Dynamic optimization with CasADi*

Joel Andersson[1] and Johan Åkesson[2] and Moritz Diehl[1]

***Abstract*— We demonstrate how CasADi, a recently developed, free, open-source, general purpose software tool for nonlinear optimization, can be used for dynamic optimization in a flexible, interactive and numerically efficient way.**

CasADi is best described as a minimalistic computer algebra system (CAS) implementing automatic differentiation (AD) in eight different flavors. Similar to algebraic modeling languages like AMPL or GAMS, it includes high-level interfaces to state-of-the-art numerical codes for nonlinear programming, quadratic programming and integration of differential-algebraic equations. CasADi is implemented in self-contained C++ code and contains full-featured front-ends to Python and Octave for rapid prototyping.

In this paper, we discuss CasADi from the perspective of the developer or advanced user of algorithms for dynamic optimization for the first time, leaving out details on the implementation of the tool. We demonstrate how the tool can be used to model highly complex dynamical systems directly or import existing models formulated in the algebraic modeling language AMPL or the physical modeling language Modelica. Given this symbolic representation of the process models, the resulting optimal control problem can be solved using a variety of methods, including transcription methods (collocation), methods with embedded integrators (multiple shooting) as well as indirect methods.

I. Introduction and motivation

In *dynamic optimization* or *optimal control*, solutions are sought for decision-making problems constrained by dynamic equations in the form of ordinary differential equations (ODE) or differential-algebraic equations (DAE). Optimal control problems (OCPs) can cover many industry-relevant features, such as multi-stage problem formulations, problems with integer valued decision variables, problems with multi-point constraints and problems with uncertainty. The fact that the problem formulation easily becomes so general, along with a large set of different solution algorithms, makes it difficult to implement software tools that treat optimal control problems with some generality. While tools do exist that deal with a broad range of OCPs – examples include ACADO Toolkit [1], DIRCOL [2], DyOS [3] and MUSCOD-II [4], [5] – the usage of these tools, in particular for a user not directly involved in their actual developement, is typically limited to comparably restricted problem formulations.

The CasADi project addresses optimal control problems using a different approach than the above packages. Rather than providing the end user with a ”black box” OCP solver, the focus is a framework that allows advanced users to implement their method of choice, with any complexity. The framework takes the form of a minimalistic computer algebra system (CAS) with a general implementation of automatic differentiation (AD). Around this *symbolic core* is a set of interfaces to numerical codes for nonlinear programming (NLP), ODE/DAE integration, quadratic programming (QP) and solution of linear systems as well as front-ends to the scripting languages Python and Octave for fast prototyping and easy access to facilities for visualization and user interaction.

The main contribution of this paper is that we present CasADi in a form that we believe makes it accessible to a practitioners of optimal control. We show and discuss how popular optimal control methods like direct single-shooting, direct multiple shooting and direct collocation in a high-level language such as Python or Octave, using a fraction of the amount of code needed to implement the same method in a low-level language like C. We see, for example, that a direct single-shooting method, with exact first and second derivatives, can be implemented with only 35 lines of code. Thanks to convenient interfaces and efficient virtual machines, we can ensure that the implementation in the high-level language is as fast as an equivalent C implementation.

The remainder of the paper is organized as follows. Section II gives an overview of numerical methods for optimal control and motivates the need for a generic symbolic framework. It also briefly discusses automatic differentiation (AD), which is the most central capability of CasADi. Section III gives an overview of the structure and syntax of CasADi which is followed by Section IV, where we discuss different optimal control approaches using CasADi in a tutorial fashion. The paper ends with a real-world example solved with CasADi in Section V and some concluding remarks in Section VI.

*Joel Andersson and Moritz Diehl were supported by the Research Council KUL: PFV/10/002 Optimization in Engineering Center OPTEC, GOA/10/09 MaNet and GOA/10/11 Global real- time optimal control of autonomous robots and mechatronic systems. Flemish Government: IOF/KP/SCORES4CHEM, FWO: PhD/postdoc grants and projects: G.0320.08 (convex MPC), G.0377.09 (Mechatronics MPC); IWT: PhD Grants, projects: SBO LeCoPro; Belgian Federal Science Policy Office: IUAP P7 (DYSCO, Dynamical systems, control and optimization, 2012-2017); EU: FP7-EMBOCON (ICT-248940), FP7-SADCO (MC ITN-264735), ERC ST HIGHWIND (259 166), Eurostars SMART, ACCM.

[1]J. Andersson and M. Diehl are with the Electrical Engineering Department (ESAT) and Optimization in Engineering Center (OPTEC), K.U. Leuven, Kasteelpark Arenberg 10, 3001 Heverlee, Belgium, joel.andersson at esat.kuleuven.be

[2]J. Åkesson is with the Department of Automatic Control, Faculty of Engineering, Lund University, BOX 118, 21100 Lund, Sweden, johan.akesson at control.lth.se

II. Background

A basic formulation of an *optimal control problem* (OCP) is:

$$\underset{\substack{x\in[0,T]\times\mathbb{R}^{N_x}\\ u\in[0,T]\times\mathbb{R}^{N_u}\\ p\in\mathbb{R}^{N_p}}}{\text{minimize}} \quad \int_0^T l(t,x(t),u(t),p)\,dt + E(T,x(T),p) \tag{1}$$

$$\text{subject to} \quad \dot{x} = f(t,x(t),u(t),p), \quad t\in[0,T] \tag{2}$$

$$\underline{x}_0 \le x(0) \le \overline{x}_0, \quad \underline{x}_\mathrm{f} \le x(T) \le \overline{x}_\mathrm{f} \tag{3}$$

$$\underline{x} \le x(t) \le \overline{x}, \quad t\in(0,T) \tag{4}$$

$$\underline{u} \le u(t) \le \overline{u}, \quad t\in[0,T] \tag{5}$$

$$\underline{p} \le p \le \overline{p}, \tag{6}$$

where $x(\cdot)\in\mathbb{R}^{N_x}$ denote differential states, $u(\cdot)\in\mathbb{R}^{N_u}$ are free control signals and $p\in\mathbb{R}^{N_p}$ a set of free parameters in the model.

The OCP consists of an objective function (1) with a way-cost (or *Lagrange term*) and an end-cost (or *Mayer term*), a *dynamic constraint* (2) in the form of an ODE with initial and terminal conditions (3) as well as bounds on the states (4), control (5) and parameters (6).

A. Numerical methods for optimal control

Numerical methods for solving the OCP given by Equations (1) to (6) as well as more general OCPs emerged with the birth of the electronic computer in the 1950's and were at the beginning typically based on either *dynamic programming*, which is limited to small state dimension, or methods based on the calculus of variations, so-called *indirect methods*. The difficulty of indirect methods to deal with inequality constraints, represented above as bounds on the state and control trajectories and the emergence of powerful nonlinear programming (NLP) solvers, 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 NLP. Direct methods can, in turn, be subdivided into *sequential* and *simultaneous* methods. Wheras a sequential method will eliminate the state trajectories from the NLP problem, leading to a small but dense NLP, a simultaneous method will include both state and control trajectories in a larger, but more sparse and often less nonlinear NLP. The relation between the methods is illustrated in Figure 1

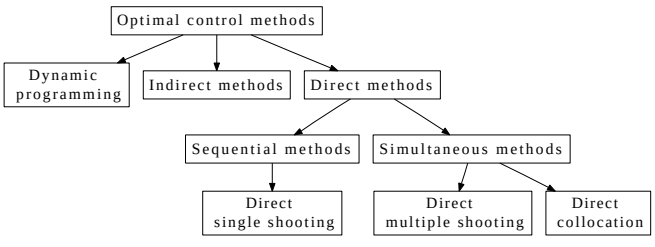


Fig. 1. Direct methods for dynamic optimization

A key issue when solving NLPs originating from both direct and indirect optimal control approaches is the accurate and fast generation of first and (preferably) second order derivatives. When using a *shooting method* such as *direct single-shooting* or *direct multiple-shooting* [4], [5], the NLP formulation will contain calls to an embedded solver for ODE or DAE initial value problems. When using *direct collocation*, the NLP will involve a large number of variables and constraints. The method of choice for these derivative calculations involving such structurally complex and often large-scale NLP functions is *automatic differentiation*, covered next.

B. Automatic differentiation

Automatic differentiation (AD), is a technique for evaluating derivatives of computer represented functions, which has proved useful in nonlinear optimization. An introduction to AD can be found in Griewank and Walther's book [6].

The technique delivers directional derivatives, up to machine precision, of arbitrary differentiable functions using at most four times the number of floating point operations necessary to calculate the original function. In the *forward mode*, the technique delivers a Jacobian-times-vector product of a vector-valued function with a computational cost of the same order as the cost of evaluating the original function. In the *reverse mode*, on the other hand, a (row)vector-times-Jacobian product can be obtained at a comparable cost. The latter is particularly interesting when the function is scalar valued, giving the so-called *cheap gradient result* [6].

Complete Jacobians can be computed by repeating the procedure above and conceptually multiplying the Jacobian with a set of vectors, the simplest choice being the coordinate vectors ($e_j = [0,\ldots,1^{(\text{position } j)},\ldots,0]$). The cost of evaluating the Jacobian this way is proportional to the number of independent variables, in the forward mode, or the dimension of the function, in the reverse mode. There are, however, considerable savings to be made by exploiting structure, sparsity and symmetry of the Jacobian. See [7] for an overview of such techniques.

III. Structure of CasADi

CasADi is designed to have the *look-and-feel* of convensional computer algebra systems such as SymPy or MATLAB's Symbolic Math Toolbox, but uses a different, more economical way of representing symbolic expressions taken from the field of automatic differentiation. To see this, consider the following piece of code for CasADi's Python front-end (Section III-A):

```
from casadi import *
x = ssym("x",7,7)
f = det(x)
grad_f = gradient(f,vec(x))
```

In the second line of the code, the function `ssym` is invoked to declare a 7-by-7 matrix with symbolic variables (cf. the function `sym` in MATLAB) and in the third line of the code we form an expression of the determinant of the matrix using minor expansion[1], an operation that grows

[1]Note that there are certainly better ways of calculating determinants.

exponentially in complexity with the matrix dimension. In the fourth and last row we form an expression for the gradient of the determinant with respect to all 49 inputs (as one column, hence "`vec`") using the reverse mode of automatic differentiation (see e.g. [6]).

The above script, which will result in an expression graph (or computational graph) with 393.000 edges, completes in less than a second in CasADi.

There is a growing set of operations normally associated with computer algebra tools in CasADi. This includes symbolic evaluations, (rudimentary) expression simplifications as well as factorization and solution of linear systems of equations that contain symbolic expressions. While the capabilities in terms of such symbolic manipulations are modest in comparison with advanced computer algebra systems such as Maple, CasADi does have a very general and very fast state-of-the-art implementation of AD.

The AD framework of CasADi, implements AD in eight different flavors, with each flavors being a combination of the following three mutually orthogonal dimensions:

- Forward and reverse mode
- Two different ways of representing symbolic expressions; using expression graphs with nodes that are either *scalar-valued* or *matrix-valued*
- Both an *operator overloading* and a *source code transformation* approach (see [6]) for calculating directional derivatives as well as complete Jacobians and Hessians

For a less superficial overview of the AD framework of CasADi, we refer to [8].

A. Python and Octave front-ends

While C++ is an excellent language for writing high-performance code, scripting languages such as Matlab, Python or Octave are often preferred when working interactively and when the user wishes to make use of packages for scientific computing or visualization. CasADi uses the open-source tool SWIG [9] to parse CasADi's C++ header files, and then automatically generate full-featured and easily maintained front-ends to Python and Octave. These front-ends make rapid prototyping possible without any compiler in the loop and can typically be used with small differences in performance, since the computationally heavy calculations take place in CasADi's virtual machines and in interfaced code.

B. Software interfaces

For the development of optimization code, several solver packages have been interfaced. Interfaced software tools include nonlinear programming (NLP) solvers such as IPOPT [10], KNITRO [11] and WORHP [12], where functions for first and second order derivatives with the appropriate sparsity information are automatically generated and interfaced. Interfaced are also a number of quadratic programmin (QP) solvers such as qpOASES [13], OOQP [14] and CPLEX [15] as well as ODE and DAE integrators such as the Sundials integrators [16]. Directional derivatives of the integrators are calculated through automatic formulation and solution of the forward and adjoint sensitivity equations (see e.g. [17], [18]).

C. Model import from AMPL and Modelica

Modeling of dynamical systems can be performed directly in CasADi using a MATLAB-inspired "everything-is-a-matrix" syntax as the simple example in Section IV shows, but it is also possible to import models written in another modeling environment. Currently, model import is possible from the physical modeling language Modelica [19] via the open-source compiler JModelica.org [20], [21] as well as from the algebraic modeling language AMPL [22]. Since the model import is symbolic, imported expressions can be manipulated like native CasADi expressions.

IV. Dynamic optimization with CasADi

This section deals with using CasADi to solve optimal control problems. We shall demonstrate how CasADi can be used for different numerical optimal control algorithms using an approach which is significantly more flexible and extendable than that of existing environments for dynamic optimization. CasADi has been written to be able to deal with the optimization of large-scale optimal control problems, but for simplicity of the discussion, we shall demonstrate the approach for a simple toy example, taken from [23]:

A. A simple optimal control problem

Consider the problem of driving a *Van der Pol* oscillator to the origin with a quadratic integral cost:

$$\underset{\substack{x\in[0,10]\times\mathbb{R}^3\\ u\in[0,10]\times\mathbb{R}}}{\text{minimize}} \quad x_2(10) \tag{7}$$

$$\text{subject to} \quad \dot{x} = \begin{bmatrix} (1-x_1^2)\,x_0 - x_1 + u \\ x_0 \\ x_0^2 + x_1^2 + u^2 \end{bmatrix}, \; t \in [0,10] \tag{8}$$

$$[0,1,0]^{\mathrm{T}} \leq x(0) \leq [0,1,0]^{\mathrm{T}},$$

$$[0,0,-\infty]^{\mathrm{T}} \leq x(10) \leq [0,0,\infty]^{\mathrm{T}} \tag{9}$$

$$-0.75 \leq u(t) \leq 1.0, \quad t \in [0,10], \tag{10}$$

which is a special case of Equations (1) to (6). The OCP can be formulated using CasADi's Python front-end as follows:

```
from casadi import *
x=ssym("x",3,1); u=ssym("u",1,1)
f =      [(1-x[1]*x[1])*x[0]-x[1]+u,\
          x[0],\
          x[0]*x[0]+x[1]*x[1]+u*u]
T      = 10.0;       E      = x[2]
x0     = [0,1,0];
xf_min = [0,0,-inf]; xf_max = [0,0,inf]
u_min  = -0.75;      u_max  = 1.0
```

where `ssym("name",n,m)` denotes the declaration of an n-by-m symbolic expression named `name`. In the code above, `f` and `E` are symbolic expressions for the ODE right-hand-side and cost function respectively. From the

expressions, we can easily generate functions that can be evaluated efficiently both symbolically and numerically:

```
ffcn = SXFunction([x,u],[f]); ffcn.init()
Efcn = SXFunction([x,u],[E]); Efcn.init()
```

B. Control parametrization (direct methods)

If we choose a direct optimal control method (Section II-A), we need to parametrize the control trajectory. The simplest choose is a piecewise constant parametrization:

$$u(t) = u_k, \quad t \in \left(\frac{10\,k}{N}, \frac{10\,(k+1)}{N} \right], \quad k = 0, \ldots, N-1 \tag{11}$$

Choosing $N = 20$ intervals, we can declare the parametrized control in CasADi:

```
N = 20; U = msym("U",N,1)
```

where `msym("name",n,m)` denotes declaration of an n-by-m symbolic expression named `name`, as above, but using the second symbolic representation with matrix-valued instead of scalar valued nodes in the expression graph (for a discussion on when to use which symbolic representation, we refer to [8]).

C. Direct single-shooting

The conceptually simplest way to solve the Van der Pol problem is to write a direct single-shooting method. This is a method where we call ODE or DAE integrators in order to eliminate the state x from the problem leaving only the discretized controls in the NLP, which will be of the form:

$$\underset{U \in \mathbb{R}^N}{\text{minimize}} \quad f_{\text{NLP}}(U) \tag{12}$$

$$\text{subject to} \quad [0,0]^{\text{T}} \leq g_{\text{NLP}}(U) \leq [0,0]^{\text{T}} \tag{13}$$

$$-0.75 \leq U \leq 1.0, \tag{14}$$

To implement this method for the Van der Pol problem, we need an ODE integrator with forward and adjoint sensitivity capabilities. Here we can choose to use one of the interfaced third party integrators, e.g. CVodes from the Sundials suite (see Section III-B), or we can write one in CasADi directly. As a simple (and not very efficient) example, the following code will produce an explicit Euler integrator for the Van der Pol example taking $M = 100$, uniformly distributed steps:

```
xk=msym("xk",3,1); uk=msym("uk",1,1)
M = 100; dt = T/(M*N); xkj = xk
for j in range(M):
  [f_kj] = ffcn.call([xkj,uk])
  xkj = xkj + dt*f_kj
integrator = MXFunction([xk,uk],[xkj])
integrator.init()
```

The code defines a function `integrator` that is defined by the expression `xkj` for the state at the end of a control interval, given the state at the initial time and control during the interval. Note that this symbolic expression contains *calls* (hence the `call` statement) to the ODE right hand side function `ffcn` defined earlier.

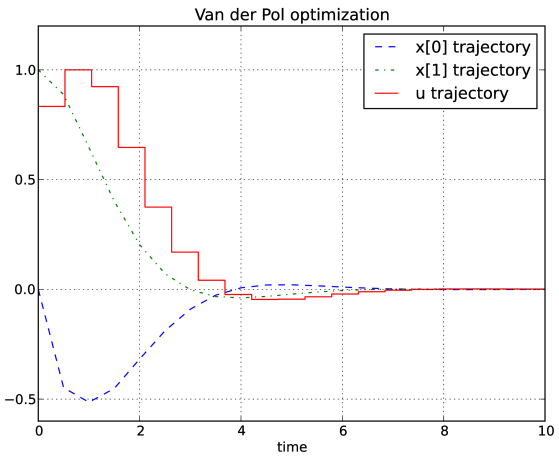


Fig. 2. Optimal solution for the Van der Pol optimal control problem

With the dynamics now in discrete time form, we are able to form an expression for the state at the final time, as a function of only the parametrized control:

```
Xk = x0
for k in range(N):
  [Xk] = integrator.call([Xk,U[k]])
```

This expression, in turn, allows us to form objective and constraint functions of the NLP:

```
[obj] = Efcn.call([Xk,U[N-1]])
f_NLP = MXFunction([U],[obj])
con = vertcat([Xk[0],Xk[1]])
g_NLP = MXFunction([U],[con])
```

We can then solve this NLP, e.g. using CasADi's interface to IPOPT, relying on CasADi to calculate first and second order derivatives:

```
solver = IpoptSolver(f_NLP,g_NLP)
solver.init()
solver.setInput(N*[1],NLP_X_INIT)
solver.setInput(N*[u_min],NLP_LBX)
solver.setInput(N*[u_max],NLP_UBX)
solver.setInput([0,0],NLP_LBG)
solver.setInput([0,0],NLP_UBG)
solver.solve()
```

We show the results of the optimization in Figure 2.

D. Simultaneous methods

The main purpose of the above example was to convey the idea of CasADi and cannot be characterized as a very efficient optimal control problem solver, not least due to the choice of the integrator. Said that, the above script, with 34 lines of code in total, readily generalizes to structurally complex model formulations and to other optimal control methods.

For example, the direct multiple shooting method by Bock and Plitt [4], [5] can be obtained by adding the state

trajectory (`Xk` in the example above) to the list of NLP variables and using the defining equation of `Xk` as an equality constraint. This causes the Jacobian of the NLP constraint function and the Hessian of the Hessian of the Lagrangian to become large and sparse, which is efficienctly exploited by CasADi's AD framework as well as in interfaced NLP solvers.

Similarily, the direct collocation method (see e.g. [24]), can be implemented by replacing the explicit Euler integrator above with an implicit Runge-Kutta method (the simplest choice being the implicit Euler method) and this time moving not only the parametrization of the state trajectory at the M "shooting nodes" (`Xk` above), but also the step at all the $M \times N$ integration steps (`xjk` above) and their defining equations to the NLP level. The even larger size, and additional sparsity is again exploited in the AD framework as well as in NLP solvers such as IPOPT.

We can also generalize the OCP formulation beyond that of Equations (1) to (6) by allowing a different differential equation and possibly a different number of states for different parts of the optimization horizon (a multi-stage formulation), simply by forming multiple different integrator instances and/or ODE right hand side functions.

E. Indirect optimal control methods

CasADi is not restricted to be used together with *direct* methods for optimal control. In fact, the symbolic framework provides all the necessary tools needed for an indirect, or *optimize-then-discretize* approach. This includes the formulation of the costate equations (by using reverse mode automatic differentiation symbolically) as well as the possibility to embed parametric NLP-solvers into integration algorithms. For a concrete example, we refer to the CasADi Users' Guide [25].

V. Real-world applications

CasADi has been written to deal with dynamic optimization of large-scale, nonlinear DAE systems. A frequent source of such models are component-based physical modeling environments such as Modelica [19]. We will now report on how CasADi can be used for such systems.

A. Optimized start-up of a combined-cycle power plant

A relevant and challenging dynamic optimization problem is the minimal start-up of combined cycle power plants (CCPP). These plants, which are often used for peak-load electricity production, are frequently started up and shut down causing large energy losses as well as strains in components, shortening their lifetimes and potentially leading to catastrophic failure. By using optimization-based control techniques such as nonlinear model predictive control (NMPC) [26], it is possible to minimize fuel use to (grid) electricity production, while at the same time respecting these hard constraints on strains and pressures.

To model the plant, we use a first-principle model of the plant written in Modelica as well as Optimica [20], an extension of Modelica allowing the formulation of dynamic optimization constructs such as cost functions and constraints.

A shematic diagram of the model, which consists of a gas turbine and a three stage (economizer, evaporator, superheater) steam turbine is shown in Figure 3. For details of the model and the optimal control problem formulation, the minimum time start-up under a turbine rotor stress constraint, we refer to [27].

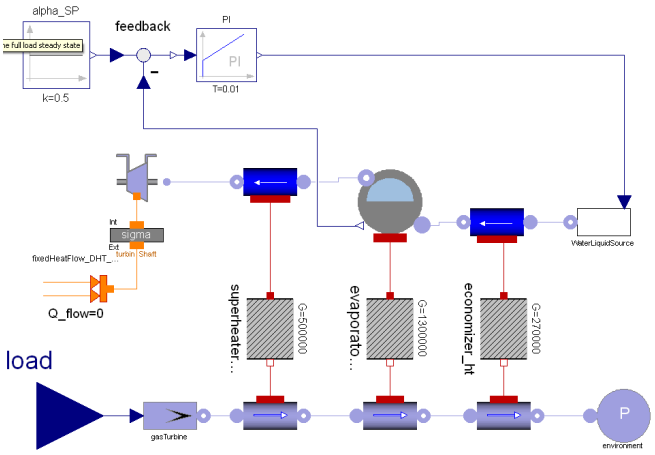


Fig. 3. A combined cycle power plant

B. Model import into CasADi and NLP transcription

We use JModelica.org [20] to translate Modelica/Optimica code into an OCP in DAE, with 137 states (10 of which appearing differentiated in the DAE) and an integral (Lagrange) cost function. This OCP is then exported in a symbolic XML-based format and imported into CasADi as reported in [21]. With the model now completely formulated as symbolic expressions in CasADi, the OCP can be used and manipulated just like the Van der Pol model in Section IV.

To reformulate the OCP to an NLP, we choose direct collocation approach on 40 finite elements using fourth order Radau points and Lagrange polynomials [24]. The collocation implementation used is freely available as part of the JModelica.org package. A first prototype of the CasADi-based collocation implementation was presented in [21], and it was later completed in [28].

C. Solution using IPOPT

The NLP was solved in [28] using CasADi's interface to IPOPT (version 3.10) using MA27 as the linear solver. With default options and exact Hessian of the Lagrangian calculation using CasADi's AD framework, IPOPT's primal-dual interior point method needed 79 iterations to converge to the optimal solution. A break-up of the timings into NLP solver initialization as well as the IPOPT internal and CasADi internal part of the NLP solution time, is shown in Table I.

TABLE I

TIMINGS FOR THE CCPP EXAMPLE [28]

Solution step	**Time**
OCP→NLP transcription & solver initialization	4.2 s
IPOPT internally & linear solver (MA27)	4.0 s
NLP function and derivative evaluation in CasADi	0.8 s

As seen in the table, over 80 % of the NLP solution time was spent in IPOPT internally (mainly in the linear solver). The 0.8 s solution time should compared with 46 s needed by a previous implementation of the same method, based on a general purpose AD tool applied to generated C code. This in addition to being implemented in a high level language (Python) rather than C. As shown in [8], the time spent in CasADi can furthermore be cut by a factor five by generating C-code for the NLP functions.

VI. CONCLUSIONS

We have presented the recently developed, free, open-source optimization framework CasADi, which is a symbolic tool aimed at giving users an efficient, yet transparent approach to automatic differentiation and to shortening the development times for algorithms for large-scale nonlinear optimization in general and optimal control in particular. It gives developers and advanced users of optimal control a flexible tool that competes favorably with state-of-the-art automatic differentiation tools in terms of numerical efficiency, but with a syntax and usage similar to computer algebra systems.

We demonstrated how the tool can be used to implement different direct optimal control methods, including single shooting, multiple shooting and collocation and reported on a how the package was used successfully in a real-world application.

Successful applications of CasADi is now starting to emerge and it is the belief and hope of the authors of this paper that practitioners of optimal control will find the package useful in application-oriented research, in development of optimization methods as well as in teaching.

ACKNOWLEDGMENT

The authors would like to thank everyone who has contributed to the CasADi project, in particular Joris Gillis, Attila Kozma and Carlo Savorgnan. Special thanks also go to Fredrik Magnusson, for his work on JModelica.org's CasADi-based collocation algorithm and to Francesco Casella, for his work on the combined cycle benchmark problem.

REFERENCES

[1] B. Houska, H. Ferreau, and M. Diehl, "An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range," *Automatica*, vol. 47, no. 10, pp. 2279–2285, 2011.

[2] O. von Stryk, *User's Guide for DIRCOL*. Fachgebiet Simulation und Systemoptimierung (SIM), Technische Universität Darmstadt, 2000.

[3] RWTH Aachen, Germany, *DyOS User Manual*, 2.1 ed., 2002.

[4] H. Bock and K. Plitt, "A multiple shooting algorithm for direct solution of optimal control problems," in *Proceedings 9th IFAC World Congress Budapest*, pp. 243–247, Pergamon Press, 1984.

[5] D. Leineweber, *Efficient reduced SQP methods for the optimization of chemical processes described by large sparse DAE models*, vol. 613 of *Fortschritt-Berichte VDI Reihe 3, Verfahrenstechnik*. Düsseldorf: VDI Verlag, 1999.

[6] A. Griewank and A. Walther, *Evaluating Derivatives*. SIAM, 2 ed., 2008.

[7] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your Jacobian? Graph coloring for computing derivatives," *SIAM Rev*, vol. 47, pp. 629–705, 2005.

[8] J. Andersson, J. Åkesson, and M. Diehl, "CasADi – A symbolic package for automatic differentiation and optimal control," in *Recent Advances in Algorithmic Differentiation* (S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, eds.), Lecture Notes in Computational Science and Engineering, (Berlin), Springer, 2012.

[9] D. M. Beazley, "Automated scientific software scripting with SWIG," *Future Gener. Comput. Syst.*, vol. 19, pp. 599–609, 2003.

[10] A. Wächter and L. Biegler, "On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.

[11] R. H. Byrd, J. Nocedal, and R. A. Waltz, "KNITRO: An integrated package for nonlinear optimization," in *Large Scale Nonlinear Optimization* (G. Pillo and M. Roma, eds.), pp. 35–59, Springer Verlag, 2006.

[12] T. Nikolayzik, C. Büskens, and M. Gerdts, "Nonlinear large-scale Optimization with WORHP," in *Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, 2010.

[13] H. Ferreau, "qpOASES – An Open-Source Implementation of the Online Active Set Strategy for Fast Model Predictive Control," in *Proceedings of the Workshop on Nonlinear Model Based Control – Software and Applications, Loughborough*, pp. 29–30, 2007.

[14] E. Gertz and S. Wright, "Object-Oriented Software for Quadratic Programming," *ACM Transactions on Mathematical Software*, vol. 29, no. 1, pp. 58–81, 2003.

[15] IBM Corp., *IBM ILOG CPLEX V12.1, User's Manual for CPLEX*, 2009.

[16] A. Hindmarsh, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Transactions on Mathematical Software*, vol. 31, pp. 363–396, 2005.

[17] S. Li and L. Petzold, "Software and algorithms for sensitivity analysis of large-scale differential algebraic systems," *Journal of Computational and Applied Mathematics*, vol. 125, pp. 131–145, 2000.

[18] Y. Cao, S. Li, L. Petzold, and R. Serban, "Adjoint Sensitivity Analysis for Differential-Algebraic Equations: The Adjoint DAE System and its Numerical Solution," *SIAM Journal on Scientific Computing*, vol. 24, no. 3, pp. 1076–1089, 2003.

[19] The Modelica Association, "The Modelica Association Home Page." http://www.modelica.org, 2012.

[20] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit, "Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem," *Computers and Chemical Engineering*, vol. 34, pp. 1737–1749, 2010.

[21] J. Andersson, J. Åkesson, F. Casella, and M. Diehl, "Integration of CasADi and JModelica.org," in *8th International Modelica Conference*, 2011.

[22] D. M. Gay, "Automatic differentiation of nonlinear ampl models," in *In Automatic Differentiation of Algorithms: Theory, implementation and application*, pp. 61–73, SIAM, 1991.

[23] "JModelica.org website." http://www.jmodelica.org/.

[24] L. T. Biegler, *Nonlinear Programming*. MOS-SIAM Series on Optimization, SIAM, 2010.

[25] J. Andersson, A. Kozma, J. Gillis, and M. Diehl, *CasADi Users' Guide, version 1.3*. Electrical Engineering Department (ESAT-SCD) and Optimization in Engineering Center (OPTEC), K.U. Leuven, 2012.

[26] R. Findeisen and F. Allgöwer, "An Introduction to Nonlinear Model Predictive Control," in *Proc. of 21st Benelux Meeting on Systems and Control*, 2002.

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

[28] F. Magnusson, "Collocation Methods in JModelica.org," Master's thesis, Department of Automatic Control, Lund University, Sweden, 2012.