NTNU

Norwegian University of
Science and Technology

Efficient optimization for Model Predictive Control in reservoir models

Jørgen Frenken Borgesen

Master of Science in Engineering Cybernetics
Submission date: June 2009
Supervisor: Bjarne Anton Foss, ITK
Co-supervisor: John Petter Jensen, StatoilHydro

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. Adjoint-based methods is an alternative since these are 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 MPC for large scale reservoir models is further studied.

Tasks:
1. Present adjoint-based methods and review central literature. The presentation shall focus on MPC, with a sequential approach, applied to reservoir models.
2. Output constraints can be detrimental to the efficiency of adjoint-based methods. Hence, optimization algorithms where output constraints can be removed, as for instance in barrier methods, or the number of output constraints are significantly reduced are approaches to exploit the efficiency of adjoint-based methods. Discuss and propose such methods.
3. Evaluate the methods above by comparing them with forward methods. This should be done on suitable reservoir examples, preferably available benchmark cases, using realistic test scenarios.

Assignment given: 12. January 2009
Supervisor: Bjarne Anton Foss, ITK

Abstract

The purpose of this thesis was to study the use of adjoint methods for gradient calculations in Model Predictive Control (MPC) applications. The goal was to find and test efficient optimization methods to use in MPC on oil reservoir models. Handling output constraints in the optimization problem has been studied closer since they deteriorate the efficiency of the MPC applications greatly.

Adjoint- and finite difference approaches for gradient calculations was tested on reservoir models to determine there efficiency on this particular type of problem. Techniques for reducing the number of output constraints was also utilized to decrease the computation time further.

The results of this study shows us that adjoint methods can decrease the computation time for reservoir simulations greatly. Combining the adjoint methods with techniques that reduces the number of output constraints can reduce the computation time even more. Adjoint methods require some more work in the modeling process, but the simulation time can be greatly reduced.

The principal conclusion is that more specialized optimization algorithms can reduce the simulation time for reservoir models.

Keywords: Gradient calculation, adjoint method, MPC, Oil reservoir

Contents

Chapter 1

Introduction

Over the years linear Model Predictive Control (MPC) has become popular due to the theoretical results provided by the academic community and the successful installations in the industry. Linear systems with linear constraints results in a linear or quadratic optimization problem that easily and effective can be solved in the MPC application. Nonlinear Model Predictive control (NMPC) on the other hand consists of nonlinear components, making the optimization problem nonlinear, which is solved with far less efficiency. NMPC has gained more interest, both in industry and in the academic community, the last years due to the desire to controlling more complex systems. Using nonlinear models that are more realistic enables operation closer to constraints and better output prediction which can results in better control performance. The optimization algorithms requires in NMPC are much more computational expensive than for linear MPC. Many of them uses gradient information to solve the problem. Calculation these gradients requires a lot of calculations using traditional methods as finite difference since the systems needs to be simulated $N_u + 1$ times, where N_u is the total number of decisions variables, which again is the number of system inputs (n_u) times the length of the optimization horizon (N_{H_u}). The computation time for large systems, such as a reservoir model, becomes unpractical, especially when the time horizon is extended.

There exists an alternative method used to do gradient calculation, the adjoint approach, which only requires 2 system simulations over the optimization horizon, one forward in time and one backward in time. Implementing this method into the NMPC optimization algorithm can increase the speed of NMPC application greatly.

This thesis will study the use of the adjoint approach in an nonlinear Model Predictive Control application utilized on a reservoir model. A NMPC application is developed and used to compare the efficiency of calculating gradients using the adjoint approach up against using traditional methods. The main focus will be on a NMPC application optimizing on a reservoir model.

This thesis start by giving some background informations about MPC and reservoir modeling in chapter 2. Chapter 3 will explain the adjoint approach to gradient calculations, discuss the advantages/disadvantages and look at how constraint handling in the optimization problem can be done. A reservoir model used in later tests is developed in chapter 5. Test cases for the reservoir model is defined in chapter 6, while the results obtained from simulations of this case is presented and discussed in chapter 7. Chapter 8 gives a general discussion of the work in this thesis and the results obtained from is. At last chapter 9 gives a conclusion and proposals to further work.

Chapter 2

Background

This section will provide background information needed to understand the rest of this thesis.

First a short introduction to the Model Predictive Control (MPC) [5, 12] strategy is presented to help readers not familiar with MPC to understand the concept and the connection to optimization. Then some general optimization theory is presented.

2.1 Model Predictive Control

In this section there will be given a short introduction to model predictive control (MPC). MPC is one of the most popular techniques used for advanced system control. It uses an internal model to calculate a sequence of optimal inputs that minimizes a function that describes the desired behavior of the system. The future input sequence is re-optimized at each time-step for a finite period of time into the future, and it is therefor also called Receding Horizon Control (RHC) [1]. Compared to traditional linear control, MPC has a mayor advantage as it handles constraints on inputs, outputs and internal states. This way it can take saturation into account, and it is possible to operate closer to system limitation which can be utilized to increase profitability. This is one of the reasons why MPC in the process industry has gained such high popularity [12]. Another advantage the MPC has is that it is naturally multi-variable, giving the algorithm large freedom in choosing which inputs to manipulate to achieve its goal.

Figure 2.1 illustrates the MPC principle. At time $t = k$, T_p is the prediction horizon, telling how far into the future the system is to be simulated. T_u is the control horizon, telling how far into the future the inputs are changed, after this, the inputs are kept constant for the rest of the simulation.

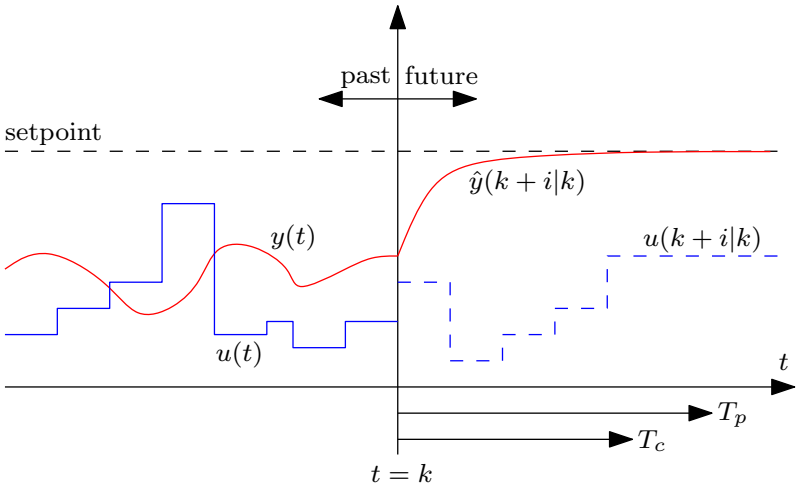


Figure 2.1: MPC principle

If the internal system model is perfect and the systems is not exposed to any noise the future inputs calculated at time $t = k$ would be the optimal solution on the whole prediction horizon, meaning that the optimization problem only needs to be solved once. This is not a robust solution as the plant model is never perfect and the system will be exposed to noise. The solution used in MPC is to re-optimize the problem with a receding horizon when new information about the system (such as measurements from the real system) is available. This introduces feedback to the system even thou the future prediction is open-loop calculations.

A simple presentation of the MPC principle is described in algorithm 2.1. System states are referred to as x, inputs as u and outputs as z.

Algorithm 2.1 Model Predictive Control

1: Define control goals in the form of an objective function (J(z,x,u))
2: Define control, state and output constraints
3: **while** MPC application is running **do**
4: Update start state. Set x^0 = measured/estimated states from the real process.
5: Find inputs u that minimize $J(z, x, u)$ subject to constraints.
6: Implement first step of optimal input and shift optimal input / optimization horizon one step forward.
7: **end while**

Linear MPC has a linear system model and linear constraints. The objective function is usually a quadratic function. The optimization problem associated with a linear MPC application then becomes a quadratic optimization problem that can be solved efficient and the computation time is deterministic [12]. Solving the a more general optimization problem where the objective function and its constraints can contain nonlinear components is far more challenging. This kind of problem is found in a nonlinear MPC application. Finding the solution by using gradient based optimization solvers on such problems is studied in this thesis. The focus will be on efficient gradient calculations.

2.1.1 Problem definition

This section will formulate the problem studied in this thesis. We will be using a first-principle nonlinear reservoir model described by continuous differential equations on the form

$$\dot{x} = f_c(x(t), u(t)), \qquad x(0) = x_0 \tag{2.1}$$

where $x(t) \in \mathcal{R}_x^n$ is the state vector and $u(t) \in \mathcal{R}_u^n$ is the input vector of the system. The output of the system is described by the algebraic

$$z = g(x(t), u(t)) \tag{2.2}$$

where $z \in \mathcal{R}^{n_z}$ is the output vector.

This model is then discretized using the numerical integration scheme *Euler's method* [7]. Giving the following discrete model, where k is the time-step.

$$x^{k+1} = x^k + \Delta t f_c(x^k, u^k) = f(x^k, u^k) \tag{2.3}$$

$$z^k = g(x^k, u^k) \tag{2.4}$$

The objective function $J \in \mathcal{R}$ is chosen to be quadratic and only containing the input- and output variables

$$J(z, u) \sum_{k=1}^{T_p-1} [(z^k)^T Q x^k] + \sum_{k=0}^{T_u-1} [(u^k)^T R u^k] + (z^{T_p})^T P z^{T_p} \tag{2.5}$$

where T_p is the prediction horizon, T_u is the control horizon, Q and R is diagonal matrices weighting system outputs and inputs while P is a diagonal matrix weighting the outputs at the end of the prediction horizon.

To complete the problem we need to add constraints.

$$c(u) \leq 0 \tag{2.6}$$
$$U_{min} \leq u^k \leq U_{max} \tag{2.7}$$
$$\tag{2.8}$$

where nonlinear constraints are introduced through (2.6) while (2.7) give simple block constraints on inputs. If we combine all these elements we get an optimization problem that can be used in a MPC application

$$\text{minimize} \qquad J(z,u) = \sum_{k=1}^{T_p-1}[(z^k)^T Q z^k] + \sum_{k=0}^{T_u-1}[(u^k)^T R u^k] + (z^{T_p})^T P z^{T_p} \tag{2.9}$$
$$\text{subject to} \tag{2.10}$$
$$x^0 = x(t_0) \tag{2.11}$$
$$x^{k+1} = f(x^k, u^k) \tag{2.12}$$
$$z^k = g(x^k, u^k) \tag{2.13}$$
$$c(u) \leq 0 \tag{2.14}$$
$$U_{min} \leq u^k \leq U_{max} \tag{2.15}$$
$$\tag{2.16}$$

This optimization problem that is going to be used in the MPC application can be solved by standard nonlinear optimization solvers.

2.2 Optimization

Optimization is the procedure of finding the best alternative for a particular situation. To do this it is important to know what the objective is, as well as what the alternatives are. Optimization problems may has multiple objectives that often conflicts. The optimization solver then has to find a middle-way. A simple example would be to create a car that runs as fast as possible and uses as little fuel as possible. The optimization problem in this case would be to find the car parts that gives a fast car with low fuel consumption. The weighting between the speed and fuel consumption is something the that has to be determined by the designer.

A optimization problem may have restrictions or constraints associated with it. In the car example above there may be an upper cost of the car, or maximum weight limit. Any optimization solver solving such problems has to take these restrictions into account and make sure that they are not violated. Optimization is used in various fields as finance (maximize profit, minimize risk),

manufacturers (efficient design and operation of products), engineers (parameter optimization) etc.[9]

To be able to do optimization we need a quantitative measure of the performance of the system, a way of telling how good the objectives are reached. This can be done by using a objective function that calculates a number representing the performance of the system. The optimization task is then to find parameters that maximizes/minimizes the objective function.

2.2.1 Choosing an optimization algorithms

There exist various optimization algorithms, each which is designed for a specific type of problem. There is no such thing as an effective general optimization algorithm. Every algorithm has strengths and weaknesses, and is designed for a specific type of problem. The main categories of optimization problems listed in order of complexity is:

- Unconstrained linear problems.
- Constrained linear problems.
- Unconstrained nonlinear problems.
- Constrained nonlinear problems.

Every problem can be put in one of these categories, and a proper solver fitting that category should be chosen. It is possible to use a constrained nonlinear solver on problems in all the above categories, but the efficiency on for instance a constrained linear problem will be far from as good as if a solver designed for that kind of problem had been used. Each of the above categories will also have subcategories with even more specialized optimization solvers.

The choice of optimization algorithm must be taken with a certain type of problem in mind. Choosing a optimization algorithm with constraint support is not necessary for an unconstrained problem and will result in lower performance. There exist many optimization solvers that are tailored for a specific *type* of problem [9], choose one that fits your problem. Tailoring a solver for a special/specific *problem* can result in great performance increase, but the development cost has to be taken into account. Throughout this thesis different optimization techniques will be studied and tested to find a combination that results in a efficient optimization solver for MPC applications used on reservoir models.

2.2.2 Optimization solver for MPC application

Any MPC controller needs a optimization solver to solve the optimization problem associated with the control applications. In the optimization theory there exists a large number of solvers which is suitable for MPC applications. MPC applications will always require an optimization solver that can handle constraints since the system equation will be present as a equality constraint. The reservoir model presented in chapter 5 is nonlinear, thus a nonlinear optimization solver is required. The optimization solver used in this thesis belongs to the Sequential Quadratic Programming (SQP) class. The SQP concept is described in section 2.2.2.

The MPC application developed in this study utilizes the function *fmincon* [13] provided by the optimization toolbox in Matlab. The function is highly customizable allowing for user-provided functions to do gradient calculations.

Sequential quadratic programming

Sequential Quadratic Programming (SQP) [4, 11] is a popular an efficient method for solving nonlinear optimization problems [16]. SQP methods solves a series of quadratic subproblems to find search directions that decreases the objective function value. Let us consider the nonlinear optimization problem

$$\min_{x} \quad f(x) \tag{2.17}$$

such that

$$c(x) \leq 0 \tag{2.18}$$

$$h(x) = 0 \tag{2.19}$$

where $f(x)$ is the objective function, x is a vector containing the decision variables, $c(x)$ is the inequality constraints and $h(x)$ is the equality constraints. The Lagrange function[12] for this problem is defined as

$$\mathcal{L}(x, \lambda, \sigma) = f(x) + \lambda^T c(x) + \sigma^T h(x) \tag{2.20}$$

where λ and σ is Lagrange multipliers. If d^k is the search direction at time k, then $x^{k+1} = x^k + d^k$. Inserting x^{k+1} into (2.20) gives $\mathcal{L}(x^{k+1}) = \mathcal{L}(x^k + d^k)$ which can be approximated by a second order Taylor expansion

$$\mathcal{L}(x^k + d) \approx \mathcal{L}(x^k) + \nabla_{x^k}\mathcal{L}(x^k)d^k + \frac{1}{2}(d^k)^T \nabla^2_{x^k}\mathcal{L}(x^k)d^k \tag{2.21}$$

The constraints can be linearized using the a first order Taylor expansion

$$c(x^{k+1}) \approx c(x^k) + \nabla_{x^k} c(x^k) d^k \tag{2.22}$$

$$h(x^{k+1}) \approx h(x^k) + \nabla_{x^k} h(x^k) d^k \tag{2.23}$$

The approximation of the objective function and the constraints gives us a local quadratic problem (2.24) - (2.26) that is repeatedly solved by the SQP solver to find new search directions d^k until a convergence criteria is reached, or the maximum number of iterations is reached (meaning no solution was found).

$$\min_{x^k} \quad \mathcal{L}(x^k) + \nabla_{x^k} \mathcal{L}(x^k) d^k + \frac{1}{2} (d^k)^T \nabla^2_{x^k} \mathcal{L}(x^k) d^k \tag{2.24}$$

such that

$$c(x^k) + \nabla_{x^k} c(x^k) d \leq 0 \tag{2.25}$$

$$h(x^k) + \nabla_{x^k} h(x^k) d = 0 \tag{2.26}$$

Chapter 3

Gradient calculation

Gradient calculation is an important and time consuming part of gradient based optimization algorithms. The gradients are used to find a search direction than decreases the objective function value. Choosing an effective method for gradient calculations can greatly reduce the computation time for the optimization problem. This chapter, together with the next one will present two methods, the adjoint method and the finite difference method, which will be compared later by simulations.

3.1 Finite differences

Finite difference methods is the most common way of calculating gradients in MPC applications. The simplest and fastest finite difference method is the one-sided finite difference method. The method is based on the definition of the derivative (3.1).

$$\frac{d}{dt}f(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{3.1}$$

The definition of the partial derivate (3.2) becomes:

$$\frac{\partial}{\partial x_i} f(x_1, ..., x_i, ...x_n) = \lim_{\Delta x_i \to 0} \frac{f(x_1, ..., x_i + \Delta x_i, ...x_n) - f(x_1, ..., x_i, ..., x_n)}{\Delta x_i} \tag{3.2}$$

The definition (3.2) requires Δx_i to approach zero, while the finite difference replaces Δx_i with ϵ which is chosen to be a number sufficiently small. The result is an approximation of the derivative:

$$\frac{\partial}{\partial x_i} f(x_1, ..., x_i, ..., x_n) \approx \frac{f(x_1, ..., x_i + \epsilon, ...x_n) - f(x_1, ..., x_i, ..., x_n)}{\epsilon} \tag{3.3}$$
$$\epsilon = \text{small number}$$

Choosing the size of ϵ is not a trivial task. A smaller ϵ will give a more accurate approximation, but if ϵ is chosen too small numerical problems can arise. If the function f is highly nonlinear a small ϵ is even more important to ensure a good approximation, while just a slightly nonlinear function will give a quite accurate result even with a moderate sized ϵ-value.

The gradient $\nabla f(x_1, ..., x_n)$ can be calculated by using the approximation (3.3) and calculate $\frac{\partial}{\partial x_i} f(x_1, ..., x_n)$ for $i = \{1, ..., n\}$. This requires one nominal evaluation of the function f and one evaluation for each of the variables to find the partial derivatives for all of them. This results in the total of $n + 1$ function evaluations to determine an approximation of the gradient.

It can be noted that there exist another method called two-sided finite difference method that uses one perturbation in each direction:

$$\frac{\partial}{\partial x_i} f(x_1, ..., x_i - \epsilon, ..., x_n) \approx \frac{f(x_1, ..., x_i + \epsilon, ...x_n) - f(x_1, ..., x_i, ..., x_n)}{2\epsilon} \tag{3.4}$$
$$\epsilon = \text{small number}$$

The two-sided finite difference method gives a better approximation than the one-sided method. The computational cost is the same for the one-sided method if only one partial derivative is calculated, but if more than one partial derivative is calculated the method requires more function evaluations. To calculate the gradient of the function the two-sided finite difference method require $2n$ function evaluations, almost twice as much as the one-sided version.

The finite difference method is well suited for problems with few decision variables, since the method requires one pertubation for each of those variables. As the number of decision variables increase, the number of pertubations and simulations that need to be done also increases.

3.1.1 Finite difference methods in MPC applications

The one-sided finite differences method is the most commonly used method to calculate gradients in a MPC application. The two-sided method is too time consuming, at least for on-line applications where the computation time of the optimization in the MPC application may be critical to keep the system stable. Throughout the rest of this thesis finite difference methods will mean the one-sided version of the method.

The optimization problem solver in a MPC application requires the gradients of the function to minimize, namely the objective function. The objective function is generally a function of the system states, the system outputs and the system inputs. The system states and outputs is coupled to the system inputs through the system. The system inputs is the decision variables of the optimization problem, meaning that the gradient of each of the input variables must be calculated. We are interested in:

$$\frac{d}{du_k} J(\mathbf{z}, \mathbf{x}, \mathbf{u}), \qquad k = 1, ..., N \tag{3.5}$$

,where J is the objective function, $\mathbf{u} = \{u_1, ..., u_N\}$, u_k is the input vector at time k, $\mathbf{z}$ is the system outputs and $\mathbf{x}$ is the vector containing all states on the optimization horizon.

Since the objective function value is depending on system states and outputs we need to simulate the system over the prediction horizon to obtain these. This is a computational costly process that must be repeated for each decision variable over the whole control horizon since perturbing a decision variable changes the states and outputs. Since the system is causal, simulation for the perturbation of a particular input variable can start at the time-step where this variable affects the system. All the previous states/outputs are equal to those calculated in the nominal simulation where none of the inputs were perturbed. Algorithm 3.1 presents the procedure.

In the algorithm belowthe gradient for one particular set of inputs are calculated. An iterative optimization solver (like SQP) will try to change the set of inputs (in a clever way) at each iteration to decrease the value of the objective

Algorithm 3.1 Calculate gradients using finite difference method

Require: u - nominal input vector
Require: ϵ - perturbation value

{Nominal simulation}
for k = 1:N-1 **do**
  $x^{k+1} = f^k(x^k, u^k)$
  $z^k = g^k(x^k, u^k)$
end for
$J = calculateObjectiveValue(z, x, u)$

{Perturb inputs and simulate}
for k = 1:N **do**
  $\tilde{x} = x$
  $\tilde{z} = z$
  for i = 1:N_{H_u} **do**
    $\tilde{u} = u$
    $\tilde{u}_i^k = \tilde{u}_i^k + \epsilon$
    for j = k:N **do**
      $\tilde{x}^{j+1} = f^j(\tilde{x}^j, \tilde{u}^j)$
      $\tilde{z}^j = g^j(\tilde{x}^j, \tilde{u}^j)$
    end for
    $\tilde{J} = calculateObjectiveValue(\tilde{z}, \tilde{x}, \tilde{u})$
    $\frac{\partial J}{\partial u_i^k} = \frac{\tilde{J} - J}{\epsilon}$
  end for
end for

function. meaning that Algorithm 3.1 may be executed a large number of times before a sufficient solution is found. The computation time for the algorithm grows fast with increasing prediction horizon and input variables due to the 3 nested for-loops. The number of simulation steps required to calculate the gradients would be

$$\frac{n_u N_{H_u}(N_{H_u}+1)}{2} \tag{3.6}$$

where n_u is the number of inputs and N_{H_u} is the control horizon. (3.6 can be explained by the fact that we need one nominal simulation in addition to $N_{H_u} n_u$ simulation with one input pertubated. This needs to be done for each timestep over the control horizon N_{H_u}, but causality can be exploited giving us only half that many simulations on average. This results (3.6) which is a quadratic function with respect to the control horizon. The quadratic computation time with respect to the control horizon was confirmed by simulations in [3].

3.2 Adjoint method

The adjoint method for calculating gradients are the main topic of this thesis and the study is therefor placed in its own chapter, chapter 4.

Chapter 4

Gradient calculations using an adjoint approach

This section presents the adjoint approach [10, 6, 14] to gradients calculations in an NMPC application. The adjoint approach uses information about the dynamic system, that is controlled by the NMPC application, to calculate the gradients. By utilizing this information instead at looking at the system as a black box the adjoint approach is able to do gradient calculations more efficient than traditional methods.

4.1 Objective function gradients

Lets start with a simplified version of the optimization problem (2.9) presented in section 2.1.1. The simplified problem (4.1-4.2) consist of a general objective function with the system equation as a equality constraint to preserve the system dynamics during the optimization.

$$\min_u J(\mathbf{u}) = \sum_{k=0}^{N-1} G_k(x_k, u_k) + H_N(x_N) \tag{4.1}$$

subject to

$$x_{k+1} - f_k(x_k, u_k) = 0, \qquad i = 0, \ldots, N-1 \tag{4.2}$$

The function G_k is general function that can be nonlinear and changing with time, H_N is a function penalizing the state at the end of the horizon. Lets continue by defining the Lagrange function:

$$\mathcal{L}(x,u,\lambda) = \sum_{k=0}^{N-1}[G_k(x_k,u_k)] + H_N(x_N) - \sum_{k=0}^{N-1}[\lambda_{k+1}^T(x_{k+1} - f_k(x_k,u_k))] \quad (4.3)$$

, where λ_k is called the Lagrange multipliers. If we rearrange the Lagrange function to group the term concerning the first time-step, those in between and the last time-step the Lagrange function can be presented as:

$$\begin{aligned}\mathcal{L}(x,u,\lambda) = {} & G_0(x_0,u_0) + \lambda_1^T f_0(x_0,u_0) \\ & + \sum_{k=1}^{N-1}[G_k(x_k,u_k) + \lambda_{k+1}^T f_k(x_k,u_k) - \lambda_k^T x_k] \\ & + H_N(x_N) - \lambda_N^T x_N \end{aligned} \quad (4.4)$$

This representation of the Lagrange function makes it easy to define the first order Karush-Kuhn-Tucker (KKT) [9, 8] conditions for the problem:

$$\nabla_{\lambda_{k+1}}\mathcal{L} = f_k(x_k,u_k) - x_{k+1} = 0 \quad (4.5)$$

$$\nabla_{x_N}\mathcal{L} = \nabla_{x_N}H_N(x_N) - \lambda_N = 0 \quad (4.6)$$

$$\nabla_{x_k}\mathcal{L} = \nabla_{x_k}G_k(x_k,u_k) + \nabla_{x_k}f_k(x_k,u_k)\lambda_{k+1} - \lambda_k = 0 \quad (4.7)$$

$$\nabla_{u_k}\mathcal{L} = \nabla_{u_k}G_k(x_k,u_k) + \nabla_{u_k}f_k(x_k,u_k)\lambda_{k+1} = 0 \quad (4.8)$$

The first order KKT conditions are necessary conditions for an optimal solution of the optimization problem. A closer look on the KKT conditions shows us that the first KKT condition (4.5) is the system equation, preserving the system dynamics. Obviously this equation has to be satisfied at the optimum.

Simulating the system along a nominal input trajectory $u = \{u_0, ..., u_{N-1}\}$ will satisfy the equality constraints and give us a nominal value for the states $x = \{x_0, ..., x_{N-1}\}$.

To Lagrange multipliers can be obtained using equation (4.6) and (4.7). Starting with (4.6), we find the last Lagrange multiplier:

$$\lambda_N = \nabla_{x_N}H_N(x_N) \quad (4.9)$$

To obtain the other Lagrange multipliers and the gradients for the objective function we continue from the last time-step and iterate backward using equation (4.7) and (4.8). If the input vector u does not optimize the objective function ,then the last KKT condition (4.8) is not satisfied, instead it provide us with the gradients of the objective function with respect to the inputs.

$$\nabla_{u_k}\mathcal{L} = \nabla_{u_k}G_k(x_k,u_k) + \nabla_{u_k}f_k(x_k,u_k)\lambda_{k+1} \quad (4.10)$$

This approach results in the following psudo-code for gradient calculations:

Algorithm 4.1 Calculate gradients using adjoint method

Require: u - nominal input vector

{Forward simulation}
for k = 1:N-1 **do**
  $x_{k+1} = f_k(x_k, u_k)$
end for

{Backward simulation}
$\lambda_N \leftarrow \nabla_{x_k} f_k(x_k, u_k)$
$k \leftarrow N$
while k $>$ 0 **do**
  $\nabla_{u_k} J = \nabla_{u_k} G_k(x_k, u_k) + \nabla_{u_k} f_k(x_k, u_k)^T \lambda_{k+1}$
  $\lambda_k = \nabla_{x_k} G_k(x_k, u_k) + \nabla_{x_k} f_k(x_k, u_k) \lambda_{k+1}$
  $k = k - 1$
end while

The adjoint method is well suited for problems with many decision variables and few outputs. The minimizaition of a objective function in an MPC application would be such a problem. Since the method only require 2 simulations over the control horizon regardless of the number of inputs the computation time will increase linearly with the length of the control horizon. This was confirmed by simulations in [3].

4.2 Constraints

Adjoint based methods for gradient calculations are efficient for problems with many decision variables and few outputs, however, the present of output constraints may deteriorate the efficiency. Output constraints are typically used in MPC. This section discusses how to handle output constraint in an SQP algorithm (used in an MPC application) that uses an adjoint approach for gradient calculations.

There are several techniques that may reduce the effect of output constraints in optimization problems solved with adjoint gradient calculations. The techniques will be studied further, and tested on the reservoir model described in chapter 5.

4.2.1 Softening constraints

The terms soft constraints are used to describe a set of constraints that are allowed to be violated occasionally, but this is not desired. Soft constraints are incorporated into the optimization problem by adding a cost to the objective function whenever the constraints are violated.

In MPC applications, soft constraints are often used on system states and outputs. The states and outputs are coupled to the inputs through the system, making the prediction of them uncertain due to model error and noise in the real process. Since the states and outputs cant be calculated exact, and the fact that most real systems do not naturally have hard constraints, states and output makes them good candidates for soft constraints. Constraints on inputs are often hard by nature, mainly because of saturation in actuators. Inputs are under direct control by the MPC application, making the handling of them simpler.

Softening constraints is the process of transforming hard constraints into soft constraints. This is done by removing the original hard constraints and adding a penalty function [9] to the objective function. The penalty function is zero as long as the original constraints are adhered, but as soon as the constraints are violated the penalty function will be greater than zero, thus increasing the value of the objective function. Consider the following constrained optimization problem

$$\min_x f(x) \tag{4.11}$$

subject to

$$c_i(x) \leq 0, \quad i \in \mathcal{I} \tag{4.12}$$

which can be transformed to a unconstrained optimization problem by softening the constraints and adding a penalty function to the original objective function $f(x)$.

$$\min_x \mathcal{Q}(x, \sigma) = f(x) + \sigma_i \sum_{i \in \mathcal{I}} g(c_i(x)) \tag{4.13}$$

where σ_i is the penalty coefficients and $g()$ is the penalty function.

The choice of penalty function depends on the desired behavior and properties of the resulting optimization problem. Penalty functions can be either be exact or not. Exact penalty functions gives the softened problem the same solution as the original one as long as the original problem is feasible [9]. Inexact penalty functions often results in different solution that the original problem, no matter how large the penalty coefficients are chosen. The drawback of softening constraints are that introducing penalty functions makes the minimization of the new objective function harder. One problem is that penalty functions may

not be smooth (this is almost always true for exact penalty functions), resulting in undefined differentials for some values of x. Another problem is that even for smooth penalty function the resulting objective function can become less smooth by choosing large penalty coefficients, this is quite critical as large penalty coefficients are required for the solution of the unconstrained problem to be a good approximation of the original solution. [9] proposes a algorithm (4.2) for inexact penalty functions that increases the penalty coefficients repeatedly:

Algorithm 4.2 Penalty method

1: Given σ_0, a nonnegative sequence $\{\tau_k\}$ with $t_k \to 0$, and a starting point x_0^s;
2: **for** $k = 0, 1, 2, ...$ **do**
3: Find an approximate minimizer x_k of $\mathcal{Q}(\cdot; \sigma_k)$, starting at x_k^s,
4: and terminating when $\|\nabla_x \mathcal{Q}(x; \sigma_k)\| \leq \tau_k$;
5: **if** final convergence test satisfied **then**
6: **stop** with approximate solution x_k;
7: **end if**
8: Choose new penalty parameter $\sigma_{k+1} > \sigma_k$;
9: Choose new starting point x_{k+1}^s;
10: **end for**

4.2.2 Lumping constraints

Nonlinear problem solvers needs the gradient of all active constraints to find a feasible search direction. This can be time-consuming to calculate for many constraints. One way to reduce the calculations of these gradients is to lump the constraints, creating one equivalent constraint for all the active constraints. Various lumping schemes are available, with the following being commonly used for optimal control problems:

$$\sum_{n=0}^{N-1} \max[c_n(x_n, u_n), 0] = 0 \tag{4.14}$$

$$\sum_{n=0}^{N-1} (\max[c_n(x_n, u_n), 0])^2 = 0 \tag{4.15}$$

The following approach is proposed in [2] and used in the General Purpose Research Simulator (GPRS) developed at Stanford University. The proposed

lumping scheme is a smooth differentiable approximation of the *max* function. The constraints are assumed to be on the form $c_n(x_k, u_k) \leq 0$. The *max* function (4.14) is the integral of the unit step function (4.16):

$$\sigma(y) = \begin{pmatrix} 1 & y > 0 \\ 0 & y \leq 0 \end{pmatrix} \tag{4.16}$$

$$\max(x, 0) = \int_{-\infty}^{x} \sigma(y)dy \tag{4.17}$$

The unit step function can be approximated with (4.18):

$$s(y, \alpha) = \{1 + exp(-\alpha y)\}^{-1} \qquad \forall \alpha > 0 \tag{4.18}$$

By substituting $s(y, \alpha)$ for $\sigma(y)$ in (4.17), we get an approximation for the *max* function:

$$p(x, \alpha) = \int_{-\infty}^{x} s(y, \alpha)dy = x + \frac{1}{\alpha} log\{1 + exp(-\alpha x)\} \tag{4.19}$$

One important property for this approximation is that it can be derived infinitely many times, meaning that $p(c_n(x_n, u_n), \alpha)$ is differentiable as many times as $c_n(x_n, u_n)$. Some other relevant properties of $p(x, \alpha)$ are:

$$p(x, \alpha) > \max\{x, 0\} \qquad \forall x \in R \tag{4.20}$$

$$\lim_{|x| \to \infty} \{p(x, \alpha) - \max(x, 0)\} = 0 \qquad \forall \alpha > 0 \tag{4.21}$$

$$\lim_{\alpha \to \infty} \{p(x, \alpha) - \max(x, 0)\} = 0 \qquad \forall x \in R \tag{4.22}$$

The above properties makes the function $p(x, \alpha)$ suitable as an approximation to the *max* function that can be used for constraint lumping.

We can now define the equivalent constraint by lumping the constraints and use the *max* approximation. The new equivalent constraint C is now defined as:

$$C = \sum_{n=0}^{N-1} \left[c_n + \frac{1}{\alpha} log\{1 + exp(-\alpha c_n)\} \right] \leq \frac{log2}{\alpha} \qquad \forall \alpha > 0 \tag{4.23}$$

The new equivalent constraint is less than or equal to $\frac{log2}{\alpha}$, the reason is that $p(0, \alpha) = \frac{log2}{\alpha}$ and $p(c_n, \alpha)$ increases monotonically with c_n. Solving the optimization problem using the lumped constraint C can produce a search direction that is feasible for the lumped constraint, but not for the original constraints. We then have an infeasible search direction with respect to the original constraints.

To illustrate the method a simple example is provided. Figure 4.1 illustrates the original problem. There are two decision variables u_1 and u_2, and two constraints c_1 and c_2. The figure shows the constraints as green lines, objective function contours as red dotted lines, objective function gradient as a purple arrow and constraint gradients as blue arrows. Figure 4.1 shows the lumped constraint as a black line. As we see the lumped constraint are less restrictive than the original constraints, giving a infeasible search direction (red arrow). This can be handled by projecting the search direction onto the constraint, as described later.

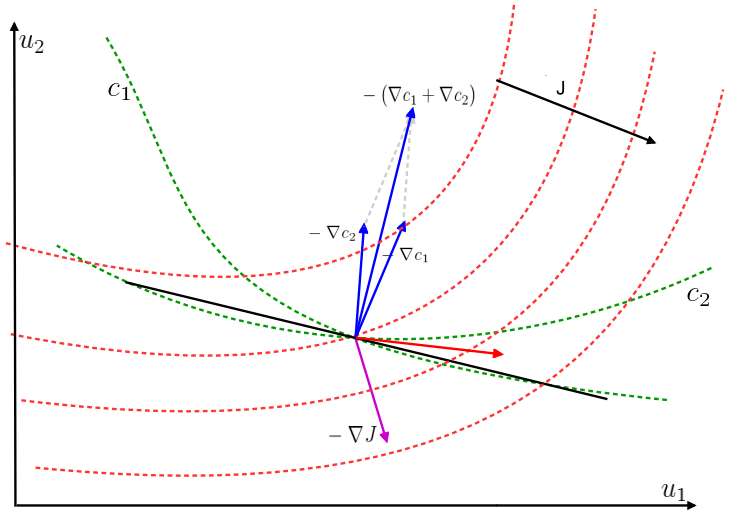


Figure 4.1: Illustration of constraint lumping.

The fact that the search direction calculated with the lumped constraint may actual be infeasible with respect to the original constraints is a problem. To solve this problem a feasible line search can be employed. The basic idea behind this algorithm is to implement the constraints into the forward model and modify the search direction if any constraints are violated. The modification consist of projecting the infeasible search direction onto the violated constraints during line search by solving the violated constraints during the forward simulation. This is equivalent to performing a curved line search along the violated constraints. If a constraint is violated at a given time-step the search direction has to be modified and the time-step has to be simulated again. The projection in the case of the simple example described above is illustrated in figure 4.2. The search direction red arrow) is projected onto the violated constraint c_2.

To modify the search direction we need some knowledge about which inputs that can be modified to satisfy the violated constraints. It is not clear if there exists a "best" strategy to select these inputs, so knowledge about the dynamic system and previous iterations must be used to find suitable inputs. An ex-

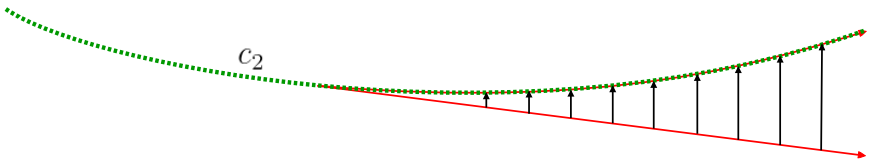


Figure 4.2: Projecting search direction onto constraint.

ample where system knowledge is used can be found in [2] , where the solution is applied to a reservoir model with a maximum total water injection constraint and a maximum total liquid production constraint. An approximated linear connection between the injection rate of an injector and the bottom hole pressure is used in an iterative manner to satisfy the violated constraints.

Lumping constraints in an MPC application

In an MPC application the number of constraints for the optimization solver is the sum of the number of constraints at each time-step over the prediction horizon. Lumping constraints can reduce the number of constraint by a large factor. A optimization problem with lumped constraints results in a system where the number of inputs n_u is larger than the number of constraints n_c.

$$n_u > n_c \tag{4.24}$$

If all constraints are lumped into one single constraint, then $n_c = 1$. The process of calculating the constraint gradients is a problem quite similar to calculating the objective function gradient. Both problems have a "large" number of decision variables that results in a final value (the objective function value and the lumped constraint). In fact, they have the same inputs (system inputs). The system equation (4.2) is the same for both problems and it has to be respected at all times for both problems. The similarities can be exploited during the gradient calculations.

The gradients of the lumped constraint can be calculated using an adjoint approach, reducing the calculation time compared to finite difference methods. The number of decision variables is greater than the number of outputs making the use of an adjoint approach a great candidate for gradient calculations. Using an adjoint approach to calculate both the objective function gradients and the constraint gradients have certain advantages that can be exploited to increase the computation time even more. Some of the terms calculated during the computation of objective function gradients can be stored an reused when the constraints gradients is calculated. States (x) and outputs (z) from the

forward simulation is expensive to calculate and should be stored and reused when calculating constraint gradients. Since the system equation is the same for both problems the gradients $\nabla_{x_k} f$ and $\nabla_{u_k} f$ will be the same and can be stored an reused.

Chapter 5

Reservoir modeling

This chapter will present the notation and reservoir model used in this thesis. The reservoir model is based on the black oil model. Many simplifications has been done to reduce the time spent on developing the model. This gives more focus on the important aspects of this thesis, namely efficient gradient calculation.

5.1 Introduction

The reservoir model used in this thesis is based on the black oil model. The black oil model [17, 18] is used to determine the pressure, volume and temperature of the different phases(oil, water and gas) in the reservoir. The model developed here is highly simplified to make the derivation of analytic gradients easier. The model only consider a reservoir with two phases, oil and water. The simplified model is far from realistic, but is still contains the properties that are important for this thesis. The system still contains a large number of states, it is nonlinear and the number of inputs and outputs can be chosen. It is also possible to introduce constraints on both inputs and outputs. Because of similar properties, the work presented here regarding gradient calculations should be transferable to reservoir models that are more realistic.

Simplifications and assumptions

The analytic derivation of gradients is time consuming. To reduce the complexity of this process, the model presented here has many simplifications and assumptions. The model will be less realistic, but the main focus of this thesis is efficient gradient calculations, making the simplification defensible.

Fluids are assumed to be incompressible. The mixed fluid (oil and water) is assumed to have a constant density regardless of the ratio between them. Water pressure and oil pressure are equal.

5.2 Derivation of model equations

The reservoir model is discretized in space. The reservoir is divided into a large grid of cells, each cell having two states, pressure p and water saturation S.

Most of the variables used in the rest of this chapter will have sub- and superscripts. Subscripts are used for position indices and other describing information, while superscript are used for time indices. The variable k will be used as time index.

Each cell in the grid interacts with four neighbor cells (north, east, south and west). The flow between two cells are determined by the pressure difference. The notation will be as followed. The current cell has index i, while the neighbor cells has the index j, where j belongs to a set $\mathcal{N}_i$ that represents neighbor indices ($j \in \mathcal{N}_i$). Figure 5.2 explains the relationship between the i and j indices. The figure shows cell i in the middle with 4 neighbor cells (north, east, south and west) marked with j. Flow from neighbor cells only occurs over the thick black border.

The flow consist of water and oil, and the ratio between them is determined by the water saturation in the cell with the highest pressure. Since two cells generally do not have the same water saturation we get a instant change in saturation as the flow changes direction. To avoid numerical problems as the flow approaching zero and changes direction, a saturation on the flow is introduced. A flow between two cells require the pressure difference between those two cells to be higher than a given tolerance ϵ_p. Flow $q_{i,j}$ from cell j to i is the sum of the water flow $q_{w,i,j}$ and the oil flow $q_{o,i,j}$. The water and oil

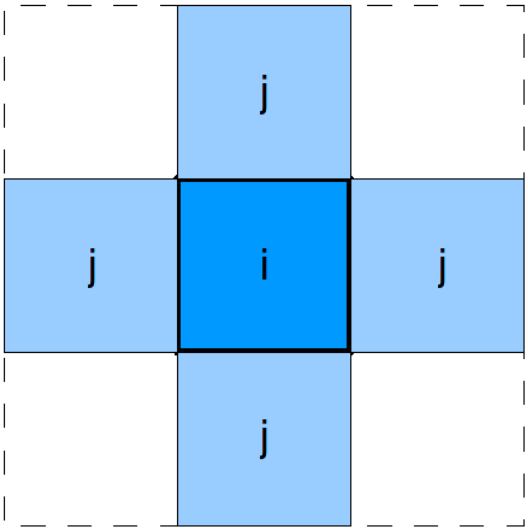


Figure 5.1: Relationship between cells

flow is described by the following equations:

$$q_{w,i,j} = \left\{ \begin{array}{ll} \eta_{i,j} S_i (p_j - p_i) & \text{if } p_j - p_i \leq -\epsilon_p \\ \eta_{i,j} S_j (p_j - p_i) & \text{if } p_j - p_i \geq \epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.1}$$

$$q_{o,i,j} = \left\{ \begin{array}{ll} \eta_{i,j} (1 - S_i)(p_j - p_i) & \text{if } p_j - p_i \leq -\epsilon_p \\ \eta_{i,j} (1 - S_j)(p_j - p_i) & \text{if } p_j - p_i \geq \epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.2}$$

,where $\eta_{i,j}$ is the permeability between cell i and j , S_x is the water saturation in cell x and p_x is the pressure in cell x.

The saturation introduced in the flow equations results in zero flow when the pressure difference is less than the ϵ_p. This can be a problem when the derivative of the equations are calculated since the optimization solver may not see any reason to change a input variable because it seems like it would not affect the system. To prevent this the derivatives are adjusted so that the derivatives is given a value even if the pressure difference is too low to generate any flow.

The flow equations (5.1)-(5.2) are derived with respect to p_i, p_j, S_i and S_j. Equations (5.3)-(5.10) shows the result.

$$\frac{\partial q_{w,i,j}}{\partial p_i^k} = \left\{ \begin{array}{ll} -\eta_{i,j} S_i & \text{if } p_j - p_i < 0 \\ -\eta_{i,j} S_j & \text{if } p_j - p_i \geq 0 \end{array} \right\} \tag{5.3}$$

$$\frac{\partial q_{o,i,j}}{\partial p_i^k} = \left\{ \begin{array}{ll} -\eta_{i,j}(1 - S_i) & \text{if } p_j - p_i < 0 \\ -\eta_{i,j}(1 - S_j) & \text{if } p_j - p_i \geq 0 \end{array} \right\} \tag{5.4}$$

The derivative for the flow with respect to p_i is modified so that it seems like a change in p_i will result in a change of flow even if we got no flow. This is done so that the optimization solver can escape from some no-flow situations.

$$\frac{\partial q_{w,i,j}}{\partial S_i} = \left\{ \begin{array}{ll} \eta_{i,j}(p_j - p_i) & \text{if } p_j - p_i \leq -\epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.5}$$

$$\frac{\partial q_{o,i,j}}{\partial S_i} = \left\{ \begin{array}{ll} -\eta_{i,j}(p_j - p_i) & \text{if } p_j - p_i \leq -\epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.6}$$

As seen from the above equations, the derivatives with respect to S_i, the water/oil flows will only be affected if we have flow going from cell i to j.

$$\frac{\partial q_{w,i,j}}{\partial p_j} = \left\{ \begin{array}{ll} \eta_{i,j} S_i & \text{if } p_j - p_i < 0 \\ \eta_{i,j} S_j & \text{if } p_j - p_i \geq 0 \end{array} \right\} \tag{5.7}$$

$$\frac{\partial q_{o,i,j}}{\partial p_j} = \left\{ \begin{array}{ll} \eta_{i,j}(1 - S_i) & \text{if } p_j - p_i < 0 \\ \eta_{i,j}(1 - S_j) & \text{if } p_j - p_i \geq 0 \end{array} \right\} \tag{5.8}$$

Derivation of the flow equations with respect to p_j results in the same behavior as for p_i. The derivatives is manipulated to make it seems like a change in the pressure p_j will result in change of flow even if the flow is zero.

$$\frac{\partial q_{w,i,j}}{\partial S_j} = \left\{ \begin{array}{ll} \eta_{i,j}(p_j - p_i) & \text{if } p_j - p_i \geq \epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.9}$$

$$s\frac{\partial q_{o,i,j}}{\partial S_j} = \left\{ \begin{array}{ll} -\eta_{i,j}(p_j - p_i) & \text{if } p_j - p_i \geq \epsilon_p \\ 0 & \text{otherwise} \end{array} \right\} \tag{5.10}$$

Changing the water saturation S_j will only affect the flow if the flow is leaving cell j.

System inputs

System inputs are the variables that can be controlled. For this model it is the flow into the reservoir and the opening on the producer valve. The flow into the system through a injector well is controlled by the variable $q_{in,i}$, while the valve opening is controlled by $u_{out,i}$, where i is the injector/producer well number. The variables is stacked into one input vector u.

$$u = \begin{bmatrix} q_{in,1} \\ \dots \\ q_{in,n_{inj}} \\ u_{out,1} \\ \dots \\ u_{out,n_{prod}} \end{bmatrix} \tag{5.11}$$

n_{inj} and n_{prod} is the number of injector and producer wells respectively. The relationship between the valve opening $u_{out,i}$ and the output of that cell is

$$q_{out,i} = \frac{p_i}{p_0} u_{out,i} \tag{5.12}$$

where p_0 is a constant.

Differential equations

The update laws for the states S and p are controlled by differential equations. We begin by finding the equation for the pressure state of each of the reservoir cells.

The pressure is defined as

$$p_i = \rho g h_i \tag{5.13}$$

where ρ the constant density of the fluid and h_i is the height in the cell, then the total volume $V_{tot,i}$ can be described by

$$V_{tot,i} = A h_i \tag{5.14}$$

$$= \frac{A}{\rho g} p_i \tag{5.15}$$

$$= \mu p_i, \qquad \mu = \frac{A}{\rho g} \tag{5.16}$$

It is easy to see that the pressure is depending on the volume in the cell,

$p_i = \frac{V_{tot,i}}{\mu}$. The time derivative then becomes

$$\begin{aligned}\dot{p}_i &= \frac{1}{\mu}\dot{V}_{tot,i} \\ &= \frac{1}{\mu}(\dot{V}_w + \dot{V}_o) \\ &= \frac{1}{\mu}(\sum_j [q_{w,i,j} + q_{o,i,j}] + q_{in,i} - q_{out,i}) \end{aligned} \tag{5.17}$$

where $q_{out,i}$ is the output through a producer well if present in cell i, otherwise it is zero, $q_{in,i}$ is the water injection through a injector well if present, otherwise it is zero. The term $q_{w,i,j}$ and $q_{o,i,j}$ is the flow of water/oil from neighbor cells with index $j \in \mathcal{N}_i$. Rearranging (5.17) results in the the final equation for $\dot{p}_i$

$$\dot{p}_i = \frac{1}{\mu}(\sum_j [q_{w,i,j} + q_{o,i,j}]) + \frac{1}{\mu}(q_{in,i} - q_{out,i}) \tag{5.18}$$

The water saturation S_i for cell i is defined by

$$S_i = \frac{V_{w,i}}{V_{tot,i}}, \qquad V_{tot,i} = V_{w,i} + V_{o,i} \tag{5.19}$$

where $V_{w,i}$, $V_{o,i}$ is the volume of water and oil in cell i. Derivation of (5.19) with respect to time

$$\frac{d}{dt}S_i = \dot{S}_i = \frac{\dot{V}_{w,i}(V_{w,i} + V_{o,i}) + V_{w,i}(\dot{V}_{w,i} + \dot{V}_{o,i})}{V_{tot,i}^2} \tag{5.20}$$

The rate of change in the oil and water volume ($\dot{V}_{w,i}$ and $\dot{V}_{o,i}$) is determined by the oil/water flowing in/out of the cell.

$$\dot{V}_{w,i} = \sum_{j \in \mathcal{N}_i} q_{w,i,j} + q_{in,i} - S_i q_{out,i} \tag{5.21}$$

$$\dot{V}_{o,i} = \sum_{j \in \mathcal{N}_i} q_{o,i,j} + (1 - S_i) q_{out,i} \tag{5.22}$$

Combining (5.20) with (5.14), (5.21) and (5.22) and rearranging gives us the $\dot{S}_i$:

$$\dot{S}_i = \frac{\sum_j q_{w,i,j} - S_i \sum_j [q_{w,i,j} + q_{o,i,j}]}{p_i \mu} + \frac{(1 - S_i) q_{in}}{p_i \mu} \tag{5.23}$$

Each cell in the reservoir have a pressure and a water saturation state. The

states for all the cells are combined into the vectors p and S

$$p = \begin{bmatrix} p_1 \\ \vdots \\ p_{n_{cells}} \end{bmatrix} \tag{5.24}$$

$$S = \begin{bmatrix} S_1 \\ \vdots \\ S_{n_{cells}} \end{bmatrix} \tag{5.25}$$

These vectors are again combined into a single state vector x

$$x = \begin{bmatrix} p \\ S \end{bmatrix} \tag{5.26}$$

The total system model can now be described in a compact form by

$$\dot{x}(t) = \bar{A}(x(t))x(t) + \bar{B}(x(t))u(t) \tag{5.27}$$

System outputs

The output from the system tells us how much each of the producer wells are producing. For each producer well we have two outputs, the produced water and the produced oil. The total amount of produced fluid for a well i is controlled by the input variable $u_{out,i}$, where i is the index of the cell where the well is placed. To calculate the output of water $q_{out,w,i}$ and the output of oil $q_{out,o,i}$ the following equations are used.

$$q_{out,w,i} = q_{out,i}S_i \tag{5.28}$$

$$q_{out,o,i} = q_{out,i}(1 - S_i) \tag{5.29}$$

where

$$q_{out,i} = \frac{p_i}{p_0}u_{out,i} \tag{5.30}$$

The ratio between the phases is determined by the water saturation in the cell that the well is producing from. The outputs are stacked in the variable z. First water output from all the producing wells are stacked, then the oil outputs.

$$z = \begin{bmatrix} q_{out,w,1} \\ \dots \\ q_{out,w,N_{prod}} \\ q_{out,o,1} \\ \dots \\ q_{out,o,N_{prod}} \end{bmatrix} \tag{5.31}$$

5.2.1 Time-discretization of the model

At this point the model exist in a continuous form with respect to time. To implement the model on a computer we need to discretize it. This is done by using the Euler integration scheme [7] described in section 2.1.1.

$$\begin{aligned} x^{k+1} &= x^k + \Delta t \dot{x} \\ &= x^k + \Delta t \bar{A}(x^k)x^k + \Delta t \bar{B}(x^k)u^k \end{aligned} \tag{5.32}$$

Finally we can write the discrete model in compact form

$$x^{k+1} = A(x^k)x^k + B(x)u^k \tag{5.33}$$

where

$$A(x^k) = (1 + \Delta t \bar{A}(x^k)) \tag{5.34}$$

$$B(x^k) = \Delta t \bar{B}(x^k) \tag{5.35}$$

Cost and restrictions

We are going to use MPC to control the reservoir. The MPC application needs a specification of what we mean by optimal control as it using this specification to find the optimal inputs to the system. In our case optimal control is specified as recovering as much oil as possible at a low cost. Injection water into the reservoir represents a cost as the water must be transported and pumped into the reservoir. Recovering water from the reservoir also represents a cost as this water must be separated from the oil and processed (cleansed to make sure it is pure enough to be disposed without environmental damage) before it can be deposed again. The only profitable situation considered is the recovery of oil. Since the MPC application needs a mathematical definition of the optimal case. This can be done by weighting water injection, water production and oil production. The objective function is chosen to be quadratic meaning that the quadratic term is weighted. The water injection and water production is weighted by $\bar{R}$ and $\bar{Q}_w$ respectively. Both the weights are a positive constants as they represents a cost. The production of oil is weighted by the constant $\bar{Q}_o$ which is negative since this represents a profit. The resulting objective

function J used in the MPC application can be described by:

$$J = \sum_{k=1}^{N_{H_x}} (z^k)^T Q z^k + \sum_{k=0}^{N_{H_u}} (u^k)^T R u^k \tag{5.36}$$

$$Q = \begin{bmatrix} \bar{Q}_w & 0 & \cdots & 0 \\ 0 & \ddots & & \\ & & \bar{Q}_w & \vdots \\ \vdots & & \bar{Q}_o & \\ & & & \ddots \quad 0 \\ 0 & \cdots & 0 & \bar{Q}_o \end{bmatrix} \tag{5.37}$$

$$R = \begin{bmatrix} \bar{R} & 0 & \cdots & 0 \\ 0 & \ddots & & \\ & & \bar{R} & \vdots \\ \vdots & & 0 & \vdots \\ & & & \ddots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} \tag{5.38}$$

The matrix Q is a diagonal matrix containing $\bar{Q}_w$ and $\bar{Q}_o$, this is because the output vector z consist of the water- and oil output for all production wells, see equation (5.31). The diagonal matrix R is also divided into two parts, one containing the water injection weight $\bar{R}$ and zeros. This is because the input vector u contains both the water injection variables q_in and the output variables q_{out}. It is not desirable to weight q_{out} as this is done one the the outputs z.

5.3 Partial derivatives of the system equations

The use of adjoint gradient calculation require some additional information about the system. The gradients $\nabla_{x^k} f(x^k, u^k)$ and $\nabla_{x^k} f()$ are required. The function $f(x^k, u^k)$ is the system equation, $x^{k+1} = f(x^k, u^k)$. To calculates these gradients each of the states in the system equations needs to be derivated with respect to all the states and inputs. It can be shown that the derivatives are

$$\begin{aligned} \frac{\partial}{\partial p_i^k}(p_i^{k+1}) &= 1 + \frac{\Delta t}{\mu} \sum_j \left[\frac{\partial}{\partial p_i^k}(q_{w,i,j}^k) + \frac{\partial}{\partial p_i^k}(q_{o,i,j}^k) \right] \\ &= 1 + \frac{\Delta t}{\mu} \sum_j \left[-\eta_{i,j} \right], \quad j \in \mathcal{N}_\rangle \end{aligned} \tag{5.39}$$

Derivation of p_i^{k+1} with respect to p_i is linearly depending on the permeability between cell i and the neighbor cells $j \in \mathcal{N}$.

$$\begin{aligned}\frac{\partial}{\partial p_j^k}(p_i^{k+1}) &= \frac{\Delta t}{\mu}\left(\frac{\partial}{\partial p_j^k}(q_{w,i,j}^k) + \frac{\partial}{\partial p_j^k}(q_{o,i,j}^k)\right) \\ &= \frac{\Delta t}{\mu}(\eta_{i,j}), \quad j \in \mathcal{N}_\rangle \end{aligned} \tag{5.40}$$

The derivative with respect to the pressure in a neighbor cell is only depending on the permeability between cell i and j.

$$\frac{\partial}{\partial S_i^k}(p_i^{k+1}) = 0 \tag{5.41}$$

The pressure p_i is not depending on the saturation S_i in cell i. Pressure will remain constant even if the ratio between oil an water changes because of equal density (simplification).

$$\frac{\partial}{\partial S_j^k}(p_i^{k+1}) = 0 \tag{5.42}$$

The pressure p_i does not change if the saturation in any of the neighboring cell changes, because of equal density for water and oil.

$$\frac{\partial}{\partial q_{in,i}^k}(p_i^{k+1}) = \left\{ \begin{array}{ll} \frac{\Delta t}{\mu} & \text{if } i \in \mathcal{I} \\ 0 & otherwise \end{array} \right\} \tag{5.43}$$

where $\mathcal{I}$ is the set of injector indices. If a injector well is present, then pressure increases linearly with the volume of injected water.

$$\frac{\partial}{\partial q_{out,i}^k}(p_i^{k+1}) = \left\{ \begin{array}{ll} -\frac{\Delta t}{\mu} & \text{if } i \in \mathcal{P} \\ 0 & otherwise \end{array} \right\} \tag{5.44}$$

where $\mathcal{P}$ is the set of producer indices. If a producer is present, then pressure decreases linearly with the volume of produced liquid.

$$\frac{\partial}{\partial p_i^k}(S_i^{k+1}) = \frac{\Delta t}{(p_i^k)\mu}\left(\sum_j\left[\frac{\partial}{\partial p_i^k}(q_{w,i,j}^k)\right] - S_i^k\sum_j[-\eta_{i,j}]\right) - \frac{\Delta t}{(p_i^k)^2\mu}\left(\sum_j[q_{w,i,j}^k] - S_i^k\sum_j[q_{w,i,j}^k + q_{o,i,j}^k] + (1-S_i^k)q_{in,i}^k\right) \quad (5.45)$$

$$, \quad j \in \mathcal{N}_\rangle$$

The above equation shows that changing the pressure in cell i has great influence on the water saturation in that cell. This is because the flow from neighbor cells changes and the fact that total fluid volume (and therefor water/oil volume) is determined by the pressure.

$$\frac{\partial}{\partial p_j^k}(S_i^{k+1}) = \frac{\Delta t}{p_i^k\mu}\left(\frac{\partial}{\partial p_j^k}(q_{w,i,j}^k) - S_i^k\eta_{i,j}\right), \quad j \in \mathcal{N}_\rangle \quad (5.46)$$

The change of pressure in a neighbor cell j only affects the flow between that cell and cell i giving a simple expression only depending on the flow and permeability.

$$\frac{\partial}{\partial S_i^k}(S_i^{k+1}) = 1 + \frac{\Delta t}{p_i^k\mu}\left(\sum_j\left[\frac{\partial}{\partial S_i^k}(q_{w,i,j}^k)\right] - \sum_j[q_{w,i,j}^k + q_{o,i,j}^k] - q_{in}^k\right), \quad j \in \mathcal{N}_\rangle \quad (5.47)$$

A change in the water saturation in cell i at time-step k will definitely affect the water saturation S_i at the next time-step $(k+1)$.

$$\frac{\partial}{\partial S_j^k}(S_i^{k+1}) = \frac{\Delta t}{p_i^k\mu}\left(\frac{\partial}{\partial S_j^k}(q_{w,i,j}^k)\right), \quad j \in \mathcal{N}_\rangle \quad (5.48)$$

Changing the water saturation in a neighbor cell of i will only affect the saturation in cell i if the flow is going from cell i to cell j, this is handled by the derivative of the flow equation with respect to the water saturation S_j^k

$$\frac{\partial}{\partial q_{in,i}^k}(S_i^{k+1}) = \left\{\begin{array}{ll} \frac{\Delta t}{p_i^k\mu}(1-S_i^k) & \text{if } i \in \mathcal{I} \\ 0 & otherwise \end{array}\right\} \quad (5.49)$$

where $\mathcal{I}$ is the set of injector indices. Injection water into a cell will definitely change its water saturation.

$$\frac{\partial}{\partial q^k_{out,i}}(S_i^{k+1}) = 0 \tag{5.50}$$

A producing well in cell i will not affect its water saturation because the fluid produced will have the same ratio between water/oil as the cell itself.

Injection of water into the reservoir is done to keep the reservoir pressure above a certain level. If and injector well is perforated into the well at grid block j we can directly control the source terms q_w and q_o. Since we only inject water into the reservoir, the term q_o will be zero, while the water injection q_w depends on the volume and density in the grid-block, which gives us

$$q_o^j = 0 \tag{5.51}$$

$$q_w^j = \frac{\rho(p^j)}{v^j} q^j, \quad j \in \mathcal{N}_{inj} \tag{5.52}$$

where v^j is the grid volume, q^j is the rate of the injected water and $\mathcal{N}_{inj}$ is the set of grid block indices where an injection well is perforated.

In producer wells the liquid drained is a combination of oil and water, resulting in indirect control of the phases. The model used for the produced liquid is

Chapter 6

Simulations

This section presents the different simulation cases used in this thesis. The reservoir model used is presented in chapter 5. The reservoir size used and the placement of wells is described here. The choice of objective function and constraints used is also included here, since they vary for each of the cases. Each subsection will describe a different case.

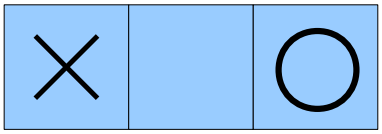
The number of optimization variables are defined by the number of inputs multiplied by the prediction horizon. Extending the horizon increases the number of optimization variables linearly.

6.1 Case 1

Case 1 is a simple case illustrating the efficiency of the adjoint gradient calculations (even for a small system) when there is no output constraints present. The reservoir starts out as steady state meaning that the pressure in all the cells are equal. The reservoir is tiny, only divided into three cell with one injector and one producer. Figure 6.1 shows an illustration of the reservoir. Figure 6.1a shows well placement, where the cross shows the injector well and the circle shows the producer cell. Figure 6.1b shows the water saturation in the reservoir cells, where light areas shows a high water saturation and dark areas shows large oil saturation.

The constraints on the inputs are simple box constraints:

$$0 \leq u_i^k \leq 1 \tag{6.1}$$



(a) Well placement

(b) Water saturation map

Figure 6.1: Tiny reservoir model

In this example there is no cost of producing water, but injecting water is given a cost while producing oil is associated with a profit.

The reservoir model are simulated several times with a set of different control horizons using both the adjoint and the finite difference method for gradient calculations. This gives results that can be used to compare the calculation time for the adjoint and finite difference method.

6.2 Case 2

Case 2 uses the same reservoir as in case 1, but introduces output constraints to the problem. A constrained problem will deteriorate the efficiency of the optimization algorithm compared to an unconstrained problem. This case will show how the introduction of output constraints will affect the calculation time for both the adjoint and finite difference method.

Production equipment will often have production limits. Water produced from oil reservoirs have to go through a process of separation and purification before it can be released into the nature again and the production plant will have an upper limit on production capacity. In this case we have chosen to set a limit on the water production. Since there is only one producing well, the limit will only affect one output:

$$z_1^k \leq z_{w,max} \tag{6.2}$$

where z_1^k is the water production at time k and $z_{w,max}$ is the maximum water production. $z_{w,max}$ is set to 0.2.

Now that output constraints are introduced into the problem the optimization algorithm requires gradient information about the constraints. The constraints gradients are calculated using a finite difference approach.

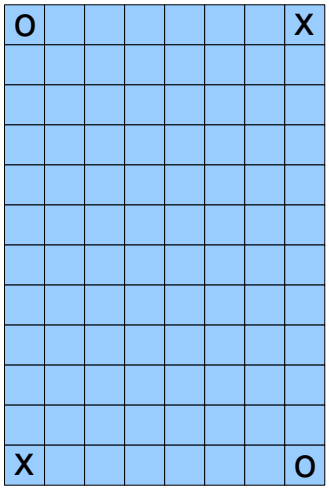
Another way of handeling constraints is by softening them, as described in 4.2.1. This technique is also tested.

6.3 Case 3

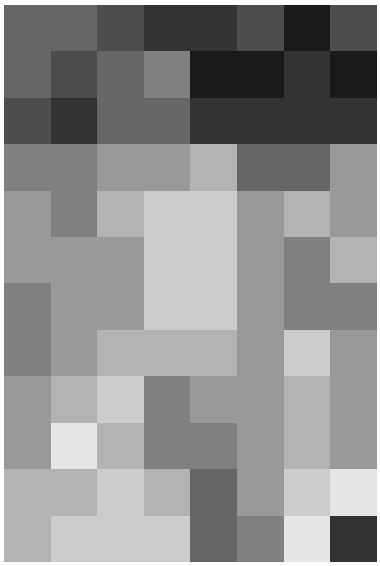
This case will be using a larger reservoir model. The dimesions will be 8 x 12 gridcells, which results in 96 cells with 192 states. The model will have 4 wells, two injectors and two producers. Simulations will be done with a horizon from 5-30 timesteps. The total number of states when the control horizon is 30 will be 5760, which can be considered a large system. The number of inputs will at the same time be 120. Since each case is simulated many times, several times with different control horizon for all the methods tested, a larger system would be not be practical in this thesis.

An illustration of the reservoir is presented in figure 6.3. The subfigure 6.2a shows the well placements, a cross is a injector well while the circle represents a producer well. The water saturation is shown in subfigure 6.2b, darker areas contains mostly oil, while lighter areas contains more water.

The reservoir will be simulated with different approaches for output constraint handeling as well as without any output constraints. The results of the simulations are presented in section 7.4. One of the method tested is the adjoint method with lumped constraints. Lumping the constraints makes the feasible region of the problem larger, making voilation of the original constraints possible. This is solved by checking the outputs after the optimization at each timestep to see if the original constraints is voilated, if it is the valve on the producer well where the voilation occures is adjusted until the constraint is satisfied.



(a) Well placement



(b) Water saturation map

Figure 6.2: Tiny reservoir model

Chapter 7

Results

Simulation results will be presented and commented in this chapter. Results concerning reservoir state, inputs and outputs are not that interesting in this thesis. The most important results will be how good output constraints are handled, and how efficient the optimization is done. A comparison of the optimization time for the adjoint and the finite difference method will be presented for each simulation case, and the handling of possible output constraints will be presented.

A more general discussion on the total results is done in chapter 8.

7.1 Comparison of optimization time

Testing techniques and theories are done by defining multiple scenarios/cases and then simulate to see how well the different configurations perform. We are mainly interested in efficiency for the different techniques. To measure how effective the preform it is natural to look at the time spent executing the optimization. The number of gradient calculations needed could also be interesting, however, simulations shows that for most simulations, the number is equal regardless of the method used.

Comparison of optimization time will be presented for all cases. The time that is compared is time spent in the optimization algorithm, and not the rest of the MPC application. A profiler (Matlab profiler [13]) is used to record the time. CPU-time is used instead of real-time, giving a better result that is less

dependent on overall CPU load.

Within each case the time-axis is normalized, going from 0-1. This is done because the actual time is irrelevant as it is depending on the hardware used in the simulation. The time is only interesting when compared between different optimization methods, which can be done just as easily with normalized data sets.

7.1.1 Θ-notation

To compare the running time for the finite difference method and the adjoint method we need to define a notation suitable for this purpose. To describe the asymptotic running time of an algorithm we use the Θ-notation [15]. The Θ-notation gives an upper and lower asymptotic bound on computation time. For a given function $g(n)$ we denote $\Theta(g(n))$ as the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\} \tag{7.1}$$

Example: The function $g(n) = 5n^2$ belongs to $\Theta(n^2)$ because we are able to find a function $f(n)$ that satisfy (7.1). We could choose $f(n) = n^2$ and the variables $c_1 = 1$, $c_2 = 10$ and $n_0 = 2$. Then $c_1 f(n)$ is a lower bound for $g(n)$ and $c_2 f(n)$ is a upper bound for $g(n)$. This example is illustrated on figure 7.1.

7.2 Results - Case 1

Simulations were done with a control horizon spanning from 5 timesteps to 50 timesteps. Only plots from simulations with 5 and 50 timesteps horizon is included since they represents the extreme situation of the simulations.

Figure 7.2 and 7.2 shows the inputs and outputs of the system. From these plots we can see that the water injection and oil production is greater when the control horizon is 50 timesteps rather than 5 timesteps. The water injection has increased the most, this is done to produce more oil since the cost of injecting water is lower than the profit of the extra oil produced. A longer horizon looks further into the future, making water injection more profitable

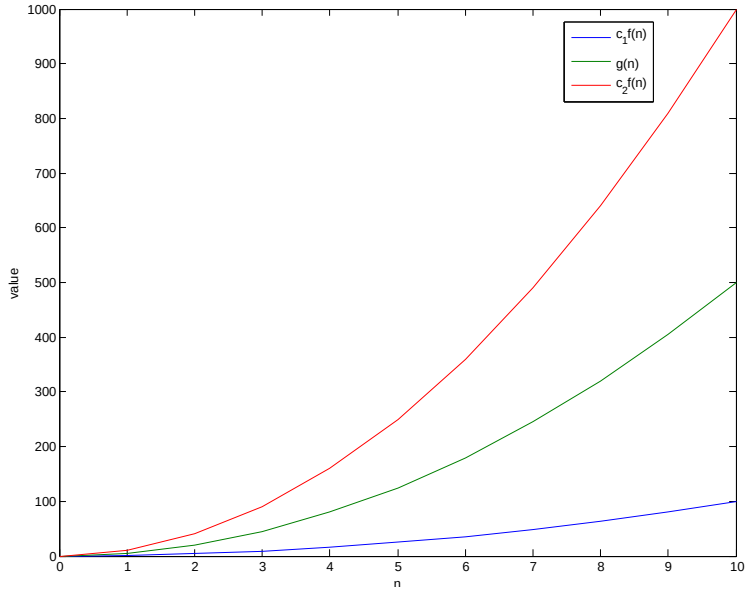
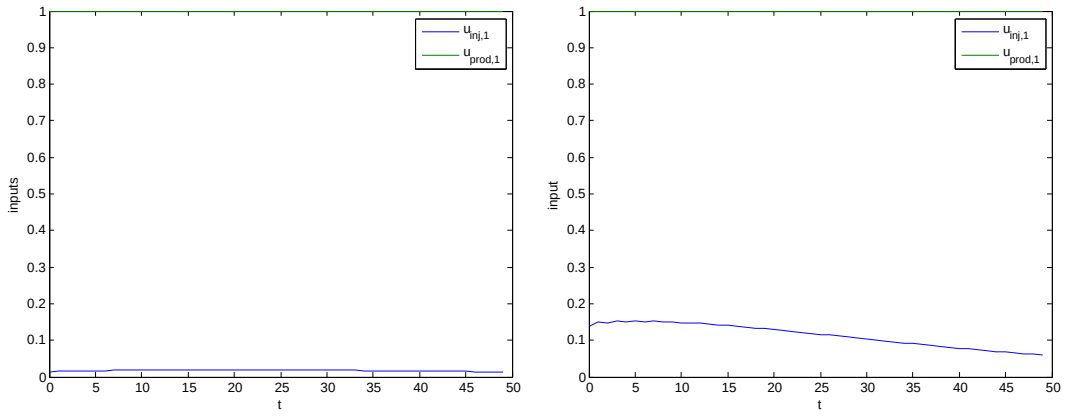


Figure 7.1: Θ-notation example.

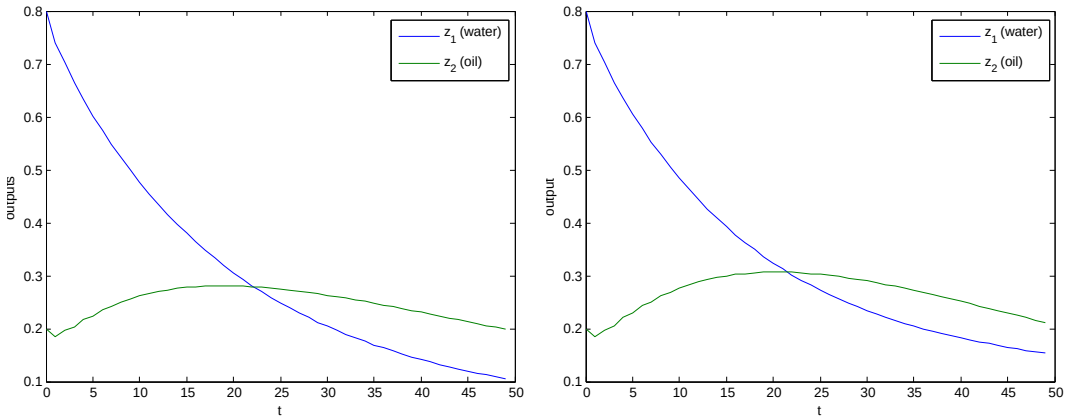
because sees that injecting more water leads to a higher oil production longer into the future. This simple example clearly show that a longer optimization horizon is preferred.



(a) Control horizon: 5 timesteps (b) Control horizon: 50 timesteps

Figure 7.2: System inputs

Extending the horizon clearly gives a better result, but the computation time will also increase. Let us take a closer look on the computation time. Figure 7.4 shows the computation time as a function of the horizon. It includes both the adjoint and finite difference computation times, as well as two function that are fitted to the original graphs.



(a) Control horizon: 5 timesteps (b) Control horizon: 50 timesteps

Figure 7.3: System outputs

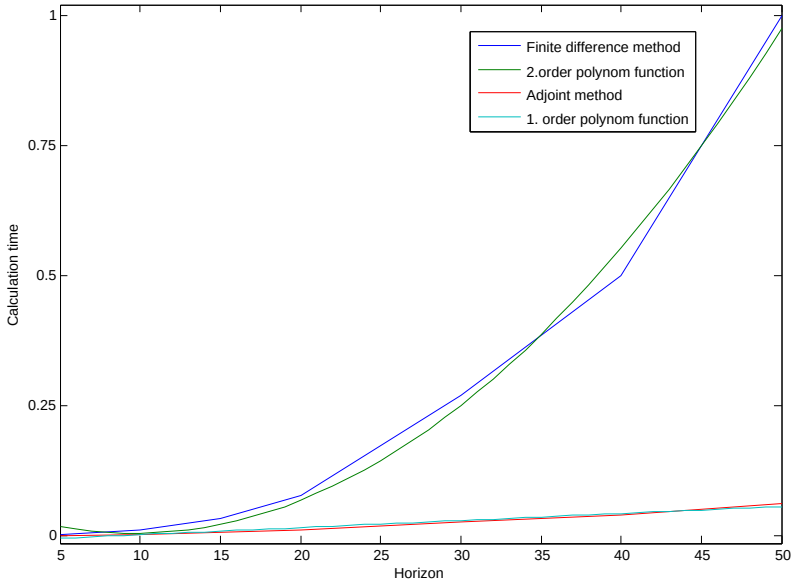
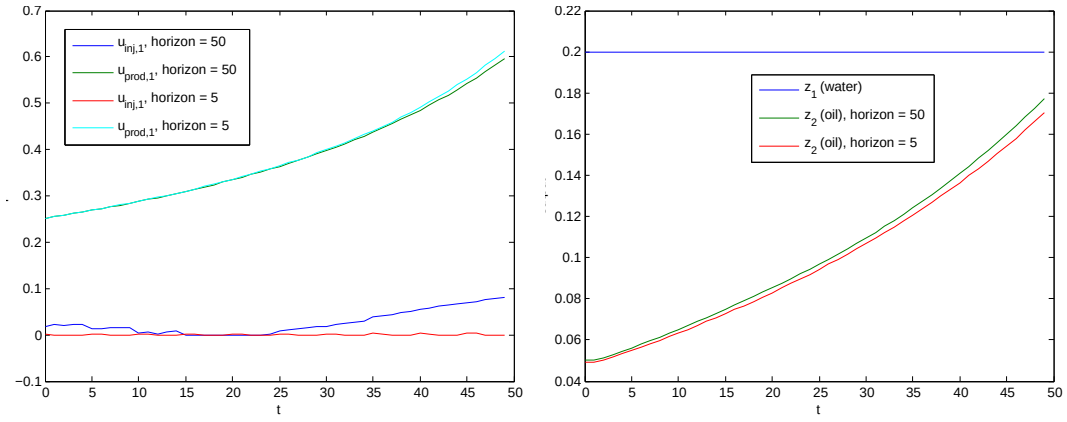


Figure 7.4: Runtime comparison between adjoint and finite difference method.

The plot clearly shows us the strength of the adjoint method with no output constraints. Computation time is close to linear with respect to the horizon, while the computation time for the finite difference method follows a second order polynomial function. Using the Θ-notation, this means that the adjoint method has $\Theta(n)$ computation time and the finite difference method has $\Theta(n^2)$ computation time, where n is the horizon (or number of decision variables, which is increasing linear with the horizon length).

7.3 Results - Case 2

Case 2 was simulated with a horizon of 5-50 timesteps. Figure 7.3 shows the inputs and outputs of the system for simulation with 5 and 50 time-step horizon. A longer horizon gave a higher oil output and water injection, just as expected. We can also see that the water production was held at a maximum at all times. The reservoir cell where the producer well is places contains much more water than oil, making the water production constraint active at all times during the simulation. The constraint is never violated as this is a hard constraint.



(a) Inputs

(b) Outputs

Figure 7.5: System inputs and outputs

Gradient calculation time is compared in figure 7.6. The graphs clearly shows that the adjoint approach to gradient calculations is the fastest. The finite difference method still follows a $\Theta(n^2)$ calculation time with respect to the length of the horizon. What is more interesting is that the adjoint approach also follows a $\Theta(n^2)$ trajectory. Figure 7.3 presents the same information as figure 7.6, but the graphs showing the finite difference method and the adjoint method is separated into two subplots, figure 7.7a and figure 7.7b.

The reason for this behavior is that as the horizon length grow the problem becomes larger and more complex making it harder to solv. The larger problem is harder to solv, thus more iterations (and gradients) in the SQP algorithm is required. This means that when the control horizon grows, the number of gradients calculated for the complete optimization and simulation also grows, introducing extra optimization time in addition to the time used to calculate the gradients.

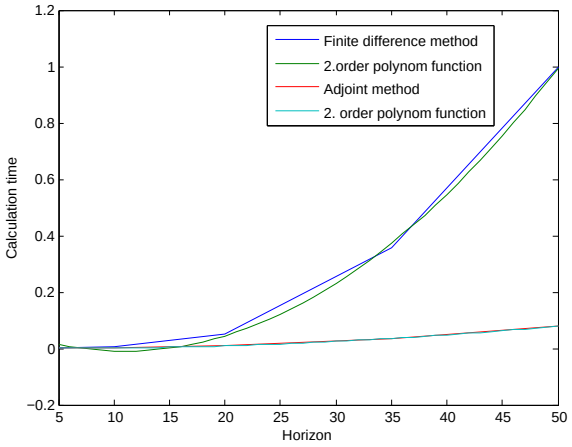
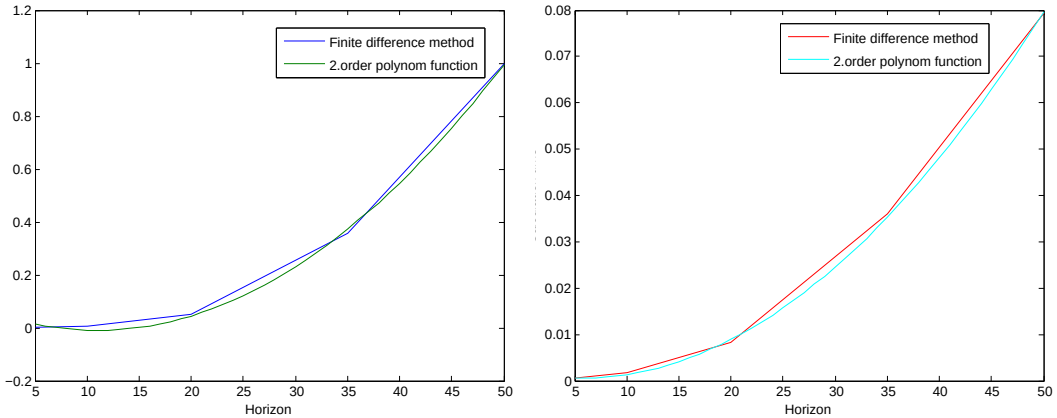


Figure 7.6: Runtime comparison between adjoint and finite difference method.



(a) Finite difference method (b) Adjoint method

Figure 7.7: Runtime for gradient calculation

Figure 7.8 shows the runtime comparison of the different simulation done on the tiny reservoir model. The datapoints are shown as dots with stipled dots connecting them. The solid lines are polynom functions fitted to the data to

show a interpolation of the results(simulations time would most likely follow the solid line if done for every horizon length).

As expected the adjoint method without any constraints is the fastest. The adjoint method with soft constraints is a little slower than the unconstrained case, this is a result of a the extra complexity introduces when the constraints were softened. The softening extends the objective function, creating a problem that is slightly harder to solve. The adjoint method with hard constraints are the third fastest simulation. It is interesting that the adjoint method with hard constraints, where the constraints gradients is calculated by finite differences, is faster than the unconstrained finite difference method. The three slowest simulations used the finite difference methods.

It is clear that for this tiny system, adjoint methods would be very effictive compared to the finite difference method.

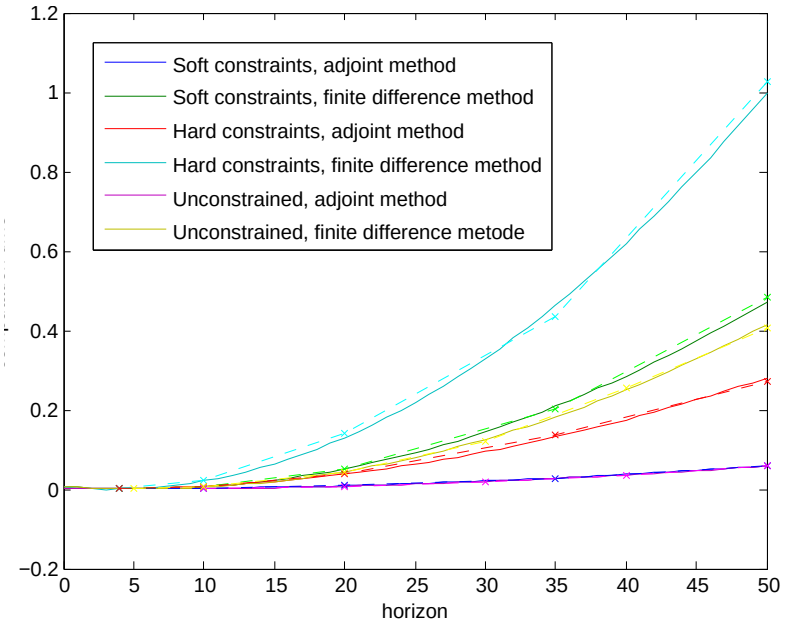


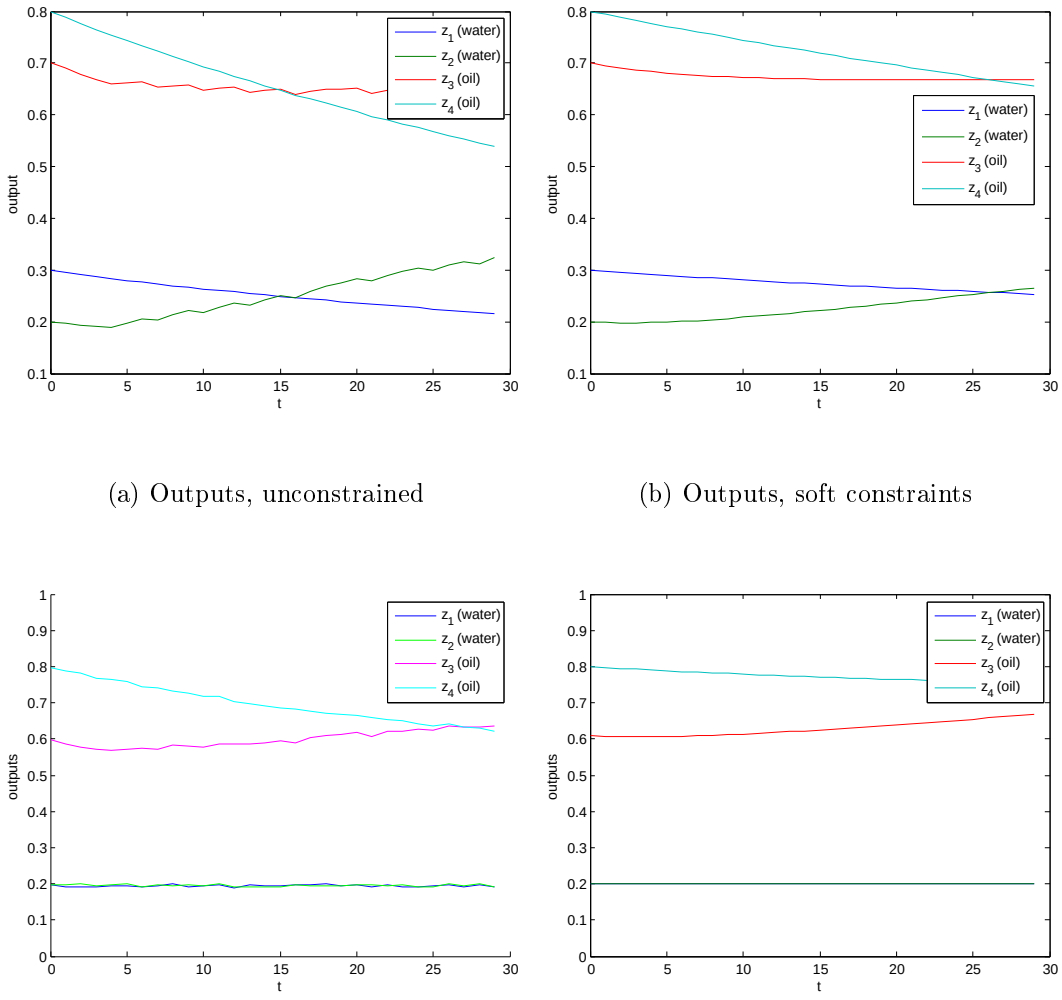
Figure 7.8: Runtime comparison between adjoint and finite difference method.

7.4 Results - Case 3

Four different types of simulations were done. The first simulation was done on a unconstrained system, the second simulation used soft output constraints, the third simulations used hard constraints where constraints gradients was simulated using the finite difference method and the fourth simulation teste

the lumping constraint scheme. Only the adjoint method was utilized on the lumped constraint case.

Figure 7.4 show the output in the different cases.



(a) Outputs, unconstrained

(b) Outputs, soft constraints

(c) Outputs, lumped constraints

(d) Outputs, hard constraints

Figure 7.9: System outputs

The unconstrained and soft constraint cases both voilates the output constraints. The hard constraints and the lumping scheme cases never voilates the constraint, but as seen from figure 7.9c the inputs was modified more than nessasary.

Simulation time for each of the cases and each of the methods are placed

together in figure 7.10. Data points is represented by a dot, while the solid line is a polynomial function fitted to the data. The legend in the figure speaks for it selfs. The unconstrained adjoint method was the fastest, just as expected and the hard constrained finite difference method was the slowest. As we see the adjoint method with lumped constraints is faster than the case with hard constraints. The lumped method is clearly an good alternative, at least for this case.

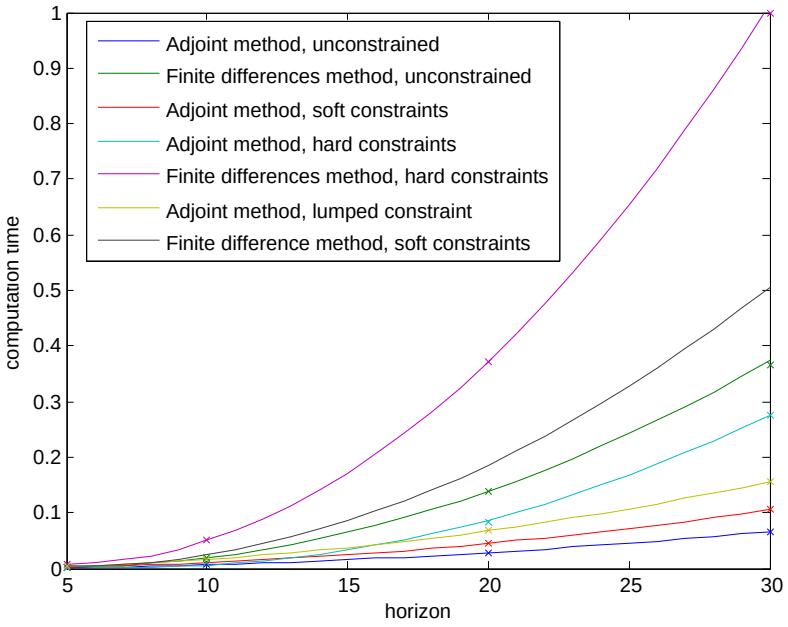


Figure 7.10: Runtime comparison between adjoint and finite difference method.

All the cases seems to be following a quadratic simulation time ($\Theta(n^2)$).

Chapter 8

Discussion

This chapter contains some general discussion based on the results in chapter 7. Two different reservoir was used in the simulations. The first reservoir is tiny and was included to make the testing of methods and do model/optimization modification easy without the need for long simulations. The other reservoir was used to test the methods on a large system which has several thousand states with the longest horizon. The results obtaint from both reservoir are quite simular when considering the computation time. It shows that the adjoint method can save a lot of computation time even with output constraints.

Reservoir simulations is often used to find well placement and parameter configuration on oil reservoirs. Engineers need to do an simulation to test their configuration. Often modifications need to be done after an simulations is done and a resimulation is needed. This process can be repeated many times resulting in a large number of simulations. These testing simulations often do not require hard constraints on the output, soft constraints would be a satifying compromise if the simulation time is reduced considerable. The adjoint method with softened constraints would often be good enough.

In the case of the unconstrained case, the adjoint method performs far better then the finite difference method. Using the adjoint method with softened constraints will give almost as good performance as the unconstrained case. Softening the output constraints seems like a good alternative for reservoir simulations used to deside different configurations since such simulations often needs to be repeated to find a good configuration. In the simulations a quadratic penalty functin was used to penalize a voilated constraint. Other penalty functions giving a better result may exist.

Using soft constraints instead of hard constraints may not always give a solution that is accurate enough, for instance reservoir engineers has found a good configuration and wants to do a simulations producing more accurate data to use for other purposes. In such a situations output constraints needs to be handled properly as hard constraints that never gets voilated. For an SQP-algorithm using an active-set and line-search technique for optimization then needs the gradients of the constraints. For the adjoint method this will reduce the efficiency because a new set of langrange variables needs to be calculated for each active constraint. This weekness can be reduced by lumping the constraints. Lumping constraints reduces the number of constraints, but the feasible region of the optimization problem is often expanded. This was solved by manipulating the input after the optimization if the inputs computed by the optimization solver voilated any of the original constraints. The manipulation used an linear approximation of the relationship between the opening of the output well and the outpout to do an iterative correction of the input. This method worked quite well, the constraints was never voilated, but some times the input was corrected a little more than nessasary.

The more simple approach to handling the constraints was to use a finite difference technique to calculate the gradients of all the constraints without lumping. This approach is less effective but using the adjoint method for calculating the objective function gradients made the result far better than using the finite difference method.

Chapter 9

Conclusion and further work

9.1 Conclusion

Through this thesis we have studied the adjoint and finite difference method for gradient calculations used in a MPC application on reservoir models. Both methods has weaknesses and strengths. The adjoint method is well suited for problems as long as no output constraints is present. The present of output constraints will deteriate the efficiency of the adjoint method. This can however be handled by softening the output constraints of lumping them together. Softening constraints results in a unconstrainted problem which the adjoint method handles well. The disadvantage is that softening constraints often results in voilation of the original constraints.

The lumping scheme is more complex but for some problems, as reservoir simulations where there can be found a quite simple approximated relationsship between the control signal controlling the output and the actual output, the lumping of constraints works good and reduces the computation time compared to using hard constraints.

The use of adjoint methods require knowledge about the the equations in the model used in the simulations. Information about these equations are not always availible, for instance in commercial simulators. In situations where the model is not known the use of other methods as the finite difference method must be used. The process of creating a model to use with the adjoint method will also require extra development work since $\nabla_x f$, $\nabla_u f$, $\nabla_x G$, $\nabla_u G$ needs to be generated in addition to just the objective function and the system equation.

Adjoint method seems promising for use on reservoir models, it clearly has the potetial to reduce the simulation time.

9.2 Further work

This thesis has studied the adjoint method for gradients calculations in MPC applications. The method was tested on a simple reservoir model to test its performance. The results shows that the adjoint method has great potential in reservoir simulations. The optimization scheme in this thesis was an SQP-algorithm.

It would be interesting to test the adjoint method in other types of optimization algorithms with gradient based searching. The adjoint method can give a great computation time reduction in a SQP-algorithm, but other algorithms may reduce this even more. Since the handling of output constraints may deteriorate the efficiency of the adjoint method more research in the constraint handling topic should be done. For instance barrier methods and penalty functions should be studied further as they are able to remove the need for output constraints.

Adjoint method are particular effective for large systems. The method should be tested on even larger system that those presented in this thesis. Testing the method on full-scale simulators, preferably on commercial simulators would be interesting.

Appendix A

Program code

This chapter contains some code that may be interesting for the reader. The code was written in Matlab using object-oriented programming.

A.1 Optimization problem class

The most important class is the optimization problem (OP) class that represents a somewhat general optimization problem. The class is abstract, making it impossible to initiate it, but its used as a superclass for specialized problems defined by the user. The class defines what methods that any subclass must implements, for instance the system equation function and functions for calculating output constraints.

The OP class implements two function, both calculating the objective function value and gradients, one using the adjoint method (function J_Adjoint()) the other uses a finite difference method (function J_FiniteDiff()).

Listing A.1: OP.m

```
1  classdef OP < handle
2      properties
3          % Number of inputs/states/outputs
4          nInputs = 0;
5          nStates = 0;
6          nOutputs = 0;
7
8          % Current state
9          x0;
10
11         % Properties used in the optimization / MPC
```

```
12          epsilon = 0.00001; % pertubation size used in finite differnce
                calculations.
13          simTime = 100; % Simulation length
14          horizon = 20; % Horizon of the optimization
15          useAdjoint = 1; % If this equals 1, adjoint gradient methods are used
16          con_A = []; % Inequality contraints on decision variables: con_A * u <=
                con_b
17          con_b = []; % ----^
18          con_Aeq = []; % Equality contraints on decision variables: con_Aeq * u
                = con_beq
19          con_beq = []; % ----^
20          con_lb = []; % lower bound on decision variables
21          con_ub = []; % upper bound on decision variables
22          options = optimset('Display','notify', 'Algorithm','active-set');
23          uNom;
24
25          xNom;
26          zNom;
27          D_uk_fNom;
28          D_xk_fNom;
29      end
30
31      % Abstract methods that needs to be implemented in subclasses
32      methods(Abstract=true)
33          % Calculate next state, given current states, a input-vector and the
34          % timestep.
35          newX = f(o,xk,uk,t)
36
37          %Calculate outputs
38          z = g(o,xk,uk,t);
39
40          % Calculate the gradient of the last timestep for objective function
41          res = D_xN_HN(o,xN)
42
43          % Calculate the system equation gradients with respect to the
44          % inputs
45          res = D_uk_f(o,xk,uk,t)
46
47          % Calculate the objective function gradients with respect to the
48          % state vector
49          res = D_xk_G(o,xk,uk,t)
50
51          % Calculate the objective function gradient with respenct to the
52          % input vector
53          res = D_uk_G(o,xk,uk,t)
54
55          % Calculate the system equation gradients with respect to the state
56          % vector
57          res = D_xk_f(o,xk,uk,t)
58
59          % Calculate the objective function value
60          res = J(o,x,u)
61
62          % nonlinear constraints
63          [c ceq] = con_nonlcon(o,uk)
64      end
65
66      methods
67          % Calculate objective function value and gradients using the
68          % adjoint method
69          % @param u: Input vector
70          % @return J: Objective function value
71          % @return D_uk_J: Gradients for the objective function value
72          function [J, D_uk_J] = J_Adjoint(o, u)
73              % Check input length
74              if mod(length(u), o.nInputs) > 0
75                  error('J_new: Wrong input length');
```

```
76              else
77                  % Input length ok, continue
78
79                  N = o.horizon; % Lenght of horizon
80
81                  % Preallocate space
82                  x = [o.x0; zeros(o.nStates*N, 1)]; % State vector
83                  z = zeros(o.nOutputs*N, 1);
84
85                  % Do forward simulation
86                  for i=1:N
87                      xk = x((o.nStates*(i-1)+1):(o.nStates*i));
88                      uk = u((o.nInputs*(i-1)+1):(o.nInputs*i));
89                      x((o.nStates*i+1):(o.nStates*(i+1))) = o.f(xk,uk,i);
90                      z(((i-1)*o.nOutputs + 1):(i*o.nOutputs)) = o.g(xk,uk,i);
91                  end
92                  % Calc objective function value
93                  J = o.J(z,x,u);
94
95                  % Save states and outputs from simulation around nominal
96                  % trajectory
97                  o.xNom = x;
98                  o.zNom = z;
99
100                 % Preallocate space
101                 o.D_xk_fNom = zeros(o.nStates*N, o.nStates);
102                 o.D_uk_fNom = zeros(o.nStates*N, o.nInputs);
103
104                 % Preallocate space
105                 D_uk_J = zeros(size(u));
106
107                 %Initiate langrange multipliers vector lamda
108                 lambda = zeros(o.nStates*(N+1),1);
109
110                 % Assign the last Lagrange multiplicator
111                 lambda((o.nStates*N + 1):(o.nStates*(N+1)),1) = o.D_xN_HN(x((o
                        .nStates*i+1):(o.nStates*(i+1))));
112
113                 % Do the backward iteration
114                 k = N;
115                 while k > 0
116                         % Precalc. indices
117                         xi = (o.nStates*(k-1) + 1):(o.nStates*(k));
118                         ui = (o.nInputs*(k-1) + 1):(o.nInputs*(k));
119
120                         % Extract states and inputs for the current
121                         % timestep
122                         x_k = x(xi, 1);
123                         u_k = u(ui, 1);
124
125                         % Calculate system equation gradients with respect
126                         % to inputs and outputs
127                         o.D_uk_fNom(xi,:) = o.D_uk_f(x_k,u_k,k);
128                         o.D_xk_fNom(xi,:) = o.D_xk_f(x_k,u_k,k);
129
130                         % Temp. variables
131                         tmp1 = o.D_uk_G(x_k,u_k,k);
132                         tmp2 = (lambda((o.nStates*k + 1):(o.nStates*(k+1)),1)'
                                * o.D_uk_fNom(xi,:))';
133
134                         % Calculate objective function gradients
135                         D_uk_J(ui, 1) = tmp1 + tmp2;
136
137                         % Calculate previous lambda values
138                         lambda(xi,1) = o.D_xk_G(x_k,u_k,k) + o.D_xk_fNom(xi,:)
                                * lambda((o.nStates*k + 1):(o.nStates*(k+1)),1);
139
```

```
140                     k = k - 1;
141                 end
142             end
143         end
144
145         % Calculate objective function value and gradients using the finite
146         % difference method
147         % @param u: Input vector
148         % @return J: Objective function value
149         % @return D_uk_J: Gradients for the objective function value
150         function [J, D_uk_J] = J_FiniteDiff(o, u)
151             % Check input vector length
152             if mod(length(u), o.nInputs) > 0
153                 error('J_new: Wrong input length');
154             else
155                 % Input is ok, start calculation
156
157                 N = length(u) / o.nInputs; % Number of timesteps
158
159                 % Preallocate space
160                 x = [o.x0; zeros(o.nStates*N, 1)]; % Nominal statevector
161                 z = zeros(o.nOutputs*N, 1);
162
163                 % Do nominal forward simulation
164                 for i=1:N
165                     xk = x((o.nStates*(i-1)+1):(o.nStates*i));
166                     uk = u((o.nInputs*(i-1)+1):(o.nInputs*i));
167                     x((o.nStates*i+1):(o.nStates*(i+1))) = o.f(xk,uk,i);
168                     z(((i-1)*o.nOutputs + 1):(i*o.nOutputs)) = o.g(xk,uk,i);
169                 end
170                 % Calculate objective function
171                 J = o.J(z,x,u);
172
173                 % Save state/output around nominal trajectory
174                 o.xNom = x;
175                 o.zNom = z;
176
177                 % Preallocate space
178                 D_uk_J = zeros(size(u));
179
180                 % Iterate through every timestep
181                 for k = 1:N
182                     % Precalculate some indices
183                     x_k_end = k * o.nStates;
184
185                     % Initiate temporal x-vector, exploiting causality
186                     xTemp = [x(1:x_k_end, 1); zeros((N-k) * o.nStates + 1, 1)
                            ]; % Exploit causality
187                     zTemp = [z(1:k*o.nOutputs,1); zeros((N-k-1) * o.nOutputs +
                             1, 1)]; %
188
189                     % Iterate through all inputs
190                     for i=1:o.nInputs
191
192                         % Initiate temporal input vector (Pertubate one of
193                         % the inputs
194                         uTemp = u;
195                         uTemp((k-1)*o.nInputs + i, 1) = uTemp((k-1)*o.nInputs
                                + 1, 1) + o.epsilon;
196
197                         % Calculate the remaining elements of the temporal
198                         % state-vector
199                         for l=k:N
200                             % Calc. indices
201                             x_s = (l-1) * o.nStates + 1;
202                             x_e = l * o.nStates;
203                             x_s1 = x_s + o.nStates;
```

```
204                                 x_e1 = x_e + o.nStates;
205                                 u_s = (l-1) * o.nInputs +1;
206                                 u_e = l * o.nInputs;
207
208                                 % Extract inputs and outputs for current
209                                 % timestep
210                                 xkTemp = xTemp(x_s:x_e,1);
211                                 ukTemp = uTemp(u_s:u_e,1);
212
213                                 % Calc. states and outputs with pertubatet
214                                 % input
215                                 xTemp(x_s1:x_e1,1) = o.f(xkTemp,ukTemp,l);
216                                 zTemp(((l-1)*o.nOutputs + 1):(l*o.nOutputs)) = o.g
                                        (xkTemp, ukTemp,l);
217                             end
218
219                             % Calculate objective function gradients
220                             D_uk_J((k-1)*o.nInputs + i, 1) = (o.J(zTemp, xTemp,
                                    uTemp) - J)/o.epsilon;
221                         end
222                     end
223                 end
224             end
225         end
226 end
```

Bibliography

[1] Frank Allgöwer, Rolf Findeisen, and Zlotan K. Nagy. Nonlinear model predictive control: From theory to application. J. Chin. Inst. Chem. Engrs., 35:299–315, 2004.

[2] P. Sarma & W. H. Chen & L. J. Durlofsky & K. Aziz. Production optimization with adjoint models under nonlinear control-state path inequality constraints. 2006.

[3] Jørgen Borgesen. Adjoint based gradient calculation for model predictive control for large scale models.

[4] Christof Büskens and Helmut Maurer. Sqp-methods for solving optimal control problems with control and state constraints: adjoint variables, sensitivity analysis and real-time control. 1998.

[5] Eduardo F. Camacho and Carlos Bordons. Model Predictive Control. Springer, 2007.

[6] L. Wirsching J. Albersmyer P. Kühl M. Diehl and H.G. Bock. An adjoint-based numerical method for fast nonlinear model predictive control. The International Federation of Automatic Control, Seoul, Korea, 2008.

[7] Olav Egeland and Jan Tommy Gravdahl. Modeling and Simulation for Automatic Control. Marine Cybernetics AS, 2002.

[8] Frederick S. Hillier and Gerald J. Lieberman. Introduction to Operations Research. McGraw-Hill, 2005.

[9] Stephen J. Wright J. Nocedal. Numerical Optimization. Springer, 2006.

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

[11] C. T. Kelley. Iterative Methods for Optimization. 1999.

[12] J.M. Maciejowski. Predictive Control with Constraints. Pearson Education Limited, 2002.

[13] Mathworks. Matlab documentation.

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

[15] R.L. Rivest T.H. Cormen, C.E. Leiserson and C. Stein. Introduction to algorithms. McGraw-Hill, 2003.

[16] Ya-Xiang Yuan Wenyu Sun. Optimization Theory and Methods - Nonlinear programming. Springer.

[17] Curtis H. Whitson and Michael R. Brulé. Phase Behavior. SPE, 2000.

[18] Maarten Johan Zandvliet. Model-based lifecycle optimization of well locations and production settings in petroleum reservoirs. Technical report, 2008.