tag{3.1}$$

subject to

$$\begin{aligned} g(x) &= 0, & (3.2)\\ h(x) &\leq 0. & (3.3)\end{aligned}$$

Let the current iterate be defined by x_k and the next iterate by $x_{k+1} = x_k + p_k$ where p_k is the search direction. In unconstrained optimization algorithms the search direction can be determined by either going the steepest descent of the objective function ($x_{k+1} - x_k = p_k = -\nabla_x f(x_k)$), by the Newton direction or maybe a quasi Newton direction in the presence of inexact Hessians. In contrast, SQP algorithms compute the search direction by solving a local quadratic program.

First, introduce the Lagrangian $\mathcal{L}(x, \lambda, \mu) = f(x) - \lambda^T g(x) - \eta^T h(x)$. Let $m_{\mathcal{L}}(p_k)$ be an approximate quadratic model of $\mathcal{L}(x_{k+1}) = \mathcal{L}(x_k + p_k)$ given by the second order Taylor expansion

$$m_{\mathcal{L}}(p_k) = \mathcal{L}(x_k) + \nabla_x \mathcal{L}(x_k)^T p_k + \frac{1}{2} p_k^T \nabla_{xx}^2 \mathcal{L}(x_k) p_k,$$

and the linearized constraints be given by the first order Taylor expansion

$$\begin{aligned} g_i(x_k) + \nabla_x g_i(x_k)^T p_k &= 0, \\ h_j(x_k) + \nabla_x h_j(x_k)^T p_k &\leq 0. \end{aligned} \tag{3.4}$$

SQP methods iteratively solve the quadratic problem[1]

$$\min \; m_{\mathcal{L}}(p_k),$$

subject to

$$\begin{aligned} g_i(x_k) + \nabla_x g_i(x_k)^T p_k &= 0, \\ h_j(x_k) + \nabla_x h_j(x_k)^T p_k &\leq 0, \end{aligned}$$

until convergence is achieved. This QP can in fact be interpreted as applying Newton's method to the Karush–Kuhn–Tucker (KKT) optimality conditions [20] given by

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, \lambda^*, \eta^*) &= 0, \\ g(x^*) &= 0, \\ 0 \geq h(x^*) &\perp \eta^* \geq 0, \end{aligned}$$

where the notation $\perp$ is used to denote perpendicularity. That is, also the complementary condition $h(x^*)^T \eta^* = 0$ should hold.

Inequality based QP (IQP)

This approach makes the decision about which of the inequality constraints that are active in the quadratic subproblems, and states the QP problems with all the linearized inequality constraints present.

$$\min \; f(x_k) + \nabla_x f(x_k)^T p_k + \frac{1}{2} p_k^T \nabla_{xx}^2 \mathcal{L}(x_k) p_k,$$

subject to

$$\begin{aligned} g_i(x_k) + \nabla_x g_i(x_k)^T p_k &= 0, \\ h_j(x_k) + \nabla_x h_j(x_k)^T p_k &\leq 0. \end{aligned}$$

[1] $m_{\mathcal{L}}(p_k)$ may be simplified to $m_{\mathcal{L}}(p_k) = f(x_k) + \nabla_x f(x_k)^T p_k + \frac{1}{2} p_k^T \nabla_{xx}^2 \mathcal{L}(x_k) p_k$ since $g_i(x_k) + \nabla_x g_i(x_k)^T p_k = 0$ and $\eta_j(h_j(x_k) + \nabla_x h_j(x_k)^T p_k) = 0$.

Equality based QP (EQP)

This variant makes the decision about which of the inequality constraints that seem to be active before stating the QP subproblems and maintain these in a working set W_k. The QP subproblem at each iteration is stated as

$$\min\ f(x_k) + \nabla_x f(x_k)^T p_k + \frac{1}{2} p_k^T \nabla_{xx}^2 \mathcal{L}(x_k) p_k,$$

subject to

$$\begin{aligned} g_i(x_k) + \nabla_x g_i(x_k)^T p_k &= 0, \\ h_j(x_k) + \nabla_x h_j(x_k)^T p_k &= 0\ \forall j \in W_k. \end{aligned}$$

The working set W_k is updated based on Lagrange multipliers of the QP subproblem and evaluation of $h_j(x_{k+1})\ \forall j \notin W_k$.

Line Search

The convergence can be improved by using a line search method to determine the distance α_k to go along the search direction. One approach is to use a merit function to decide which point is better than the other. The l_1 merit function is defined as

$$\phi(x_k, \nu) = f(x_k) + \nu \sum_i |g_i(x_k)| + \nu \sum_j \max(h_j(x_k), 0). \tag{3.5}$$

The step size α_k is chosen such that the merit function $\phi(x_k + \alpha_k p_k, \nu)$ has sufficient decrease which will be a trade-off between decreasing the value of f and infeasibility of the current iterate. A desirable property with the l_1 penalty function is that it is exact. A challenge with the l_1 merit function is that it is not differentiable [20].

3.3 Penalty methods

Quadratic penalty function

Consider the NLP (3.1)-(3.3). Penalty methods reformulate the problem to an unconstrained formulation by penalizing violation of the constraints in the objective function. In the quadratic penalty method a sequence of unconstrained problems on the following form are solved

$$\min\ Q(x, \mu) = f(x) + \frac{\mu}{2} \sum_i g_i(x)^2 + \frac{\mu}{2} \sum_j \max(h_j(x), 0)^2. \tag{3.6}$$

The objective function Q is minimized for increasing values of μ until convergence is achieved. The rationale is that the constraints are resolved as they are penalized more by increasing μ, resulting in the solution of (3.8) approaching the solution of (3.1)-(3.3).

The problem with this method is that (3.8) becomes ill conditioned as μ gets large. This will result in the Hessian matrix in quasi Newton methods like Broyden-Fletcher-Goldfarb-Shanno (BFGS) and even Newton's method to be ill conditioned. To see this, for simplicity consider the problem (3.8) with only equality constraints. At the optimum x^* of (3.1)-(3.2) (only equality constraints) we have that

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla_x f(x) - \sum_i \lambda_i \nabla_x g_i(x) = 0.$$

By comparison with

$$\nabla_x Q(x,\mu) = \nabla_x f(x) + \mu \sum_i g_i(x) \nabla_x g_i(x) = 0,$$

we see that

$$\mu g_i(x) \to -\lambda_i, \tag{3.7}$$

as $x_k \to x^*$ and $\mu \to \infty$ in (3.8). Then consider the Hessian matrix

$$\nabla^2_{xx} Q(x,\mu) = \nabla_x \left[\nabla_x f(x) + \mu \sum_i g_i(x) \nabla_x g_i(x) \right] = \nabla^2_{xx} f(x) + \mu \sum_i g_i(x) \nabla^2_{xx} g_i(x) + \mu A(x)^T A(x),$$

where $A(x) = [\nabla_x g_1(x) \nabla_x g_2(x) \ldots]^T$. Now, near the optimum we will have $\mu g(x) \approx -\lambda$, and thus

$$\nabla^2_{xx} Q(x,\mu) \approx \nabla^2_{xx} \mathcal{L}(x,\lambda) + \mu A(x)^T A(x).$$

The problem here is that the eigenvalues of the matrix $A(x)^T A(x)$ will be either zero ($A(x)^T A(x)$ not full rank) or in the same order of magnitude as μ. This makes the Hessian matrix arbitrarily ill conditioned near the optimum x^*, and will cause problems when computing the Newton direction p by solving

$$\left[\nabla^2_{xx} Q(x,\mu) \right] p = -\nabla_x Q(x,\mu).$$

l_1 penalty function

Instead of using a quadratic penalty we may instead formulate the penalty term using an l_1 penalty function which gives the following optimization problem

$$\min \; P_{l_1}(x,\mu) = f(x) + \mu \sum_i |g_i(x)| + \mu \sum_j \max(h_j(x), 0). \tag{3.8}$$

The most desirable property of the l_1 penalty function is exactness which in this context means that the minimizer of (3.8) will coincide with the minimizer of the original NLP (3.1)-(3.3) as long as μ is chosen large enough. That is, the particular value of μ is not crucial for that solving (3.8) will in fact give the solution of the original NLP. The condition on the penalty parameter is that

$$\mu > \|\lambda^*\|_\infty \,, \tag{3.9}$$

where λ^* is the vector of Lagrange multipliers of (3.1)-(3.3) at x^*.

Even though this exactness is a nice property, the l_1 penalty method has some disadvantages as well. Due to the absolute value function and the $\max$ function in (3.8), the l_1 penalty function is not differentiable at certain points, making it hard to employ derivative based optimization techniques. In fact, nonsmoothness of the penalty function is a necessity for exactness of penalty functions [20].

Another obvious challenge with this method is how to choose the penalty parameter to ensure $\mu > \|\lambda^*\|_\infty$ since the Lagrange multipliers at the optimum are of course not known in advance. One might consider choosing μ very large and hope that this works for most cases, but this may lead to a poorly scaled problem which may be hard to solve.

Due to (3.9), estimates of the Lagrange multipliers may provide some information about choosing μ. This may not work well if the estimates of these multipliers are inexact and even only for the fact that the estimates of these multipliers far from the solution may not be a good value for choosing μ.

Another approach may be to start by solving (3.8) for a small value of μ. If the solution is not feasible, μ is increased and (3.8) is solved again with the solution for the previous problem as starting point. The algorithm may adaptively update μ depending on the difficulty of solving the optimization problem in the previous iteration. However, if μ is chosen too small, violation of constraints may not be penalized enough. This may result in a problem that is unbounded below and consequently, the iterates may diverge. If this happens, the initial point for the next iteration should be reset and μ should be increased until the problem is bounded below. Thus, some monitoring and heuristics may be required.

3.4 Augmented Lagrangian

Consider adding a penalty term to the Lagrangian instead of the objective function as in the previous section to define the augmented Lagrangian (assume no inequality constraints)

$$\mathcal{L}_\mathcal{A}(x, \lambda, \mu) = f(x) - \lambda^T g(x) + \frac{\mu}{2} \|g(x)\|_2^2 \,.$$

Now, comparing

$$\nabla_x \mathcal{L}_\mathcal{A}(x, \lambda, \mu) = \nabla_x f(x) - \sum_j (\lambda_j - \mu g_j(x)) \nabla_x g_j(x)$$

to the optimality condition $\nabla_x \mathcal{L}(x, \lambda) = 0$ reveals that

$$\lambda - \mu g(x) \to \lambda^*, \tag{3.10}$$

and by rearrangement that $g(x) \to \frac{1}{\mu}(\lambda - \lambda^*)$ as $x \to x^*$. Now, $g(x) \approx 0$ can be achieved even for μ not large provided that $\lambda \approx \lambda^*$. Augmented Lagrangian methods therefore keep track of estimates of the Lagrange multipliers λ. A simple iterative formula for estimating λ can be deduced from (3.10)

$$\lambda_{k+1} = \lambda_k - \mu g(x_k)$$

(with slight abuse of notation since k here denotes the iteration index).

3.5 Barrier and interior point methods

(3.1)-(3.3) may be reformulated as

$$\min\ f(x) + \sum_j I_f(h_j(x)), \tag{3.11}$$

subject to

$$g(x) = 0, \tag{3.12}$$

where I_f is the indicator function that specifies the feasible domain and is defined as

$$I_f(x) = \begin{cases} \infty & x > 0 \\ 0 & x \leq 0 \end{cases}. \tag{3.13}$$

The problem (3.11)-(3.12) can be approximated by approximating the indicator function (3.13) by a a logarithmic barrier

$$\min \psi(x, \mu) = f(x) - \mu \sum_j \ln(-h_j(x)), \tag{3.14}$$

subject to

$$g(x) = 0. \tag{3.15}$$

The logarithmic term added to the objective function acts as a barrier that tends to infinity when its boundary is approached. This barrier is a smooth approximation to the indicator function (3.13) and will prevent the search algorithm from leaving the interior of the feasible set. Feasibility of the iterates can be a desirable property in algorithms used for online optimization since we may stop before reaching the optimum. The approximation of (3.13) becomes more accurate as $\mu \to 0$. Therefore, the optimization problem is solved for decreasing values of μ until some convergence criterion is satisfied. Very similar to penalty methods discussed above, the problem will become ill conditioned as $\mu \to 0$. Good interior point methods take elaborate precautions to deal with this ill conditioning. An effective class of interior point algorithms is the so called primal-dual methods which solve the primal and dual problems simultaneously [20].

4 Calculating Derivatives

Most optimization algorithms used for NMPC are derivative based optimization techniques. In this section we will look at some general techniques for calculating derivatives.

In some applications one might expect the user to provide the derivatives explicitly. In other applications one can provide code to calculate the derivatives, either by approximation or exactly (up to numerical precision). If the functions involved are too complex, the user will be unable to provide explicit code for the derivatives. A number of different approaches are available, and different flavors exist. Some methods rely on the user to provide code for certain parts of the calculation. In this section we will leave NMPC and optimization for a while and look at some different approaches for computing derivatives in general. In section 5 we will look at these methods in the context of sensitivity analysis of mathematical models.

4.1 Algebraic differentiation

In this approach analytical algebraic expressions are computed by symbolic manipulation either by hand or in a computer. The functions need to be specified analytically. The problem with this technique is that the analytical expression for the derivatives of a function can grow to be very complex. Especially for complex functions and higher order derivatives.

4.2 Finite differencing

The derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at a point x is defined to be

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

and the partial derivative of a function $g : \mathbb{R}^n \to \mathbb{R}$ is defined by

$$\frac{\partial g(x_1,...,x_n)}{\partial x_i} = \lim_{h \to 0} \frac{g(x_1,...,x_i + h,...,x_n) - g(x_1,...,x_n)}{h} = \lim_{h \to 0} \frac{g(x + he_i) - g(x)}{h},$$

where e_i denotes the unit vector with all elements zero except from element i which is one. One approach that then suggests itself is to replace h by a small perturbation ϵ. Ideally we would approximate the derivative with infinite precision just by choosing ϵ small enough, but we need to keep in mind that this calculation would be implemented in a computer with final precision arithmetic. In this case, the optimal choice for ϵ will be a trade-off between arithmetic precision and error made by not choosing ϵ infinitesimally small.

By using this scheme an approximation of the partial derivative of g is given by

$$\frac{\partial g(x)}{\partial x_i} \approx \frac{g(x+\epsilon e_i) - g(x)}{\epsilon}, \tag{4.1}$$

and the gradient $\nabla_x g(x)$ can be calculated by iterating (4.1) over i. The error made by using this approximation can be estimated from Taylor's Theorem [20].

Theorem 1. *Taylor's Theorem*

Suppose that $g : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable. Then from the Fundamental Theorem of Calculus we have

$$f(x+p) = f(x) + \int_0^p \nabla_x f(x+\tau)d\tau = f(x) + \int_0^1 \nabla_x f(x+pt)pdt$$

By the mean value theorem

$$f(x+p) = f(x) + \nabla_x f(x+pt)^T p$$

for some $t \in (0,1)$. Moreover, if f is twice continuously differentiable we have that

$$\nabla_x f(x+p) = \nabla_x f(x) + \int_0^1 \nabla^2_{xx} f(x+pt)pdt$$

and that

$$f(x+p) = f(x) + \nabla_x f(x)^T p + \frac{1}{2} p^T \nabla^2_{xx} f(x+pt)p$$

for some $t \in (0,1)$.

Throughout this thesis a notation to describe growth of functions is needed.

Definition 2. $\mathcal{O}$ notation

$\mathcal{O}(g(n)) = \{f(n) : \exists\ c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0\}$.

Basically $f(n) \in \mathcal{O}(g(n))$ means that the function $f(n)$ can be bounded above by some function $cg(n)$ for c and n_0 large enough. This notation will later be used to describe the worst-case running time of an algorithm. In this case, the domains of these functions may consist of only integers.

By using Taylor's theorem

$$g(x+\epsilon e_i) = g(x) + \nabla_x g(x)^T \epsilon e_i + \frac{1}{2}\epsilon e_i^T \nabla^2_{xx} g(x+\epsilon e_i t)\epsilon e_i = g(x) + \frac{\partial g(x)}{\partial x_i}\epsilon + \frac{1}{2}\epsilon^2 \frac{\partial^2 g(x+\epsilon e_i t)}{\partial x_i^2},$$

for some $t \in (0,1)$. Suppose that M is an upper bound on $\left|\frac{\partial^2 g(x+\epsilon e_i t)}{\partial x_i^2}\right|$ for $t \in (0,1)$. Then, rearrangement yields

$$\frac{\partial g(x)}{\partial x_i} = \frac{g(x+\epsilon e_i) - g(x)}{\epsilon} + R,$$

where the error term R is bounded by $|R| \leq \frac{1}{2} M |\epsilon|$. This shows that the error made by computing derivatives by finite difference ϵ in (4.1) is of $\mathcal{O}(\epsilon)$ [20].

A more accurate result can be obtained by using two sided finite differences defined as

$$\frac{\partial g(x)}{\partial x_i} \approx \frac{g(x+\epsilon e_i) - g(x-\epsilon e_i)}{2\epsilon}.$$

It can be shown that the error by using two sided finite difference is of $\mathcal{O}(\epsilon^2)$, but this also requires $2n+1$ function evaluations while the one sided method only requires $n+1$ evaluations.

4.3 Automatic differentiation

Automatic differentiation is a method to numerically evaluate the derivative of a function. The website `www.autodiff.org` - Community Portal for Automatic Differentiation contains a lot of material and references on this topic. Automatic differentiation is a technique based on breaking the problem at hand down to a composition of elementary arithmetic operations, which any one is simple enough to be trivially differentiated by a table look up. Once the structure of this problem is lined out, the chain rule can be applied. Two different main approaches are available, namely the forward mode, and the reverse mode of automatic differentiation. The basic idea of automatic differentiation is simple. Consider the the chain rule applied to the function $f(g(x))$.

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}. \tag{4.2}$$

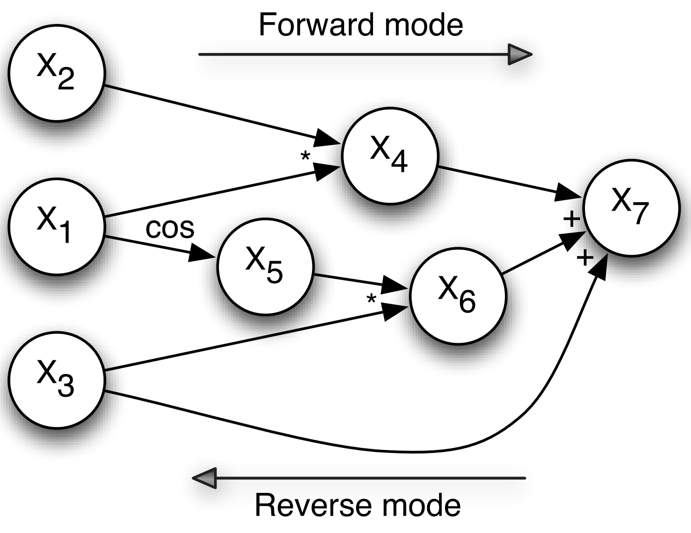
Automatic differentiation can be applied in two different ways (and combinations thereof), namely the forward mode and the reverse mode of automatic differentiation. In the simplest case one can say that the forward mode calculates (4.2) from left to right, while the reverse mode calculates (4.2) from right to left. The difference between the forward mode and the reverse mode is best illustrated by an example.

Example 3. Automatic differentiation

Consider the function $f(x_1, x_2, x_3) = x_1 x_2 + x_3 \cos(x_1) + x_3$. Suppose that we want to calculate all partial derivatives of f. First we introduce some intermediate variables to break down the problem in elementary arithmetic operations

$$\begin{aligned}
x_4 &= x_1 x_2, \\
x_5 &= \cos(x_1), \\
x_6 &= x_5 x_3, \\
x_7 &= f(x_1, x_2, x_3) = x_4 + x_6 + x_3.
\end{aligned}$$

Figure 4.1: Computational graph for automatic differentiation



The graph in figure 4.1 illustrates the problem broken down in elementary arithmetic operations. To compute the partial derivatives $\frac{\partial f}{\partial x_i} = \frac{\partial x_7}{\partial x_i}, i = \{1,2,3\}$, the forward mode of automatic differentiation traverses the graph in figure 4.1 from left to right, while the reverse mode of automatic differentiation traverses the graph from right to left. One essential part of automatic differentiation is that the derivatives of all the intermediate functions are simple enough to be predefined in a look up table of elementary derivatives.

Table 4.1: Look up table for automatic differentiation

Variables	Derivatives
$x_4 = x_1 x_2$	$\frac{\partial x_4}{\partial x_1} = x_2, \frac{\partial x_4}{\partial x_2} = x_1$
$x_5 = \cos(x_1)$	$\frac{\partial x_5}{\partial x_1} = -\sin(x_1)$
$x_6 = x_5 x_3$	$\frac{\partial x_6}{\partial x_3} = x_5, \frac{\partial x_6}{\partial x_5} = x_3$
$x_7 = x_4 + x_6 + x_3$	$\frac{\partial x_7}{\partial x_3} = \frac{\partial x_7}{\partial x_4} = \frac{\partial x_7}{\partial x_6} = 1$

We will now show how this simple example is solved with both the forward and the reverse mode and outline some of the differences between the methods. Say we want to compute $\nabla_x f(x)$ at the point $x = \begin{bmatrix} \pi & 4 & 3 \end{bmatrix}^T$.

The forward mode of automatic differentiation

The forward mode of automatic differentiation is possibly the most intuitive of the two. As mentioned above, the forward mode traverses the graph in figure 4.1 from left to right.

Start by initializing

$$\begin{aligned}\nabla_x x_1 &= e_1,\\ \nabla_x x_2 &= e_2,\\ \nabla_x x_3 &= e_3.\end{aligned}$$

Then calculate

$$\nabla_x x_4 = \frac{\partial x_4}{\partial x_1}\nabla_x x_1 + \frac{\partial x_4}{\partial x_2}\nabla_x x_2 = \left[\begin{array}{ccc} x_2 & x_1 & 0\end{array}\right]^T = \left[\begin{array}{ccc} 4 & \pi & 0\end{array}\right]^T,$$

by using information from its parent node. Now, since $\nabla_x x_4$ has been computed and node x_4 is the only child of node x_2, $\nabla_x x_2$ is not needed anymore to compute $\nabla_x f(x)$ and may be overwritten in memory. In general we can overwrite $\nabla_x x_i$ for node x_i when $\nabla_x x_j$ has been computed for all the children of node x_i. Further we compute

$$\begin{aligned}\nabla_x x_5 &= \frac{\partial x_5}{\partial x_1}\nabla_x x_1 = -\sin(x_1)e_1 = 0,\\ \nabla_x x_6 &= \frac{\partial x_6}{\partial x_3}\nabla_x x_3 + \frac{\partial x_6}{\partial x_5}\nabla_x x_5 = x_5 e_3 + x_3 0 = \cos(\pi)e_3 = -e_3,\end{aligned}$$

by traversing the graph from left to right. And we can finally compute

$$\nabla_x f(x) = \frac{\partial x_7}{\partial x_4}\nabla_x x_4 + \frac{\partial x_7}{\partial x_6}\nabla_x x_6 + \frac{\partial x_7}{\partial x_3}\nabla_x x_3 = \begin{bmatrix} 4\\ \pi \\ 0\end{bmatrix} - e_3 + e_3 = \underline{\begin{bmatrix} 4\\ \pi \\ 0\end{bmatrix}}.$$

One essential point with the forward mode is that both x_i and $\nabla_x x_i$ is evaluated during a forward sweep of the graph.

The reverse mode of automatic differentiation

The reverse mode of automatic differentiation first traverses the graph in figure 4.1 from left to right evaluating x_i for all the nodes. Then all the derivatives are calculated by a reverse sweep from right to left. The forward sweep for calculating x_i is trivial. During the reverse sweep we keep track of the value of $\frac{\partial f}{\partial x_i}$ for each node. These values are known as the adjoint variables. For node x_i we have

$$\frac{\partial f}{\partial x_i} = \sum_{x_j\, child\, of\, x_i} \frac{\partial f}{\partial x_j}\frac{\partial x_j}{\partial x_i}. \tag{4.3}$$

First we initialize $\frac{\partial f}{\partial x_7} = \frac{\partial x_7}{\partial x_7} = 1$. Then for all the parent nodes of x_7 we can use (4.3).

$$\begin{aligned}
\frac{\partial f}{\partial x_4} &= \frac{\partial f}{\partial x_7}\frac{\partial x_7}{\partial x_4} = 1, \\
\frac{\partial f}{\partial x_6} &= \frac{\partial f}{\partial x_7}\frac{\partial x_7}{\partial x_6} = 1, \\
\frac{\partial f}{\partial x_3} &= \frac{\partial f}{\partial x_7}\frac{\partial x_7}{\partial x_3} + \frac{\partial f}{\partial x_6}\frac{\partial x_6}{\partial x_3} = 1 + x_5 = 1 + \cos(\pi) = 0.
\end{aligned}$$

Once these are known we can compute

$$\begin{aligned}
\frac{\partial f}{\partial x_5} &= \frac{\partial f}{\partial x_6}\frac{\partial x_6}{\partial x_5} = x_3 = 3, \\
\frac{\partial f}{\partial x_1} &= \frac{\partial f}{\partial x_4}\frac{\partial x_4}{\partial x_1} + \frac{\partial f}{\partial x_5}\frac{\partial x_5}{\partial x_1} = x_2 - 3\sin(x_1) = 4 - 3\sin(\pi) = 4, \\
\frac{\partial f}{\partial x_2} &= \frac{\partial f}{\partial x_4}\frac{\partial x_4}{\partial x_2} = x_1 = \pi, \\
\frac{\partial f}{\partial x_3} &= \frac{\partial f}{\partial x_6}\frac{\partial x_6}{\partial x_3} + \frac{\partial f}{\partial x_7}\frac{\partial x_7}{\partial x_3} = x_5 + 1 = \cos(\pi) + 1 = 0,
\end{aligned}$$

which of course yields the same result as the forward method. Analogously as for the forward mode $\frac{\partial f}{\partial x_i}$ may be overwritten in memory when all the parents x_j of node x_i have calculated $\frac{\partial f}{\partial x_j}$.

Computational complexity

Consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$ which of the gradient is to be obtained. Calculating the gradient with the forward mode will require n forward sweeps of the graph. Calculation by the reverse mode will require m reverse sweeps of the graph. This suggests that in the general case, the forward mode is probably best when $m > n$, and the reverse mode better suited for the opposite situation.

5 Sensitivity analysis

In general terms, sensitivity analysis is the study of change in the output variables of a mathematical model with respect to perturbations of the input variables. In this section these sensitivities are calculated both by forward and adjoint methods [21]. In the sections later it is shown how the results of this section specialize to gradient calculation for the NMPC optimization problem.

Consider the general nonlinear function

$$G(x,p) = 0, x \in \mathbb{R}^n, p \in \mathbb{R}^m, G : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n. \tag{5.1}$$

Assume that $\frac{\partial G}{\partial x}$ is invertible everywhere. Then by the implicit function theorem it is possible to write $G(x,p) = 0$ as a function $x(p)$ explicitly by solving G for x. The basic idea of sensitivity analysis is to compute $\frac{dx}{dp}$. That is, compute the sensitivities of the solution variables x with respect to the input variables p around a given trajectory.

In some applications one is also concerned with a function of the state variables x and the input variables p, $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^k$. In these applications one might be interested in how perturbations of the input variables p affects the function f. That is, to compute the sensitivity

$$\frac{df}{dp} = \frac{\partial f}{\partial x}\frac{dx}{dp} + \frac{\partial f}{\partial p}. \tag{5.2}$$

As we will see later, computing the gradient of the objective function with respect to the control variables for the NMPC optimization problem will have this structure. G will correspond to the system model (equality condition), and f will correspond to the NMPC objective function.

As for automatic differentiation, there are two main approaches for obtaining these sensitivities.

5.1 The forward method

The linearization of (5.1) is

$$\frac{\partial G}{\partial x}\frac{dx}{dp} + \frac{\partial G}{\partial p} = 0, \tag{5.3}$$

where $\frac{\partial G}{\partial x} \in \mathbb{R}^{n \times n}$, $\frac{dx}{dp} \in \mathbb{R}^{n \times m}$, $\frac{\partial G}{\partial p} \in \mathbb{R}^{n \times m}$. We can solve (5.3) for $\frac{dx}{dp}$ since $\frac{\partial G}{\partial x}$ is assumed to be non-singular everywhere. To solve for $\frac{dx}{dp}$ we need to solve a number of m linear systems. The forward method is therefore efficient when the number of parameters m is small. When $\frac{dx}{dp}$ has been calculated, we can calculate $\frac{df}{dp}$ from (5.2).

5.2 The adjoint method

Say we are interested in $\frac{df}{dp}$.

From (5.3) we have that

$$\frac{dx}{dp} = -\left[\frac{\partial G}{\partial x}\right]^{-1} \frac{\partial G}{\partial p}. \tag{5.4}$$

Substituting (5.4) into (5.2) yields

$$\frac{df}{dp} = -\frac{\partial f}{\partial x}\left[\frac{\partial G}{\partial x}\right]^{-1} \frac{\partial G}{\partial p} + \frac{\partial f}{\partial p}.$$

Define

$$\lambda^T = \frac{\partial f}{\partial x}\left[\frac{\partial G}{\partial x}\right]^{-1}.$$

We then first need to solve

$$\left[\frac{\partial G}{\partial x}\right]^T \lambda = \left[\frac{\partial f}{\partial x}\right]^T, \left[\frac{\partial G}{\partial x}\right]^T \in \mathbb{R}^{n\times n}, \lambda \in \mathbb{R}^{n\times k}, \left[\frac{\partial f}{\partial x}\right]^T \in \mathbb{R}^{n\times k},$$

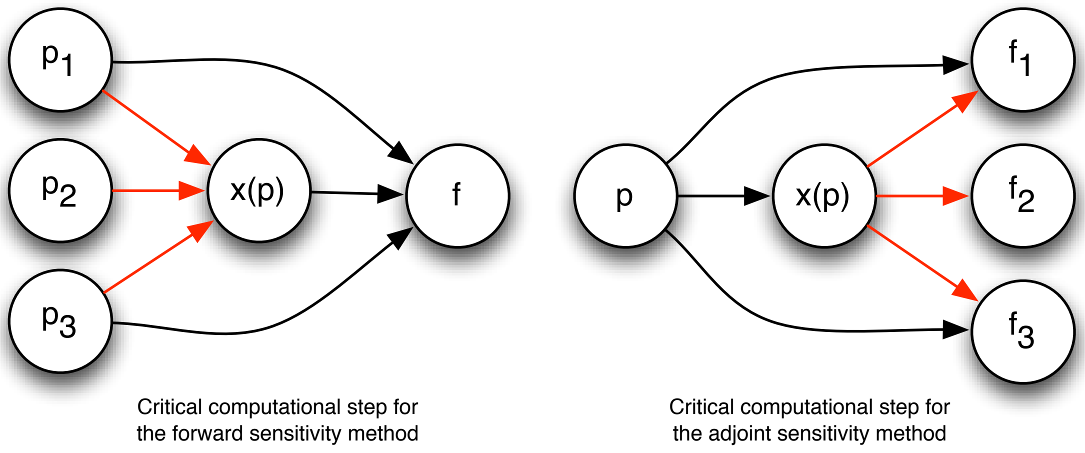
for λ, which can be done by solving a number of k linear systems. Then substitute for λ in

$$\frac{df}{dp} = -\lambda^T \frac{\partial G}{\partial p} + \frac{\partial f}{\partial p}.$$

The adjoint method is suitable for problems where the number of parameters (m) is large and the dimension of the function f (k) is small.

One important observation is that both methods require the derivatives $\frac{\partial G}{\partial p}, \frac{\partial f}{\partial p}, \frac{\partial G}{\partial x}, \frac{\partial f}{\partial x}$. These derivatives may be calculated using any of the methods for derivative calculation discussed in the previous section. That is, algebraic differentiation either by hand or by automatic symbolic manipulation, by finite differences or by the reverse or the forward mode of automatic differentiation.

Figure 5.1: Computational graphs for sensitivity analysis


p_1
p_2
p_3
x(p)
f
Critical computational step for the forward sensitivity method
p
x(p)
f_1
f_2
f_3
Critical computational step for the adjoint sensitivity method

With the derivatives $\frac{\partial G}{\partial p}, \frac{\partial f}{\partial p}, \frac{\partial G}{\partial x}, \frac{\partial f}{\partial x}$ given, the red edges in the left and the right computational graph in figure 5.1 illustrate which part of the sensitivity computation that affects the computation time by the forward and the adjoint method the most, respectively.

The leftmost graph illustrates a scenario with many parameters and only one output variable. In this case the adjoint method is preferable. The red edges illustrate the computational costly step for the forward method, which is to compute $\frac{dx}{dp}$.

For the rightmost graph, the situation is the opposite. This graph represents a problem with only one input parameter and many output variables. Calculating the sensitivities for this problem would be best solved by the forward method. The red edges illustrate the computational costly step for the adjoint method which would be to calculate λ. The reader may note that this is very similar to the complexity analysis for automatic differentiation. Roughly speaking, forward calculation is best suited for problems with few input variables and many output variables, whereas the reverse or adjoint method is better for the opposite situation.

This section is finished off with a very trivial example to see how the forward and the adjoint method compare. This following simple example offers another perspective on what is happening in the adjoint calculation. Namely that λ is chosen such that all the terms multiplying with $\frac{dx}{du}$ sum to zero. Hence, there is no need to compute $\frac{dx}{du}$, but instead λ must be found through the so called adjoint equations.

Example 4. Simple example

Given the functions

$$f(x,u) = \frac{1}{2}(x^2 + u^2), \quad g(x,u) = x - \frac{u^3}{3} = 0.$$

Say that we are interested in computing $\frac{df}{du}$.

Forward Method

To compute $\frac{df}{du}$ by the forward method we first calculate

$$\frac{dx}{du} = \frac{d}{du}\left(\frac{u^3}{3}\right) = u^2.$$

Then the chain rule is applied straight forward

$$\frac{df}{du} = \frac{\partial f}{\partial u} + \frac{\partial f}{\partial x}\frac{dx}{du} = u + x\frac{dx}{du} = u + xu^2 = u + \frac{u^5}{3}.$$

Adjoint Method

First the augmented function is introduced

$$L = f + \lambda g = \frac{1}{2}(x^2 + u^2) + \lambda(x - \frac{u^3}{3}).$$

Since $g(x, u) = 0$ we have that

$$\frac{df}{du} = \frac{dL}{du} = \frac{\partial L}{\partial u} + \frac{\partial L}{\partial x}\frac{dx}{du} + \frac{\partial L}{\partial \lambda}\frac{d\lambda}{du} = (u - \lambda u^2) + (x + \lambda)\frac{dx}{du} + (x - \frac{u^3}{3})\frac{d\lambda}{du}.$$

By letting λ satisfy $(x + \lambda) = 0$ we can avoid computing $\frac{dx}{du}$. From the equality $g(x, u) = x - \frac{u^3}{3} = 0$ we also see that there is no need to compute $\frac{d\lambda}{du}$. By inserting $\lambda = -x$ and the equality constraint $x = \frac{u^3}{3}$ we obtain

$$\frac{df}{du} = (u - \lambda u^2) = (u + xu^2) = (u + \frac{u^5}{3}).$$

6 Calculating derivative information for NMPC optimization

So far general methods for calculating derivatives have been presented. In this section these methods will be used to develop algorithms for obtaining derivatives required for NMPC optimization. First the nonlinear system model is linearized similar as in section 5.

In the next section finite differencing, the forward and the adjoint method are used to calculate objective function gradient which is required by derivative based optimization algorithms.

Next, the concern is how to efficiently compute the impulse response matrix of the linearized system model. This matrix is of interest when there are constraints on the output variables since it essentially is the constraint gradient needed by optimization algorithms like for instance SQP.

The algorithms are implemented in Matlab using suitable benchmarking examples. Theoretical bounds for runtimes of each algorithm are given and compared to actual simulation results.

Recall that it is assumed that the NMPC optimization problem is posed using a single shooting formulation where the control variables u are regarded as free variables and the state variables x and the output variables z as implicit functions of u which can be obtained by simulation.

6.1 Linearization

Let $(x^{nom}, u^{nom}, z^{nom})$ be a given nominal feasible trajectory. The dynamics around $(x^{nom}, u^{nom}, z^{nom})$ can be approximated by the linearized system model

$$\begin{aligned} \triangle x_{i+1} &= A_i \triangle x_i + B_i \triangle u_i,\ \triangle x_i = x_i - x_i^{nom},\ \triangle u_i = u_i - u_i^{nom}, && (6.1)\\ \triangle z_i &= C_i \triangle x_i + D_i \triangle u_i,\ \triangle z_i = z_i - z_i^{nom}, && (6.2)\end{aligned}$$

where

$$\begin{aligned} A_i &= \frac{\partial f}{\partial x_i}(x_i^{nom}, u_i^{nom}),\ B_i = \frac{\partial f}{\partial u_i}(x_i^{nom}, u_i^{nom}), && (6.3)\\ C_i &= \frac{\partial g}{\partial x_i}(x_i^{nom}, u_i^{nom}),\ D_i = \frac{\partial g}{\partial u_i}(x_i^{nom}, u_i^{nom}). && (6.4)\end{aligned}$$

This system will in general be linear time variant (LTV). Close to $(x^{nom}, u^{nom}, z^{nom})$ we will have that

$$\frac{\partial x_{i+1}}{\partial x_i} = A_i,\ \frac{\partial x_{i+1}}{\partial u_i} = B_i,\ \frac{\partial z_i}{\partial x_i} = C_i,\ \frac{\partial z_i}{\partial u_i} = D_i.$$

For discrete systems, these matrices are probably rather easy to obtain by using any of the methods described in section 4. That is, either symbolically, by finite differences or by the forward or reverse mode of automatic differentiation. Obtaining these by finite differences could be done straightforward by applying (4.1), but one might argue that automatic differentiation is a better approach due to exactness.

If automatic differentiation is applied, we should quantify if the forward or the reverse mode should be used to minimize the computational complexity.

For calculation of A_i the forward and the reverse mode will probably perform equally as $A_i : \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{N_x}$. This results in either N_x forward or reverse sweeps and suggests that the computational complexity for the forward and adjoint mode would be about the same.

Similarly, calculation of $B_i : \mathbb{R}^{N_u} \rightarrow \mathbb{R}^{N_x}$ can either be done by N_x reverse sweeps or N_u forward sweeps. Typically, $N_x > N_u$ and suggests that the forward mode in most cases would be preferable for calculating B_i.

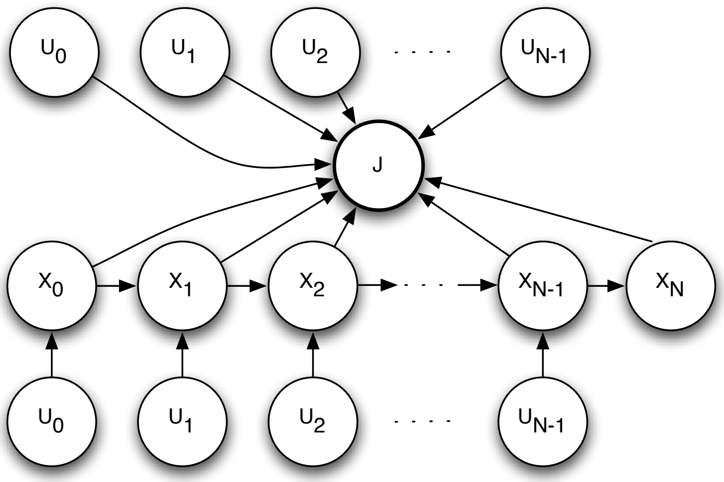
Obtaining $C_i : \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{N_z}$ can be done by N_x forward sweeps or N_z reverse sweeps. Typically $N_z < N_x$, suggesting that in this situation, the reverse mode would be best suited.

$D_i : \mathbb{R}^{N_u} \rightarrow \mathbb{R}^{N_z}$ is obtained by either N_u forward sweeps or N_z reverse sweeps. Thus, the forward mode looks better than the reverse mode for a system were $N_u < N_z$, while the opposite is true for $N_u > N_z$.

6.2 Calculation of objective function gradient

The structure of variable dependencies in the objective function is illustrated in figure 6.1. Observe that there are NN_u decision variables where N and N_u is the horizon length and the dimension of the control vector, respectively. Thus, $J : \mathbb{R}^N \times \mathbb{R}^{N_u} \rightarrow \mathbb{R}$ and by the analysis in the previous sections the adjoint method looks very appealing in terms of calculating $\nabla_u J$.

Figure 6.1: Computational graph for NMPC



In the following we analyze how $\nabla_u J$ can be computed by finite differences, the forward and the adjoint method.

Finite differences

The objective function gradient can be obtained by finite differences by in turn perturbing all the control variables and calculating the objective function value. The size of the perturbation ϵ should be decided keeping the calculation in section 4.2 in mind. That is, the optimal choice of ϵ is a trade-off between loss of precision due to final precision arithmetic and by not choosing ϵ infinitesimally small. For systems where the inputs are in different orders of magnitude, the size of the perturbation parameter should be scaled accordingly for each control variable. Choosing the right size for the perturbation parameter is highly problem dependent.

The following algorithm is based on (4.1). Each control variable is in turn perturbed, and the corresponding objective function value is calculated by simulation of the system model. In each step causality is exploited such that simulation over the entire horizon is not required for each iteration. This will result in a total of $NN_u + 1$ simulations.

Algorithm 6.1 Calculate $\nabla_u J$ - Finite differences

```
1: $x_0 = x_{init}$
2: $J_0 = 0$
3: for $k = 0$ to $N-1$ do
4:     $x_{k+1} = f(x_k, u_k)$
5:     $z_k = g(x_k, u_k)$
6:     $J_{k+1} = J_k + z_k^T Q z_k + u_k^T R u_k$
7: end for
8: $J_{N+1} = J_N + x_N^T P x_N$
9:
10: Choose perturbation value $\epsilon$
11: for $k = 0$ to $N-1$ do
12:     for $l = 1$ to $N_u$ do
13:         $\bar{J}_{kl} = J_k$
14:         $\bar{u} = u$
15:         $\bar{u}_k[l] = \bar{u}_k[l] + \epsilon e_l$
16:         $\bar{x}_k = x_k$
17:         for $n = k$ to $N-1$ do
18:             $\bar{x}_{n+1} = f(\bar{x}_n, \bar{u}_n)$
19:             $\bar{z}_n = g(\bar{x}_n, \bar{u}_n)$
20:             $\bar{J}_{kl} = \bar{J}_{kl} + \bar{z}_n^T Q \bar{z}_n + \bar{u}_n^T R \bar{u}_n$
21:         end for
22:         $\bar{J}_{kl} = \bar{J}_{kl} + \bar{x}_N^T P \bar{x}_N$
23:         $\nabla_u J[kNu + l] = (\bar{J}_{kl} - J_{N+1})/\epsilon$
24:     end for
25: end for
```

Observe that conceptually gradient calculation by finite differences is very similar to the forward method presented in the next section in the sense that both methods calculate the sensitivity from all the control variables to all the state and output variables opposed to the adjoint method that does this the other way around in a reverse manner. Also note that gradient calculation by finite differences is not exact, whereas the forward and adjoint method presented in the next sections are (up to numerical precision). This is perhaps not always so important in practice where there are lot of other uncertainties present.

In the algorithm above, it is easy to see why gradient calculation can be a challenge in real time applications. With 3 nested for-loops the computation time will grow fast with increasing time horizon length and dimension of the control vector. In an optimization algorithm like for instance SQP, these gradients are typically needed many times in each full SQP iteration.

Forward method

With A_k, B_k, C_k and D_k given, $\nabla_u J = \left[\begin{array}{ccc} \frac{\partial J}{u_0} & \cdots & \frac{\partial J}{u_{N-1}} \end{array} \right]^T$ can be computed by applying the chain rule of differentiation

$$\begin{aligned}
\frac{\partial J}{\partial u_k} =& u_k^T R + \sum_{i=0}^{N-1} \left[z_i^T Q \frac{\partial z_i}{\partial u_k} \right] + x_N^T P \frac{\partial x_N}{\partial u_k} \qquad (6.5)\\
=& u_k^T R + \sum_{i=k}^{N-1} \left[z_i^T Q \frac{\partial z_i}{\partial u_k} \right] + x_N^T P \frac{\partial x_N}{\partial u_k}\\
=& u_k^T R + z_k^T Q \frac{\partial z_k}{\partial u_k} + z_{k+1}^T Q \frac{\partial z_{k+1}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k} + z_{k+2}^T Q \frac{\partial z_{k+2}}{\partial x_{k+2}} \frac{\partial x_{k+2}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k}\\
&+ \ldots + z_{N-1}^T Q \frac{\partial z_{N-1}}{\partial x_{N-1}} \frac{\partial x_{N-1}}{\partial x_{N-2}} \ldots \frac{\partial x_{k+2}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k} + x_N^T P \frac{\partial x_N}{\partial x_{N-1}} \ldots \frac{\partial x_{k+2}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k}\\
=& u_k^T R + z_k^T Q D_k + z_{k+1}^T Q C_{k+1} B_k + z_{k+2}^T Q C_{k+2} A_{k+1} B_k\\
&+ \ldots + z_{N-1}^T Q C_{N-1} A_{N-2} \ldots A_{k+1} B_k + x_N^T P A_{N-1} \ldots A_{k+1} B_k\\
=& u_k^T R + z_k^T Q D_k + z_{k+1}^T Q C_{k+1} B_k +\\
&\sum_{i=k+2}^{N-1} \left[z_i^T Q C_i A_{i-1} \ldots A_{k+1} B_k \right] + x_N^T P A_{N-1} \ldots A_{k+1} B_k .
\end{aligned}$$

The change of lower summation index in the second equality is due to causality. Observe that calculating $\nabla_u J$ will involve a lot of matrix multiplications since the sum in (6.5) must be computed for every $\frac{\partial J}{\partial u_k}$ and can therefore be very expensive. In the algorithm below, ϕ and Ψ are placeholders that exploit structure and avoid doing a lot of the same matrix multiplications more than one time.

Algorithm 6.2 Calculate $\nabla_u J$ - Forward method

```
1: $x_0 = x_{init}$
2: for $k = 0$ to $N-1$ do
3:     $x_{k+1} = f(x_k, u_k)$
4:     $z_k = g(x_k, u_k)$
5: end for
6:
7: for $k = 0$ to $N-2$ do
8:     $\phi = I$
9:     $\psi = 0$
10:    for $i = k+2$ to $N-1$ do
11:        $\phi = A_{i-1}\phi$
12:        $\psi = \psi + z_i^T Q C_i \phi$
13:    end for
14:    $\nabla_u J[kN_u + 1 : (k+1)N_u] = Ru_k + D_k^T Q z_k + B_k^T C_{k+1}^T Q z_{k+1} + B_k^T \psi^T + B_k^T \phi^T A_{N-1}^T P x_N$
15: end for
16: $\nabla_u J[(N-1)N_u + 1 : NN_u] = Ru_{N-1} + D_{N-1}^T Q z_{N-1} + B_{N-1}^T P x_N$
```

Adjoint method

From the discussion earlier, since $J : \mathbb{R}^N \times \mathbb{R}^{N_u} \to \mathbb{R}$, the adjoint method seems very appealing for calculating the objective function gradient. In this section we will follow the same idea as in example 4 in section 5, namely to choose the adjoint variables λ such that all the terms multiplying with $\frac{dx_i}{du}$ sum to zero. Since these terms sum to zero, there is no need to compute $\frac{dx_i}{du}$, $i \in \{1, ..., N-1\}$ which essentially is the full impulse response matrix of the linearized system model.

Define the Lagrangian function

$$\begin{aligned}\mathcal{L} =& J - \sum_{i=0}^{N-1} \left[\lambda_{i+1}^T (x_{i+1} - f(x_i, u_i))\right] \\ =& \sum_{i=0}^{N-1} \left[\frac{1}{2}(z_i^T Q z_i + u_i^T R u_i)\right] + \frac{1}{2} x_N^T P x_N - \sum_{i=0}^{N-1} \left[\lambda_{i+1}^T (x_{i+1} - f(x_i, u_i))\right].\end{aligned}$$

Since $x_{k+1} = f(x_k, u_k)$, we have that $\frac{dJ}{du} = \frac{d\mathcal{L}}{du}$. By the chain rule of differentiation.

$$\begin{aligned}\frac{dJ}{du} =& \sum_{i=0}^{N-1} \frac{\partial \mathcal{L}}{\partial z_i} \left(\frac{\partial z_i}{\partial x_i} \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du}\right] + \frac{\partial z_i}{\partial u_i} \frac{du_i}{du} \right) + \sum_{i=0}^{N-1} \frac{\partial \mathcal{L}}{\partial u_i} \frac{du_i}{du} + \\ & \frac{\partial \mathcal{L}}{\partial x_N} \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du}\right] + \sum_{i=0}^{N-1} \frac{\partial \mathcal{L}}{\partial x_{i+1}} \left[\sum_{k=0}^{N-1} \frac{\partial x_{i+1}}{\partial u_k} \frac{du_k}{du}\right] + \\ & \sum_{i=0}^{N-1} \frac{\partial \mathcal{L}}{\partial x_i} \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du}\right] + \sum_{i=0}^{N-1} \frac{\partial \mathcal{L}}{\partial \lambda_{i+1}} \left[\sum_{k=0}^{N-1} \frac{\partial \lambda_{i+1}}{\partial u_k} \frac{du_k}{du}\right] \\ =& \sum_{i=0}^{N-1} z_i^T Q \left(\frac{\partial z_i}{\partial x_i} \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du}\right] + \frac{\partial z_i}{\partial u_i} \frac{du_i}{du} \right) + \sum_{i=0}^{N-1} \left(u_i^T R + \lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial u_i} \right) \frac{du_i}{du} + \\ & x_N^T P \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du}\right] - \sum_{i=0}^{N-1} \lambda_{i+1}^T \left[\sum_{k=0}^{N-1} \frac{\partial x_{i+1}}{\partial u_k} \frac{du_k}{du}\right] + \\ & \sum_{i=0}^{N-1} \left(\lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial x_i} \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du}\right] + \sum_{i=0}^{N-1} (x_{i+1} - f(x_i, u_i))^T \left[\sum_{k=0}^{N-1} \frac{\partial \lambda_{i+1}}{\partial u_k} \frac{du_k}{du}\right].\end{aligned}$$

Inspired by [6], where continuous systems are studied and integration of parts is used, we instead define the following identity for the discrete case, which is easy to verify just by inspection

$$\sum_{i=0}^{N-1} \lambda_{i+1}^T \left[\sum_{k=0}^{N-1} \frac{\partial x_{i+1}}{\partial u_k} \frac{du_k}{du}\right] = \sum_{i=0}^{N-1} \lambda_i^T \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du}\right] + \lambda_N^T \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du}\right] - \lambda_0^T \left[\sum_{k=0}^{N-1} \frac{\partial x_0}{\partial u_k} \frac{du_k}{du}\right].$$

By inserting this identity and noting that $\sum_{i=0}^{N-1}(x_{i+1} - f(x_i, u_i)^T \left[\sum_{k=0}^{N-1} \frac{\partial \lambda_{i+1}}{\partial u_k} \frac{du_k}{du}\right] = 0$ in order to satisfy the nonlinear model, we obtain

$$\begin{aligned} \frac{dJ}{du} &= \sum_{i=0}^{N-1} z_i^T Q \left(\frac{\partial z_i}{\partial x_i} \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du} \right] + \frac{\partial z_i}{\partial u_i} \frac{du_i}{du} \right) + \sum_{i=0}^{N-1} \left(u_i^T R + \lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial u_i} \right) \frac{du_i}{du} + \\ & \quad x_N^T P \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du} \right] - \sum_{i=0}^{N-1} \lambda_i^T \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du} \right] - \lambda_N^T \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du} \right] + \\ & \quad \lambda_0^T \left[\sum_{k=0}^{N-1} \frac{\partial x_0}{\partial u_k} \frac{du_k}{du} \right] + \sum_{i=0}^{N-1} \left(\lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial x_i} \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du} \right]. \end{aligned}$$

Rearrangement yields

$$\begin{aligned} \frac{dJ}{du} &= \sum_{i=0}^{N-1} \left(z_i^T Q \frac{\partial z_i}{\partial x_i} - \lambda_i^T + \lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial x_i} \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du} \right] + \left(x_N^T P - \lambda_N^T \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du} \right] \\ & \quad + \sum_{i=0}^{N-1} \left(z_i^T Q \frac{\partial z_i}{\partial u_i} + u_i^T R + \lambda_{i+1}^T \frac{\partial f(x_i, u_i)}{\partial u_i} \right) \frac{du_i}{du} + \lambda_0^T \left[\sum_{k=0}^{N-1} \frac{\partial x_0}{\partial u_k} \frac{du_k}{du} \right]. \end{aligned}$$

Note that $\lambda_0^T \left[\sum_{k=0}^{N-1} \frac{\partial x_0}{\partial u_k}\right] = 0$ since the initial state is constant not dependent on u. Further, we can utilize that the dynamics around a nominal trajectory are governed by the linearized system model (6.1)-(6.4). Substitution of A_i, B_i, C_i, D_i for $\frac{\partial f(x_i,u_i)}{\partial x_i}$, $\frac{\partial f(x_i,u_i)}{\partial u_i}$, $\frac{\partial z_i}{\partial x_i}$, $\frac{\partial z_i}{\partial u_i}$ yields

$$\begin{aligned} \frac{dJ}{du} &= \sum_{i=0}^{N-1} \left(z_i^T Q C_i - \lambda_i^T + \lambda_{i+1}^T A_i \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_i}{\partial u_k} \frac{du_k}{du} \right] + \left(x_N^T P - \lambda_N^T \right) \left[\sum_{k=0}^{N-1} \frac{\partial x_N}{\partial u_k} \frac{du_k}{du} \right] \\ & \quad + \sum_{i=0}^{N-1} \left(z_i^T Q D_i + u_i^T R + \lambda_{i+1}^T B_i \right) \frac{du_i}{du}. \end{aligned}$$

Now, by letting λ_k fulfill

$$z_k^T Q C_k - \lambda_k^T + \lambda_{k+1}^T A_k = 0 \tag{6.6}$$

with the final condition

$$\lambda_N = P x_N, \tag{6.7}$$

we have that

$$\frac{dJ}{du} = \sum_{i=0}^{N-1} \left(z_i^T Q D_i + u_i^T R + \lambda_{i+1}^T B_i \right) \frac{du_i}{du}. \tag{6.8}$$

By definition

$$\frac{dJ}{du} = \sum_{i=k}^{N-1} \frac{\partial J}{\partial u_i} \frac{du_i}{du}. \tag{6.9}$$

Hence, by comparison of (6.8) and (6.9)

$$\frac{\partial J}{\partial u_k} = z_k^T Q D_k + u_k^T R + \lambda_{k+1}^T B_k. \tag{6.10}$$

Sensitivity computation by the adjoint method can therefore be done by only two simulations. One forward simulation to obtain z and x from u. Then initialize (6.7) and iterate (6.6) and (6.10) in reverse time. One unavoidable drawback is that the variables $x_k,\ k \in \{1, ..., N\}$ need to be stored during the forward solve as they are needed to obtain z_k and (A_k, B_k, C_k, D_k) in (6.6) and (6.10) during the reverse solve. For discrete time systems this may not be an issue as long as internal memory is not a limitation, but for continuous time systems this becomes more cumbersome. Especially, if a variable step-size solver is used it is not clear in which time instants to store the solutions as the step-sizes used during the reverse solve will probably not coincide with the stored trajectory from the forward solve. In this case, interpolation techniques can be used to approximate the solution at the desired time instants. For simplicity one might instead consider freezing adaptive parameters in the solver like order control and step size.

Algorithm 6.3 Calculate $\nabla_u J$ - Adjoint method

```
1:  $x_0 = x_{init}$
2:  for $k = 0$ to $N - 1$ do
3:      $x_{k+1} = f(x_k, u_k)$
4:      $z_k = g(x_k, u_k)$
5:  end for
6:
7:  $\lambda = P x_N$
8:  for $k = N - 1$ to $0$ do
9:      Obtain LTV model $(A_k, B_k, C_k, D_k)$
10:     $\nabla_u J[kN_u + 1 : (k + 1)N_u] = D_k^T Q z_k + R u_k + B_k^T \lambda$
11:     $\lambda = C_k^T Q z_k + A_k^T \lambda$
12: end for
```

Note that the time index k in for the adjoint variables is omitted, since only the value in the previous iteration is needed.

6.3 Calculation of impulse response matrix

In this section we will look at different methods for computing the impulse response matrix of the linearized system model. This matrix is of interest when there are constraints on the output variables since it essentially is the constraint gradient needed by optimization algorithms like for instance SQP. Since these constraints typically are enforced at every sample instant on the horizon they are also referred to as path constraints.

We discuss different approaches for computing this matrix and develop some algorithms. The discussion includes the forward and the adjoint method and different model structures.

The algorithms developed are presented in pseudo code and also illustrated iteration by iteration where appropriate. These algorithms are later implemented with suitable test examples to compare efficiency and highlight some of the theoretical considerations made earlier.

The impulse response matrix Ξ of the linearized system model (6.1)-(6.2) is given by

$$\begin{pmatrix} \triangle z_0 \\ \triangle z_1 \\ \triangle z_2 \\ \vdots \\ \triangle z_{N-1} \end{pmatrix} = \underbrace{\begin{pmatrix} D_0 & 0 & 0 & \dots & 0 \\ C_1 B_0 & D_1 & 0 & \ddots & \vdots \\ C_2 A_1 B_0 & C_2 B_1 & D_2 & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ C_{N-1}A_{N-2}\dots A_1 B_0 & C_{N-1}A_{N-2}\dots A_2 B_1 & C_{N-1}A_{N-2}\dots A_3 B_2 & \dots & D_{N-1} \end{pmatrix}}_{\Xi} \begin{pmatrix} \triangle u_0 \\ \triangle u_1 \\ \triangle u_2 \\ \vdots \\ \triangle u_{N-1} \end{pmatrix}. \tag{6.11}$$

Close to $(x^{nom}, u^{nom}, z^{nom})$ the dynamics of the nonlinear system will be governed by linearized system model. Hence,

$$\frac{\partial x_{i+1}}{\partial x_i} = A_i,\ \frac{\partial x_{i+1}}{\partial u_i} = B_i,\ \frac{\partial z_i}{\partial x_i} = C_i,\ \frac{\partial z_i}{\partial u_i} = D_i.$$

By the chain rule of differentiation

$$\Xi = \begin{pmatrix} \frac{\partial z_0}{\partial u_0} & 0 & 0 & \dots & 0 \\ \frac{\partial z_1}{\partial u_0} & \frac{\partial z_1}{\partial u_1} & 0 & \ddots & \vdots \\ \frac{\partial z_2}{\partial u_0} & \frac{\partial z_2}{\partial u_1} & \frac{\partial z_2}{\partial u_2} & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \frac{\partial z_{N-1}}{\partial u_0} & \frac{\partial z_{N-1}}{\partial u_1} & \frac{\partial z_{N-1}}{\partial u_2} & \dots & \frac{\partial z_{N-1}}{\partial u_{N-1}} \end{pmatrix}. \tag{6.12}$$

Calculating this matrix can be very expensive since it requires a lot of system simulations. In the following we will look at different methods for computing the impulse response matrix and we analyze how the forward and adjoint method can aid different model structures and properties.

Finite differences

The following algorithm will calculate the impulse response matrix Ξ using finite differences. This algorithm is quite simple and easy to implement since it does not require the linearized system model. A total of $NN_u + 1$ simulations of the system model are needed. Note that this algorithm conceptually is very similar to calculating the objective function gradient by finite differences. Again, the size of the perturbation parameter should be chosen with care.

Note that calculation of derivative information by finite differences can also be applied to a continuous time model. The only requirement is that the algorithm must be able to invoke an ODE solver routine to obtain solutions for each of the perturbed control variables. This also applies to algorithm 6.1.

Algorithm 6.4 Calculate impulse response matrix Ξ - Finite differences

```
1: $x_0 = x_{init}$
2: for $k = 0$ to $N - 1$ do
3:     $x_{k+1} = f(x_k, u_k)$
4:     $z_k = g(x_k, u_k)$
5: end for
6:
7: Choose perturbation value $\epsilon$
8: for $k = 1$ to $N$ do
9:     for $l = 1$ to $N_u$ do
10:        $\bar{u} = u$
11:        $\bar{u}_{k-1}[l] = \bar{u}_{k-1}[l] + \epsilon$
12:        $\bar{x}_{k-1} = x_{k-1}$
13:        for $n = k$ to $N$ do
14:            $\bar{z}_{n-1} = g(\bar{x}_{n-1}, \bar{u}_{n-1})$
15:            $\Xi[(n-1)N_z + 1 : nN_z, (k-1)N_z + l] = (\bar{z}_{n-1} - z_{n-1})/\epsilon$
16:            $\bar{x}_n = f(\bar{x}_{n-1}, \bar{u}_{n-1})$
17:        end for
18:    end for
19: end for
```

Forward method

By the chain rule of differentiation

$$\begin{aligned} \frac{\partial z_k}{\partial u_k} &= D_k, \\ \frac{\partial z_{k+1}}{\partial u_k} &= \frac{\partial z_{k+1}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k} = C_{k+1} B_k, \\ \frac{\partial z_j}{\partial u_k} &= \frac{\partial z_j}{\partial x_j} \frac{\partial x_j}{\partial x_{j-1}} \frac{\partial x_{j-1}}{\partial x_{j-2}} \cdots \frac{\partial x_{k+2}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k} = C_j A_{j-1} A_{j-2} \ldots A_{k+1} B_k \ \forall j > k+1. \end{aligned} \tag{6.13}$$

When iterating (6.13) over j to obtain all the elements in on column of Ξ in 6.12 we should exploit that the sequence of matrices $A_{j-1}A_{j-2}\ldots A_{k+1}$ only differ by one additional matrix multiplication from left between each iteration. We can make use of this structure by calculating Ξ column-wise. When computing one column of Ξ, we calculate the sensitivities from the control variables at one time instant to the output variables at all time instants on the horizon. We will therefore refer to this as the forward method for computing Ξ. Most of the matrix multiplications carried out here will be the same as those performed by algorithm 6.2 which calculates the objective function gradient by the forward method.

The role of the placeholder $\phi \in \mathbb{R}^{N_x \times (N-1)N_u}$ is to exploit the aforementioned column-wise structure.

Algorithm 6.5 Calculate impulse response matrix Ξ - Forward method

```
1:  $x_0 = x_{init}$
2:  for $k = 0$ to $N - 1$ do
3:      $x_{k+1} = f(x_k, u_k)$
4:      $z_k = g(x_k, u_k)$
5:  end for
6:
7:  Obtain $D_0$
8:  $\Xi[1 : N_z, 1 : N_u] = D_0$
9:  for $j = 2$ to $N$ do
10:     Obtain $(A_{j-2}, B_{j-2}, C_{j-1}, D_{j-1})$
11:     for $i = 1$ to $j - 2$ do
12:         $\phi[1 : N_x, (i-1)N_u + 1 : iN_u] = A_{j-2}\phi[1 : N_x, (i-1)N_u + 1 : iN_u]$
13:     end for
14:     $\phi[1 : N_x, (j-2)N_u + 1 : (j-1)N_u] = B_{j-2}$
15:     for $i = 1$ to $j - 1$ do
16:         $\Xi[(j-1)N_z + 1 : jN_z, (i-1)N_u + 1 : iN_u] = C_{j-1}\phi[1 : N_x, (i-1)N_u + 1 : iN_u]$
17:     end for
18:     $\Xi[(j-1)N_z + 1 : jN_z, (j-1)N_u + 1 : jN_u] = D_{j-1}$
19: end for
```

Table 6.1 illustrates the progression of algorithm 6.5. The same structure could have been exploited by calculating each of the columns in Ξ one at the time. By instead calculating all the columns in parallel we have the advantage that $(A_{j-2}, B_{j-2}, C_{j-1}, D_{j-1})$ are needed in iteration j which will allow these matrices to be inserted 'on the-fly' if they are obtained from integration of sensitivity equations.

Table 6.1: Illustration of steps in algorithm 6.5

	Code	ϕ
		$\begin{pmatrix} 0 & 0 & 0 & \dots \end{pmatrix}$
$j=2$	$\phi(1,1) = B_0$	$\begin{pmatrix} B_0 & 0 & 0 & \dots \end{pmatrix}$
$j=3$	$\begin{gathered}\phi(1,1) = A_1\phi(1,1) \\ \phi(1,2) = B_1\end{gathered}$	$\begin{pmatrix} A_1B_0 & B_1 & 0 & \dots \end{pmatrix}$
$j=4$	$\begin{gathered}\phi(1,1) = A_2\phi(1,1) \\ \phi(1,2) = A_2\phi(1,2) \\ \phi(1,3) = B_2\end{gathered}$	$\begin{pmatrix} A_2A_1B_0 & A_2B_1 & B_2 & \dots \end{pmatrix}$
$\vdots$	$\vdots$	$\vdots$
	Code	Ξ
	$\Xi(1,1) = D_0$	$\begin{pmatrix} D_0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$
$j=2$	$\begin{gathered}\Xi(2,1) = C_1\phi(1,1) \\ \Xi(2,2) = D_1\end{gathered}$	$\begin{pmatrix} D_0 & 0 & 0 & 0 & \dots \\ C_1B_0 & D_1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$
$j=3$	$\begin{gathered}\Xi(3,1) = C_2\phi(1,1) \\ \Xi(3,2) = C_2\phi(1,2) \\ \Xi(3,3) = D_2\end{gathered}$	$\begin{pmatrix} D_0 & 0 & 0 & 0 & \dots \\ C_1B_0 & D_1 & 0 & 0 & \dots \\ C_2A_1B_0 & C_2B_1 & D_2 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$
$j=4$	$\begin{gathered}\Xi(4,1) = C_3\phi(1,1) \\ \Xi(4,2) = C_3\phi(1,2) \\ \Xi(4,3) = C_3\phi(1,3) \\ \Xi(4,4) = D_3\end{gathered}$	$\begin{pmatrix} D_0 & 0 & 0 & 0 & \dots \\ C_1B_0 & D_1 & 0 & 0 & \dots \\ C_2A_1B_0 & C_2B_1 & D_2 & 0 & \dots \\ C_3A_2A_1B_0 & C_3A_2B_1 & C_3B_2 & D_3 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$
$\vdots$	$\vdots$	$\vdots$

Adjoint method

In this section it will be shown how the impulse response matrix can be calculated using adjoints. In the forward method, we exploited that adjacent elements in each column only differ by one matrix multiplication from left. It will turn out that the following analysis correspond to instead exploiting that adjacent elements in each row only differ by one matrix multiplication from right. An algorithm could have been deduced just from this fact, but it is still interesting to perform the analysis below which derives the adjoint equations.

Define the function

$$\mathcal{L}_j = z_j - \sum_{i=0}^{j} \left[\lambda_{i+1}^T (x_{i+1} - f(x_i, u_i))\right].$$

Note that $\lambda_k \in \mathbb{R}^{N_x \times N_z}$. Since $x_{k+1} = f(x_k, u_k)$, we have that $\frac{dz_j}{du} = \frac{d\mathcal{L}_j}{du}$. By the chain rule of differentiation

$$\begin{aligned}\frac{dz_j}{du} = \quad & \frac{\partial \mathcal{L}_j}{\partial z_j}\left(\frac{\partial z_j}{\partial x_j}\left[\sum_{k=0}^{j}\frac{\partial x_j}{\partial u_k}\frac{du_k}{du}\right] + \frac{\partial z_j}{\partial u_j}\frac{du_j}{du}\right) + \sum_{i=0}^{j}\frac{\partial \mathcal{L}_j}{\partial u_i}\frac{du_i}{du} + \\ & \sum_{i=0}^{j}\frac{\partial \mathcal{L}_j}{\partial x_{i+1}}\left[\sum_{k=0}^{j}\frac{\partial x_{i+1}}{\partial u_k}\frac{du_k}{du}\right] + \sum_{i=0}^{j}\frac{\partial \mathcal{L}_j}{\partial x_i}\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] + \sum_{i=0}^{j}\frac{\partial \mathcal{L}_j}{\partial \lambda_{i+1}}\left[\sum_{k=0}^{j}\frac{\partial \lambda_{i+1}}{\partial u_k}\frac{du_k}{du}\right] \\ = \quad & I\left(\frac{\partial z_j}{\partial x_j}\left[\sum_{k=0}^{j}\frac{\partial x_j}{\partial u_k}\frac{du_k}{du}\right] + \frac{\partial z_j}{\partial u_j}\frac{du_j}{du}\right) + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial u_i}\frac{du_i}{du}\right) - \\ & \sum_{i=0}^{j}\lambda_{i+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{i+1}}{\partial u_k}\frac{du_k}{du}\right] + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial x_i}\right)\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] + \\ & \sum_{i=0}^{j}(x_{i+1} - f(x_i,u_i))^T\left[\sum_{k=0}^{j}\frac{\partial \lambda_{i+1}}{\partial u_k}\frac{du_k}{du}\right].\end{aligned}$$

Inspired by [6], where continuous systems are studied and integration of parts is used, we instead define the following identity for the discrete case, which is easy to verify by inspection

$$\sum_{i=0}^{j}\lambda_{i+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{i+1}}{\partial u_k}\frac{du_k}{du}\right] = \sum_{i=0}^{j}\lambda_i^T\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] + \lambda_{j+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{j+1}}{\partial u_k}\frac{du_k}{du}\right] - \lambda_0^T\left[\sum_{k=0}^{j}\frac{\partial x_0}{\partial u_k}\frac{du_k}{du}\right].$$

By inserting this identity and noting that $\sum_{i=0}^{j}(x_{i+1} - f(x_i,u_i))^T\left[\sum_{k=0}^{j}\frac{\partial \lambda_{i+1}}{\partial u_k}\frac{du_k}{du}\right] = 0$, we obtain

$$\begin{aligned}\frac{dz_j}{du} \quad = \quad & \left(\frac{\partial z_j}{\partial x_j}\left[\sum_{k=0}^{j}\frac{\partial x_j}{\partial u_k}\frac{du_k}{du}\right] + \frac{\partial z_j}{\partial u_j}\frac{du_j}{du}\right) + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial u_i}\frac{du_i}{du}\right) - \\ & \sum_{i=0}^{j}\lambda_i^T\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] - \lambda_{j+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{j+1}}{\partial u_k}\frac{du_k}{du}\right] + \lambda_0^T\left[\sum_{k=0}^{j}\frac{\partial x_0}{\partial u_k}\frac{du_k}{du}\right] \\ & + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial x_i}\right)\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right].\end{aligned}$$

Rearrangement yields

$$\begin{aligned}\frac{dz_j}{du} = \quad & \frac{\partial z_j}{\partial x_j}\left[\sum_{k=0}^{j}\frac{\partial x_j}{\partial u_k}\frac{du_k}{du}\right] + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial x_i} - \lambda_i^T\right)\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] - \\ & \lambda_{j+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{j+1}}{\partial u_k}\frac{du_k}{du}\right] + \lambda_0^T\left[\sum_{k=0}^{j}\frac{\partial x_0}{\partial u_k}\frac{du_k}{du}\right] + \frac{\partial z_j}{\partial u_j}\frac{du_j}{du} + \sum_{i=0}^{j}\left(\lambda_{i+1}^T\frac{\partial f(x_i,u_i)}{\partial u_i}\frac{du_i}{du}\right).\end{aligned}$$

Note that $\lambda_0^T \left[\sum_{k=0}^{j} \frac{\partial x_0}{\partial u_k}\frac{du_k}{du}\right] = 0$ since the initial state is constant and not dependent on u. Further, we can utilize that the dynamics around a nominal trajectory are governed by the linearized system model. Substitution of A_i, B_i, C_i, D_i for $\frac{\partial f(x_i,u_i)}{\partial x_i}, \frac{\partial f(x_i,u_i)}{\partial u_i}, \frac{\partial z_i}{\partial x_i}, \frac{\partial z_i}{\partial u_i}$ yields

$$\begin{aligned}\frac{dz_j}{du} = \quad & C_j\left[\sum_{k=0}^{j}\frac{\partial x_j}{\partial u_k}\frac{du_k}{du}\right] + \sum_{i=0}^{j}\left(\lambda_{i+1}^T A_i - \lambda_i^T\right)\left[\sum_{k=0}^{j}\frac{\partial x_i}{\partial u_k}\frac{du_k}{du}\right] - \\ & \lambda_{j+1}^T\left[\sum_{k=0}^{j}\frac{\partial x_{j+1}}{\partial u_k}\frac{du_k}{du}\right] + D_j\frac{du_j}{du} + \sum_{i=0}^{j}\left(\lambda_{i+1}^T B_i\frac{du_i}{du}\right). \qquad (6.14)\end{aligned}$$

Now, let λ_k fulfill

$$\begin{aligned}\lambda_{j+1}^T &= 0, \qquad (6.15)\\ \lambda_j^T &= \lambda_{j+1}^T A_j + C_j = C_j,\\ \lambda_k^T &= \lambda_{k+1}^T A_k \ \forall k < j.\end{aligned}$$

Then (6.14) reduces to

$$\begin{aligned}\frac{dz_j}{du} &= D_j\frac{du_j}{du} + \sum_{i=0}^{j}\left(\lambda_{i+1}^T B_i\frac{du_i}{du}\right) \qquad (6.16)\\ &= \left(D_j + \lambda_{j+1}^T B_j\right)\frac{du_j}{du} + \sum_{i=0}^{j-1}\left(\lambda_{i+1}^T B_i\frac{du_i}{du}\right). \qquad (6.17)\end{aligned}$$

By definition the total derivative is given by

$$\frac{dz_j}{du} = \sum_{i=0}^{j}\frac{\partial z_j}{\partial u_i}\frac{du_i}{du}. \qquad (6.18)$$

Comparison of (6.16) and (6.18) yields

$$\begin{aligned}\frac{\partial z_j}{\partial u_j} &= D_j + \lambda_{j+1}^T B_j = D_j, \qquad (6.19)\\ \frac{\partial z_j}{\partial u_k} &= \lambda_{k+1}^T B_k \ \forall k < j.\end{aligned}$$

Given a nominal control trajectory u, x and z can be obtained by simulating the system forward in time. Then λ_k can be found by marching (6.15) in reverse time. Once these adjoint variables are calculated, the gradient of interest can be obtained from (6.19).

Observe that calculating $\frac{dz_j}{du} = \sum_{i=0}^{j} \frac{\partial z_j}{\partial u_i}\frac{du_i}{du}$ involves all the elements in row $j+1$ in the impulse response matrix in (6.12). Therefore, it is now clear that the adjoint method exploits structure of the impulse response matrix in a row-wise manner. The following algorithm calculates the full impulse response matrix by this principle. Similar as for the forward method, all the rows are calculated i parallel.

Algorithm 6.6 Calculate impulse response matrix Ξ - Adjoint method

```
1: $x_0 = x_{init}$
2: for $k = 0$ to $N-1$ do
3:    $x_{k+1} = f(x_k, u_k)$
4:    $z_k = g(x_k, u_k)$
5: end for
6:
7: Obtain $D_{N-1}$
8: $\Xi[(N-1)N_z+1 : NN_z, (N-1)N_u+1 : NN_u] = D_{N-1}$
9: for $k = N-1$ to $1$ do
10:    Obtain $(A_k, B_{k-1}, C_k, D_{k-1})$
11:    $\lambda[(k-1)N_z+1 : kN_z, 1 : N_x] = C_k$
12:    for $i = k+1$ to $N-1$ do
13:       $\lambda[(i-1)N_z+1 : iN_z, 1 : N_x] = \lambda[(i-1)N_z+1 : iN_z, 1 : N_x]A_k$
14:    end for
15:    $\Xi[(k-1)N_z+1 : kN_z, (k-1)N_u+1 : kN_u] = D_{k-1}$
16:    for $i = k$ to $N-1$ do
17:       $\Xi[iN_z+1 : (i+1)N_z, (k-1)N_u+1 : kN_u] = \lambda[(i-1)N_z+1 : iN_z, 1 : N_x]B_{k-1}$
18:    end for
19: end for
```

Table 6.2 illustrates the progression of algorithm 6.6 and shows how the impulse response matrix is computed in reverse time. By inspecting every iteration it is also easy to see why the adjoint variable $\lambda \in \mathbb{R}^{(N-1)N_z \times N_x}$ in algorithm 6.6 plays a similar role as ϕ in the forward method. Moreover, the reader should observe how λ in table 6.2 connects with (6.19).

Table 6.2: Illustration of steps in algorithm 6.6

	Code	λ
		$\begin{pmatrix} \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix}$
$k = N-1$	$\lambda(N-1,1) = C_{N-1}$	$\begin{pmatrix} \vdots \\ 0 \\ 0 \\ C_{N-1} \end{pmatrix}$
$k = N-2$	$\lambda(N-2,1) = C_{N-2}$ $\lambda(N-1,1) = \lambda(N-1,1)A_{N-2}$	$\begin{pmatrix} \vdots \\ 0 \\ C_{N-2} \\ C_{N-1}A_{N-2} \end{pmatrix}$
$k = N-3$	$\lambda(N-3,1) = C_{N-3}$ $\lambda(N-2,1) = \lambda(N-2,1)A_{N-3}$ $\lambda(N-1,1) = \lambda(N-1,1)A_{N-3}$	$\begin{pmatrix} \vdots \\ C_{N-3} \\ C_{N-2}A_{N-3} \\ C_{N-1}A_{N-2}A_{N-3} \end{pmatrix}$
$\vdots$	$\vdots$	$\vdots$
	Code	Ξ
	$\Xi(N,N) = D_{N-1}$	$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & 0 & 0 & D_{N-1} \end{pmatrix}$
$k = N-1$	$\Xi(N-1,N-1) = D_{N-2}$ $\Xi(N,N-1) = \lambda(N-1,1)B_{N-2}$	$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & 0 & D_{N-2} & 0 \\ \cdots & 0 & 0 & C_{N-1}B_{N-2} & D_{N-1} \end{pmatrix}$
$k = N-2$	$\Xi(N-2,N-2) = D_{N-3}$ $\Xi(N-1,N-2) = \lambda(N-2,1)B_{N-3}$ $\Xi(N,N-2) = \lambda(N-1,1)B_{N-3}$	$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots \\ \cdots & 0 & 0 & 0 & 0 \\ \cdots & 0 & D_{N-3} & 0 & 0 \\ \cdots & 0 & C_{N-2}B_{N-3} & D_{N-2} & 0 \\ \cdots & 0 & C_{N-1}A_{N-2}B_{N-3} & C_{N-1}B_{N-2} & D_{N-1} \end{pmatrix}$
$k = N-3$	$\Xi(N-3,N-3) = D_{N-4}$ $\Xi(N-2,N-3) = \lambda(N-3,1)B_{N-4}$ $\Xi(N-1,N-3) = \lambda(N-2,1)B_{N-4}$ $\Xi(N,N-3) = \lambda(N-1)B_{N-4}$	$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots \\ \cdots & D_{N-4} & 0 & 0 & 0 \\ \cdots & C_{N-3}B_{N-4} & D_{N-3} & 0 & 0 \\ \cdots & C_{N-2}A_{N-3}B_{N-4} & C_{N-2}B_{N-3} & D_{N-2} & 0 \\ \cdots & C_{N-1}A_{N-2}A_{N-3}B_{N-4} & C_{N-1}A_{N-2}B_{N-3} & C_{N-1}B_{N-2} & D_{N-1} \end{pmatrix}$
$\vdots$	$\vdots$	$\vdots$

The forward and the adjoint method may not be entirely equivalent in terms of efficiency when employing a SQP type algorithm or any other optimization technique where constraint gradients are needed. The adjoint method may benefit from not enforcing constraints on the output variables at all the points on the horizon. By doing this not all of the rows in the impulse response matrix are needed. Constraints can for example be removed in the beginning of the horizon to ensure stability and feasibility [18], or when it takes some time before control moves in the beginning of the horizon to have effect on the output variables. The adjoint method for calculating Ξ may benefit from this since structure is exploited row-wise, where one adjoint system is solved for each point on the horizon where output constraints are enforced. The forward method would need to calculate the full impulse response matrix first, in order to pick out the rows needed in the constraint gradient, and thus the advantage of enforcing constraints on only parts of the horizon can not be exploited.

6.4 Continuous-discrete model formulation

Assume that the provided system model is a continuous time model on the following form

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)), && (6.20) \\ z(t) &= g(x(t), u(t)). && (6.21) \end{aligned}$$

Moreover, assume that the control signal $u(t)$ is parameterized as piecewise constant over a sample interval T_s, while the model of the system still runs in continuous time.

$$u(t) = u(kT_s) = u_k, \ kT_s \leq t < [k+1]\, T_s.$$

To be able to handle continuous-discrete system models in our previous discrete time framework for calculating the objective function gradient and the impulse response matrix we need to discretize the system. That is, to find linear maps $A_k : \mathbb{R}^{N_x} \rightarrow \mathbb{R}^{N_x}$, $B_k : \mathbb{R}^{N_u} \rightarrow \mathbb{R}^{N_x}$ by integration of the linearized continuous time system model over a sample interval T_s. The linearized continuous time system model is given by

$$\begin{aligned} \Delta\dot{x}(t) &= A(t)\Delta x(t) + B(t)\Delta u(t), \\ \Delta z(t) &= C(t)\Delta x(t) + D(t)\Delta u(t), && (6.22) \end{aligned}$$

where

$$\begin{aligned} A(t) &= \frac{\partial f}{\partial x}(x(t)^{nom}, u(t)^{nom}), \ B(t) = \frac{\partial f}{\partial u}(x(t)^{nom}, u(t)^{nom}), && (6.23) \\ C(t) &= \frac{\partial g}{\partial x}(x(t)^{nom}, u(t)^{nom}), \ D(t) = \frac{\partial g}{\partial u}(x(t)^{nom}, u(t)^{nom}). && (6.24) \end{aligned}$$

We are interested in

$$\frac{\partial x_{k+1}}{\partial x_k} = \frac{\partial x([k+1]\, T_s)}{\partial x(kT_s)} = A_k, \ \frac{\partial x_{k+1}}{\partial u_k} = \frac{\partial x([k+1]\, T_s)}{\partial u(kT_s)} = B_k.$$

A_k is the state transition matrix $\Phi([k+1]\, T_s, kT_s)$. The state transition matrix for a LTV system satisfies

$$\begin{aligned} \dot{\Phi}(t, kT_s) &= A(t)\Phi(t, kT_s), && (6.25) \\ \Phi(kT_s, kT_s) &= I. \end{aligned}$$

A_k can be found by integrating (6.25) from $t = kT_s$ to $t = [k+1]\, T_s$. That is, to obtain the zero input response i.e. $\triangle u(kT_s) = 0$, of (6.22) from $t = kT_s$ to $t = [k+1]\, T_s$.

B_k can be found by finding the zero state response i.e. $\triangle x(kT_s) = 0$, of the system (6.22) from $t = kT_s$ to $t = [k+1]\,T_s$ which is given by

$$\begin{aligned} x([k+1]\,T_s) &= \int_{kT_s}^{[k+1]T_s} \Phi([k+1]\,T_s, \tau)B(\tau)u(\tau)d\tau \\ &= \left[\int_{kT_s}^{[k+1]T_s} \Phi([k+1]\,T_s, \tau)B(\tau)d\tau\right] u(kT_s). \end{aligned}$$

This reveals that

$$B_k = \int_{kT_s}^{[k+1]T_s} \Phi([k+1]\,T_s, \tau)B(\tau)d\tau, \tag{6.26}$$

which also is the solution of the following matrix ODE at time $t = [k+1]T_s$.

$$\dot{S}(t) = A(t)S(t) + B(t), \tag{6.27}$$

$$S(kT_s) = 0. \tag{6.28}$$

where
$$S(t) = \frac{\partial x(t)}{\partial u_k}.$$

The matrix ODE (6.27) is derived from applying the chain rule of differentiation to the original system model (6.20). Since there are no dynamics in the output equations, C_k and D_k can be obtained as before.

This shows that a continuous-discrete system model introduces some extra calculations in order to discretize the system and make this formulation fit our previous framework. The solver CVODES in the Suite of Nonlinear and Differential/Algebraic Equation Solvers (SUNDIALS) package includes forward and adjoint sensitivity analysis capabilities for finding sensitivity of the solution with respect to model parameters [13]. By regarding the control variables as constant parameters over a sampling interval in the system ODE (6.20) which upon the solution depends, the forward sensitivity capabilities in CVODES can be used to integrate the sensitivity equations (6.25) and (6.27).

It is also possible to obtain A_k and B_k by finite differences. This approach is perhaps best suited for systems that are easy to integrate, while integration of the sensitivity matrix ODEs probably becomes more advantageous when the system is harder to integrate. If finite differences are used there is however no need to obtain A_k and B_k first since the nonlinear ODE can be perturbed directly as in algorithm 6.1 and algorithm 6.4.

When using forward sensitivity analysis, the fact that (6.25) and (6.27) share Jacobians with (6.20) can be exploited by solving them in parallel. Overhead in the solver like step size and order selection can be then performed for all the systems simultaneously [13]. CVODES also provides error control for the sensitivity systems. One can choose whether error control should be performed for the sensitivity systems or just the nonlinear ODE. When obtaining these matrices by finite differences, choosing the size of the perturbation parameter can be very delicate matter resulting in that the error can be hard to control.

The following algorithm outlines the most basic steps that need to be performed in order to obtain the required sensitivities

Algorithm 6.7 Solve sensitivity equations

```
1:  // set initial condition
2:  $x(0) = x_{init}$
3:  for $k = 1$ to $N$ do
4:      // initialize sensitivity equations
5:      $S((k-1)T_s) = 0$
6:      $\Phi((k-1)T_s, (k-1)T_s) = I$
7:
8:      // integrate from $t = [k-1]T_s$ to $t = kT_s$
9:      $\dot{x}(t) = f(x(t), u_{k-1})$
10:     $\dot{\Phi}(t, (k-1)T_s) = A(t)\Phi(t, (k-1)T_s)$
11:     $\dot{S}(t) = A(t)S(t) + B(t)$
12:
13:     // store result
14:     $x_k = x(kT_s)$
15:     $A_{k-1} = \Phi(kT_s))$
16:     $B_{k-1} = S(kT_s)$
17: end for
```

Algorithm 6.5 and algorithm 6.2 will have the advantage that the matrices can be inserted in the impulse response matrix 'on the-fly' since also algorithm 6.7 provides these in forward time. Algorithm 6.6 and algorithm 6.3 need these matrices in reverse order. Since the sensitivity equations are solved by forward analysis, the matrices will need to be obtained and stored in memory first during the forward solve. They cannot be inserted 'on the-fly' which may be a disadvantage for large scale models where internal memory might be a limitation.

One possibility for circumventing storage of all the sensitivity matrices is to first solve the nonlinear ODE forward in time. Then, the sensitivity systems may be solved starting with the last sample interval and iterating backwards. Doing it this way, the sensitivities are provided in the same order as needed by algorithm 6.3 and 6.6 and can therefore be inserted 'on the-fly'. However, a major drawback is that the solution of the nonlinear ODE must be stored. If a variable step-size integration method is used it is not clear at which time instants to store the solution as the solver probably will choose different step-sizes for the solution of the sensitivity systems than for the nonlinear ODE. In this case a sophisticated storing system may be needed where interpolation techniques are used to obtain the solution at the time instants needed when solving the sensitivity ODEs. This makes it harder to integrate this approach with existing simulator codes. The fact that the nonlinear ODE and the sensitivity ODEs share Jacobians is not exploited either.

Another approach may be to first integrate the nonlinear ODE forward in time and only store the solution at the sampling instants which is the initial values needed for solving sensitivity equations on each sample interval. Then, it would be possible to start with the last sample interval, but now both the nonlinear ODE and the sensitivity systems must be solved. In this case one will not need to worry about variable step-size, storing and interpolation since the nonlinear ODE is solved one more time in parallel with the sensitivity equations. However, a drawback here will be that the nonlinear model is integrated two times. None of these two latter methods seems very promising since they involve some extra calculations but they may be worth considering in applications with large scale models where internal memory size is the fundamental limitation.

Alternative formulation

Even though the approach discussed in the previous section fits directly into our discrete time framework, it is possible to discretize the system using what we in the following will refer to as the alternative formulation for integration of sensitivity equations. The impulse response matrix can also be written as

$$\Xi = \begin{pmatrix} D_0 & 0 & 0 & \dots & 0 \\ C_1 \frac{\partial x_1}{\partial u_0} & D_1 & 0 & \ddots & \vdots \\ C_2 \frac{\partial x_2}{\partial u_0} & C_2 \frac{\partial x_2}{\partial u_1} & D_2 & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ C_{N-1} \frac{\partial x_{N-1}}{\partial u_0} & C_{N-1} \frac{\partial x_{N-1}}{\partial u_1} & C_{N-1} \frac{\partial x_{N-1}}{\partial u_2} & \dots & D_{N-1} \end{pmatrix}. \tag{6.29}$$

The terms $\frac{\partial x_j}{\partial u_k}$ can be obtained by again defining $S(t) = \frac{\partial x(t)}{\partial u_k}$, but instead of integrating just one sample interval as in (6.27) we define the following ODE which again is obtained by applying the chain rule of differentiation to the nonlinear ODE.

$$\dot{S}(t) = \begin{cases} A(t)S(t) + B(t) & kT_s \le t \le [k+1]\,T_s \\ A(t)S(t) & t \ge [k+1]\,T_s \end{cases}, \; S(kT_s) = 0. \tag{6.30}$$

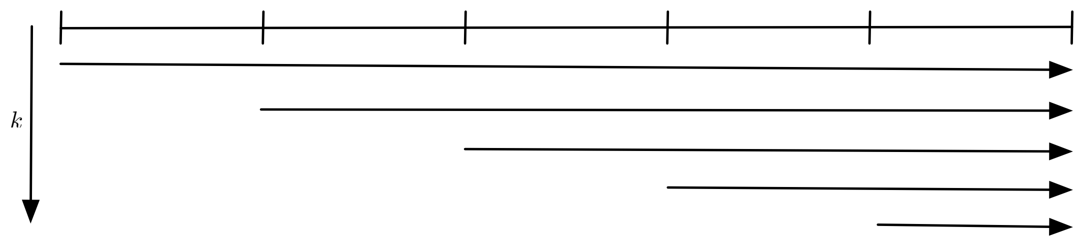
Now, the sensitivities in (6.29) are given by the solution of (6.30) at the end of each sample interval. That is, $\frac{\partial x_j}{\partial u_k} = S([j+1]T_s),\; j > k$. When there are many states compared to inputs, i.e. $N_x >> N_u$, integrating (6.30) to obtain $\frac{\partial x_j}{\partial u_k}$ which is inserted in (6.29) might be advantageous compared to integrating the sensitivities in algorithm 6.7. Only integration of (6.30) is needed where $S \in \mathbb{R}^{N_x \times N_u}$. Integration of the sensitivity system (6.25) used in algorithm 6.7 where $\Phi \in \mathbb{R}^{N_x \times N_x}$ is not needed. Thus, this formulation will reduce the dimension of the sensitivity ODE from $(N_x + N_u)N_x$ to $N_x N_u$. The main drawback with this method is that the total integration length is more than one time the horizon length.

Algorithm 6.8 Solve sensitivity equations, alternative formulation

```
// set initial condition
$x_0 = x_{init}$
for $k = 1$ to $N-1$ do
    // initialize
    $x((k-1)T_s) = x_{k-1}$
    $S((k-1)T_s) = 0$

    // integrate from $t = [k-1]T_s$ to $t = kT_s$
    $\dot{x}(t) = f(x(t), u_{k-1})$
    $S(t) = A(t)S(t) + B(t)$

    // store result
    $S_{k,k} = S(kT_s)$
    $x_k = x(kT_s)$

    for $n = k+1$ to $N-1$ do
        // integrate from $t = [n-1]T_s$ to $t = nT_s$
        $\dot{x}(t) = f(x(t), u_{n-1})$
        $\dot{S}(t) = A(t)S(t)$

        // store result
        $x_k = x(kT_s)$
        $S_{n,k} = S(nT_s)$
    end for
end for
```

An illustration of the integration intervals in each iteration is shown in figure 6.2. Note that the integration intervals are quite similar to what is done in algorithm 6.1 and 6.4 when using finite differences in the sense that the intervals decrease in length due to causality.

Figure 6.2: Illustration of integration intervals



For the objective function we have that

$$\begin{aligned}
\frac{\partial J}{\partial u_k} &= u_k^T R + \sum_{i=0}^{N-1} \left[z_i^T Q \frac{\partial z_i}{\partial u_k} \right] + x_N^T P \frac{\partial x_N}{\partial u_k} \\
&= u_k^T R + \sum_{i=k}^{N-1} \left[z_i^T Q \frac{\partial z_i}{\partial u_k} \right] + x_N^T P \frac{\partial x_N}{\partial u_k} \\
&= u_k^T R + z_k^T Q \frac{\partial z_k}{\partial u_k} + z_{k+1}^T Q \frac{\partial z_{k+1}}{\partial x_{k+1}} \frac{\partial x_{k+1}}{\partial u_k} + \\
&\quad z_{k+2}^T Q \frac{\partial z_{k+2}}{\partial x_{k+2}} \frac{\partial x_{k+2}}{\partial u_k} + \ldots + z_{N-1}^T Q \frac{\partial z_{N-1}}{\partial x_{N-1}} \frac{\partial x_{N-1}}{\partial u_k} + x_N^T P \frac{\partial x_N}{\partial u_k} \\
&= u_k^T R + z_k^T Q D_k + z_{k+1}^T Q C_{k+1} S_{k+1,k+1} + z_{k+2}^T Q C_{k+2} S_{k+2,k+1} + \ldots + \\
&\quad z_{N-1}^T Q C_{N-1} S_{N-1,k+1} + x_N^T P S_{N,k+1} \\
&= u_k^T R + z_k^T Q D_k + \sum_{i=k+1}^{N-1} \left[z_i^T Q C_i S_{i,k+1} \right] + x_N^T P S_{N,k+1},
\end{aligned}$$

which is used in the following algorithm that calculates the objective function gradient when the sensitivities are obtained by the alternative approach in algorithm 6.8.

Algorithm 6.9 Calculate $\nabla_u J$, alternative formulation

```
1: for $k = 0$ to $N - 2$ do
2:     $\psi = 0$
3:     for $i = k + 1$ to $N - 1$ do
4:         $\psi = \psi + z_i^T Q C_i S_{i,k+1}$
5:     end for
6:     $\nabla_u J[kN_u + 1 : (k + 1)N_u] = R^T u_k + D_k^T Q z_k + \psi^T + S_{N,k+1}^T P x_N$
7: end for
8: $\nabla_u J[(N - 1)N_u + 1 : NN_u] = R^T u_{N-1} + D_{N-1}^T Q z_{N-1} + S_{N,N}^T P x_N$
```

The sensitivities calculated in algorithm 6.8 can also be used to build the impulse response matrix in algorithm 6.10. Observe that these two algorithms are quite simple as most of the work is done in algorithm 6.8. In fact, algorithm 6.9 and 6.10 only calculate how the sensitivity from the control variables to the state variables propagates through the output equations and the objective function.

Algorithm 6.10 Calculate impulse response matrix Ξ, alternative formulation

```
1: for $k = 1$ to $N$ do
2:     $\Xi[(k - 1)N_z + 1 : kN_z, (k - 1)N_u + 1 : kN_u] = D_{k-1}$
3:     for $n = k + 1$ to $N$ do
4:         $\Xi[(n - 1)N_z + 1 : nN_z, (k - 1)N_u + 1 : kN_u] = C_{n-1} S_{n-1,k}$
5:     end for
6: end for
```

6.5 Runtime considerations

In this section, theoretical bounds for runtimes of the different algorithms are developed. The notation in definition 2 which describes the limiting behavior of functions is used to describe runtime as function of problem input size. The results from this section will be compared with actual simulation results on some real world benchmark examples later.

Algorithm 6.1 - Calculate $\nabla_u J$ - Finite differences

First, the model is simulated forward in time using the nominal input, involving N iterations of the nonlinear model equations. Then, the algorithm iterates the nonlinear model $N_u \sum_{k=1}^{N} k = \frac{1}{2} N_u N(N+1)$ times in order to calculate the perturbed objective function values for all the inputs. This results in runtime $\mathcal{O}(N^2)$.

If a continuous time model is used the only modification needed is to instead invoke an ODE solver to obtain the solution at the sampling instants instead of iterating the discrete model equations. In this case, the runtime for iterating the model one sample interval will also depend on the length of the sampling intervals T_s and thus, the runtime will be $\mathcal{O}(N^2 T_s^2) = \mathcal{O}(t_f^2)$ where t_f is the end of the horizon.

Algorithm 6.2 - Calculate $\nabla_u J$ - Forward method

The model equations are iterated N times during the forward solve. Next, with the two nested for-loops a number of $\mathcal{O}(N^2)$ matrix multiplications are required, and thus the runtime will be $\mathcal{O}(N^2)$.

Algorithm 6.3 - Calculate $\nabla_u J$ - Adjoint method

The model is again simulated forward in time resulting in N iterations of the model equations. The state variables $x_k,\ k \in \{1, \ldots, N\}$ need to be stored for the backward solve, and therefore the internal memory must be large enough to fit this trajectory. Then $\nabla_u J$ can be found by doing one additional reverse simulation. The runtime will only be $\mathcal{O}(N)$ which is a huge improvement compared to $\mathcal{O}(N^2)$ for algorithm 6.2 and 6.1.

Algorithm 6.4 - Calculate impulse response matrix Ξ - Finite differences

Similar as for the objective function gradient, N iterations of the model are needed for the nominal control input, and additional $N_u \sum_{k=1}^{N-1} k = \frac{1}{2} N_u (N-1) N$ iterations for the perturbed control inputs, resulting in runtime $\mathcal{O}(N^2)$.

Again, if a continuous time model is used, the runtime will instead be $\mathcal{O}(N^2 T_s^2) = \mathcal{O}(t_f^2)$ since the length of each sampling interval will also be of importance for a continuous time ODE solver.

Algorithm 6.5 - Calculate impulse response matrix Ξ - Forward method

First the model equations are iterated N times. Then, inspection of algorithm 6.5 and table 6.1 reveals that $\sum_{k=1}^{N-1} k + \sum_{k=1}^{N} k = N^2$ matrix multiplications are needed. The runtime is $\mathcal{O}(N^2)$.

Algorithm 6.6 - Calculate impulse response matrix Ξ - Adjoint method

First the model equations are iterated N times. By inspecting algorithm 6.6 and table 6.2 we see that $\sum_{k=1}^{N-1} k + \sum_{k=1}^{N} k = N^2$ matrix multiplications are needed. Therefore also this algorithm will have runtime $\mathcal{O}(N^2)$ if the full impulse response matrix is to be found. If output constraints are only enforced in a number of N_{zc} points on the horizon, this algorithm will have runtime $\mathcal{O}(NN_{zc})$ since only N_{zc} rows of the impulse response matrix are computed.

Algorithm 6.7 - Solve sensitivity equations

The ODEs are integrated over a interval of length $NT_s = t_f$ and the runtime will therefore be $\mathcal{O}(NT_s) = \mathcal{O}(t_f)$.

Algorithm 6.8 - Solve sensitivity equations, alternative formulation

The ODEs are integrated over a interval of length $\sum_{k=1}^{N-1} kT_s = \frac{1}{2}N(N-1)T_s = \frac{1}{2}(N-1)t_f$ resulting in runtime $\mathcal{O}(N^2T_s) = \mathcal{O}(Nt_f)$.

Algorithm 6.9, Calculate $\nabla_u J$, alternative formulation

This algorithm will also have runtime $\mathcal{O}(N^2)$. Compared to algorithm 6.2 fewer matrix multiplications are needed in each iteration as some of these matrix multiplications will already be incorporated in the integration of the sensitivity equations by the alternative approach.

Algorithm 6.10, Calculate impulse response matrix Ξ, alternative formulation

Again the runtime will be $\mathcal{O}(N^2)$. Also here, compared to algorithm 6.5 fewer matrix multiplications are needed in each iteration since these are incorporated in the integration of the sensitivity equations by the alternative approach.

Summary

For discrete time systems algorithm 6.1, 6.2 or 6.3 may be used to calculate objective function gradient $\nabla_u J$. Here, the adjoint method (algorithm 6.3) should be preferred since this method is much faster. This will be demonstrated later. Calculating the impulse response matrix Ξ for discrete time systems can be done by either algorithm 6.4, 6.5 or 6.6. This matrix is required by algorithms that use constraint gradient information of output constraints. If constraints are not enforced on all the points of the horizon, the adjoint method can benefit from this by only calculating some of the rows in the impulse response matrix. The forward method can not exploit this since structure of the impulse response matrix is exploited column-wise.

For continuous-discrete systems the algorithms presented earlier can be combined in different ways to obtain $\nabla_u J$ or Ξ. Some properties and runtimes are summarized in the following tables.

Table 6.3: Summary of algorithms for calculating $\nabla_u J$ for continuous-discrete systems

Algorithms	Properties
6.1	• Integration length $\mathcal{O}(t_f^2)$ for nonlinear system ODE of dimension N_x.
6.7 and 6.2	• Integration length $\mathcal{O}(t_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension $N_x(N_x + N_u)$. • $\mathcal{O}(N^2)$ matrix multiplications. • Insertion 'on the-fly' possible.
6.7 and 6.3	• Integration length $\mathcal{O}(t_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension $N_x(N_x + N_u)$. • $\mathcal{O}(N)$ matrix multiplications. • Insertion 'on the-fly' not possible. Sensitivity matrices (A_k, B_k) need to be stored in memory during the forward solve as they are needed in reverse order during the reverse solve.
6.8 and 6.9	• Integration length $\mathcal{O}(N^2 T_s) = \mathcal{O}(N t_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension $N_x N_u$. • $\mathcal{O}(N^2)$ matrix multiplications. • Efficient when $N_x >> N_u$ as dimension of sensitivity ODE is reduced. • Insertion 'on the-fly' possible.

Table 6.4: Summary of algorithms for calculating Ξ for continuous-discrete systems

Algorithms	Properties
6.4	• Integration length $\mathcal{O}(t_f^2)$ for nonlinear system ODE of dimension N_x.
6.7 and 6.5	• Integration length $\mathcal{O}(t_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension $N_x(N_x + N_u)$. • $\mathcal{O}(N^2)$ matrix multiplications. • Insertion 'on the-fly' possible.
6.7 and 6.6	• Integration length $\mathcal{O}(t_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension $N_x(N_x + N_u)$. • $\mathcal{O}(NN_{zc})$ matrix multiplications. • Insertion 'on the-fly' not possible. Sensitivity matrices (A_k, B_k) need to be stored in memory during the forward solve as they are needed in reverse order during the reverse solve.
6.8 and 6.10	• Integration length $\mathcal{O}(N^2T_s) = \mathcal{O}(Nt_f)$ for nonlinear system ODE of dimension N_x and sensitivity ODE of dimension N_xN_u. • $\mathcal{O}(N^2)$ matrix multiplications. • Efficient when $N_x >> N_u$ as dimension of sensitivity ODE is reduced. • Insertion 'on the-fly' possible.

6.6 Simulation examples

In this section efficiency of implementations of the different algorithms will be investigated and compared to the theoretical considerations above. All the simulations in this thesis are performed on a laptop computer with software and hardware listed in appendix A-2. All simulations involving runtime measurement are done several times and averaged to obtain more accurate results (10 - 100 times, depending on runtime for each algorithm). Runtime is measured using a stop watch timer measuring CPU time.

Simple nonlinear discrete model

For the discrete time case the runtime for each of the algorithms will depend on how fast the linear algebra can be done. The following simple discrete time nonlinear system is used to investigate and compare efficiency of algorithm 6.1, 6.2 and 6.3 for computing the objective function gradient.

$$\begin{aligned} x_{k+1}[1] &= \frac{x_k[1]}{k} + x_k[2]^2 + u_k[1], \\ x_{k+1}[2] &= x_k[2]\sin(k) - x_k[1]k + u_k[2], \\ z_k &= x_k[1] + x_k[2] + 0.1u_k[1] + 0.2u_k[2], \end{aligned} \tag{6.31}$$

which of the Jacobians are given by

$$A_k = \begin{bmatrix} \frac{1}{k} & 2x_k[2] \\ -k & \sin(k) \end{bmatrix},\ B_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ C_k = \begin{bmatrix} 1 & 1 \end{bmatrix},\ D_k = \begin{bmatrix} 0.1 & 0.2 \end{bmatrix}.$$

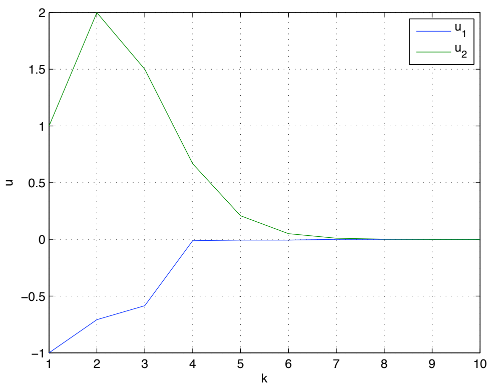
This system can be stabilized by using the state feedback linearization control law

$$\begin{aligned} u_1[k] &= -x_2[k]^2, \\ u_2[k] &= x_1[k]k. \end{aligned}$$

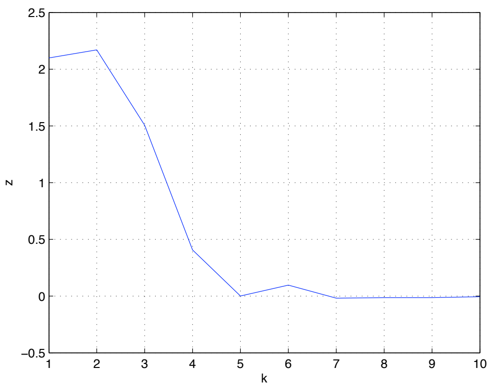
Gradient calculation of the objective function $J = \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N$, where Q, R and P set to the identity, is implemented in `obj_grad.m`.

First the system is simulated with the feedback linearization controller with initial value $x_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$. The input and output trajectories are shown in 6.3.

Figure 6.3: Simple nonlinear discrete model - Solution



(a) Control



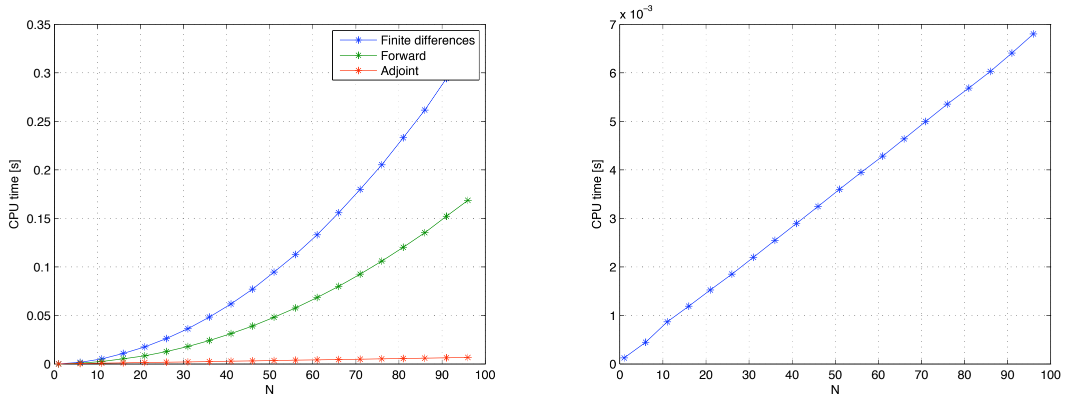
(b) Output

The objective function gradient with time horizon length $N = 10$ is calculated first by finite differences in algorithm 6.1. Then the calculation is done by the forward method in algorithm 6.2, and finally by the adjoint method in algorithm 6.3. The gradient found by the adjoint and the forward method deviate by order of magnitude 10^{-16} which is about the double precision in Matlab. The gradient calculated using finite differences is computed using perturbation size 10^{-10}. The gradient obtained by finite differences deviates slightly from those obtained by the forward and the adjoint method. This is expected and must be attributed to that gradient computation by finite differences is not exact in contrast to the forward and the adjoint method which will be exact up to numerical precision.

Runtime simulations

Gradient calculation for the test system (6.31) was measured using different time horizon lengths. The runtime of the different methods are compared in figure 6.4. As expected, the adjoint method becomes more superior as the horizon length is increased. The runtime of the forward method and calculation by finite differences for this example is clearly $\mathcal{O}(N^2)$. Runtime for the adjoint method is shown in more detail in figure 6.4b (same as red line in figure 6.4a). As we see, the runtime for the adjoint method for this example is $\mathcal{O}(N)$. These results are consistent with the theoretical bounds found earlier.

Figure 6.4: Runtime - Calculate $\nabla_u J$



(a) Finite differences, forward method and adjoint method

(b) Adjoint method (zoomed)

This simple example demonstrates that the adjoint method can be very powerful for gradient calculation of the NMPC objective function.

Van de Vusse reactor

In this section we will present a model of a Van de Vusse reaction scheme which is to be used for benchmarking of the continuous-discrete case. This reactor has frequently been used as a benchmark problem for nonlinear control design. Its highly nonlinear behavior is among several of the properties which is desirable in terms of offering comparison fairness.

The reaction scheme is comprised by two concentration balances of the reactant c_A and the the wanted product c_B. Thus, the main reaction is $A \rightarrow B$. In addition the side reactions $2A \rightarrow D$ and $B \rightarrow C$ which both yield unwanted products take place.

The model also contains energy balances for the reactor temperature T and the cooling jacket temperature T_c. The model and parameters for the reactor are taken from [22].

Figure 6.5: Van de Vusse reactor

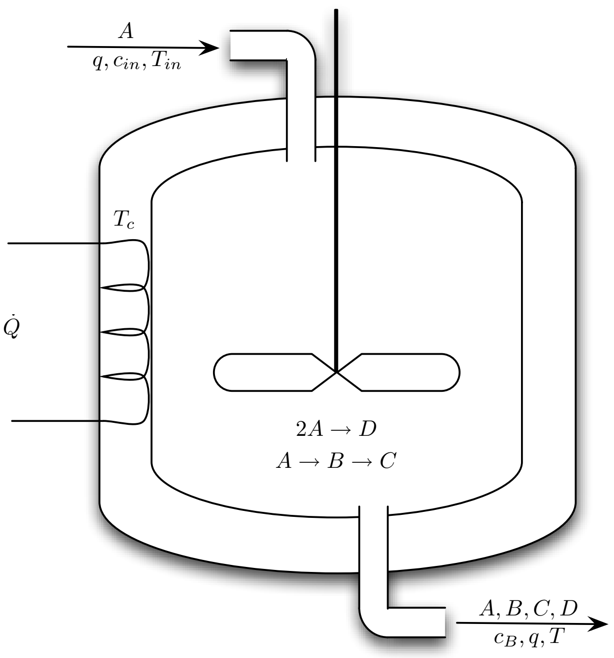


Table 6.5: Van de Vusse reactor parameters

Parameter	Unit
$\alpha = 30.8285$	$\frac{1}{h}$
$\gamma = 100$	$\frac{K}{MJ}$
$k_{10} = (1.287) \cdot 10^{12}$	$\frac{1}{h}$
$k_{20} = (9.042) \cdot 10^{6}$	$\frac{m^3}{mol \cdot h}$
$\Delta H_{AB} = 4.2$	$\frac{kJ}{mol}$
$\Delta H_{AD} = -41.85$	$\frac{kJ}{mol}$
$\Delta H_{BC} = -11$	$\frac{kJ}{mol}$
$c_{in} = 5100 \pm 600$	$\frac{mol}{m^3}$
$\beta = 86.668$	$\frac{1}{h}$
$\delta = 3.556 \cdot 10^{-4}$	$\frac{m^3 K}{kJ}$
$T_{in} = 104.9$	K
$E_1 = 9758.3$	dimensionless
$E_2 = 8560$	dimensionless

The nonlinear system model is given by

$$\begin{aligned}
\dot{c}_A &= -k_1(T)c_A - k_2(T)c_A^2 + [c_{in} - c_A]\,u_1, \\
\dot{c}_B &= k_1(T)\,[c_A - c_B] - c_B u_1, \\
\dot{T} &= h(c_A, c_B, T) + \alpha\,[T_c - T] + [T_{in} - T]\,u_1, \\
\dot{T}_c &= \beta\,[T - T_c] + \gamma u_2,
\end{aligned} \tag{6.32}$$

where the reaction enthalpy contribution is modeled by

$$h(c_A, c_B, T) = -\delta\left(k_1(T)\,[c_A \Delta H_{AB} + c_B \Delta H_{BC}] + k_2(T)c_A^2 \Delta H_{AD}\right).$$

The reaction kinetics are modeled using Arrhenius functions for the temperature

$$k_i(T) = k_{i0} e^{\left(\frac{-E_i}{T+273.15}\right)},\ i = 1, 2.$$

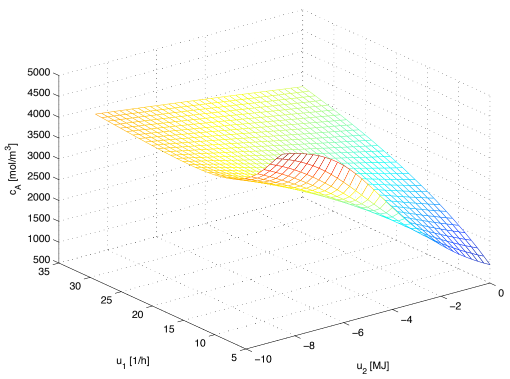
The controlled inputs are

$$\begin{aligned}
u_1 &= \frac{q}{V_R}, \\
u_2 &= \dot{Q}.
\end{aligned}$$

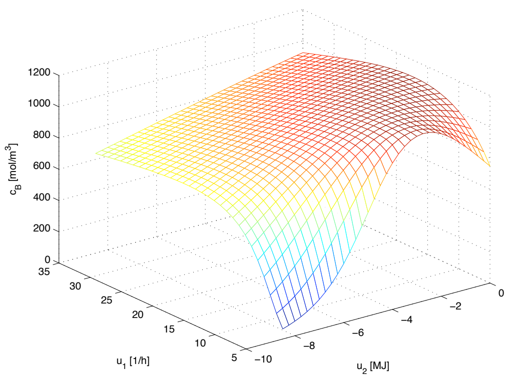
u_1 is the flow rate into the reactor q scaled by the reactor volume V_R and is therefore measured in $\frac{1}{h}$. u_2 is the the cooling capacity measured in $\frac{kJ}{h}$. Note that $u_2 < 0$.

To understand the fundamental behavior and properties of the reactor, its steady state solution has been found in `vdv_ss.m` where (6.32) is numerically solved for the state variables with all derivatives set to zero. The solutions for a grid of different values for u_1 and u_2 are shown in figure 6.6.

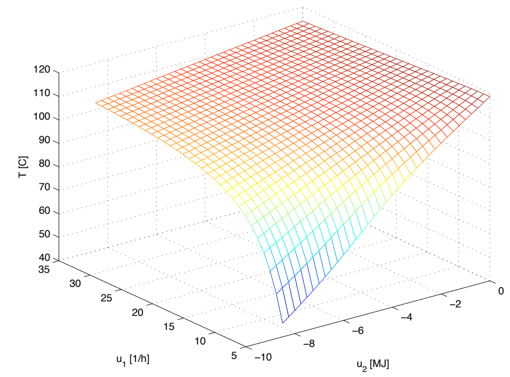
Figure 6.6: Steady state solution of Van de Vusse reactor



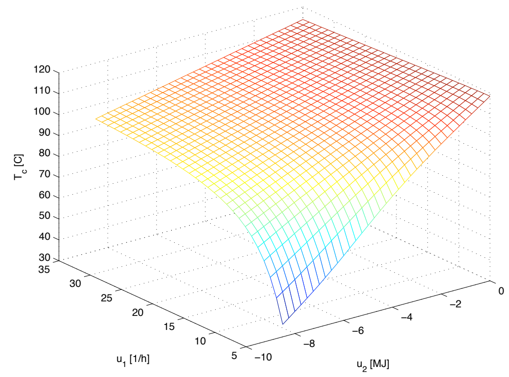
(a) c_A



(b) c_B



(c) T



(d) T_c

One would probably like to maximize the concentration of the wanted product c_B around the top shown in the figure above.

The reactor model is implemented and simulated using the solver CVODES. Documentation for mathematical considerations regarding the solver and user interface for CVODES can be found in [13].

The following scenario is simulated:

The initial conditions are set to

$$\begin{aligned} c_A(0) &= 0\,\frac{mol}{m^3}, \\ c_B(0) &= 750\,\frac{mol}{m^3}, \\ T(0) &= 85\,^\circ\mathrm{C}, \\ T_c(0) &= 100\,^\circ\mathrm{C}. \end{aligned}$$

The system is simulated for 1 hour, with a sampling interval set to 72 seconds ($T_s = 72s$, $t_f = 1h$ and $N = 50$)

The following inputs are used to excite the system dynamics

$$u_1(t) = \begin{cases} 12\,\frac{1}{h} & 0 \le t \le 2/3\;h \\ 1\,\frac{1}{h} & 2/3 \le t \le 1\;h \end{cases},$$

$$u_2(t) = \begin{cases} -3000\,\frac{kJ}{h} & 0 \le t \le 1/3\;\frac{1}{h} \\ -1000\,\frac{kJ}{h} & 1/3 \le t \le 1\;\frac{1}{h} \end{cases}.$$

Simulation of the reactor model, integration of sensitivity equations using algorithms 6.7 and 6.8, and calculation of the impulse response matrix Ξ using both algorithm 6.5, 6.6 and 6.10 is implemented. Calculation of the impulse response matrix where no structure is exploited is also implemented in order to compare runtime with the forward and the adjoint method.

The simulation environment is written in Matlab which invokes the CVODES solver via `mex` files which were built as an additional interface when compiling CVODES.

Table 6.6: Organization of files for simulation of Van de Vusse reactor

File	Purpose
`cvodes_vdv.m`	This is the main script which implements algorithm 6.7 and invokes the CVODES solver routine to solve the nonlinear ODE and the sensitivity systems. Then the impulse response matrix is computed using algorithm 6.5 and algorithm 6.6.
`cvodes_vdv_alt.m`	This is the main script which implements algorithm 6.8 and invokes the CVODES solver routine to solve the nonlinear ODE and the sensitivity systems. Then the impulse response matrix is computed using algorithm 6.10.
`cvodes_vdv_f.m`	Provides the nonlinear model.
`cvodes_vdv_J.m`	Provides Jacobian of the nonlinear model.
`cvodes_vdv_fS.m`	Provides right hand side of the sensitivity systems (6.25) and (6.27).
`cvodes_vdv_alt_fS.m`	Provides right hand side of the sensitivity system (6.30).

The files `cvodes_vdv_J.m`, `cvodes_vdv_fS.m` and `cvodes_vdv_alt_fS.m` provide the Jacobian of the nonlinear ODE and the sensitivity systems. The Jacobians needed in these equations are

$$A(t) = \begin{pmatrix} \frac{\partial \dot{c}_A}{\partial c_A} & \frac{\partial \dot{c}_A}{\partial c_B} & \frac{\partial \dot{c}_A}{\partial T} & \frac{\partial \dot{c}_A}{\partial T_c} \\ \frac{\partial \dot{c}_B}{\partial c_A} & \frac{\partial \dot{c}_B}{\partial c_B} & \frac{\partial \dot{c}_B}{\partial T} & \frac{\partial \dot{c}_B}{\partial T_c} \\ \frac{\partial \dot{T}}{\partial c_A} & \frac{\partial \dot{T}}{\partial c_B} & \frac{\partial \dot{T}}{\partial T} & \frac{\partial \dot{T}}{\partial T_c} \\ \frac{\partial \dot{T}_c}{\partial c_A} & \frac{\partial \dot{T}_c}{\partial c_B} & \frac{\partial \dot{T}_c}{\partial T} & \frac{\partial \dot{T}_c}{\partial T_c} \end{pmatrix}, \qquad B(t) = \begin{pmatrix} \frac{\partial \dot{c}_A}{\partial u_1} & \frac{\partial \dot{c}_A}{\partial u_2} \\ \frac{\partial \dot{c}_B}{\partial u_1} & \frac{\partial \dot{c}_B}{\partial u_2} \\ \frac{\partial \dot{T}}{\partial u_1} & \frac{\partial \dot{T}}{\partial u_2} \\ \frac{\partial \dot{T}_c}{\partial u_1} & \frac{\partial \dot{T}_c}{\partial u_2} \end{pmatrix}.$$

Each of the elements can be obtained from direct differentiation of (6.32) by hand.

$$
\begin{aligned}
\frac{\partial \dot{c}_A}{\partial c_A} &= -k_1(T) - 2k_2(T)c_A - u_1, \\
\frac{\partial \dot{c}_A}{\partial c_B} &= 0, \\
\frac{\partial \dot{c}_A}{\partial T} &= -k_1(T)\frac{E_1}{(T+273.15)^2}c_A - k_2(T)\frac{E_2}{(T+273.15)^2}c_A^2, \\
\frac{\partial \dot{c}_A}{\partial T_c} &= 0, \\
\frac{\partial \dot{c}_B}{\partial c_A} &= k_1(T), \\
\frac{\partial \dot{c}_B}{\partial c_B} &= -k_1(T) - u_1, \\
\frac{\partial \dot{c}_B}{\partial T} &= k_1(T)\frac{E_1}{(T+273.15)^2}(c_A - c_B), \\
\frac{\partial \dot{c}_B}{\partial T_c} &= 0, \\
\frac{\partial \dot{T}}{\partial c_A} &= -\delta\left[k_1(T)\Delta H_{AB} + 2k_2(T)c_A\Delta H_{AD}\right], \\
\frac{\partial \dot{T}}{\partial c_B} &= -\delta k_1(T)\Delta H_{BC}, \\
\frac{\partial \dot{T}}{\partial T} &= -\delta\left[k_1(T)\frac{E_1}{(T+273.15)^2}\left[c_A\Delta H_{AB} + c_B\Delta H_{BC}\right] + k_2(T)\frac{E_2}{(T+273.15)^2}c_A^2\Delta H_{AD}\right] - \alpha - u_1, \\
\frac{\partial \dot{T}}{\partial T_c} &= \alpha, \\
\frac{\partial \dot{T}_c}{\partial c_A} &= 0, \\
\frac{\partial \dot{T}_c}{\partial c_B} &= 0, \\
\frac{\partial \dot{T}_c}{\partial T} &= \beta, \\
\frac{\partial \dot{T}_c}{\partial T_c} &= -\beta,
\end{aligned}
$$

with respect to the state variables, and

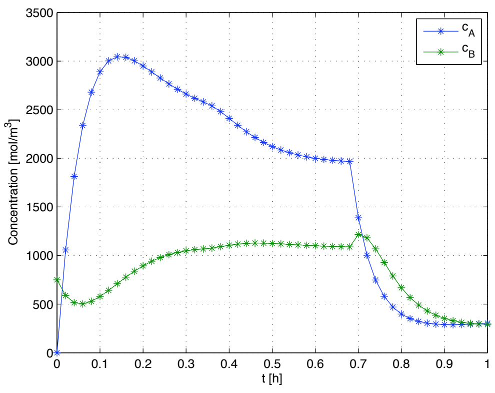
$$\begin{aligned}
\frac{\partial \dot{c}_A}{\partial u_1} &= c_{in} - c_A, \\
\frac{\partial \dot{c}_A}{\partial u_2} &= 0, \\
\frac{\partial \dot{c}_B}{\partial u_1} &= -c_B, \\
\frac{\partial \dot{c}_B}{\partial u_2} &= 0, \\
\frac{\partial \dot{T}}{\partial u_1} &= T_{in} - T, \\
\frac{\partial \dot{T}}{\partial u_2} &= 0, \\
\frac{\partial \dot{T}_c}{\partial u_1} &= 0, \\
\frac{\partial \dot{T}_c}{\partial u_2} &= \gamma,
\end{aligned}$$

with respect to the control variables.

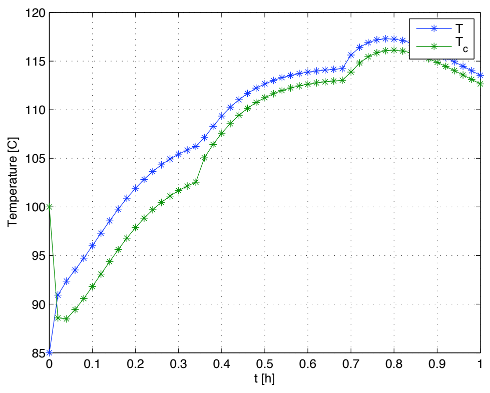
For more complex systems which may be intractable to differentiate by hand these Jacobians could have been obtained by finite differences or automatic differentiation. The latter approach should be preferred since it is exact.

The solution of the system ODE at each sample instant is shown in figure 6.7.

Figure 6.7: Van de Vusse reactor - Solution



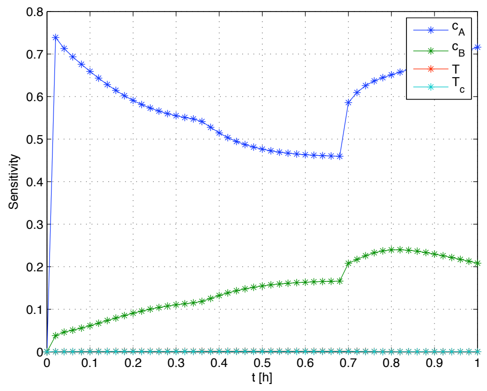
(a) Concentrations



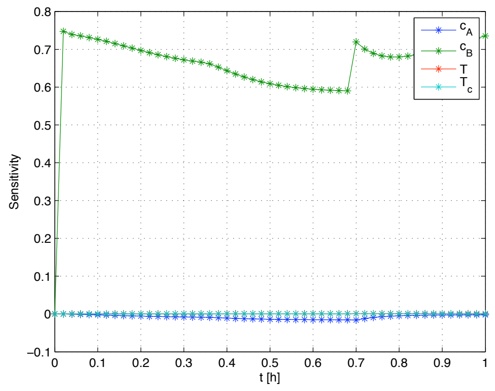
(b) Temperatures

The sensitivities found at each sample instant are shown for the state variables in figure 6.8 and for the control variables in figure 6.9. These are the sensitivities obtained from the implementation of algorithm 6.7 in `cvodes_vdv.m`.

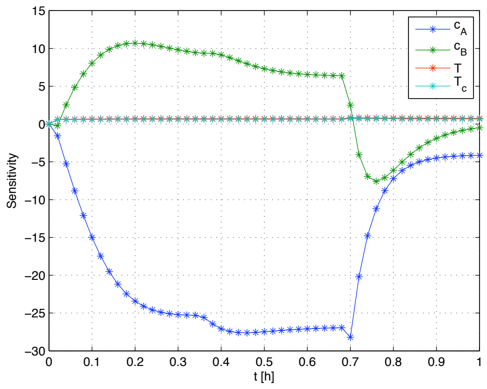
Figure 6.8: Van de Vusse reactor - State sensitivities



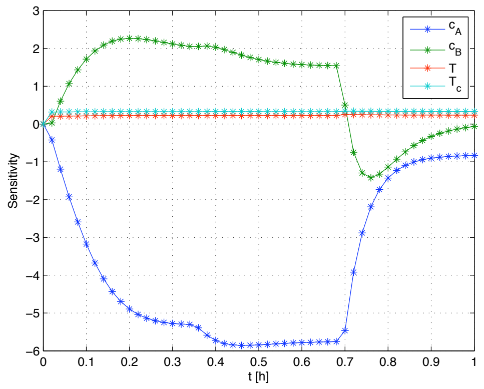
(a) c_A sensitivities



(b) c_B sensitivities

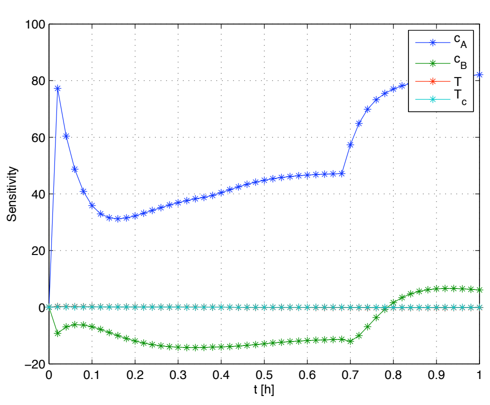


(c) T sensitivities

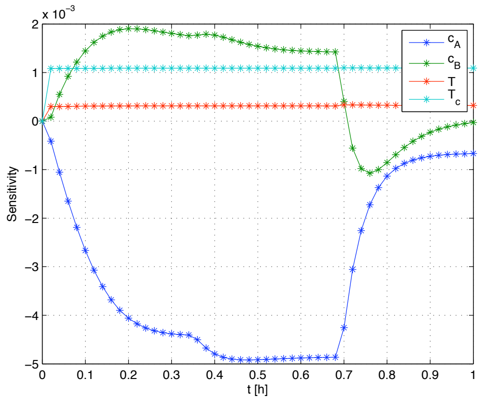


(d) T_c sensitivities

Figure 6.9: Van de Vusse reactor - Control sensitivities



(a) u_1 sensitivities



(b) u_2 sensitivities

Simulation of the reactor model and calculation of Ξ by using algorithm 6.4 with finite differences has also been implemented in `finite_diff_vdv.m` using the Matlab solver $ode15s$ to verify that this gave the same result.

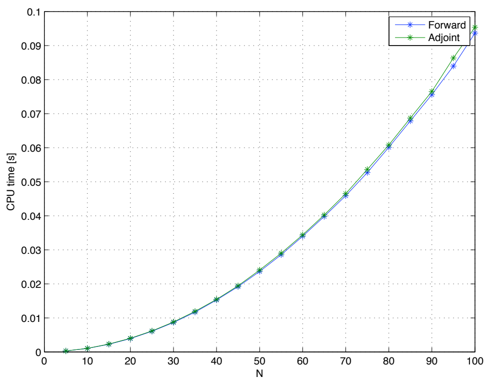
Even though the implementation of the algorithms was fairly straightforward from the developed pseudo code, some modifications had to be made to algorithm 6.7 and 6.8 for integrating sensitivities. When invoking CVODES to integrate the ODEs to a specific time $t = kT_s$ it integrates to a value past $t = kT_s$ and then interpolates back and returns the solution at $t = kT_s$. However, when invoking CVODES again, it continues from its internal time value. Since the integration continued past $t = kT_s$ with control input u_{k-1}, we cannot continue the integration from this point. Both the nonlinear ODE and the sensitivity ODEs need to be reinitialized at $t = kT_s$ before the integration continues with control input u_k to obtain the solution at the next sampling instant $t = [k+1]T_s$. Even though this is quite obvious, reinitialization of the ODEs at every sampling instant has severe impact on the runtime as the solver has to start over with small step-sizes each time and essentially a new ODE is solved for each sample interval.

Runtime simulations

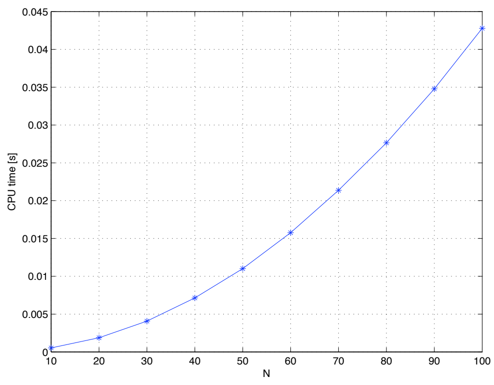
In this part, runtimes of the different algorithms for calculating the impulse response matrix Ξ will be measured and compared with the runtime considerations made for each algorithm earlier.

To compare runtimes we would like to run the algorithms with different input sizes. The number of operations needed to build the impulse response matrix in algorithm 6.5, 6.6 and 6.10 will only depend on the number of sampling instants on the horizon and the speed at which the linear algebra can be performed as no integration of sensitivities is involved. Runtimes for different input sizes N for these algorithms are shown in figure 6.10. A method that does not exploit any structure of the impulse response matrix has also been implemented to highlight the importance of taking structure into account.

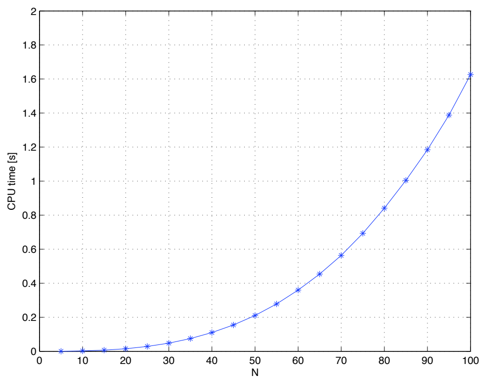
Figure 6.10: Runtime - Calculate Ξ



(a) Algorithm 6.5, 6.6 - Forward and adjoint method



(b) Algorithm 6.10 - Alternative approach



(c) Without exploiting any structure

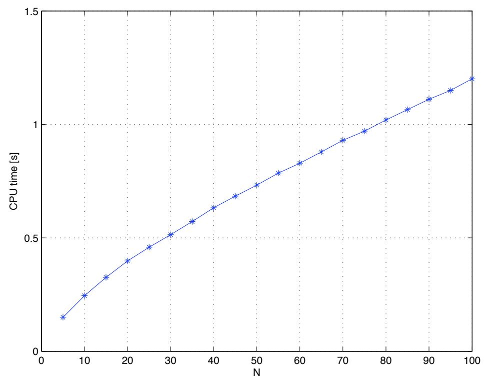
As expected figure 6.10 shows that runtime for the forward and the adjoint method and also the alternative method is $\mathcal{O}(N^2)$. Observe the dramatic increase in runtime when no structure of the impulse response matrix is exploited. The runtime for this method is $\mathcal{O}(N^3)$ which is consistent with the figure above. Note that this method is only included for comparison purpose, and should not be used since the forward or the adjoint method is much faster.

The runtime for algorithm 6.7 where sensitivity ODEs are solved will depend on the length of integration which is $t_f = NT_s$, i.e. the runtime is $\mathcal{O}(t_f) = \mathcal{O}(NT_s)$ (under the assumption that the solver has a predefined minimum step-length). Solution of the sensitivity ODEs using the alternative approach in algorithm 6.8 will depend both on simulation time t_f and number of sampling instants N since the runtime was found to be $\mathcal{O}(N^2T_s) = \mathcal{O}(Nt_f)$. This leaves 3 different possibilities for simulating runtime with different input sizes. All these 3 combinations are investigated and the values used to compare runtime for the different scenarios are listed in table 6.7.

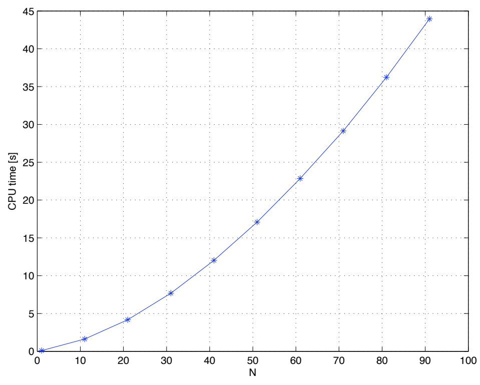
Table 6.7: Inputs for comparing runtime

N, T_s variable, t_f constant	t_f, T_s variable, N constant	t_f, N variable, T_s constant
$t_f = 1$	$t_f = \frac{1}{20}, \frac{2}{20}, \frac{3}{20}, \ldots, 1$	$t_f = \frac{1}{20}, \frac{2}{20}, \frac{3}{20}, \ldots, 1$
$N = 5, 10, 15, \ldots, 100$	$N = 100$	$N = \frac{t_f}{T_s} = 5, 10, 15, \ldots 100$
$T_s = \frac{t_f}{N} = \frac{1}{5}, \frac{1}{10}, \frac{1}{15}, \ldots, \frac{1}{100}$	$T_s = \frac{t_f}{N} = \frac{1}{2000}, \frac{2}{2000}, \frac{3}{2000}, \ldots, \frac{1}{100}$	$T_s = \frac{1}{100}$

Figure 6.11: Runtime - CVODES integration, N, T_s variable, t_f constant



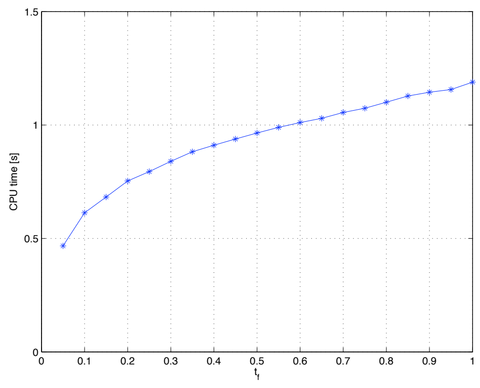
(a) Algorithm 6.7



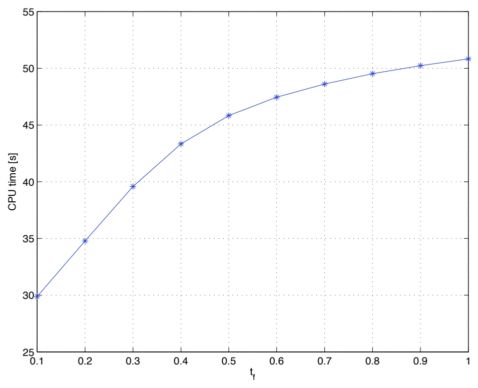
(b) Algorithm 6.8 - Alternative method

Observe that for t_f constant, the integration length for algorithm 6.7 is constant, and since the runtime was found to be $\mathcal{O}(t_f) = \mathcal{O}(NT_s)$ one might not expect the result shown in the figure above. However, as noted earlier, the required reinitialization of both the nonlinear ODE and the sensitivity systems at each sample instant will cause that the runtime also will increase by increasing the number of sampling points N on a fixed horizon length t_f since the solver starts over again with small step-sizes after each reinitialization. By not doing the reinitialization the runtime for algorithm 6.7 was observed to be constant and not dependent on the length of the sampling interval T_s. Omitting the reinitializations will of course yield an incorrect algorithm and produce an inexact impulse response matrix but this shows that the increase in runtime shown for algorithm 6.7 in figure 6.11 should be attributed to reinitialization of the ODEs. The theoretical runtime for algorithm 6.8 was earlier found to be $\mathcal{O}(N^2T_s) = \mathcal{O}(Nt_f)$ and from this one might expect a linear growth in runtime with t_f kept constant. The same issue also occurs here, namely that reinitialization of the ODEs also results in the simulated runtime not appearing as $\mathcal{O}(N)$. Again, omitting these reinitializations resulted in the measured runtime to appear as $\mathcal{O}(N)$ when t_f is kept constant. Despite that the runtimes for these algorithms does not appear to be equal to the theoretical bounds, they will satisfy definition 2. Under the assumption that the solver has a minimum step-size and thus, there exists an upper bound on the amount of time required to integrate across an interval, these theoretical upper bounds for runtime will still hold.

Figure 6.12: Runtime - CVODES integration, t_f, T_s variable, N constant



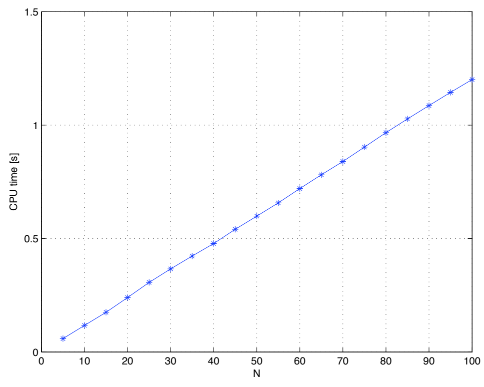
(a) Algorithm 6.7



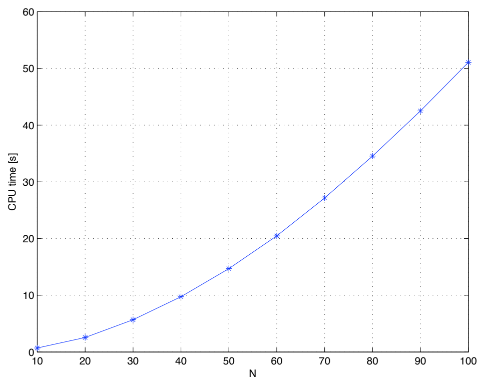
(b) Algorithm 6.8 - Alternative method

Also in figure 6.12 the observed runtime is not entirely as expected from the theoretical considerations. The runtimes for algorithm 6.7 and 6.8 are both expected to be $\mathcal{O}(t_f) = \mathcal{O}(T_s)$ when N is kept constant. Also here, the runtime is affected by reinitializations. One might argue that the reason for the runtime not to appear as linear here is that when t_f and thereby T_s is increased the solver takes longer steps at the end of a sample interval compared to when T_s is very small. In other words, the effect of reinitializing the ODEs will have more impact on the runtime for shorter sample intervals.

Figure 6.13: Runtime - CVODES integration, t_f, N variable, T_s constant



(a) Algorithm 6.7



(b) Algorithm 6.8 - Alternative method

The results shown in figure 6.13 are consistent with considerations made earlier. Runtime for algorithm 6.7 is $\mathcal{O}(N)$ when T_s is kept constant. In this case, the number of reinitializations also increase by increasing t_f and N and since T_s is kept constant the effect discussed above when N is kept constant will not occur here which is the reason for the runtime still appearing as linear. The same can be said about algorithm 6.8 whose runtime is $\mathcal{O}(N^2)$ during which T_s is kept constant.

Observe that when the alternative formulation in algorithm 6.8 is used for integration of sensitivities, the runtime becomes very large compared to the other approach in algorithm 6.7. The reactor benchmark example used here has fixed number of states $N_x = 4$ and thus it is not favored by algorithm 6.8. The alternative approach would probably prove itself better for a system with many states compared to inputs. This is to be investigated next by considering a high dimensional approximation of an infinite dimensional partial differential equation (PDE). Another lesson learned from implementing these algorithms with the Van de Vusse reactor model is that runtime for integration of sensitivities is large compared to runtime for algorithm 6.5, 6.6 or 6.10 which builds the impulse response matrix from these sensitivities. When structure of the impulse response matrix is not exploited, also observe that runtime for building Ξ could easily be in the same order of magnitude as the sensitivity integration. An important note to make here is therefore that this approach should never be used as it does not exploit the special structure of this matrix and does a lot of the same matrix multiplications more than one time.

The heat equation

In the previous section algorithm 6.8 did not prove itself very efficient for the Van de Vusse reactor with $N_x = 4$. In this section a finite dimensional approximation of the heat equation is used as a benchmarking example to investigate the effect of having a large number of states compared to inputs.

Consider the heat equation in one dimension

$$\frac{\partial \Psi}{\partial t}(x,t) = \frac{\partial^2 \Psi}{\partial x^2}(x,t),\ x \in [0,1]\,,\ t \geq 0.$$

$\psi(x,t)$ is the temperature distribution in the medium which is assumed to have length 1. Moreover, assume that the temperature at the end of the medium is fixed to $\Psi(1,t) = 0$, and that the temperature on the other end can be directly controlled, i.e. $\psi(0,t) = u(t)$. This system is infinite dimensional, but it can be approximated by a finite dimensional system of high dimension by partitioning the medium in finite intervals of length Δ, each of which will be assigned a state variable.

Define

$$\psi_i = \Psi(i\Delta),\ \Delta = 1\ /(N_x+1),\ i = 0,\dots,N_x+1.$$

Note that ψ_0 and ψ_{N_x+1} are given by the boundary conditions. The PDE can be discretized by approximating the term $\frac{\partial \Psi}{\partial x^2}$ by a second order central finite difference approximation. Consider the Taylor series

$$\psi_{i+1}(t) \quad = \quad \psi_i(t) + \frac{\partial \Psi}{\partial x}(i\Delta,t)\Delta + \frac{\partial^2 \Psi}{\partial x^2}(i\Delta,t)\Delta^2 + \frac{\partial^3 \Psi}{\partial x^3}(i\Delta,t)\Delta^3 + \dots, \quad (6.33)$$

$$\psi_{i-1}(t) \quad = \quad \psi_i(t) - \frac{\partial \Psi}{\partial x}(i\Delta,t)\Delta + \frac{\partial^2 \Psi}{\partial x^2}(i\Delta,t)\Delta^2 - \frac{\partial^3 \Psi}{\partial x^3}(i\Delta,t)\Delta^3 + \dots. \quad (6.34)$$

Adding (6.33) and (6.34) and rearranging yields

$$\frac{\partial^2 \Psi}{\partial x^2}(i\Delta, t) = \frac{1}{\Delta^2}\left(\psi_{i+1}(t) - 2\psi_i(t) + \psi_{i-1}(t)\right) + \mathcal{O}(\Delta^2).$$

By using this approximation and neglecting higher order terms we can write

$$\dot{\psi}_i(t) \approx \frac{1}{\Delta^2}\left(\psi_{i+1}(t) - 2\psi_i(t) + \psi_{i-1}(t)\right)\ i = 1, \dots N_x.$$

This equation can be transformed to state space form (recall that $\psi_0(t) = u(t)$ and $\psi_{N_x+1} = 0$)

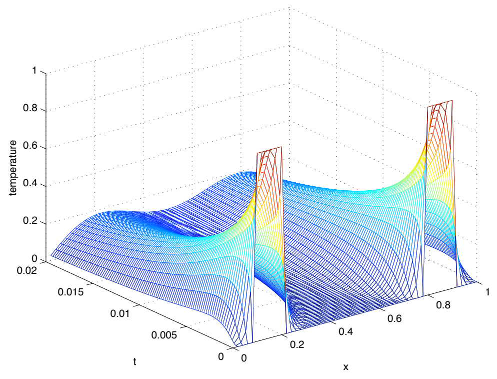
$$\begin{pmatrix} \dot{\psi}_1(t) \\ \dot{\psi}_2(t) \\ \dot{\psi}_3(t) \\ \vdots \\ \dot{\psi}_{N_x}(t) \end{pmatrix} \approx \underbrace{\frac{1}{\Delta^2}\begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix}}_{A} \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \\ \psi_3(t) \\ \vdots \\ \psi_{N_x}(t) \end{pmatrix} + \underbrace{\frac{1}{\Delta^2}\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{B} u(t).$$

Table 6.8: Organization of files for simulation of the heat equation

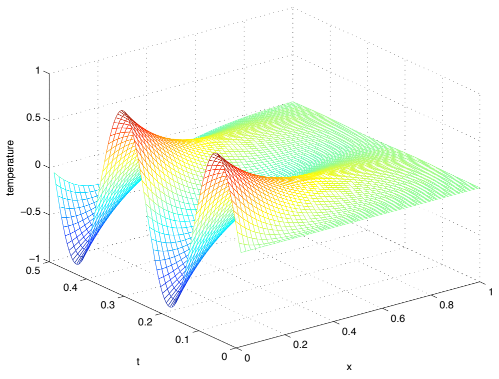
File	Purpose
`cvodes_pde.m`	This is the main script which implements algorithm 6.7 and invokes the CVODES solver routine to solve the nonlinear ODE and the sensitivity systems. Then the impulse response matrix is computed by algorithm 6.5 and algorithm 6.6.
`cvodes_pde_alt.m`	This is the main script which implements algorithm 6.8 and invokes the CVODES solver routine to solve the nonlinear ODE and the sensitivity systems. Then the impulse response matrix is computed by algorithm 6.10.
`cvodes_pde_f.m`	Provides the nonlinear model.
`cvodes_pde_J.m`	Provides Jacobian of the nonlinear mode.
`cvodes_pde_fS.m`	Provides right hand side of the sensitivity systems (6.25) and (6.27).
`cvodes_pde_fS_alt.m`	Provides right hand side of the sensitivity system (6.30).

The zero input response with a concentrated initial temperature profile in two locations and the zero state response with sinusoidal input is shown in figure 6.14. The number of states for these simulations was set to $N_x = 50$.

Figure 6.14: Heat equation - Solution



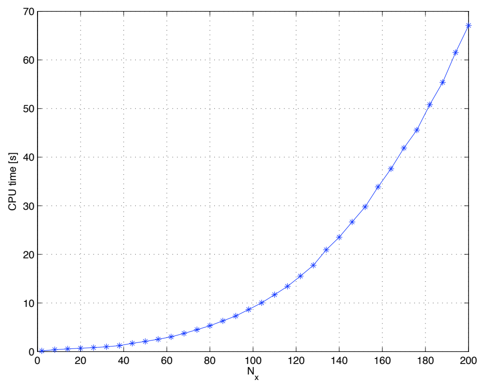
(a) Zero input response



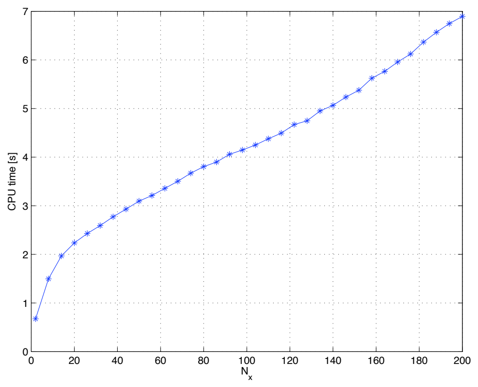
(b) Zero state response

Runtime simulations

Figure 6.15: Runtime - CVODES integration



(a) Algorithm 6.7



(b) Algorithm 6.8 - Alternative method

As seen from figure 6.15 where runtimes are shown for different number of state variables N_x, algorithm 6.8 may outperform algorithm 6.7 when the dimension of the sate vector is large compared to the dimension of the control vector. The dimension of the sensitivity ODE is $(N_x + N_u)N_x$ for algorithm 6.7 and N_xN_u for algorithm 6.8. It is expected that the runtime will somehow be related to the dimension of the sensitivity ODE which increases quadratically with N_x for algorithm 6.7 but only linearly with N_x for algorithm 6.8.

Even though this example may be extreme in terms of number of state variables compared to the number of control variables it proves that algorithm 6.8 may outperform algorithm 6.7 for certain types of systems.

6.7 Concluding remarks

In this section different approaches for calculating the objective function gradient and the impulse response matrix have been developed. Some theoretical bounds on runtimes were given for each algorithm. Efficiency of the different algorithms was compared by implementing suitable benchmark problems.

The results from these simulations are not to be considered as a definite answer to which algorithm that is best suited. Hopefully some general issues that need to be considered when choosing an approach for a specific example has been highlighted.

The main concern in the discussion above has been how to minimize computation time and it is assumed that internal memory is not a limitation. In applications of NMPC the information may fit well in the internal memory, but this may not be true if these algorithms are to be used in for example simulations of reservoirs or any other large scale model.

We have demonstrated that calculation of the objective function gradient can be done very efficiently by the adjoint method. Calculation of the impulse response matrix using adjoints can can benefit from removing output constraints on parts of the horizon. However much of the potential of adjoints is lost since one adjoint system must be solved for each point on the horizon.

We also developed some algorithms for integration of sensitivities for continuous-discrete models. Simulations demonstrated that the alternative method may in some cases be a better approach for system models where N_x is large as the dimension of the sensitivity ODE is reduced significantly. However, this comes at the expense of increasing the total integration length.

7 Efficiently handling output constraints using adjoint gradient calculation

In the previous section, methods and algorithms for calculating the objective function gradient and the impulse response matrix have been discussed. As demonstrated, obtaining this matrix can be very expensive since it involves time-consuming simulations. The impulse response matrix is the constraint gradient information needed by SQP type optimization algorithms when constraints on the outputs are present. As already discussed, there are several issues to consider and choices to make when choosing a method for computing this matrix.

Compared to obtaining the full impulse response matrix calculation of $\nabla_u J$ by the adjoint method will be very efficient since only the sensitivity of a scalar function with respect to a large number of parameters is of interest. Motivated by this fact, it would be desirable if the optimization algorithm did not require calculation of the full impulse response matrix, but only the sensitivity of a scalar function or functional, still with output constraints present. In this section the main concern will be different approaches for NMPC optimization where constraint gradient information is not required and thus adjoint gradient calculation will be extremely efficient compared to the other available approaches. This section continues the presentation of the more general optimization methods in section 3 in the context of NMPC.

7.1 Interior point methods

Barrier and interior point methods were briefly introduced in section 3.5. These methods ensure that the inequality constraints are satisfied by adding a logarithmic penalty term which tends to infinity at the barrier. Stating the NMPC problem this way would lead to an unconstrained problem provided that the problem is posed by single shooting removing the system model as equality constraint.

Barrier function methods for MPC and gradient recentered self-concordant barrier functions are discussed in [23] and [19].

Let the self-concordant barrier function for the outputs $B(t)$ be defined by

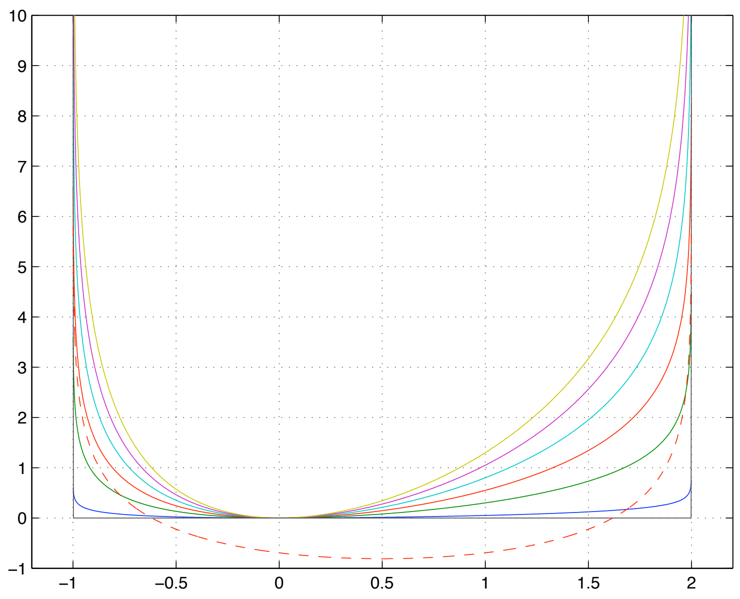
$$B(t) = -\ln(t_{max} - t) - \ln(-t_{min} + t),$$

and the gradient recentered self-concordant barrier function $\bar{B}(t)$ be defined as

$$\bar{B}(t) = B(t) - B(0) - \left[\nabla_t B(0)\right]^T t.$$

The functions $B(t)$ (dashed red) and $\mu\bar{B}(t)$ for different values of μ are illustrated in figure 7.1 where $t_{min} = -1$ and $t_{max} = 2$ together with the exact indicator function (3.13) which is zero everywhere except from on the boundary.

Figure 7.1: Barrier functions



Observe that the minimum of $B(t)$ is located at $t = 0.5$ while the recentered barrier function $\mu\bar{B}(t)$ has its minimum at the origin. This is desirable when the objective function also achieves its minimum at the origin. Not recentering the barrier functions could lead to a bias that results in the steady state solution not converging to the origin [23]. It is also possible to recenter the barrier function around a trajectory, if an objective function is on the form (2.1) is used.

If the feasible set G is defined by a number of m constraints, each of them defined by G_i so that the resulting feasible set is the intersection between them $G = \bigcap_{i=1}^{m} G_i$, one may define the barrier function F for G as $F = \sum_{i=1}^{m} F_i$ where F_i are the barrier functions for G_i [19].

Using this approach, the original NMPC optimization problem (2.2)-(2.7) may then instead be posed as

$$\min\ J = \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N \tag{7.1}$$

$$+\mu\sum_{k=0}^{N-1}\sum_{i=1}^{N_u}\bar{B}(u_k[i]) + \mu\sum_{k=1}^{N-1}\sum_{i=1}^{N_z}\bar{B}(z_k[i]), \tag{7.2}$$

subject to

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, t_k), && (7.3)\\ z_k &= g(x_k, u_k, t_k), && (7.4)\\ x_0 &= x(t_0). && (7.5) \end{aligned}$$

Posing this problem with only u_k as the free optimization variables it is an unconstrained optimization problem where the adjoint method could be employed efficiently for gradient calculation.

As shown in figure 7.1, the function $\mu\bar{B}(t)$ approaches the exact indicator function as $\mu \to 0$. However, the optimization problem becomes arbitrarily ill conditioned as $\mu \to 0$. The problem is that the objective becomes extremely nonlinear at the boundary. This is the reason why interior point methods fell out of favor for decades before again gaining popularity. Modern interior point methods take precautions to deal with this ill conditioning.

[23] argues that it might not be necessary to let μ tend to zero (thus the importance of recentering the barrier functions). It has been demonstrated that freezing μ to a value much greater than zero results in a controller that will be more cautious near the constraint boundary and will provide a more smooth transition between the active and inactive constraints [23]. It was also demonstrated that the trajectories away from the boundary are similar to the exact solution.

Early interior point methods did not use slack variables and states the problem on the form (3.14)-(3.15) in contrast to more recent methods where the initial point may be infeasible. A problem on the form (7.1)-(7.5) will require a feasible initial starting point. In NMPC applications, feedback is incorporated by solving the optimization problem over and over when new measurements become available. If the plant is subject to large disturbances, these measurements may deviate from the predicted trajectory as illustrated in figure (2.2). This deviation can result in a shifting of the predicted initial condition and thereby may result in that the initial starting point is infeasible.

In NMPC it is also desirable to be able to specify soft constraints by stating the problem as in (2.8)-(2.13). This formulation includes slack variables ϵ and will also require constraint gradients in the optimization.

As mentioned, early interior point methods require a feasible initial starting point and iterates in the optimization will always be feasible [20]. More recent methods can be provided with an infeasible starting point, but may be designed such that once a feasible iterate is generated, all subsequent iterates will be feasible. In online applications with real time requirements, feasibility of the iterates may be a desirable property as the optimization may be stopped before the actual optimum is reached.

7.2 Constraint lumping

In section 6, we argued that the adjoint method would benefit from not enforcing constraints on all the parts of the horizon as this reduces the number of output variables to which the sensitivity with respect to the control variables is desired. Another way to make use of the efficiency of adjoints is to reduce the number of constraints by instead of removing them, lumping them together. If all the output constraints can be specified in a scalar function, its gradient with respect to the control variables can be obtained in a similar way as for the objective function.

The idea is that all the path constraints also will be satisfied by instead satisfying the scalar function

$$C = \sum_{k=1}^{N-1} \sum_{i=1}^{N_z} [\max(0, z_k[i] - z_{max}[i]) + \max(0, z_{min}[i] - z_k[i])] \leq 0. \tag{7.6}$$

Since $C : \mathbb{R}^{N-1} \times \mathbb{R}^{N_z} \to \mathbb{R}$, $\nabla_u C$ can be calculated efficiently using the adjoint method. Only one forward simulation and two reverse simulations (one for C and one for J) are needed.

A disadvantage with this method is that the $\max$ function is not differentiable, and therefore it is not desirable to use in gradient based optimization techniques [20, 1].

Note that the $\max$ function can also be expressed as

$$\max(x, 0) = \int_{-\infty}^{x} \sigma(\tau) d\tau, \tag{7.7}$$

where $\sigma(\tau)$ is the unit step function

$$\sigma(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}. \tag{7.8}$$

By instead using the sigmoid function

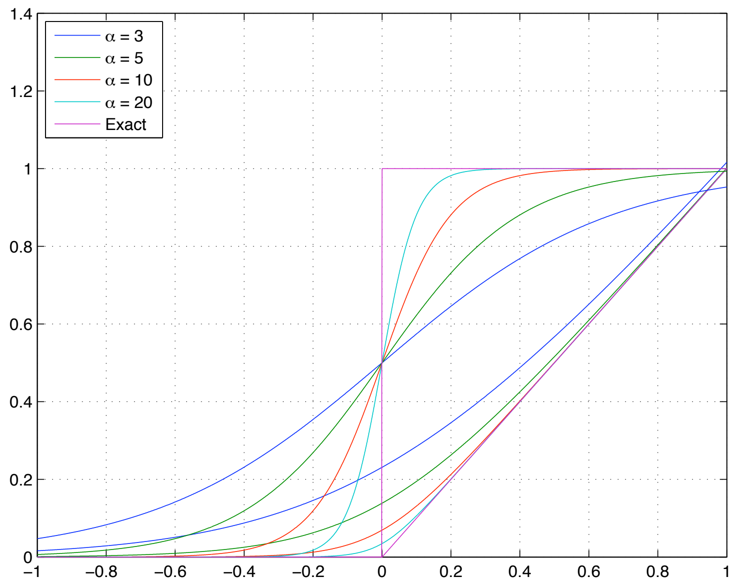
$$s(x, \alpha) = \left[1 + e^{-\alpha x}\right],^{-1} \tag{7.9}$$

as an approximation to the unit step function in (7.7) the following differentiable smooth approximation of the $\max$ function is obtained

$$\max(x, 0) \approx x + \frac{1}{\alpha} \ln(1 + e^{-\alpha x}). \tag{7.10}$$

α is a design parameter to be chosen. One should also probably consider using different weighting for each of the elements in the output vector. Choosing α large results in a more exact, but also less smooth approximation. Figure 7.2 shows the exact $\max$ function and its derivative along with approximations for different values of α.

Figure 7.2: Approximations of $\max(x, 0)$ and $\sigma(x)$



This lumping scheme comes at the expense of some ill conditioning. In [2] they apply an SQP algorithm with a combined trust region and line search method to cope with this. Trust region methods are known to be more robust against ill conditioning, while line search methods are cheaper since they do not require repeated solution of the quadratic subproblems. In the combined trust region and line search framework used in [2], the trust region radius is for that reason adjusted less often. Hopefully this reduces the total number of quadratic subproblems while still being able to handle the ill conditioning.

7.3 Penalty methods

Strategies to add the inequality constraints to the objective function are discussed in section 3.3 about penalty methods and in section 3.4 where augmented Lagrangian methods are briefly discussed. By using a l_1 penalty function for the output constraints, the NMPC optimization problem can be posed as

$$\begin{aligned}\min\ J &= \frac{1}{2}\sum_{k=0}^{N-1}[z_k^T Q z_k + u_k^T R u_k] + \frac{1}{2}x_N^T P x_N \\ &\quad + \mu \sum_{k=1}^{N}\sum_{i=1}^{N_z}[\max(z_{min}[i] - z_k[i], 0) + \max(-z_{max}[i] + z_k[i], 0)], \quad (7.11)\end{aligned}$$

subject to

$$
\begin{aligned}
x_{k+1} &= f(x_k, u_k), && (7.12)\\
z_k &= g(x_k, u_k), && (7.13)\\
x_0 &= x(t_0), && (7.14)\\
u_{min} \leq u_k &\leq u_{max}. && (7.15)
\end{aligned}
$$

This problem is not unconstrained since there are constraints on the control variables, but these constraints are much easier to handle. As pointed out in section 3.3, the l_1 penalty function is exact provided that $\mu > \|\lambda\|_\infty$. Choosing μ too large would result in an ill conditioned problem, and choosing μ too small would not satisfy $\mu > \|\lambda\|_\infty$ resulting in that constraints may be violated. As also pointed out earlier, choosing μ too small could also result in the problem being unbounded from below. However, this is not an issue here due to positive (semi)definiteness of P, Q and R. Nevertheless, the iterates may still diverge from the optimal solution, meaning that they are not useful in the sense of progressing towards the optimum when μ is chosen too small.

Similar as for constraint lumping an unavoidable drawback with the $\max$ function is indifferentiability and ill conditioning of the objective function. Smooth approximations like for instance (7.10) should also be used here.

Another alternative may be to use a smoother penalty function like for example the quadratic formulation in section 3.3. The quadratic penalty function will not be exact, resulting in that constraints may be violated even though there exists a feasible solution to the hard constrained problem.

7.4 Simulation example

In this section the model of the Van de Vusse reactor presented in section 6.6 will be used as a test example to investigate if an exact penalty method can be used in the presence of output constraints, still allowing for efficient gradient computation by the adjoint method.

Consider the following objective functional including a l_1 penalty function for constraints on the output variables

$$
\begin{aligned}
J &= \frac{1}{2}\int_{t_0}^{t_f} \left[(z(t)-z^r(t))^T Q (z(t)-z^r(t)) + (u(t)-u^r(t))^T R (u(t)-u^r(t))\right] dt \qquad (7.16)\\
&\quad + \mu \left(\sum_{i=1}^{N_z} \mu_s[i] \int_{t_0}^{t_f} [\max(z_{min}[i]-z(t)[i],0) + \max(-z_{max}[i]+z(t)[i],0)] dt \right).
\end{aligned}
$$

μ_s is a vector intended for scaling.

The adjoint equations for a functional on this form are derived in appendix A-1 using calculus of variations. These equations are also specialized to the case of zero order hold parameterization of the control signal. Refer to the procedure in the end of appendix A-1 which evaluates J during the forward integration phase and $\nabla_u J$ during the reverse integration phase by integration of quadratures.

By comparison of (7.16) and (9.1)

$$\begin{aligned}
\varphi(z(t),u(t)) &= \frac{1}{2}\left[(z(t)-z^r(t))^T Q(z(t)-z^r(t)) + (u(t)-u^r(t))^T R(u(t)-u^r(t))\right] \\
&\quad +\mu\left(\sum_{i=1}^{N_z}\mu_s[i][\max(z_{min}[i]-z(t)[i],0)+\max(-z_{max}[i]+z(t)[i],0)]\right), \\
\nu(x(t_f)) &= 0.
\end{aligned}$$

The derivatives are given by

$$\begin{aligned}
\frac{\partial\varphi(z(t),u(t))}{\partial z(t)} &= (z(t)-z^r(t))^T Q + \mu\left(\sum_{i=1}^{N_z}\mu_s[i][\sigma(z_{min}[i]-z(t)[i],0)+\sigma(-z_{max}[i]+z(t)[i],0)]\right), \\
\frac{\partial\varphi(z(t),u(t))}{\partial u(t)} &= (u(t)-u^r(t))^T R.
\end{aligned}$$

$\varphi(z(t),u(t))$ is not differentiable, and for that reason not suitable to use with derivative based optimization techniques. This is a well known issue for exact penalty methods and instead we use the smooth approximations (7.9) and (7.10) for $\sigma(x)$ and the $\max$ function, respectively.

The procedure in the end of appendix A-1 was implemented using CVODES as solver routine. As noted earlier CVODES uses variable step size methods. When asking for the solution at a sampling instant CVODES will return the solution at that time instant by using interpolation techniques. The internal time variable has however continued past that particular sampling instant such that reinitialization will be needed in order to change the control input at the right time. This approach was implemented in the algorithms earlier where sensitivities were obtained by forward analysis, and causes no trouble apart from loosing some speed since essentially a new ODE is solved for every sampling interval.

When using the adjoint sensitivity capabilities in CVODES this matter will cause some more trouble. The problem is that solver reinitializations are not allowed during the forward integration phase since the solution obtained here is also needed during the backward solve. In order to reproduce the sequence of steps used in the forward solve CVODES relies on a checkpointing algorithm. This is impossible to do if the solver at some time is reinitialized. Another problem that makes this nontrivial is discontinuities in the right hand side function due to the zero order hold parameterization of the control signal.

The first problem was handled by not doing the reinitializations, deliberately letting the solver integrate past a sample instant without altering the control input. To minimize the error, the maximum step size was set to a small fraction of the length of a sample interval preventing the integration running to far into the next sampling interval with the control input from the previous sampling interval. Since setting a maximum step size will slow the algorithm down and affect runtime severely, a runtime comparison with forward techniques is not attempted here. Discontinuities due to piecewise constant controls were handled by simply by integrating over them. Explicitly time dependent discontinuities are known to be easier to handle than those which are implicitly time dependent through the state vector. In the latter case, a root finding algorithm would be needed.

The reference trajectory is a ramp transition from set point 1 to set point 2 where c_B and T are the controlled variables.

	Set point 1	Set point 2
u_1	$8.256\ \frac{1}{h}$	$18.037\ \frac{1}{h}$
u_2	$-6.239\ \frac{MJ}{h}$	$-4.556\ \frac{MJ}{h}$
c_B	$740\ \frac{mol}{m^3}$	$960\ \frac{mol}{m^3}$
T	$87\ °\mathrm{C}$	$106\ °\mathrm{C}$

The constraints

$$\begin{aligned} c_{Bmax} &= 800\ \frac{mol}{m^3}, \\ T_{max} &= 100\ °\mathrm{C}, \end{aligned}$$

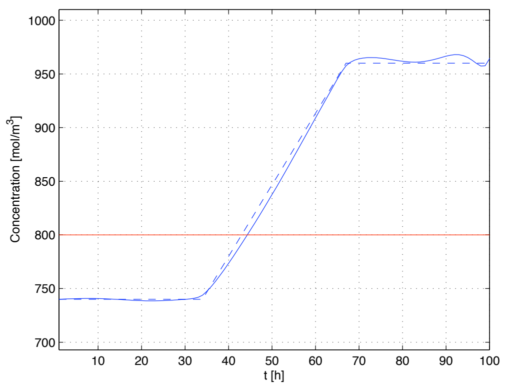
are imposed on the controlled variables resulting in that the optimum will be located outside the feasible region. Trial and error resulted in the following objective function parameters

$$Q = \begin{pmatrix} 0.1 & 0 \\ 0 & 10 \end{pmatrix},\ R = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix},\ \mu_s = \begin{bmatrix} 1 & 5 \end{bmatrix}^T.$$

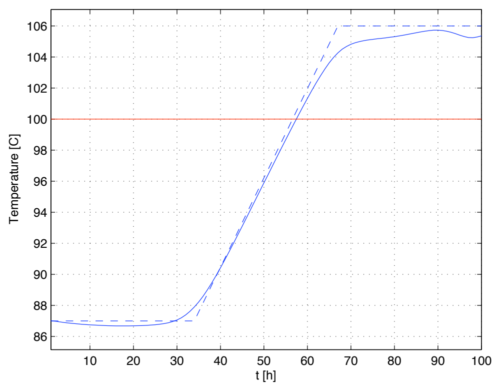
Table 7.1: Organization of files for NMPC optimization using l_1 penalty method and adjoints

File	Purpose
`nmpc_penalty.m`	This is the main script which implements adjoint gradient calculation and NMPC optimization using a l_1 penalty method for the output constraints.
`cvodes_vdv_f.m`	Provides the nonlinear model.
`cvodes_vdv_fB.m`	Provides right hand side of adjoint system.
`cvodes_vdv_J.m`	Provides Jacobian of the nonlinear model.
`cvodes_vdv_JB.m`	Provides Jacobian of adjoint system.
`cvodes_vdv_q.m`	Provides right hand side function for evaluating J using integration of quadratures.
`cvodes_vdv_qB.m`	Provides right hand side function for evaluating $\nabla_u J$ using reverse integration of quadratures.
`sigmoid.m`	Smooth approximation of $\max$ function.
`sigmoid_z.m`	Smooth approximation of step function.
`l1_penalty.m`	l_1 penalty function.
`l1_penalty_z.m`	Derivative of l_1 penalty function.
`plot_fig.m`	Plot solution and constraints.

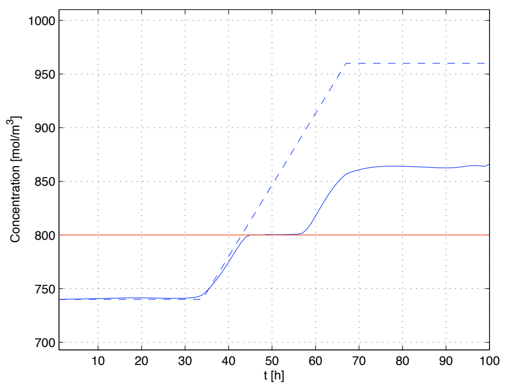
Figure 7.3: NMPC optimization with different penalty weights



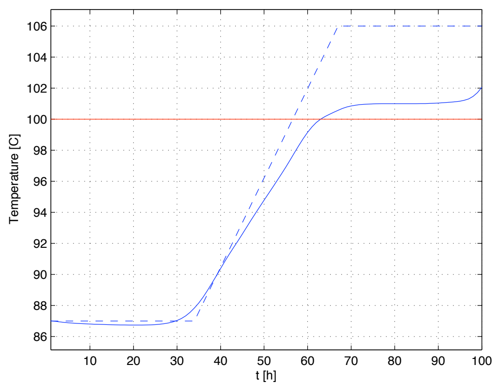
(a) $c_B, \mu = 0$



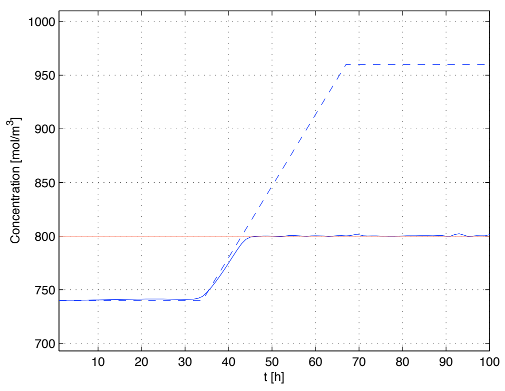
(b) $T, \mu = 0$



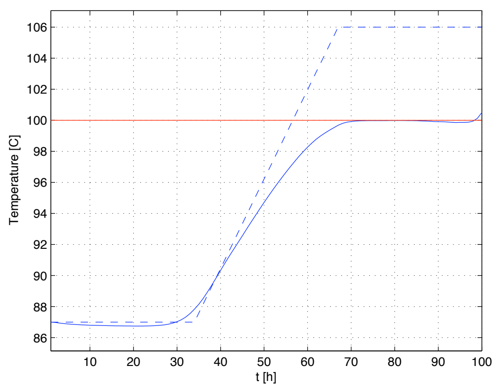
(c) $c_B, \mu = 10$



(d) $T, \mu = 10$



(e) $c_B, \mu = 20$



(f) $T, \mu = 20$

As seen from figure 7.3, the constraints are finally resolved as they are penalized more by increasing μ. When increasing μ the optimal solution from the previous value of μ was used as initial value. By doing this, the constraints are penalized increasingly as the iterates move towards the optimum, which makes sense intuitively.

Starting with a too large value of μ resulted in the algorithm not being able to produce a good solution. The high nonlinearities in the model equations introduce nonconvexity in the problem through implicit nonlinear equality constraints. Intuitively speaking, starting with a too ambitious value of μ constrains the problem more and limits the number of paths the iterates are allowed to take when progressing towards the solution.

The results above demonstrate that an exact penalty method can be used to cope with output constraints using efficient adjoint techniques for gradient calculation. However, the problem seemed hard to tune and not very robust. Even though the optimization in NMPC typically has ha hot start from the previous iteration, finding the right value for μ online is not a trivial issue. The goal here is only to demonstrate feasibility of this method, not to necessarily point out as better alternative than the other approaches with explicit inequality constraints using forward or finite difference techniques to obtain derivatives.

As noted earlier, the Van de Vusse reactor is known to be a hard problem which makes it suitable for comparative evaluation of methods. The exact penalty approach was also implemented with a much simpler nonlinear discrete time example. This problem was much easier to handle, and the exact penalty method worked remarkably well for this simple test case. The interested reader may run `nmpc_penalty_disc.m` for a demonstration. Calculating the full impulse response matrix for a discrete system involves the same matrix multiplications as calculating $\nabla_u J$ using the forward method. Due to this fact figure 6.4 is representative in terms of runtime comparison for one gradient evaluation. While the adjoint method can be employed very efficiently using a l_1 penalty method, the forward method has the advantage of having the full impulse response matrix available. While evaluation of $\nabla_u J$ using the adjoint method is much faster this method also needs to update the penalty parameter μ wisely in order to make useful progress. If one can satisfy all constraints and achieve the optimum using only a couple of iterations adjusting μ, figure 6.4 suggests that the adjoint l_1 penalty method will outperform every other method using forward techniques to obtain gradient information.

For both these examples using a smooth approximations of the $\max$ function and its derivative was a necessity in order for the optimization algorithm to produce useful iterates and for the algorithm to work.

Another good point to make about penalty and barrier methods is that an objective functional as for instance (7.16) will have the advantage that constraints can be satisfied exactly for the whole continuous time output trajectory (not only at the sampling instants).

8 Summary and final remarks

Through this thesis, the main focus is how to efficiently compute derivative information for use in NMPC optimization. This is essential for a fast NMPC algorithm as computation of derivative information can be very time-consuming.

In the first sections NMPC is briefly introduced and notation is defined. The next section introduces some general optimization techniques. Section 4 is about general techniques for derivative calculation. The reader is introduced to the concept of forward and the reverse mode of automatic differentiation. Section 5 is about sensitivity analysis of mathematical models, which is very closely related to NMPC. The point to be made in these two sections is that forward methods are efficient when sensitivity of a high dimensional function with respect to a few number of parameters is to be found. For the opposite case, when the sensitivity of a low dimensional function with respect to a large number of parameters is required, adjoint or reverse techniques are most efficient.

Section 6 discusses calculation of derivatives in the context of NMPC. First, algorithms for obtaining the objective function gradient $\nabla_u J$ are derived. As one might expect from the discussion in section 4 and 5, in terms of runtime the adjoint method is superior for obtaining $\nabla_u J$. However, most optimization algorithms like for instance SQP also need constraint gradient information which is given by the impulse response matrix of the linearized system model. Constraint gradients for the control variables are trivial to obtain. If the problem is posed by single shooting, obtaining the constraint gradient for the output variables is however not trivial. Since these constraints typically are enforced on every point on the horizon, they are sometimes referred to as path constraints. These output variables are coupled through nonlinear dynamics from the control variables. Hence, the sensitivity of a large number of output variables with respect to a large number of control variables is needed. In this case the forward method may be just as efficient as the adjoint method. In addition the adjoint method requires more memory as the state trajectory from the forward solve is needed in the reverse solve. Roughly speaking, obtaining the output constraint gradient at one time instant on the horizon requires one additional reverse simulation when using the adjoint method. For that reason, not enforcing output constraints at every point on the horizon will result in the adjoint method being faster than the forward method, but also requiring more memory.

Next, continuous-discrete models are discussed. Integration of sensitivity equations are needed in order to discretize the system. Two different methods are presented. The first method is best suited for systems with rather few states. The alternative formulation favors systems with a large number of states where the dimension of the sensitivity equations are reduced at the expense that the total integration length is more than one time the horizon length.

All the different methods in section 6 are presented as pseudo code. In the end of this section these algorithms are implemented with some benchmark examples to demonstrate and compare theoretical bounds for runtimes with practical experience from simulations.

As noted earlier, when there are no constraints on the output variables, derivative information can be obtained very efficiently using the adjoint method. The next question is at which extent adjoint gradient calculation still can be employed efficiently in the presence of output constraints. In section 7 some methods presented in the literature are reviewed. The key element in all these methods is to use an optimization algorithm that does not specify the output constraints explicitly. Instead they are satisfied by including a penalty term in the objective function or by lumping the constraints into one equivalent scalar constraint. This approach will introduce some new challenges like for instance how to update the penalty parameter or the barrier weighting parameter in penalty and interior point methods. However, this will allow for very efficient gradient calculation by the adjoint method as only sensitivity of a low dimensional function is needed. The hope is that the overall performance will increase despite of the introduction of some additional difficulties to overcome. In the end of this section NMPC optimization for the Van de Vusse reaction scheme using a l_1 penalty method was implemented. There were some difficulties implementing this example using the adjoint capabilities in CVODES due to discontinuities in the right hand side function and the fact that CVODES uses variable step size methods. For that reason a complete comparable evaluation of forward and adjoint methods was not attempted. Nevertheless, implementing a l_1 penalty function to cope with output constraints still demonstrates the feasibility of the method on a hard control problem.

The various methods for calculation of derivatives all have advantages and disadvantages. Choosing the right method will be highly problem dependent. The purpose is not to point out one method as better than the other, but to highlight some of the issues one might want to consider when choosing the appropriate method for a specific problem.

Through this thesis we have demonstrated that adjoint methods can be extremely efficient, but they also introduce some other severe difficulties. The author's impression is that despite the efficiency of adjoints for these type of problems, solving sensitivity equations using forward techniques can in many cases be a much better alternative since structure can be exploited by solving these in parallel with the nonlinear ODE, minimizing overhead. For the Van de Vusse reactor example, the time required to build the impulse response matrix from these sensitivities was almost negligible. However, this may not be true for systems with higher dimension of the state vector, but in that case, the alternative method for integration of sensitivities may be a better approach in which these potentially time-consuming matrix multiplications already will be incorporated in the integration of the sensitivity equations.

Through this thesis we have presented different combinations of optimization algorithms and methods for calculating derivatives. The final remark to be made is that nothing seems to come for free as the nice properties of these methods sometimes conflict. The use of adjoints for NMPC requires both tailor made optimization algorithms and ODE solvers and will probably be subject for further research.

9 Appendix

A-1 Continuous time adjoint sensitivity analysis

In the following, we will derive the adjoint equations for a functional on the form

$$J = \int_{t_0}^{t_f} \varphi(z(t), u(t))dt + \nu(x(t_f)). \tag{9.1}$$

Then these equations will be specialized to a procedure for evaluating J and $\nabla_u J$ when the control signal is parameterized using zero order hold.

Define the Lagrangian

$$\begin{aligned}
\mathcal{L} =& J - \int_{t_0}^{t_f} \left[\lambda^T(t)(\dot{x}(t) - f(x(t), u(t)))\right] dt \\
=& \int_{t_0}^{t_f} \left[\varphi(z(t), u(t)) - \lambda^T(t)(\dot{x}(t) - f(x(t), u(t)))\right] dt + \nu(x(t_f)).
\end{aligned}$$

Since $\dot{x}(t) = f(x(t), u(t))$, $\mathcal{L} = J$ and the first variation of J with respect to u is given by

$$\begin{aligned}
\delta J =& \int_{t_0}^{t_f} \frac{\partial\varphi(z(t), u(t))}{\partial z(t)} \left(\frac{\partial z(t)}{\partial x(t)}\delta x(t) + \frac{\partial z(t)}{\partial u(t)}\delta u(t)\right) dt + \\
& \int_{t_0}^{t_f} \left(\frac{\partial\varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial u(t)}\right) \delta u(t)dt - \int_{t_0}^{t_f} \lambda^T(t)\delta\dot{x}(t)dt + \\
& \int_{t_0}^{t_f} \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial x(t)}\delta x(t)dt + \int_{t_0}^{t_f} (\dot{x}(t) - f(x(t), u(t)))^T\delta\lambda(t)dt + \frac{\partial\nu(x(t_f))}{\partial x(t_f)}\delta x(t_f).
\end{aligned}$$

By integration by parts

$$\int_{t_0}^{t_f} \lambda^T(t)\delta\dot{x}(t)dt = \lambda^T(t_f)\delta x(t_f) - \lambda^T(t_0)\delta x(t_0) - \int_{t_0}^{t_f} \dot{\lambda}^T(t)\delta x(t)dt.$$

By inserting this relationship and noting that $\int_{t_0}^{t_f}(\dot{x}(t) - f(x(t), u(t)))^T \delta\lambda(t)dt = 0$ in order to comply with the model we get

$$\begin{aligned}
\delta J \;=\; & \int_{t_0}^{t_f} \frac{\partial\varphi(z(t), u(t))}{\partial z(t)} \left(\frac{\partial z(t)}{\partial x(t)}\delta x(t) + \frac{\partial z(t)}{\partial u(t)}\delta u(t)\right) dt + \\
& \int_{t_0}^{t_f} \left(\frac{\partial\varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial u(t)}\right) \delta u(t)dt - \lambda^T(t_f)\delta x(t_f) + \lambda^T(t_0)\delta x(t_0) + \\
& \int_{t_0}^{t_f} \dot{\lambda}^T(t)\delta x(t)dt + \int_{t_0}^{t_f} \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial x(t)}\delta x(t)dt + \frac{\partial\nu(x(t_f))}{\partial x(t_f)}\delta x(t_f).
\end{aligned}$$

Note that $\lambda^T(t_0)\delta x(t_0) = 0$ since the initial condition is fixed. Rearrangement yields

$$\begin{aligned}
\delta J \;=\; & \int_{t_0}^{t_f} \left(\frac{\partial\varphi(z(t), u(t))}{\partial z(t)}\frac{\partial z(t)}{\partial x(t)} + \dot{\lambda}^T(t) + \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial x(t)}\right) \delta x(t)dt + \left(\frac{\partial\nu(x(t_f))}{\partial x(t_f)} - \lambda^T(t_f)\right) \delta x(t_f) \\
& + \int_{t_0}^{t_f} \left(\frac{\partial\varphi(z(t), u(t))}{\partial z(t)}\frac{\partial z(t)}{\partial u(t)} + \frac{\partial\varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial u(t)}\right) \delta u(t)dt.
\end{aligned}$$

By letting $\lambda(t)$ fulfill

$$\dot{\lambda}^T(t) = -\frac{\partial\varphi(z(t), u(t))}{\partial z(t)}\frac{\partial z(t)}{\partial x(t)} - \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial x(t)}, \tag{9.2}$$

with the final condition

$$\lambda^T(t_f) = \frac{\partial\nu(x(t_f))}{\partial x(t_f)}. \tag{9.3}$$

the first variation of J with respect to u is nothing but

$$\delta J = \int_{t_0}^{t_f} \left(\frac{\partial\varphi(z(t), u(t))}{\partial z(t)}\frac{\partial z(t)}{\partial u(t)} + \frac{\partial\varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t)\frac{\partial f(x(t), u(t))}{\partial u(t)}\right) \delta u(t)dt. \tag{9.4}$$

Further, if the control signal is parameterized using zero order hold, we can rewrite (9.4) as

$$\delta J = \sum_{k=0}^{N-1} \int_{t_k}^{t_{k+1}} \left(\frac{\partial \varphi(z(t), u(t))}{\partial z(t)} \frac{\partial z(t)}{\partial u_k} + \frac{\partial \varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t) \frac{\partial f(x(t), u_k)}{\partial u_k} \right) dt \delta u_k. \quad (9.5)$$

By definition

$$\delta J = \sum_{k=0}^{N-1} \frac{\partial J}{\partial u_k} \delta u_k. \quad (9.6)$$

Then, by comparison of (9.5) and (9.6)

$$\frac{\partial J}{\partial u_k} = \int_{t_k}^{t_{k+1}} \left(\frac{\partial \varphi(z(t), u(t))}{\partial z(t)} \frac{\partial z(t)}{\partial u_k} + \frac{\partial \varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t) \frac{\partial f(x(t), u_k)}{\partial u_k} \right) dt.$$

A complete algorithm for evaluating J and calculating $\nabla_u J$ using the adjoint method when the control signal is parameterized as piecewise constant will involve the following steps

1. For each sample interval $t_k \leq t < t_{k+1},\ k \in \{0, \ldots, N-1\}$, solve $\dot{x}(t) = f(x(t), u_k)$ by forward integration. Evaluate J by integration of quadratures.
2. Initialize $\lambda^T(t_f) = \frac{\partial \nu(x(t_f))}{\partial x(t_f)}$.
3. For each sample interval $t_k \leq t < t_{k+1},\ k \in \{N-1, \ldots, 0\}$, solve

$$\dot{\lambda}^T(t) = -\frac{\partial \varphi(z(t), u(t))}{\partial z(t)} \frac{\partial z(t)}{\partial x(t)} - \lambda^T(t) \frac{\partial f(x(t), u_k)}{\partial x(t)}$$

by reverse integration and evaluate

$$\frac{\partial J}{\partial u_k} = \int_{t_k}^{t_{k+1}} \left(\frac{\partial \varphi(z(t), u(t))}{\partial z(t)} \frac{\partial z(t)}{\partial u_k} + \frac{\partial \varphi(z(t), u(t))}{\partial u(t)} + \lambda^T(t) \frac{\partial f(x(t), u_k)}{\partial u_k} \right) dt$$

by reverse integration of quadratures.

A-2 Hardware and software

All the simulations have been performed on a laptop computer. Hardware and software is listed below.

Table 9.1: Simulation environment

Software	Hardware
Operating System: Mac OS X 10.5.7	CPU: Intel Core 2 Duo @ 2.4 GHz
MATLAB R2008a	Memory: 4 GB DDR2 SDRAM @ 667 MHz
SUNDIALS 2.3.0	Bus Speed: 800 MHz

A-3 Software

Software	Folder	Main script
Runtime simulations of algorithms for calculating $\nabla_u J$	`obj_grad`	`obj_grad.m`
Runtime simulations of algorithms for calculating Ξ using integration of sensitivity equations	`cvodes_vdv`	`cvodes_vdv.m`
Runtime simulations of algorithms for calculating Ξ using alternative method for integration of sensitivities	`cvodes_vdv`	`cvodes_vdv_alt.m`
NMPC optimization for Van de Vusse reactor using l_1 penalty method to handle output constraints. $\nabla_u J$ is calculated using continuous time adjoint equations	`nmpc_penalty_vdv`	`nmpc_penalty_vdv.m`
NMPC optimization for discrete system using l_1 penalty method to handle output constraints. $\nabla_u J$ is calculated using continuous time adjoint equations	`nmpc_penalty_disc`	`nmpc_penalty_disc.m`
Computation of Ξ using finite differences and solver routine `ode15s` in Matlab	`finite_diff_vdv`	`finite_diff_vdv.m`
Steady state solution of Van de Vusse reactor	`vdv_ss`	`vdv_ss.m`

Bibliography

[1] Pallav Sarma; Wen H. Chen; Louis J. Durlofsky; Khalid Aziz. Production optimization with adjoint models under nonlinear control-state path inequality constraints. *SPE Reservoir Evaluation and Engineering*, 11(2):326–339, 2006.

[2] K.F. Bloss, L.T. Biegler, and W.E. Schiesser. Dynamic process optimization through adjoint formulations and constraint aggregation. *Industrial and engineering chemistry research*, 38(2):421–432, 1999.

[3] H. G. Bock, M. Diehl, and E. Kostina. Sqp methods with inexact jacobians for inequality constrained optimization. Technical report, 2005.

[4] H.G. Bock, M. Diehl, E.A. Kostina, and J.P. Schlöder. Constrained optimal feedback control of systems governed by large differential algebraic equations. In L. Biegler, O. Ghattas, M. Heinkenschloss, D. Keyes, and B. van Bloemen Waanders, editors, *Real-Time and Online PDE-Constrained Optimization*. SIAM, 2006.

[5] Yang Cao, Shengtai Li, and Linda Petzold. Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software. *Journal of computational and applied mathematics*, pages 171–191, 2002.

[6] Yang Cao, Shengtai Li, Linda Petzold, and Radu Serban. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint dae system and its numerical solution. *SIAM Journal on Scientific Computing*, 24(3):1076–1089, 2002.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[8] Moritz Diehl, Hans Joachim Ferreau, and Niels Haverbeke. Efficient numerical methods for nonlinear mpc and moving horizon estimation. *Int. Workshop on Assessment and Future Directions of NMPC*, 2008.

[9] Moritz Diehl, Andrea Walther, Hans Georg Bock, and Ekaterina Kostina. An adjoint-based sqp algorithm with quasi-newton jacobian updates for inequality constrained optimization. 2005.

[10] Daniel Christopher Doublet. Optimisation of production from an oil-reservoir using augmented lagrangian methods. *PhD Thesis*, 2007.

[11] Rolf Findeisen and Frank Allgöwer. An introduction to nonlinear model predictive control. In *21st Benelux Meeting on Systems and Control, Veidhoven*, pages 1–23, 2002.

[12] Matthias Gerdts. Gradient evaluation in dae optimal control problems by sensitivity equations and adjoint equations. *Proceedings in Applied Mathematics and Mechanics*, 5(1):43–46, 2005.

[13] Alan C. Hindmarsh and Radu Serban. *User Documentation for cvodes v2.5.0*. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2006.

[14] J. B. Jørgensen. Adjoint sensitivity results for predictive control, state- and parameter-estimation with nonlinear models. 2008.

[15] Hassan K. Khalil. *nonlinear systems*. Prentice Hall, 3 edition, 2001.

[16] Shu-Qin Liu, Jianming Shi, Jichang Dong, and Shouyang Wang. A modified penalty function method for a modified penalty function method for inequality constraints minimization. 2004.

[17] J. M. Maciejowski. *Predictive Control With Constraints*. Prentice Hall, Essex, England, 2002.

[18] Kenneth R. Muske and James B. Rawlings. Model predictive control with linear models. 1993.

[19] Yurii Nesterov and Arkadii Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. SIAM, 1994.

[20] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, second edition, 2006.

[21] Linda Petzold, Shengtai Li, Yang Cao, and Radu Serban. Sensitivity analysis of differential-algebraic equations and partial differential equations. 2006.

[22] T. Utz and B. Mahn V. Hagenmeyer. Comparative evaluation of nonlinear model predictive and flatness-based two-degree-of-freedom control design in view of industrial application. *Journal of Process Control*, 17(129-141), 2007.

[23] Adrian Wills and Will P. Heath. Barrier function based model predictive control. *Automatica*, pages 1415–1422, 2004.

[24] L. Wirsching, J. Albersmeyer, P. Kuehl, M. Diehl, and H.G. Bock. An adjoint-based numerical method for fast nonlinear model predictive control. In *Proceedings of the 17th IFAC World Congress, Seoul 2008*, 2008.