Derivative-free Parameter Optimization of Functional Mock-up Units

Sofia Gedda[a,c] Christian Andersson[a,c] Johan Åkesson[b,c] Stefan Diehl[a]

[a]Centre for Mathematical Sciences, Lund University, Sweden
[b]Department of Automatic Control, Lund University, Sweden
[c]Modelon AB, Sweden

Abstract

Representing a physical system with a mathematical model requires knowledge not only about the physical laws governing the dynamics but also about the parameter values of the system. The parameters can sometimes be measured or calculated, but some of them are often difficult or impossible to obtain directly. Never the less, finding accurate parameter values is crucial for the accuracy of the mathematical model.

Estimating the parameters using optimization algorithms which attempt to minimize the error between the response from the mathematical model and the real physical system is a common approach for improving the accuracy of the model.

Optimization algorithms usually require information about the derivatives which may not always be easily available or which may be difficult to compute due to, e.g., hybrid dynamics. In such cases, derivative-free optimization algorithms offer an alternative for design and parameter optimization.

In this paper, we present an implementation of derivative-free optimization algorithms for parameter estimation in the JModelica.org platform. The implementation allows the underlying dynamic system to be represented as a *Functional Mock-up Unit (FMU)*, and thus enables parameter optimization of models exported from modeling tools compliant with the *Functional Mock-up Interface (FMI)*.

Keywords: Derivative-free optimization; Parameter Estimation; JModelica.org; FMI; Assimulo

1 Introduction

Increasingly, industry rely on mathematical modeling for evaluating and designing new machines and devices. As the models grow increasingly complex, the need for estimating parameters which are unknown or uncertain is put into focus. Estimating unknown parameters in the mathematical model using optimization algorithms is a commonly used approach to increase the accuracy of models. In this paper, we focus on parameter estimation problems where the objective is to minimize the error between the simulated profiles of the mathematical model and measurements from the corresponding physical system. The objective function considered

$$f(x) = \sum_{i=0}^{M} (y^{\text{sim}}(t_i, x) - y^{\text{meas}}(t_i))^2 \tag{1}$$

where y^{sim} is the model output trajectory and y^{meas} are the measurements. The parameters to be estimated are $x \in \mathbb{R}^n$, where n is the number of parameters. M is the number of measurements at the time points t_i. The optimization problem is then formulated as

$$\min_{x \in \mathbb{R}^n} \quad f(x). \tag{2}$$

subject to the system dynamics, in the FMI case given by a hybrid Ordinary Differential Equation (ODE). Additionally, the parameters may be subject to bounds, $l \leq x \leq u$.

This optimization problem may be solved by transcribing the problem into a non-linear programming problem using either shooting methods [6] or collocation methods [6]. These methods, however, both use derivative information, which may be difficult or expensive to compute, e.g., in the case of hybrid systems. The idea is then to use algorithms which do not depend on derivative information, such as the Nelder-Mead simplex method [7]. In a derivative-free method, instead of using information from the derivatives to improve the solution, the objective is evaluated at a chosen set of points which are then used to improve

the solution. How the points are chosen and which strategy is used to improve the solution depends on the method. Typically, computation times are longer than for derivative-based methods, but on the other hand, derivative-free methods offer a feasible and robust option when other algorithms fail.

In this paper, we evaluate three derivative-free optimization algorithms for parameter estimation available in the JModelica.org platform: the Nelder-Mead simplex method, the differential evolution method and a genetic algorithm. Based on this evaluation, the Nelder-Mead algorithm seems most appropriate to solve the class of parameter optimization problems considered.

The main contribution of the paper is an implementation of the Nelder-Mead simplex algorithm. The algorithm supports parameter bounds and parallel evaluation of function evaluations where FMU models are loaded and simulated.

We also briefly present the underlying packages FMI Library (FMIL), PyFMI[1] and ASSIMULO[2]. These packages are part of JModelica.org, but also available stand-alone, and are used for simulating the model response. In Figure 1, an overview of the interaction between the packages in JModelica.org when solving a derivative-free optimization problem is shown.

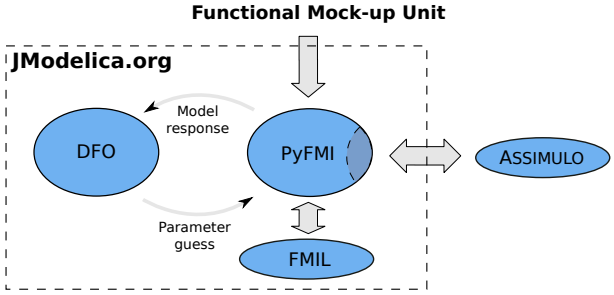

Figure 1: Overview of the interaction between the packages in JModelica.org when solving a derivative-free optimization problem.

The paper is outlined as follows. In Section 2, the *Functional Mock-up Interface* is presented together with an overview of optimization tools. In Section 3, an introduction to the JModelica.org platform is given together with the simulation package ASSIMULO as well as the Python package PyFMI for interaction with FMUs. Next, derivative-free optimization algorithms are introduced, followed by a description of the implementation in JModelica.org. In Section 6, the implementation is applied to two different problems where the second is a large industrial example where a model of an engine is calibrated. Finally, Section 7 concludes the paper with a summary and conclusions.

[1] `http://www.pyfmi.org`
[2] `http://www.assimulo.org`

2 Background

2.1 The Functional Mock-up Interface

The *Functional Mock-up Interface* [1] defines an open standard for model exchange. The intention is to allow exchange of models between different modeling and simulation tools. The standard describes models as hybrid ODEs with state, step and time events. A model that implements the FMI standard is called a *Functional Mock-up Unit* and is distributed as a compressed directory containing a shared object file or source code containing the model equations, and a set of functions for data access, and an XML file, which describes the model parameters and variables. The standard has received a significant amount of attention among vendors since the release in 2010 and currently there are 34 environments that support or plan to support the standard.

2.2 Optimization tools

There exist many tools for optimization of complex systems, both in the public domain and commercially available. Broadly, there are three different categories of optimization tools, although the scope is sometimes overlapping. In *Model integration tools* the problem of interfacing several design tools into a a single computation environment, where analysis, simulation and optimization can be performed is addressed. Examples include ModelCenter [23], OptiY [22], modeFRONTIER [12], and iSIGHT [10]. Such environments are capable of integrating several simulation and design tools into one computational chain, where the results are optimized. The integrated tools may be heterogeneous in the sense that they model different physical domains by means of different algorithms. Due to this heterogeneity amongst supported tools, optimization algorithm that does not exploit derivatives or model structure such as sparsity is commonly employed. Model integration tools typically also have strong support for model approximation and visualization.

Many *modeling and simulation tools* has optimization add-ons, e.g., Dymola [9], gPROMS [24], Jacobian [19], and OMOptim [18]. The level of support for optimization in this category differs between the

DOI 10.3384/ecp12076819

tools. Dymola, for example, offers add-ons for parameter identification and design optimization [11, 20]. gPROMS on the other hand, offers support for solution of optimal control problems and has the additional benefit in comparison with Modelica tools to provide support for partial differential equations (PDEs). Tools in this category usually support a set of derivative-based and derivative-free optimization algorithms. Optimization problems are typically formulated by means of graphical user interfaces.

In the third category there are *numerical packages* for dynamic optimization, often developed as part of research programs. Examples are ACADO [21], Muscod II [28], and DynoPC [17]. Such packages are typically focused on efficient implementation of an optimization algorithm for a particular class of dynamic systems. Also, detailed information about the model to optimize is generally required in order for such algorithms to work, including accurate derivatives and in some cases also sparsity patterns. While these packages offer state of the art algorithms, they typically come with simple or no user interface. Their usage is therefore limited due to the effort required to code the model and optimization descriptions. A notable example is CasADi [4], which provides an efficient AD kernel, interfaces to numerical optimization algorithms and a comprehensible Python interface for custom development of dynamic optimization algorithms. CasADi also support import of Modelica models in XML format, see [5].

The approach presented in this paper falls into the category of additions to modeling and simulation tools. Specifically, models exported from FMI compliant tools can be optimized. The presented algorithm uses Python scripting as a means to formulate optimization problems, and in this respect it differs from, e.g., the approach taken in Dymola.

3 JModelica.org

JModelica.org[3] [26] is a platform for modeling, simulation and optimization of complex physical systems primarily based on the Modelica[4] modeling language. JModelica.org is a community-based open-source project started at Lund University with the following aim:

> "To offer a community-based, free, open-source, accessible, user and application oriented Modelica environment for optimization and simulation of complex dynamic systems, built on well-recognized technology and supporting major platforms."

JModelica.org provides compilers for the Modelica language and the extension Optimica [25]. For simulations, the Python package ASSIMULO is used for both simulating ODEs and DAEs. Dynamic optimization is available using direct local collocation algorithms based on the DAE formulation of the model. The user interaction with JModelica.org is based on the programming language Python.

Included in JModelica.org are packages that can also be used stand-alone. In the following subsections, the packages FMI Library, PyFMI and ASSIMULO are presented.

3.1 FMI Library

FMI Library (FMIL) is a *C* package designed for working with FMUs and serving as support for applications interfacing the FMI. The package contains convenient methods for decompressing of FMUs, parsing XML information and connecting the binary[5]. The library supports FMI 1.0 for model exchange and for co-simulations and is intended for custom integration of FMI technology in applications. FMIL is also used as a basis of the Python package PyFMI.

3.2 PyFMI

PyFMI [2] is a package for interacting with FMUs using Python, based on the FMI Library. It provides convenient high-level functions for interacting with an FMU, retrieving values and accessing variable information from the XML information. Additionally, a low-level mapping of the functions specified in the interface can also be accessed. A model can be loaded and made available from Python using the following Python code:

```
#Import the model class
from pyfmi import FMUModel

#Load the model into Python
model = FMUModel("bouncingBall.fmu")
```

PyFMI also provides a connection to the simulation package ASSIMULO and thus enables access to state-of-the-art solvers such as CVode and IDA from the Sundials suite, capable of simulating hybrid systems.

[3] http://www.jmodelica.org
[4] http://www.modelica.org
[5] http://www.jmodelica.org/FMILibrary

A simulation is performed by using the `simulate` method.

```
#Simulate the model using Assimulo
res = model.simulate(final_time=10)
```

3.3 ASSIMULO

ASSIMULO [3] is a Python package for solving first or second order explicit ordinary differential equations (ODEs) or implicit ordinary differential equations (DAEs).

ASSIMULO combines a variety of different solvers written in FORTRAN, *C* and Python via a common high-level interface. The state-of-the-art solvers CVode and IDA from the SUNDIALS suite [15] as well as RADAU5 [14] are amongst the available solvers.

ASSIMULO is divided into two parts, namely problem definitions and solvers. A problem definition may in addition to the right-hand side of the differential equation also contain for instance the Jacobian as well as event functions in order to support simulation of hybrid systems. The idea is to separate information related to a problem from the solver. For instance, which states are algebraic is information that is related to the problem and not the solver. In Figure 2, an overview is given showing the available problem definitions and solvers in ASSIMULO. Also shown is the connection between the different problem formulations.

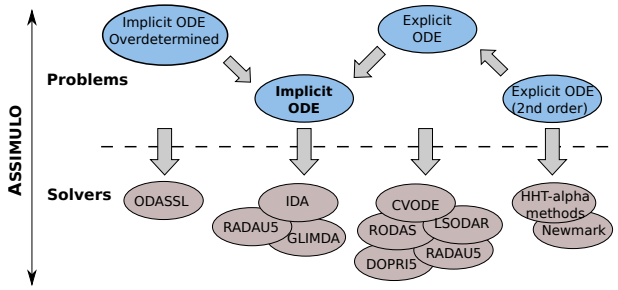

Figure 2: Connection between the different problem formulations and the different solvers available in ASSIMULO.

4 Derivative-free Optimization

In applications where derivatives are difficult or computationally expensive to obtain, there is a need for derivative-free optimization methods. Examples include very large models which also contains hybrid elements.

δ	Operation type
$-\frac{1}{2}$	inside contraction
$\frac{1}{2}$	outside contraction
1	reflection
2	expansion

Table 1: Different δ-values with corresponding operation types.

We shall now introduce three different derivative-free optimization algorithms which have been implemented or interfaced in the JModelica.org platform [13]: the Nelder-Mead simplex method, the differential evolution method and a genetic algorithm.

4.1 The Nelder-Mead simplex method

The Nelder-Mead simplex method has obtained its name from the fact that each iteration is based on a simplex. A simplex in $\mathbb{R}^n$ is a set of $n+1$ vertices $x_1,\ldots,x_{n+1} \in \mathbb{R}^n$ such that the vectors $x_i - x_1, i = 2,\ldots,n+1$ are linearly independent, i.e. it is a generalization of a triangle to arbitrary dimension.

In each iteration of the Nelder-Mead algorithm, the objective is to replace the vertex with the highest cost in the n-dimensional simplex with a better point. The vertices are ordered by increasing value of f such that $f(x_1) \leq \ldots \leq f(x_{n+1})$. The new point is searched for along the line through the vertex with the highest cost, x_{n+1}, and the centroid,

$$x_c = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{3}$$

of the remaining vertices $x_1,\ldots,x_n$. This line has the equation

$$x = x_c + \delta(x_c - x_{n+1}),\ \delta \in \mathbb{R}. \tag{4}$$

The parameter δ defines the type of the operation performed on the simplex. There are four different operation types that are performed by the algorithm: reflection, expansion, inside contraction or outside contraction, resulting in the reflection point, x_r, the expansion point, x_e, the inside contraction point, x_{ic}, or the outside contraction point, x_{oc} respectively. Table 4.1 displays the δ-values corresponding to these four operations. If none of these operations results in a better point than x_{n+1}, the simplex is shrunk toward the vertex with the lowest cost, x_1. That is, the n points with the highest costs are replaced by new points obtained

DOI 10.3384/ecp12076819

from

$$x = x_1 + \frac{1}{2}(x_i - x_1),\ i = 2,\ldots,n+1. \qquad (5)$$

This procedure is repeated until some termination criterion is fulfilled. There are usually three different termination criteria, one of which has to be fulfilled in order for the algorithm to terminate:

- Convergence criterion for x – the simplex is sufficiently small according to a user-provided tolerance.
- Convergence criterion for f – the function values at the simplex vertices are sufficiently close according to a user-provided tolerance.
- Termination criterion without convergence – the maximum number of iterations or function evaluations has been reached.

In Figure 3, two iterations of the algorithm are shown, illustrating how the simplex changes form and position.

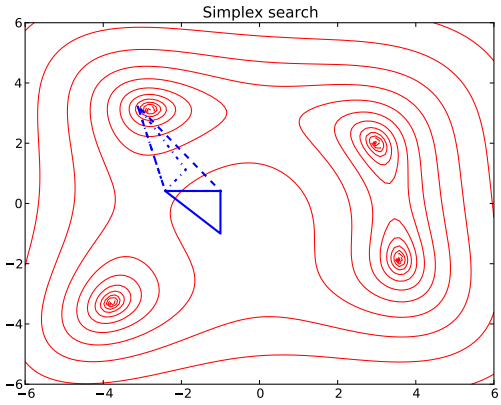

Figure 3: Two simplex iterations where the solid triangle is the initial simplex which transforms into the dashed triangle ($\delta = 2$) and then the dash-dot triangle ($\delta = -\frac{1}{2}$).

4.2 Evolutionary algorithms

The differential evolution method and genetic algorithms belong to the class of evolutionary algorithms, which consists of stochastic optimization algorithms inspired by the principles of biological evolution theory. In such algorithms, each candidate solution, $\bar{x} \in \mathbb{R}^n$, represents an individual and the objective function, $f(x)$, or fitness function, represents the environment within which the individuals live. The value $f(\bar{x})$ determines how fit the individual $\bar{x}$ is to survive in the environment; a lower value means a better fit. At each iteration, or generation, a new population of possible solutions is produced through mutation, crossover and selection. Mutation is a mechanism for maintaining genetic diversity by modifying an existing solution while crossover means combining two existing solutions into a new one.

4.2.1 The differential evolution method

The differential evolution method [27] works according to the following steps.

Initialization: An initial population of N individuals, or vectors, is generated randomly inside the feasible region.

Mutation: At each iteration, the population consists of N vectors, $x_i \in \mathbb{R}^n, i = 1,\ldots,N$. For each vector x_i, the target vector, a mutant vector, $v_i \in \mathbb{R}^n$, is produced by adding the weighted difference between two vectors in the current population to a third one according to the following formula:

$$v_i = x_{r_1} + F\left(x_{r_2} - x_{r_3}\right),$$

where $r_1, r_2, r_3 \in \{1,2,\ldots,N\}$ are random indices, distinct from each other and from i, and $F \in [0,2]$ is a constant.

Crossover: The mutant vector, v_i, is then recombined with its corresponding target vector, x_i, through a mixing of their elements, generating a trial vector, $u_i \in \mathbb{R}^n$. The trial vector receives elements of the mutant vector with probability $P \in [0,1]$ and elements of the target vector with probability $1 - P$.

Selection: The trial vector, u_i, is compared with the target vector, x_i, and the one giving the lowest value of the fitness function, f, is selected for the next generation.

The phases mutation, crossover and selection continue until a termination criterion is fulfilled.

4.2.2 Genetic algorithms

In genetic algorithms [16] the individuals are encoded as bit strings. There are various genetic algorithms which differ from one another but the following is a general description.

Initialization: An initial population of size N is generated randomly inside the feasible region.

Selection: In each generation, a selection probability, $p(x_i)$, is defined for each individual, $x_i \in \mathbb{R}^n$. The selection probability depends on the fitness function

value for the individual, $f(x_i)$, a smaller value gives a larger probability. Two individuals are then selected randomly according to their selection probabilities.

Crossover: Crossover is performed on the two selected individuals with a certain probability, the crossover rate. A common choice for this probability is around 0.7. There are different crossover techniques but a common approach is to randomly choose a position in the bit strings and swap all bits between the two strings after that position.

Mutation: Mutation is performed by flipping bits (from 0 to 1 or vice versa) at random positions in the bit strings. The probability of flipping a bit, the mutation rate, should be much lower than the crossover rate.

Selection, crossover and mutation is repeated until a termination criterion is reached.

5 Implementation

The algorithms evaluated in Section 4, have been made available in JModelica.org. The Nelder-Mead simplex algorithm has been implemented and is now provided as part of JModelica.org, while the differential evolution algorithm and a genetic algorithm has been interfaced through the OpenOpt package [6]. The algorithms are available through the Python function `fmin` in JModelica.org.

The method `fmin` requires as input the objective function together with the initial conditions as well as options for specifying the intended optimization algorithm and tolerances. In Section 6.1, it is shown how the objective function can be defined when the dynamic model is contained in an FMU.

In the Nelder-Mead algorithm, support for parallel evaluation of the objective function, $f(x)$, has been implemented. In each iteration of the algorithm, the evaluations of the $n+1$ vertices are distributed over a user-supplied number of processes, as well as the evaluations of the reflection, expansion and contraction points.

For further implementation details, see [13].

6 Examples

In [13], the different derivative-free algorithms was tested and the result indicated that the Nelder-Mead algorithm is the preferred algorithm for the tested parameter estimation problems. The evaluation was done based based on execution time and convergence to the optimal solution.

[6] http://openopt.org/

6.1 Furuta pendulum

The Furuta pendulum is a system consisting of a horizontal arm driven by a motor which is connected to a vertical pendulum, see Figure 4. The system has two degrees of freedom, namely the angle of the arm, ϕ, and the angle of the pendulum, θ. Additionally, there is friction in both the arm joint and the pendulum joint. Due to the discontinuities introduced by the friction, the system is not well suited for derivative-based optimization algorithms.

Figure 4: The Furuta pendulum.

The Furuta pendulum is modeled by a Modelica model, see Figure 5. The problem at hand is to calibrate the unknown friction coefficients of the arm and pendulum, respectively, against the given measurements using the Nelder-Mead simplex algorithm. The objective is thus

$$f(x) = \sum_{i=1}^{M} (\phi^{\text{sim}}(t_i,x) - \phi^{\text{meas}}(t_i))^2 + \sum_{i=1}^{M} (\theta^{\text{sim}}(t_i,x) - \theta^{\text{meas}}(t_i))^2 \quad (6)$$

where x is a vector containing the friction coefficients for the arm and the pendulum respectively.

The measurements were generated by simulation of the Modelica model for the Furuta pendulum and white measurement noise was added to the outputs. The measurements were given for a period of 40 seconds and were contained in a data file. The data was loaded into Python by the following code:

DOI 10.3384/ecp12076819

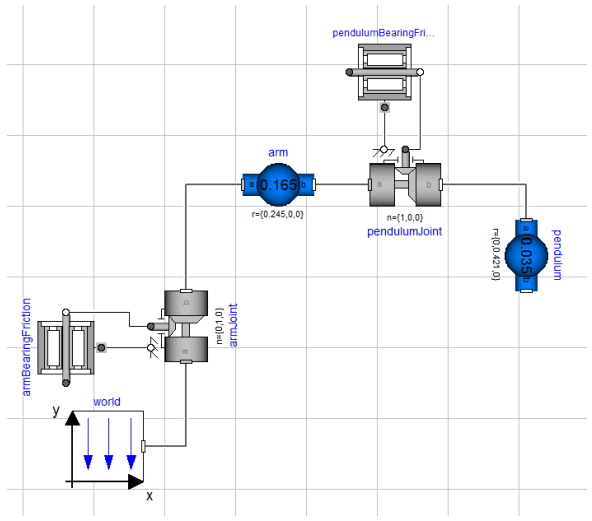

Figure 5: A Modelica model for the Furuta pendulum.

```
from scipy.io import loadmat
import numpy as N

# Load measurement data from file
data = loadmat('FurutaData')

# Extract data series
t_meas = data['time'][:,0]
phi_meas = data['phi'][:,0]
theta_meas = data['theta'][:,0]

y_meas = N.vstack((phi_meas,theta_meas))
```

The objective function is defined as a Python function where the FMU, generated by Dymola, for the Furuta pendulum is loaded and simulated for given parameter values.

```
from pyfmi import FMUModel
from pyjmi.optimization import dfo

# Define the objective function
def furuta_dfo_cost(x):
    #Scale down
    armFriction = x[0]/1e3
    pendFriction = x[1]/1e3

    # Load the FMU Model
    model = FMUModel('Furuta.fmu')

    # Set new parameter values
    model.set('armFriction',
              armFriction)
    model.set('pendulumFriction',
              pendFriction)

    # Simulate the model response
    res = model.simulate(final_time=40)

    # Load simulation result
    phi_sim =   res['armJoint.phi']
    theta_sim = res['pendulumJoint.phi']
    t_sim   =     res['time']

    # Evaluate the objective function
    y_sim = N.vstack((phi_sim,theta_sim))
    obj = dfo.quad_err(t_meas,y_meas,
                       t_sim,y_sim)

    return obj
```

Finally, the objective is provided to the optimization function `fmin` together with the initial guess and the parameter bounds. The initial guess, i.e., the nominal values, were obtained through manual testing. The object returned by `fmin` contains the optimized parameters together with statistics, such as the number of iterations performed:

```
# Specify initial conditions (scaled)
x0 = N.array([0.012,0.002])*1e3

# Lower and Upper bounds
lb = N.zeros(2)
ub = x0 + 10

# Solve using the Nelder-Mead algorithm
res = dfo.fmin(furuta_dfo_cost,
          xstart=x0,lb=lb,ub=ub,
          x_tol=1e-3,f_tol=1e-2)

# Optimal point rescaled
[armFriction_opt,pendFriction_opt] =
res[0]/1e3
```

The optimized parameter values were found to be 0.010 for the arm friction coefficient and 0.0010 for the pendulum friction coefficient. The result is visualized in Figure 6, where it can be seen that the model response is significantly more accurate using the optimized parameters as compared to the response given from the nominal parameters. In Figure 7, the error is shown between the measurements and the simulated response using both the nominal parameters and the optimized parameters.

6.2 Diesel Engine

In this example, parameters in a model of an exhaust gas pipe in a diesel engine is calibrated against measurements. The model was developed in Dymola using the Engine Dynamics Library and models a 13 liters Volvo truck engine [8]. The energy of the exhaust gas after the combustion is converted to torque, before releasing the gas to the purification process. In Figure 8, an overview of the model is shown. The energy is converted into torque by two turbines, shown as two trapezoids, where the first drives a compressor at the

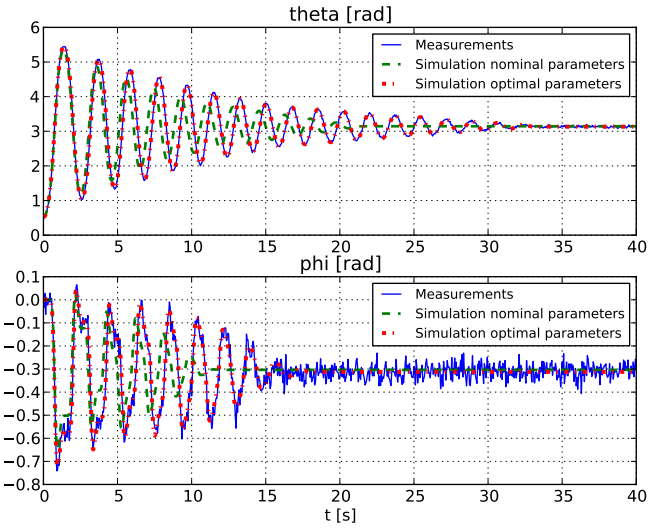

Figure 6: Simulation profiles corresponding to the optimized parameters (dashed-dotted), profiles resulting from simulation with nominal parameter values)dashed) and measurements (solid).

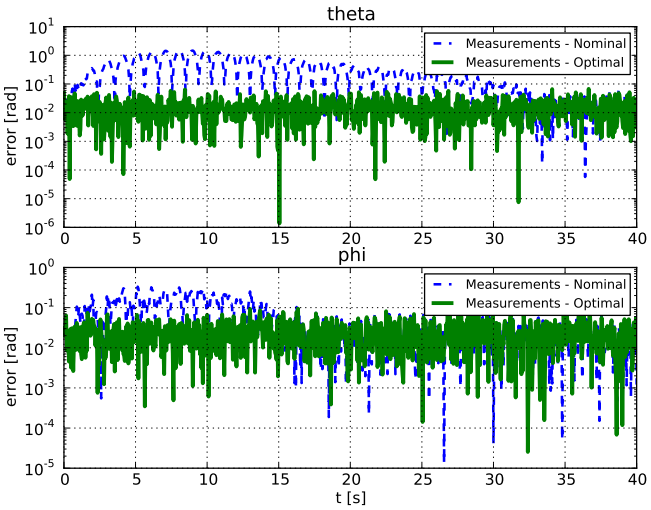

Figure 7: Error between the measurements and the simulated profiles using the nominal parameters (dashed) and the optimized parameters (dashed-dotted).

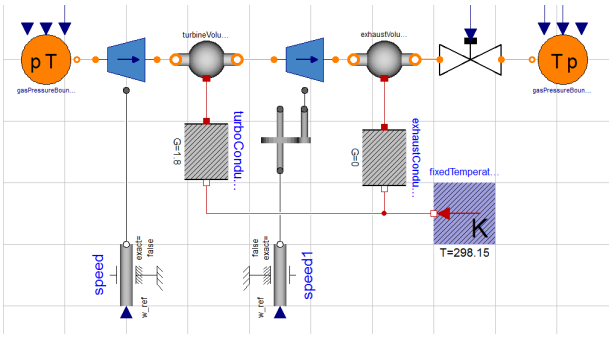

Figure 8: Overview of the model of the diesel engine.

air intake of the engine and the second is connected to the drive shaft. Additionally, there are two gas volumes which are connected to two thermal conductors that transport heat to the surrounding air. The endpoint circles represent the boundary conditions for the gas.

The uncertain parameters are the thermal capacities in the walls of the gas volumes together with the thermal conductance from gas to wall in the volumes.

The inputs of the model are the gas temperature and pressure entering the system, angular velocity of the turbines and the gas pressure exiting the system. The output is the gas temperature exiting the system.

Measurements are provided for the inputs and the output sampled every second over a thirty minute period. In Figure 9, the result is shown when simulating the model using nominal parameter values.

The problem is to minimize the error between the simulated gas temperature that exits the system and the measured temperature,

$$\min_{x\in\mathbb{R}^n} \quad \sum_{i=1}^{M} (T^{\text{sim}}(t_i,x) - T^{\text{meas}}(t_i))^2 \qquad (7)$$

$$\text{subject to} \quad x \geq 0 \qquad (8)$$

where M is the number of measurement points and n the number of parameters.

Instead of optimizing the four uncertain parameters simultaneously, the problem is divided into two problems. The first problem is to determine the thermal capacity and the thermal conductance in the right volume. The second is to determine the thermal capacity and the thermal conductance in the left volume, using the results from the first problem. This procedure is used since the parameters of the first and second volume are correlated. Optimizing all parameters simultaneously then results in over-parameterization.

For each optimization problem, the first third of the measurement data sequences are used for calibration and the remaining part is used for validation.

The model was exported from Dymola as an FMU and thereby made available to the DFO algorithms in JModelica.org. The two problems are then solved using the Nelder-Mead simplex algorithm. Figure 9 shows the resulting simulation response for the optimized parameters. In Figure 10, the corresponding error profiles are shown for the calibration and validation data sets respectively. As can be seen, the optimized parameters significantly increase the accuracy of the model. The (scaled) RMS error was decreased from 1.0 to 0.18 for the calibration data set and from 1.0 to

DOI 10.3384/ecp12076819

0.36 for the validation data set.

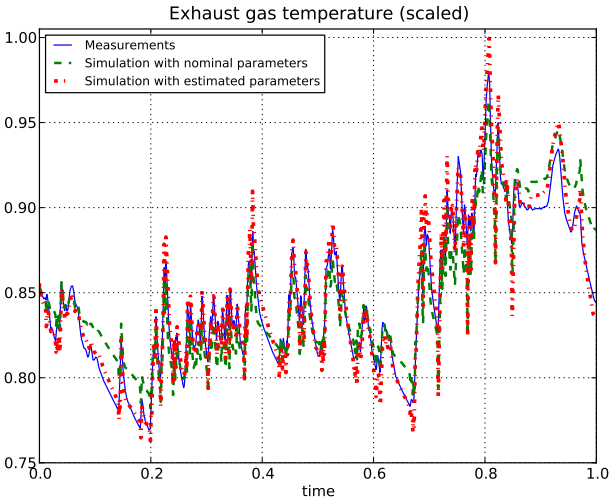

Figure 9: Simulation result with the optimized parameters together with result using the nominal parameter values and measurements.

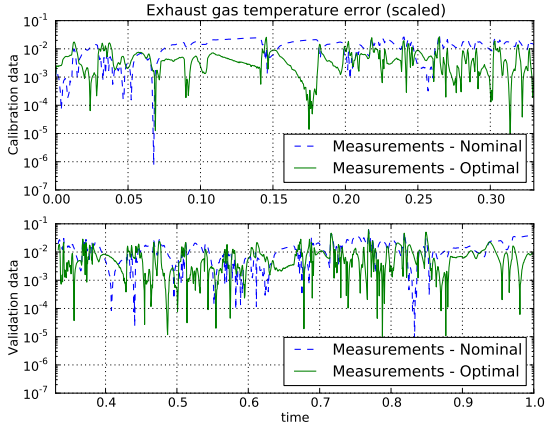

Figure 10: Error profiles for the calibration data set (top) and the validation data set (bottom).

7 Summary

An implementation of derivative-free optimization algorithms in JModelica.org has been presented. The implementation has been successfully applied to two dynamic models where the dynamics are contained in a *Functional Mock-up Unit*. In one of the examples, a Volvo truck engine was calibrated against measurement data, demonstrating the industrial applicability of the approach.

The Python-based user interface enables flexible implementation of complex cost functions involving, e.g., simulation of FMUs and comparison with measurement data or algorithmic evaluation of complex discontinuous costs.

8 Acknowledgments

The authors gratefully acknowledges financial support from Vinnova under the contract (Project number P35278-5) and from the Lund Center for Control of Complex Systems, LCCC, funded by the Swedish Research Council.

References

[1] Functional Mock-up Interface for Model Exchange. Interface specification, MODELISAR, January 2010.

[2] C. Andersson, J. Åkesson, C. Führer, and M. Gäfvert. Import and export of Functional Mock-up Units in JModelica.org. In *In 8th International Modelica Conference 2011*. Modelica Association, 2011.

[3] C. Andersson, J. Andreasson, C. Führer, and J. Åkesson. A workbench for multibody systems ODE and DAE solvers. In *The Second Joint International Conference on Multibody System Dynamics*, 2012.

[4] J. Andersson, J. Åkesson, and M. Diehl. CasADi—A symbolic package for automatic differentiation and optimal control. In S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors, *Proc. 6th International Conference on Automatic Differentiation*, Lecture Notes in Computational Science and Engineering. Springer, 2012.

[5] Joel Andersson, Johan Åkesson, Francesco Casella, and Moritz Diehl. Integration of casadi and jmodelica.org. In *8th International Modelica Conference*, March 2011.

[6] T. Binder, L. Blank, H.G. Bock, R. Bulirsch, W. Dahmen, M. Diehl, T. Kronseder, W Marquardt, J.P. Schlöder, and O. v. Stryk. *Online Optimization of Large Scale Systems*, chapter Introduction to model based optimization of chemical processes on moving horizons, pages 295–339. Springer-Verlag, Berlin Heidelberg, 2001.

[7] A.R. Conn, K. Scheinberg, and L.N. Vicente. *Introduction to Derivative-Free Optimization*. Mps-siam Series on Optimization. Society for Industrial and Applied Mathematics/Mathematical Programming Society, 2009.

[8] J. Dahl and D. Andersson. Gas exchange and exhaust condition modeling of a diesel engine using the Engine Dynamics Library. In *In 9th International Modelica Conference 2012*. Modelica Association, 2012.

[9] Dassault Systèmes. Dymola Home Page, 2012. `http://www.3ds.com/products/catia/portfolio/dymola`.

[10] Dassault Systèmes. iSIGHT Home Page, 2012. `http://www.3ds.com/products/simulia/portfolio/isight-simulia-execution-engine/overview/`.

[11] H. Elmqvist, H. Olsson, S.E. Mattsson, D. Brück, C. Schweiger, D. Joos, and M. Otter. Optimization for design and parameter estimation. In *In 4th International Modelica Conference 2005*. Modelica Association, 2005.

[12] ESTECO. modeFRONTIER Home Page, 2012. `http://www.esteco.com/`.

[13] Sofia Gedda. Calibration of Modelica models using derivative-free optimization. Master's thesis, Lund University, August 2011.

[14] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations: Stiff and differential-algebraic problems*. Springer series in computational mathematics. Springer-Verlag, 1993.

[15] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, September 2005.

[16] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[17] Y.D. Lang and L.T. Biegler. A software environment for simultaneous dynamic optimization. *Computers and Chemical Engineering*, 31(8):931–942, 2007.

[18] Linköping University. OMOptim Home Page, 2012. `https://openmodelica.org/index.php/developer/tools/176`.

[19] Numerica Technology. Jacobian, 2012. `http://www.numericatech.com/jacobian.htm`.

[20] H. Olsson, J. Eborn, S.E. Mattsson, and H. Elmqvist. Calibration of static models using Dymola. In *In 5th International Modelica Conference 2006*. Modelica Association, 2006.

[21] OPTEC K.U. Leuven. ACADO Home Page, 2012. `http://www.acadotoolkit.org/`.

[22] OptiY. OptiY Home Page, 2012. `http://www.optiy.de/`.

[23] Phoenix Integration. ModelCenter Home Page, 2012. `http://www.phoenix-int.com/software/phx_modelcenter.php`.

[24] Process Systems Enterprise. gPROMS Home Page, 2012. `http://www.psenterprise.com/gproms/index.html`.

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

[26] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010.

[27] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, December 1997.

[28] University of Heidelberg. MUSCOD-II Home Page, 2009. `http://www.iwr.uni-heidelberg.de/~agbock/RESEARCH/muscod.php`.

 DOI 10.3384/ecp12076819