Integration of CasADi and JModelica.org

Joel Andersson[c] Johan Åkesson[a,b] Francesco Casella[d] Moritz Diehl[c]
[a]Department of Automatic Control, Lund University, Sweden
[b]Modelon AB, Sweden
[c]Department of Electrical Engineering and Optimization in Engineering Center (OPTEC), K.U. Leuven, Belgium
[d]Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy

Abstract

This paper presents the integration of two open source softwares: CasADi, which is a framework for efficient evaluation of expressions and their derivatives, and the Modelica-based platform JModelica.org. The integration of the tools is based on an XML format for exchange of DAE models. The JModelica.org platform supports export of models in this XML format, wheras CasADi supports import of models expressed in this format. Furthermore, we have carried out comparisons with ACADO, which is a multiple shooting package for solving optimal control problems.

CasADi, in turn, has been interfaced with ACADO Toolkit, enabling users to define optimal control problems using Modelica and Optimica specifications, and use solve using direct multiple shooting. In addition, a collocation algorithm targeted at solving large-scale DAE constrained dynamic optimization problems has been implemented. This implementation explores CasADi's Python and IPOPT interfaces, which offer a convenient, yet highly efficient environment for development of optimization algorithms. The algorithms are evaluated using industrially relevant benchmark problems.

Keywords: Dynamic optimization, Symbolic manipulation, Modelica, JModelica.org, ACADO Toolkit, CasADi

1 Introduction

High-level modeling frameworks such as Modelica are becoming increasingly used in industrial applications. Existing modeling languages enable users to rapidly develop complex large-scale models. Traditionally, the main target for such models has been simulation, i.e., to define virtual experiments where a simulation software computes the model response.

During the last two decades, methods for large scale dynamic optimization problems have been developed. Notably, linear and non-linear model predictive control (MPC) have had a significant impact in the industrial community, in particular in the area of process control. In MPC, an optimal control problem is solved for a finite horizon, and the first optimal control interval is applied to the plant. At the next sample, the procedure is repeated, and the optimal control problem is solved again, based on updated state estimates. The advantages of MPC as compared to traditional control strategies are that it takes into account state and input constraints and that it handles systems with multiple inputs and multiple outputs. Also, MPC offers means to trade performance and robustness by tuning of a cost function, where the importance of different, often contradictory, control objectives are encoded. A bottleneck when applying MPC strategies, in particular in the case of non-linear systems, is that the computational effort needed to solve the optimal control problem in each sample is significant. Development of algorithms compatible with the real-time requirements of MPC has therefore been a strong theme in the research community, see e.g., [15, 40].

Driven by the impact of high-level modeling languages in the industrial community, there have been several efforts to integrate frameworks for such languages with algorithms for dynamic optimization. Examples include gPROMS [34], which supports dynamic optimization of process systems models and Dymola [13], which supports parameter and design optimization of Modelica models. Several other applications of dynamic optimization of Modelica models have been reported, e.g., [20, 35, 3, 33, 28].

This paper reports results of an effort where three different open source packages have been integrated: JModelica.org, [2], ACADO Toolkit, [26], and CasADi, [4]. The integration relies on the XML

model exchange format previously reported in [32]. Two main results are presented in the paper. Firstly, it is shown how CasADi, supporting import of the mentioned XML model format, has been used to integrate JModelica.org with ACADO Toolkit. Secondly, a novel direct collocation method has been developed based on the innovative symbolic manipulation features of CasADi. From a user's perspective, both CasADi and JModelica.org come with Python interfaces, which makes scripting, plotting and analysis of results straightforward.

The benefit of integrating additional algorithms for solution of dynamic optimization problems in the JModelica.org platform is that users may experiment with different algorithms and choose the one that is most suited for their particular problem. To some extent, the situation is similar to that of choosing an integrator for a simulation experiment, where it is well known that stiff systems require more sophisticated solvers than non-stiff systems.

The paper is organized as follows: in Section 2, background on dynamic optimization, Modelica and Optimica, ACADO and JModelica.org is given. Section 3 describes CasADi and recent extensions thereof. Section 4 reports a novel Python-based collocation implementation and in Section 5, benchmark results are presented. The paper ends with a summary and conclusions in Section 6.

2 Background

2.1 Dynamic optimization

Dynamic optimization is the solution of decision making problems constrained by differential or differential-algebraic equations. A common formulation is the optimal control problem (OCP) based on differential-algebraic equations (DAE) on the form

$$\begin{aligned}
&\min_{x,u,z,p} \int_0^T l(t,x(t),\dot{x}(t),z(t),u(t),p)\,dt \\
&\qquad + E(T,x(T),z(T),p) \\
&\text{subject to} \\
&f(t,x(t),\dot{x}(t),z(t),u(t),p) = 0 \quad t \in [0,T] \\
&h(t,x(t),\dot{x}(t),z(t),u(t),p) \leq 0 \quad t \in [0,T] \\
&x(0) = x_0 \\
&u_{\min} \leq u(t) \leq u_{\max} \quad t \in [0,T] \\
&p_{\min} \leq p \leq p_{\max}
\end{aligned} \tag{1}$$

where $x \in \mathbf{R}^{N_x}$ and $z \in \mathbf{R}^{N_z}$ denote differential and algebraic states respectively, $u \in \mathbf{R}^{N_u}$ are the free control signals and $p \in \mathbf{R}^{N_p}$ a set of free parameters in the model. The DAE is represented by the $N_x + N_z$ equations $f(t,x(t),\dot{x},z(t),u(t),p) = 0$, with the initial value for x explicitly given.

The objective function consists of an integral cost contribution (or Lagrange term) and an end time cost contribution (or Mayer term). The time horizon $[0,T]$ may or may not be fixed.

Numerical methods for solving this optimization problem emerged with the birth of the electronic computer in the 1950's and were typically based on either *dynamic programming*, which is limited to very small problems, or methods based on the calculus of variation, so-called *indirect methods*. The inability of indirect methods to deal with inequality constraints, represented above as the path constraint $h(t,x(t),\dot{x},z(t),u(t),p) \leq 0$ and the control bounds $u(\cdot) \in [u_{\min},u_{\max}]$, shifted the focus in the early 1980's to *direct-methods*, where instead the control, and (possibly) the state, trajectories are parametrized to form a finite-dimensional non-linear program (NLP), for which standard solution methods exist. In this work, we employ two of the most popular methods in this field, namely *direct multiple shooting* and *direct collocation*, see [8, 9] for an overview.

2.1.1 Direct multiple shooting

After parametrizing the control trajectories, for example by using a piecewise constant approximation, the time varying state trajectories can be eliminated by making use of standard ODE or DAE integrators. This method of embeddding DAE integrators in the NLP formulation is referred to as *single shooting*. The advantage is that it makes use of two standard problem formulations, the solution of initial value problems for DAEs and the solution of unstructured NLPs. For both of these problems, there exist several standard solvers, facilitating the implementation of the method. The drawback of single shooting is that the integrator call is a highly nonlinear operation, even if the differential equation is a linear one, making the NLP-optimizer prone to ending up in a local, rather than global, minimum and/or to slow convergence speeds To overcome this, Bock's *direct multiple shooting* method [10] includes in the optimization problem the differential state at a number of points, "shooting nodes", and the continuity of the state trajectories at these points is enforced by adding additional constraints to the NLP. These additional degrees of freedom can be used for suitable initialization of the state trajectories and often increase the radius of convergence at the cost of a

larger NLP.

The main difficulty of the method, and often the bottleneck in terms of solution times, is to efficiently and accurately calculate first and often second order derivatives of the DAE integrator, needed by the NLP solver. Implementations of the method include MUSCOD-II and the open-source ACADO Toolkit [26], used here.

2.1.2 Direct collocation

A common alternative to shooting methods is direct collocation on finite elements. Direct collocation is a simultaneous method, where both the differential and algebraic states as well as the controls are approximated by polynomials. But in contrast to multiple shooting, no integrator is used to compute the state and algebraic profiles. Rather, the original continuous time optimal control problem (1) is transcribed directly into an algebraic problem in the form of a non-linear program (NLP). This non-linear program is usually large, but is also typically very sparse. Algorithms, such as IPOPT, [39], exist that explores the sparsity structure of the resulting NLP in order to compute solutions of the problem in a fast and robust way.

The interpolation polynomials used to approximate the state profiles are usually chosen to be orthogonal Lagrange polynominals, and common choices for the collocation proints include Lobatto, Radau and Gauss schemes. In this paper, Radau collocation will be used, since this scheme has the advantage of featuring a collocation point at the end of each finite element, which makes encoding of continuity constraint for the states at element junction points straightforward.

The direct collocation method shares some characteristics with multiple shooting, since both are simultaneous methods. For example, unstable systems can be handled, and it is easy to incorporate state and control constraints. There are, however, also differences between multiple shooting and direct collocation. While a multiple shooting method typically requires computation of DAE sensitivities by means of integration of additional differential equations, direct collocation relies only on evaluation of first and (if analytic Hessian is used) second order derivatives of the DAE residual. For further discussion on the pros and cons of multiple shooting and direct collocation, see [9]

2.2 Modelica and Optimica

The Modelica language targets modeling of complex heterogeneous physical systems, [37]. Modelica permits specification of models in a wide range of physical domains, including mechanics, thermodynamics, electronics, chemistry and thermal systems. Also, recent versions of the language support modeling of embedded control systems and mapping of controller code to real-time control hardware. Modelica is object-oriented and equation-based, where the former property provides a means to construct modular and reusable models and the latter enables the user to state declarative equations. It is worth noticing that both differential and algebraic equations are supported and that there is no need, for the user, to solve the model equations for the derivatives, which is common in block-based modeling frameworks. In addition, Modelica supports acausal modeling, enabling explicit modeling of physical interfaces. This feature is the foundation of the component model in Modelica, where components can be connected to each other in connection diagrams.

Whereas Modelica offers state-of-the-art modeling of complex physical systems, it lacks constructs for expressing optimization problems. For simulation applications, this is not a problem, but when integrating Modelica models with optimization frameworks, it is inconvenient. In order to improve the support for formulation of optimization problems, the Optimica extension [1] has been proposed. Optimica adds to Modelica a small number of constructs for expressing cost functions, constraints, and what parameters and controls to optimize.

2.3 JModelica.org

JModelica.org is a Modelica-based open source platform targeting optimization simulation and analysis of complex systems, [2]. The platform offers compilers for Modelica and Optimica, a simulation package called Assimulo and a direct collocation algorithm for solving large-scale DAE-based dynamic optimization problems. The user interface in JModelica.org is based on Python, which provides means to conveniently develop complex scripts and applications. In particular, the packages Numpy [30], Scipy [16] and Matplotlib [27] enable the user to perform numerical computations interactively.

Recent developments of the JModelica.org platform includes import and export of Functional Mock-up Units (FMUs) and the integration with ACADO Toolkit and CasADi reported in this paper. The JModelica.org platform has been used in several industrial applications, including [28, 5, 35, 31, 23, 11]

The JModelica.org compilers generate C-code in-

tended for compilation and linking with numerical solvers. While this is a well established procedure for compiling Modelica models, it suffers from some drawbacks. Compiled code indeed offers very efficient evaluation of the model equations, but it also requires the user to regard the model as a black box. In contrast, there are many algorithms that can make efficient use of models expressed in symbolic form. Examples include tools for control design, optimization algorithms, and code generation, see [12] for a detailed treatment of this topic. In order to offer an alternative format for model export, JModelica.org supports the XML format described in [32]. This format is an extension of the XML scheme specified by the Functional Mock-up Interface (FMI) specification [29] and contains, apart from model meta data also the model equations. The equations are given in a format that is closely related to the expression trees that are common in compilers. The XML export functionality is explored in this paper to integrate the packages ACADO Toolkit and CasADi with the JModelica.org platform.

2.4 ACADO Toolkit

ACADO Toolkit [26] is an open-source tool for automatic control and dynamic optimization developed at the Center of Excellence on Optimization in Engineering (OPTEC) at the K.U. Leuven, Belgium. It implements among other things Bock's direct multiple shooting method [10], and is in particular designed to be used efficiently in a closed loop setting for nonlinear model predictive control (NMPC). For this aim, it uses the *real-time iteration scheme*, [14], and solves the NLP by a structure exploiting sequential quadratic programming method using the active-set quadratic programming (QP) solver qpOASES, [18].

Compared to other tools for dynamic optimization, the focus of ACADO Toolkit has been to develop a complete toolchain, from the DAE integration to the solution of optimal control problems in realtime. This vertical integration, together with its implementation in self-contained C++ code, allows for the tool to be efficiently deployed on embedded systems for solving optimization-based control and estimation problems.

3 CasADi

CasADi is a minimalistic computer algebra system implementing automatic differentiation, AD (see [22]) in forward and adjoint modes by means of a hybrid symbolic/numeric approach, [4]. It is designed to be a low-level tool for quick, yet highly efficient implementation of algorithms for numerical optimization, as illustrated in this paper, see Section 4. Of particular interest is dynamic optimization, using either a collocation approach, or a shooting-based approach using embedded ODE/DAE-integrators. In either case, CasADi relieves the user from the work of efficiently calculating the relevant derivative or ODE/DAE sensitivity information to an arbitrary degree, as needed by the NLP solver. This together with an interface to Python, see Section 3.1, drastically reduces the effort of implementing the methods compared to a pure C/C++/Fortran approach.

Whereas conventional AD tools are designed to be applied black-box to C or Fortran code, CasADi allows the user to build up symbolic representations of functions in the form of computational graphs, and then apply the automatic differentiation code to the graph directly. These graphs are allowed to be more general than those normally used in AD tools, including (sparse) matrix-valued operations, switches and integrator calls. To prevent that this added generality comes to the cost of lower numerical efficiency, CasADi also includes a second, more restricted graph formulation with only scalar, built-in unary and binary operations and no branches, similar to the ones found in conventional AD tools.

CasADi is an open source tool, written as a self-contained C++ code, relying only on the standard template library.

3.1 Python interface to CasADi

Whereas the C++ language is highly efficent for high performance calculations, and well suited for integration with numerical packages written in C or Fortran, it lacks the interactivity needed for rapid prototyping of new mathematical algorithms, or applications of an existing algorithm to a particular model. For this purpose, a scripting language such as Python, [36], or a numerical computing environment such as Matlab, is more suitable. We choose here to work with Python rather than Matlab due to its open source availabiliy and ease of interfacing with other programming languages.

Interfacing C++ with Python can be done in several ways. One way is to wrap the C++ classes in C functions and blend them into a Python-to-C compiler such as Cython. While this approach is simple enough for small C++ classes, it becomes prohibitively cumbersome for more complex classes. Fortunately, there exist excellent tools that are able to automate

this process, such as the Simplified Wrapper and Interface Generator (SWIG) [17, 6] and the Boost-Python package. We have chosen to work with SWIG due to SWIG's support for a large subset of C++ constructs, a large and active development community and the possibility to interface the code to a variety of languages in addition to Python, in particular JAVA and Octave.

The latest version of SWIG at the time of writing, version 2.0, maps C++ language constructs onto equivalent Python constructs. Examples of features that are supported in SWIG are polymorphism, exceptions handling, templates and function overloading.

By carefully designing the CasADi C++ source code, it was possible to automatically generate interface code for all the public classes of CasADi. Since the interface code is automatically generated, it is easy to maintain as the work on CasADi progresses with new features being added. Using CasADi from Python renders little or no speed penalty, since virtually all work-intensive calculations (numerical calculation, operations on the computational graphs etc.), take place in the built-in virtual machine. Functions formulated in Python are typically called only once, to build up the graph of the functions, thereafter the speed penalty is neglible.

3.2 The CasADi interfaces to numerical software

In addition to being a modeling environment and an efficient AD environment, CasADi offers interfaces to a set of numeric software packages, in particular:

- The sensitivity capable ODE and DAE integrators CVODES and IDAS from the Sundials suite [25]
- The large-scale, primal-dual interior point NLP solver IPOPT [39]
- The ACADO toolkit

In the Sundials case, CasADi automatically formulates the forward or adjoint sensitivity equations and provides Jacobian information with the appropriate sparsity needed by the linear solvers, normally an involved and error prone process. For IPOPT, the gradient of the objective function is generated via adjoint AD. Also, a sparse Jacobian of the NLP constraints as well as an exact sparse Hessian of the Lagrangian can be generated using AD by source code transformation. The ACADO Toolkit interface makes it possible to use the tool from Python and attach an arbitrary ODE/DAE integrator (currently CVODES, IDAS or fixed-step explicit integrators that have been implemented symbolically by the user) to ACADO.

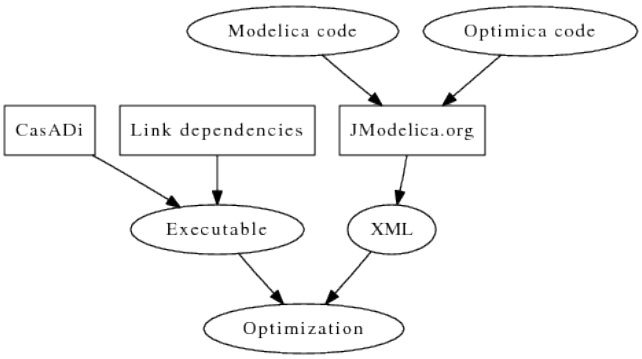

Figure 1: Optimization toolchain for JModelica/CasADi.

3.3 Complete tool chain

Though models for relatively simple dynamic systems can be efficiently formulated directly in CasADi, for more complex models it is beneficial to use a more expressive approach based on an object-oriented modelling language such as Modelica. To transmit model information about the dynamic system between Modelica and CasADi, we use the XML exchange format reported in [32], which is supported by JModelica.org. On the CasADi side, an XML interpreter based on the open source XML parser TinyXML, [38], is used to parse the generated XML code and build up the corresponding C++ data structures. The complete toolchain is presented in Figure 1.

This approach contrasts to the more conventional approach currently used in the current optimization framework of JModelica.org. This approach is based on C-code generation, which then needs to be compiled by a C compiler and linked with JModelica.org's runtime environment. See Figure 2.

The fact that the approach does not rely on a C-compiler in the optimization loop means that the program code can be compiled and linked once and for all for a particular system and then distributed as executables. It is also important to note that as models grow in size, the time needed to compile the code may be large.

3.4 A simple example

To demonstrate the tool, we show how to implement a simple, single shooting method for the Van der Pol oscillator used as a benchmark in section 5.1:

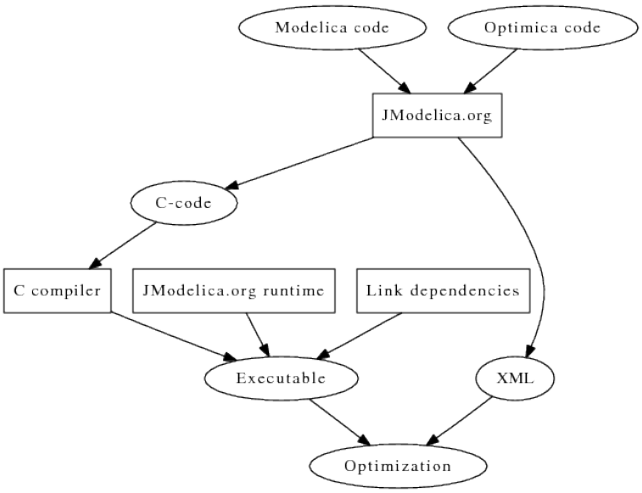

Figure 2: Optimization toolchain for JModelica.

```
from casadi import *

# Declare variables (use simple, efficient DAG)
t = SX("t") # time
x=SX("x"); y=SX("y"); u=SX("u"); L=SX("cost")

# ODE right hand side function
f = [(1 - y*y)*x - y + u, x, x*x + y*y + u*u]
rhs = SXFunction([[t],[x,y,L],[u]],[f])

# Create an integrator (CVODES)
I = CVodesIntegrator(rhs)
I.setOption("ad_order",1) # enable AD
I.setOption("abstol",1e-10) # abs. tolerance
I.setOption("reltol",1e-10) # rel. tolerance
I.setOption("steps_per_checkpoint",1000)
I.init()

# Number of control intervals
NU = 20

# All controls (use complex, general DAG)
U = MX("U",NU) # NU-by-1 matrix variable

# The initial state (x=0, y=1, L=0)
X  = MX([0,1,0])

# Time horizon
T0 = MX(0);  TF = MX(20.0/NU)

# State derivative and algebraic state
XP = MX(); Z = MX() # Not used

# Build up a graph of integrator calls
for k in range(NU):
  [X,XP,Z] = I.call([T0,TF,X,U[k],XP,Z])

# Objective function: L(T)
F = MXFunction([U],[X[2]])

# Terminal constraints: 0<=[x(T);y(T)]<=0
G = MXFunction([U],[X[0:2]])

solver = IpoptSolver(F,G)
solver.setOption("tol",1e-5)
solver.setOption("hessian_approximation", \
                 "limited-memory")
solver.setOption("max_iter",1000)
solver.init()

# Set bounds and initial guess
solver.setInput(NU*[-0.75], NLP_LBX)
solver.setInput(NU*[1.0],NLP_UBX)
solver.setInput(NU*[0.0],NLP_X_INIT)
solver.setInput([0,0],NLP_LBG)
solver.setInput([0,0],NLP_UBG)

# Solve the problem
solver.solve()
```

In CasADi, symbolic variables are instances of either the scalar expression class `SX`, or the more general matrix expression class `MX`.

```
x=SX("x"); y=SX("y"); u=SX("u"); L=SX("cost")
...
U = MX("U",NU) # NU-by-1 matrix variable
```

In the example above, we declare variables and formulate the right-hand-side of the integrator symbolically:

```
f = [(1 - y*y)*x - y + u, x, x*x + y*y + u*u]
```

Note that at the place where this is encountered in the script, neither `x`, `y` or `u` have taken a particular value. This representation of the ordinary differential equation is passed to the ODE integrator CVODES from the Sundials suite [25]. Since the ODE is in symbolic form, the integrator interface is able to derive any information it might need to be able to solve the initial value problem efficiently, relieving the user of a tedious and often error prone process. The information that can be automatically generated includes derivative information for sparse, dense or banded methods, as well as the formulation of the forward and adjoint sensitivity equations (required here since the integrator is being used in an optimal control setting).

The next interesting line is:

```
for k in range(NU):
  [X,XP,Z] = I.call([T0,TF,X,U[k],XP,Z])
```

Here, the `call` member function of the `CVodesIntegrator` instance is used to construct a graph with function calls to the integrator. Since the

matrix variable `U` is not known at this point, actually solving the IVP is not possible. This allows us to get a completely symbolic representation not only of the ODE, but of the nonlinear programming program. We then pass the NLP to the open source dual-primal interior point NLP solver IPOPT. Again, since the formulation is symbolic, the IPOPT interface will generate all the information it needs to solve the problem, including the gradient of the NLP objective function and the Jacobian of the NLP constraint function, both of which can be best calculated using automatic differentiation in adjoint mode for this particular example.

Note that the example above, with comments removed, consists of about 30 lines of code, which is a very compact way to implement the single shooting method. With only moderately more effort, other methods from the field of optimal control can be formulated including multiple-shooting and direct collocation, see Section 4. When executing the script above, it iterates to the the correct solution in 592 NLP iterations, which is considerably slower than the corresponding results for the simultaneous methods. Adapting the script to implement multple-shooting rather than single-shooting (the code of which is available in CasADi's example collection), decreases the number of NLP iterations to only 17.

4 A Python-based Collocation Algorithm

As described in Section 2.1, one strategy for solving large-scale dynamic optimization problems is direct collocation, where the dynamic DAE constraint is replaced by a discrete time approximation. The result is an non-linear program (NLP), which can be solved with standard algorithms. A particular challenge when implementing collocation algorithms is that the algorithms typically used to solve the resulting NLP require accurate derivative information and sparsity structures. In addition, second order derivatives can often improve robustness and convergence of such algorithms.

One option for implementing collocation algorithm is provided by optimization tools such as AMPL [19] and GAMS [21]. These tools support formulation of linear and non-linear programs and the user may specify collocation problems by encoding the model equations as well as the collocation constraints. The AMPL platform also provides a solver API, supporting evaluation of the cost function and the constraints, as well as first and second order derivatives, including sparsity information. A benefit for the user is that the tool internally computes these quantities using an automatic differentiation strategy that is very efficient, which in turn enables a solver algorithm to operate fast and reliably. On the other hand, AMPL, and similar systems, does not offer appropriate support for physical modeling. The description format is inherently flat, which makes construction of reusable models intractable.

Physical modeling systems, on the other hand, offer excellent support for modeling and model reuse, but typically offer only model execution interfaces that often do not provide all the necessary API functions. Typically, sparsity information and second order derivative information is lacking. The model execution interface in JModelica.org, entitled the JModelica.org Model Interface (JMI) overcomes some of these deficiencies by providing a DAE interface supporting sparse Jacobians, which in turn are computed using the CppAD package [7]. Based on JMI, a direct collocation algorithm has been implemented in C and the resulting NLP has been interfaced with the algorithm IPOPT [39]. While this approach has been successfully used in a number of industrially relevant applications, it also requires a significant effort in terms of implementation and maintenance. In many respects, implementation of collocation algorithms reduces to book keeping problems where indices of states, inputs and parameters need to be tracked in the global variable vector. Also, the sparsity structure of the DAE Jacobian needs to be mapped into the composite NLP resulting from collocation. In this respect, the approach taken in AMPL and GAMS has significant advantages.

In an effort to explore the strengths of the physical modeling framework JModelica.org and the convenience and efficiency in evaluation of derivatives offered by CasADi, a direct collocation algorithm similar to the one existing in JModelica.org has been implemented. The implementation is done completely in Python relying on CasADi's model import feature and its Python interface. As compared to the approach taken with AMPL, the user is relieved from the burden of implementing the collocation algorithm itself. In this respect the new implementation does not differ from the current implementation in JModelica.org, but instead, the effort needed to implement the algorithm is significantly reduced. Also, advanced users may easily tailor the collocation algorithm to their specific needs.

The implementation used for the benchmarks presented in this paper is a third order Radau scheme,

which is also supported by the C collocation implementation in JModelica.org.

5 Benchmarks

Three different optimal control benchmark problems with different properties have been selected for comparison of the different algorithms: the Van der Pol oscillator, a Continuously Stirred Tank Reactor (CSTR) with an exothermic reaction, and a combined cycle power plant. The first bencmark, the Van der Pol oscillator, is a system commonly studied in non-linear control courses, and demonstrates the ability of all methods evaluated to solve optimal control problems. The CSTR problem features highly non-linear dynamics in combination with a state contstraint. The final benchmark, the combined cycle power plant, is of larger scale, consisting of nine states and more than 100 algebraics.

Whereas the Van der Pol problem has been successfully solved using both multiple shooting and collocation, a solution to the CSTR and combined cycle problem has been obtained only using a collocation approach.

For reference, the original collocation implementation, written in C, is included in the benchmarks. This algorithm is referred to as JM collocation.

All the calculations have been performed on an Dell Latitude E6400 laptop with an Intel Core Duo processor of 2.4 GHz, 4 GB of RAM, 3072 KB of L2 Cache and 128 kB if L1 cache, running Linux.

In the benchmarks where IPOPT is used, the algorithm is compiled with the linear solver MA57 from the HSL suite.

5.1 Optimal control of the Van der Pol Oscillator

As a first example, consider the Van der Pol oscillator, described by the differential equations

$$\begin{aligned} \dot{x_1} &= x_2, \quad x_1(0) = 1 \\ \dot{x_2} &= (1 - x_1^2)x_2 - x_1 \quad x_2(0) = 0. \end{aligned} \tag{2}$$

The optimization problem is formulated as to minimize the following cost

$$\min_u \int_0^{20} x_1^2 + x_2^2 + u^2 dt \tag{3}$$

subject to the constraint

$$u \leq 0.75. \tag{4}$$

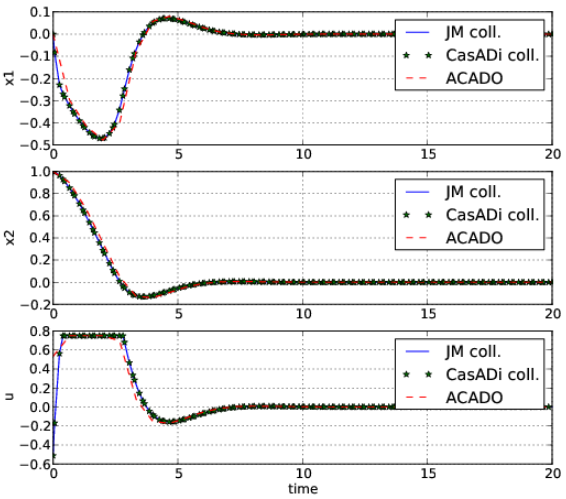

Figure 3: Optimization results for the Van der Pol oscillator.

First, we solve the optimal control problem using three different algorithms, all based on the Modelica model and the Optimica specification encoding the optimal control problem. 20 uniformly distributed control segements were used in all cases. The first method used to solve the problem is JModelica's native, C-based implementation of direct collocation, which relies on code generation and the AD-tool CppAD to generate derivatives. Secondly, the novel, CasADi and Python-based implementation of direct collocation presented in Section 4 is applied. The third algorithm is ACADO Toolkit's implementation of multiple shooting, using CasADi's interface to evaluate functions and and directional derivatives. Forward mode AD was used to generate DAE sensitivities and a BFGS approximation of the Hessian of the Lagrangian of the NLP. The optimal solutions for the three different algorithms are shown in Figure 3.

Table 1 shows the number of NLP iterations and total CPU time for the optimization (in seconds) for 5 different algorithmic approaches: direct collocation implemented in C and in Python using CasADi, ACADO Toolkit via the CasADi interface, and implementations of single and multiple shooting based on CasADi's Python interface. The implementations of single and multiple shooting are included to demonstrate usage of CasADi's integrator and NLP solver implementation, rather than to achive high performance. For the single shooting code, the complete script was presented in Section 3.4.

The last column of the table contains the share of the

total time spent in the NLP solver, the rest is mostly spent in the DAE functions (DAE residual, Jacobian of the constraints, objective function etc.) which are interfaced to the NLP solver. This information is not available for ACADO Toolkit.

Table 1: Execution times for the Van der Pol benchmark.

Tool	NLP iterations	Total time [s]	Time in NLP solver [s]
JModelica.org coll.	21	0.32	0.14
CasADi collocation	103	0.97	0.92
ACADO via CasADi	28	3.45	n/a
CasADi single shooting	616	167.7	1.22
CasADi mult. shooting	16	59.1	0.20

We can see that JModelica's current C-based implementation of direct collocation and ACADO's multiple shooting implementation, both of them using an inexact Hessian approximation with BFGS updating, show a similar number of NLP iterations. In terms of speed, the JModelica.org implementation clearly outperforms ACADO which is at least partly explained by the fact that JModelica.org involves a code generation step, significantly reducing the function overhead in the NLP solver. Also, CasADi involves a code generation step, but not to C-code which is then compiled, but to a virtual machine implemented inside CasADi. Clearly, this virtual machine is able to compete with the C implementation in terms of efficiently during evaluation and is also fast in the translation phase, as no C-code needs to be generated and compiled. Also note that for this small problem size, the function overhead associated with calling the DAE right hand side is major for all of the shooting methods. A more fair comparison here would involve the use of ACADO Toolkit's own symbolic syntax coupled with a code generation step in ACADO.

In this particular example, IPOPT, using CasADi to generate the exact Hessian of the Lagrangian, require more NLP steps than ACADO and JModelica.org's native implementation of collocation which are both using an inexact Hessian approximation. Despite the fact that the CasADi-based implementation does not rely on C code generation, and although it is calculating the exact Hessian, the time it takes to evaluate the functions only constitute only a small fraction (around 5%) of the total execution time. In contrast, in the C-based implmentation, the time spent in DAE functions, make up more than half of the total CPU time. This result cleary demonstrates one of the main strengths of CasADi, namely computational efficiency.

Looking at the number of iterations required in the single shooting algorithm, the superior convergence speed of simultaneous methods (collocation and multiple shooting) is obvious.

5.2 Optimal control of a CSTR reactor

We consider the Hicks-Ray Continuously Stirred Tank Reactor (CSTR) containing an exothermic reaction, [24]. The states of the system are the reactor temperature T and the reactant concentration c. The reactant inflow rate, F_0, concentration, c_0, and temperature, T_0, are assumed to be constant. The input of the system is the cooling flow temperature T_c. The dynamics of the system is then given by:

$$\begin{aligned} \dot{c}(t) &= F_0(c_0 - c(t))/V - k_0 e^{-EdivR/T(t)} c(t) \\ \dot{T}(t) &= F_0(T_0 - T(t))/V - \\ &\quad dH/(\rho C_p) k_0 e^{-EdivR/T(t)} c(t) + \\ &\quad 2U/(r\rho C_p)(T_c(t) - T(t)) \end{aligned} \tag{5}$$

where r, k_0, $EdivR$, U, ρ, C_p, dH, and V are physical parameters.

Based on the CSTR model, the following dynamic optimization problem is formulated:

$$\begin{aligned} \min_{T_c(t)} \int_0^{t_f} &(c^{\text{ref}} - c(t))^2 + (T^{\text{ref}} - T(t))^2 + \\ &(T_c^{\text{ref}} - T_c(t))^2 dt \end{aligned} \tag{6}$$

subject to the dynamics (5). The cost function corresponds to a load change of the system and penalizes deviations from a desired operating point given by target values c^{ref}, T^{ref} and T_c^{ref} for c, T and T_c respectively. Stationary operating conditions were computed based on constant cooling temperatures $T_c = 250$ (initial conditions) and $T_c = 280$ (reference point).

In order to avoid too high temperatures during the ignition phase of the reactor, the following temperature bound was enforced:

$$T(t) \leq 350. \tag{7}$$

The optimal trajectories for the three different algorithms are shown in Figure 4, where we have used a control discretization of 100 elements and a 3rd order Radau-discretization of the state trajectories for the two collocation implementations.

Table 2 shows the performance of the two compared implementations of collocation in terms of NLP iterations and CPU time.

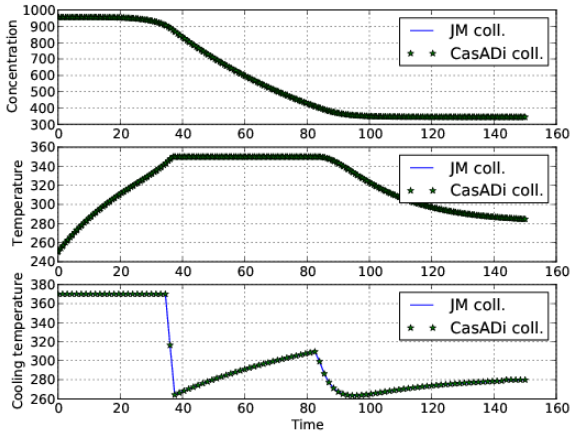

Figure 4: Optimization results for the CSTR reactor.

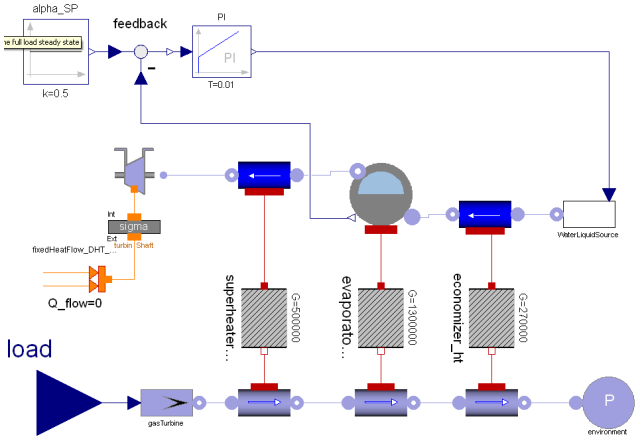

Figure 5: Diagram of the combined-cycle plant model.

Table 2: Execution times for the CSTR benchmark.

Tool	JM coll.	CasADi coll.
NLP iterations	85	20
Total time	3.2	0.18
Time in NLP solver	1.1	0.16
Time in DAE functions	2.1	0.02

In this example, the Python-based collocation algorithm is clearly superior, it converges more quickly, which is likely due to the provided exact Hessian. Also, the lion's share of the execution time is spent internally in IPOPT, leaving little opportunities to optimize the code in function evaluations further.

5.3 Optimal start-up of a combined cycle power plant

5.3.1 Physical model setup

A simplified model of a one-level-of-pressure combined-cycle power plant is considered in this benchmark, see Figure 5 for the object diagram.

The gas turbine model (lower left) generates a prescribed flow of exhaust gas at a prescribed temperature, which are both a function of the load input signal.

The turbine exhaust gases enter the hot side of a counter-current heat exchanger, and are then discharged to the atmosphere. The economizer and superheater are modelled by a dynamic energy balance equation for each side, neglecting compressibility and friction effects. The drum boiler evaporator instead includes both dynamic mass and energy balance equations, assuming thermodynamic equilibrium between the liquid and the vapour phases. The energy storage in all the steel walls is accounted for assuming they are at the same temperature of the water/steam fluid.

The feedwater system is described by a prescribed flow rate source with fixed temperature, driven by a PI level controller that stabilizes the level dynamics and keeps the void fraction in the drum around 0.5.

Finally, the superheated steam enters the steam turbine, which is modeled as an expansion to the condenser pressure, assuming a constant isentropic efficiency. The turbine also exposes a thermal port, corresponding to the surface where the inlet steam comes into contact with the turbine rotor. This port is connected to a thermal model of the hollow shaft, given by Fourier's heat equation, discretized by the finite difference method. The thermal stress on the shaft surface, which is the main limiting factor in the start-up transients, is proportional to the difference between the surface and the average temperature of the shaft.

In order to keep the complexity low, constant specific heat c_p is assumed in the economizers and superheaters; lumped-parameter models are assumed for the heat exchanger segments, with just one temperature state for each side. Last, but not least, also the turbine rotor thermal model has only one temperature state resulting from the discretization of Fourier's equation. The resulting nonlinear model has nine state variables and 127 algebraic variables. A more detailed discussion on the physical modelling of the plant can be found in [11].

5.3.2 Minimum-time start-up

The goal of the optimization problem is to reach the full load level as fast as possible, while limiting the peak stress value on the rotor surface, which deter-

mines the lifetime consumption of the turbine. Since the steam cycle is assumed to operate in a pure sliding pressure mode, the full load state is reached when the load level of the turbine, $u(t)$ (which is the control variable), has reached 100% and the normalized value of the evaporator pressure, p_{ev}, has reached the target reference value p_{ev}^{ref}. A Lagrange-type cost function, penalizing the sum of the squared deviations from the target values, drives the system towards the desired set-point $(p_{\text{evap}}, u) = (p_{\text{evap}}^{\text{ref}}, 1)$ as quickly as possible.

Inequality constraints are prescribed on the maximum admissible thermal stress in the steam turbine, $\sigma(t)$, as well as on the rate of change of the gas turbine load: on one hand, the load is forbidden to decrease, in order to avoid cycling of the stress level during the transient; on the other hand, it cannot exceed the maximum rate prescribed by the manufacturer.

The start-up optimization problem is then defined as:

$$\min_{u(t)} \int_{t_0}^{t_f} (p_{\text{evap}}(t) - p_{\text{evap}}^{\text{ref}})^2 + (u(t) - 1)^2 \, dt \qquad (8)$$

subject to the constraints

$$\begin{aligned} \sigma(t) &\leq \sigma_{\max} \\ \dot{u}(t) &\leq du_{\min} \\ \dot{u}(t) &\geq 0 \end{aligned} \qquad (9)$$

and to the DAE dynamics representing the plant. The initial state for the DAE represents the state of the plant immediately after the steam turbine roll-out phase and the connection of the electric generator to the grid.

The optimization result is shown in Figure 6. During the first 200 seconds, the gas turbine load is increased at the maximum allowed rate and the stress builds up rapidly, until it reaches the target limit. Subsequently, the load is slowly increased, in order to maintain the stress level approximately constant at the prescribed limit. When the 50% load level is reached, further increases of the load do not cause additional increase of the gas exhaust temperature, and therefore cause only small increases of the steam temperature. It is then possible to resume increasing the load at the maximum allowed rate, while the stress level starts to decrease. The full load is reached at about 1400 s. Between 1000 and 1100 seconds, the load increase rate is actually zero; apparently, this small pause allows to increase the load faster later on, leading to an overall shorter start-up time.

This problem, which is of a more realistic size has been solved with the two direct collocation implementations and the results are shown in Table 3.

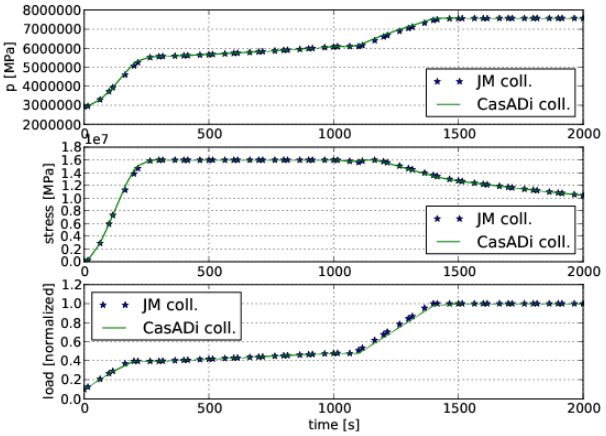

Figure 6: Optimal startup of a combined cycle power plant.

Table 3: Execution times for the combined cycle benchmark.

Tool	JM coll	CasADi coll
NLP iterations	49	198
Total time	19.9	16.3
Time in NLP solver	2.4	15.3
Optimal cost	6487	6175

We note that the CasADi-based collocation algorithm needs more iterations to reach the optimal solution, but on the other hand, the optimum that it found is indeed better than the one found using BFGS. Whether this is due to some minor differences in the collocation algorithms (since the models are identical), is beyond the scope of this paper. What can be said with certainty is that whereas most of the time spent in the DAE functions in the existing C-based approach, the opposite is true for the CasADi approach. Indeed, with more than 90% of the computational time spent internally in IPOPT, optimizing the CasADi execution time further would do little to reduce the overall execution time.

6 Summary and Conclusions

In this paper, the integration of CasADi and the JModelica.org platform has been reported. It has been shown how an XML-based model exchange format supported by JModelica.org and CasADi is used to combine the expressive power provided by Modelica and Optimica with state of the art optimization algorithms. The use of a language neutral model exchange format simplifies tool interoperability and allows users

to use different optimization algorithms without the need to reencode the problem formulation. As compared to traditional optimization frameworks, typically requiring user's to encode the model, the cost function and the constraints in a algorithm-specific manner, the approach put forward in this paper increases flexibility significantly.

7 Acknowledgments

This research at KU Leuven was supported by the Research Council KUL via the Center of Excellence on Optimization in Engineering EF/05/006 (OPTEC, http://www.kuleuven.be/optec/), GOA AMBioRICS, IOF-SCORES4CHEM and PhD/postdoc/fellow grants, the Flemish Government via FWO (PhD/postdoc grants, projects G.0452.04, G.0499.04, G.0211.05,G.0226.06, G.0321.06, G.0302.07, G.0320.08, G.0558.08, G.0557.08, research communities ICCoS, ANMMM, MLDM) and via IWT (PhD Grants, McKnow-E, Eureka-Flite+), Helmholtz Gemeinschaft via vICeRP, the EU via ERNSI, Contract Research AMINAL, as well as the Belgian Federal Science Policy Office: IUAP P6/04 (DYSCO, Dynamical systems, control and optimization, 2007-2011).

Johan Åkesson gratefully acknowledges financial support from the Swedish Science Foundation through the grant *Lund Center for Control of Complex Engineering Systems (LCCC)*.

References

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

[2] 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. Doi:10.1016/j.compchemeng.2009.11.011.

[3] Johan Åkesson and Ola Slätteke. Modeling, calibration and control of a paper machine dryer section. In *5th International Modelica Conference 2006*, Vienna, Austria, September 2006. Modelica Association.

[4] J. Andersson, B. Houska, and M. Diehl. Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling languages. In *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Oslo, Norway, October 3*, 2010.

[5] Niklas Andersson, Per-Ola Larsson, Johan Åkesson, Staffan Haugwitz, and Bernt Nilsson. Calibration of a polyethylene plant for grade change optimizations. In *21st European Symposium on Computer-Aided Process Engineering*, May 2011. Accepted for publication.

[6] D. M. Beazley. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst.*, 19:599–609, July 2003.

[7] B. M. Bell. CppAD Home Page, 2010. `http://www.coin-or.org/CppAD/`.

[8] Lorenz T. Biegler. *Nonlinear programming: concepts, algorithms, and applications to chemical processes*. SIAM, 2010.

[9] 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.

[10] H.G. Bock and K.J. Plitt. A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings 9th IFAC World Congress Budapest*, pages 243–247. Pergamon Press, 1984.

[11] F. Casella, F. Donida, and J. Åkesson. Object-oriented modeling and optimal control: A case study in power plant start-up. In *Proc. 18th IFAC World Congress*, 2011. In submission.

[12] Francesco Casella, Filippo Donida, and Johan Åkesson. An XML representation of DAE systems obtained from Modelica models. In *Proceedings of the 7th International Modelica Conference 2009*. Modelica Association, September 2009.

[13] Dassault Systemes. Dymola web page, 2010. `http://www.3ds.com/products/catia/portfolio/dymola`.

[14] M. Diehl, H.G. Bock, J.P. Schlöder, R. Findeisen, Z. Nagy, and F. Allgöwer. Real-time optimization and Nonlinear Model Predictive Control of Processes governed by differential-algebraic equations. *J. Proc. Contr.*, 12(4):577–585, 2002.

[15] M. Diehl, H. J. Ferreau, and N. Haverbeke. *Nonlinear model predictive control*, volume 384 of *Lecture Notes in Control and Information Sciences*, chapter Efficient Numerical Methods for Nonlinear MPC and Moving Horizon Estimation, pages 391–417. Springer, 2009.

[16] Inc. Enthought. SciPy, 2010. `http://www.scipy.org/`.

[17] Dave Beazley et al. SWIG – Simplified Wrapper and Interface Generator, version 2.0, 2010. `http://www.swig.org/`.

[18] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8):816–830, 2008.

[19] R Fourer, D. Gay, and B Kernighan. *AMPL – A Modeling Language for Mathematical Programming*. Brooks/Cole — Thomson Learning, 2003.

[20] R. Franke, M. Rode, and K Krü$\frac{1}{2}$ger. On-line optimization of drum boiler startup. In *Proceedings of Modelica'2003 conference*, 2003.

[21] Gams Development Corporation. GAMS web page, 2010. `http://www.gams.com/`.

[22] A. Griewank. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.

[23] Staffan Haugwitz, Johan Åkesson, and Per Hagander. Dynamic start-up optimization of a plate reactor with uncertainties. *Journal of Process Control*, 19(4):686–700, April 2009.

[24] G. A. Hicks and W. H. Ray. Approximation methods for optimal control synthesis. *Can. J. Chem. Eng.*, 20:522 –529, 1971.

[25] 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 Transactions on Mathematical Software*, 31:363–396, 2005.

[26] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 2010. DOI: 10.1002/oca.939 (in print).

[27] J. Hunter, D. Dale, and M. Droettboom. MatPlotLib: Python plotting, 2010. `http://matplotlib.sourceforge.net/`.

[28] P-O. Larsson, J. Åkesson, Staffan Haugwitz, and Niklas Andersson. Modeling and optimization of grade changes for multistage polyethylene reactors. In *Proc. 18th IFAC World Congress*, 2011. In submission.

[29] Modelisar. Functional Mock-up Interface for Model Exchange, 2010. `http://www.functional-mockup-interface.org`.

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

[31] B. Olofsson, H. Nilsson, A. Robertsson, and J. Åkesson. Optimal tracking and identification of paths for industrial robots. In *Proc. 18th IFAC World Congress*, 2011. In submission.

[32] Roberto Parrotto, Johan Åkesson, and Francesco Casella. An XML representation of DAE systems obtained from continuous-time Modelica models. In *Third International Workshop on Equation-based Object-oriented Modeling Languages and Tools - EOOLT 2010*, September 2010.

[33] J. Poland, A. J. Isaksson, and P. Aronsson. Building and solving nonlinear optimal control and estimation problems. In *7th International Modelica Conference*, 2009.

[34] Process Systems Enterprise. gPROMS Home Page, 2010. `http://www.psenterprise.com/gproms/index.html`.

[35] K. Prölss, H. Tummescheit, S. Velut, Y. Zhu, J. Åkesson, and C. D. Laird. Models for a post-combustion absorption unit for use in simulation, optimization and in a non-linear model predictive control scheme. In *8th International Modelica Conference*, 2011.

[36] Python Software Foundation. Python Programming Language – Official Website, January 2011. `http://www.python.org/`.

[37] The Modelica Association. Modelica – a unified object-oriented language for physical systems modeling, language specification, version 3.2. Technical report, Modelica Association, 2010.

[38] Lee Thomason. TinyXML, January 2011. `http://www.grinninglizard.com/tinyxml/`.

[39] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–58, 2006.

[40] V. M. Zavala and L. T. Biegler. The advanced step NMPC controller: Optimality, stability and robustness. *Automatica*, 45(1):86–93, 2009.