Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling languages

Joel Andersson Boris Houska Moritz Diehl

Department of Electrical Engineering and Optimization in Engineering Center (OPTEC), K.U. Leuven, Belgium,
{joel.andersson,boris.houska,moritz.diehl}@esat.kuleuven.be

Abstract

The Directed Acyclic Graph (DAG), which can be generated by object oriented modelling languages, is often the most natural way of representing and manipulating a dynamic optimization problem. With this representation, it is possible to step-by-step reformulate an (infinite dimensional) dynamic optimization problem into a (finite dimensional) non-linear program (NLP) by parametrizing the state and control trajectories.

We introduce CasADi, a minimalistic computer algebra system written in completely self-contained C++. The aim of the tool is to offer the benifits of a computer algebra to developers of C++ code, without the overhead usually associated with such systems. In particular, we use the tool to implement automatic differentiation, AD.

For maximum efficiency, CasADi works with two different graph representations interchangeably: One supporting only scalar-valued, built-in unary and binary operations and no branching, similar to the representation used by today's tools for automatic differentiation by operator overloading. Secondly, a representation supporting matrix-valued operations, branchings such as if-statements as well as function calls to arbitrary functions (e.g. ODE/DAE integrators).

We show that the tool performs favorably compared to CppAD and ADOL-C, two state-of-the-art tools for AD by operator overloading. We also show how the tool can be used to solve a simple optimal control problem, minimal fuel rocket flight, by implementing a simple ODE integrator with sensitivity capabilities and solving the problem with the NLP solver IPOPT. In the last example, we show how we can use the tool to couple the modelling tool JModelica with the optimal control software ACADO Toolkit.

Keywords computer algebra system, automatic differentiation, algorithmic differentiation, dynamic optimization, Modelica

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

1. Introduction

Simulation of dynamic systems formulated in languages for physical modelling in continuous time typically amounts to solving initial-value problems in ordinary differential equations. This, in turn, requires the repeated evaluation of the ODE right-hand-sides, root-finding functions corresponding to hybrid behavior and user output functions.

With the rising interest of employing the developed dynamic models not only for simulation purposes, but also for dynamic optimization (i.e. optimization problems where the dynamic system enters as a constraint), it becomes critically important to have efficient ways of accurately evaluating not only the these functions, but also their derivatives. Gradient information is needed by implicit methods for integrating ODEs as well as in both simultaneous (e.g. direct collocation) and sequential methods (single shooting, multiple shooting) for dynamic optimization, the two families of methods that have been the most successful in solving large-scale dynamic optimization problems [4].

While dynamic optimization can certainly be useful for users of modelling languages, there are also large synergies for developers of optimization routines. The Directed Acyclic Graph (DAG) representation is used by computer algebra systems as well as object oriented modelling languages and is often the most convenient way to represent a complex non-linear function. The representation provides more information for the optimization routine than a set of "black-box" functions for evaluating nonlinear functions (objective function, constraints, etc.) and their derivatives in addition to the sparsity structure of Hessians and Jacobians, which is the representation used in most of today's tools for nonlinear optimization.

Optimization routines can benefit from the DAG representation since it gives a possibility to manipulate the graph and replace certain nodes in order to give the (dynamic or not dynamic) optimization problem a more favorable form. Examples of this is replacing *internal switches* by *external switches* to form a mixed-integer optimal control problem (MIOCP) when dynamic constraints are present or as a mixed-integer nonlinear problem (MINLP) when this is not the case. Another example, used by software such as CVX [8], is the possibility of reformulating a convex optimization problem, with an infavorable structure, into an equivalent convex problem in form required by numerical

solvers [7]. Finally, we can also easily parametrize the control and/or state trajectory, reducing our dynamic optimization problem into either a parameter estimation problem or even a nonlinear program (NLP) [14].

A second way in which optimization routines can use the DAG representation is through *structure exploitation*. For example, the *Lifted Newton method* [12] offers a way to treat the intermediate variables of a nonlinear optimization problem as degrees of freedom of the problem *without* increasing the size of the problem. Including intermediate variables in the problem formulation is known to increase both the convergence rate and the area of attraction for nonlinear problems. The Lifted Newton method works with a problem formulation which is easily obtained from a linear ordering of the nodes of the DAG.

1.1 Automatic differentiation

Automatic differentiation (or, alternatively, *algorithmic* differentiation), AD, is a technique for evaluating derivatives of complex functions that has proved very useful in non-linear optimization.

The technique delivers derivatives, up to machine precision, of arbitrary differentiable functions for a small multiple of the cost of evaluating the original function. In the *forward mode*, the technique is able to deliver directional derivatives of an arbitrary vector-valued function with a cost of the same order as the cost of evaluating the original function. In the *reverse mode*, on the other hand, it is for the cost of one evaluation possible to get the derivative in an all directions for a scalar function[1].

Efficiently implementing AD, especially in the reverse mode, is a complex task and it is advisable to use one of the existing software implementations dedicated for this. These software tools are typically divided into AD by *operator overloading* and AD by *source code transformation*, see [9] for details.

Since modelling languages such as Modelica typically involve a step of C-code generation, the natural approach of using AD has been to apply one of the existing AD tools to this code.

Automatic differentiation can also be implemented inside a computer algebra system (CAS), first demonstrated by Monagan and Neuenschwander [13].

2. Proposed AD framework for object oriented modelling tools

The method proposed in this paper follows the CAS/AD approach, but instead of using an existing computer algebra system, it implements a minimalistic computer algebra system in C++ using operator overloading. The main difference from the conventional operator overloading approach for AD is that we do not attempt to maintain a linear ordering of the nodes when the graph is constructed. Instead, we generate an ordering after the graph has been constructed, which can be done efficiently using linear time algorithms, described in Section 2.2.

Also different from conventional AD tools is that we shall build a graph of sparse, matrix-valued operations, instead of just scalar operations, and allow the graph to contain ”switches” in the form of if-statements, for-loops etc. This means that there is no need to reconstruct the graph or its linear ordering when a switch fires. When there are a lot of switching events relative to the number of function evaluations, this method promises to be significantly faster.

When calculating the full Hessian or Jacobian (as opposed to only a directional derivative), CasADi uses the *Sparse Jacobian/Hessian* methods rather than compression techniques ([9]). The former methods are more efficient from a theoretical point of view, but not widely used since having to keep track of dependencies for each evaluation node results in significant overhead. CasADi is able to avoid this overhead by resorting to source code transformation, whenever multiple derivative directions (forward or adjoint) are involved.

Given the linear ordering of a graph, it is generally straightforward to generate source code for evaluation of functions and derivatives, but in general we will stop short of doing so and instead evaluate the expressions on a *virtual machine*. This eliminates the need of a C compiler in the loop, which has large practical consequences, and also saves us the trouble of working with potentially very large files. As we shall see, a DAG with millions of nodes can be constructed and sorted within seconds and evaluated in milliseconds, but generating a C source file of it may require hundreds of MB, which have to be written to disk. With the current approach, the graph representation never leave RAM.

2.1 Forming the graph

Consider the following recursion describing the horizontal motion of a ball under the action of friction:

$$s_{k+1} = s_k + v_k, \quad k = 0, \ldots, N-1 \tag{1}$$

$$v_{k+1} = v_k - v_k * v_k \tag{2}$$

For given s_0, v_0 and N, this recursion defines a function $f : \mathbf{R}^2 \rightarrow \mathbf{R}^2$:

$$[s_N, v_N] = f(s_0, v_0; N) \tag{3}$$

Figure 1 shows the DAG, also referred to as the *computational graph* in automatic differentiation terminology [5], of this expression when $N = 2$.

A DAG like this can easily be constructed in an object-oriented programming language like C++. The basic building block can be an object that contains the elementary op-

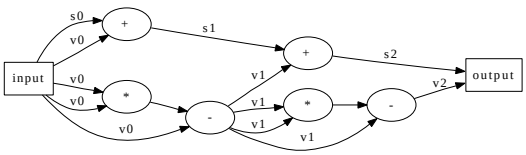

Figure 1. The DAG for the ball example ($N = 2$)

[1] when multiple directions (for the forward mode) or multiple outputs (for the reverse mode) are involved, the process must be repeated for each direction (output), unless the Jacobian has some special structure

eration to be performed as well as references to its dependent nodes. Since a node can have an arbitrary number of nodes pointing to it, we advocate the use of smart pointers (or, more precisely *shared pointers*) in order to be able to keep track of the ownerships of the nodes [2].

More complex functions, which similarly to the ball example arise after parametrization of an optimal control problem may have millions of nodes and it is important to find efficient ways to form and manipulate such graphs.

2.2 Topological sorting

Before a function can be evaluated, we have to order the nodes of the DAG in the order of dependencies, known as finding a *topological sorting* in graph theory. This step is not necessary in AD by operator overloading, since a good linear ordering can be recorded during the process of constructing the graph. In AD by source code transformation, the compiler is responsible for finding this ordering.

A topological ordering can however also be found cheaply after the DAG has been formed, which greatly facilitates the implementation. We propose to use a modified *breadth-first search*, described in the following, to obtain a good topological ordering. Our implementation of this sorting depends on another sorting, the *depth-first search*, which we will desribe first.

2.2.1 Depth-first search

One way of finding a linear ordering is by so-called *depth-first search* [6], which can be implemented in the following way, where top$(\cdot)$ refers to the topmost element of a *stack*[2]:

- Add the output nodes to a stack S
- Create an empty list A for the linear ordering
- While S is nonempty
  - If top(S) has not yet been visited
    - Mark the top(S) as visited
    - Add non-visited nodes to S on which top(S) depends
  - else
    - Add top(S) to A
    - Remove top(S) from S

The algorithm visits each node at most one time and has thus linear complexity in the number of nodes. Figure 2 shows an ordering obtained by this algorithm for the ball example. The example illustrates two problems with this sorting. Firstly, one needs to store v_1 in memory until we calculate s_2. Similarly, for a large N, we would need to keep $v_1, \ldots, v_{N-1}$ in memory at the time we evaluate v_N, since they will be needed later.

Secondly, several operations could be done in parallel, e.g. operation $\{1\}$ and $\{5\}$, in the graph.

2.2.2 Breadth-first search

These problems lead us to look at another ordering, namely the *breadth-first search* [6]. The standard algorithm for this ordering cannot readily be applied to the graph, since we are not able to iterate over *the nodes that depend on a given node*, but only over *the nodes that a given node depends on*. This can, however, be solved with the following algorithm:

- Find a topological sorting A by a depth-first search
- Create a vector of dependency levels L. An operation associated with one level depends only on the results from previous levels and can thus be evaluated independently. Constants and inputs have level 0.
- For $i = \text{begin}(A), \ldots, \text{end}(A)$
  - Find $l_{\text{max},i}$, the maximum level of any of i's dependent nodes
  - Assign level$(i) := 1 + l_{\text{max},i}$
- Sort the nodes by their level using a *bucket sort*

The result of the sorting, for the ball example, is shown in Figure 3. Since all nodes of one level can be evaluated independently of each other, the sorting is suitable for parallelization. Like the depth-first search, this algorithm will run in linear time.

A problem with the breadth-first search is that it tends to evaluate nodes earlier than they are actually needed. For example, when the expression is a simple sum of squares, $f(x_1, \ldots, x_N) = x_1^2 + \ldots + x_N^2$, all multiplications will take place on level 1, creating unnecessary memory overhead. We can solve this by iterating over the levels in reverse order and "move up" dependent nodes as much as possible. This operation has also linear complexity.

3. Software implementation

The proposed algorithms have been been implemented in the open-source C++ tool CasADi, which will be released under the LGPL licence. A stated goal of the tool has been to keep the data structures as transparent as possible to allow a user to easily extend the code with methods from the field of computer algebra, as well as numerical optimization.

The focus of the code is to generate highly efficient runtime code and for this purpose we propose to use a combination of two different DAG representations, one DAG which is restricted to built-in binary (and unary) scalar functions and a general one representing a matrix syntax tree of sparse matrix operations as well as nodes corresponding to switches, loops, function evaluations, element access and concatenation, similar to the graph representation used in a modelling language such as Modelica.

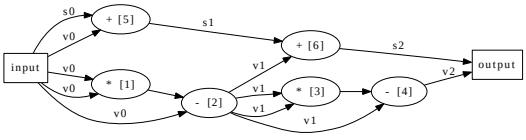

Figure 2. Linear ordering (in brackets) from a depth-first search for the ball example

[2] with stack is meant a *last in, first out* data structure

3.1 A scalar DAG representation

The purpose of the scalar DAG representation is to represent and evaluate an algorithm containing a series of elementary operations (`+`,`-`,`*`,`/`, `*`) and built-in functions from the C's `math.h` library (`floor`, `pow`, `sqrt`, etc.). No branching (if-statements) is allowed. With these restrictions, it is possible for the AD algorithm to access memory in a strictly sequential manner, also for the reverse AD algorithm, which is the reason that existing AD tools usually *only* allows operations of this form [9].

With these restrictions, the class hierarchy simply becomes, with the most important members in brackets:

- Expression class, `SX` [pointer to an `SXNode`]
- Abstract base class for all nodes, `SXNode` [reference counter]
  - Symbolic scalar [name of the variable]
  - Constant [value of the constant]
  - Binary Node [two `SX` instances, index of a binary function]

The graph is represented by a smart pointer class, called `SX`, and a polymorphic node class consisting of an abstract base class and three derived classes, a node corresponding to a constant node, a node corresponding to a variable and a node corresponding to a binary operation. We have gathered all binary operations in a single node rather than deriving a class for each binary operation. The binary nodes, in turn, has members of the class `SX`, correspond to its dependent nodes. This simple representation is flexible enough to be able to construct trees of millions of variables in C++. Using the topological sorting outlined in the previous section, we obtain a linear ordering of the nodes equivalent to the *trace* of automatic differentiation tools [9].

3.2 A matrix DAG representation

The scalar DAG representation is designed to be very efficient for code made up entirely by standard unary and binary operations. In a well-designed code, the lion's share of the calculation time should indeed in sections of this type, so it makes sense to specialize the solver to be efficient in this case.

To be able to represent more general functions, we propose to use a second, much more general, DAG representation. In this representation, the graph is made up by, possibly sparse, *matrix operations* and we include nodes for elementwise operations as well as operations such as matrix products. With this we can also represent a much wider range of operations such as vertical and horizontal concatenation, element access, function evaluation, loops

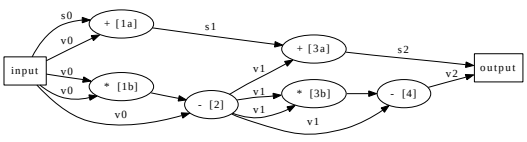

Figure 3. Linear ordering (in brackets) from a breadth-first search for the ball example

(for, while) and switches (if-statements, etc.). This is much more general than is allowed by AD tools, which generally freezes all the if-statements during the *tracing* step and inlines all loops to create a representation similar to the scalar DAG in the previous section. With the proposed approach, we instead leave it to the user to decide which loops and function calls actually should be inlined. By partially inlining the code, we arrive with an approach which is equivalent to the *checkpointing schemes* used by standard AD tools [9].

We node that both forward and adjoint AD readily generalizes to work with graphs of this form. Many operations, like element access, just translate to linear matrix operations (which are often sparse). Other operations, such as matrix products and function evaluations, have explicit chain rules defined for them.

When evaluating a function represented by the Matrix DAG on a multi-core (shared memory) architecture, independent nodes, as obtained from the breadth-first-search, can be evaluated in parallel (multiple threads) using OpenMP.

The implementation of the matrix expression class, `MX`, is similar to the `SX` class above with the following differences.

- The Binary Node class becomes polymorphic, with one derived node for each type of operation. This makes it possible for the binary nodes to contain more information, such as pointers to functions or more than two arguments.
- The *MX* class contains arrays for the evaluation, and the evaluation takes place by looping over a vector of pointers to the nodes, rather than in a separate data structure.

4. Numerical tests

4.1 An AD example: determinant calculation

We first test the algorithm on an AD benchmark, namely the calculation of the adjoint derivivative of the determinants of matrices of different sizes. The determinant is calculated by expansion along rows, an exponentially expensive calculation.

This example is implemented in the speed benchmark collection of CppAD. We use this implementation to test the performance of CasADi against CppAD as well as ADOL-C, [3]. Figure 4 shows the speed, in number of solves per second, for the three tools for matrices of sizes ranging from 1-by-1 to 9-by-9. The tests 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. The operator system is Linux.

We use the latest version of ADOL-C at the time of writing, version 2-1-5, and the latest version of CppAD, released 17 June 2010.

For the current test, CasADi outperforms the CppAD and ADOL-C for sizes up to 8-by-8 (corresponding to some 100.000 elementary operations), but we want to stress that

the test only covers one single example and we make no claims that these results will hold generally. More tests are needed to better assess the performance of the solver.

4.2 Dynamic optimization example: minimal fuel rocket flight

We then study an example from optimal control, namely the minimal fuel flight of a rocket described by the following continuous time model:

$$\dot{s} = v \tag{4}$$
$$\dot{v} = (u - \alpha\, v^2)/m \tag{5}$$
$$\dot{m} = -\beta\, u^2 \tag{6}$$
$$\tag{7}$$

We assume that the rocket starts at rest at ($s(0) = 0$, $v(0) = 0$) with mass $m(0) = 1$. We simulate for $T = 10$ seconds and require that the rocket lands at rest ($v(T) = 0$) at $s(T) = 10$. The optimization problem is to minimize the fuel consumption $m(0) - m(T)$.

4.2.1 Solution approach

We discretize the control into $N_u = 1000$ piecewise constant controls and solve the problem using a *single-shooting approach*. The single-shooting approach requires a sensitivity-capable integrator which we can easily construct in CasADi with a few lines of code using an *explicit Euler approach* with 1000 steps per control interval of equal length (we wish to stress that there are certainly much better ways of solving this optimal control problem, but our purpose here is only to illustrate our AD approach).

We show two different ways of solving the problem using CasADi, with scalar graphs and a combination of scalar and matrix graphs (next section). In both approaches we arrive at a non-linear programming problem (NLP) [14] of the form:

$$\begin{array}{cc} \text{minimize:} & \sum_{i=0}^{999} u_i^2 \\ u_0, \ldots, u_{999} & \\ \text{subject to:} & g(u) = 0, \\ & -10 \leq u \leq 10 \end{array} \tag{8}$$

where $g(u) : \mathbf{R}^{1000} \rightarrow \mathbf{R}^2$ is a non-linear function that we shall construct. This NLP is then solved by the non-linear optimization code Ipopt [16], which requires not only function evaluation, but also the gradient of the objective function and the Jacobian of the constraints. We use CasADi to obtain this information. Since there are 1000 variables but only two constraints, it makes sense to use the adjoint mode AD to calculate the Jacobian of g.

4.2.2 Using scalar graphs

In the first approach, we use two nested for loops to calculate one large graph with around 13 million nodes (13 being the number of elementary operations in each step). This is the approach taken by default by existing AD tools. In the CasADi notation, we get:

```
SX s = 0, v = 0, m = 1;
for(int k=0; k<1000; ++k){
  for(int j=0; j<1000; ++j){
    s += dt*v;
    v += dt / m * (u[k] - alpha * v*v);
    m += -dt * beta*u[k]*u[k];
  }
}
```

where SX is the name of the symbolic expression type used in CasADi, which can be used in the same way as C/C++ `double`. We assume that the input u is stored in a vector of symbolic variables of length 1000. After this recursion, the expression for g is then simply obtained as $g(u) = [s - 10; v]$.

Ipopt requires 11 iterations to solve this problem and the total solution time was 19.6 seconds out of which 7.8 seconds was needed for the function evaluations (the lion's share of the rest being needed to form and sort the graphs). A single function evaluation, corresponding to around 13 million elementary operations, takes about 0.27 seconds, or around 20 ns per elementary operation.

4.2.3 Using scalar and matrix graphs

The approach above is basically to *inline everything* and it is clear that the graphs will soon become too large. We therefore show a second way to represent the same function based on a two level approach. We first use the above approach over an interval with a constant control only:

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
SX u("u");
SX s = s_0, v = v_0, m = m_0;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u[k] - alpha * v*v);
  m += -dt * beta*u*u;
}
```

which we use to generate an function (an *integrator*) with some 13 thousand nodes (instead of 13 million). This function will take as input (s_0,v_0,m_0 and u) and return the three outputs (s,v,m). We then evaluate this function 1000 times using out matrix graph representation:

```
MX X = {0,0,1}; // initial value
for(int k=0; k<1000; ++k){
  // Integrate
  vector<MX> input = {U[k],X};
  X = integrator(input);
}
```

where MX is the name of CasADi's matrix expression class. The second loop will be represented by a graph with about 2000 nodes, 1000 "element access" nodes (`U[k]`) and 1000 "function evaluation" nodes. For Ipopt, both approaches are equivalent, they are simply two ways of calculating the same function g, so the optimization results are indeed identical. The significantly lower memory need in the second approach, however, enables us to construct much larger problems (e.g. taking 100 times more steps) without running out of memory.

The solution of the problem using the scalar and matrix graph combination took 10.4 seconds, out of which 9.7

seconds was needed for the function and derivative evaluation. The example clearly shows how this approach makes forming the graphs significantly faster (less than one second instead of 11), but in terms of solution times for a single function evaluation, the scalar graph approach is still significantly quicker, as it requires less operations.

4.3 A Modelica example: Van der Pol oscillator

In the third example, we will use CasADi as an interface between two optimization tools, the Optimal control package ACADO Toolkit [11] and the Modelica-compiler JModelica [1]. We wish to solve the following optimal control problem describing a Van der Pol oscillator:

$$\begin{array}{ll} \text{minimize:} & \\ x_1(\cdot), x_2(\cdot), u(\cdot) & \displaystyle\int_0^{20} e^{p_3}\left(x_1^2 + x_2^2 + u^2\right) dt \\ & \\ & \dot{x_1} = (1 - x_2^2)\, x_1 - x_2 + u \\ \text{subject to:} & \dot{x_2} = p_1\, x_1 \\ & x_1(0) = 0, \quad x_2(0) = 1 \\ & u \leq 0.75 \end{array} \tag{9}$$

where the p_0, p_1, p_2 are parameters and $x_1(\cdot)$ and $x_2(\cdot)$ are state variables (time dependent) and $u(\cdot)$ is a control.

The example is taken from the JModelica benchmark collection and has been implemented there in the Optimica extension of Modelica [2]. We use the newly added XML export functionality of JModelica to export the optimal control problem in a fully symbolic form. This XML code is then parsed in CasADi using the XML parser TinyXML [15].

We use the optimal control package ACADO Toolkit to solve the optimal control problem coupled to CasADi for evaluating the objective function, the ODE right hand side of their and derivatives. ACADO Toolkit uses a multiple-shooting methods to discretize the optimal control problem to a non-linear program (NLP) and then solves the NLP using an Sequential Quadratic Programming (SQP) method [14]. Using a limited memory Hessian approximation and initialized with an zero control, 26 iterations were needed to solve the problem.

Figure 5 shows the state and control trajectories for the optimal solution as obtained by the tool coupling. The results agree with those obtained by JModelica's built-in optimal control problem solver, which is based on direct collocation [2].

5. Conclusions and outlook

The directed graph is a natural way of representing a non-linear function and this formulation can be used not only to formulate an optimal control problem, but also to reformulate the problem into the canonical form used by current state-of-the-art solvers for large-scale optimization problem. Automatic differentiation, both in forward and in reverse mode, can be implemented efficiently directly on the graph instead of taking the detour over generating C-code and then using an existing AD tool for iteratively getting generating linear orderings corresponding to different branches.

We have presented CasADi, an AD tool using a function representation borrowed from the field of computer algebra, also found in object oriented modelling languages, and certainly much richer than that of most existing AD tools. The tool has been coupled to optimal control software ACADO Toolkit, the nonlinear programming solver Ipopt [16] as well as the CVodes of the Sundials suite [10], but we want to stress that CasADi is *not* intended to be just an interface between optimization tools and modelling environments. The idea is instead to actually implement the optimization algorithms using the graph representation and exploit the structure as much as possible.

The main scope of the tool thus starts off where current tools, e.g. JModelica, generate C-code to be used in a numerical solver. Using the graph representation, it is possible to step-by-step reformulate the infinite dimensional, possibly non-smooth OCP into a nonlinear problem (NLP), even going as far as to even solve the NLP. The latter is of particular interest if we wish to generate code for a, say, nonlinear model predictive control to be used on embedded systems.

Since the graphs after parameterizing states as well as controls will be significantly larger than the graphs needed to represent the optimal control problem, it makes sense to have the graph constructed in a language such as C++, rather than, say, Java or a scripting language such as Python. Given the excellent possibilities to interface C++ to other languages, this should not be an issue.

Acknowledgments

This research 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).

References

[1] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a modelica compiler using jastadd attribute grammars. *Science of Computer Programming*, 75(1-2):21 – 38, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[2] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2008.

[3] Andreas Griewank and David Juedes and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.

[4] Lorenz T. Biegler. An overview of simultaneous strategies for dynamic optimization. *Chemical Engineering and Processing: Process Intensification*, 46(11):1043–1053, 2007. Special Issue on Process Optimization and Control in Chemical Engineering and Processing.

[5] Christian H. Bischof and Paul D. Hovland and Boyana Norris. Implementation of automatic differentiation tools (invited talk). *j-SIGPLAN*, 37(3):98–107, March 2002. Proceedings of the 2002 ACM SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02).

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[7] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008.

[8] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `http://cvxr.com/cvx`, May 2010.

[9] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. SIAM, 2008.

[10] 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*, 2005.

[11] Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. Acado toolkit - an open-source framework for automatic control and dynamic optimization. *Optimal Control Methods and Application*, 2010.

[12] Jan Albersmeyer and Moritz Diehl. The Lifted Newton Method and Its Application in Optimization. *SIAM J. Optim*, 20(3):1655–1684, 2010.

[13] Monagan, Michael B. and Neuenschwander, Walter M. GRADIENT: algorithmic differentiation in Maple. In *ISSAC '93: Proceedings of the 1993 international symposium on Symbolic and algebraic computation*, pages 68–76, New York, NY, USA, 1993. ACM.

[14] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, August 2000.

[15] Lee Thomason, Yves Berquin, and Andrew Ellerton. Tinyxml, version 2.6.0. `http://www.grinninglizard.com/tinyxml/`, May 2010.

[16] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

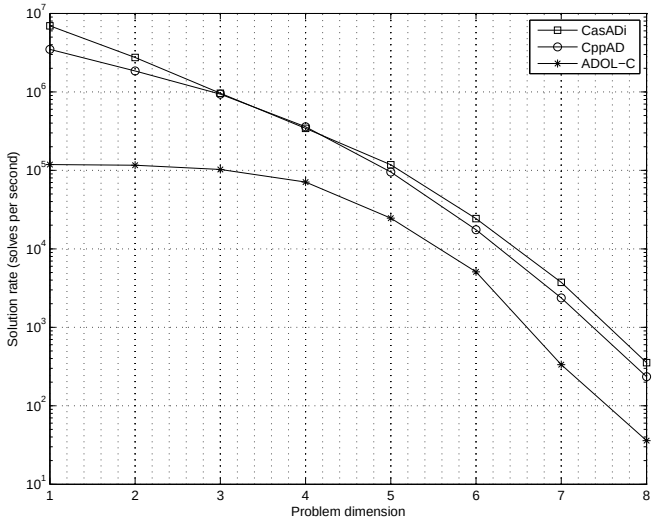

Figure 4. AD benchmark test: determinant by minor expansion

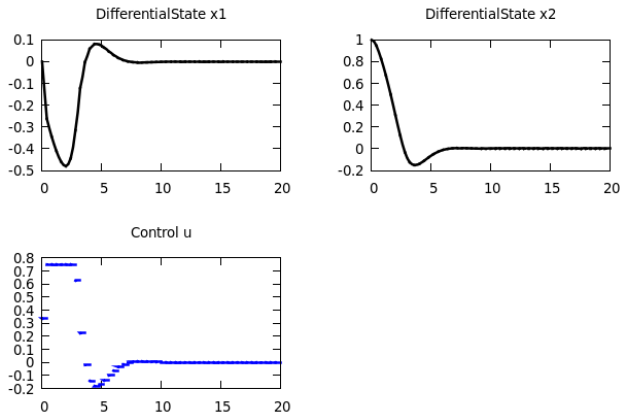

Figure 5. The optimal state and control trajectories for the Van der Pol oscillator example