Open-Source Software for Nonlinear Constrained Optimization of Dynamic Systems

Rune Brus

Kongens Lyngby 2010
IMM-MSC-2010-26

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC

Abstract

The motivation behind the thesis lies in the growing range of open-source software packages available for solving constrained dynamic optimization problems. The thesis will investigate and promote the use of these software packages. More specific, the software packages ACADO Toolkit, IPOPT, CppAD and JModelica will be thoroughly presented and tested with respect to their ability to solve constrained dynamic optimization problems. The solution methods for solving these kinds of problems will be introduced and connected to the solution strategies of the software packages. The thesis aim to enable the reader to get started using the software packages immediately. A thorough guide on how to install the software packages across operating systems is therefore provided. Further more a virtual machine has been created based on open-source licenses, where all software packages has been preinstalled. This virtual machine is available on request as a OVF formatted preconfigured operating system. The Quadruple Tank Process will serve as a test model for the softwares. This model will enable me to demonstrate the capabilities of the softwares and the limitations they contain.

Resumé

Motivationen bag denne afhandling beror på det voksende udvalg af open source software, der er tilgængeligt til at løse dynamiske optimerings problemer. Afhandlingen vil undersøge og fremme brugen af disse softwares. Software pakkerne ACADO Toolkit, IPOPT, CppAD og JModelica vil blive grundigt præsenteret og testet med hensyn til deres egenskaber til at løse dynamiske optimerings problemer.

Preface

This thesis was prepared at Informatics Mathematical Modeling, the Technical University of Denmark in fulfillment of the requirements for a master thesis.

The key goal of the thesis is to investigate, test, and supply a user guide for using open source mathematical programming software for modeling and solving non-linear dynamic optimization problems with fixed time horizon.

Open-source licensing is needed to manage costs associated with the deployment of optimization solutions, and to facilitate the integration of modeling capabilities from a broader technical community, open source software provides many advantages beyond simple cost savings, including supporting open standards and avoiding being locked in to a single vendor[1].

A key limitation of commercial modeling tools is the inability to customize the modeling or optimization processes. These open source projects allows a diverse range of developers to prototype new capabilities. Thus, developers can customize the software for specific applications, and can prototype capabilities that may eventually be integrated into future software releases.

Further, these open source projects work on a diverse range of computing platforms and is written in general-purpose high-level languages, such as C, C++, Fortran and Python, which supply transparency in software design and implementation. Because any developer can study and modify the software, bugs and performance limitations can be identified and resolved by a wide range of developers with diverse software experience.

A widely used high-level programming language provides a robust foundation

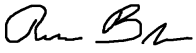
for developing and solving mathematical programming models: these languages has been well-tested in a wide variety of application contexts and deployed on a range of computing platforms. Further, extensions almost never require changes to the core language but instead involve the definition of additional classes and compute routines that can be immediately leveraged in the modeling process. Further, support of the modeling language itself is not a long-term factor when managing software releases.

Modern high-level programming languages are typically well-documented, and there is often a large on-line community to provide feedback to new users.

The softwares is not intended to facilitate modeling better than existing AML tools. Instead, it supports a different modeling approach in which the software is designed for flexibility, extensibility, portability, and maintainability.

The remainder of this thesis is organized as follows. Chapter 1 describes the general types of models, which the software packages of this thesis aim to solve. These models are Dynamic Optimizations Problems. Chapter 2 then introduce the solution method for solving these types of problems in a detailed manner. Chapter 3 goes through the formulation of the Quadruple tank process in terms of a dynamic optimization problem. The purpose of this section is to supply an adequate complex model to test the various softwares of the thesis. Chapter 4,5,6 and 7 describes respectively ACADO Toolkit, the IPOPT solver, the CppAD package and JModelica for the purpose of solving Dynamic Optimizations Problems. Finally, a chapter describing the creation of a preconfigured operating system, where all software packages from the thesis is preinstalled.

Kgs. Lyngby, June 2010



Rune Brus

Contents

Chapter 1

Introducing Dynamic Optimization Problems

In this chapter, I introduce the notion of model based optimization of processes with fixed horizon. The model elements will be described, including the model variations within this field of problems. A simple Optimal Control Problem will be stated. This will act as an introductory model implementation for the various software packages of the thesis. The solution methods relevant for the thesis will then be described and evaluated. These methods are Direct Single Shooting, which is the foundation of ACADO Toolkit, Direct Multiple Shooting, which is a future feature in JModelica, and Direct Collocation, which is the method offered by IPOPT. The chapter will work as a short introduction to the types of models, I am going to handle for the rest of this thesis.

References: Adaptive multiscale methods [2]

1.1 Dynamic Optimization Problems

Moving horizon optimization includes model predictive control (MPC) and receding horizon estimation (RHE). The basis for the majority of these models are the principles of conservation of mass, momentum, and energy. The moving

horizon process phenomenas is modeled using a wide variety of process models which vary over a large range starting from simple algebraic equation systems, to ordinary (ODE) or differential-algebraic (DAE) equations systems, and more complicated (partial-) integro-differential equations. In my thesis, I will limit myself to mathematical process models with fixed horizon, which can be represented as ODE systems by the compressed equation system

$$0 = f(\dot{x}, x(t), u(t), w(t), p, t), \text{ for all } t \in I, \tag{1.1}$$

where $x(t) \in \mathbb{R}^{n_x}$ denotes the differential system state vector. $u(t) \in \mathbb{R}^{n_u}$ are operational variables which can be directly manipulator by process operators, also called the control variables. Modeling uncertainties and disturbances are concentrated without further specifications into a vector function $w(t) \in \mathbb{R}^{n_w}$. $p \in \mathbb{R}^{n_p}$ denotes a vector of time-invariant system parameters. The time interval of interest is denoted in the sequel by $I := [t_0, t_f]$ where $t_0,\ t_f$ are starting and final times respectively.

This model can also be used for simulation. Given particular values of $u^*(t), w^*(t), t \in I, p^*$, where the star indicates that these should be specific but arbitrary values. Using appropriate initial conditions the process model (1.1) can be solved using a suitable integration routine.

Focusing on fixed horizon optimization models given as model predictive control problems, these models consist of two solution phases. First phase is a sensor model, estimating the model parameters based on the data input from physical process. Second phase is the actual model prediction on the time horizon based on the measured model parameters formulated as a dynamic optimization problem.

The softwares of the thesis have both capabilities, but I will only focus on the second part of model predictive control, since this is the most difficult and solver demanding part of model predictive control.

The problem of my interest require the minimization of an objective function by adjusting the free operational variable u, in an appropriate manner within the finite time interval $I^r = [t_0^r, t_f^r]$ which denotes an operational phase of the process, such as the time required for a grade change of a continuous process. Minimizing over a time interval with the purpose of stabilizing some process variable is formulated as an integral over this time interval. The integrator of this integral is called the Lagrange term and is typically given as a weighted Euclidean norm of the difference between the measurements $\mu(t)$, with the particular weighting S. That is

$$\int_{t_0}^{t_f} L(x, u, p, t)dt = \int_{t_0}^{t_f} \|h(x(t), u(t), p, t) - \mu(t)\|_S^2 dt$$

where h is some sort of measurement function.

This is although not always sufficient to describe the optimization process. A penalty term is therefore sometime added to the objective. This penalty term is called the Mayor term and will typically take the final time value as input. For example will a control process minimizing the time horizon of a process consist of such a Mayor term. This will be seen in the simple Optimal Control Problem stated further below in this chapter. The objective is fully described as

$$\min_{x(\cdot),u(\cdot),t_f} M(x(t_f),p) + \int_{t_0}^{t_f} L(x,u,p,t)dt \tag{1.2}$$

The controls u cannot be adjusted arbitrarily, since they might be restricted by constraints which are typically with physical limits such as, e.g. restrictions on valve position.

Further constraints on controls and states comprise, e.g. limits on component capacities and other physical limits. Both types of restrictions are compressed into a general constraint vector function $c(x,u,p,t)$. The constraints c have to be enforced during process operations at any time $t \in I$.

Finally, the model also can consist of initial and final condition on the model variables. These are compressed into the functions $s(x(t_0),p)$ and $r(x(t_f),p)$.

Optimal operations of the process with respect to the specific cost functional could then be achieved through the solution of the following dynamic optimization problem (provided that it is solvable)

$$\min_{x(\cdot),u(\cdot),t_f} M(x(t_f),p) + \int_{t_0}^{t_f} L(x,u,p,t)dt \tag{1.2}$$

$$\begin{aligned}
\text{s.t.} \quad & 0 = f(\dot{x},x(t),u(t),w(t),p,t), \text{ for all } t \in I, && (1.1)\\
& 0 = s(x(t_0),p), && (1.3)\\
& 0 \leq c(x(t),u(t),w(t),p,t), \text{ for all } t \in I, && (1.4)\\
& 0 = r(x(t_f),p) && (1.5)
\end{aligned}$$

A typical weighting matrix S is the inverse of the covariance matrix of the measurement error, but also more general weights like time dependent operators are possible too. The measurements mights as well be included point wise by substituting the integral by a finite sum.

1.2 Example Case

An example of this type of model is the Optimal Rocket Time Problem. The problem is to minimize the travel time t of a rocket in such a way that the rocket have velocity zero at the final destination. I enforce constraints on the positive and negative acceleration of the rocket together with a maximum velocity. I also consider fuel use. The parameters have no parallel to a real case and is chosen for there convenience.

Position is given as s, velocity as v, force as u, and mass as m. The model looks as follows

$$\min_{t_f} \quad t_f \tag{1.6a}$$

$$\text{s.t.} \quad \dot{s}(t) = v(t) \tag{1.6b}$$
$$\dot{v}(t) = \frac{u(t) - 0.2 * v(t)^2}{m(t)} \tag{1.6c}$$
$$\dot{m}(t) = -0.01 * u(t)^2 \tag{1.6d}$$

$$s(t_0) = 0.0 \tag{1.6e}$$
$$v(t_0) = 0.0 \tag{1.6f}$$
$$m(t_0) = 1.0 \tag{1.6g}$$

$$v(t) \leq 1.7 \tag{1.6h}$$
$$v(t) \geq 0.0 \tag{1.6i}$$
$$u(t) \leq 1.1 \tag{1.6j}$$
$$u(t) \geq -1.1 \tag{1.6k}$$

$$s(t_f) = 10.0 \tag{1.6l}$$
$$v(t_f) = 0.0 \tag{1.6m}$$

This model have the following solution

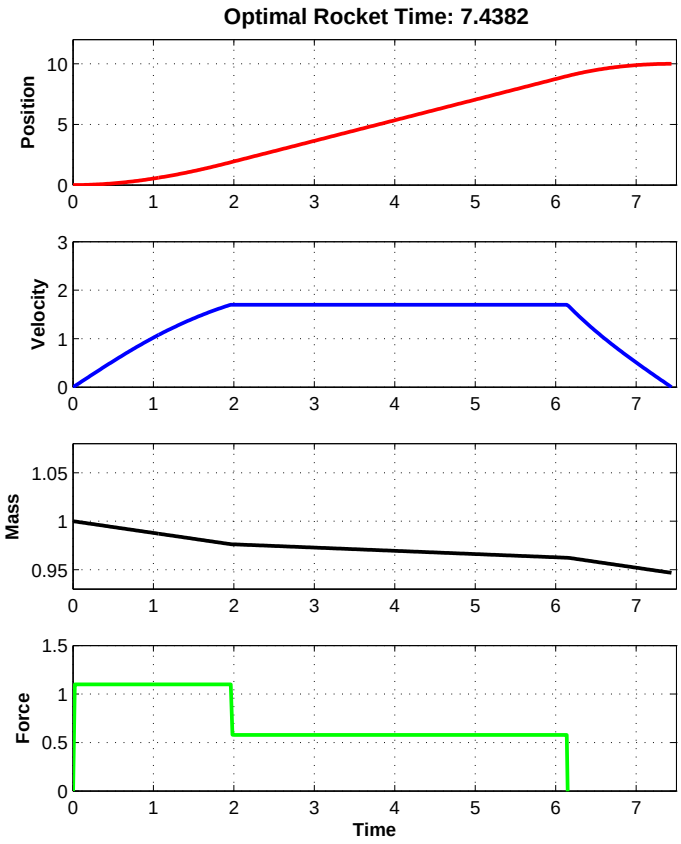


Figure 1.1: Position, Velocity, Mass, and Force of the Optimal Rocket Time Problem

1.3 Direct Solution Algorithms

The basic idea of direct methods for solution of optimal control problems introduces above is to transcribe the original infinite dimensional problem into a finite dimensional Nonlinear Programming Problem (NLP). Two basically different solution strategies for this finite reformulated problem exist.

(i) Sequential optimization: In every iteration step of the optimization method, the model equations are solved exactly by a numerical integration method for the current guess of control parameters.

(ii) Simultaneous optimization: The discretized differential equations enter the transcribed optimization problem as nonlinear constraints that can be violated during the optimization procedures. At the solution, however, they have to be satisfied.

Three methods will be mentioned, which differ in the way the transcription is achieved. These are Direct Single Shooting, - Collocation, and - Multiple Shooting.

Direct Single Shooting represents a pure sequential approach, whereas Collocation is a pure simultaneous approach. Direct Multiple Shooting may be considered a hybrid method, as the model equations are solved "exactly" only on intervals during the solution iterations.

The softwares of this thesis covers all these methods. ACADO Toolkit is a Runge-Kutta based Direct Single Shooting solver. Using IPOPT requires a complete discretized reformulation of ones optimal control problem, since this solver only offers the Direct Collocation method. The descretization of the Quadruple Tank Process model will be covered in the Chapter on IPOPT. JModelica uses IPOPT and thereby uses the Direct Collocation method. JModelica will soon provide an optional Direct Multiple Shooting solver, this option is although not available in current stable version (see https://trac.jmodelica.org/ticket/526 to follow the progress of developing this feature).

Chapter 2

Direct Solution Algorithms

In this chapter, I describe the direct solution methods introduced in chapter 1 in more details.

References: Adaptive multiscale methods [2]

2.1 Direct Solution Algorithms

The basic idea of direct methods for solution of optimal control problems is to transcribe the original infinite dimensional problem into a finite dimensional Nonlinear Programming Problem (NLP). Two basically different solution strategies for this finite reformulated problem exist.

(i) Sequential optimization: In every iteration step of the optimization method, the model equations are solved exactly by a numerical integration method for the current guess of control parameters.

(ii) Simultaneous optimization: The discretized differential equations enter the transcribed optimization problem as nonlinear constraints that can be

violated during the optimization procedures. At the solution, however, they have to be satisfied.

As mentioned in chapter 1, I will focus on three methods. These methods differ in the way the transcription is achieved. The methods are Direct Single Shooting, - Collocation, and - Multiple Shooting.

Direct Single Shooting represents a pure sequential approach, whereas Collocation is a pure simultaneous approach. Direct Multiple Shooting may be considered a hybrid method, as the model equations are solved "exactly" only on intervals during the solution iterations.

2.1.1 Direct Single Shooting

In the direct single shooting method, the infinite many degrees of freedom $u(t)$ for $t \in I$ are reduced by a control parameterization $\tilde{u}(t, q)$ that depends on a finite dimensional vector $q \in \mathbb{R}^{n_q}$. The parameterization of the control can be based on general functions with local or global support or a mixture of both. A representation of the last is a parameterization using a polynomial with N coefficients $q_0, \ldots, q_{N-1}$, given by

$$\tilde{u}(t, q_0, \ldots, q_{N-1}) := \sum_{i=0}^{N-1} q_i t^i, \quad t \in I.$$

Another example is a localized parameterization obtained using a piecewise constant control representation on a partition of the interval I into N subintervals I_i, $i = 0, 1, \ldots, N-1$, such that

$$\tilde{u}(t, q_0, \ldots, q_{N-1}) := q_i, \quad t \in I_i.$$

This last approach is employed by ACADO Toolkit. The parameterizations is illustrated in Figure 2.1.

Besides these two explicit parameterizations of the controls one can also define controls implicitly via additional parameterized ODE's. An example of this is the parameterization

$$\dot{\tilde{u}}(t, \mathbf{q}) := \tilde{f}(x(t), \tilde{u}(t, \mathbf{q}), t, \mathbf{q}), \quad \mathbf{q} = (q_0, \ldots, q_{N-1}), t \in I,$$

$$\tilde{u}(t_0, \mathbf{q}) := \tilde{u}_0(\mathbf{q}), \quad \mathbf{q} = (q_0, \ldots, q_{N-1}).$$

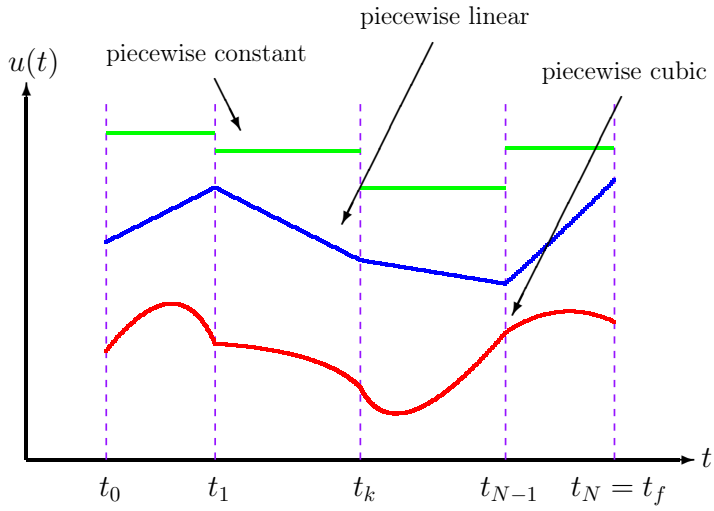


Figure 2.1: Polynomial - and piecewise constant representation of a control.

The additional equations can be added to the model equations (1.2) - (1.5). In this case, the parameterized controls $\tilde{u}$ are reinterpreted as states.

Given an initial value x_0 and a parameter vector $\mathbf{q}$, the following Initial Value Problem can be solved:

$$\dot{x}(t) := \tilde{f}(x(t), \tilde{u}(t, \mathbf{q}), t), \quad \mathbf{q} = (q_0, \ldots, q_{N-1}), t \in I,$$

$$x(t_0) := x_0.$$

The solution of this problem is a trajectory $x(t)$ which is a function of $\mathbf{q}$ only. To keep this dependency in mind we will denote this solution by $\tilde{x}(t, \mathbf{q})$ in the following. By substituting this trajectory into the objective functional defined in (1.2) I can define the cost function $\tilde{J} : \mathbb{R}^{n_q} \to \mathbb{R}$ as

$$\tilde{J}(\mathbf{q}) := M(\tilde{x}(t_f, \mathbf{q})) + \int_{t_0}^{t_f} L(\tilde{x}(t, \mathbf{q}), \tilde{u}(t, \mathbf{q}), t) dt$$

In order to incorporate the inequality constraints $\mathbf{c}$ into the NLP, different methods have been developed. I will mention two approaches.

1. Introduction of a penalty term in the objective function:

$$\hat{J}[u(\cdot), x(\cdot)] := J[u(\cdot), x(\cdot)] + \sum_{j=1}^{n_h} k_j \cdot \int_{t_0}^{t_f} \left(\max(0, -c_j(\cdot))\right)^2 dt$$

   where $k_j \in \mathbb{R}^+, j = 1, \ldots, n_h$ are large positive constants. A difficulty with the max operator is that it hides all information about a constraint as long as it is inactive, and that its smoothness is limited.

2. Using a time grid $t_0 < t_1 < \cdots < t_N = t_f$ the infinite dimensional inequality constraints (1.4) are reformulated into $N + 1$ vector inequality constraints

$$0 \leq \tilde{c}_i(\mathbf{q}) := \mathbf{c}(\tilde{x}(t_i, \mathbf{q}), \tilde{u}(t_i, \mathbf{q}), t_i), \quad i = 0, \ldots, N.$$

   By construction, this method enforces the inequalities constraints at the points on the time grid only. A sufficiently good approximation of the original constraint can be obtained by a sufficiently fine grid. Also a combination with the first method is possible.

In summary, the finite dimensional NLP in the direct single shooting parameterization is given as

$$\min_{q \in \mathbb{R}^{n_q}} \tilde{J}(\mathbf{q})$$

$$\begin{aligned} \text{s.t.} \quad & 0 \leq \tilde{c}_i(\mathbf{q}), \quad i = 0, \ldots, N, \\ & 0 = \tilde{r}(\mathbf{q}). \end{aligned}$$

The numerical effort to solve this NLP is determined to a large extent by the complexity of the parameterization of the control vector. The solution also requires sensitivity information of the states with respect to the control parameter $\mathbf{q}$. The computation of these sensitivities should not be done by trying to generate derivatives by finite differences of independently computed approximations of the solution of disturbed initial value problems, but rather by the principle of *Internal Numerical Differentiation*. Many ODE and DAE solvers exist that can efficiently compute sensitivities according to the principle of *Internal Numerical Differentiation*.

2.1.2 Direct Multiple Shooting

In the direct multiple shooting method, the transcription of the optimal control problem into an NLP starts similar to the direct single shooting method with a

local control representation. First, the time horizon $I = [t_0, t_f]$ is divided into N subintervals $i_i := [t_i, t_{i+1}],\ i = 0, 1, \ldots, N-1$, with $t_0 < t_1 < \cdots < t_n = t_f$. Then, the control trajectory is parameterized by a piecewise representation

$$\tilde{u}_i(t, \mathbf{q}_i) \text{ for } t \in [t_i, t_{i+1}]$$

with N local control parameter vectors $\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_{N-1},\ \mathbf{q}_i \in \mathbb{R}^{n_q}$. The trivial example for such a parameterization is again the piecewise constant representation shown in Figure 2.1.

In a crucial second step, $N+1$ additional vectors $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_N$ of the same dimension n_x as the system state are introduced. I will refer to these as the multiple shooting node values. All but the last serve as initial value for N independent Initial Value Problems on the interval I_i:

$$\dot{x}_i(t) := f(x_i(t), \tilde{u}_i(t, \mathbf{q}_i), t), \quad t \in [t_i, t_{i+1}],$$

$$x_i(t_i) := \mathbf{s}_i.$$

The solution of these problems are N independent trajectories $x_i(t)$ on $[t_i, t_{i+1}]$, which are a function of $\mathbf{s}_i$ and $\mathbf{q}_i$ only. I will denote these solutions by $\tilde{x}_i(t, \mathbf{s}_i, \mathbf{q}_i)$. These solutions are illustrated in Figure 2.2. By substituting the independent

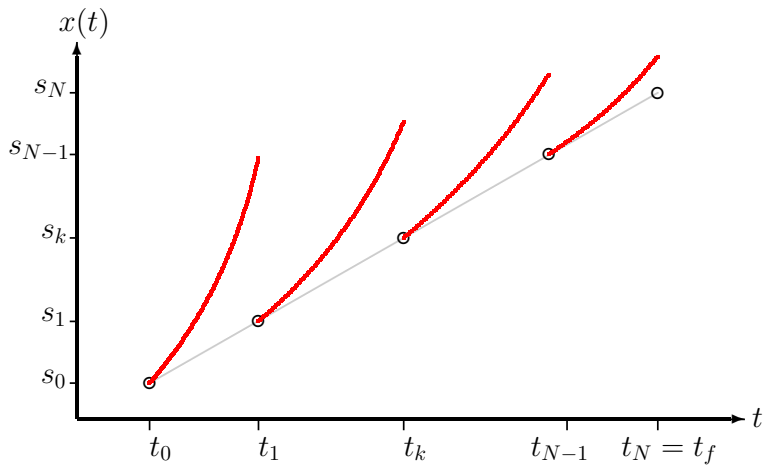


Figure 2.2: Trajectories in the multiple shooting parameterization.

trajectories $\tilde{x}_i(t, \mathbf{s}_i, \mathbf{q}_i)$ into the Lagrange term L in Equation (1.2) I can calculate the objective contributions $\tilde{J}_i : \mathbb{R}^{n_x} \times \mathbb{R}^{n_q} \to \mathbb{R}$ for $i = 0, \ldots, N-1$

as

$$\tilde{J}_i(\mathbf{s}_i, \mathbf{q}_i) := \int_{t_i}^{t_{i+1}} L(\tilde{x}_i(t, \mathbf{s}_i, \mathbf{q}_i), \tilde{u}_i(t, \mathbf{q}_i), t)dt$$

The decoupled Initial Value Problems are connected by matching conditions which require that each node value should equal the final value of the preceding trajectory:

$$\mathbf{s}_{i+1} = \tilde{x}_i(t_{i+1}, \mathbf{s}_i, \mathbf{q}_i), \quad i = 0, \ldots, N-1. \tag{2.1}$$

The first multiple shooting node variable $\mathbf{s}_0$ is required to be equal to the initial value x_0 of the optimization problem:

$$\mathbf{s}_0 = x_0. \tag{2.2}$$

Together, the constraints (2.1) and (2.2) remove the additional degrees of freedom which were introduced with the parameters $\mathbf{s}_i$, $i = 0, \ldots, N$. It is by no means necessary that the constraints (2.1) and (2.2) are satisfied during the optimization iterations. On the contrary, it is a crucial feature of the direct multiple shooting method that it can deal with infeasible initial guesses of the variables $\mathbf{s}_i$ and $\mathbf{q}_i$.

Using the same grid as for the multiple shooting parameterization, the infinite dimensional path inequality constraints (1.4) are transcribed into $N+1$ vector inequalities constraints

$$0 \leq \tilde{c}_i(\mathbf{s}_i, \mathbf{q}_i) := \mathbf{c}(\mathbf{s}_i, \tilde{u}_i(t_i, \mathbf{q}_i), t_i), \quad i = 0, \ldots, N.$$

In summary, the finite dimensional NLP in the direct multiple shooting parameterization is given as

$$\begin{aligned} \min_{s_0,\ldots,s_N,q_0,\ldots,q_{N-1}} \; & M(\mathbf{s}_N) + \sum_{i=0}^{N-1} \tilde{J}_i(\mathbf{s}_i, \mathbf{q}_i) \\ \text{s.t.} \quad & \mathbf{s}_{i+1} = \tilde{x}_i(t_{i+1}, \mathbf{s}_i, \mathbf{q}_i), \quad i = 0, \ldots, N-1, \\ & \mathbf{s}_0 = x_0, \\ & 0 \leq \tilde{c}_i(\mathbf{s}_i, \mathbf{q}_i), \quad i = 0, \ldots, N, \\ & 0 = r(\mathbf{s}_N). \end{aligned}$$

An important feature of the direct multiple shooting method is the sparse structure of this large scale NLP. Its Hessian matrix $\nabla^2_{s,q}\mathcal{L}$ is block diagonal with non-zero blocks $\nabla^2_{s_i,q_i}\mathcal{L}$ that correspond to local variables $\mathbf{s}_i, \mathbf{q}_i$ only.

2.1.3 Direct Collocation

Last, I consider a general direct collocation discretization of the optimal control problem Equations (1.2) - (1.5). For simplicity, I assume that the functional Equation (1.2) is in Mayer form $J[u,x] = M(x(t_f))$. This is no restriction of generality, as the transcription of the Bolza functional (1.2) to Mayer form is easily done.

As a first step, an additional state x_{n_x+1} and an additional differential equation

$$\dot{x}_{n_x+1}(t) = L(x(t), u(t), t), \quad x_{n_x+1}(t_0) := 0$$

are introduced. In the second step, the objective $M(x(t_f))$ is redefined as $M(x(t_f)) + x_{n_x+1}(t_f)$.

Both state and control variables are approximated by piecewise defined functions $\tilde{x}(t,\cdot)$ and $\tilde{u}(t,\cdot)$ on the time grid

$$t_0 < t_1 < \cdots < t_{N+1} = t_f.$$

Within each collocation interval $[t_i, t_{i+1}[$, $0 \le i \le N$, these functions are chosen as parameter dependent polynomials of order $k, l \in \mathbb{N}$ respectively:

$$\tilde{x}(t,\mathbf{s})|_{[t_i,t_{i+1}[} := \tilde{x}_i(t,\mathbf{s}_i) := \pi_i^x(t,\mathbf{s}_i) \in \Pi_k^{n_x},$$

$$\tilde{u}(t,\mathbf{q})|_{[t_i,t_{i+1}[} := \tilde{u}_i(t,\mathbf{q}_i) := \pi_i^u(t,\mathbf{q}_i) \in \Pi_l^{n_c}.$$

Here, Π_μ^ν denotes the space of ν-dimensional vectors of polynomials up to degree μ.

The coefficients of the polynomials are collected in the vectors

$$\mathbf{s} := (\mathbf{s}_0^T, \ldots, \mathbf{s}_N^T)^T \in \mathbb{R}^{N\cdot(k+1)\cdot n_x},\ \mathbf{s}_i \in \mathbb{R}^{(k+1)\cdot n_x}, i = 0,\ldots,N,$$

$$\mathbf{q} := (\mathbf{q}_0^T, \ldots, \mathbf{q}_N^T)^T \in \mathbb{R}^{N\cdot(k+1)\cdot n_c},\ \mathbf{q}_i \in \mathbb{R}^{(k+1)\cdot n_c}, i = 0,\ldots,N.$$

Matching conditions of the form

$$\pi_i(t_{i+1}^-,\cdot) = \pi_i(t_{i+1}^+,\cdot), \quad i = 0,\ldots,N-1$$

have to be imposed at the boundaries of the subintervals to enforce continuity of the approximating functions in $[t_0, t_f]$. Additionally, higher order differentiability may be imposed by

$$\frac{d^k}{dt^k}\pi_i(t_{i+1}^-,\cdot) = \frac{d^k}{dt^k}\pi_{i+1}(t_{i+1}^+,\cdot), \quad \begin{cases} k = 1,\ldots,J \\ i = 0,\ldots,N-1 \end{cases}$$

where J denotes the desired order of differentiability.

In order to formulate a nonlinear optimization problem, the model equations and the continuous constraints are explicitly discretized:

1. The model Equations (1.1) are only to be satisfied at the collocation points $t_{i,\mu}$, $\mu = 1, \ldots, M$, within each subinterval $[t_i, t_{i+1}[$, $i = 0, \ldots, N-1$, and within $[t_N, t_{N+1}]$:

$$t_i \leq t_{i0} < \cdots < t_{iM} < t_{i+1}, \quad i = 0, \ldots, N-1$$

$$t_N \leq t_{N0} < \cdots < t_{NM} \leq t_{N+1}.$$

2. The inequality constraints $c(\cdot)$ are sampled on a second grid within $[t_0, t_f]$:

$$t_0 \leq t_1^c < \cdot < t_L^c \leq t_f$$

Altogether, this leads to the formulation of the discretized optimal control problem derived from (1.2) - (1.5) (in Mayer form) by collocation:

$$\min_{s,q} \quad \tilde{M}(\mathbf{s}) = M(\tilde{x}(t_f, \mathbf{s})) \tag{2.3a}$$

$$\text{s.t.} \quad f(\tilde{x}(t_{il}, \mathbf{s}), \tilde{u}(t_{il}, \mathbf{q}), t) - \dot{\tilde{x}}(t_{il}, \mathbf{s}) = 0, \quad \begin{cases} i = 0, \ldots, N \\ l = 0, \ldots, M \end{cases}, \tag{2.3b}$$

$$c(\tilde{x}(t_\gamma^c, \mathbf{s}), \tilde{u}(t_\gamma^c, \mathbf{q}), t_\gamma^c) \geq 0, \quad \gamma = 1, \ldots, L, \tag{2.3c}$$

$$\tilde{x}(t_0, \mathbf{s}) - x_0 = 0, \tag{2.3d}$$

$$r(\tilde{x}(t_f, \mathbf{s})) = 0. \tag{2.3e}$$

If the solution is restricted to continuously differentiable state and control variables, the matching conditions have to be fulfilled additionally:

$$\frac{d^k}{dt^k}\pi_i^x(t_{i+1}^-, \mathbf{s}_i) - \frac{d^k}{dt^k}\pi_{i+1}^x(t_{i+1}^+, \mathbf{s}_{i+1}) = 0, \quad \begin{cases} k = 1, \ldots, J_s \\ i = 0, \ldots, N-1 \end{cases} \tag{2.4a}$$

$$\frac{d^k}{dt^k}\pi_i^u(t_{i+1}^-, \mathbf{q}_i) - \frac{d^k}{dt^k}\pi_{i+1}^u(t_{i+1}^+, \mathbf{q}_{i+1}) = 0, \quad \begin{cases} k = 1, \ldots, J_c \\ i = 0, \ldots, N-1 \end{cases} \tag{2.4b}$$

where J_s is the order of differentiability in the state variables and J_s is the order of differentiability in the control variables.

The constrained nonlinear optimization problems Equations (2.3a)-(2.3e), (2.4a)-(2.4b) can be efficiently solved using SQP algorithms[4]. SQP methods are based on the availability of gradient information.

Due to the full discretization of both control and state space, the NLPs generated by direct collocation tend to become very large for practically interesting problems. Thus, special care has to be taken in the implementation of a collocation algorithm to account for the special structure and the high sparsity of the constraints Equations (2.3a)-(2.3e), (2.4a)-(2.4b).

Chapter 3

The Quadruple Tank Process

In this chapter, I present the quadruple tank process and develop a mathematical model that describes its dynamics. This model will act as a benchmark model for testing the various solvers and softwares. This means that this model will be implemented in each of the softwares presented in the coming chapters, such that outputs and software capabilities can be compared on the same foundation. The outputs below will act as a the correct solution of the model with respect to the parameters set for the model throughout the thesis.

The nature of the quadruple tank process is to some extent fabricated, its purpose is solely to provide a theoretical example of an optimal control problem, meaning that the problem enables me to model simple processes using differential equations which can be solved using various solving tools.

References: Constrained Predictive Control - A Computational Approach [3]

3.1 The Quadruple Tank Process

The quadruple tank system consist of four interconnected tanks containing a flow of a common water source using pumps an valves. The water flow can be manipulated be altering the valves position in the tank setup. The setup can be viewed in Figure 3.1. The variables F_1 and F_2 control the flow from the

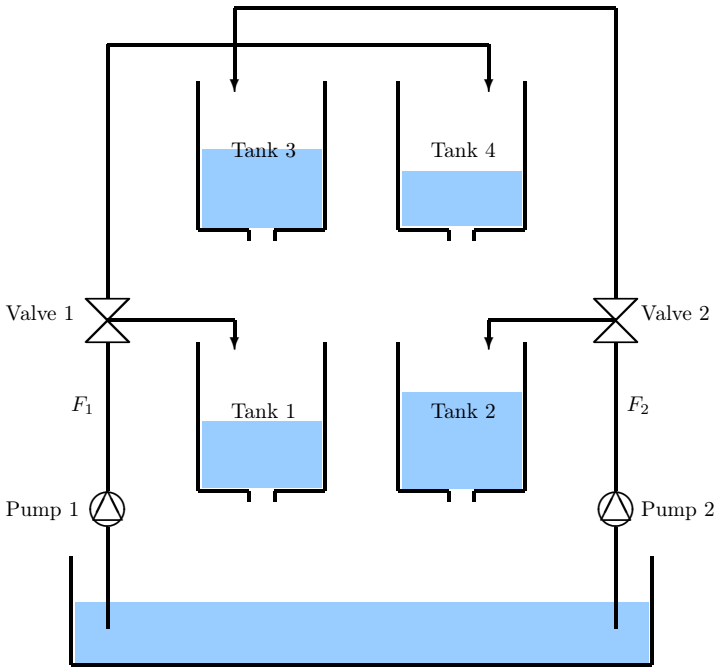


Figure 3.1: Diagram of the quadruple tank process

pumps as seen in Figure 3.1. I call these the manipulated variables and represent them as the vector u. For measuring the state of the the system the process is equipped with water level sensors in each tank. The vector y will represent these measuring variables given as the height h_i in each tank $i \in \{1, 2, 3, 4\}$. The mechanical nature of the sensors also add the notion of measuring noise to the output values y. z is then the output I want to control. The control element of the problem is the water level in tank 1 and tank 2, in the sense that I wish to stabilize these water levels.

For the manipulated variables u, measuring variables y and the control variables

z assuming an ideal environment without noise I then state

$$u = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \quad y = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} \quad z = \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

Let the state of the system be denoted x. As the process evolve in continuous time, the process modeled using differential equations. The model formulation will use ordinary differential equations and can be stated as 3.1a

$$\frac{dx(t)}{dt} = f(x(t), u(t)) \quad x(t_0) = x_0 \tag{3.1a}$$
$$y(t) = g(x(t)) \tag{3.1b}$$
$$z(t) = h(x(t)) \tag{3.1c}$$

$t \in \mathbb{R}$ is time, $x \in \mathbb{R}^{n_x}$ is the state vector, $u \in \mathbb{R}^{n_u}$ is the vector of manipulated variables, $y \in \mathbb{R}^{n_y}$ is the measured (observed) values, and $z \in \mathbb{R}^{n_x}$ is the outputs. $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \mapsto \mathbb{R}^{n_x}$ is the model of the system, $g : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_y}$ is the sensor function, and $h : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_x}$ is the output function. As seen, f does not depend directly on the time t. This kind of process is thus called an *autonomous* system.

3.1.1 Modeling a Single Tank

The four tank system will be modeled using the mass balance of water in each tank and the valves will be placed as seen in figure 3.1.
The mass balance principle gives me that the mass of water in the system is conserved. For a single tank this gives the water balance equation

$$\text{Accumulated} = \text{In} - \text{Out}$$

Let Δt be a step in time, such that $[t, t + \Delta t]$ is the time interval in which the flow rate to and from the tank can be considered constant. For $\Delta t \to 0$ the approximated time interval will eventually recede.
Let $m_1(t)$ be the mass of water in tank 1, ρ [g/cm^3] be the density of water, q_1^{in} [cm^3/s] the volumetric flow rate into tank 1 from valve 1, q_3 [cm^3/s] the volumetric flow rate from tank 3 into tank 1, and q_1 [cm^3/s] the volumetric flow rate out of tank 1 (see figure 3.2). Then

$$\begin{aligned} \text{Accumulated} &= m_1(t + \Delta t) - m_1(t) \\ \text{In} &= \rho q_1^{in}(t)\Delta t + \rho q_3(t)\Delta t \\ \text{Out} &= \rho q_1(t)\Delta t \end{aligned}$$

with the mass balance

$$\underbrace{m_1(t+\Delta t) - m_1(t)}_{\text{Accumulated}} = \underbrace{\rho q_1^{in}(t)\Delta t + \rho q_3(t)\Delta t}_{\text{In}} - \underbrace{\rho q_1(t)\Delta t}_{\text{Out}}$$

Dividing with Δt this yields

$$\frac{m_1(t+\Delta t) - m_1(t)}{\Delta t} = \rho q_1^{in}(t) + \rho q_3(t) - \rho q_1(t)$$

By letting $\Delta t \to 0$ the balance equation finally becomes

$$\frac{d}{dt}\left(m_1(t)\right) = \rho q_1^{in}(t) + \rho q_3(t) - \rho q_1(t) \tag{3.2}$$

It is here noted that the differential operator is defined as

$$\frac{d}{dt}\left(m_1(t)\right) \triangleq \lim_{\Delta t \to 0} \frac{m_1(t+\Delta t) - m_1(t)}{\Delta t} \tag{3.3}$$

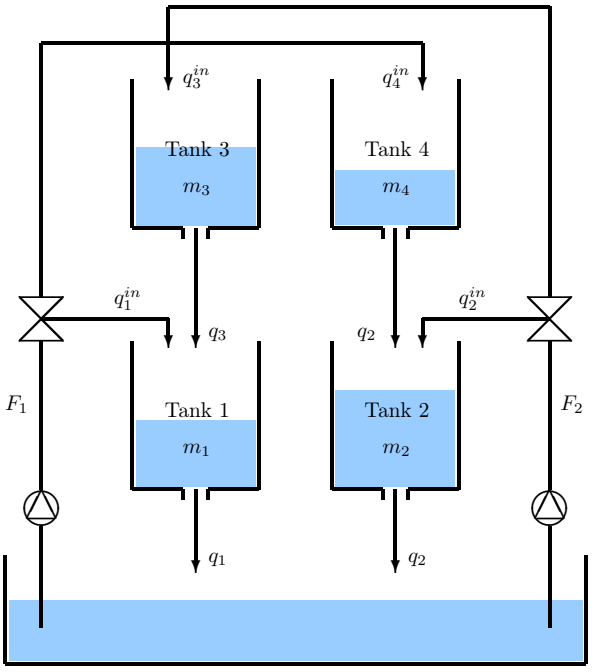


Figure 3.2: Diagram of the quadruple tank process

3.1.2 Mass Balances for the Four Tanks

As deduced above the mass balance principle now gives me that I can formulate the mass balance for each of the four tanks as differential equations

$$\frac{d}{dt}\left(m_1(t)\right) = \rho q_1^{in}(t) + \rho q_3(t) - \rho q_1(t) \tag{3.4a}$$
$$\frac{d}{dt}\left(m_2(t)\right) = \rho q_2^{in}(t) + \rho q_4(t) - \rho q_2(t) \tag{3.4b}$$
$$\frac{d}{dt}\left(m_3(t)\right) = \rho q_3^{in}(t) - \rho q_3(t) \tag{3.4c}$$
$$\frac{d}{dt}\left(m_4(t)\right) = \rho q_4^{in}(t) - \rho q_4(t) \tag{3.4d}$$

The initial values for the m_i's, that is the mass of water in each tank at time t_0 are

$$m_1(t_0) = m_{1,0} \tag{3.5a}$$
$$m_2(t_0) = m_{2,0} \tag{3.5b}$$
$$m_3(t_0) = m_{3,0} \tag{3.5c}$$
$$m_4(t_0) = m_{4,0} \tag{3.5d}$$

I now lack to formulate the inflows and outflows (the variables q) in an appropriate mathematical way.

3.1.3 Inflows

The flow rate from the valves into each of the four tanks are obtained using static balance around each of the two valves. Let γ_1 be a constant expressing the fraction of water from pump 1 that flows into tank 1

$$\gamma_1 = \frac{\rho q_1^{in}(t)}{\rho F_1(t)} = \frac{q_1^{in}(t)}{F_1(t)} \tag{3.6}$$

Since mass flowing into valve 1 equals the mass flowing out of valve 1, I have

$$\rho F_1(t) = \rho q_1^{in}(t) + \rho q_3(t)$$

which implies

$$q_4^{in}(t) = F_1(t) - q_1^{in}(t) = F_1(t) - \gamma_1 q_1^{in}(t) = (1-\gamma_1)F_1(t)$$

This then determines the flow rates for $\gamma_1^{in}(t)$ and $\gamma_4^{in}(t)$ as

$$q_1^{in}(t) = \gamma_1 F_1(t) \tag{3.7a}$$
$$q_4^{in}(t) = (1-\gamma_1) F_1(t) \tag{3.7b}$$

Similarly defining γ_2 as the constant expressing the fraction of water from pump 2 that flows into tank 2

$$\gamma_2 = \frac{\rho q_2^{in}(t)}{\rho F_2(t)} = \frac{q_2^{in}(t)}{F_2(t)} \tag{3.8}$$

implying

$$q_2^{in}(t) = \gamma_2 F_2(t) \tag{3.9a}$$
$$q_3^{in}(t) = (1-\gamma_2) F_2(t) \tag{3.9b}$$

Finally, this enables me to express the flow rates into each of the tanks from the valves by the equations

$$q_1^{in}(t) = \gamma_1 F_1(t) \tag{3.10a}$$
$$q_2^{in}(t) = \gamma_2 F_2(t) \tag{3.10b}$$
$$q_3^{in}(t) = (1-\gamma_2) F_2(t) \tag{3.10c}$$
$$q_4^{in}(t) = (1-\gamma_1) F_1(t) \tag{3.10d}$$

3.1.4 Outflows

Each of the four tanks are equipped with a drain in the bottom in the form of a small pipe. The water flows out of these drains completely controlled by gravity. This water flow can therefore be described by Bernoulli's Principle. Bernoulli's Principle states that mechanical energy of fluid along a stream line is conserved, i.e. the sum of potential energy, kinetic energy, and work remains constant along the stream line

$$\rho g h + \frac{1}{2}\rho v^2 + p = \text{constant}$$

ρ is the density of the fluid, h is the liquid height above a baseline, v is the velocity, and p is the pressure.
Again I consider the dynamics of tank 1. Let h_1 be the liquid height above a baseline in tank 1, where the baseline is the outlet of the tank. The pressure at both the surface of the water and the pipe outlet is the same, and more precisely it equals atmospheric pressure. Consider v_1 as the linear velocity [m/s] of water

flowing in the outlet pipe from tank 1. I now consider the velocity at the top of the tank given as $v_{1,top}$. I then have that $v_{1,top} \ll v_1$ as $a_1 \ll A_1$, in which a_1 is the cross section area of the pipe and A_1 is the cross section area of the tank, i.e. the water flows much faster through the outlet pipe than it sinks in the tank. Then Bernoulli's equation applied to the liquid level in the top and the pipe outlet gives

$$\rho g h_1 + \frac{1}{2}\rho v_{1,top}^2 + p = \rho g 0 + \frac{1}{2}\rho v_1^2 + p = \frac{1}{2}\rho v_1^2 + p$$

such that

$$v_1 = \sqrt{v_{1,top}^2 + 2gh_1} \approx \sqrt{2gh_1}$$

The volumetric flow rate in the outlet pipe from tank 1 is thus

$$q_1 = a_1 v_1 = a_1\sqrt{2gh_1} \tag{3.11}$$

For the whole system I then get the following volumetric flow rates when applying Bernoulli's Principle

$$q_1 = a_1\sqrt{2gh_1} \tag{3.12a}$$
$$q_2 = a_2\sqrt{2gh_2} \tag{3.12b}$$
$$q_3 = a_3\sqrt{2gh_3} \tag{3.12c}$$
$$q_4 = a_4\sqrt{2gh_4} \tag{3.12d}$$

I will further assume that the cross section areas of the tanks is unchanged depending on the height of the water levels. Given this, I have that the volume in tank $i \in \{1, 2, 3, 4\}$ is

$$V_i = A_i h_i \quad i \in \{1, 2, 3, 4\},$$

implying that the mass of water is described as

$$m_i = \rho V_i = \rho A_i h_i \quad i \in \{1, 2, 3, 4\}$$

and last having the liquid height for each tank given as

$$h_1 = \frac{m_1}{\rho A_1} \tag{3.13a}$$
$$h_2 = \frac{m_2}{\rho A_2} \tag{3.13b}$$
$$h_3 = \frac{m_3}{\rho A_3} \tag{3.13c}$$
$$h_4 = \frac{m_4}{\rho A_4} \tag{3.13d}$$

3.1.5 The Complete Model of the Four Tank System

The complete model to simulate the four tank system consist of the equations for the flow from the valves to each tank (3.10), the relations for the liquid heights (3.13), the relations for the outlet flow rates (3.12), the differential equations (3.4), and their initial conditions (3.5).

Flow rates from the valves

$$q_1^{in} = \gamma_1 F_1 \tag{3.10a}$$
$$q_2^{in} = \gamma_2 F_2 \tag{3.10b}$$
$$q_3^{in} = (1-\gamma_2) F_2 \tag{3.10c}$$
$$q_4^{in} = (1-\gamma_1) F_1 \tag{3.10d}$$

Liquid heights

$$h_1 = \frac{m_1}{\rho A_1} \tag{3.13a}$$
$$h_2 = \frac{m_2}{\rho A_2} \tag{3.13b}$$
$$h_3 = \frac{m_3}{\rho A_3} \tag{3.13c}$$
$$h_4 = \frac{m_4}{\rho A_4} \tag{3.13d}$$

Flow rates out of each tank

$$q_1 = a_1 \sqrt{2gh_1} \tag{3.12a}$$
$$q_2 = a_2 \sqrt{2gh_2} \tag{3.12b}$$
$$q_3 = a_3 \sqrt{2gh_3} \tag{3.12c}$$
$$q_4 = a_4 \sqrt{2gh_4} \tag{3.12d}$$

Mass balances

$$\frac{d}{dt}\left(m_1(t)\right) = \rho q_1^{in}(t) + \rho q_3(t) - \rho q_1(t) \tag{3.4a}$$
$$\frac{d}{dt}\left(m_2(t)\right) = \rho q_2^{in}(t) + \rho q_4(t) - \rho q_2(t) \tag{3.4b}$$
$$\frac{d}{dt}\left(m_3(t)\right) = \rho q_3^{in}(t) - \rho q_3(t) \tag{3.4c}$$
$$\frac{d}{dt}\left(m_4(t)\right) = \rho q_4^{in}(t) - \rho q_4(t) \tag{3.4d}$$

initial conditions for the state variables

$$m_1(t_0) = m_{1,0} \tag{3.5a}$$
$$m_2(t_0) = m_{2,0} \tag{3.5b}$$
$$m_3(t_0) = m_{3,0} \tag{3.5c}$$
$$m_4(t_0) = m_{4,0} \tag{3.5d}$$

3.1.6 The Objective of the Four Tank System

The four tank system is to be considered as a deterministic optimal control problem in the Bolza form on a fixed horizon $I := [t_0, t_f]$ with

$$\phi = M(x(t_f)) + \int_{t_0}^{t_f} L(x, u, t)dt$$

subject to the initial conditions and differential state equations above.

Since my optimization goal is to keep the levels in tank 1 and 2, h_1 and h_2, at the set points, μ_1 and μ_2, I will represent the Lagrange term as a weighted Euclidean norm of the difference between these (equal weights) giving

$$\begin{aligned}\phi &= \textstyle\int_{t_0}^{t_f} \lambda_1 \|h_1(t) - \mu_1\|^2 + \lambda_2 \|h_2(t) - \mu_2\|^2 dt \\ &= \textstyle\int_{t_0}^{t_f} \lambda_1 (h_1(t) - \mu_1)^2 + \lambda_1 (h_2(t) - \mu_2)^2 dt\end{aligned}$$

In my case the variables $h_1(t)$ and $h_2(t)$ is dependent both of the masses $m_1(t)$, $m_2(t)$ and the control variables $F_1(t)$ and $F_2(t)$ thereby determining the whole system. Since I value the balance of h_1 and h_2 equal, I will use unit weights such that

$$\phi = \int_{t_0}^{t_f} (h_1(t) - \mu_1)^2 + (h_2(t) - \mu_2)^2 dt$$

3.1.7 The Four Tank System stated as an optimal control problem

The entire optimal control problem of the Four Tank System can then be summarized as

$$\min_{F_1,F_2,m_1,m_2,t} \quad \phi(t) = \int_{t_0}^{t_f} (h_1(t)-\mu_1)^2 + (h_2(t)-\mu_2)^2 dt \tag{3.19a}$$

$$\begin{aligned}
\text{s.t.} \quad \dot{m}_1(t) &= \rho q_1^{in}(t) + \rho q_3(t) - \rho q_1(t) && (3.19b)\\
\dot{m}_2(t) &= \rho q_2^{in}(t) + \rho q_4(t) - \rho q_2(t) && (3.19c)\\
\dot{m}_3(t) &= \rho q_3^{in}(t) - \rho q_3(t) && (3.19d)\\
\dot{m}_4(t) &= \rho q_4^{in}(t) - \rho q_4(t) && (3.19e)\\
h_1(t) &= \frac{m_1(t)}{\rho A_1} && (3.19f)\\
h_2(t) &= \frac{m_2(t)}{\rho A_2} && (3.19g)\\
h_3(t) &= \frac{m_3(t)}{\rho A_3} && (3.19h)\\
h_4(t) &= \frac{m_4(t)}{\rho A_4} && (3.19i)\\
q_1^{in}(t) &= \gamma_1 F_1(t) && (3.19j)\\
q_2^{in}(t) &= \gamma_2 F_2(t) && (3.19k)\\
q_3^{in}(t) &= (1-\gamma_2) F_2(t) && (3.19l)\\
q_4^{in}(t) &= (1-\gamma_1) F_1(t) && (3.19m)\\
q_1(t) &= a_1\sqrt{2gh_1(t)} && (3.19n)\\
q_2(t) &= a_2\sqrt{2gh_2(t)} && (3.19o)\\
q_3(t) &= a_3\sqrt{2gh_3(t)} && (3.19p)\\
q_4(t) &= a_4\sqrt{2gh_4(t)} && (3.19q)
\end{aligned}$$

$$m_1(t_0) = m_{1,0} \tag{3.20a}$$
$$m_2(t_0) = m_{2,0} \tag{3.20b}$$
$$m_3(t_0) = m_{3,0} \tag{3.20c}$$
$$m_4(t_0) = m_{4,0} \tag{3.20d}$$
$$F_1(t) \leq F_{1,max} \tag{3.20e}$$
$$F_2(t) \leq F_{2,max} \tag{3.20f}$$
$$h_1(t) \leq h_{1,max} \tag{3.20g}$$
$$h_2(t) \leq h_{2,max} \tag{3.20h}$$
$$h_3(t) \leq h_{3,max} \tag{3.20i}$$
$$h_4(t) \leq h_{4,max} \tag{3.20j}$$
$$F_1(t) \geq F_{1,min} \tag{3.20k}$$
$$F_2(t) \geq F_{2,min} \tag{3.20l}$$
$$h_1(t), h_2(t), h_3(t), h_4(t) \geq 0 \tag{3.20m}$$
$$m_1(t), m_2(t), m_3(t), m_4(t) \geq 0 \tag{3.20n}$$

In order to have a sound foundation to compare the different softwares, I will defined a fixed set of parameters for this model. These parameters will then be used together with the model to test the softwares.

First of all, I set the physical dimensions of the tanks. The tanks will be of equal size and have a cross section area of 380.1327 cm^2. The hight of the tanks is 20.0 cm and have an outlet cross section area of 1.2272 cm^2.

The pumps has a maximum capacity of 300 cm^3/s and is restricted to have positive flow.

The valves in the system will have a fixed position. In order to created a more interesting dynamic in the system, I will set the two valve position at two different position, namely at 0.15 and 0.25 on a scale from 0 to 1.

The natural constant of the liquid density in the system will be that of water. Hence a density of 1.00 g/cm^3. The acceleration of gravity is 981 cm/s^2

I aim at stabilizing the water levels in tank 1 and 2 at 12 cm.

The exact solution of this system is then

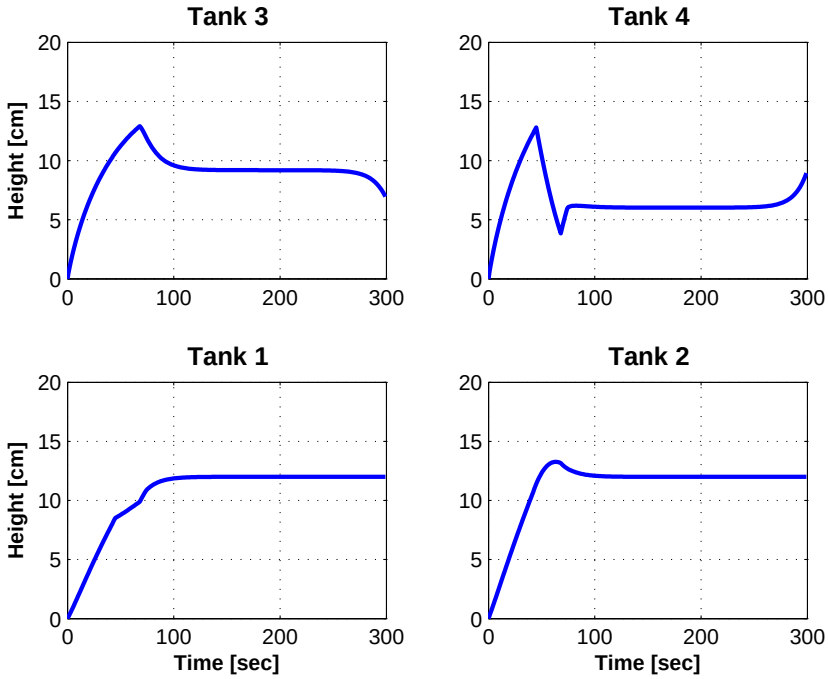


Figure 3.3: Height of liquid in tanks

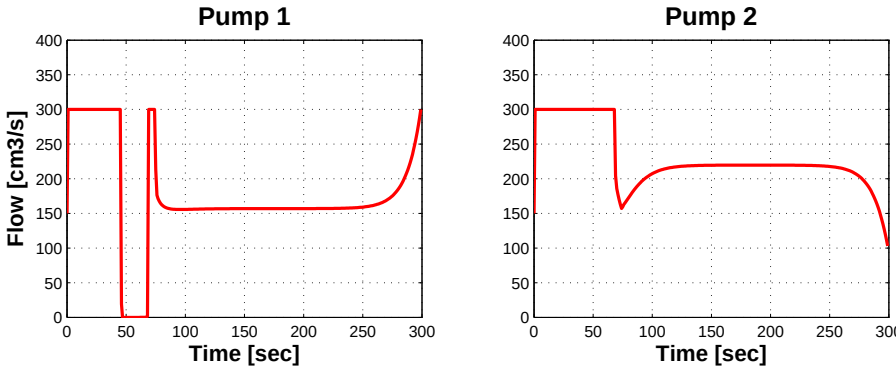


Figure 3.4: Control variables (pump flow)

It is seen that the solution leaves the stable state towards the end of the time interval. This is caused by the fact that I have a fixed time horizon. Since the model does not have to stabilize the system beyond the time horizon, the pumps can be regulated, without the water level of tank 1 and tank 2 is effected, within the time horizon.

Chapter 4

ACADO Toolkit

In this chapter, I introduce the optimization and simulation software ACADO Toolkit. A short introduction of the origin and main features will be provided. Afterwards, I will go through the installation of the software, providing installation procedures for both Linux, Mac OS X and Windows. Then a guide to producing and running a simple program will be given. Finally, I will go through all elements on how to simulate and solve the Quadruple Tank Process model using this software, also providing and explaining the outputs and plots of the model compilation.

References: Sequential Quadratic Programming [4], Applications in Renewable Energy Systems [6], ACADO Toolkit - An Open-Source Framework [7], Fast Pareto set generation [8], Embedded Optimisation [9], Scientific - and Technical Software [10]

URL: [ACADO Home] http://www.acadotoolkit.org/

4.1 ACADO Toolkit - Getting Started

The ACADO Toolkit is a general Open-Source framework for using a variety of algorithms i order to solve dynamic optimal control problem. These problems include model predictive control, state and parameter estimation and general non-linear optimization. The software environment of ACADO Toolkit is written as object oriented C++ code language, which makes it ideal for the user to implement self-written optimization routines as extensions. The program is used by means of a unix user interface. ACADO Toolkit is provided under a GNU General Public License and under a GNU Lesser General Public License. The later of these licenses provide additional user rights and is even less restrictive than the GNU General Public License. These licenses include the freedom to use the software for any purpose, the freedom to change the software to suit ones needs, the freedom to share the software with anyone, and the freedom to share the changes you make.

4.1.1 Installation

Before demonstrating, an installation guide will be provided for the most common used operating systems.

4.1.1.1 Installation under Linux

In order to install and run ACADO Toolkit models you need to make sure that a C++ compiler is installed on the system. This can be checked by prompting the unix command

```
> g++ -v
```

For graphical outputs Gnuplot is needed to be installed, however ACADO Toolkit will run without, the plots will simply be turned of. This can be checked by prompting the unix command

```
> which gnuplot
```

If Gnuplot or GCC is not installed, I recommendation using the APT repository installation tool (Advanced Packages Tool, http://www.apt-get.org/). This tool is preinstalled on UBUNTU. Alternatively the required compilers and softwares source files can be downloaded at http://sourceforge.net/projects.

Using APT, begin by making a software update of the package tool. This is done by the following prompt commands

```
> sudo apt-get update
> sudo apt-get install build-essential
```

This action should install GCC, else prompt

```
> sudo apt-get install g++
```

Gnuplot can then be installed using the prompt command

```
> sudo apt-get install gnuplot
```

The ACADO Toolkit program package can now be installed by going to

http://acadotoolkit.org/download/ACADOtoolkit-1.0beta.tar.gz.

Go to your download folder and extract the files

```
> tar xfvz ACADOtoolkit-1.0beta.tar.gz
```

Go to the directory `ACADOtoolkit-1.0beta` and build the package

```
> cd ACADOtoolkit-1.0beta
> make
```

4.1.1.2 Installation under Mac OS X

The dependencies is the same as on Linux and the same unix command can to some extend be reused using the Mac Terminal. Installing Gnuplot can be accompliced by installing MacPorts (http://www.macports.org/). Like APT this is a package tool. A requirement is that Apple's Xcode Developer Tool is installed on your system. Installing MacPorts will take some time and requires about 460 MB space. Gnuplot can use X11 as graphics terminal (X-server), but AquaTerm is the default graphics terminal and should therefore be installed before installing Gnuplot. Using macPorts, then prompt

```
> sudo port install aquaterm
```

And when the installment is completed, prompt

```
> sudo port install gnuplot
```

From this point on install ACADO Toolkit by following the installment guide for Linux.

4.1.1.3 Installation under Windows

In order to install ACADO Toolkit on Windows, you need to install a Linux environment (emulator). This could for example be Cygwin (http://www.cygwin.com/). As for the above operating systems a C++ compiler, Gnuplot and a X-server is required to run ACADO Toolkit.

Using Cygwin select "gcc-g++" and "make" (Developer category), Gnuplot (Graphics category) and xinit (X11 category) as optional add-intallments during the installation procedure. Start Cygwin and run "c:/cygwin/bin/startwin.bat". This will provide an environment for plotting.

Create a directory, where you want to install ACADOtoolkit.

```
> cd c:
> mkdir src
> cd src
```

Copy ACADOtoolkit-1.0beta.tar.gz to the installation directory (here c:/src) and unpack it.

```
> tar xfvz ACADOtoolkit-1.0beta.tar.gz
```

Open the file include/include.mk in the ACADO directory and make sure that the system is set to WIN32 and that the GNU compiler is used:

- COMPILER = GNU
- SYSTEM = WIN32

Finally, go to the directory ACADOtoolkit-1.0beta and build the package

```
> cd ACADOtoolkit-1.0beta
> make
```

4.1.2 Running a program

A simple example of an OCP problem modeled in ACADO for optimization is the Optimal Time of Rocket model introduced in chapter 1 of this thesis. The model is a part of ACADO's example library and the implementation looks as follows

```
/**
 *    \file   examples/getting_started/simple_ocp.cpp
 *    \author Boris Houska, Hans Joachim Ferreau
 *    \date   2009
 */


#include <acado_toolkit.hpp>
#include <gnuplot/acado2gnuplot.hpp>


int main( ){

    USING_NAMESPACE_ACADO

    // the differential states
    DifferentialState        s,v,m      ;
    // the control input u
    Control                  u          ;
    // the time horizon T
    Parameter                T          ;
    // the differential equation
    DifferentialEquation     f( 0.0, T );

//  ---------------------------------------
    OCP ocp( 0.0, T );              // time horizon of the OCP: [0,T]
    ocp.minimizeMayerTerm( T );     // the time T should be optimized

    f << dot(s) == v;               // an implementation
    f << dot(v) == (u-0.2*v*v)/m;   // of the model equations
    f << dot(m) == -0.01*u*u;       // for the rocket.

    // minimize T s.t. the model, the initial values for s,
    // v and m,
    ocp.subjectTo( f                     );
    ocp.subjectTo( AT_START, s ==  0.0 );
    ocp.subjectTo( AT_START, v ==  0.0 );
    ocp.subjectTo( AT_START, m ==  1.0 );

    // the terminal constraints for s, and v,
    ocp.subjectTo( AT_END  , s == 10.0 );
    ocp.subjectTo( AT_END  , v ==  0.0 );

    // as well as the bounds on v, the control input u,
    // and the time horizon T.
    ocp.subjectTo( -0.1 <= v <=  1.7   );
```

```
73      ocp.subjectTo( -1.1 <= u <=  1.1    );
        ocp.subjectTo(  5.0 <= T <= 15.0    );
75  //  ------------------------------------
77      GnuplotWindow window;
            window.addSubplot( s, "THE DISTANCE s"       );
79          window.addSubplot( v, "THE VELOCITY v"       );
            window.addSubplot( m, "THE MASS m"           );
81          window.addSubplot( u, "THE CONTROL INPUT u" );
83      OptimizationAlgorithm algorithm(ocp); // the optimization
            algorithm
        algorithm << window;
85      algorithm.solve();                    // solves the problem.
87
        return 0;
89  }
```

The model can be run with the following commands

```
> cd ACADOtoolkit-1.0beta/examples/getting_started
> ./simple_ocp
```

When building your own program, start by creating a folder under the directory "ACADOtoolkit-1.0beta/examples/" (placement of the folder is of course optional). Copy the *Makefile* in the directory "examples" and place it in your own folder. Open the *Makefile* and delete all filenames under

```
54
SRCS =
56
DEV_SRCS =
```

Type the name of your own C++ code-file under SRCS. If you have chosen a different placement of the folder than suggested, then you should make sure that the include-paths in the top of the *Makefile* reflect the directory of your own newly created model library.

All the elements of a C++ library are declared within a namespace. In order to access its functionality, I declare using this expression that I will be using these entities. In our case the library namespace has the call "USING_NAMESPACE_ACADO". It is recommended to use an existing example file as template for your code. A standard code-file should have the form

```
#include <acado_optimal_control.hpp>
2  #include <gnuplot/acado2gnuplot.hpp>
```

```
4  int main( ){
6      USING_NAMESPACE_ACADO
8        // Constants, Variables, Optimization setup ...
10     return 0;
   }
```

You are now ready to build a Unix-Archive of your code-file. Go to your folder, build, and run your program

```
> cd ACADOtoolkit-1.0beta/examples/myfolder
> make
> ./myprogram
```

4.2 Simulating the Four Tank System using ACADO Toolkit

Referring back to the Four Tank System from chapter 3, I will start by summarizing the model which is to be implemented. Then I will meticulously go through the implementation of the model using ACADO toolkit, focusing on attention demanding challenges and their solutions. A range of ACADO functionalities and properties will be presented, when new objects and functions is required while implementing the model.

The complete model is given as (3.19a) - (3.20n) and the parameter vector p in the model is defined as

$$p = [a_1\, a_2\, a_3\, a_4\, A_1\, A_2\, A_3\, A_4\, \gamma_1\, \gamma_2\, g\, \rho\, r_1\, r_2\, s]^T$$

Using this notation, the system of differential equations determining the evolution of the system can then be represented as

$$\frac{dx(t)}{dt} = f(t, x(t), u(t), p) \quad x(t_0) = x_0 \quad t \in [t_0, t_{end}]$$

4.2.1 Implementing the model in ACADO Toolkit

In the following I will build up the ACADO script file for solving the Four Tank System seen as an optimal control problem. The following code is used as the content of the template file above. First, I define the parameters for the system.

```
USING_NAMESPACE_ACADO

// ---------------------------------------------------
// Parameters
// ---------------------------------------------------

// Cross sectional area of outlet [cm2]
const double a1 = 1.2272;
const double a2 = 1.2272;
const double a3 = 1.2272;
const double a4 = 1.2272;

// Cross sectional area of inlet [cm2]
const double A1 = 380.1327;
const double A2 = 380.1327;
const double A3 = 380.1327;
const double A4 = 380.1327;

// Height of tank [cm]
const double H1 = 20;
const double H2 = 20;
const double H3 = 20;
const double H4 = 20;

// Valve position
const double gamma1 = 0.15;
const double gamma2 = 0.25;

// Set point (water level goal)
const double r1 = 12;
const double r2 = 12;

// Acceleration of gravity [cm/s2]
const double g = 981;

// Density of water [g/cm3]
const double rho = 1.00;
```

Then the equations for the flow from the valves to each tank (3.10), the relations for the liquid heights (3.13), the relations for the outlet flow rates (3.12), the differential equations (3.4), and their initial conditions (3.5) will be implemented.

Since equations (3.10), (3.13) and (3.12) all are function values describing a system state with respect to a given time point, these variables are declared

as the data type "IntermediateState" (symbolic state variable) in ACADO. Likewise, (3.5) describe a system state with respect to a given time point of the differential equations (initial values). These are declared as the data type "DifferentialState" (symbolic state variable) in ACADO. u is declared as data type "Control" and the differential equations are declared as "DifferentialEquation".

When deriving the numerical solution to the system, the derivative of the outgoing flow rates of tanks will be calculated by the solver. The derivatives are

$$\frac{dq}{dh_i} = \frac{a_i g}{\sqrt{2gh_i}}, \quad i = 1, \ldots, 4$$

If we set the initial h_i's to zero, we will get an error. To avoid this, I therefore need an approximation of the h_i's to substitute these. For a sufficiently small constant s and $h \geq 0$, I can approximate h by the expression

$$s \cdot \log(\exp(\tfrac{h}{s}) + 1) > 0$$

The validity of the approximation and influence of s can be shown by plotting the absolute error of the approximation for different values of h and s (see figure 4.1).

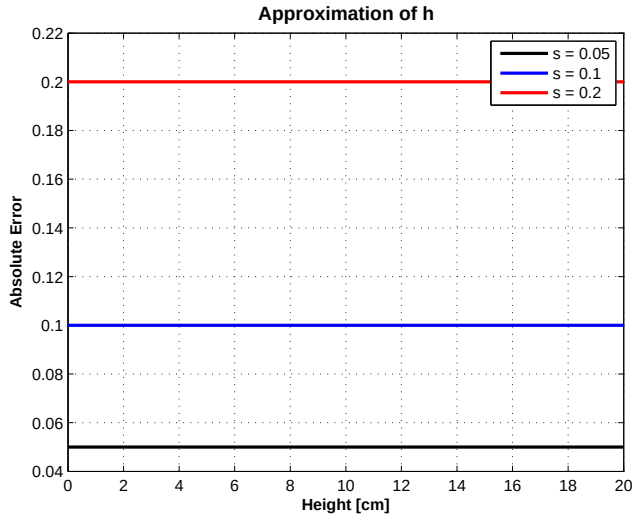


Figure 4.1: Absolute error using approximation of h

I therefore use the following approximation of h

$$\sqrt{h} \approxeq h_s = \frac{h}{\sqrt{s \cdot \log(\exp(h/s) + 1)}}$$

As for the differential equations (3.4) I need to write all of them into the single output f. The insertion operator "<<" is used for this in C++.

The implementation of these declarations and variable definitions looks as follow

```
53  // ---------------------------------------------------
    // Declare controls, equations, and states
    // ---------------------------------------------------
55
    // Mass in tanks [g]
57  DifferentialState      m1,m2,m3,m4;
    // Equations constraints
59  IntermediateState      q1i,q2i,q3i,q4i;
    IntermediateState      h1,h2,h3,h4;
61  IntermediateState      h1s,h2s,h3s,h4s;
    IntermediateState      q1,q2,q3,q4;
63
    // Controls (Flow rates in valves [cm3/s])
65  Control                F1,F2;

67  // System Equation
    DifferentialEquation   f;
69
    // ---------------------------------------------------
71  // Equation Definition
    // ---------------------------------------------------
73  // Flow rate in valves [cm3/s]
    q1i = gamma1*F1;
75  q2i = gamma2*F2;
    q3i = (1-gamma2)*F2;
77  q4i = (1-gamma1)*F1;

79  // Height of liquid [cm]
    h1  = m1/(rho*A1);
81  h2  = m2/(rho*A2);
    h3  = m3/(rho*A3);
83  h4  = m4/(rho*A4);

85  // Approximation variables
    const double s = 0.1;
87  h1s = h1/(sqrt(s*log(exp(h1/s)+1)));
    h2s = h2/(sqrt(s*log(exp(h2/s)+1)));
89  h3s = h3/(sqrt(s*log(exp(h3/s)+1)));
    h4s = h4/(sqrt(s*log(exp(h4/s)+1)));
91
    //  Flow rate in tanks [cm3/s]
93  q1  = a1*sqrt(2*g)*h1s;
    q2  = a2*sqrt(2*g)*h2s;
```

```
95          q3  = a3*sqrt(2*g)*h3s;
            q4  = a4*sqrt(2*g)*h4s;
97
            // System Equations (Mass balance equation 1-4)
99          f  << dot(m1) == rho*q1i + rho*q3 - rho*q1;
            f  << dot(m2) == rho*q2i + rho*q4 - rho*q2;
101         f  << dot(m3) == rho*q3i - rho*q3;
            f  << dot(m4) == rho*q4i - rho*q4;
```

The optimization model of the quadruple tank process can now be implemented. First, a time interval has to be defined. In my case the grid is chosen by the solvers error control step method using the default options with initial step size at 10^{-3}, minimal step size 10^{-8}, maximum step size 10^{+8}, a step size tuning at 0.5, and a corrector tolerance at 10^{-14}. These can all be altered by the user. I will come back to this below. So using the default grid options I just define a start and an end time point. These values are defined as constants and is the first input of the `ACADO OCP` solver (see code below). In my case the time interval of 5 minutes are chosen. The user can add more options to this call, such as "number of discretization intervals", or the user can simple load her own discretization intervals using the call

```
VariablesGrid measurements;
2 measurements = readFromFile( "data.txt" );

4 OCP ocp( measurements.getTimePoints() );
```

data.txt is a datafile, where the discretization values are the first column in the file.

The type of objective function is assigned to the `ocp` object, including information on whether it is a maximization or minimization problem. The problem can consist of a Lagrange term, a Mayer term and a Least Square term (see chapter 1 for more on this). The Least Square term has the optional input of a weighting matrix, the least square function, and a right hand side vector. The terms is assigned one at a time to the `ocp` object and has the calls

```
minimizeLagrangeTerm(z),
 minimizeMayerTerm(z),
    minimizeLSQ(z)
```

"`minimize`" can be replaced with "`maximize`" and z is the objective function.

The condition terms is then assigned to the ocp object using the call "`subjectTo(c)`", where c is the condition equation. Boundaries are set using "==", "<=" or ">=".

Further input to this call can also be added, such as initial condition information. The options can be

```
AT_START,
AT_END,
AT_TRANSITION
```

The implementation of my optimal control problem then looks as follows

```
105 // ------------------------------------------------------
    // DEFINE AN OPTIMAL CONTROL PROBLEM:
    // ------------------------------------------------------
107
    // Start time [sec]
109 const double t_start = 0.0;
    // End time [sec]
111 const double t_end   = 60*5;

113 OCP ocp( t_start, t_end);
    ocp.minimizeLagrangeTerm( (h1-r1)*(h1-r1) + (h2-r2)*(h2-r2)
        );
115
    // Satisfy model dynamics.
117 ocp.subjectTo( f );

119 // initialize masses.
    ocp.subjectTo( AT_START, m1 == 0 );
121 ocp.subjectTo( AT_START, m2 == 0 );
    ocp.subjectTo( AT_START, m3 == 0 );
123 ocp.subjectTo( AT_START, m4 == 0 );

125 // Only non-negative inputs.
    ocp.subjectTo( 0 <= F1 <= 300 );
127 ocp.subjectTo( 0 <= F2 <= 300 );

129 // No tank underflow or overflow.
    ocp.subjectTo( 0 <=  h1 <= H1 );
131 ocp.subjectTo( 0 <=  h2 <= H2 );
    ocp.subjectTo( 0 <=  h3 <= H3 );
133 ocp.subjectTo( 0 <=  h4 <= H4 );
```

For plotting the results of the simulation and optimal solution I then declare a `"GnuplotWindow"` object. The plots are then assigned to the plot object using the call

```
addSubplot(x,"Title")
```

x is one of the symbolic variables of the model or a system defined command option. "Title" is the title text of the plotting window. The system defined command inputs include plotting the various symbolic variables types, such as the inputs

PLOT_LINESEARCH_STEPLENGTH,
PLOT_KKT_TOLERANCE,
PLOT_OBJECTIVE_VALUE

The implementation of my plotting outputs then looks as follows

```
135         // ------------------------------------------------------
            // Plot
137         // ------------------------------------------------------

139         GnuplotWindow window;

141         window.addSubplot(h1,"Height Tank 1 [cm]");
            window.addSubplot(h2,"Height Tank 2 [cm]");
143         window.addSubplot(h3,"Height Tank 3 [cm]");
            window.addSubplot(h4,"Height Tank 4 [cm]");
145         window.addSubplot(F1,"Flow Rate 1 [cm3/s]");
            window.addSubplot(F2,"Flow Rate 2 [cm3/s]");
147         window.addSubplot(PLOT_LINESEARCH_STEPLENGTH,"Step length")
                ;
            window.addSubplot(PLOT_KKT_TOLERANCE,"KKT Tolerance");
```

Finally, I ask ACADO to solve the problem. I declare a "OptimizationAlgorithm" object. As with other ACADO objects I have the opportunity of adding user defined options to the solver. In the following I will just mention the options, I have chosen, namely, type of Hessian approximation (default is the block BFGS update) and maximum number of iterations (default is 200) (a user set KKT tolerance could also have been defined as "PLOT_LINESEARCH_STEPLENGTH" instead of the default value of 10^{-6}). The plotting object is written to the solver object as output (insertion) and the problem is finally solved.

```
151         // DEFINE AN OPTIMIZATION ALGORITHM AND
            // SOLVE THE OCP:
153         // ------------------------------------------------------

155         OptimizationAlgorithm algorithm(ocp);
            algorithm.set( MAX_NUM_ITERATIONS, 300 );
157         algorithm.set( HESSIAN_APPROXIMATION, EXACT_HESSIAN );

159         algorithm << window;
            algorithm.solve();
```

The complete implementations thereby looks as follows

```
/**
 *    \file tank_system_control.cpp
 *    \author Leo Emil Sokoler, Rune Brus
 *    \date 25-01-2010
 */
// Includes.

#include <acado_optimal_control.hpp>
#include <gnuplot/acado2gnuplot.hpp>

int main( ){

        // Acado namespace
        USING_NAMESPACE_ACADO

        // ---------------------------------------------
        // Parameters
        // ---------------------------------------------

        // Cross sectional area of outlet [cm2]
        const double a1 = 1.2272;
        const double a2 = 1.2272;
        const double a3 = 1.2272;
        const double a4 = 1.2272;

        // Cross sectional area of inlet [cm2]
        const double A1 = 380.1327;
        const double A2 = 380.1327;
        const double A3 = 380.1327;
        const double A4 = 380.1327;

        // Height of tank [cm]
        const double H1 = 20;
        const double H2 = 20;
        const double H3 = 20;
        const double H4 = 20;

        // Valve position
        const double gamma1 = 0.15;
        const double gamma2 = 0.25;

        // Set point (water level goal)
        const double r1 = 12;
        const double r2 = 12;

        // Acceleration of gravity [cm/s2]
        const double g = 981;

        // Density of water [g/cm3]
        const double rho = 1.00;

        // ---------------------------------------------
        // Declare controls, equations, and states
```

```
54          // ------------------------------------------------------
56          // Mass in tanks [g]
            DifferentialState     m1,m2,m3,m4;
58          // Equations constraints
            IntermediateState     q1i,q2i,q3i,q4i;
60          IntermediateState     h1,h2,h3,h4;
            IntermediateState     h1s,h2s,h3s,h4s;
62          IntermediateState     q1,q2,q3,q4;

64          // Controls (Flow rates in valves [cm3/s])
            Control               F1,F2;
66
            // System Equation
68          DifferentialEquation  f;

70          // ------------------------------------------------------
            // Equation Definition
72          // ------------------------------------------------------
            // Flow rate in valves [cm3/s]
74          q1i = gamma1*F1;
            q2i = gamma2*F2;
76          q3i = (1-gamma2)*F2;
            q4i = (1-gamma1)*F1;
78
            // Height of liquid [cm]
80          h1  = m1/(rho*A1);
            h2  = m2/(rho*A2);
82          h3  = m3/(rho*A3);
            h4  = m4/(rho*A4);
84
            // Approximation variables
86          const double s = 0.1;
            h1s = h1/(sqrt(s*log(exp(h1/s)+1)));
88          h2s = h2/(sqrt(s*log(exp(h2/s)+1)));
            h3s = h3/(sqrt(s*log(exp(h3/s)+1)));
90          h4s = h4/(sqrt(s*log(exp(h4/s)+1)));

92          //  Flow rate in tanks [cm3/s]
            q1  = a1*sqrt(2*g)*h1s;
94          q2  = a2*sqrt(2*g)*h2s;
            q3  = a3*sqrt(2*g)*h3s;
96          q4  = a4*sqrt(2*g)*h4s;

98          // System Equations (Mass balance equation 1-4)
            f  << dot(m1) == rho*q1i + rho*q3 - rho*q1;
100         f  << dot(m2) == rho*q2i + rho*q4 - rho*q2;
            f  << dot(m3) == rho*q3i - rho*q3;
102         f  << dot(m4) == rho*q4i - rho*q4;

104         // ------------------------------------------------------
            // DEFINE AN OPTIMAL CONTROL PROBLEM:
106         // ------------------------------------------------------

108         // Start time [sec]
```

```
    const double t_start = 0.0;
    // End time [sec]
    const double t_end   = 60*5;

    OCP ocp( t_start, t_end);
    ocp.minimizeLagrangeTerm( (h1-r1)*(h1-r1) + (h2-r2)*(h2-r2) );

    // Satisfy model dynamics.
    ocp.subjectTo( f );

    // initialize masses.
    ocp.subjectTo( AT_START, m1 == 0 );
    ocp.subjectTo( AT_START, m2 == 0 );
    ocp.subjectTo( AT_START, m3 == 0 );
    ocp.subjectTo( AT_START, m4 == 0 );

    // Only non-negative inputs.
    ocp.subjectTo( 0 <= F1 <= 300 );
    ocp.subjectTo( 0 <= F2 <= 300 );

    // No tank underflow or overflow.
    ocp.subjectTo( 0 <=  h1 <= H1 );
    ocp.subjectTo( 0 <=  h2 <= H2 );
    ocp.subjectTo( 0 <=  h3 <= H3 );
    ocp.subjectTo( 0 <=  h4 <= H4 );

    // -----------------------------------------
    // Plot
    // -----------------------------------------

    GnuplotWindow window;

    window.addSubplot(h1,"Height Tank 1 [cm]");
    window.addSubplot(h2,"Height Tank 2 [cm]");
    window.addSubplot(h3,"Height Tank 3 [cm]");
    window.addSubplot(h4,"Height Tank 4 [cm]");
    window.addSubplot(F1,"Flow Rate 1 [cm3/s]");
    window.addSubplot(F2,"Flow Rate 2 [cm3/s]");
    window.addSubplot(PLOT_LINESEARCH_STEPLENGTH,"Step length");
    window.addSubplot(PLOT_KKT_TOLERANCE,"KKT Tolerance");

    // -----------------------------------------
    // DEFINE AN OPTIMIZATION ALGORITHM AND
    // SOLVE THE OCP:
    // -----------------------------------------

    OptimizationAlgorithm algorithm(ocp);
    algorithm.set( MAX_NUM_ITERATIONS, 300 );
    algorithm.set( HESSIAN_APPROXIMATION, EXACT_HESSIAN );

    algorithm << window;
    algorithm.solve();
```

```
162         return 0;

164 }
```

I then compile the code file (make) and run the unix-archive of the code. As a way of comparing the various Hessian approximations, I here display Table 4.1, stating the terminal outputs of a number of these.

Hessian Approx.	# Iterations	KKT tolerance	Objective value
Exact Hessian	11	2.355e-07	5.3518e+03
Gauss Newton with .. Block BFGS update	27	8.487e-07	5.3518e+03
Gauss Newton	27	8.487e-07	5.3518e+03
Block BFGS update	27	8.487e-07	5.3518e+03
Full BFGS update	78	6.868e-07	5.3518e+03
Constant (1)	300	9.860e-06	5.3518e+03
Unknown	300	2.996e-01	2.1989e+04

Table 4.1: Iterations Statistics of ACADO Toolkit

As it appears from the table. The algorithm was not able to converge using a constant - or no Hessian (unknown). The KKT tolerance becomes asymptotic around 10^{-6} using a constant Hessian.

Using the exact hessian I get the outputs Figure 4.2, 4.3 and 4.4.

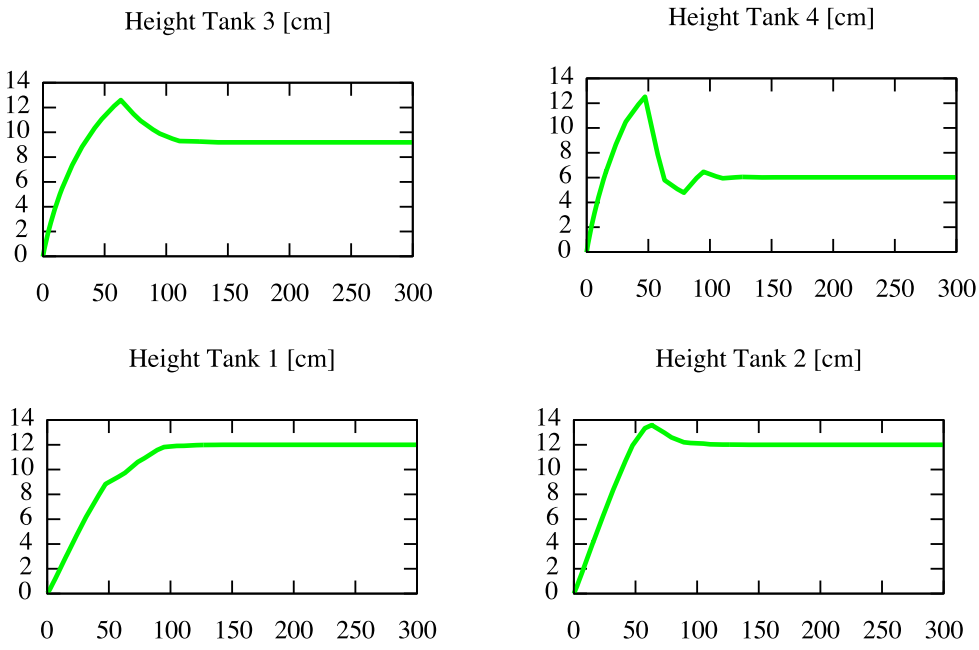


Figure 4.2: Height of liquid in tanks

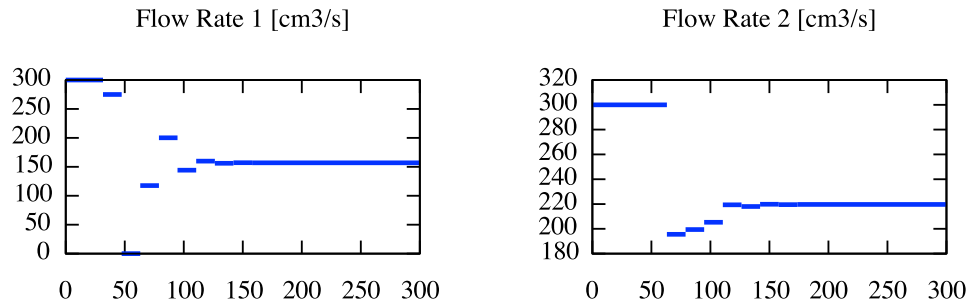


Figure 4.3: Control varibles. Flow rate in pumps

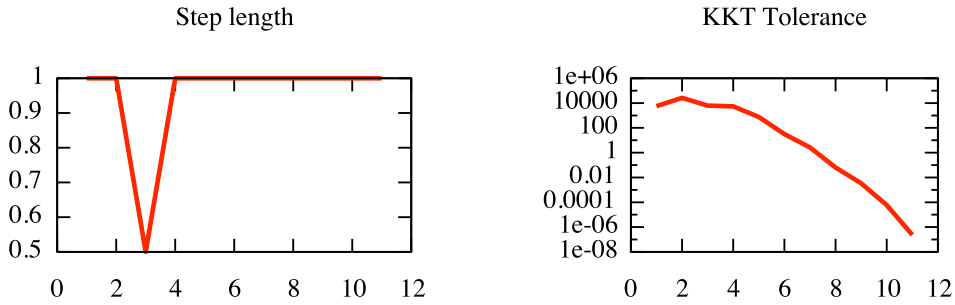


Figure 4.4: Iteration information. Step sizes and KKT values

It is seen that the difference in valve flow rate effects the optimal flow rates from the pumps and the required water level in tank 3 and 4.

A great influence on the convergence of the model is the choice of Hessian approximation.

Chapter 5

IPOPT

In this chapter, I introduce the optimization software IPOPT (**I**nterior **P**oint **Opt**imizer (pronounced "I-P-Opt"). A short introduction of the origin and main features will be provided. Afterwards, I will go through the installation of IPOPT, providing installation procedures for both Linux, Mac OS X and Windows. Then I will give a guide on how to interface IPOPT, spanning from standard programming languages, such as C++ and Fortran, to actual algebraic modeling languages (AML's) . Then I will go through all elements on how to simulate and solve the Quadruple Tank Process model using IPOPT together with some of the interfaces introduces in the previous section. Finally, I will present ways of visualizing the results.

References: Sequential Quadratic Programming [4], Numerical Computation [5], Adaptive Barrier Strategies [11], Combinatorial Approaches [12], Interior-Point Filter Line-Search Algorithm [13], Line Search Filter Methods [14], Interior Point Algorithm [15], Damped Newton Methods [16], Interior point methods [17]

URLs: [IPOPT Home] http://www.coin-or.org/projects/Ipopt.xml,

[IPOPT Documentation] http://www.coin-or.org/Ipopt/documentation/

5.1 IPOPT - Getting Started

IPOPT is an Open-Source package for large-scale nonlinear optimization, provided under a Common Public License. Unlike ACADO toolkit, IPOPT only provides the solver and not an entire framework with interface. IPOPT is written in C++ and provides a large range of user setting options. IPOPT is developed, such that it supports 5 different sparse linear solvers (this will be elaborated below). It can be used for general nonlinear optimization of the form

$$\min_{x \in \mathbb{R}^n} f(x) \tag{5.1}$$

$$\text{s.t.} \quad g^L \leq g(x) \leq g^U \tag{5.2}$$

$$x^L \leq x \leq x^U \tag{5.3}$$

$x \in \mathbb{R}^n$ are the optimization variables, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, and $g : \mathbb{R}^n \to \mathbb{R}^m$ are the general nonlinear constraints. The functions $f(x)$ and $g(x)$ have to be twice differentiable, but can be linear or nonlinear and convex or non-convex.

5.1.1 Installation

The IPOPT code package is available from IPOPT Home under a Common Public License, meaning that it is available free of charge both for private and commercial purposes.

The installation of IPOPT requires some third party components. The first requirement is some algorithm libraries. These libraries are BLAS (Basic Linear Algebra Subroutines), these are used for dense linear algebra, and LAPACK (Linear Algebra PACKage), with is used for the quasi-Newton solver options in IPOPT. The second requirement is a sparse symmetric indefinite linear solver. The following solvers is supported and are available for this purpose.

- MA27 - http://www.cse.clrc.ac.uk/nag/hsl/ (the HSL package)
- MA57 - http://www.cse.clrc.ac.uk/nag/hsl/ (the HSL package)
- MUMPS (MUltifrontal Massively Parallel sparse direct Solver) - http://graal.ens-lyon.fr/MUMPS/ (the MUMPS package)
- PARDISO (The parallel Sparse Direct Solver) - http://www.computational.unibas.ch/cs/scicomp/software/pardiso/

- WSMP (The Watson Sparse Matrix Package) - http://www-users.cs.umn.edu/$\sim$agupta/wsmp.html

The best choice of linear solver depends entirely on ones application, but IPOPT can be compiled with all options.

If the MUMPS or MA57 solver is used, then it is also recommended to include the METIS (Linear System Ordering package) package in the installation of IPOPT. This is a recommendation, but not a requirement, meaning that IPOPT works fine without METIS using MUMPS or MA57 as solver. METIS is able to reduce the storage and computational requirements of sparse matrix factorization and is recommended for large-scale problems in order to reduce computation time (see http://glaros.dtc.umn.edu/gkhome/metis/metis/overview to learn more).

Finally, the ASL (AMPL Solver Library) package should be included in the installation, if the user wishes to use the AMPL interface with IPOPT as solver (more about this below).

The IPOPT download package include unix-installation scripts for most of the above requirements, so the user should just download, unpack IPOPT, and run these, before compiling the IPOPT solver package (guidance provided below).

Before demonstrating the use of the solver, an installation guide will be provided for Linux, Mac OS X and Windows.

5.1.1.1 Installation under Linux

In order to install and run IPOPT you need to make sure that a C++ and Fortran compiler are installed on the system. This can be checked by prompting the unix commands

```
> g++ -v
```

The compiler contains a version of the gfortran compiler and can be enabled to run fortran f77. The IPOPT installation script may not be able to recognize the fortran compiler. In this case gfortran can be installed separately. If one of these compilers is missing or unrecognized, they can be installed using APT (see 4.1.1.1 for information). Install gfortran by prompting

```
> sudo apt-get install gfortran
```

IPOPT does not provide a direct plotting function package, but outputs can be produced to be plotted using Gnuplot (installation see 4.1.1.1) or ones own favorite plotting tool.

The IPOPT program package can now be downloaded and unpacked from

http://www.coin-or.org/download/source/Ipopt/ - Latest stable version is 3.8.0.

Go to your download folder and extract the files

```
> gunzip IPOPT-3.8.1.tgz
> tar xvf IPOPT-3.8.1.tar
```

Move the directory to a preferred system directory.

IPOPT also provide a subversion repository for downloading the file package. Using this option, start by checking, if your system support subversion. If this is not the case, subversion support can be installed using APT (see 4.1.1.1). Go to a preferred system directory and prompt

```
> sudo apt-get install subversion
> svn co https://projects.coin-or.org/svn/Ipopt/stable/3.8.1 CoinIpopt
```

Go to the newly created IPOPT directory and install the solver dependencies

```
> cd Thirdparty/Blas
> ./get.Blas
> cd ../Lapack
> ./get.Lapack
> cd ../Mumps
> ./get.Mumps
> cd ../ASL
> ./get.ASL
```

For installing the HSL, PARDISO or WSMP package see the IPOPT Documentation.

The IPOPT solver can now be installed. Create a build folder for compiling

```
> mkdir build
```

```
> cd build
```

Run the configure script

```
> ../configure
```

Build the code

```
> make
```

Check if all executables work

```
> make test
```

Install IPOPT

```
> make install
```

5.1.1.2 Installation under Mac OS X

The dependencies is the same as on Linux and the same unix command can to some extend be reused using a Mac Terminal. The user should though be warned that installing IPOPT can have problem issues with mac OS X (see IPOPT Current Issues).

In my own case I installed it on mac OS X 10.6 (Snow Leopard). This OS is a 64-bit operating system. As C and C++ compiler I used GCC which default produces 64-bit code strings. As Fortran compiler I used gfortran which default produces 32-bit code strings. This caused a fatal building error for my sparse linear solvers (Mumps and MA27). The problem was solved by forcing gfortran to produce 64-bit code stings during the code configuration, i.e. I used the configuration command

```
> ../configure ADD_FFLAGS="-arch x86_64"
```

See 4.1.1.1 for more tips on mac OS X.

5.1.1.3 Installation under Windows

In order to install IPOPT on Windows, you need to install a Linux environment (emulator). This could for example be Cygwin. Although it is also possible to use IPOPT through MS Visual Studio. See the instructions from 4.1.1.1, but be aware that IPOPT need some further Cygwin add-ons. See the IPOPT Documentation for an extensive guide to the unix installation and installation under MS Visual Studio.

5.1.2 Interfacing IPOPT

IPOPT supports a range of optimization interfaces, but I will only go into details with the C++ interface.

A simple test model is provided in the IPOPT installation

$$\min_x \quad x_0 \cdot x_3 \cdot (x_0 + x_1 + x_2) + x_2$$

$$\begin{aligned} \text{s.t.} \quad & x_0 \cdot x_1 \cdot x_2 \cdot x_3 \geq 25 \\ & x_0^2 + x_1^2 + x_2^2 + x_3^2 = 40 \\ & x_0, x_1, x_2, x_3 \geq 1 \\ & x_0, x_1, x_2, x_3 \leq 5 \end{aligned}$$

This test model (using C++ interface) can be run with the following commands

```
> cd build/Ipopt/test/
> ./hs071_cpp
```

5.1.2.1 General-Purpose Programming Languages

With these interfaces the model code is all constructed the same way, containing the same methods. The user will need to hardcode the complete model and all the mathematical objects needed to solved a nonlinear optimization problem. Examples of how to construct models in these interfaces can be seen in the installation directory "build/Ipopt/tutorial/CodingExercise". IPOPT supports models in

- C
- C++
- Fortran
- Python

The Python interface is not included in the default installation package of IPOPT, but can be downloaded at http://code.google.com/p/pyIPOPT/. A Java support package is also under developments, but is not yet available (see IPOPT Home).

5.1.2.2 The C++ interface

Start by creating a new folder in the directory "build/MyIPOPT/", where your C++ code program can be stored. Copy the *Makefile* in the folder "build/Ipopt/tutorial/CodingExercise/Cpp/3-solution" and place it in your own folder. Open the *Makefile* and delete all filenames under

```
12 # This should be the name of your executable
   EXE =
14
   # Here is the name of all object files corresponding to the source
16 # code that you wrote in order to define the problem statement
   OBJS =
```

Type the preferred name of your C++ code execution script under `EXE`. The execution script has no file extension. Then type the name of your C++ object files under `OBJS`. These names should match the source file names of your program code and be given the extension .o.

It is of course possible to compile and create C++ code executables without using such a make script. In our case though, we wish to use the IPOPT namespace and the executable need to know, what this is and where it can be found. Thus, we need to use a carefully written make script.

Before we can compile and create an executable file, the C++ program need to be written. First of all, a C++ header-file need to be created. This file specifies what member functions (methods) and data members (fields) the class will have. This header file can be used almost unchanged in all your C++ IPOPT models. Only the class name has to be changed to fit the name used in the two source code files. The methods in the header file will be explained in detail, when I

implement the Four tank system. The first source file then contains the function bodies of the methods from the header file and the second source file (the main source file) contains the remaining fields and methods for creating the problem and finally solving it.

A header file looks as follows

```
//
// Author:  Rune Brus      17-02-2010
//

#ifndef __My_NLP_HPP__

#define __My_NLP_HPP__

#include "IpTNLP.hpp"

using namespace Ipopt;

class My_NLP : public TNLP
{
public:
  // constructor
  My_NLP(Index N, const Number* c);

  // destructor
  virtual ~My_NLP();

  virtual bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                            Index& nnz_h_lag,
                            IndexStyleEnum& index_style);

  virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                               Index m, Number* g_l, Number* g_u);

  virtual bool get_starting_point(Index n, bool init_x, Number* x,
                                  bool init_z, Number* z_L,
                                  Number* z_U, Index m,
                                  bool init_lambda,
                                  Number* lambda);

  virtual bool eval_f(Index n, const Number* x, bool new_x,
                      Number& obj_value);

  virtual bool eval_grad_f(Index n, const Number* x, bool new_x,
                           Number* grad_f);

  virtual bool eval_g(Index n, const Number* x, bool new_x,
                      Index m, Number* g);

  virtual bool eval_jac_g(Index n, const Number* x, bool new_x,
                          Index m, Index nele_jac, Index* iRow,
                          Index *jCol, Number* values);
```

```
48   virtual bool eval_h(Index n, const Number* x, bool new_x,
                         Number obj_factor, Index m,
50                       const Number* lambda,
                         bool new_lambda, Index nele_hess,
52                       Index* iRow, Index* jCol,
                         Number* values);
54
   virtual void finalize_solution(SolverReturn status,
56                                 Index n, const Number* x,
                                   const Number* z_L,
58                                 const Number* z_U,
                                   Index m, const Number* g,
60                                 const Number* lambda,
                                   Number obj_value,
62                                 const IpoptData* ip_data,
                                   IpoptCalculatedQuantities* ip_cq);
64
private:
66   // Methods to block default compiler methods.
   // The compiler automatically generates
68   // the following three methods.
   My_NLP();
70   My_NLP(const My_NLPP&);
   My_NLP& operator=(const My_NLP&);
72
   Index N_;
74   Number* c_;

76 };

78
#endif
```

5.1.2.3 Mathematical Modeling Languages

With mathematical modeling interfaces the model can be implemented in a simpler and more intuitive way. A common denominator for this group of interfaces is though that they are traditionally licensed, meaning that they are Non-Free and can be very expensive. Because of this, mathematical modeling interfaces have little interest of this thesis. Examples of how to construct models in these interfaces can be seen in the installation directory

`"build/Ipopt/tutorial/CodingExercise"`

IPOPT is supported by the following mathematical modeling interfaces

- Matlab

- AMPL
- GAMS

As a way of getting started with IPOPT it is although easier to start out writing models in a mathematical modeling interfaces. I will therefore give a short introduction of the use of AMPL and GAMS. AMPL and GAMS receive my attention, because they both provide a free demo license.

5.1.2.4 The AMPL interface

The AMPL interface is a very intuitive way to access the IPOPT solver. A free student/demo licensed version of AMPL can be downloaded from the AMPL Download page. This license is limited to 300 variables. An unrestricted trail license or full license can by attained by contacting the AMPL company (see AMPL Vendors).

In order for AMPL to recognize the IPOPT solver, a system environmental variable has to be created pointing to the solver. Alternatively, a copy (or a soft link) of the IPOPT solver can be moved to an existing system environmental variable directory. To see the the existing system environmental variable directories prompt

```
> echo $PATH
```

A program formulated in the AMPL language only has 2 types of objects. "`param`" meaning parameters and "`var`" meaning variables. The user need to specify that the IPOPT solver should be used by the code line "`option solver ipopt`" in the top of the program. A "`var`" can not be defined using another "`var`" object, "`var`" is only a declaration, where the name, dimension and restrictions of the variables can be defined. The model is then formulated using first a command such as "`minimize z:`" or "`subject to x {i in 1..N}:`" followed by an equation. The program is finalized by the command line "`solve`".

As a way of exemplifying the implementation of an Optimal Control Problem in AMPL, I here present a complete implementation of the Four Tank System. I present this without going into further details.

```
1 # Author:  Rune Brus
  # Date: 05-02-2010
3
  # This AMPL script includes a model, the "solve" command, and a
      call
```

```
# to gnuplot.  You can "run" all this by just typing
#
# $ ampl Four_tank_system.run
#
# in your shell.  Or, in an ongoing AMPL session, you can do
#
# ampl: reset;
# ampl: include Four_tank_system.run;

option solver ipopt;

# ------------------------------------------------------
# ---------------------- PARAMETERS --------------------
# ------------------------------------------------------

# Number of discretization intervals
param N := 300;

# Cross sectional area of outlet [cm2]
param a1 := 1.2272;
param a2 := 1.2272;
param a3 := 1.2272;
param a4 := 1.2272;

# Cross sectional area of inlet [cm2]
param A1 := 380.1327;
param A2 := 380.1327;
param A3 := 380.1327;
param A4 := 380.1327;

# Height of tank [cm]
param H1 := 20;
param H2 := 20;
param H3 := 20;
param H4 := 20;

# Valve position
param gamma1 := 0.15;
param gamma2 := 0.25;

# Acceleration of gravity [cm/s2]
param g := 981;

# Density of water [g/cm3]
param rho := 1.00;

# Approximation constant (for heights)
param s := 0.1;

# Optimization goal (water level)
param r1 := 12;
param r2 := 12;

# Initial presure value for pumps
param F1_init := 300;
```

```
param F2_init := 300;

# time horizont
param tf := 300;

# ------------------------------------------------------
# ---------------------- Variables ---------------------
# ------------------------------------------------------


# Size of discretization intervals
param h = tf/N;

# Control Variables
var F1{i in 0..N} >= 0, <= 300, := F1_init;
var F2{i in 0..N} >= 0, <= 300, := F2_init;

# Masses
var m1{i in 0..N} >= 0;
var m2{i in 0..N} >= 0;
var m3{i in 0..N} >= 0;
var m4{i in 0..N} >= 0;

# Height of liquid [cm]
var h1{i in 0..N} >= 0, <= H1;
var h2{i in 0..N} >= 0, <= H2;
var h3{i in 0..N} >= 0, <= H3;
var h4{i in 0..N} >= 0, <= H4;

# Flow rate in valves [cm3/s]
var q1i{i in 0..N};
var q2i{i in 0..N};
var q3i{i in 0..N};
var q4i{i in 0..N};

# Flow rate in tanks [cm3/s].
var q1{i in 0..N};
var q2{i in 0..N};
var q3{i in 0..N};
var q4{i in 0..N};

# ------------------------------------------------------
# ---------------- OPTIMAL CONTROL PROBLEM -------------
# ------------------------------------------------------

# Objective function
minimize Water_level: h*0.5*((h1[0]-r1)*(h1[0]-r1) + (h2[0]-r2)*(h2
    [0]-r2)) + h*sum{i in 1..(N-1)} ((h1[i]-r1)*(h1[i]-r1) + (h2[i
    ]-r2)*(h2[i]-r2)) + h*0.5*((h1[N]-r1)*(h1[N]-r1) + (h2[N]-r2)*(
    h2[N]-r2));


# Differential equations. Mass balances
subject to dm1 {i in 1..N}: (m1[i]-m1[i-1])/h = rho*q1i[i] + rho*q3
    [i] - rho*q1[i];
```

```
111
subject to dm2 {i in 1..N}: (m2[i]-m2[i-1])/h = rho*q2i[i] + rho*q4
    [i] - rho*q2[i];
113
subject to dm3 {i in 1..N}: (m3[i]-m3[i-1])/h = rho*q3i[i] - rho*q3
    [i];
115
subject to dm4 {i in 1..N}: (m4[i]-m4[i-1])/h = rho*q4i[i] - rho*q4
    [i];
117
# Boundary conditions for masses
119 subject to m10: m1[0] = 0.0;

121 subject to m20: m2[0] = 0.0;

123 subject to m30: m3[0] = 0.0;

125 subject to m40: m4[0] = 0.0;

127 # Equation conditions system
subject to tank1 {i in 0..N}: h1[i] = m1[i]/(rho*A1);
129
subject to tank2 {i in 0..N}: h2[i] = m2[i]/(rho*A2);
131
subject to tank3 {i in 0..N}: h3[i] = m3[i]/(rho*A3);
133
subject to tank4 {i in 0..N}: h4[i] = m4[i]/(rho*A4);
135
subject to inflow1 {i in 0..N}: q1i[i] = gamma1*F1[i];
137
subject to inflow2 {i in 0..N}: q2i[i] = gamma2*F2[i];
139
subject to inflow3 {i in 0..N}: q3i[i] = (1-gamma2)*F2[i];
141
subject to inflow4 {i in 0..N}: q4i[i] = (1-gamma1)*F1[i];
143
subject to outflow1 {i in 0..N}: q1[i] = a1*sqrt(2*g*h1[i]);
145
subject to outflow2 {i in 0..N}: q2[i] = a2*sqrt(2*g*h2[i]);
147
subject to outflow3 {i in 0..N}: q3[i] = a3*sqrt(2*g*h3[i]);
149
subject to outflow4 {i in 0..N}: q4[i] = a4*sqrt(2*g*h4[i]);
151
# Solve the optimization problem
153 solve;

155 # write the data into a file for gnuplot
for {i in 0..N}
157   printf : "%16.4e %16.4e %16.4e %16.4e %16.4e %16.4e %16.4e \n", i
        *h, h1[i], h2[i], h3[i], h4[i], F1[i], F2[i] > plotfts.dat;

159 shell "gnuplot fts.gp";
```

5.1.2.5 The GAMS interface

Like AMPL, GAMS is a somewhat intuitive way to access the IPOPT solver. A free student/demo licensed version of GAMS can be downloaded from the GAMS Download page. This license is limited to 300 variables, an academic- or commercial license can be attained by contacting the GAMS Development Corporation (see GAMS Vendors).

The IPOPT solver is not part of the standard installation of GAMS. The additional library GAMSlinks has to installed afterwords. This library can be downloaded and installed by following the guide at `https://projects.coin-or.org/GAMSlinks`.

The user is suggested to look at the extensive model library at GAMS Model Library in order to learn how to use the GAMS programming languages.

A complete implementation of the Four Tank System can be seen in the appendix (A.1).

5.2 Simulating the Four Tank System using IPOPT

Referring back to the Four Tank System from chapter 3, I will start by summarizing the model which is to be implemented. Then I will meticulously go through the implementation of the model using the C++ interface and IPOPT, focusing on attention demanding challenges and their solutions. A range of IPOPT functionalities and properties will be presented, when new objects and functions is required while implementing the model.

The complete model is given as (3.19a) - (3.20n) and the parameter vector, p, in the model is defined as

$$p = [a_1\, a_2\, a_3\, a_4\, A_1\, A_2\, A_3\, A_4\, \gamma_1\, \gamma_2\, g\, \rho\, r_1\, r_2\, s]^T$$

IPOPT does not offer a way of defining differential equations or integration expressions exact. The system therefore has to be discretized and these types of expression has to be approximated.

There are many ways of defining differential equations as difference equations. I will not go into detail about what to choose in respect to the Four tank System. My method of choice will be the forward Euler method with fixed time step. Let the system state be given as (3.1a). The system of differential equations

determining the evolution of the system can then be represented as

$$0 = (x(t_i) - x(t_{i-1})) - h \cdot f(t_i, x(t_i), u(t_i), p) \quad x(t_1) = x_1 \quad t \in [t_1, t_{end}]$$

As for differential equations, there are many ways of approximating integration expressions. My method of choice will be Eulers' method (the trapezoidal rule). Let the objective be given as (3.19a). The objective $S(h)$ for the system can then be represented as

$$\begin{aligned} S(h) = h \cdot (&\tfrac{1}{2} \cdot (h_1(t_0) - r_1)^2 + (h_2(t_0) - r_2)^2) \\ &+ \sum_{i=t_1,\ldots,t_{f-1}} ((h_1(i) - r_1)^2 + (h_2(i) - r_2)^2) \\ &+ \tfrac{1}{2} \cdot (h_1(t_f) - r_1)^2 + (h_2(t_f) - r_2)^2)) \end{aligned}$$

5.2.1 Implementing the model in C++

In the following I will build up the two C++ source files for solving the Four Tank System seen as an optimal control problem. The following code is used as the content of the main code file. First, I present the main structure of the program, where I will fill code into. I need the IPOPT solver, my header file and a package for output printing.

```
1  #include "IpIPOPTApplication.hpp"
   #include "FourTankSystem.hpp"
3  #include <stdio.h>

5  using namespace ipopt;

7  int main(int argv, char* argc[]) {

9  }
```

Secondly, I declare and define the parameters for the system.

```
12 // ---------- Declarations ----------

14   // Problem size
     Index N = 300;
16
     // parameters
18   Number* c = new double[21];
     // Time horizont
20   c[0] = (double)(300);
     // Cross sectional area of outlet [cm2]
22   c[1] = (double)(1.2272);
```

```
    c[2] = (double)(1.2272);
    c[3] = (double)(1.2272);
    c[4] = (double)(1.2272);
    // Cross sectional area of inlet [cm2]
    c[5] = (double)(380.1327);
    c[6] = (double)(380.1327);
    c[7] = (double)(380.1327);
    c[8] = (double)(380.1327);
    // Height of tank [cm]
    c[9] = (double)(20);
    c[10] = (double)(20);
    c[11] = (double)(20);
    c[12] = (double)(20);
    // Valve presure
    c[13] = (double)(0.15);
    c[14] = (double)(0.25);
    // Acceleration of gravity [cm/s2]
    c[15] = (double)(981);
    // Density of water [g/cm3]
    c[16] = (double)(1.00);
    // Water stability level
    c[17] = (double)(12);
    c[18] = (double)(12);
    // Step size
    c[19] = (double)(c[0])/(double)(N);
    // Approximation constant
    c[20] = (double)(0.1);
```

Now the input model has to be created and various options for the solver has to be set. The constructor from the header file receives the parameter input, and together with the IPOPT solver method, is declared and created. The `SmartPtr` class is an IPOPT specific template class that takes care of deleting objects for us. Consequently we need not to be concerned about memory. This replaces the task of pointing to an object with a raw C++ pointer (e.g. FourTankSystem `NLP*`). It is a requirement when using IPOPT to only use this type of class.

The C++ interface for IPOPT offers many types of solver settings (see section 5 and appendix C of the IPOPT Documentation). In my case I wish to set the convergence tolerance and iteration method. As part of defining the function bodies of the second source file, the gradient of the objective, the Jacobian matrix for the constraints and the Karush-Kuhn-Tucker matrix for the whole system model (second-derivatives) need to be supplied. This can be a vast assignment and some feedback on the correctness of this implementation would be desired. For this purpose IPOPT provide the option `"derivative_test"`. This option compares numerical values of the first derivatives (`"first-order"`), or both the first derivative and the second derivatives (`"second-order"`), with the ones defined in the source file. See section 4.1 of the IPOPT Documentation to learn how to read the outputs of this option.

Last, the solver object is initialized and the options is processed. The final status of the solver algorithm is then outputted.

```
51 // ------------------ Create the Problem ------------------
   SmartPtr<TNLP> mynlp = new FourTankSystem_OCP(N, c);
53
   SmartPtr<IpoptApplication> app = new IpoptApplication();
55
   // Set some options
57 app->Options()->SetNumericValue("tol", 1e-7);
   app->Options()->SetStringValue("mu_strategy", "adaptive");
59 // app->Options()->SetStringValue("derivative_test", "second-
       order");
   app->Options()->SetStringValue("output_file", "4tank.out");
61
   // Intialize the IpoptApplication and process the options
63 app->Initialize();

65 // ------------------ Solve the Problem ------------------
   ApplicationReturnStatus status = app->OptimizeTNLP(mynlp);
```

This "status" output is then used to determine whether or not the algorithm converged to a feasible solution or not.

```
   // ------------------ Output solution ------------------
69 if (status == Solve_Succeeded) {
     printf("\n\n*** The problem solved!\n");
71 }
   else {
73   printf("\n\n*** The problem FAILED!\n");
   }
75
   delete [] c;
77
   return (int) status;
```

The C++ script containing the actual content of the header methods, now needs to be created.

I call the header file, a package for output printing and a package for using mathematical functions. I use the IPOPT namespace.

```
   #include "FourTankSystem.hpp"
6  #include <stdio.h>
   #include <math.h>
8
   using namespace Ipopt;
```

The constructor then receives my system size (N), that is the number of time steps in my model, i.e. the fineness of my descretization. The space for the

parameters is allocated and parameter names are given. The destructor need to release the allocated space of the parameters, when the algorithm has finalized.

```
// Constructor
FourTankSystem_OCP::FourTankSystem_OCP(Index N, const Number* c)
  :
  N_(N)
{
  // Copy the values for the constants appearing in the constraints
  c_ = new Number[21];
  for (Index i=0; i<21; i++) {
    c_[i] = c[i];
  }
}

// Destructor
FourTankSystem_OCP::~FourTankSystem_OCP()
{
  // make sure we delete everything we allocated
  delete [] c_;
}
```

The general model information is provided. The total number of model variables is 22. Further, each model variable consist of N subvariables representing each time step. I have 5 times 4 constraints (differential equations, heights, inflows, outflows, height approximations). Again, I then have N of each of these. I only have $N-1$ of each of the differential equations. In order to to simplify indexation later on, when I construct the model derivates, I compensate this index asymmetry by introducing 4 zero constraints (dummy constraints). This will then give me N of each of the differential equations.

The numbers of elements in the Jacobian is hard to comprehend, before it has actually been implemented. Easing this task, I only need to allocate and provide the non-zero elements of the Jacobian. I supply the input by first allocating the specific location of derived value. Then by assigning the derived value to the beforehand allocated matrix location. My chosen sparse linear equation solver then takes care of interpreting this input as an actual matrix. I have 5 different variables in each of the two first differential equations and 4 in next two and 2 in the last 16 constraints. Most of my constraints are linear, giving me only 4 times N non-zero elements in the KKT matrix.

Last, I declare that indexing starting in zero should be used.

```
bool FourTankSystem_OCP::get_nlp_info(Index& n, Index& m,
                                      Index& nnz_jac_g,
                                      Index& nnz_h_lag,
                                      IndexStyleEnum& index_style)
{
  // number of variables is given in constructor
```

```
  n = 22*N_;

  // number of constraints is given in constructor
  m = 20*N_;

  // number of elements in the contraints jacobian is given
  // in the constructor
  nnz_jac_g = (5+5+4+4)*(N_-1) + (4*2+4*2+4*2+4*2)*N_;

  // number of elements in the complete hessian is given
  // in the constructor
  nnz_h_lag = 4*N_;

  // use the C style indexing (0-based) for the matrices
  index_style = TNLP::C_STYLE;

  return true;
}
```

The bounds for each variable and constraints then have to be set. A lower and upper bound has to supplied for all. IPOPT has a default absolute maximum and minimal numerical value of $1e19$. Any absolute values above this is considered to be infinity by the algorithm. The setting command `"nlp_lower_bound_inf"` and `"nlp_upper_bound_inf"` of the main file can regulate this system bound. An equality constraint is declared by applying the same lower and upper bound for a variable or equation constraint.

```
bool FourTankSystem_OCP::get_bounds_info(Index n, Number* x_l,
                                         Number* x_u, Index m,
                                         Number* g_l, Number* g_u)
{

  // masses (m1 - m4)
  x_l[0] = x_u[0] = 0.0;
  x_l[1*N_] = x_u[1*N_] = 0.0;
  x_l[2*N_] = x_u[2*N_] = 0.0;
  x_l[3*N_] = x_u[3*N_] = 0.0;

  for (Index i=0; i<N_; i++) {
    if (i > 0) {
      // masses (m1 - m4)
      x_l[i] = 0.0;       x_u[i] = 2e19;
      x_l[1*N_+i] = 0.0; x_u[1*N_+i] = 2e19;
      x_l[2*N_+i] = 0.0; x_u[2*N_+i] = 2e19;
      x_l[3*N_+i] = 0.0; x_u[3*N_+i] = 2e19;
    }
    // water levels (h1 - h4)
    x_l[4*N_+i] = 0.0; x_u[4*N_+i] = c_[9];
    x_l[5*N_+i] = 0.0; x_u[5*N_+i] = c_[10];
    x_l[6*N_+i] = 0.0; x_u[6*N_+i] = c_[11];
    x_l[7*N_+i] = 0.0; x_u[7*N_+i] = c_[12];
    // inflows (q1i - q4i)
```

```
82      x_l[8*N_+i] = -2e19;  x_u[8*N_+i] = 2e19;
        x_l[9*N_+i] = -2e19;  x_u[9*N_+i] = 2e19;
84      x_l[10*N_+i] = -2e19; x_u[10*N_+i] = 2e19;
        x_l[11*N_+i] = -2e19; x_u[11*N_+i] = 2e19;
86      // non-zeros approximations (h1s - h4s)
        x_l[12*N_+i] = -2e19; x_u[12*N_+i] = 2e19;
88      x_l[13*N_+i] = -2e19; x_u[13*N_+i] = 2e19;
        x_l[14*N_+i] = -2e19; x_u[14*N_+i] = 2e19;
90      x_l[15*N_+i] = -2e19; x_u[15*N_+i] = 2e19;
        // outflows (q1 - q4)
92      x_l[16*N_+i] = -2e19; x_u[16*N_+i] = 2e19;
        x_l[17*N_+i] = -2e19; x_u[17*N_+i] = 2e19;
94      x_l[18*N_+i] = -2e19; x_u[18*N_+i] = 2e19;
        x_l[19*N_+i] = -2e19; x_u[19*N_+i] = 2e19;
96      // valve presure (F1 - F2)
        x_l[20*N_+i] = 0.0; x_u[20*N_+i] = 300.0;
98      x_l[21*N_+i] = 0.0; x_u[21*N_+i] = 300.0;
    }
100
    // all constraints are equality constraints with
102 // right hand side zero
    for (Index i=0; i<N_; i++) {
104   g_l[i] = g_u[i] = 0.0;
      g_l[1*N_+i] = g_u[1*N_+i] = 0.0;
106   g_l[2*N_+i] = g_u[2*N_+i] = 0.0;
      g_l[3*N_+i] = g_u[3*N_+i] = 0.0;
108   // Water level equations
      g_l[4*N_+i] = g_u[4*N_+i] = 0.0;
110   g_l[5*N_+i] = g_u[5*N_+i] = 0.0;
      g_l[6*N_+i] = g_u[6*N_+i] = 0.0;
112   g_l[7*N_+i] = g_u[7*N_+i] = 0.0;
      // Inflow equations
114   g_l[8*N_+i] = g_u[8*N_+i] = 0.0;
      g_l[9*N_+i] = g_u[9*N_+i] = 0.0;
116   g_l[10*N_+i] = g_u[10*N_+i] = 0.0;
      g_l[11*N_+i] = g_u[11*N_+i] = 0.0;
118   // Non-zeros approximation equations
      g_l[12*N_+i] = g_u[12*N_+i] = 0.0;
120   g_l[13*N_+i] = g_u[13*N_+i] = 0.0;
      g_l[14*N_+i] = g_u[14*N_+i] = 0.0;
122   g_l[15*N_+i] = g_u[15*N_+i] = 0.0;
      // Outflow equations
124   g_l[16*N_+i] = g_u[16*N_+i] = 0.0;
      g_l[17*N_+i] = g_u[17*N_+i] = 0.0;
126   g_l[18*N_+i] = g_u[18*N_+i] = 0.0;
      g_l[19*N_+i] = g_u[19*N_+i] = 0.0;
128 }

130 return true;
}
```

An initial value is needed for all model variables in order to initiate the algorithm. It is also possible to supply initial values for the dual variables of the LP

solver and for the Lagrange variables of the KKT matrix, but only if these have been declared. I have not supplied these variables. I confirm this by using the "assert" statement. When initializing variables of a difference approximation it is wise to avoid a zero difference as initial guess. I circumvent this issue by separating each of the initial values with a small value.

```
bool FourTankSystem_OCP::get_starting_point(Index n, bool init_x,
                                            Number* x, bool init_z,
                                            Number* z_L, Number*
                                                z_U,
                                            Index m,
                                            bool init_lambda,
                                            Number* lambda)
{
  // I only have starting values for x
  assert(init_x == true);
  assert(init_z == false);
  assert(init_lambda == false);

  // initialize to the given starting point
  for (Index i=0; i<N_; i++) {
    // masses (m1 - m4)
    x[i] = x[1*N_+i] = x[2*N_+i] = x[3*N_+i] = 0.0+0.1*i/N_;
    // water levels (h1 - h4)
    x[4*N_+i] = x[5*N_+i] = x[6*N_+i] = x[7*N_+i] = 0.0;
    // inflows (q1i - q4i)
    x[8*N_+i] = x[9*N_+i] = x[10*N_+i] = x[11*N_+i] = 0.0;
    // non-zeros approximations (h1s - h4s)
    x[12*N_+i] = x[13*N_+i] = x[14*N_+i] = x[15*N_+i] = 0.0;
    // outflows (q1 - q4)
    x[16*N_+i] = x[17*N_+i] = x[18*N_+i] = x[19*N_+i] = 0.0;
    // valve presure (F1 - F2)
    x[20*N_+i] = x[21*N_+i] = 300.0;
  }

  return true;
}
```

The next step is to define the objective. Using the definition from (3.1.6) I store the objective in the object "obj_value".

```
bool FourTankSystem_OCP::eval_f(Index n, const Number* x,
                                bool new_x, Number& obj_value)
{
  obj_value = c_[19]*0.5*((x[4*N_] - c_[17])*(x[4*N_] - \
              c_[17]) + (x[5*N_] - c_[18])*(x[5*N_] - c_[18]));
  for (Index i=1; i<N_-1; i++) {
    obj_value += c_[19]*((x[4*N_+i] - c_[17])*(x[4*N_+i] - \
                 c_[17]) + (x[5*N_+i] - \
                 c_[18])*(x[5*N_+i] - c_[18]));
  }
  obj_value += c_[19]*0.5*((x[5*N_-1] - c_[17])*(x[5*N_-1] - \
               c_[17]) + (x[6*N_-1] - \
```

```
178                 c_[18])*(x[6*N_-1] - c_[18]));
    return true;
180 }
```

As part of the first derivatives the gradient of the objective is then defined. The gradient need to have the correct dimension.

```
    bool FourTankSystem_OCP::eval_grad_f(Index n, const Number* x,
184                                      bool new_x, Number* grad_f)
    {
186   grad_f[0] = grad_f[1*N_] = \
      grad_f[2*N_] = grad_f[3*N_] = 0.;
188   grad_f[4*N_] = c_[19]*(x[4*N_] - c_[17]);
      grad_f[5*N_] = c_[19]*(x[5*N_] - c_[18]);
190   grad_f[6*N_] = grad_f[7*N_] = \
      grad_f[8*N_] = grad_f[9*N_] = 0.;
192   grad_f[10*N_] = grad_f[11*N_] = \
      grad_f[12*N_] = grad_f[13*N_] = 0.;
194   grad_f[14*N_] = grad_f[15*N_] = \
      grad_f[16*N_] = grad_f[17*N_] = 0.;
196   grad_f[18*N_] = grad_f[19*N_] = \
      grad_f[20*N_] = grad_f[21*N_] = 0.;
198
      for (Index i=1; i<N_-1; i++) {
200     grad_f[i] = grad_f[1*N_+i] = \
        grad_f[2*N_+i] = grad_f[3*N_+i] = 0.;
202     grad_f[4*N_+i] = c_[19]*2*(x[4*N_+i] - c_[17]);
        grad_f[5*N_+i] = c_[19]*2*(x[5*N_+i] - c_[18]);
204     grad_f[6*N_+i] = grad_f[7*N_+i] = \
        grad_f[8*N_+i] = grad_f[9*N_+i] = 0.;
206     grad_f[10*N_+i] = grad_f[11*N_+i] = \
        grad_f[12*N_+i] = grad_f[13*N_+i] = 0.;
208     grad_f[14*N_+i] = grad_f[15*N_+i] = \
        grad_f[16*N_+i] = grad_f[17*N_+i] = 0.;
210     grad_f[18*N_+i] = grad_f[19*N_+i] = \
        grad_f[20*N_+i] = grad_f[21*N_+i] = 0.;
212   }
      grad_f[N_-1] = grad_f[2*N_-1] = \
214   grad_f[3*N_-1] = grad_f[4*N_-1] = 0.;
      grad_f[5*N_-1] = c_[19]*(x[5*N_-1] - c_[17]);
216   grad_f[6*N_-1] = c_[19]*(x[6*N_-1] - c_[18]);
      grad_f[7*N_-1] = grad_f[8*N_-1] = \
218   grad_f[9*N_-1] = grad_f[10*N_-1] = 0.;
      grad_f[11*N_-1] = grad_f[12*N_-1] = \
220   grad_f[13*N_-1] = grad_f[14*N_-1] = 0.;
      grad_f[15*N_-1] = grad_f[16*N_-1] = \
222   grad_f[17*N_-1] = grad_f[18*N_-1] = 0.;
      grad_f[19*N_-1] = grad_f[20*N_-1] = \
224   grad_f[21*N_-1] = grad_f[22*N_-1] = 0.;

226   return true;
    }
```

The constraints are then defined. This is the actual model.

```
230 bool FourTankSystem_OCP::eval_g(Index n, const Number* x,
                                    bool new_x, Index m, Number* g)
232 {
    g[0] = 0;
234 g[1*N_] = 0;
    g[2*N_] = 0;
236 g[3*N_] = 0;

238 for (Index i=1; i<N_; i++) {
      // Diff_Masses equations
240   g[i] = (x[i]-x[i-1]) - c_[19]*(c_[16]*x[8*N_+i] + \
            c_[16]*x[18*N_+i] - c_[16]*x[16*N_+i]);
242   g[1*N_+i] = (x[1*N_+i]-x[1*N_+i-1]) - \
                  c_[19]*(c_[16]*x[9*N_+i] + \
244               c_[16]*x[19*N_+i] - c_[16]*x[17*N_+i]);
      g[2*N_+i] = (x[2*N_+i]-x[2*N_+i-1]) - \
246               c_[19]*(c_[16]*x[10*N_+i] - \
                  c_[16]*x[18*N_+i]);
248   g[3*N_+i] = (x[3*N_+i]-x[3*N_+i-1]) - \
                  c_[19]*(c_[16]*x[11*N_+i] - \
250               c_[16]*x[19*N_+i]);
    }
252 for (Index i=0; i<N_; i++) {
      // Water level equations
254   g[4*N_+i] = x[4*N_+i] - x[i]/(c_[16]*c_[5]);
      g[5*N_+i] = x[5*N_+i] - x[1*N_+i]/(c_[16]*c_[6]);
256   g[6*N_+i] = x[6*N_+i] - x[2*N_+i]/(c_[16]*c_[7]);
      g[7*N_+i] = x[7*N_+i] - x[3*N_+i]/(c_[16]*c_[8]);
258   // Inflow equations
      g[8*N_+i] = x[8*N_+i] - c_[13]*x[20*N_+i];
260   g[9*N_+i] = x[9*N_+i] - c_[14]*x[21*N_+i];
      g[10*N_+i] = x[10*N_+i] - (1-c_[14])*x[21*N_+i];
262   g[11*N_+i] = x[11*N_+i] - (1-c_[13])*x[20*N_+i];
      // non-zeros approximation equations
264   g[12*N_+i] = x[12*N_+i] - x[4*N_+i]/\
              sqrt(c_[20]*log(exp(x[4*N_+i]/c_[20])+1));
266   g[13*N_+i] = x[13*N_+i] - x[5*N_+i]/\
              sqrt(c_[20]*log(exp(x[5*N_+i]/c_[20])+1));
268   g[14*N_+i] = x[14*N_+i] - x[6*N_+i]/\
              sqrt(c_[20]*log(exp(x[6*N_+i]/c_[20])+1));
270   g[15*N_+i] = x[15*N_+i] - x[7*N_+i]/\
              sqrt(c_[20]*log(exp(x[7*N_+i]/c_[20])+1));
272   // outflow equations
      g[16*N_+i] = x[16*N_+i] - c_[1]*sqrt(2*c_[15])*x[12*N_+i];
274   g[17*N_+i] = x[17*N_+i] - c_[2]*sqrt(2*c_[15])*x[13*N_+i];
      g[18*N_+i] = x[18*N_+i] - c_[3]*sqrt(2*c_[15])*x[14*N_+i];
276   g[19*N_+i] = x[19*N_+i] - c_[4]*sqrt(2*c_[15])*x[15*N_+i];
    }
278
    return true;
280 }
```

The big task of creating the Jacobian of the constraints has three types of value input. Since the Jacobian is supplied as a sparse matrix, first, the matrix entry has to be allocated defining the row and column value of the entry. Secondly, the actual value is assigned to the allocated matrix location. To keep track of this task I do this in two stages, each followed by an assertion statement. I have N of each constraint, so I loop through all N steps of each variable and constraint.

```
bool FourTankSystem_OCP::eval_jac_g(Index n, const Number* x,
                                    bool new_x, Index m,
                                    Index nele_jac, Index* iRow,
                                    Index *jCol, Number* values)
{
  if (values == NULL) {
    // return the structure of the jacobian
    Index inz = 0;
    for (Index i=1; i<N_; i++) {
      // Diff_Mass 1
      iRow[inz] = i; jCol[inz] = i-1;
      inz++;
      iRow[inz] = i; jCol[inz] = i;
      inz++;
      iRow[inz] = i; jCol[inz] = 8*N_+i;
      inz++;
      iRow[inz] = i; jCol[inz] = 18*N_+i;
      inz++;
      iRow[inz] = i; jCol[inz] = 16*N_+i;
      inz++;
      // Diff_Mass 2
      iRow[inz] = 1*N_+i; jCol[inz] = 1*N_+i-1;
      inz++;
      iRow[inz] = 1*N_+i; jCol[inz] = 1*N_+i;
      inz++;
      iRow[inz] = 1*N_+i; jCol[inz] = 9*N_+i;
      inz++;
      iRow[inz] = 1*N_+i; jCol[inz] = 19*N_+i;
      inz++;
      iRow[inz] = 1*N_+i; jCol[inz] = 17*N_+i;
      inz++;
      // Diff_Mass 3
      iRow[inz] = 2*N_+i; jCol[inz] = 2*N_+i-1;
      inz++;
      iRow[inz] = 2*N_+i; jCol[inz] = 2*N_+i;
      inz++;
      iRow[inz] = 2*N_+i; jCol[inz] = 10*N_+i;
      inz++;
      iRow[inz] = 2*N_+i; jCol[inz] = 18*N_+i;
      inz++;
      // Diff_Mass 4
      iRow[inz] = 3*N_+i; jCol[inz] = 3*N_+i-1;
      inz++;
      iRow[inz] = 3*N_+i; jCol[inz] = 3*N_+i;
      inz++;
      iRow[inz] = 3*N_+i; jCol[inz] = 11*N_+i;
```

```
        inz++;
330     iRow[inz] = 3*N_+i; jCol[inz] = 19*N_+i;
        inz++;
332   }
      for (Index i=0; i<N_; i++) {
334     // Water level 1
        iRow[inz] = 4*N_+i; jCol[inz] = 4*N_+i;
336     inz++;
        iRow[inz] = 4*N_+i; jCol[inz] = i;
338     inz++;
        // Water level 2
340     iRow[inz] = 5*N_+i; jCol[inz] = 5*N_+i;
        inz++;
342     iRow[inz] = 5*N_+i; jCol[inz] = 1*N_+i;
        inz++;
344     // Water level 3
        iRow[inz] = 6*N_+i; jCol[inz] = 6*N_+i;
346     inz++;
        iRow[inz] = 6*N_+i; jCol[inz] = 2*N_+i;
348     inz++;
        // Water level 4
350     iRow[inz] = 7*N_+i; jCol[inz] = 7*N_+i;
        inz++;
352     iRow[inz] = 7*N_+i; jCol[inz] = 3*N_+i;
        inz++;
354     // Inflow 1
        iRow[inz] = 8*N_+i; jCol[inz] = 8*N_+i;
356     inz++;
        iRow[inz] = 8*N_+i; jCol[inz] = 20*N_+i;
358     inz++;
        // Inflow 2
360     iRow[inz] = 9*N_+i; jCol[inz] = 9*N_+i;
        inz++;
362     iRow[inz] = 9*N_+i; jCol[inz] = 21*N_+i;
        inz++;
364     // Inflow 3
        iRow[inz] = 10*N_+i; jCol[inz] = 10*N_+i;
366     inz++;
        iRow[inz] = 10*N_+i; jCol[inz] = 21*N_+i;
368     inz++;
        // Inflow 4
370     iRow[inz] = 11*N_+i; jCol[inz] = 11*N_+i;
        inz++;
372     iRow[inz] = 11*N_+i; jCol[inz] = 20*N_+i;
        inz++;
374     // Approx 1
        iRow[inz] = 12*N_+i; jCol[inz] = 12*N_+i;
376     inz++;
        iRow[inz] = 12*N_+i; jCol[inz] = 4*N_+i;
378     inz++;
        // Approx 2
380     iRow[inz] = 13*N_+i; jCol[inz] = 13*N_+i;
        inz++;
382     iRow[inz] = 13*N_+i; jCol[inz] = 5*N_+i;
        inz++;
```

```
384         // Approx 3
            iRow[inz] = 14*N_+i; jCol[inz] = 14*N_+i;
386         inz++;
            iRow[inz] = 14*N_+i; jCol[inz] = 6*N_+i;
388         inz++;
            // Approx 4
390         iRow[inz] = 15*N_+i; jCol[inz] = 15*N_+i;
            inz++;
392         iRow[inz] = 15*N_+i; jCol[inz] = 7*N_+i;
            inz++;
394         // Outflow 1
            iRow[inz] = 16*N_+i; jCol[inz] = 16*N_+i;
396         inz++;
            iRow[inz] = 16*N_+i; jCol[inz] = 12*N_+i;
398         inz++;
            // Outflow 2
400         iRow[inz] = 17*N_+i; jCol[inz] = 17*N_+i;
            inz++;
402         iRow[inz] = 17*N_+i; jCol[inz] = 13*N_+i;
            inz++;
404         // Outflow 3
            iRow[inz] = 18*N_+i; jCol[inz] = 18*N_+i;
406         inz++;
            iRow[inz] = 18*N_+i; jCol[inz] = 14*N_+i;
408         inz++;
            // Outflow 4
410         iRow[inz] = 19*N_+i; jCol[inz] = 19*N_+i;
            inz++;
412         iRow[inz] = 19*N_+i; jCol[inz] = 15*N_+i;
            inz++;
414       }
          /* sanity check */
416       assert(inz==nele_jac);
        }
418     else {
          // return the values of the jacobian of the constraints
420       Index inz = 0;
          for (Index i=0; i<(N_-1); i++) {
422         // Diff_Mass 1
            values[inz] = -1;
424         inz++;
            values[inz] = 1;
426         inz++;
            values[inz] = -c_[19]*c_[16];
428         inz++;
            values[inz] = -c_[19]*c_[16];
430         inz++;
            values[inz] = c_[19]*c_[16];
432         inz++;
            // Diff_Mass 2
434         values[inz] = -1;
            inz++;
436         values[inz] = 1;
            inz++;
438         values[inz] = -c_[19]*c_[16];
```

```
        inz++;
440     values[inz] = -c_[19]*c_[16];
        inz++;
442     values[inz] = c_[19]*c_[16];
        inz++;
444     // Diff_Mass 3
        values[inz] = -1;
446     inz++;
        values[inz] = 1;
448     inz++;
        values[inz] = -c_[19]*c_[16];
450     inz++;
        values[inz] = c_[19]*c_[16];
452     inz++;
        // Diff_Mass 4
454     values[inz] = -1;
        inz++;
456     values[inz] = 1;
        inz++;
458     values[inz] = -c_[19]*c_[16];
        inz++;
460     values[inz] = c_[19]*c_[16];
        inz++;
462   }
      for (Index i=0; i<N_; i++) {
464     // Water level 1
        values[inz] = 1;
466     inz++;
        values[inz] = -1/(c_[16]*c_[5]);
468     inz++;
        // Water level 2
470     values[inz] = 1;
        inz++;
472     values[inz] = -1/(c_[16]*c_[6]);
        inz++;
474     // Water level 3
        values[inz] = 1;
476     inz++;
        values[inz] = -1/(c_[16]*c_[7]);
478     inz++;
        // Water level 4
480     values[inz] = 1;
        inz++;
482     values[inz] = -1/(c_[16]*c_[8]);
        inz++;
484     // Inflow 1
        values[inz] = 1;
486     inz++;
        values[inz] = -c_[13];
488     inz++;
        // Inflow 2
490     values[inz] = 1;
        inz++;
492     values[inz] = -c_[14];
        inz++;
```

```
494         // Inflow 3
            values[inz] = 1;
496         inz++;
            values[inz] = -(1-c_[14]);
498         inz++;
            // Inflow 4
500         values[inz] = 1;
            inz++;
502         values[inz] = -(1-c_[13]);
            inz++;
504         // Approx 1
            values[inz] = 1;
506         inz++;
            values[inz] = -1/(2*sqrt(c_[20]*log(exp(x[4*N_+i]/c_[20])+1))
                );
508         inz++;
            // Approx 2
510         values[inz] = 1;
            inz++;
512         values[inz] = -1/(2*sqrt(c_[20]*log(exp(x[5*N_+i]/c_[20])+1))
                );
            inz++;
514         // Approx 3
            values[inz] = 1;
516         inz++;
            values[inz] = -1/(2*sqrt(c_[20]*log(exp(x[6*N_+i]/c_[20])+1))
                );
518         inz++;
            // Approx 4
520         values[inz] = 1;
            inz++;
522         values[inz] = -1/(2*sqrt(c_[20]*log(exp(x[7*N_+i]/c_[20])+1))
                );
            inz++;
524         // Outflow 1
            values[inz] = 1;
526         inz++;
            values[inz] = -c_[1]*sqrt(2*c_[15]);
528         inz++;
            // Outflow 2
530         values[inz] = 1;
            inz++;
532         values[inz] = -c_[2]*sqrt(2*c_[15]);
            inz++;
534         // Outflow 3
            values[inz] = 1;
536         inz++;
            values[inz] = -c_[3]*sqrt(2*c_[15]);
538         inz++;
            // Outflow 4
540         values[inz] = 1;
            inz++;
542         values[inz] = -c_[4]*sqrt(2*c_[15]);
            inz++;
544     }
```

```
    assert(inz==nele_jac);
  }

  return true;
}
```

I use the same strategy when defining the KKT matrix. In the case of the Four Tank System, almost all constraints are linear, making most of the values in the KKT matrix zero. My sparse linear equation solver is able to handle symmetric matrices, so only the upper or lower triangle of the matrix need to be supplied.

IPOPT offers a way of avoiding the implementation of the KKT matrix. Using the option `"SetStringValue("hessian_test","limited-memory")"` will approximate this hessian using the L-BFGS (sparse quasi-Newton) method [4]. This will of course slow down the algorithm compared to using the exact Hessian. Iterations statistics is compared in Table 5.1.

```
bool FourTankSystem_OCP::eval_h(Index n, const Number* x,
                                bool new_x, Number obj_factor,
                                Index m, const Number* lambda,
                                bool new_lambda, Index nele_hess,
                                Index* iRow, Index* jCol,
                                Number* values)
{
  if (values == NULL) {

    Index inz = 0;
    for (Index i=0; i<N_; i++) {
      // Outflows
      iRow[inz] = 4*N_+i; jCol[inz] = 4*N_+i;
      inz++;
      iRow[inz] = 5*N_+i; jCol[inz] = 5*N_+i;
      inz++;
      iRow[inz] = 6*N_+i; jCol[inz] = 6*N_+i;
      inz++;
      iRow[inz] = 7*N_+i; jCol[inz] = 7*N_+i;
      inz++;
    }
    assert(inz == nele_hess);
  }
  else {
    // return the values. This is a symmetric matrix, filling
    // the upper right triangle only

    Index inz = 0;
    // Outflows
    values[inz] = obj_factor * c_[19];
    values[inz] -= lambda[12*N_] * \
           (-1/(4*(c_[20]*log(exp(x[4*N_]/c_[20])+1))*\
           sqrt(c_[20]*log(exp(x[4*N_]/c_[20])+1))));
    inz++;
```

```
586     values[inz] = obj_factor * c_[19];
        values[inz] -= lambda[13*N_] * \
588             (-1/(4*(c_[20]*log(exp(x[5*N_]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[5*N_]/c_[20])+1))));
590     inz++;
        values[inz] = obj_factor * 0.;
592     values[inz] -= lambda[14*N_] * \
                (-1/(4*(c_[20]*log(exp(x[6*N_]/c_[20])+1))*\
594             sqrt(c_[20]*log(exp(x[6*N_]/c_[20])+1))));
        inz++;
596     values[inz] = obj_factor * 0.;
        values[inz] -= lambda[15*N_] * \
598             (-1/(4*(c_[20]*log(exp(x[7*N_]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[7*N_]/c_[20])+1))));
600     inz++;
        for (Index i=1; i<N_-1; i++) {
602       // Outflows
          values[inz] = obj_factor * c_[19]*2.;
604       values[inz] -= lambda[12*N_+i] * \
                (-1/(4*(c_[20]*log(exp(x[4*N_+i]/c_[20])+1))*\
606             sqrt(c_[20]*log(exp(x[4*N_+i]/c_[20])+1))));
          inz++;
608       values[inz] = obj_factor * c_[19]*2.;
          values[inz] -= lambda[13*N_+i] * \
610             (-1/(4*(c_[20]*log(exp(x[5*N_+i]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[5*N_+i]/c_[20])+1))));
612       inz++;
          values[inz] = obj_factor * 0.;
614       values[inz] -= lambda[14*N_+i] * \
                (-1/(4*(c_[20]*log(exp(x[6*N_+i]/c_[20])+1))*\
616             sqrt(c_[20]*log(exp(x[6*N_+i]/c_[20])+1))));
          inz++;
618       values[inz] = obj_factor * 0.;
          values[inz] -= lambda[15*N_+i] * \
620             (-1/(4*(c_[20]*log(exp(x[7*N_+i]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[7*N_+i]/c_[20])+1))));
622       inz++;
        }
624     // Outflows
        values[inz] = obj_factor * c_[19];
626     values[inz] -= lambda[13*N_-1] * \
                (-1/(4*(c_[20]*log(exp(x[5*N_-1]/c_[20])+1))*\
628             sqrt(c_[20]*log(exp(x[5*N_-1]/c_[20])+1))));
        inz++;
630     values[inz] = obj_factor * c_[19];
        values[inz] -= lambda[14*N_-1] * \
632             (-1/(4*(c_[20]*log(exp(x[6*N_-1]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[6*N_-1]/c_[20])+1))));
634     inz++;
        values[inz] = obj_factor * 0.;
636     values[inz] -= lambda[15*N_-1] * \
                (-1/(4*(c_[20]*log(exp(x[7*N_-1]/c_[20])+1))*\
638             sqrt(c_[20]*log(exp(x[7*N_-1]/c_[20])+1))));
        inz++;
640     values[inz] = obj_factor * 0.;
```

```
      values[inz] -= lambda[16*N_-1] * \
642             (-1/(4*(c_[20]*log(exp(x[8*N_-1]/c_[20])+1))*\
                sqrt(c_[20]*log(exp(x[8*N_-1]/c_[20])+1))));
644     inz++;

646     assert(inz == nele_hess);
    }
648
    return true;
650 }
```

Finally, I finalize the algorithm by defining a self chosen output. Since I would like to plot my result, I create a data file containing the solution of my favorite variables. In the shell window I make the algorithm display the final optimal value of the objective.

```
void FourTankSystem_OCP::finalize_solution(SolverReturn status,
653                                        Index n,
                                           const Number* x,
655                                        const Number* z_L,
                                           const Number* z_U,
657                                        Index m,
                                           const Number* g,
659                                        const Number* lambda,
                                           Number obj_value,
661                                        const IpoptData* ip_data
                                               ,
                                           IpoptCalculatedQuantities
                                               * ip_cq)
663 {

665   // Write output to dat-file for plotting
      FILE *fp;
667     const char *name;
      const char *options;
669
      name="plot4tank.dat";
671     options="w";

673     fp=fopen(name,options);
      for (Index i=0; i<N_; i++) {
675       fprintf(fp,
                "%16.4e %16.4e %16.4e %16.4e %16.4e %16.4e %16.4e\n",
677             i*c_[19], x[4*N_+i], x[5*N_+i], x[6*N_+i],
                x[7*N_+i], x[20*N_+i], x[21*N_+i]);
679     }
      fclose (fp);
681
      /* Display solution */
683     printf("\n\nObjective value\n");
      printf("f(x*) = %e\n", obj_value);
685 }
```

I then compile the code file (make) and run the unix-archive of the code. As a way of comparing the various sparse linear equation solvers and Hessian approximations, I here display Table 5.1, stating the terminal outputs of a number of these. The table is sorted with respect to number of iterations steps and Total CPU time.

Sparse linear eq. solver	Hessian Approx.	Time Steps	# Iter.	CPU Time [sec]	Obj. value
MA27	Exact	300	42	0.662	5.3798e+03
Mumps	Exact	300	35	3.884	5.3798e+03
Mumps with Metis	Exact	300	38	4.316	5.3798e+03
MA27	L-BFGS	300	324	10.184	5.3798e+03
Mumps	L-BFGS	300	168	36.939	5.3798e+03
Mumps with Metis	L-BFGS	300	231	52.113	5.3798e+03
MA27	Exact	100	39	0.218	5.4467e+03
Mumps with Metis	Exact	100	37	1.565	5.4467e+03
Mumps	Exact	100	37	1.605	5.4467e+03
MA27	L-BFGS	100	292	3.609	5.4467e+03
Mumps	L-BFGS	100	140	11.269	5.4467e+03
Mumps with Metis	L-BFGS	100	140	11.343	5.4467e+03
MA27	Exact	30	36	0.077	5.7126e+03
MA27	L-BFGS	30	73	0.305	5.7126e+03
Mumps	Exact	30	32	0.388	5.7126e+03
Mumps with Metis	Exact	30	32	0.391	5.7126e+03
Mumps	L-BFGS	30	132	3.188	5.7126e+03
Mumps with Metis	L-BFGS	30	132	4.006	5.7126e+03

Table 5.1: Iterations Statistics of IPOPT

For more information on the terminal output values see section 6 of the IPOPT Documentation.

I use MatLab as plotting tool. The solution to the model, I just presented, is provided in Figure 5.1 and 5.2.

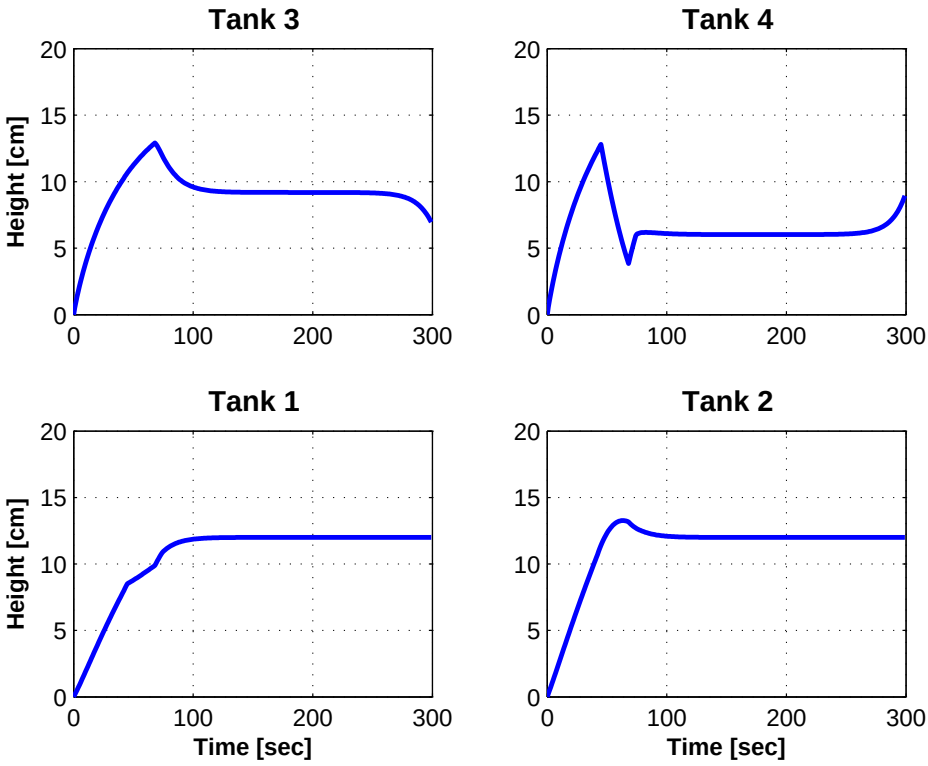


Figure 5.1: Height of liquid in tanks

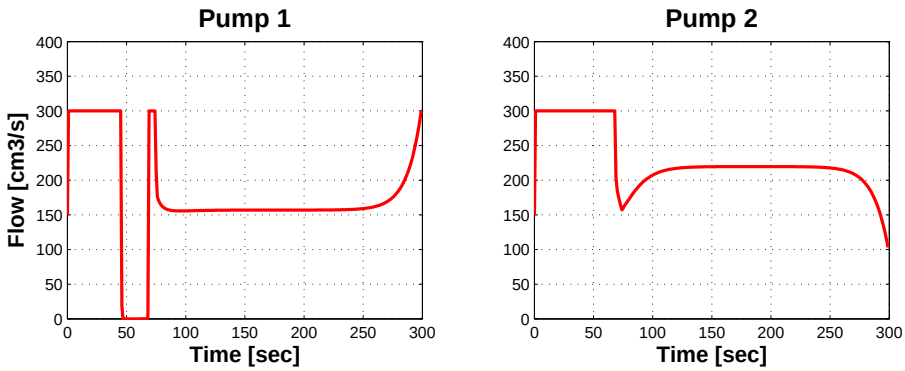


Figure 5.2: Control variables (pump flow)

Chapter 6

CppAD

In this chapter, I touch upon a package for differentiation of C++ algorithms, CppAD (C++ **A**lgorithmic **D**ifferentiation). This packages is a way to use IPOPT without having to provide information on the derivatives of ones optimization model. The chapter will not be extensive, but just introduce the package and how it can be used. I will go through the installation of CppAD, providing an installation procedure only for unix systems. Then I will provide an implementation of the four tank system model using CppAD.

URL: [CppAD Home] https://projects.coin-or.org/CppAD/wiki

6.1 CppAD - Getting Started

CppAD is an Open-Source package for differentiation of C++ algorithms, provided under a Common Public License. CppAD should be considered an add-on package for IPOPT. CppAD is interfaced just like IPOPT C++ and provides the same range of user setting options as IPOPT using the same syntax. The software JModelica presented in the next chapter uses both IPOPT and CppAD in order to solve optimization models.

6.1.1 Installation

In order to use CppAD it is required that IPOPT has been installed (see 5.1) and that the IPOPT shared libraries is located at a system environment variable directory. The code package for CppAD is available from CppAD Home under a Common Public License, meaning that it is available free of charge both for private and commercial purposes.

In order to install and run CppAD you need to make sure that a C++ compiler is installed on the system. This requirement is the same as for IPOPT and further information on this subject can be found at 5.1.

As for IPOPT, CppAD does not provide a direct plotting function package, but outputs can be produced to be plotted using Gnuplot (installation see 4.1.1.1) or ones own favorite plotting tool.

The CppAD program package can now be downloaded (CppAD also provides the option of subversion check out) and unpacked from

http://www.coin-or.org/download/source/CppAD/ - Latest stable version is 20100425.

Go to your download folder and extract the files

```
> tar -xvzf cppad-20100425.gpl.tgz
```

Move the directory to a preferred system directory.

CppAD can now be installed. Run the configure script

```
> ./configure IPOPT_DIR=path/to/IPOPT/build-dir
```

Build and install CppAD

```
> make install
```

Check if all executables work and run all examples

```
> make test
```

6.2 Simulating the Four Tank System using CppAD with IPOPT

Referring back to the Four Tank System from chapter 3, I will start by summarizing the model that is to be implemented. Then I will meticulously go through the implementation of the model. A range of CppAD functionalities and properties will be presented, when new objects and functions is required, while implementing the model.

The complete model is given as (3.19a) - (3.20n) and the parameter vector, p, in the model is defined as

$$p = [a_1\, a_2\, a_3\, a_4\, A_1\, A_2\, A_3\, A_4\, \gamma_1\, \gamma_2\, g\, \rho\, r_1\, r_2\, s]^T$$

CppAD does not offer a way of defining differential equations or integration expressions exact. The system therefore has to be discretized and these types of expression has to be approximated. The implementation will follow the implementation of IPOPT C++ and the syntax will be exactly the same. The difference between CppAD and IPOPT is that I do not have to supply the derivatives of the model, only the model itself. For more on the discretization and equation approximation see section 5.1.

In the following I will build up the C++ source files for solving the Four Tank System seen as an optimal control problem. First, I present the main structure of the program, where I will fill code into.

I include the CppAD IPOPT non-linear problem solver, and a package for output printing. Secondly, I construct a namespace containing the class `FG_info`. This class is derived from the base class `cppad_ipopt_fg_info`. Certain virtual member functions of `fg_info` are used to compute the value of $fg(x)$. $fg(x)$ is a vector function containing the objective and constraints of the model. The member function `eval_r` is a pure virtual function and must be defined in the derived class `FG_info` as an `ADVector` object. This function computes the value of $r_k(u)$ used in the representation for $fg(x)$. $r_k(u)$ is a simple function representations of $fg(x)$ (for more on this see http://www.coin-or.org/CppAD/Doc/cppad_ipopt_nlp.htm). `ADVector` is a SimpleVector class with elements of type `ADNumber`. `ADNumber` is an AD type that can be used to compute derivatives. The member function `retape` has type bool. `retape` is either set to true or false. If `retape` is true, `cppad_ipopt_nlp` will retape the operation sequence corresponding to $r_k(u)$ for every value of u. An `cppad_ipopt_nlp` object should use much less memory and run faster if `retape` is false.

The main method then contains the lower and upper limits of the variables and

constraints together with the initial values of the variables. Finally, an IPOPT interface is created and solved just like an IPOPT problem.

```
/* $Id: FourTankSystem.cpp 2010-04-24 Rune Brus $ */
// BEGIN PROGRAM

# include "../src/cppad_ipopt_nlp.hpp"
# include <stdio.h>

namespace {

   class FG_info : public cppad_ipopt_fg_info
   {
   private:
         bool retape_;
   public:
     // derived class part of constructor
     FG_info(bool retape)
         : retape_ (retape)
     { }
     // Evaluation of the objective f(x), and constraints g(x)
     // using an Algorithmic Differentiation (AD) class.
     ADVector eval_r(size_t k, const ADVector&  x)
     {
       return fg;
     }

     bool retape(size_t k)
     {   return retape_; }

   };
}

int main(void)
{
   // initial value of the independent variables
   NumberVector x_i(n);

   // lower and upper limits for x
   NumberVector x_l(n);
   NumberVector x_u(n);

   // lower and upper limits for g
   NumberVector g_l(m);
   NumberVector g_u(m);

   // create the Ipopt interface
   cppad_ipopt_solution solution;
   Ipopt::SmartPtr<Ipopt::TNLP> cppad_nlp = new cppad_ipopt_nlp(n, m
       , x_i, x_l, x_u, g_l, g_u, fg_info, &solution);

   // Create an instance of the IpoptApplication
   using Ipopt::IpoptApplication;
   Ipopt::SmartPtr<IpoptApplication> app = new IpoptApplication();

```

```
  // Initialize the IpoptApplication and process the options
53  app->Initialize();

55  // Run the IpoptApplication
    Ipopt::ApplicationReturnStatus status = app->OptimizeTNLP(
        cppad_nlp);
57
    return (int) status;
59
}
61
// END PROGRAM
```

I will now supply the code and solver options for the program structure just presented.

First, I declare a global variable, given as N, for defining the size of the problem.

```
/* $Id: FourTankSystem.cpp 2010-04-24 Rune Brus $ */
2 // BEGIN PROGRAM

4 # include "../src/cppad_ipopt_nlp.hpp"
  # include <stdio.h>
6
  // size of problem
8 int N = 300;
```

Before constructing the FG_info class within my namespace, I declare and define a parameter function for the system.

```
10 namespace {

12 NumberVector Param()
     {
14     // parameters
       NumberVector c(21);
16     // Time horizont
       c[0] = 300;
18     // Cross sectional area of outlet [cm2]
       c[1] = 1.2272;
20     c[2] = 1.2272;
       c[3] = 1.2272;
22     c[4] = 1.2272;
       // Cross sectional area of inlet [cm2]
24     c[5] = 380.1327;
       c[6] = 380.1327;
26     c[7] = 380.1327;
       c[8] = 380.1327;
28     // Height of tank [cm]
       c[9] = 20;
30     c[10] = 20;
```

```
        c[11] = 20;
        c[12] = 20;
        // Valve presure
        c[13] = 0.15;
        c[14] = 0.25;
        // Acceleration of gravity [cm/s2]
        c[15] = 981;
        // Density of water [g/cm3]
        c[16] = 1.00;
        // Water stability level
        c[17] = 12;
        c[18] = 12;
        // Step size
        c[19] = c[0]/N;
        // Approximation constant
        c[20] = 0.1;

        return c;
    }
```

Then I construct the FG_info class described above. The syntax of the objective and constraints follows the syntax from the IPOPT implementation from section 5.2.1.

```
class FG_info : public cppad_ipopt_fg_info
{
private:
      bool retape_;
public:
  // derived class part of constructor
  FG_info(bool retape)
      : retape_ (retape)
  { }
  // Evaluation of the objective f(x), and constraints g(x)
  // using an Algorithmic Differentiation (AD) class.
  ADVector eval_r(size_t k, const ADVector&  x)
  {
    size_t i;

    // number of independent variables
    // (domain dimension for f and g)
    size_t n = 20*N+1;

    ADVector fg(n);

    NumberVector c(21);
    c = Param();

    // f(x)
    fg[0] = c[19]*0.5*((x[4*N] - c[17])*(x[4*N] - c[17]) + (x[5*N
        ] - c[18])*(x[5*N] - c[18]));

    for (i=1; i<N-1; i++) {
```

```
79          fg[0] += c[19]*((x[4*N+i] - c[17])*(x[4*N+i] - c[17]) + (x
                [5*N+i] - c[18])*(x[5*N+i] - c[18]));
        }
81
        fg[0] += c[19]*0.5*((x[5*N-1] - c[17])*(x[5*N-1] - c[17]) + (
            x[6*N-1] - c[18])*(x[6*N-1] - c[18]));
83
        // g_1 (x) - g_20 (x)
85      fg[1] = 0;
        fg[1*N+1] = 0;
87      fg[2*N+1] = 0;
        fg[3*N+1] = 0;
89
        for (i=1; i<N; i++) {
91        // Diff_Masses equations
          fg[i+1] = (x[i]-x[i-1]) - c[19]*(c[16]*x[8*N+i] + c[16]*x
              [18*N+i] - c[16]*x[16*N+i]);
93        fg[1*N+i+1] = (x[1*N+i]-x[1*N+i-1]) - c[19]*(c[16]*x[9*N+i]
               + c[16]*x[19*N+i] - c[16]*x[17*N+i]);
          fg[2*N+i+1] = (x[2*N+i]-x[2*N+i-1]) - c[19]*(c[16]*x[10*N+i
              ] - c[16]*x[18*N+i]);
95        fg[3*N+i+1] = (x[3*N+i]-x[3*N+i-1]) - c[19]*(c[16]*x[11*N+i
              ] - c[16]*x[19*N+i]);
        }
97      for (i=0; i<N; i++) {
          // Water level equations
99        fg[4*N+i+1] = x[4*N+i] - x[i]/(c[16]*c[5]);
          fg[5*N+i+1] = x[5*N+i] - x[1*N+i]/(c[16]*c[6]);
101       fg[6*N+i+1] = x[6*N+i] - x[2*N+i]/(c[16]*c[7]);
          fg[7*N+i+1] = x[7*N+i] - x[3*N+i]/(c[16]*c[8]);
103       // Inflow equations
          fg[8*N+i+1] = x[8*N+i] - c[13]*x[20*N+i];
105       fg[9*N+i+1] = x[9*N+i] - c[14]*x[21*N+i];
          fg[10*N+i+1] = x[10*N+i] - (1-c[14])*x[21*N+i];
107       fg[11*N+i+1] = x[11*N+i] - (1-c[13])*x[20*N+i];
          // non-zeros approximation equations
109       fg[12*N+i+1] = x[12*N+i] - x[4*N+i]/sqrt(c[20]*log(exp(x[4*
              N+i]/c[20])+1));
          fg[13*N+i+1] = x[13*N+i] - x[5*N+i]/sqrt(c[20]*log(exp(x[5*
              N+i]/c[20])+1));
111       fg[14*N+i+1] = x[14*N+i] - x[6*N+i]/sqrt(c[20]*log(exp(x[6*
              N+i]/c[20])+1));
          fg[15*N+i+1] = x[15*N+i] - x[7*N+i]/sqrt(c[20]*log(exp(x[7*
              N+i]/c[20])+1));
113       // outflow equations
          fg[16*N+i+1] = x[16*N+i] - c[1]*sqrt(2*c[15])*x[12*N+i];
115       fg[17*N+i+1] = x[17*N+i] - c[2]*sqrt(2*c[15])*x[13*N+i];
          fg[18*N+i+1] = x[18*N+i] - c[3]*sqrt(2*c[15])*x[14*N+i];
117       fg[19*N+i+1] = x[19*N+i] - c[4]*sqrt(2*c[15])*x[15*N+i];
        }
119
        return fg;
121   }
      bool retape(size_t k)
123   {   return retape_; }
```

```
    };
}
```

The main method is then constructed. I define the number of variables and the number of constraints before calling the parameter function.

```
int main(void)
{
    // number of independent variables (domain dimension for f and g)
    size_t n = 22*N;
    // number of constraints (range dimension for g)
    size_t m = 20*N;

    size_t i;

    NumberVector c(21);
    c = Param();
```

I then define the initial values of the model variables

```
    // initial value of the independent variables
    NumberVector x_i(n);
    for (i=0; i<N; i++) {
        // masses (m1 - m4)
        x_i[i] = x_i[1*N+i] = x_i[2*N+i] = x_i[3*N+i] = 0.0+0.1*i/N;
        // water levels (h1 - h4)
        x_i[4*N+i] = x_i[5*N+i] = x_i[6*N+i] = x_i[7*N+i] = 0.0;
        // inflows (q1i - q4i)
        x_i[8*N+i] = x_i[9*N+i] = x_i[10*N+i] = x_i[11*N+i] = 0.0;
        // non-zeros approximations (h1s - h4s)
        x_i[12*N+i] = x_i[13*N+i] = x_i[14*N+i] = x_i[15*N+i] = 0.0;
        // outflows (q1 - q4)
        x_i[16*N+i] = x_i[17*N+i] = x_i[18*N+i] = x_i[19*N+i] = 0.0;
        // valve presure (F1 - F2)
        x_i[20*N+i] = x_i[21*N+i] = 300.0;
    }
```

I then define the lower and upper limits of the variables

```
    // lower and upper limits for x
    NumberVector x_l(n);
    NumberVector x_u(n);
    // masses (m1 - m4)
    x_l[0] = x_u[0] = 0.0;
    x_l[1*N] = x_u[1*N] = 0.0;
    x_l[2*N] = x_u[2*N] = 0.0;
    x_l[3*N] = x_u[3*N] = 0.0;

    for (i=0; i<N; i++) {
        if (i > 0) {
```

```
168         // masses (m1 - m4)
            x_l[i] = 0.0;       x_u[i] = 2e19;
170         x_l[1*N+i] = 0.0; x_u[1*N+i] = 2e19;
            x_l[2*N+i] = 0.0; x_u[2*N+i] = 2e19;
172         x_l[3*N+i] = 0.0; x_u[3*N+i] = 2e19;
        }
174     // water levels (h1 - h4)
        x_l[4*N+i] = 0.0; x_u[4*N+i] = c[9];
176     x_l[5*N+i] = 0.0; x_u[5*N+i] = c[10];
        x_l[6*N+i] = 0.0; x_u[6*N+i] = c[11];
178     x_l[7*N+i] = 0.0; x_u[7*N+i] = c[12];
        // inflows (q1i - q4i)
180     x_l[8*N+i] = -2e19;  x_u[8*N+i] = 2e19;
        x_l[9*N+i] = -2e19;  x_u[9*N+i] = 2e19;
182     x_l[10*N+i] = -2e19; x_u[10*N+i] = 2e19;
        x_l[11*N+i] = -2e19; x_u[11*N+i] = 2e19;
184     // non-zeros approximations (h1s - h4s)
        x_l[12*N+i] = -2e19; x_u[12*N+i] = 2e19;
186     x_l[13*N+i] = -2e19; x_u[13*N+i] = 2e19;
        x_l[14*N+i] = -2e19; x_u[14*N+i] = 2e19;
188     x_l[15*N+i] = -2e19; x_u[15*N+i] = 2e19;
        // outflows (q1 - q4)
190     x_l[16*N+i] = -2e19; x_u[16*N+i] = 2e19;
        x_l[17*N+i] = -2e19; x_u[17*N+i] = 2e19;
192     x_l[18*N+i] = -2e19; x_u[18*N+i] = 2e19;
        x_l[19*N+i] = -2e19; x_u[19*N+i] = 2e19;
194     // valve presure (F1 - F2)
        x_l[20*N+i] = 0.0; x_u[20*N+i] = 300.0;
196     x_l[21*N+i] = 0.0; x_u[21*N+i] = 300.0;
    }
```

I then define the lower and upper limits of the constraints

```
198 // lower and upper limits for g
    NumberVector g_l(m);
200 NumberVector g_u(m);
    for (i=0; i<N; i++) {
202   g_l[i] = g_u[i] = 0.0;
      g_l[1*N+i] = g_u[1*N+i] = 0.0;
204   g_l[2*N+i] = g_u[2*N+i] = 0.0;
      g_l[3*N+i] = g_u[3*N+i] = 0.0;
206   // Water level equations
      g_l[4*N+i] = g_u[4*N+i] = 0.0;
208   g_l[5*N+i] = g_u[5*N+i] = 0.0;
      g_l[6*N+i] = g_u[6*N+i] = 0.0;
210   g_l[7*N+i] = g_u[7*N+i] = 0.0;
      // Inflow equations
212   g_l[8*N+i] = g_u[8*N+i] = 0.0;
      g_l[9*N+i] = g_u[9*N+i] = 0.0;
214   g_l[10*N+i] = g_u[10*N+i] = 0.0;
      g_l[11*N+i] = g_u[11*N+i] = 0.0;
216   // Non-zeros approximation equations
      g_l[12*N+i] = g_u[12*N+i] = 0.0;
218   g_l[13*N+i] = g_u[13*N+i] = 0.0;
```

```
    g_l[14*N+i] = g_u[14*N+i] = 0.0;
    g_l[15*N+i] = g_u[15*N+i] = 0.0;
    // Outflow equations
    g_l[16*N+i] = g_u[16*N+i] = 0.0;
    g_l[17*N+i] = g_u[17*N+i] = 0.0;
    g_l[18*N+i] = g_u[18*N+i] = 0.0;
    g_l[19*N+i] = g_u[19*N+i] = 0.0;
  }
```

I now create my IPOPT interface setting `retape` to false. This setting will limit the memory requirements of the solver.

```
  // object in derived class
  bool retape = false;
  FG_info my_fg_info(retape);
  cppad_ipopt_fg_info *fg_info = &my_fg_info;

  // create the Ipopt interface
  cppad_ipopt_solution solution;
  Ipopt::SmartPtr<Ipopt::TNLP> cppad_nlp = new cppad_ipopt_nlp(n, m
      , x_i, x_l, x_u, g_l, g_u, fg_info, &solution);
```

I then create an instance of the IpoptApplication, supplying solver settings using IPOPT syntax.

```
  // Create an instance of the IpoptApplication
  using Ipopt::IpoptApplication;
  Ipopt::SmartPtr<IpoptApplication> app = new IpoptApplication();

  // maximum number of iterations
  app->Options()->SetIntegerValue("max_iter", 300);

  // approximate accuracy in first order necessary conditions;
  app->Options()->SetNumericValue("tol", 1e-7);
  app->Options()->SetStringValue("mu_strategy", "adaptive");
  app->Options()->SetStringValue("output_file", "fts.out");

  // Initialize the IpoptApplication and process the options
  app->Initialize();
```

Finally, I solve the problem and write the results to an output file.

```
  // Run the IpoptApplication
  Ipopt::ApplicationReturnStatus status = app->OptimizeTNLP(
      cppad_nlp);

  // ---------------------- Output solution -----------------------
  if (status == Ipopt::Solve_Succeeded) {
    printf("\n\n*** The problem solved!\n");

```

```
    // Write output to dat-file for plotting
260 FILE *fp;
    const char *name;
262 const char *options;

264 name="fts.dat";
    options="w";
266
    fp=fopen(name,options);
268 for (i=0; i<N; i++) {
      fprintf(fp,"%16.4e %16.4e %16.4e %16.4e %16.4e %16.4e\
          n", i*c[19], solution.x[4*N+i], solution.x[5*N+i],
          solution.x[6*N+i], solution.x[7*N+i], solution.x[20*N+i],
           solution.x[21*N+i]);
270 }
    fclose (fp);
272
  }
274 else {
    printf("\n\n*** The problem FAILED!\n");
276 }

278 return (int) status;

280 }

282 // END PROGRAM
```

I then compile the code file, using my own *makefile* and run the unix-archive of the code. As a way of comparing computation times, I here displays Table 6.1 stating the terminal outputs of a number of different discretizations.

Sparse linear eq. solver	Algorithmic Differentiation	Time Steps	# Iter.	CPU Time [sec]	Obj. value
Mumps with Metis	CppAD	300	36	724.607	5.3798e+03
Mumps with Metis	CppAD	100	29	65.436	5.4467e+03
Mumps with Metis	CppAD	30	24	5.198	5.7126e+03

Table 6.1: Iterations Statistics of CppAD

The long computation time stem from the large number of functions evaluations CppAD have to compute in order to calculate the derivatives of the model. The same phenomena is seen in section 5.2.1 when the IPOPT solver is set to use the optional Hessian approximation instead of the exact Hessian. In CppAD's case, all derivatives have to be approximated which demand a lot of memory and many function evaluation.

For more information on the terminal output values see section 6 of the IPOPT

Documentation.

I use MatLab as plotting tool. The solution to the model, I just presented is provided in Figure 6.1 and 6.2.

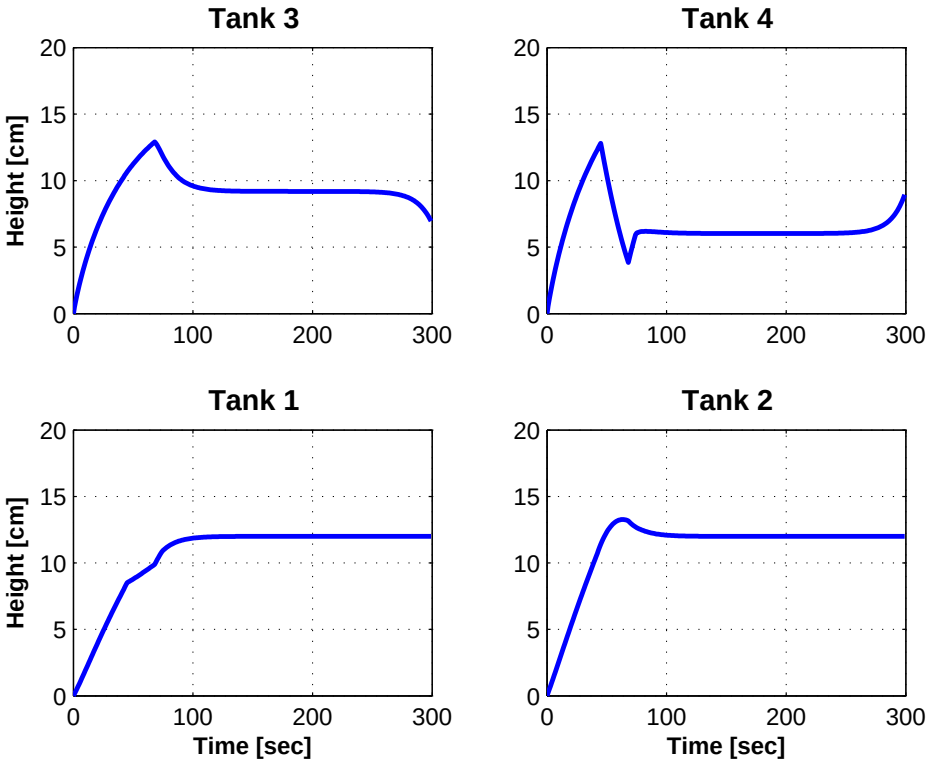


Figure 6.1: Height of liquid in tanks

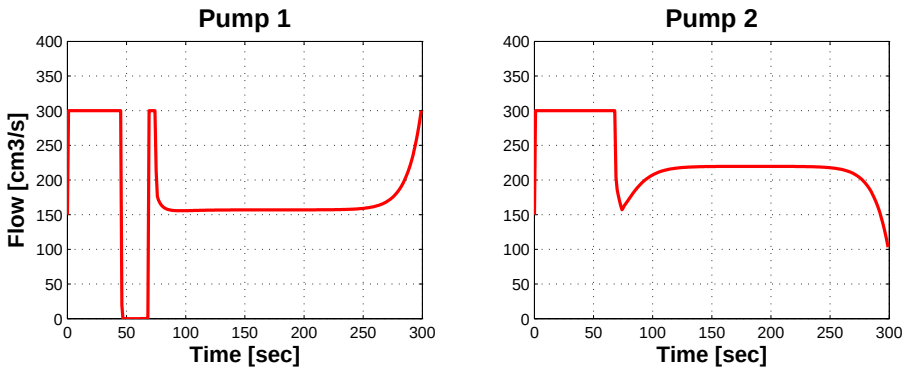


Figure 6.2: Control variables (pump flow)

Chapter 7

JModelica

In this chapter, I introduce the optimization software JModelica. A short introduction of the origin and main features will be provided. Afterwards, I will go through the installation of the software, providing installation procedures for both Linux, Mac OS X and Windows. Then a guide to producing and running a simple program will be given. Finally, I will go through all elements on how to simulate and solve the Quadruple Tank Process using this software. In addition I provide the outputs and plots of the model compilation. This presentation should not be seen as a full user guide, only a presentation of how to simulate the four tank system model using this software.

References: Sequential Quadratic Programming [4], Optimica and JModelica.org-Languages [18], Tools for Optimization of Large-Scale Systems [19], Optimica [20], Modelica and JastAdd [21], JModelica - an Open Source Platform [22], Modelica and JastAdd [23], Modelica and Optimica [24], XML Representation of DAE Systems [25], Multiple-Shooting and JModelica [26], Numerical Methods for Dynamic Optimization [27]

URLs: [JModelica Home] http://www.jmodelica.org/,
[IPOPT Home] http://www.coin-or.org/projects/Ipopt.xml,
[SUNDIALS Home] https://computation.llnl.gov/casc/sundials/main.html,

[Python Home] http://www.python.org/

7.1 JModelica - Getting Started

JModelica.org is an open source platform for optimization using the Modelica compiler. JModelica provide an environment to simulate and optimize complex dynamic systems in a intuitive modeling language. JModelica is only a language to express dynamic optimization problems and transcribe them into code. As such, JModelica.org is intended to provide a platform where state of the art algorithms can be propagated into industrial use. The JModelica developers currently suggest IPOPT as the NLP solver. For simulations of algebraic systems the open-source software SUNDIALS is used. JModelica is a result of research at the Department of Automatic Control, Lund University, and is now maintained and developed by Modelon AB in collaboration with academia.

For the origin of the name of the software see the following article http://www.jmodelica.org/faq/5.

7.1.1 Installation

The software package is available from the address http://www.jmodelica.org/page/12 under the GNU Affero General Public License (see more at http://www.jmodelica.org/page/24). Modelon AB also offers complementing commercial licenses, which is even less restrictive than GNU Affero General Public License.

The interface of JModelica is Python user environment based and Python is therefore a requirement. The version Python 2.6 is recommended and can be installed from http://www.python.org/download/releases/2.6/. A Java JRE/SDK installation is needed as well. A full overview of the architecture of JModelica can be seen at http://www.jmodelica.org/page/25.

Besides the above, the Python user environment need the following extension packages (a guide on how to install these is provided in the sections below)

- JPype (http://jpype.sourceforge.net/)
- lxml (http://codespeak.net/lxml/)
- NumPy 1.2.0 (http://numpy.scipy.org/)

- SciPy 0.7 (http://www.scipy.org/)
- PySundials 2.3 (http://sourceforge.net/projects/pysundials/)
- pyreadline >=1.5 (http://ipython.scipy.org/dist/)
- Matplotlib (http://matplotlib.sourceforge.net/)
- Nose (http://code.google.com/p/python-nose/)

Before demonstrating the use of the solver, an installation guide will be provided for Linux, Mac OS X and Windows.

7.1.1.1 Installation under Linux

For all the below compilers and packages it is possible that they are already installed on your system. If this is the case, it is of course not necessary to install them.

In order to install and run JModelica you need to make sure that a C compiler is installed on the system. This can be checked by prompting the unix command (other C-compilers than GCC is just as fine)

```
> g++ -v
```

First of all, the IPOPT solver need to be installed. See section 5.1.1.1 for an installation guide for this tool.

Secondly, SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) is needed. It is important to install the version of SUNDIALS that work together with PySundials 2.3, hence the 2.3 version of SUNDIALS is needed (PySundials 2.4 is not yet available).

Download the source file package of SUNDIALS 2.3 from
https://computation.llnl.gov/casc/sundials/download/download.html.

Go to your download folder and extract the files

```
> tar xfvz sundials-2.3.0.tar.gz
```

Move the directory to a preferred system directory. Go to your chosen system directory of sundials-2.3.0 and create a build folder

```
> mkdir build
> cd build
```

When building the package it is important that the sundials library files are created as dynamic link libraries. Otherwise the JModelica installation script will not be able to link the libraries to itself. An extensive example library, of how to use SUNDIALS as standalone solver, can also be created in connection with the SUNDIALS installation.

Run the configuration script, build the package, and install

```
> ../configure --enable-shared --enable-examples
> make
> sudo make install
```

The above configuration includes the example library and creates dynamic link libraries.

Third, the Python packages need to be installed. Begin by installing the Python Setuptools from http://pypi.python.org/pypi/setuptools. This will enable you to use the "easy_install" unix command for installing Python packages. Use APT (Advance Package Tool, see 4.1.1.1 for information) and prompt

```
> sudo apt-get install python-setuptools
```

JPype allows Python programs full access to Java class libraries. A requirement is therefore that a Java JDK/SDK is installed. Check if this is the case and locate the java home directory. On linux this directory typically is locate at "/usr/lib/jvm/name_of_JDK" or "/usr/java/name_of_JDK". If Java is not installed, a compiler can be installed using APT by prompting

```
> sudo apt-get install sun-java6
```

Download the JPype source-file package, go to the download folder, and unzip.

```
> unzip JPype-0.5.4.1.zip
```

Move the directory to a preferred system directory. Open the setup script "*setup.py*" and change line 19 to reflect the path of your JDK installation (this is only necessary on linux). Run the setup script.

```
> cd path/to/JPype-0.5.4.1
```

```
> sudo python setup.py install
```

Python can still have some problems importing JPype, if a JAVA_HOME environmental variable is not set. This can be circumvented by adding the JAVA_HOME path to the Python bash files, JModelica uses as startup Python scripts.

The next package is lxml. lxml is a binder for libxml2 and libxslt . To install lxml it is therefore required that libxml2, libxml2 developer package, libxslt, and the libxslt developer package, is installed beforehand. libxml2 is a XML C parser and toolkit and libxslt is a XSLT C library. They are needed because JModelica produces and handles XML code. All these packages can easily be installed using the Python setuptools. Install by prompting (the directory is subordinate, easy_install will place the archives in the system Python library)

```
> sudo easy_install libxml2
> sudo easy_install libxml2-dev
> sudo easy_install libxslt
> sudo easy_install libxslt-dev
> sudo easy_install lxml
```

Next step is then the scientific computing package NumPy for Python and the NumPy add-on library SciPy. These can be installed using APT. SciPy depends on NumPy, so the installation order is not unimportant. Install by prompting (as above, the directory is subordinate)

```
> sudo apt-get install python-numpy
> sudo apt-get install python-scipy
```

The PyReadline is a keyboard support package for Python. It is important to mention that it only supports windows-pc keyboards. If you are using a macintosh or third kind of keyboard, PyReadline can not be used. Install using Python setuptools.

```
> sudo easy_install pyreadline
```

Matplotlib is a python 2D plotting library using Matlab-like syntax. This can be installed using APT.

```
> sudo apt-get install python-matplotlib
```

Nose is a unittest extension for Python that JModelica uses to test, whether all Python packages are successfully installed and locatable. Install Nose using Python setuptools.

```
> sudo easy_install nose
```

It is now time to link the SUNDIALS shared library files to Python. This is done by retrieving the Pysundials source file package and installing it. The installation will only be successful if SUNDIALS has been installed correctly (see above) and if the version numbers of SUNDIALS and Pysundials are the same. Currently there are differences between the downloadable source-file package and the Subversion checkout package of Pysundials. I recommend using the Subversion checkout version. This of course requires Subversion support. Subversion can be installed using APT. Install Subversion and retrieve and install Pysundials.

```
> sudo apt-get install subversion
> cd path/to/chosen/pysundials/directory
> svn co https://pysundials.svn.sourceforge.net/svnroot/pysundials pysundials
> cd pysundials
> sudo python setup.py install
```

To finalize the Pysundials installation make sure that the shared library interpreter `libsundials_core_aux.so` has been placed among the SUNDIALS shared library, otherwise move it there yourself.

The last Python package to be installed is the enhanced interactive Python shell IPython. This can be used as a Python shell instead of Pythons' default shell. This installation is optional. The user is free to simply use Pythons' default shell. IPython can be installed using Python setuptools.

```
> sudo easy_install ipython
```

The JModelica program package can now be downloaded. Subversion support is needed for this (Subversion is preinstall for mac OS X users). Go to a chosen installation directory and download by prompting the unix command

```
> svn co https://svn.jmodelica.org/trunk JModelica
```

JModelica requires the Java-based build tool Apache Ant to be installed. APT supports Apache Ant.

```
> sudo apt-get install ant
```

JModelica can now be installed. Create a build folder for compiling

```
> mkdir build
> cd build
```

Run the configure script

```
> ../configure -with-ipopt=/path/to/ipopt-install-dir
```

Build the code

```
> make
```

Install

```
> make install
```

It is possible that JModelica is not able to locate the lapack library, this can be solved by installing liblapack-dev using APT and reinstalling JModelica. When reinstalling JModelica start by prompting

```
> make clean
> make distclean
```

The documentation can then be generated. The documentation is xml based and requires Doxygen to be installed. Install both by prompting

```
> sudo apt-get install doxygen
> make docs
```

7.1.1.2 Installation under Mac OS X

IPOPT, SUNDAILS, JPype and PySundials need to be installed just like on Linux, but there are two easy ways to get around installing the rest of required Python packages.

Using the first method, you begin by installing the Python package setuptools from http://pypi.python.org/pypi/setuptools. This will enable you to use the "easy_install" unix command. Then install lxml by prompting the unix command

```
> easy_install lxml
```

The rest of the packages can be be install using the Scipy Superpack from http://macinscience.org/.

The second method is even simpler. Use MacPorts (Macintosh packing tool, see 4.1.1.1 for information) and prompt

```
> sudo port install py26-lxml py26-numpy py26-scipy py26-matplotlib py26-nose
```

This method will though be expected to take some time, since the ports automatically installs all dependencies. You can use the prompt

```
> port deps py26-lxml py26-numpy py26-scipy py26-matplotlib py26-nose
```

to get an overview of these dependencies.

7.1.1.3 Installation under Windows

A SDK has been developed for installing JModelica on Windows, so much less footwork has to done to make a successful installation. This SDK is bundled with most of the required third-party software, namely

- MinGW, MSYS (http://www.mingw.org/)
- Ipopt (http://www.coin-or.org/Ipopt/)
- JPype (http://jpype.sourceforge.net/)
- lxml (http://codespeak.net/lxml)
- nose (http://somethingaboutorange.com/mrl/projects/nose/)
- SUNDIALS (https://computation.llnl.gov/casc/sundials)
- PySUNDIALS (http://pysundials.sourceforge.net/)

Before running this windows executable, the user still need to install

- Python 2.5 (http://www.python.org/)
- IPython (http://ipython.scipy.org/moin/Download)

- Java Runtime Environment (http://java.sun.com/javase/downloads/)
- NumPy 1.2.0 (http://numpy.scipy.org/)
- SciPy 0.7 (http://www.scipy.org/)
- pyreadline >=1.5 (http://ipython.scipy.org/dist/)
- Matplotlib (http://matplotlib.sourceforge.net/)

7.1.2 Running a program file in JModelica

JModelica supplies two bash files for setting up Python and IPython. These bash files automatically links to the required libraries when using JModelica. Windows users can just double click one of these from the JModelica program folders. Using a unix terminal, prompt

```
> sh /path/to/JModelica/build/Python/jm_python.sh
```

or

```
> sh /path/to/JModelica/build/Python/jm_ipython.sh
```

Start by testing that all packages has been installed successfully by prompting in the python shell

```
>>> import jmodelica as jm
>>> jm.check_packages()
```

The capabilities of JModelica can be tested by running the models from the JModelica model library (this will produce a lot of output files, so make sure your in a proper directory). The JModelica model library can be located at the directory `"JModelica/Python/src/jmodelica/examples"`. These models can be run by prompting

```
>>> from jmodelica.examples import NameOfModel as model
>>> model.run_demo()
```

When writing ones own JModelica program, two files has to be supplied. First a Modelica package file with the extension *.mo*, defining the parameters , objective (if there is one), the system equations, and the constraints. The second file is then the Python script code, which construct, simulate, solve and plot the model.

I will demonstrate this using the Optimal Time of Rocket model introduced in chapter 1 of the thesis. The model is given as (1.6a) - (1.6m) and the implementation looks as below. First, the Modelica package file is formulated.

The objective value is a parameter in the model. Since it is a parameter, I need to set it to a fixed value using the Modelica model language. But the value is in reality unknown before the problems is solved and I can therefore set the parameter to be a free parameter. The Optimal Time of Rocket model is only provided as a way of getting started with JModelica. Detailed informations on how to formulate these model files is provided in the section regarding the implementation of the Four Tank model.

```
package Rocket_pack

  optimization Rocket_Min_Time (objective = tf,startTime = 0,
      finalTime = 10)

    // The parameters
    parameter Real tf(free=true,min=2)=10 "Final time";

    // The states
    Real s(start=0);
    Real v(start=0);
    Real m(start=1.0);

    // The control signal with bounds
    input Real u(min=-1.1,max=1.1);

  equation
    der(s) = (v);
    der(v) = ((u)-0.2*(v)^2)/((m)+0.01);
    der(m) = -0.01*(u)^2;

  constraint
    s(finalTime)=10.0;
    v(finalTime)=0;
    v<=1.7;
    v>=-0.1;

  end Rocket_Min_Time;

end Rocket_pack;
```

Then the Python script

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Import the JModelica.org Python packages
import jmodelica
import jmodelica.jmi as jmi
```

```
from jmodelica.compiler import OptimicaCompiler
from jmodelica.optimization import ipopt

# Import numerical libraries
import numpy as N
import ctypes as ct
import matplotlib.pyplot as plt

def run(with_plots=True):
    """Demonstrate how to solve a minimum time
    dynamic optimization problem based on a
    simplified Rocket flight."""

    oc = OptimicaCompiler()
    oc.set_boolean_option('state_start_values_fixed',True)

    # Name of .mo file (Modelica model package file)
    mofile = 'Rocket.mo'
    # Retrieving the model named Rocket_Min_Time from
    # the Modelica model package file
    model_name = 'Rocket_pack.Rocket_Min_Time'
    # Name standard of JModelica. All output files of JModelica
    # uses this name standard matching the model name.
    # The user can not define his/her own package name.
    model_package = 'Rocket_pack_Rocket_Min_Time'

    # Compile the Optimica model first to C code and
    # then to a dynamic library
    oc.compile_model(mofile, model_name, target='ipopt')

    # Load the dynamic library and XML data
    rc=jmi.Model(model_package)

    # Initialize the mesh
    n_e = 50 # Number of elements
    hs = N.ones(n_e)*1./n_e # Equidistant points
    n_cp = 3; # Number of collocation points in each element

    # Create an NLP object
    nlp = ipopt.NLPCollocationLagrangePolynomials(rc,n_e,hs,n_cp)

    # Create an Ipopt NLP object
    nlp_ipopt = ipopt.CollocationOptimizer(nlp)

    # Solve the optimization problem
    nlp_ipopt.opt_sim_ipopt_solve()

    # Write to file. The resulting file can also be
    # loaded into Dymola.
    nlp.export_result_dymola()

    # Load the file we just wrote to file
    res = jmodelica.io.ResultDymolaTextual('Rocket_pack_Rocket_Min_
        Time_result.txt')
```

```
    # Extract variable profiles
    s=res.get_variable_data('s')
    v=res.get_variable_data('v')
    m=res.get_variable_data('m')
    tf=res.get_variable_data('tf')


    if with_plots:
        # Plot
        plt.figure(1)
        plt.clf()
        plt.hold(True)
        plt.plot(s.t,s.x)
        plt.plot(v.t,v.x)
        plt.plot(m.t,m.x)
        plt.ylabel('Time [s]')
        plt.grid()
        plt.show()

if __name__ == "__main__":
    run()
```

I then go to my Rocket model directory and run the IPython startup script. Within the IPython environment I then import my model and run it

```
> cd path/to/rocket_ocp-dir
> sh /path/to/jm_ipython.sh
  >>> import rocket_ocp as rc
  >>> rc.run()
```

7.2 Simulating the Four Tank System using JModelica

Referring back to the Four Tank System from chapter 3, I will start by summarizing the model which is to be implemented. Then I will meticulously go through the implementation of the model using JModelica. Doing so, I will focus on attention demanding challenges and there solutions.

The complete model is given as (3.19a) - (3.20n) and the parameter vector, p, in the model is defined as

$$p = [a_1\, a_2\, a_3\, a_4\, A_1\, A_2\, A_3\, A_4\, \gamma_1\, \gamma_2\, g\, \rho\, r_1\, r_2\, s]^T$$

Using this notation, the system of differential equations determining the evolution of the system can then be represented as

$$\frac{dx(t)}{dt} = f(t, x(t), u(t), p) \quad x(t_0) = x_0 \quad t \in [t_0, t_{end}]$$

7.2.1 Implementing the model

In the following I will build up the two Modelica model files and the Python solver file for optimizing and simulating the Four Tank System seen as an optimal control problem. First I present the main structure of the two Modelica files. The first Modelica file consist of the main packages structure "FTS" and the model structure "FTS_Opt", while the second Modelica file likewise consist of the main packages structure "FTS" and the model structure "FTS_Sim".

I start by constructing the Modelica file optimization. The objective is defined as part of the model structure definition. The objective is defined as the function "cost". The objective is a Lagrange term and will be optimized over the time interval 0 to 300 seconds. Since the "cost" function is an integrand, this function will be defined as a derivative under the equations part of "FTS_Opt".

```
1  package FTS

3      optimization FTS_Opt (objective = (cost(finalTime)),
                                 startTime = 0,finalTime = 300)
5
                           // Parameters ...
7
                           // Variables ...
9
                           // Equations ...
11
                           // Constraints ...
13
       end FTS_Opt;
15
   end FTS;
```

I will now fill out the four content parts of the "FTS_Opt" structure. First, I define the parameters.

JModelica supplies an archive of SI units that can be used as extra parameter information. The basic declaration call is "parameter Real". Using the SI unit library the declaration call could be "parameter Modelica.SIunits.Acceleration". Finally, the user can define her own units using the for example the call "parameter Real g(unit="cm/s2") = 981".

I will just use the basic declaration call.

```
    // ---------- Parameters ----------
7   // Process parameters
        parameter Real A1=380.1327, A2=380.1327, A3=380.1327, A4
            =380.1327;
9       parameter Real a1=1.2272, a2=1.2272, a3=1.2272, a4=1.2272;
        parameter Real g=981, s=0.1, rho=1.0;
11      parameter Real gamma1=0.15, gamma2=0.25;

13  // Reference values
        parameter Real r1 = 12;
15      parameter Real r2 = 12;
```

Secondly, I define the variables. The variable declarations can have four additional inputs, namely "`initialGuess`", "`start`", "`min`" and "`max`". These can be defined in random succession. I implement my variable constraints, such that no redundant constraint is defined. This means that I restrict the water levels to by positive and and the tanks to have an upper bound of 20 cm. I do not have to force the tanks to have a lower bound of 0, since I automatically gets this from the water level constraint.

```
17  // ---------- Variables ----------
    // Liquid masses
19      Real m1(min=0.0);
        Real m2(min=0.0);
21      Real m3(min=0.0);
        Real m4(min=0.0);
23  // Tank levels
        Real h1(max=20);
25      Real h2(max=20);
        Real h3(max=20);
27      Real h4(max=20);
    // Tank approximation levels
29      Real h1s;
        Real h2s;
31      Real h3s;
        Real h4s;
33
    // Inputs (Control variables)
35      input Real u1(initialGuess=150,min=0,max=300);
        input Real u2(initialGuess=150,min=0,max=300);
37
    // Objective
39      Real cost(start=0,min=0);
```

The I define the model equations. Derivatives is defined using the notation "`der(function)`".

```
41  // ---------- Equation constraints ----------
  equation
```

```
43          der(cost) = ((h1) - r1)^2 + ((h2) - r2)^2;

45          h1 = (m1)/(A1*rho);
            h2 = (m2)/(A2*rho);
47          h3 = (m3)/(A3*rho);
            h4 = (m4)/(A4*rho);
49
            h1s = (h1)/(sqrt(s*log(exp((h1)/s)+1)));
51          h2s = (h2)/(sqrt(s*log(exp((h2)/s)+1)));
            h3s = (h3)/(sqrt(s*log(exp((h3)/s)+1)));
53          h4s = (h4)/(sqrt(s*log(exp((h4)/s)+1)));

55          // System differential equations
            der(m1) = -rho*a1*sqrt(2*g)*(h1s) + rho*a3*sqrt(2*g)*(h3s)
                + rho*gamma1*u1;
57          der(m2) = -rho*a2*sqrt(2*g)*(h2s) + rho*a4*sqrt(2*g)*(h4s)
                + rho*gamma2*u2;
            der(m3) = -rho*a3*sqrt(2*g)*(h3s) +  rho*(1-gamma2)*u2;
59          der(m4) = -rho*a4*sqrt(2*g)*(h4s) +  rho*(1-gamma1)*u1;
```

Lastly, I define the variable constraints.

```
61      // ---------- Initial conditions ----------
    constraint
63          m1(startTime) = 0.0;
            m2(startTime) = 0.0;
65          m3(startTime) = 0.0;
            m4(startTime) = 0.0;
```

My second Modelica file is for simulating the found solution of the control variables. The file consisting of the main packages structure "FTS" and the model structure "FTS_Sim". This file only separate itself from the first Modelica file by having no objective function defined. The parameters, variables, equations constraints are the same giving the structure

```
    package FTS
2
    model FTS_Sim
4       // Process parameters
            parameter Real A1=380.1327, A2=380.1327, A3=380.1327, A4
                =380.1327;
6           parameter Real a1=1.2272, a2=1.2272, a3=1.2272, a4=1.2272;
            parameter Real g=981, s=0.1, rho=1.0;
8           parameter Real gamma1=0.15, gamma2=0.25;

10      // Liquid masses
            Real m1;
12          Real m2;
            Real m3;
14          Real m4;
        // Tank levels
```

```
16          Real h1;
            Real h2;
18          Real h3;
            Real h4;
20      // Tank approximation levels
            Real h1s;
22          Real h2s;
            Real h3s;
24          Real h4s;

26      // Inputs
            input Real u1;
28          input Real u2;

30    equation
            h1 = (m1)/(A1*rho);
32          h2 = (m2)/(A2*rho);
            h3 = (m3)/(A3*rho);
34          h4 = (m4)/(A4*rho);

36          h1s = (h1)/(sqrt(s*log(exp((h1)/s)+1)));
            h2s = (h2)/(sqrt(s*log(exp((h2)/s)+1)));
38          h3s = (h3)/(sqrt(s*log(exp((h3)/s)+1)));
            h4s = (h4)/(sqrt(s*log(exp((h4)/s)+1)));
40
            // System differential equations
42          der(m1) = -rho*a1*sqrt(2*g)*(h1s) + rho*a3*sqrt(2*g)*(h3s)
                + rho*gamma1*u1;
            der(m2) = -rho*a2*sqrt(2*g)*(h2s) + rho*a4*sqrt(2*g)*(h4s)
                + rho*gamma2*u2;
44          der(m3) = -rho*a3*sqrt(2*g)*(h3s) +  rho*(1-gamma2)*u2;
            der(m4) = -rho*a4*sqrt(2*g)*(h4s) +  rho*(1-gamma1)*u1;
46  end FTS_Sim;

48  end FTS;
```

The Python solver script now have to be created.

I make my Python script directly executable and define the source code encoding as UTF-8. All the needed python libraries is then imported. If your are susing Mac OS X it is possible that a backend for plotting should be defined, Python will always self provide one using Windows and Linux. I use "TkAgg", which is my x11 graphical terminal. The Modelica compiler is for simulating the system together with SUNDIALS and the Optimica compiler is for solving the system together with IPOPT.

```
#!/usr/bin/python
2 # -*- coding: utf-8 -*-

4 # Import numerical libraries
  import numpy as N
6 import ctypes as ct
```

```
import matplotlib
matplotlib.use('TkAgg')
from pylab import figure, show

# Import the JModelica.org Python packages
import jmodelica
import jmodelica.jmi as jmi
from jmodelica.compiler import ModelicaCompiler
from jmodelica.compiler import OptimicaCompiler
from jmodelica.optimization import ipopt
from jmodelica.initialization.ipopt import NLPInitialization
from jmodelica.initialization.ipopt import InitializationOptimizer
from jmodelica.simulation.sundials import
    TrajectoryLinearInterpolation
from jmodelica.simulation.sundials import SundialsDAESimulator
```

I then initiate my Python execution environment, which can then be executed using the call "run()". The "model_name" refer to the Modelica model within the main packages structure "FTS". The "model_package" is a naming convention and has to reflect the package - and model name using underscores. JModelica creates a lot of output files using this exact naming convention. The user can not choose her own model package name. I then define a compiler instance and finally a nlp object, which is then solved.

```
def run(with_plots=True):

    # Create a Modelica compiler instance
    oc = OptimicaCompiler()

    # Compile the stationary initialization model into a DLL
    mofile = 'FourTankSystem.mo'
    model_name = 'FTS.FTS_Opt'
    model_package = 'FTS_FTS_Opt'

    oc.compile_model(mofile, model_name, target='ipopt')

    # Load a model instance into Python
    ocp = jmi.Model(model_package)

    # Initialize the mesh
    n_e = 100 # Number of elements
    hs = N.ones(n_e)*1./n_e # Equidistant points
    n_cp = 3; # Number of collocation points in each element

    # Create an NLP object
    nlp = ipopt.NLPCollocationLagrangePolynomials(ocp,n_e,hs,n_cp)

    # Create an Ipopt NLP object
    nlp_ipopt = ipopt.CollocationOptimizer(nlp)

    nlp_ipopt.opt_sim_ipopt_set_int_option("max_iter",300)

```

```
    # Solve the optimization problem
51  nlp_ipopt.opt_sim_ipopt_solve()
```

The result is then written into an output file, which I again load and plot. The matplotlib packages supplies Matlab like plotting commands for Python and further commands can be seen at http://matplotlib.sourceforge.net/

```
53  # Write to file. The resulting file can also be
    # loaded into Dymola.
55  nlp.export_result_dymola()

57  # Load the file we just wrote to file
    res = jmodelica.io.ResultDymolaTextual('FTS_FTS_Opt_result.txt'
        )
59
    # Extract variable profiles
61  h1=res.get_variable_data('h1')
    h2=res.get_variable_data('h2')
63  h3=res.get_variable_data('h3')
    h4=res.get_variable_data('h4')
65  u1=res.get_variable_data('u1')
    u2=res.get_variable_data('u2')
67
    cost=res.get_variable_data('cost')
69
    if with_plots:
71      # Plot
        fig = figure(1)
73      plt = fig.add_subplot(111)
        plt.plot(h1.t,h1.x)
75      plt.plot(h2.t,h2.x)
        plt.plot(h3.t,h3.x)
77      plt.plot(h4.t,h4.x)
        plt.legend(('Tank 1','Tank 2','Tank 3','Tank 4'))
79      plt.set_xlabel('Time [s]')
        plt.set_ylabel('Water level [cm]')
81      plt.set_title('Tank Levels Solution')
        plt.grid()
83
        fig2 = figure(2)
85      plt2 = fig2.add_subplot(111)
        plt2.plot(u1.t,u1.x)
87      plt2.plot(u2.t,u2.x)
        plt2.legend(('Pump 1','Pump 2'))
89      plt2.set_xlabel('Time [s]')
        plt2.set_ylabel('Pressure [cm3/s]')
91      plt2.set_title('Control Variables Solution')
        plt2.grid()
93      show()
```

I then simulate the output control variable data using Sundials. A Modelica

instance is created and the required data is extracted. The control variables is transformed into a Trajectory object that can be passed to the simulator and a differential algebraic equation system is initialized and solved.

```
95      # Simulating the control values

97      # Create a Modelica compiler instance for simulation
        oc2 = ModelicaCompiler()
99
        # Simulate to verify the optimal solution
101     mofile2 = 'FourTankSystemSIM.mo'
        sim_name2 = 'FTS.FTS_Sim'
103     sim_package2 = 'FTS_FTS_Sim'

105     # Load the file we just wrote to file
        res = jmodelica.io.ResultDymolaTextual('FTS_FTS_Opt_result.txt'
            )
107
        # Extract variable profiles
109     u1_res=res.get_variable_data('u1')
        u2_res=res.get_variable_data('u2')
111
        # Set up input trajectory
113     t = u1_res.t
        u1 = u1_res.x
115     u2 = u2_res.x
        u = N.array([u1, u2])
117     u = N.transpose(u)
        u_traj = TrajectoryLinearInterpolation(t,u)
119
        # Comile the Modelica model first to C code and
121     # then to a dynamic library
        oc2.compile_model(mofile2, sim_name2, target='ipopt')
123
        # Load the dynamic library and XML data
125     sim_model=jmi.Model(sim_package2)

127     # Create DAE initialization object.
        init_nlp = NLPInitialization(sim_model)
129
        # Create an Ipopt solver object for the DAE initialization
            system
131     init_nlp_ipopt = InitializationOptimizer(init_nlp)

133     # Solve the DAE initialization system with Ipopt
        init_nlp_ipopt.init_opt_ipopt_solve()
```

This model is then simulated using Sundials

```
        simulator = SundialsDAESimulator(sim_model, verbosity=3, start_
            time=0, final_time=300, time_step=0.01,input=u_traj)
137     # Run simulation
        simulator.run()
```

Finally the result is plotted and the Python execution environment is finalized.

```
140     # Store simulation data to file
        simulator.write_data()
142
        # Load the file we just wrote to file
144     res = jmodelica.io.ResultDymolaTextual('FTS_FTS_Sim_result.txt'
            )

146     # Extract variable profiles
        h1=res.get_variable_data('h1')
148     h2=res.get_variable_data('h2')
        h3=res.get_variable_data('h3')
150     h4=res.get_variable_data('h4')
        u1=res.get_variable_data('u1')
152     u2=res.get_variable_data('u2')

154     if with_plots:
            # Plot
156         fig3 = figure()
            plt3 = fig3.add_subplot(111)
158         plt3.plot(h1.t,h1.x)
            plt3.plot(h2.t,h2.x)
160         plt3.plot(h3.t,h3.x)
            plt3.plot(h4.t,h4.x)
162         plt3.legend(('Tank 1','Tank 2','Tank 3','Tank 4'))
            plt3.set_xlabel('Time [s]')
164         plt3.set_ylabel('Water level [cm]')
            plt3.set_title('Tank Levels Simulated')
166         plt3.grid()
            show()
168
    if __name__ == "__main__":
170     run()
```

I go to my model directory and use the JModelica supplied bash file (Python startup script) to initiates a Python session. I load the model Python script and run the model. Using MA27 the outputs of the model can be seen in Table 7.1.

Sparse linear eq. solver	Time Steps	# Iter.	CPU Time [sec]	Obj. value
MA27	300	98	22.569	5.3449e+03
MA27	100	168	13.909	5.3450e+03
MA27	30	72	1.818	5.3499e+03

Table 7.1: Iterations Statistics of JModelica

For more information on the output values see section 6 of the IPOPT Docu-

mentation.

The solution to the model, I just presented, is provided in Figure 7.1, 7.2, and 7.3.

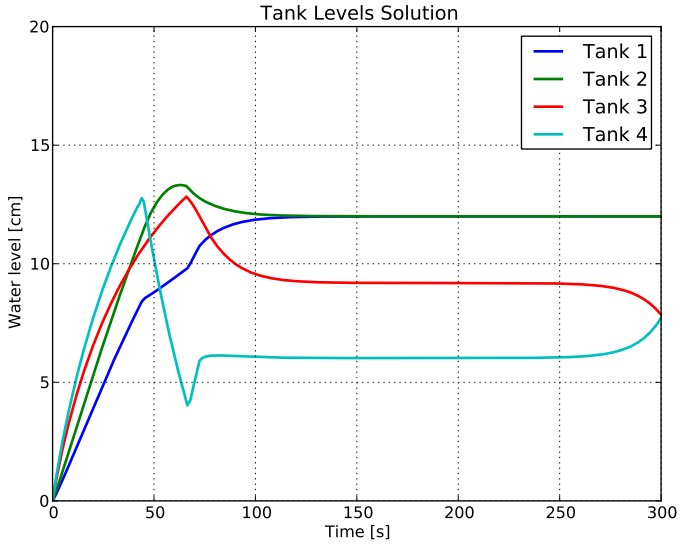


Figure 7.1: Height of liquid in the four tanks from the optimization

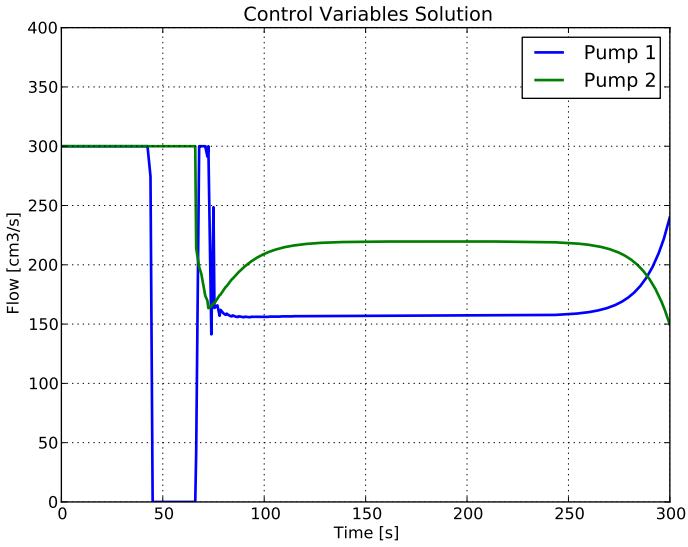


Figure 7.2: Control variables (pump flow) from the optimization

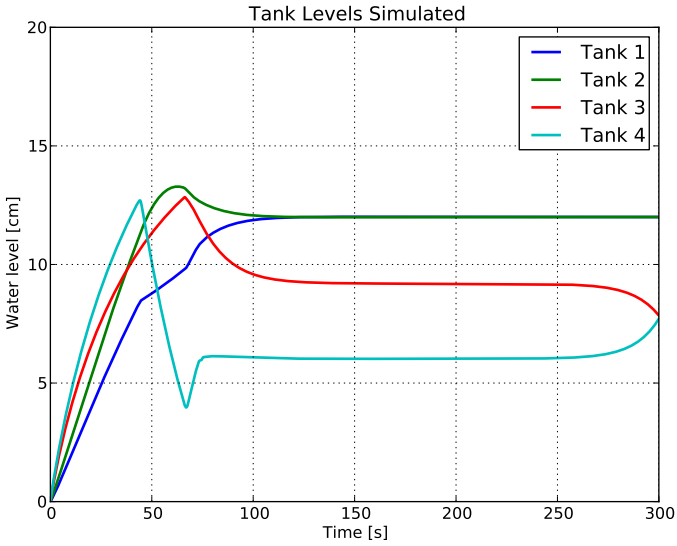


Figure 7.3: Height of liquid in the four tanks from the simulation

Chapter 8

A preconfigured OS for distribution

In this chapter, I will go through the creation of a preconfigured operating system, where all softwares mentioned in this thesis is preinstalled together with model libraries, all within the frames of an open-source alternative. The purpose of this is to provide a simple way of getting started with with the softwares described in the thesis. Installing most of the softwares from this thesis is very time demanding of can be technical challenging. A preconfigured OS, where all the softwares already are installed and ready for use, can therefore be an attractive alternative for users that wish to get started in a fast and less demanding way.

URLs: [OVF Info] http://www.vmware.com/appliances/getting-started/learn/ovf.html,

[Ubuntu Home] http://www.ubuntu.com/,

[Sun Hypervisor] http://www.virtualbox.org/

8.1 Building a virtual machine

A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine, meaning that the operating system on the virtual machine is completely isolated from the host machine. A virtual machine running a operating system is called a guest operating system.

In order to run a virtual machine, a hypervisor or virtualization software is required. This hypervisor runs on the host machine.

I have used VirtualBox as my hypervisor for building and running my virtual machine. VirtualBox is provided by Sun Microsystems and is compatible with many operating systems. VirtualBox is freely available as Open Source Software under the terms of the GNU General Public License (GPL).

As the operating system, I installed Ubuntu 9.10. Ubuntu is a complete Linux-based operating system containing free and open source software applications. Ubuntu is provided under the terms of the GNU General Public License (GPL). Ubuntu can be downloaded as an ISO file from http://www.ubuntu.com/getubuntu/download.

Using my hypervisor, I created a hard disk, defining the size and system resources allocated for the virtual machine. The hard disk was created as a dynamic disc image, meaning that it only allocates the space that it currently uses. My hard disk has a maximum capacity of 10 GB. This upper bound can not be adjusted. If more space is needed, a new hard disk has to be created. All other system resources can be adjusted continuously by the user. I then installed the operating system by adding the Ubuntu ISO image as a CD storage device, initiating the first virtual session, and following the guided installation procedure.

8.2 Distributing a virtual machine using OVF

In order for the virtual machine to be packaged and distributed in a practical way, I have chosen the Open Virtualization Format (OVF) format. OVF is an open packaging and distribution format for virtual machines. The Open Virtualization Format (OVF) standard provides a way for developers to create a single pre-packaged virtual appliance that can run on any users virtualization

platforms of choice.

8.3 Using the system

Start by opening your preferred hypervisor. Import the preconfigured operating system by choosing the option "import" and selecting the OVF file provided together with this thesis. The hypervisor will then unpack the hard disk and create a virtual machine with the name "OCP" (Optimal Control Problems). A system requirement is that the system should have 6 GB of free hard disk space. When the import is completed, the virtual machine can be turned on.

The user profile for the operating system is

- **User:** imm
- **Password:** dtu12345

8.3.1 Using ACADO Toolkit

The directory of ACADO Toolkit is located at

`/home/imm/Optimization/ACADOtoolkit`

and my model library can be found at

`/home/imm/Optimization/ACADOtoolkit/examples/MyACADO`

In order to run my implementation of the Four Tank System just open the terminal and prompt

```
> cd /home/imm/Optimization/ACADOtoolkit/examples/MyACADO
> ./tank_system_control
```

ACADO will then solve the problem and plot the results using Gnuplot.

8.3.2 Using IPOPT

The directory of IPOPT is located at

`/home/imm/Optimization/Ipopt`

and my model library can be found at

`/home/imm/Optimization/Ipopt/build/Ipopt/examples/MyIpopt`

In order to run my C++ implementation of the Four Tank System just open the terminal and prompt

```
> cd /home/imm/Optimization/Ipopt/build/Ipopt/examples/MyIpopt/CPP
> ./FourTankSystem
```

IPOPT will then solve the problem. The results can then be plotted using the Gnuplot script *fts.gp*, by prompting

```
> gnuplot fts.gp
```

8.3.3 Using CppAD

The directory of CppAD is located at

`/home/imm/Optimization/CppAD`

and my model library can be found at

`/home/imm/Optimization/CppAD/cppad_ipopt/MyCppAD`

I have written my very own *makefile* which is easy to re-use with your own models. The directory contain a hidden folder called `.deps`, where the portable object files of the models is contained.

In order to run my implementation of the Four Tank System just open the terminal and prompt

```
> cd /home/imm/Optimization/CppAD/cppad_ipopt/MyCppAD
> ./FourTankSystem
```

CppAD's IPOPT solver will then solve the problem. This particular implementation only have 30 time steps, since CppAD requires a lot of memory for running larger models and the virtual machine have very limited memory. The results can then be plotted using the Gnuplot script *fts.gp*, by prompting

```
> gnuplot fts.gp
```

The *makefile* is written in such a way that different commands will compile the model. The prompt command

```
> make test
```

will compile and run models at the same time, while the prompt command

```
> make FourTankSystem
```

will simply compile the model.

8.3.4 Using SUNDIALS

The directory of SUNDIALS is located at

`/home/imm/Optimization/Sundials-2.3.0`

and my model library can be found at

`/home/imm/Optimization/Sundials-2.3.0/build/examples/MySundials`

In order to run my implementation of the Four Tank System, as a differential equations system with fixed control variable values, just open the terminal and prompt

```
> cd /home/imm/Optimization/Sundials-2.3.0/build/examples/MySundials
> ./FourTankSystemDiff
```

SUNDIALS will then solve the differential equations system, while also finding roots of the objective function of the Four Tank System model.

8.3.5 Using JModelica

The directory of JModelica is located at

`/home/imm/Optimization/JModelica`

and my model library can be found at

`/home/imm/Optimization/JModelica/Python/src/jmodelica/examples/MyJModelica`

In order to run my implementation of the Four Tank System start by going to the model library directory. Then initiate a python session using the bash file (Python startup script) provided by JModelica. The Python or IPython startup script can be run either by specifying the complete path to the file and using the "`sh`" execute command, or by using the unix scripts I have created for this purpose. The two options looks as follows

```
> cd /home/imm/Optimization/JModelica/Python/src/jmodelica/examples/
  MyJModelica/FourTankSystem
> ../IPython
  >>> import FourTankSystem as fts
  >>> fts.run()
```

or

```
> cd /home/imm/Optimization/JModelica/Python/src/jmodelica/examples/
  MyJModelica/FourTankSystem
> sh /home/imm/Optimization/JModelica/build/Python/jm_ipython.sh
  >>> import FourTankSystem as fts
  >>> fts.run()
```

JModelica will then solve and thereafter simulate the problem. All results will also be plotted.

Chapter 9

Conclusion

The key goal of this thesis was to investigate, test and supply a user guide for using open source mathematical programming software for modeling and solving constrained dynamic optimization problems with fixed time horizon.

The choice of using open-source software, for solving constrained dynamic optimization problems based on general-purpose programming languages, was motivated in such a way that the advantages was clear compared to conventional commercial software. The branch of models investigated in the thesis was thoroughly presented and the relevant solution methods for solving them was introduced. This was done in such a way that the reader was able to connect the solution strategies, to the software packages presented in the thesis.

The Quadruple Tank Process was formulated as a mathematical model, with the aim of using it as a comparison test model for the software packages presented in the thesis. The Four Tank System represented a sufficiently complex model that was able to illustrate the capabilities and limitations of the softwares packages.

The software packages ACADO Toolkit, IPOPT, CppAD and JModelica was each thoroughly introduced. A detailed installation guide was given, supporting a range of operating platforms, which enabled the reader to follow these instruction and be able to use all software packages. A short example, on how to use the software and how to implement a model, was then given. This was

followed up be a thoroughly implementation of the Quadruple Tank Process. This implementation was then tested and the outputs from the software was presented and analyzed.

In comparison, all software packages solved the model to the same degree of satisfaction. ACADO Toolkit have a very intuitive and easy to use syntax, with many solver options. On the negative side, the user is not able to adjust the fineness of the discretization grid and it is not possible to add discontinuous constraints, as well as partial differential equation constraints. IPOPT is a very fast solver depending on the choice of sparse symmetric indefinite linear solver. IPOPT is able to handle all types of constraints and can be used together with a wide range of programming languages, as well as mathematical modeling languages. On the negative side, it is a very extensive task to implement models using general programming languages. A way of easing this task, is to use CppAD. The problem of using CppAD is though that what you gain in implementation simplicity is lost to computational time and memory requirements. Finally, JModelica also had a very intuitive way of formulating models. The extensive task of installing JModelica can though be a barrier for new users. In terms of solver options, JModelica provides a large range of tools.

Finally, a preconfigured operating system was offered to the reader, enabling the reader to test and run the models and implementations from the thesis without having to install the software packages. This was all offered within the framework of open-source licenses.

Appendix A

Appendix

A.1 IPOPT - GAMS

```
* Four_tank_system.gms
* Author:  Rune Brus
* Date: 07-02-2010
*
* This GAMS script includes a model, the "solve" command, and a
    call
* to gnuplot.  You can "run" all this by just typing
*
* $ gams Four_tank_system.gms

$eolcom //
option iterlim=999999999;  // avoid limit on iterations
option reslim=300;         // timelimit for solver in sec.
option optcr=0.0;          // gap tolerance
option solprint=ON;        // include solution print in .lst file
option limrow=100;         // limit number of rows in .lst file
option limcol=100;         // limit number of columns in .lst file
option nlp=Ipopt;

//-------------------------------------------------------

sets
        i       discretization  /0*10/
```

```
scalars
        N       Number of discretization intervals      /10/
        ao1     Cross sectional area of outlet 1        /1.2272/
        ao2     Cross sectional area of outlet 2        /1.2272/
        ao3     Cross sectional area of outlet 3        /1.2272/
        ao4     Cross sectional area of outlet 4        /1.2272/
        A1      Cross sectional area of inlet 1         /380.1327/
        A2      Cross sectional area of inlet 2         /380.1327/
        A3      Cross sectional area of inlet 3         /380.1327/
        A4      Cross sectional area of inlet 4         /380.1327/
        H1      Height of tank 1                        /20/
        H2      Height of tank 2                        /20/
        H3      Height of tank 3                        /20/
        H4      Height of tank 4                        /20/
        gamma1  Valve presure 1                         /0.15/
        gamma2  Valve presure 2                         /0.25/
        g       Acceleration of gravity                 /981/
        rho     Density of water                        /1.00/
        s       Approximation constant                  /0.1/
        r1      Optimization goal 1                     /12/
        r2      Optimization goal 2                     /12/
        tf      time horizont                           /300/
;

variables
        z       objective function
        F1(i)   Control Variable 1
        F2(i)   Control Variable 2
        m1(i)   Mass 1
        m2(i)   Mass 2
        m3(i)   Mass 3
        m4(i)   Mass 4
        h1t(i)  Water level tank 1
        h2t(i)  Water level tank 2
        h3t(i)  Water level tank 3
        h4t(i)  Water level tank 4
        q1i(i)  Inflow tank 1
        q2i(i)  Inflow tank 2
        q3i(i)  Inflow tank 3
        q4i(i)  Inflow tank 4
        q1(i)   Outflow tank 1
        q2(i)   Outflow tank 2
        q3(i)   Outflow tank 3
        q4(i)   Outflow tank 4
;

positive variables F1, F2, h1t, h2t, h3t, h4t;

equations
        Waterleveldelta         objective function
        Diff1(i)                Mass balance 1 at i
        Diff2(i)                Mass balance 2 at i
        Diff3(i)                Mass balance 3 at i
        Diff4(i)                Mass balance 4 at i
        Height1(i)              Water level 1 at i
```

```
         Height2(i)              Water level 2 at i
80       Height3(i)              Water level 3 at i
         Height4(i)              Water level 4 at i
82       Inflow1(i)              Inflow to tank 1 at i
         Inflow2(i)              Inflow to tank 2 at i
84       Inflow3(i)              Inflow to tank 3 at i
         Inflow4(i)              Inflow to tank 4 at i
86       Outflow1(i)             Outflow to tank 1 at i
         Outflow2(i)             Outflow to tank 2 at i
88       Outflow3(i)             Outflow to tank 3 at i
         Outflow4(i)             Outflow to tank 4 at i
90       h1init                  Boundary conditions for mass 1
         h2init                  Boundary conditions for mass 2
92       h3init                  Boundary conditions for mass 3
         h4init                  Boundary conditions for mass 4
94       F1init                  Boundary conditions for valve 1
         F2init                  Boundary conditions for valve 2
96       h1max(i)                Water stays in tank 1
         h2max(i)                Water stays in tank 2
98       h3max(i)                Water stays in tank 3
         h4max(i)                Water stays in tank 4
100      F1max(i)                Maximum presure in valve 1
         F2max(i)                Maximum presure in valve 2
102 ;

104 * Objective function
    Waterleveldelta          ..      z =e= (tf/N)*sum(i,(h1t(i)-r1)*(h1t
        (i)-r1) + (h2t(i)-r2)*(h2t(i)-r2));
106
    * Differential equations. Mass balances
108 Diff1(i)$(ord(i)>0)          ..  (m1(i)-m1(i-1))/(tf/N) =e= rho*q1i(
        i) + rho*q3(i) - rho*q1(i);

110 Diff2(i)$(ord(i)>0)          ..  (m2(i)-m2(i-1))/(tf/N) =e= rho*q2i(
        i) + rho*q4(i) - rho*q2(i);

112 Diff3(i)$(ord(i)>0)          ..  (m3(i)-m3(i-1))/(tf/N) =e= rho*q3i(
        i) - rho*q3(i);
    Diff4(i)$(ord(i)>0)          ..  (m4(i)-m4(i-1))/(tf/N) =e= rho*q4i(
        i) - rho*q4(i);
114
    * Water levels
116 Height1(i)                   ..  h1t(i) =e= m1(i)/(rho*A1);
    Height2(i)                   ..  h2t(i) =e= m2(i)/(rho*A2);
118 Height3(i)                   ..  h3t(i) =e= m3(i)/(rho*A3);
    Height4(i)                   ..  h4t(i) =e= m4(i)/(rho*A4);
120
    * Inflows
122 Inflow1(i)                   ..  q1i(i) =e= gamma1*F1(i);
    Inflow2(i)                   ..  q2i(i) =e= gamma1*F2(i);
124 Inflow3(i)                   ..  q3i(i) =e= (1-gamma2)*F2(i);
    Inflow4(i)                   ..  q4i(i) =e= (1-gamma1)*F1(i);
126
    Outflow1(i)                  ..  q1(i) =e= ao1*sqrt(2*g*h1t(i));
128 Outflow2(i)                  ..  q2(i) =e= ao2*sqrt(2*g*h2t(i));
```

```
Outflow3(i)                    ..  q3(i) =e= ao3*sqrt(2*g*h3t(i));
Outflow4(i)                    ..  q4(i) =e= ao4*sqrt(2*g*h4t(i));

* Boundary conditions for masses
h1init                         ..  h1t('0') =e= 0.0;
h2init                         ..  h2t('0') =e= 0.0;
h3init                         ..  h3t('0') =e= 0.0;
h4init                         ..  h4t('0') =e= 0.0;

* Boundary conditions for valves
F1init                         ..  F1('0') =e= 300;
F2init                         ..  F2('0') =e= 300;

* Water stays in tanks
h1max(i)                       ..  h1t(i) =l= H1;
h2max(i)                       ..  h2t(i) =l= H2;
h3max(i)                       ..  h3t(i) =l= H3;;
h4max(i)                       ..  h4t(i) =l= H4;

* Maximum presure of valves
F1max(i)                       ..  F1(i) =l= 300;
F2max(i)                       ..  F2(i) =l= 300;

model four_tank_system /all/;
solve four_tank_system using nlp minimizing z;

DISPLAY F1.L, F2.L, h1t.L, h2t.L, h3t.L, h4t.L;

FILE FrontierHandle /"OutputFM.txt"/;

FrontierHandle.pc = 7;
FrontierHandle.pw = 1048;

PUT FrontierHandle;

LOOP (i, PUT i.tl,h1t.L(i),h2t.L(i),h3t.L(i),h4t.L(i),F1.L(i),F2.L(
    i)/);

PUTCLOSE;
```

Bibliography

[1] Hart, William E.; Watson, Jean-Paul; Woodruf, David L.: *Python Optimization Modeling Objects (Pyomo)*, Conference paper (Proc INFORMS Computing Society), (2009)

[2] Binder, Thomas: *Adaptive multiscale methods for the solution of dynamic optimization problems*, VDI-Verl. (2002)

[3] J , John Bagterp: *Constrained Predictive Control - A Computational Approach*, Springer (2009)

[4] Nocedal, Jorge; Wright, Stephen J.: *Numerical Optimization*, Springer, Second Edition, 18 pp. 529-561 (2006)

[5] Eldèn, L.; Wittmeyer-Koch, L.; Nielsen, H. Bruun: *Introduction to Numerical Computation*, Studentlitteratur (2004)

[6] Course Material (*Numerical Optimal Control Algorithms and Applications in Renewable Energy Systems*), K.U. Leuven, (2009)

[7] Houska, Boris; Ferreau, Hans Joachim; Diehl, Moritzl: *ACADO Toolkit - An Open-Source Framework for Automatic Control and Dynamic Optimization*, Optimal Control Methods and Application, (2009)

[8] Logist, Filip; Houska, Boris; Diehl, Moritz; Impe, Jan van: *Fast Pareto set generation for nonlinear optimal control problems with multiple objectives*, Structural and Multidisciplinary Optimization, (2009)

[9] Ferreau, Hans Joachim; Houska, Boris; Diehl, Moritz: *Numerical Methods for Embedded Optimisation and their Implementation within the ACADO Toolkit*, In: 7th Conference - Computer Methods and Systems (CMS'09);

[10] Tadeusiewicz, R.; Ligeza, A.; Mitkowski, W.; Szymkat, M. (eds.): *Oprogramowanie Naukowo- Techniczne*, pp. 13-29 (2009)

[11] Nocedal, J.; Wächter, A.; Waltz, R. A.: *Adaptive Barrier Strategies for Nonlinear Interior Methods*, Research Report RC 23563, IBM T. J. Watson Research Center, Yorktown, USA (March 2005; revised January 2006)

[12] Schenk, O.; Wächter, A.; Hagemann M.: *Combinatorial Approaches to the Solution of Saddle-Point Problems in Large-Scale Parallel Interior-Point Optimization*, Research Report RC 23824, IBM T. J. Watson Research Center, Yorktown, USA (December 2005)

[13] Wächter, A; Biegler, L. T.: *On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming, Research Report RC 23149, IBM T. J. Watson Research Center, Yorktown, USA, 106(1) pp. 25-57 (2006)

[14] Wächter, A; Biegler, L. T.: *Line Search Filter Methods for Nonlinear Programming: Motivation and Global Convergence*, SIAM Journal on Optimization, Research Report RC 23036, IBM T. J. Watson Research Center, Yorktown, USA, 16(1) pp. 1-31(2005)

[15] Wächter, A, *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*, Phd Thesis, Carnegie Mellon University (2002)

[16] Armand, P.; Benoist, J.; Orban, D.: *Interpretation of Nonlinear Interior Methods as Damped Newton Methods*, Cahier du GERAD G-2005-80 (2005)

[17] Raghunathan, A. U.; Biegler, L. T.: *Interior point methods for Mathematical Programs with Complementarity Constraints (MPCCs)*, SIAM J. Optimization, 15(3) pp. 720-750 (2005)

[18] Åkesson, Johan; Årzèn, Karl-Erik; Gäfvert, Magnus; Bergdahl, Tove; Tummescheit, Hubertus: *Modeling and Optimization with Optimica and JModelica.org-Languages and Tools for Solving Large-Scale Dynamic Optimization Problem*, Journal article, Computers and Chemical Engineering, (2010)

[19] Åkesson, Johan: *Tools and Languages for Optimization of Large-Scale Systems*, PhD thesis, (2007)

[20] Åkesson, J.: *OptimicaÑAn Extension of Modelica Supporting Dynamic Optimization*, Conference paper, (2008)

[21] Åkesson, J.; Hedin, G.; Ekman, T.: *Development of a Modelica Compiler using JastAdd*, Journal article, (2008)

[22] Åkesson, J.; Gäfvert, M.; Tummescheit, T.: *JModelicaÑan Open Source Platform for Optimization of Modelica Models*, Conference paper, (2009)

[23] Åkesson, J.; Hedin, G.; Ekman, T.: *Implementation of a Modelica compiler using JastAdd attribute grammars*, Journal article, (2009)

[24] Åkesson, J.; Bergdahl, T.; Gäfvert, M.; Tummescheit, T.: *Modeling and Optimization with Modelica and Optimica Using the JModelica.org Open Source Platform*, Conference paper, (2009)

[25] Casella, F.; Donida, F.; Åkesson, J.: *An XML Representation of DAE Systems Obtained from Modelica Models*, Conference paper, (2009)

[26] Rantil, J.; Åkesson, J.; Fuhrer, C.; Gäfvert, M.: *Multiple-Shooting Optimization using the JModelica.org Platform*, Conference paper, (2009)

[27] Åkesson, J.: *Numerical Methods for Dynamic Optimization*, (2009)