# Multiple-Shooting Optimization using the JModelica.org Platform

Jens Rantil* Johan Åkesson[†‡] Claus Führer* Magnus Gäfvert[‡]

** Department of Numerical Analysis, Lund University

[†] Department of Automatic Control, Lund University

*[†] Lund University
Sölvegatan 18
SE-22100 Lund, Sweden
E-mail: claus@maths.lth.se

[‡] Modelon AB
Ideon Science Park
SE-22370 Lund, Sweden
E-mail: info@modelon.se

## Abstract

Dynamic optimization addresses the problem of finding the minimum of a cost function subject to a constraint comprised of a system of differential equations. There are many algorithms to numerically solve such optimization problems. One such algorithm is *multiple shooting*. This paper reports an implementation of a multiple shooting algorithm in Python. The implementation is based on the open source platform JModelica.org, the integrator SUNDIALS and the optimization algorithm `scipy_slsqp`. The JModelica.org platform supports model descriptions encoded in the Modelica language and optimization specifications expressed in the extension Optimica. The Modelica/Optimica combination provides simple means to express complex optimization problems in a compact and user-oriented manner. The JModelica.org platform in turn translates the high-level descriptions into efficient C code which can compiled and linked with Python. As a result, the numerical packages available for Python can be used to develop custom applications based on Modelica/Optimica specifications. An example is provided to illustrate the capabilities of the method.

*Keywords: optimization; optimal control; parameter optimization; Modelica; Optimica; JModelica.org*

## 1 Introduction

Dynamic optimization problems arise naturally in a wide range of applications and domains. Common examples are parameter optimization problems, design optimization, and optimal control. The key property of a dynamic optimization problem, which also makes such a problem hard to solve, is that it includes a constraint in form of a system of differential equations.

In this paper, we present an implementation of a particular numerical method for solving dynamic optimization problems, namely *multiple shooting*, [4, 20, 5]. In essence, a multiple shooting algorithm consists of an integrator for simulation of the system dynamics and evaluation of the cost function, and an optimization algorithm which tunes the optimization parameters of the problem. The Python language [17] was selected for implementation of the multiple shooting algorithm. Python has several advantages in the context of scientific computing, since there are several packages for high performance computing available, including Numpy [14] and Scipy [6]. Also, the package Matplotlib [9] provides methods for data visualization in a MATLAB-like manner. The main advantage is, however, that Python is a full-fledged high-level programming language offering strong support for generic concepts such as object-orientation and functional programming. Further more, Python has bindings to other languages, e.g, C and Fortran, which are common implementation languages for numerical algorithms. Accordingly, Python is commonly used as *glue language* in applications which integrate different algorithms. A common way to interface C code with Python is ctypes [7], which is based on loading of dynamically linked libraries. For the multiple shooting algorithm, Python is a suitable choice since it allows the majority of the computationally expensive subtasks to be delegated to precompiled C and Fortran codes.



DOI: 10.3384/ecp09430066

The implementation of the multiple shooting algorithm is based on the JModelica.org open source platform, [1]. JModelica.org offers support for dynamic models formulated in Modelica [21] and optimization specifications given in Optimica [2]. JModelica.org also has a Python-based execution API for the evaluation of the model equations, which has been used in this work. The purpose of this work is twofold. Firstly, solution of dynamic optimization problems by means of a multiple shooting algorithm adds to the functionality of the JModelica.org platform. Secondly, the algorithm provides an example of how different algorithms can be integrated with the JModelica.org model interface to create new algorithms.

# 2 Background

## 2.1 Modelica and Optimica

Modelica is a high-level language for encoding of complex heterogeneous physical systems, supporting object oriented concepts such as classes, components and inheritance. Also, text-book style declarative equations can be expressed as well as acausal component connections representing physical interfaces. While Modelica offers strong support for modeling of physical systems, the language lacks important constructs needed when formulating dynamic optimization problems, notably cost functions, constraints, and a mechanism to select inputs and parameters to optimize. In order to strengthen the optimization capabilities of Modelica, the Optimica extension has been proposed, [2]. Optimica adds to Modelica a small number of constructs, which enable the user to conveniently specify dynamic optimization problems based on Modelica models.

In the context of dynamic optimization, the use of high-level description formats is particularly attractive, since the interfaces of algorithms for solution of such programs are typically written in C or Fortran. Implementing the optimization formulation for such an algorithm may require a significant effort. In addition, once finalized, the implementation is typically difficult to reuse with another algorithm. The JModelica.org platform offers compilers for transforming Modelica/Optimica specifications into efficient C code which in turn may be interfaced with algorithms for dynamic optimization. The user may then focus on formulation of the actual problem at hand instead of attending to the details of encoding it to fit the requirements of a particular algorithm.

## 2.2 Dynamic optimization

Dynamic optimization problems may be formulated in many different ways, under different assumptions. In this paper, we assume that the problem is stated on the following form:

$$\begin{aligned} &\min_{p,u} \Phi(x(t_f;u,p),p) \\ &\text{subject to} \\ &\dot{x} = f(x,u,p) \end{aligned} \tag{1}$$

where $x$ is the system state, $u$ are the inputs and $p$ are free parameters in the optimization. The cost function is here assumed to be of Mayer type, that is, a function of the terminal state values and the parameters are minimized[1]. The influence of the control variable $u$ is implicit through $x$. The state $x$ is governed by an ordinary differential equation (ODE). Indeed, this formulation is somewhat limited. In particular, the dynamics of physical systems are often described by differential algebraic equations (DAEs). Also, inequality constraints representing, e.g., bounds on states and inputs are often present. However, the formulation given above is indeed general enough to demonstrate the concept of the multiple shooting algorithm.

There are three main classes of dynamic optimization problems, namely *parameter optimization*, *optimal control* and *parameter identification*.

In parameter optimization, $p$ is a vector containing a finite number of parameters. The goal is to find a parameter vector $p$ that minimizes $\Phi(x(t_f;p),p)$. $p$ can both contain model parameters and/or initial states of the model. An example of a parameter optimization problem would be to find the optimal wheel radius and tire thickness in a car to minimize the noise in the compartment.

In optimal control, the goal is to find a function $u(t)$ that minimizes $\Phi(x(t_f;u))$. This is usually done by discretization of $u$, for example by means of piecewise constant profiles or splines. Such an approximation transforms the original problem into a parameter optimization problem. An example of an optimal control problem would be to minimize the fuel consumption for a satellite moving from the moon to the earth.

Thirdly, in parameter identification, the objective is to fit a model to existing measurements. Typically, a perfect fit is usually not possible to obtain, due to the presence of measurement noise. Instead, the best

[1] Notice that a Lagrange type cost function, i.e., a function of the inputs, states and parameters integrated of the optimization interval can be cast into a Mayer cost function by introducing an additional state.

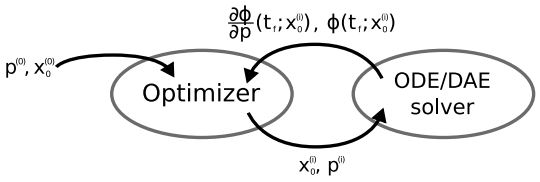


Figure 1: Shooting procedure.

fit according to some criteria penalizing the deviation between the model outputs and the measurements is sought.

## 2.3 Numerical methods for dynamic optimization

There are two main branches within the family of direct methods for dynamic optimization. *Sequential methods* rely on state of the art numerical integrators, typically also capable of computing state sensitivities, and on standard nonlinear programming (NLP) codes. The controls are then usually approximated by piece-wise polynomials (often piecewise constant functions), which render the controls to be parameterized by a finite number of parameters. These parameters are then optimized. *Simultaneous methods*, on the other hand, are based on *collocation*, and approximate *both* the state and the control variables by means of piece-wise polynomials, see [3] for an overview. This strategy requires a fine-grained discretization of the states, in order to approximate the dynamic constraint with sufficient accuracy. Accordingly, the NLPs resulting from application of simultaneous methods are very large, but also sparse. In order to solve large-scale problems, the structure of the problem needs to be explored.

### 2.3.1 Sequential shooting methods

In a sequential method, the control variables are parameterized by a finite number of parameters, for example by using a piece-wise polynomial approximation of $u$. Given fixed values of the parameters, the cost function of the optimization problem (1) can be evaluated simply by integrating the dynamic system. The parameters may then, in turn, be updated by an optimization algorithm, and the procedure is repeated, as illustrated in Figure 1. When the optimization algorithm terminates, the optimal parameter configuration is returned. Since the parameters determine the control profiles, which are then used to compute $x(t_f)$, the cost function can be written as $\Phi(x(t_f;u(p),p),p) = \Phi(p)$. The infinite dimensional optimization problem is thus transformed into a finite dimensional problem. For these reasons, sequential methods are also referred to as control parameterization methods. For a thorough description of single shooting algorithms, see [22].

Typically, the convergence of an optimization algorithm can be improved by providing it with gradients of the cost function with respect to the parameters. While finite differences is a simple method for obtaining gradients, it is not well suited for in this particular application due to scaling problems and limited accuracy, [19]. Taking the full derivative of the cost function $\Phi(x(t_f;u(p),p),p) = \Phi(p)$, we obtain

$$\left.\frac{d\Phi}{dp}\right|_{t_f} = \left.\frac{\partial\Phi}{\partial x}^T \frac{\partial x}{\partial p}\right|_{t_f}. \tag{2}$$

While $\frac{\partial\Phi}{\partial x}$ is usually straightforward to compute, the quantity $\frac{\partial x}{\partial p} = x_p$, referred to as the *state sensitivity* with respect to the parameter $p$, needs attention. A common approach for computing state sensitivities is derived by differentiating the differential equation $\dot{x} = f(x,u,p)$ with respect to $p$:

$$\frac{d}{dp}\frac{dx}{dt} = \frac{d}{dp}f(x,u,p) \tag{3}$$

$$\Rightarrow \quad \frac{d}{dt}\left(\frac{\partial x}{\partial p}\right) = \frac{\partial f}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p} \tag{4}$$

which gives the *sensitivity equations*

$$\dot{x}_p(t) = \frac{\partial f}{\partial x}x_p(t) + \frac{\partial f}{\partial u}\frac{\partial u}{\partial p} + \frac{\partial f}{\partial p}. \tag{5}$$

This suggests a method for computing derivatives by solving results a matrix valued differential equation. If the number of states of the system is $n_x$ and the number of parameters is $n_p$, then $n_x \times n_p$ additional equations must be integrated. This operation is computationally expensive, although the efficiency of the integration can be increased by exploring the structure of the sensitivity equations. There is also software available which supports integration of the sensitivity equations, for example DASPK, [13] and SUNDIALS [18].

### 2.3.2 Multiple shooting

An extension of the single shooting algorithm is *multiple shooting*. In a multiple shooting algorithm, the optimization interval $[t_0,t_f]$ is divided into a number of *subintervals* $[t_i,t_{i+1}]$, see Figure 2. New optimization variables corresponding to the initial conditions for the states in each subinterval, are then introduced. This

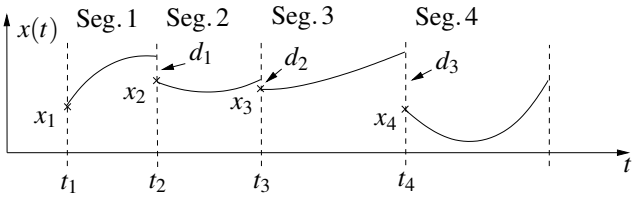


Figure 2: In a multiple shooting method, the control horizon is divided into a number of segments, which are integrated independently.

enables the dynamics, as well as the sensitivity matrices, to be computed *independently* in each segment. In order to enforce continuity of the state profiles, equality constraints are introduced in the optimization problem which ensure that the *defects*, $d_i = x(t_{i+1}^+) - x(t_{i+1}^-)$ are equal to zero.

In an optimization loop some of the intermediate control profiles $u$ and parameter values $p$ may get unphysical values. This can result in stability problems, when numerical integration has to be performed over longer time intervals. Also the computed sensitivity matrices may become unreliable. This is the advantage of multiple shooting algorithms compared to the single shooting algorithm. If the integration is performed over shorter intervals the numerical stability properties of the algorithm are improved. Another advantage of multiple shooting algorithms is that state inequality constraints can be more easily accommodated. Since the initial states in each segment are optimization variables, algebraic inequality constraints can be enforced for the states variables at the segment junctions. However, it has to be emphasized, that only the state variables at the segment junctions, $t_i$, can be restricted by inequality constraints.

### 2.4 The JModelica platform

JModelica.org is a novel Modelica-based open source project targeted at dynamic optimization [1]. JModelica.org features compilers supporting code generation of Modelica models to C, a C API for evaluating model equations and their derivatives and optimization algorithms. The compilers and the model C API have also been interfaced with Python in order to enable scripting and custom application development. In order to support formulation of dynamic optimization of Modelica models, JModelica.org supports the Optimica extension [2] of the Modelica language. Optimica offers constructs for encoding of cost functions, constraints, the optimization interval with fixed or free end points as well as the specification of the transcription scheme.

The C API providing functions for evaluating the model equations, cost function and constraints is entitled the JModelica.org Model Interface (JMI). The C code generated by the compiler front-end is compiled with a runtime library into a shared object file which in turn is loaded into Python, using the ctypes library. The JMI C functions can then be conveniently called from a Python shell or script. In addition, the input and return types of the C functions (typically pointers to vectors of *double* type) are mapped onto the types used by the Numpy package. This approach grants for a seamless integration between JMI and algorithms and data structures provided by Numpy and Scipy.

## 3 Implementation

As described in Section 2.3.2, a multiple shooting algorithm relies on a simulation algorithm, preferable capable of computing sensitivities, and a numerical optimization algorithm for algebraic optimization problems. The remaining part of the multiple shooting algorithm then consists of providing a non-linear program (NLP) to the optimization algorithm and to invoke the simulation algorithm in order to obtain function evaluations and derivative information. This part also includes representation of parameterized control signals, to keep track of optimization parameters, and to interface with the model execution API.

The simulation algorithm SUNDIALS [8, 18] was chosen for integration of the system dynamics. SUNDIALS is a high-quality integration package which is the latest evolution of a branch of ODE and DAE solvers including, e.g., DASSL. SUNDIALS contains a set of integration methods based on variable step size variable order multi-step methods using either the backward differentiation formula (BDF) or the more accurate Adams-Moulton formulae. BDF methods are well known for their good numerical stability properties for highly damped problems. In the context of this work, SUNDIALS has two major advantages. Firstly, it supports computation of sensitivity matrices. Secondly, there is a freely available Python interface for SUNDIALS; PySUNDIALS [16].

As for the optimization algorithm, the method `scipy_slsqp` was used, which is a Python wrap of the sequential quadratic programming algorithm [11]. This method is one of a variety of optimization algorithms interfaced by the Python package OpenOpt [12], which is a package that provides a unified interface to a large number of optimization algo-

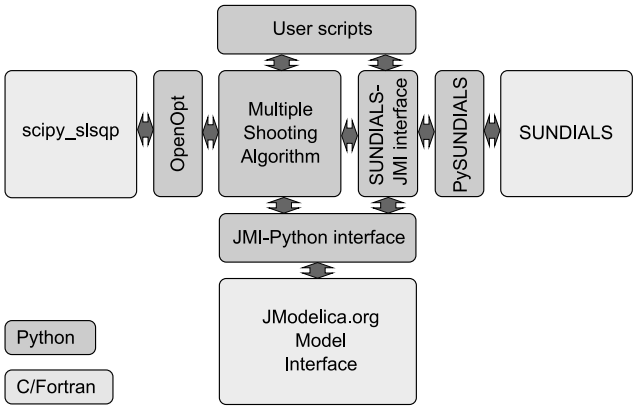


Figure 3: Architecture of the multiple shooting algorithm.

rithms.

The architecture of the algorithm is depicted in Figure 3. SUNDIALS is implemented in C, and interfaced to Python by the PySUNDIALS package. In order to provide convenient means to simulate models compiled with the JModelica.org compilers, an interface between PySUNDIALS and the Python wrappers of the JMI functions has been developed. This interface also supports computation of sensitivities, which is needed by the multiple shooting algorithm. Computation of sensitivities requires a slightly more complex setup than simulation, since in the former case, the sensitivity parameters need to be specified. SUNDIALS can make use of Jacobians provided by a model execution interface. As for simulation, the Jacobian of the right hand side of the ODE with respect to the states is required, and it is also standard for simulation oriented interfaces to provide such a function. In the case of sensitivity computations, however, the Jacobians with respect also to the inputs (in the case of an optimal control problem) and the parameters are required. These Jacobians are also available in the JMI interface. In the first implementation of the multiple shooting algorithm, Jacobians are not propagated to SUNDIALS. Rather, this is left for future improvements.

The main task of the multiple shooting algorithm is to provide call-back functions for evaluation of the cost function and constraints to the optimization algorithm. At this level, the optimization problem is a purely algebraic NLP; the dynamic part is handled by SUNDIALS and is in effect hidden from the optimizer.

OpenOpt provides standardized interfaces to different classes of optimization problems. Amongst them is a class which supports non-linear cost functions,

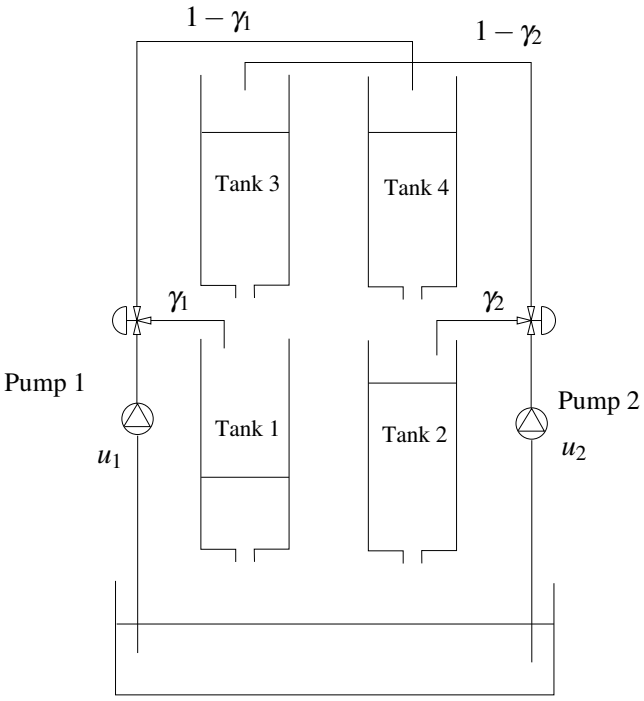


Figure 4: A schematic picture of the quadruple tank process

equality constraints, and bounded optimization variables. OpenOpt also interfaces a number of different solvers which support this class of problems. This approach makes it trivial to test different optimization algorithms with very minor changes to the code. In essence, the OpenOpt interface requires Python functions for evaluation of the cost function, the constraints and, if available, their derivatives. These functions, in turn, invoke integration and sensitivity computation by means of SUNDIALS. Also, functions in JMI are directly invoked, e.g., to evaluate the cost function. In effect, the multiple shooting algorithm provides an interface between the optimization algorithm on one hand and the simulation of the dynamic system and associated sensitivities on the other hand.

# 4 An Example

The quadruple-tank laboratory process, see Figure 4, has been used to demonstrate the multiple shooting algorithm. The model presented here is derived in [10]. The process consists of four tanks, organized in pairs (left and right), where water from the two upper tanks flows into the two lower tanks. A pump is used to pour water into the upper left tank and the lower right tank. A valve with fixed position is used to allocate pump capacity to the upper and lower tank respectively. A

 

Table 1: Parameter values of the Quadruple Tank

| Parameters | Values | Unit |
|---|---|---|
| $A_1, A_3$ | 2.8e-3 | [m$^2$] |
| $A_2, A_4$ | 3.2e-3 | [m$^2$] |
| $a_1, a_3$ | 7.1e-6 | [m$^2$] |
| $a_2, a_4$ | 5.7e-6 | [m$^2$] |
| $k_1, k_2$ | 3.14e-6, 3.29e-6 | [m$^3$/Vs] |
| $\gamma_1, \gamma_2$ | 0.7, 0.7 | |
| $g$ | 9.81 | [m/s$^2$] |

second pump is used to pour water into the upper right tank and lower left tank. The control variables are the pump voltages. Let the states of the system be defined by the water levels of the tanks (expressed in m) $x_1$, $x_2$, $x_3$ and $x_4$ respectively. The maximum level of each tank is 20 cm. The dynamics of the system is given by

$$\begin{aligned}
\dot{x}_1 &= -\frac{a_1}{A_1}\sqrt{2gx_1} + \frac{a_3}{A_1}\sqrt{2gx_3} + \frac{\gamma_1 k_1}{A_1}u_1 \\
\dot{x}_2 &= -\frac{a_2}{A_2}\sqrt{2gx_2} + \frac{a_4}{A_2}\sqrt{2gx_4} + \frac{\gamma_2 k_2}{A_2}u_2 \\
\dot{x}_3 &= -\frac{a_3}{A_3}\sqrt{2gx_3} + \frac{(1-\gamma_2)k_2}{A_3}u_2 \\
\dot{x}_4 &= -\frac{a_4}{A_4}\sqrt{2gx_4} + \frac{(1-\gamma_1)k_1}{A_4}u_1
\end{aligned} \tag{6}$$

where the $A_i$:s and the $a_i$:s represent the cross section area of the tanks and the holes respectively. The parameters $\gamma_i$:s determine the position of the valves which control the flow rate to the upper and lower tanks respectively. The control signals are given by the the $u_i$:s. Numerical values of the parameters are given in Table 1.

We consider two different stationary operation points corresponding to constant control inputs and where $\dot{x} = 0$. The first operating point, call it A, is defined by the control inputs $u_1^A = u_2^A = 2.0$, and the second, call it B, is defined by $u_1^B = u_2^B = 2.5$. The corresponding stationary state values are $x^A = (0.041, 0.066, 0.0039, 0.0056)$ and $x^B = (0.064, 0.10, 0.0062, 0.0087)$. Based on the operating points A and B, the following optimal control problem is defined:

$$\min_{u(t)} \int_0^{t_f} \alpha \sum_{i=1}^{4}(x_i(t) - x_i^B)^2 + \sum_{i=1}^{2}(u_i(t) - u_i^B)^2 dt \tag{7}$$

where $\alpha$ is a constant weight. Notice that this cost function is not on the form (1) and can therefore not be directly implemented. Instead, an additional state,

```
optimization QuadTank_Opt
  (objective = x_5(finalTime),
  startTime = 0, finalTime = 50)
  import SI = Modelica.SIunits;
  // Process parameters parameter
  SI.Area A1=2.8e-3, A2=3.2e-3,
          A3=2.8e-3, A4=3.2e-3;
  parameter SI.Area a1=7.1e-6, a2=5.7e-6,
                    a3=7.1e-6, a4=5.7e-6;
  parameter SI.Acceleration g=9.81;
  parameter Real k1_nmp(unit="m/s/V") =
                           3.14e-6,
               k2_nmp(unit="m/s/V") =
                           3.29e-6;
  parameter Real g1_nmp=0.70, g2_nmp=0.70;
  // Initial tank levels
  parameter SI.Length x1_0 = 0.04102638;
  parameter SI.Length x2_0 = 0.06607553;
  parameter SI.Length x3_0 = 0.00393984;
  parameter SI.Length x4_0 = 0.00556818;
  // Reference values
  parameter SI.Length x1_r = 0.06410371;
  parameter SI.Length x2_r = 0.10324302;
  parameter SI.Length x3_r = 0.006156;
  parameter SI.Length x4_r = 0.00870028;
  parameter SI.Voltage u1_r = 2.5;
  parameter SI.Voltage u2_r = 2.5;
  // Tank levels
  SI.Length x1(start=x1_0);
  SI.Length x2(start=x2_0);
  SI.Length x3(start=x3_0);
  SI.Length x4(start=x4_0);
  // Inputs
  input SI.Voltage u1(free=true);
  input SI.Voltage u2(free=true);
  // Cost function weight parameter
  Real alpha = 40000;
  Real x_5(start=0);
equation
  der(x1) = -a1/A1*sqrt(2*g*x1) +
            a3/A1*sqrt(2*g*x3)
            + g1_nmp*k1_nmp/A1*u1;
  der(x2) = -a2/A2*sqrt(2*g*x2) +
            a4/A2*sqrt(2*g*x4) +
            g2_nmp*k2_nmp/A2*u2;
  der(x3) = -a3/A3*sqrt(2*g*x3)
            + (1-g2_nmp)*k2_nmp/A3*u2;
  der(x4) = -a4/A4*sqrt(2*g*x4) +
            (1-g1_nmp)*k1_nmp/A4*u1;
  der(x_5) = alpha*((x1_r - x1)^2 +
                    (x2_r - x2)^2 +
                    (x3_r - x3)^2 +
                    (x4_r - x4)^2) +
                    (u1_r - u1)^2 +
                    (u2_r - u2)^2;
end QuadTank_Opt;
```

Listing 1: An Optimica specification of the quadruple tank optimization problem.

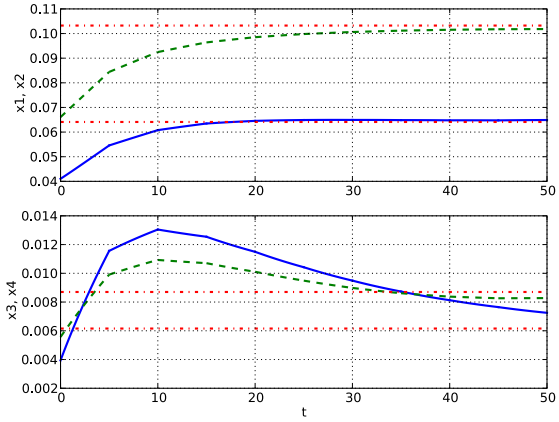


Figure 5: Optimal state profiles

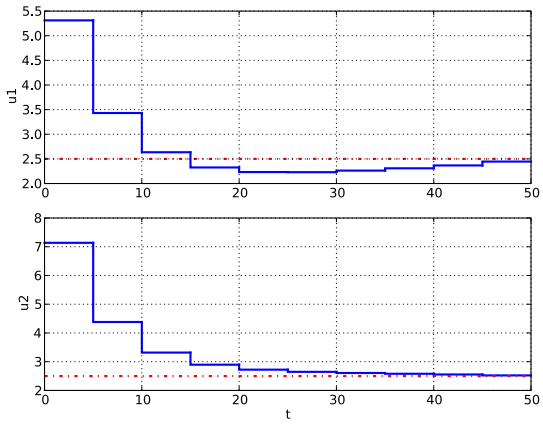


Figure 6: Optimal control profiles

$x_5$, is introduced. The additional state is governed by the differential equation

$$\dot{x}_5 = \alpha \sum_{i=1}^{4} (x_i(t) - x_i^B)^2 + \sum_{i=1}^{2} (u_i(t) - u_i^B)^2 \qquad (8)$$

The optimization criteria may now be written as

$$\min_{u(t)} x_5(t_f) \qquad (9)$$

The initial state values are assumed to be fixed and equal to $x^A$. Hence, the optimal control problem is defined as to transfer the state of the system from operating point A to operating point B. The Optimica specification for the optimal control problem is given in Listing 1.

The problem was solved using the multiple shooting algorithm, with the control signal parameterized to be constant over ten interval. Correspondingly, the number of intervals in the multiple shooting algorithm was set to ten. The optimization parameters, i.e., the control variable values in the ten elements and the initial state values in elements 2-9, were initialized in the following way. First, the control inputs were set to $u_1 = u_2 = 2.5$, and the dynamics was simulated over the optimization interval. The control variable values were then all set to 2.5 and the initial state values of each interval were initialized from the simulated state profiles.

The optimal state profiles are shown in Figure 5. The upper plot shows the levels in the lower tanks, where $x_1$ corresponds to the solid curve and $x_2$ corresponds to the dashed curve. The lower plot shows the levels in the upper tanks; $x_3$ in solid and $x_4$ in dashed. Also, the target values corresponding to operating point `B` are represented by the dash dotted lines. As can be seen, the state profiles approach the target values. The optimal control profiles are shown in 6, where $u_1$ is given in the upper plot and $u_2$ in the lower plot. As expected, the control signals approach the stationary values corresponding to operating point `B`.

## 5 Summary and future work

In this paper, an implementation of a multiple shooting algorithm has been presented. The implementation is done in Python and is based on the JModelica.org open source platform, the numerical integration package SUNDIALS and the optimization algorithm `scipy_slsqp`. It has been shown how the JModelica.org Python interface, providing access to functions for evaluation of the model equations, can be explored in order to develop custom algorithms in Python.

There are several improvements that would increase the applicability of the algorithm. The control variable parameterization is currently limited to one constant value per multiple shooting element. Implementing support for arbitrarily many elements in the parameterization of control variables as well as support for piecewise linear control profiles would be suitable extensions. The performance of the algorithm may be further improved by providing high accuracy Jacobians, available in JMI, to SUNDIALS. In addition, extending the sensitivity analysis of SUNDIALS to support discontinuities, see [15], would enable optimization of hybrid systems.

## References


[1] J. Åkesson, T. Bergdahl, M. Gäfvert, and H. Tummescheit. The JModelica.org Open Source Platform. In *7th International Modelica Conference 2009*. Modelica Association, 2009.

[2] Johan Åkesson. Optimica—an extension of modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[3] L.T. Biegler, A.M. Cervantes, and A Wachter. Advances in simultaneous strategies for dynamic optimization. *Chemical Engineering Science*, 57:575–593, 2002.

[4] H.G. Bock and K. J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Ninth IFAC world congress*, Budapest, 1984.

[5] R. Bulirch. Die Mehrzielmethode zur numerischen Lösung von nichtlinearen Randwertproblemen und Aufgaben der optimalen Steuerung. Technical report, Carl-Cranz-Gesellschaft, 1971.

[6] Inc. Enthought. SciPy, 2009. `http://www.scipy.org/`.

[7] Python Software Foundation. ctypes: A foreign function library for Python, 2009. `http://docs.python.org/library/ctypes.html`.

[8] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, 2005.

[9] J. Hunter, D. Dale, and M. Droettboom. matplotlib: python plotting, 2009. `http://matplotlib.sourceforge.net/`.

[10] Karl Henrik Johansson. *Relay Feedback and Multivariable Control*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, September 1997.

[11] Dieter Kraft. TOMP - Fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software*, 20(3):262–281, 1994.

[12] Dmitrey L. Kroshko. OpenOpt Home Page, 2009. `http://www.openopt.org/Welcome`.

[13] T. Maly and L. R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20(1-2):57–82, 1996.

[14] T. Oliphant. Numpy Home Page, 2009. `http://numpy.scipy.org/`.

[15] A. Pfeiffer. *Numerische Sensitivitätsanalyse unstetiger multidisziplinärer Modelle mit Anwendungen in der gradientenbasierten Optimierung (Numerical sensitivity analysis of discontinuous multidisciplinary models with applications in gradient based optimization)*. PhD thesis, Martin Luther University Halle-Wittenberg, 2008.

[16] Open Source Project. Pysundials. `http://pysundials.sourceforge.net`, May 2009.

[17] Open Source Project. Python programming language. `http://www.python.org`, May 2009.

[18] Open Source Project. Suite of nonlinear and differential/algebraic equation solvers (sundials). `http://www.llnl.gov/casc/sundials/`, May 2009.

[19] O. Rosen and R. Luus. Evaluation of gradients for piecewise constant optimal control. *Comput. chem. Engng.*, 15(4):273–281, 1991.

[20] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York and Berlin, 1980.

[21] The Modelica Association. The Modelica Association Home Page, 2007. `http://www.modelica.org`.

[22] V. Vassiliadis. *Computational solution of dynamic optimization problem with general differential-algebraic constraints*. PhD thesis, Imperial Collage, London, UK, 1993.


 