Mathematical Programming Computation manuscript No.
(will be inserted by the editor)

CasADi – A software framework for nonlinear optimization and optimal control

Joel A. E. Andersson · Joris Gillis · Greg Horn · James B. Rawlings · Moritz Diehl

Received: date / Accepted: date

Abstract We present CasADi, an open-source software framework for numerical optimization. CasADi is a general-purpose tool that can be used to model and solve optimization problems with a large degree of flexibility, larger than what is associated with popular algebraic modeling languages such as AMPL, GAMS, JuMP or Pyomo. Of special interest are problems constrained by differential equations, i.e. optimal control problems. CasADi is written in self-contained C++, but is most conveniently used via full-featured interfaces to Python, MATLAB or Octave. Since its inception in late 2009, it has been used successfully for academic teaching as well as in applications from multiple fields, including process control, robotics and aerospace. This article gives an up-to-date and accessible introduction to the CasADi framework, which has undergone numerous design improvements over the last seven years.

Keywords Optimization · Optimal control · Open source optimization software

Mathematics Subject Classification (2000) 90C99 · 93A30 · 97A01

J. A. E. Andersson
Department of Chemical and Biological Engineering, University of Wisconsin-Madison, USA
E-mail: jandersson@wisc.edu

J. Gillis
Production Engineering, Machine Design and Automation (PMA) Section, KU Leuven, Belgium
E-mail: joris.gillis@kuleuven.be

G. Horn
Kitty Hawk, Mountain View, USA
E-mail: greg@kittyhawk.aero

J. B. Rawlings
Department of Chemical and Biological Engineering, University of Wisconsin-Madison, USA
E-mail: james.rawlings@wisc.edu

M. Diehl
Department of Microsystems Engineering IMTEK, University of Freiburg, Germany
E-mail: moritz.diehl@imtek.uni-freiburg.de

1 Introduction

CasADi is an open-source software for numerical optimization, offering an alternative to conventional algebraic modeling languages such as AMPL [50], Pyomo [69] and JUMP [40]. Compared to these tools, the approach taken by CasADi, outlined in this paper, is more flexible, but also lower-level, requiring an understanding of the *expression graphs* the user is expected to construct as part of the modeling process.

This flexible approach is in particular valuable for problems constrained by differential equations, introduced in the following.

1.1 Optimal control problems

Consider the following basic optimal control problem (OCP) in ordinary differential equations (ODE):

$$\begin{aligned} \underset{x(\cdot),u(\cdot),p}{\text{minimize}} \quad & \int_0^T L(x(t),u(t),p)\,dt + E(x(T),p) \\ \text{subject to} \quad & \left.\begin{aligned} &\dot{x}(t) = f(x(t),u(t),p), \\ &u(t) \in \mathscr{U}, \quad x(t) \in \mathscr{X}, \end{aligned}\right\} \quad t \in [0,T] \\ & x(0) \in \mathscr{X}_0, \quad x(T) \in \mathscr{X}_T, \quad p \in \mathscr{P}, \end{aligned} \tag{OCP}$$

where $x(t) \in \mathbb{R}^{N_x}$ is the vector of (differential) states, $u(t) \in \mathbb{R}^{N_u}$ is the vector of free control signals and $p \in \mathbb{R}^{N_p}$ is a vector of free parameters in the model. The OCP here consists of a Lagrange term (L), a Mayer term (E), as well as an ODE with initial ($\mathscr{X}_0$) and terminal ($\mathscr{X}_\mathrm{f}$) conditions. Finally, there are admissible sets for the states ($\mathscr{X}$), control ($\mathscr{U}$) and parameters ($\mathscr{P}$). For simplicity, all sets can be assumed to be simple intervals.

Problems of form (OCP) can be efficiently solved with the *direct approach*, where (OCP) is transcribed into a nonlinear program (NLP):

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & J(w) \\ \text{subject to} \quad & g(w) = 0, \quad w \in \mathscr{W}, \end{aligned} \tag{NLP}$$

where $w \in \mathbb{R}^{N_w}$ is the decision variable, J the objective function and $\mathscr{W}$ again taken to be an interval set.

Popular direct methods include *direct collocation* [96,97], which reached widespread popularity through the work of Biegler and Cuthrell [21,31], and *direct multiple shooting* by Bock and Plitt [23,22]. Both these methods exhibit good convergence properties and can be easily parallelized.

Real-world optimal control problems are often significantly more general than (OCP). Industrially relevant features include multi-stage formulations, i.e., different dynamics in different parts of the time horizon, robust optimal control, problems with integer variables, differential-algebraic equations (DAEs) instead of ODEs, and multipoint constraints such as periodicity.

1.2 Scope of CasADi

CasADi started out as a tool for algorithmic differentiation (AD) using a syntax similar to a computer-algebra system (CAS), explaining its name. While state-of-the-art AD is still a key feature of CasADi, the focus has since shifted towards optimization. In its current form, CasADi provides a set of general-purpose building blocks that drastically decreases the effort needed to implement a large set of algorithms for numerical optimal control, without sacrificing efficiency. This "toolkit design" makes CasADi suitable for teaching optimal control to graduate-level students and allows researchers and industrial practitioners to write codes, with a modest programming effort, customized to a particular application or problem structure.

1.3 Organization of the paper

The remainder of the paper is organized as follows. We start by introducing the reader to CasADi's symbolic core in Section 2. The key property of the symbolic core is a state-of-the-art implementation of AD. Some unique features of CasADi are introduced, including how sparse matrix-valued atomic operations can be used in a source-code-transformation AD framework.

In Section 3, we show how systems of linear or nonlinear equations, as well as initial-value problems in ODEs or DAEs, can be embedded into symbolic expressions, while maintaining differentiability to arbitrary order. This automatic sensitivity analysis for ODEs and DAEs is, to the best knowledge of the authors, a unique feature of CasADi.

Section 4 outlines how certain optimization problems in canonical form can be solved with CasADi, including nonlinear programs (NLPs), linear programs (LPs) and quadratic programs (QPs), potentially with a subset of the variables confined to integer values, i.e. mixed-integer formulations. CasADi provides a common interface for formulating such problems, while delegating the actual numerical solution to a third-party solver, either free or commercial, or to an in-house solver, distributed with CasADi.

Section 5 consists of a tutorial, showing some basic use cases of CasADi. The tutorial mainly serves to illustrate the syntax, scope and usage of the tool.

Finally, Section 6 gives an overview of applications where CasADi has been successfully used to date before Section 7 wraps up the paper.

2 Symbolic framework

The core of CasADi consists of a symbolic framework that allows users to construct expressions and use these to define automatically differentiable functions. These general-purpose expressions have no notion of optimization and are best likened with expressions in e.g. MATLAB's Symbolic Toolbox or Python's SymPy package. Once the expressions have been created, they can be used to efficiently obtain new expressions for derivatives using AD or be evaluated efficiently, either in CasADi's virtual machines or by using CasADi to generate self-contained C code.

We detail the symbolic framework in the following sections, where we also provide MATLAB/Octave and Python code snippets[1] corresponding to CasADi 3.1 below in order to illustrate the functionality. For a self-contained and up-to-date walkthrough of CasADi's syntax, we recommend the user guide [14].

2.1 Syntax and usage

CasADi uses a MATLAB inspired "everything-is-a-matrix" type syntax, i.e., scalars are treated as 1-by-1 matrices and vectors as n-by-1 matrices. Furthermore, all matrices are *sparse* and stored in the compressed column format. For a symbolic framework like CasADi, working with a single sparse data type makes the tool easier to learn and maintain. Since the linear algebra operations are typically called only once, to construct symbolic expressions rather than to evaluate them numerically, the extra overhead of e.g. treating a scalar as a 1-by-1 sparse matrix is negligible.

The following code demonstrates loading CasADi into the workspace, creating two symbolic primitives $x \in \mathbb{R}^2$ and $A \in \mathbb{R}^{2\times 2}$ and finally the creation of an expression for $e := A\sin(x)$:

```
% MATLAB/Octave
import casadi.*
x = SX.sym('x', 2);
A = SX.sym('A', 2, 2);
e = A * sin(x);
disp(e)
```

```
# Python
from casadi import *
x = SX.sym('x', 2)
A = SX.sym('A', 2, 2)
e = mtimes(A, sin(x))
print(e)
```

Output:
```
@1=sin(x_0), @2=sin(x_1),
[((A_0*@1)+(A_2*@2)), ((A_1*@1)+(A_3*@2))]
```

The output should be interpreted as the definition of two shared subexpressions, $@1 := \sin(x_0)$ and $@2 := \sin(x_1)$ followed by an expression for the resulting column vector (2-by-1 matrix). The fact that CasADi expressions are allowed to contain shared subexpressions is essential to be able to solve large-scale problems and for CasADi to be able to implement AD as described in Section 2.6.

2.2 Graph representation – Scalar expression type

In the code snippet above, we used CasADi's *scalar expression* type - `SX` – to construct a symbolic expression. Scalar in this context does not refer to the type itself – `SX` is a general sparse matrix type – but the fact that each nonzero element is defined by a sequence of scalar-valued operations. CasADi uses the compressed column storage (CCS) [12] format to store matrices, the same used to represent sparse matrices in MATLAB, but with the difference that in CasADi entries are allowed to be structurally nonzero but numerically zero as illustrated by the following:

[1] One may run these snippets on a Windows, Linux or Mac platform by following the install instructions for CasADi 3.1 at `http://install31.casadi.org/`

```
% MATLAB/Octave
C = SX.eye(2);
C(1,1) = SX.sym('x');
C(2,1) = 0;
disp(C)
```

```
# Python
C = SX.eye(2)
C[0,0] = SX.sym('x')
C[1,0] = 0
print(C)
```

Output:
```
[[x, 00],
 [0, 1]]
```

Note the difference between structural zeros (denoted 00) and numerical zeros (denoted 0). The fact that symbolic matrices are always sparse in CasADi stands in contrast to e.g. MATLAB's Symbolic Math Toolbox, where expressions are always dense. Also note that CasADi has indices starting with 1 in MATLAB/Octave, but 0 in Python. In C++, CasADi also follows the index-0 convention.

When working with the `SX` type, expressions are stored as a directed acyclic graph (DAG) where each node – or *atomic operation* – is either:

- A symbolic primitive, created with SX.sym as above
- A constant
- A unary operation, e.g. sin
- A binary operation, e.g. $*$, $+$

This relatively simple graph representation is designed to allow numerical evaluation with very little overhead, either in a virtual machine (Section 2.4) or in generated C code (Section 2.5). Each operation also has a chain-rule that can efficiently be expressed with the other atomic operations.

2.3 Graph representation – Matrix expression type

There is a second expression type in CasADi, the *matrix expression* type – `MX`. For this type, each operation is a matrix operation; an expression such as $A+B$ where A and B are n-by-m matrices would result in a single *addition* operation, in contrast to up to mn scalar addition operations using the `SX` type. In the most general case, an `MX` operation can have multiple matrix-valued inputs and return multiple matrix-valued outputs.

The choice to implement two different expression types in CasADi – and expose them both to the end user – is the result of a design compromise. It has proven difficult to implement an expression type that works efficiently both for e.g. the right-hand-side of an ODE, where minimal overhead is critical, and at the same time be able to represent the very general symbolic expressions that make up the NLP resulting from a direct multiple shooting discretization, which contains embedded ODE integrators.

The syntax of the `MX` type mirrors that of `SX`:

```
% MATLAB/Octave
x = MX.sym('x', 2);
A = MX.sym('A', 2, 2);
e = A * sin(x);
disp(e)
```

```
# Python
x = MX.sym('x', 2)
A = MX.sym('A', 2, 2)
e = mtimes(A, sin(x))
print(e)
```

Output: `mac(A,sin(x),zeros(2x1))`

The resulting expression consists of two matrix valued symbolic primitives (for A and x, respectively), 2-by-1 all-zero constant, a unary operation (sin) and a matrix multiply-accumulate operation, $\texttt{mac}(X_1, X_2, X_3) := X_3 + X_1 X_2$.

The choice of atomic operations for the `MX` type was made so that derivatives calculated either using the forward or reverse mode of algorithmic differentiation can be efficiently expressed using the same set of operations. The choice also takes into account that CasADi's `MX` virtual machine, cf. Section 2.4, supports *in-place* operations. This last fact explains why matrix multiply-accumulate was chosen as an atomic operation instead of the standard matrix multiplication; in practice, the operation performed is $X_3 := X_3 + X_1 X_2$.

A list of the most important atomic operations for the `MX` expression graph can be found in Table 1. The list also shows how the different atomic operations are inter-dependent under algorithmic differentiation operations. For example, reverse mode AD performed on the operation to retrieve an element of a matrix (operation 9 in the table) results in an operation to *assign* a quantity to a matrix element (operation 10 in the table). Note that the assignment operation is a two step procedure consisting of copying the existing matrix into a new variable before the actual assignment (or optionally, addition) takes place. The copy operation is typically eliminated in the virtual machine, cf. Section 2.4.

Some operations, e.g., the binary operation $A+B$, assume that the arguments have the same sparsity pattern (i.e., the same location of the nonzero elements). If this is not the case, CasADi inserts "projection" nodes into the expression graph during the construction.

A special type of atomic operation is a *function call* node. It consists of a call to a function object created at runtime. Importantly, there can be multiple calls to the same function object, which can keep the size of the expression graphs small. Function object are covered in the following.

2.4 Function objects and virtual machines

The symbolic expressions in CasADi can be used to define *function objects*, class instances that behave like conventional functions but are created at runtime, cf. [11]. In addition to supporting numerical evaluation, CasADi function objects support symbolical evaluation, C code generation (Section 2.5) as well as derivative calculations (Section 2.6). They can be created by providing a display name and a list of input and output expressions:

Table 1 Selected atomic operations for CasADi's `MX` type

	Operation	Definition or example	Interdependencies	
			AD forward	AD reverse
1	Constant	$y = 0,\ \ y = [1.2,4]^{\mathrm{T}}$ etc.		
2	Unary operation	$Y = \sin(X),\ \ Y = \sqrt{X}$ etc.		
3	Binary operation	$Y = X_1 + X_2,\ \ Y = X_1 * X_2$ etc.		4 [a]
4	Inner product	$y = \mathrm{tr}({X_1}^{\mathrm{T}} X_2)$ [b]		3
5	Transpose	$Y = X^{\mathrm{T}}$		
6	Multiply-accumulate	$Y = X_3 + X_1 X_2$		3
7	Reshape	(changes dimension)		
8	Projection	(changes sparsity)		
9	Submatrix access	$y = X(i, j)$		10
10	Submatrix assignment	$Y(i, j) = x_2$ with $Y = X_1$ elsewhere		9
11	Linear solve	$y = {X_2}^{-1} x_1$ or $y = {X_2}^{-\mathrm{T}} x_1$	5,6,12,13	5,6,12,13
12	Horizontal concatenation	$Y = [x_1 \ldots x_n]$		13
13	Horizontal split	$\{y_1, \ldots, y_n\}$ such that $[y_1 \ldots y_n] = X$		12
14	Function call	(see text)		

[a] Dependency appears for e.g. $x_1 * X_2$, where x_1 is a scalar and X_2 is a matrix
[b] Efficiently evaluated by elementwise multiplication and summation, $\mathrm{sum}(X_1 * X_2)$

```
% MATLAB/Octave                    # Python
F=Function('F',{x,A},{e});         F=Function('F',[x,A],[e])
```

which defines a function object with the display name "F" with two inputs (x and A) and one output (e), as defined in the previous code segments. Function objects can have an arbitrary number of inputs and outputs, each of which is a sparse matrix. Should an input – e.g. A above – contain structural zeros, the constructed function is understood not to depend on the corresponding part of the matrix.

The creation of a function object in CasADi essentially amounts to topologically sorting the expression graph, turning the directed acyclic graph (DAG) into an *algorithm* that can be evaluated. Unlike traditional tools for AD such as ADOL-C [62] or CppAD [4], there is no relation between the order in which expressions were created (i.e. a *tracing step*) and the order in which they appear in the sorted algorithm. Instead, CasADi uses a *depth-first search* to topologically sort the nodes of the DAG.

Given the sorted sequence of operations, CasADi implements two *register based virtual machines* (VMs), one for each graph representation. For the inputs and outputs of each operation, we assign an element (`SX` VM) or an interval (`MX` VM) from a work vector. This design contrasts to the *stack based* VM used in e.g. the AMPL Solver Library [50]. To limit the size of the work vector, the *live variable range* of each operation is analyzed and work vector elements or intervals are then reused in a last-in, first-out manner. When possible, operations are performed *in-place*, e.g., an operation for assigning the top left element of a matrix:

$$Y(i,j) = \begin{cases} x_2 & \text{if } i = 1 \text{ and } j = 1 \\ X_1(i,j) & \text{otherwise} \end{cases}$$

typically simplifies to just an element assignment, $X_1(1,1) := x_2$, assuming that X_1 is not needed later in the algorithm.

Since MX expression graphs can contain calls to function objects, it is possible to define nested function objects. Encapsulating subexpressions used multiple times into separate function objects allows expression graphs to stay small and has implications for the memory use in the context of algorithmic differentiation, cf. Section 2.6.

Function objects in CasADi, as represented by the `Function` class, need not be defined by symbolic expressions as above. ODE/DAE integrators, solvers of nonlinear systems of equations and NLP solvers are examples of function objects in CasADi that are not explicitly defined by symbolic expressions. In many, but not all cases, these function objects are also automatically differentiable. We return to these classes in the following sections.

2.5 C code generation and just-in-time compilation

The VMs in CasADi are designed for high-speed and low overhead e.g. by avoiding memory allocation during numerical evaluation. In a framework such as CasADi, which is frequently used for rapid prototyping with many design iterations, fast VMs are important not only for numerical evaluation, but also for symbolic processing, which can make up a significant portion of the total solution time.

An alternative way to evaluate symbolic expressions in CasADi is to generate C code for the function objects. When compiled with the right compiler flags, the generated code can be significantly faster than CasADi's VMs. Since the generated code is self-contained C and has no dynamic memory allocation, it is suited to be deployed on embedded systems. The generated code is designed to be linked into a dynamically linked library, for static/dynamic linking (via a generated header file), to be called from the OS command-line (via a generated `main` entry point), or be called from MATLAB/Octave (via a generated `mexFunction` entry point).

The generated C code can be used for *just-in-time* compilation, which is supported either by using the system compiler or via an interface to the LLVM compiler framework with its C/C++ front-end Clang [81]. The latter is available for all platforms and is distributed with CasADi, meaning that the user does not need to have a binary compatible system compiler in order to use the generated C code.

2.6 Algorithmic differentiation

Algorithmic differentiation (AD) – also known as automatic differentiation – is a technique for efficiently calculating derivatives of functions represented as algorithms. For a function $y = f(x)$ with vector-valued x and y, the *forward mode* of AD provides a way to accurately calculate a Jacobian-times-vector product:

$$\hat{y} := \frac{\partial f}{\partial x}\hat{x}, \tag{1}$$

at a computational cost comparable to evaluating the original function $f(x)$. This definition naturally extends to a matrix-valued function $Y = F(X)$, by simply defining $x := \mathrm{vec}(X)$ and $y := \mathrm{vec}(Y)$, where $\mathrm{vec}(\cdot)$ denotes stacking the columns of the matrix

vertically. It also naturally generalizes further to the case when there are multiple matrix-valued inputs and outputs, which is the general case for functions in CasADi.

The *reverse mode* of AD, on the other hand, provides a way to accurately calculate a Jacobian-transposed-times-vector product:

$$\bar{x} := \left(\frac{\partial f}{\partial x}\right)^{\mathrm{T}} \bar{y} \tag{2}$$

also at a computational cost comparable to evaluating the original function $f(x)$. In contrast to the forward mode, the reverse mode in general carries a larger, but often avoidable, memory overhead. This definition likewise naturally extends to the case when the function takes multiple matrix-valued inputs and multiple matrix-valued outputs.

Any implementation of AD works by breaking down a calculation into a sequence of atomic operations with known, preferably explicit, chain rules. For example, the forward mode AD rule for a matrix-matrix multiplication $Y = X_1 X_2$ is given by:

$$\hat{Y} = \hat{X}_1 X_2 + X_1 \hat{X}_2$$

and the reverse mode AD rule is given by:

$$\bar{X}_1 = \bar{Y} {X_2}^{\mathrm{T}}; \qquad \bar{X}_2 = {X_1}^{\mathrm{T}} \bar{Y},$$

as shown in e.g. [59].

The forward and reverse modes thus offer two ways to efficiently and exactly calculate directional derivatives. Efficiently calculating the complete Jacobian, which can be large and sparse, is a considerably more difficult problem. For large problems, some heuristic that falls back on the above forward and/or reverse modes, is usually required. Higher order derivatives can be treated as special cases, or, which is the case here, by applying the AD algorithms recursively. The Hessian of a scalar-valued function is then simply calculated as the Jacobian of the gradient, preferably exploiting the symmetry of the Hessian.

CasADi implements AD using a source-code-transformation approach, which means that new symbolic expressions, using the same graph representation, are generated whenever derivatives are requested. Differentiable expressions for directional derivatives as well as large-and-sparse Jacobians and Hessians can be calculated.

The following demonstrates how to generate a new expression for the first column of a Jacobian using a Jacobian-times-vector product as well as an expression for the complete Jacobian:

```
% MATLAB/Octave                  # Python
J1 = jtimes(e,x,[1;0]);          J1 = jtimes(e,x,[1,0])
J = jacobian(e,x);               J = jacobian(e,x)
```

In the remainder of this section, we present the implementation of AD in CasADi, assuming that the reader is already familiar with AD. We refer to [13, Chapter 3] for a simple introduction and Griewank and Walther [64] or Naumann [107] for a more complete introduction to AD.

Directional derivatives

For a function object defined by a symbolic expression, CasADi implements the forward and reverse modes of AD by propagating symbolic seeds forward and backward through the algorithm respectively, resulting in new symbolic expressions that contain references to nodes of the expression graph of the non-differentiated function. Whenever a *function call* node is encountered, cf. Section 2.3, a new function object is generated for calculating directional derivatives. The generated function objects for the derivatives are *cached* in order to limit memory use. How a function class calculates directional derivatives is class-specific; e.g., an integrator node typically generates functions for directional derivatives by augmenting the ODE/DAE integration with its sensitivity equations.

If a symbolic expression consists of a large number of nodes, the evaluation of reverse mode derivatives may be costly in memory, since the intermediate results must be kept in memory and then accessed in reverse order. The CasADi user is responsible for avoiding such a memory blowup by breaking up large expressions into a hierarchy of smaller expressions, each encapsulated in a separate function object. Choosing a suitable hierarchy of function objects is equivalent to a *checkpointing strategy* [64, Chapter 12] in AD terminology, and as such comes at the price of a moderate increase in the number of floating point operations for reverse mode AD.

Calculation of complete Jacobians and Hessians

We use a graph coloring approach [54] to generate expressions for complete large and sparse Jacobians and Hessians. The idea of this approach is to reconstruct the Jacobian with a set of Jacobian-vector-products, i.e. directional derivatives. Using greedy graph coloring algorithms, we seek to find a set of vectors that is smaller than the naive choice of simply using the unit vectors, i.e. v_i being the i-th column of the identity matrix.

CasADi uses a heuristic to construct Jacobian and Hessian expressions. The heuristic uses a symmetry-exploiting *greedy, distance-2, star-coloring algorithm* [54, Algorithm 4.1], whenever it is known a priori that the resulting Jacobian is symmetric, in particular whenever a Hessian is being constructed. For asymmetric Jacobians, a *greedy, distance-2, unidirectional algorithm* [54, Algorithm 4.1] is attempted both column-wise (corresponding to forward mode AD) and row-wise (corresponding to reverse mode AD). Depending on the number of rows and columns of the Jacobian, one or the other is attempted first and the algorithm attempted second is interrupted prematurely if determined to need more colors, i.e. more directional derivatives. A factor α, by default 2, is introduced to take into account that a reverse mode AD sweep is usually slower than a forward mode AD sweep. We summarize the algorithm in Table 1.

The coloring algorithms in CasADi are used together with a *largest-first* preordering step [54, 136].

Algorithm 1 Heuristic to calculate complete Jacobians in CasADi

Calculate the Jacobian sparsity pattern
if J is symmetric (typically a Hessian) **then**
  Run a star-coloring algorithm.
else
  if The number of columns of J is fewer than α times the number of rows **then**
    Run a column-wise unidirectional algorithm.
    Run a row-wise unidirectional algorithm, interrupting prematurely if the number of colors surpasses the column-wise algorithm with more than a factor $1/\alpha$.
  else
    Run a row-wise unidirectional algorithm.
    Run a column-wise unidirectional algorithm, interrupting prematurely if the number of colors surpasses the row-wise algorithm with more than a factor α.
  end if
end if

Jacobian sparsity pattern calculation

A priori knowledge of the sparsity pattern of the Jacobian is a precondition to be able to implement the above graph coloring approach. Obtaining this pattern for a generic expression is a nontrivial task and often turns out to be the most expensive step in the Jacobian construction.

CasADi uses the *bitvector approach* [58], which is essentially an implementation of forward or reverse mode AD using a single bit as a datatype, indicating whether a component of a Jacobian-vector-product is structurally zero. The bitvector approach can be implemented efficiently using bitwise operations on an unsigned integer datatype. CasADi uses the 64-bit `unsigned long long` datatype for this, meaning that up to 64 rows or columns can be calculated in a single sparsity propagation sweep.

For the large and sparse Jacobians and Hessians encountered in CasADi, where the number of rows and columns can be in the millions, performing tens of thousands of sparsity propagation sweeps to determine the Jacobian sparsity pattern can be prohibitively expensive. If nothing is known about the location of the nonzero elements, probabilistic methods as shown by Griewank and Mitev [63] have been proposed. For the type of Jacobians typically encountered in CasADi, both in simulation and optimization, the nonzeros are unlikely to be encountered in random locations. A more typical sparsity pattern is one which has large regions that do not have a single nonzero entry. An example of such a structured sparsity pattern, corresponding to the Hessian of the Lagrangian of an NLP arising from direct collocation, can be seen in Figure 1.

To exploit this *block sparse* structure, CasADi implements a *hierarchical sparsity pattern calculation algorithm* based on graph coloring. The algorithm alternates between calculating successively less crude sparsity patterns and the same graph coloring algorithms used in the previous section.

In a first step, the rows and columns of the Jacobian are divided into 64 groups of similar size. The sparsity pattern propagation algorithm, either forward or reverse, is then executed yielding a coarse sparsity pattern. For the pattern in Figure 1, this will result in either a block diagonal or block tridiagonal sparsity pattern, depending

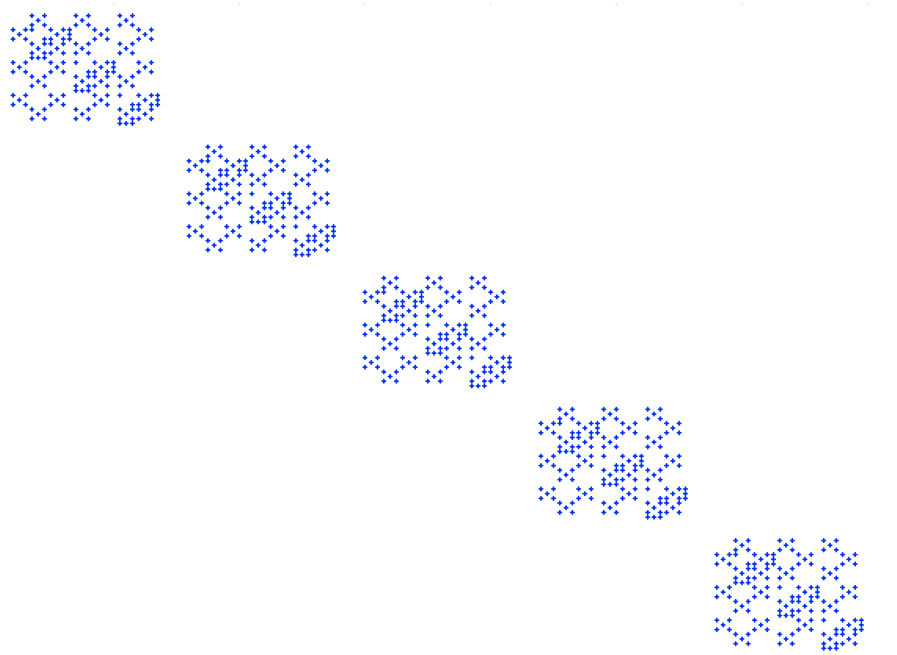

Fig. 1 The sparsity pattern of the Hessian of the Lagrangian of an NLP arising from direct collocation

on how the blocks are chosen. Either case is amenable for graph coloring. After either symmetry exploiting or non-symmetry exploiting graph coloring, the process is repeated for a finer sparsity pattern, using one sparsity propagation sweep for each distinct color found in the coarse pattern.

This process is then repeated for successively finer sparsity patterns, until the the actual sparsity pattern is obtained.

Example 1 *Assume that the Jacobian has dimension 100,000-by-100,000 and has a (a priori unknown) nonsymmetric tridiagonal sparsity pattern.*

The proposed propagation algorithm first performs one sweep which results in a block tridiagonal pattern, with block sizes no larger than ceil$(100,000/64) = 1,563$*. Tridiagonal patterns can trivially be colored with 3 colors, regardless of dimension, meaning that the columns can be divided into 3 groups. For each color, we make one sweep (3 in total) to find a finer sparsity pattern, which will also be tridiagonal. At this point, the blocks are no larger than ceil*$(1,563/64) = 25$*. The graph coloring algorithm is again executed and again resulting in 3 colors. Finally, with one sweep for each color, we obtain the true sparsity pattern.*

For this example, the sparsity pattern is thus recovered in 7 sweeps, which can be compared with ceil$(100,000/64) = 1,563$ *sweeps needed for the naive algorithm, where 64 rows or columns are calculated in each sweep.*

This proposed hierarchical sparsity pattern calculation algorithm is not efficient if the nonzero entries are spread out randomly and CasADi assumes that the user takes this into account when e.g. formulating a large NLP.

2.7 Control flow handling

A common question posed by CasADi users is how to handle expressions that involve flow control such as if-statements, for-loops and while-loops. Expressions containing flow control appear naturally in a range of applications, e.g. for physical models governed by different equations for different value ranges. Being able to calculate derivatives for such models, that are at least accurate in the *almost everywhere* sense, is essential for practical numerical optimization.

The approach chosen in CasADi is to support flow control by implementing concepts from *functional programming languages* such as Haskell.

Conditionals

Conditional expressions, which include switches and if-statements, can be expressed using a dedicated `Switch` function object class in CasADi. This construction, which is defined by a set of other function objects in CasADi, is defined by a vector of function objects corresponding to the different cases as well as the default case, all with the same input-output signature:

Algorithm 2 A `Switch` function object in CasADi defined by $(f_0,\ldots,f_N)$

input $(c,x_0,\ldots,x_{n-1})$
if floor$(c)=0$ **then**
  $(y_0,\ldots,y_{m-1})=f_0(x_0,\ldots,x_{n-1})$
else if ... **then**
  ...
else if floor$(c)=N-1$ **then**
  $(y_0,\ldots,y_{m-1})=f_{N-1}(x_0,\ldots,x_{n-1})$
else
  $(y_0,\ldots,y_{m-1})=f_N(x_0,\ldots,x_{n-1})$
end if
return $(y_0,\ldots,y_{m-1})$

Since CasADi calculates derivatives of function objects by generating expressions for its directional derivatives, the derivative rules for the above class – for forward mode and reverse mode – can be defined as new conditional function objects defined by the corresponding derivative functions. Note that derivatives with respect to the first argument (c above) are zero almost everywhere.

An important special case of the `Switch` construct is if-then-else operations, for which $N=1$.

Maps

Another readily differentiable concept from functional programming is a *map*. A map in this context is defined as a function being evaluated multiple times with different arguments. This evaluation can be performed serially or in parallel.

The `Map` function object class in CasADi allows users to formulate maps. This function object takes a horizontal concatenation of all inputs and returns a horizontal concatenation of all outputs:

Algorithm 3 A `Map` function object in CasADi defined by f and N

input $(X_0,\ldots,X_{n-1})$
for $i=0,\ldots,N-1$ **do**
  $(y_{i,0},\ldots,y_{i,m-1}) = f(x_{i,0},\ldots,x_{i,n-1})$
end for
return $(Y_0,\ldots,Y_{m-1})$

where $X_j := [x_{0,j},\ldots,x_{N-1,j}]$, $j=0,\ldots,m-1$ and $Y_k := [y_{0,k},\ldots,x_{N-1,k}]$, $k=0,\ldots,n-1$.

3 Implicitly defined differentiable functions

Optimization problems may contain quantities that are defined implicitly. An important example of this, and one of the motivations to write CasADi in the first place, is the direct multiple shooting method, where the NLP contains embedded solvers for initial-value problems in differential equations. We will showcase this method in Section 5.4. Another example is a dynamic system containing algebraic loops, which can be made explicit by embedding a root-finding solver.

In the following, we discuss how certain implicitly defined functions can be embedded into symbolic expressions, but still have their derivative and sparsity information generated automatically and efficiently.

3.1 Linear systems of equations

As shown in [59], the solution to a linear system of equations $y = X_2^{-1} x_1$ has forward and reverse mode AD rules defined by:

$$\hat{y} = X_2^{-1}(\hat{x_1} - \hat{X_2} y); \qquad \bar{x}_1 = X_2^{-\mathrm{T}} \bar{y}; \qquad \bar{X}_2 = -\bar{x}_1 y^{\mathrm{T}}$$

Apart from standard operations such as matrix multiplications, the directional derivatives can thus be expressed using a linear solve with the same linear system as the nondifferentiated expression. In the reverse mode case, a linear solve with the transposed matrix is needed.

CasADi supports linear system of equations of this form through a linear solver abstract base class, which is an oracle class able to factorize and solve dense or sparse

linear systems of equations. The solve step supports optional transposing, as required by the reverse mode of AD. The linear solver class is a plugin class in CasADi, and leaves to the derived class – typically an interface to a third-party linear solver – to actually perform the factorization and solution.

The linear solver instances are embedded into CasADi's `MX` expression graphs using a dedicated *linear solve* node, which is similar to the more generic function call node introduced in Section 2.3. The linear solve node implements the above derivative AD rules, using `MX`'s nodes for horizontal split and concatenation to be able to reuse the same factorization for multiple directional derivatives, i.e. multiple right-hand-sides.

Sparsity pattern propagation for the linear solve node was implemented by first making a block-triangular reordering of the rows and columns, exposing both uni-directional and bidirectional dependencies between the elements. The reordering is calculated only once for each sparsity pattern and then cached.

At the time of this writing, the linear solver plugins in CasADi – which may impose additional restrictions such as symmetry or positive definiteness – included CSparse [32] (sparse LU, sparse QR and sparse Cholesky decompositions), MA27 [6] (frontal method) and LAPACK [12] (dense LU and dense QR factorizations).

3.2 Nonlinear systems of equations

A more general implicitly defined function is the solution of a root-finding problem:

$$g(y,x) = 0 \Leftrightarrow y = f(x). \tag{3}$$

If regularity conditions are satisfied, in particular the existence and invertibility of the Jacobian $\frac{\partial g}{\partial y}$, the problem is well-posed and its Jacobian is given by the *implicit function theorem*:

$$\frac{\partial f}{\partial x} = -\left(\frac{\partial g}{\partial y}\right)^{-1} \frac{\partial g}{\partial x}$$

which readily gives the forward and reverse AD propagation rules of (3)

$$\hat{y} = -\left(\frac{\partial g}{\partial y}\right)^{-1} \frac{\partial g}{\partial x}\hat{x}; \qquad \bar{x} = -\left(\frac{\partial g}{\partial x}\right)^{\mathrm{T}} \left(\frac{\partial g}{\partial y}\right)^{-\mathrm{T}} \bar{y}. \tag{4}$$

For the forward mode, the problem of calculating directional derivatives for the implicitly defined function $f(x)$ is reduced to the problem of calculating forward mode directional derivatives of the residual function $g(y,x)$ followed by a linear solve. For the reverse mode, the calculation involves a linear solve for the transposed system followed by a reverse mode directional derivative for the residual function.

CasADi supports the above via so-called `rootfinder` function objects. These use a generalization of (3) which includes multiple matrix-valued input parameters

and a set of auxiliary outputs:

$$\begin{pmatrix} g_0(y_0, x_1, \ldots, x_{n-1}) \\ g_1(y_0, x_1, \ldots, x_{n-1}) - y_1 \\ \vdots \\ g_{m-1}(y_0, x_1, \ldots, x_{n-1}) - y_{m-1} \end{pmatrix} = 0 \Leftrightarrow (y_0, \ldots, y_{m-1}) = f(x_0, \ldots, x_{n-1}). \quad (5)$$

where x_0 has been introduced as an *initial guess* for y_0. Note that the derivative with respect x_0 is zero almost everywhere.

Like linear solvers, root-finders are implemented using a plugin-design. In the base class, rules for derivative calculation and sparsity pattern propagation are defined whereas solving the actual nonlinear system of equations is delegated to a derived class, typically in the form of an interfaced third-party tool. Root-finding objects in CasADi can be differentiated an arbitrary number of times since both its forward and reverse mode directional derivatives can be expressed by (arbitrarily differentiable) CasADi constructs, namely directional derivatives for the residual function and the linear solver operation treated in Section 3.1. The linear solver is typically the same as the linear solver used in a Newton method for the nonlinear system of equations. By default, the CSparse [32] plugin is used, but this can be changed by passing an option to the `rootfinder` constructor.

3.3 Initial-value problems in ODE and DAE

Certain methods for optimal control, including direct multiple shooting and direct single shooting, will result in optimization problem formulations that require solving initial-value problems (IVP) in ODEs or DAEs. Since the integrator calls appear in the constraint and/or objective functions of an NLP, we need ways to calculate first and preferably second order derivative information.

The CasADi constructs introduced until now can be used to define either explicit or implicit fixed step-step integrator schemes. For example, a Runge-Kutta 4 (RK4) scheme can be implemented with less than 10 lines of code using CasADi's `MX` type, and derivatives can be generated automatically to any order. Similarily, an implicit fixed-step scheme, such as a collocation method, can be implmemented using CasADi's `rootfinder` functionality, described in Section 3.2.

More advanced integrator schemes, such as backwards difference formula (BDF) methods with variable order and/or adaptive step-size, cannot be handled with this approach. Compared to a fixed-step integrator scheme, an adaptive scheme often results in fewer steps for the same accuracy and the user is relieved from choosing appropriate step-sizes. CasADi's `integrator` functionality enables the user to embed solvers of initial-value problems in ODEs or DAEs and have derivatives to any

order calculated exactly using state-of-the-art codes such as those in the SUNDIALS suite [72]. CasADi's `integrator` objects solve problems of the following form:

$$
\begin{aligned}
&f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_x} \times \mathbb{R}^{n_z} \times \mathbb{R}^{n_q} \times \mathbb{R}^{n_r} \times \mathbb{R}^{n_s} \times \mathbb{R}^{n_p}, \\
&\quad (x_0, z_0, u, r_T, s_T, v) \mapsto (x(T), z(T), q(T), r(0), s(0), p(0)) \\
&\begin{cases}
\dot{x}(t) = \phi(x(t), z(t), u), & x(0) = x_0 \\
0 = \theta(x(t), z(t), u), & z_0 \text{ initial guess for } z(0) \\
\dot{q}(t) = \psi(x(t), z(t), u), & q(0) = 0 \\
-\dot{r}(t) = \phi^*(x(t), z(t), u, r(t), s(t), v), \quad t \in [0,T] & r(T) = r_T \\
0 = \theta^*(x(t), z(t), u, r(t), s(t), v), & s_T \text{ initial guess for } s(T) \\
-\dot{p}(t) = \psi^*(x(t), z(t), u, r(t), s(t), v), & p(T) = 0
\end{cases}
\end{aligned} \tag{6}
$$

The problem consists of two semi-explicit DAEs with initial and terminal constraints, respectively, and both with support for calculation of quadratures. The second DAE is coupled to the solution trajectory of the first DAE. We impose the additional requirement that $\theta^*(x,z,u,r,s,v)$, $\phi^*(x,z,u,r,s,v)$ and $\psi^*(x,z,u,r,s,v)$ are affine in r, s and v. Initial guesses for $z(0)$ and $s(T)$ are included for efficiency, robustness and to ensure solution uniqueness.

This integrator formulation has two major advantages in the context of optimization and sensitivity analysis. Firstly, it is general enough to handle most industry relevant simulation problems and the quadrature functionalty allows integral terms in objective functions to be calculated efficiently using quadrature formulas. Secondly, it can be shown [13] that both forward and reverse mode directional derivatives of this problem can be calculated efficiently by solving a problem that has exactly the same structure.

By performing the differentiation repeatedly, we can calculate derivatives to any order, using an appropriate mix of forward and adjoint sensitivity analysis. In particular, we can perform forward-over-adjoint sensitivity analysis for efficient Hessian calculation. A potential drawback, which is inherent with this so-called *variational approach to sensitivity analysis*, is that when used inside a gradient-based optimization code, the calculated derivatives may not be consistent with the nondifferentiated function evaluation, due to differences in time discretization.

Like linear solvers and root-finding solvers, integrators are implemented using a plugin design in CasADi. The base class implements the rules for differentiation and sparsity pattern propagation and the derived class, which can be an interface to a third-party tool, performs the actual solution. Integrator plugins in CasADi include IDAS and CVODES from the SUNDIALS suite [72], a fixed step-size RK4 code, and an implicit Runge-Kutta code implementing Legendre or Radau collocation.

4 Optimization

Just like conventional algebraic modeling languages, CasADi combines support for modeling with support for mathematical optimization. Two classes of optimization problems are supported; nonlinear programs (NLPs) and conic optimization problems. The latter class includes both linear programs (LPs) and quadratic programs

(QPs). The actual solution typically takes place in a derived class, and may use a tool distributed with CasADi or be an interface to a third-party solver. The role of CasADi is to extract information about structure and generate the required derivative information as well as to provide a common interface for all solvers.

4.1 Nonlinear programming

CasADi uses the following formulation for a nonlinear program (NLP):

$$\begin{aligned} &\underset{x,p}{\text{minimize}} && f(x,p) \\ &\text{subject to} && \underline{x} \le x \le \overline{x}, \quad p = \overline{p}, \quad \underline{g} \le g(x,p) \le \overline{g}. \end{aligned} \tag{7}$$

This is a parametric NLP where the objective function $f(x,p)$ and the constraint function $g(x,p)$ depend on the decision variable x and a known parameter p. For equality constraints, the variable bounds $[\underline{x},\overline{x}]$ or constraint bounds $[\underline{g},\overline{g}]$ are equal.

The solution of (7) yields a primal (x,p) and a dual $(\lambda_x,\lambda_p,\lambda_g)$ solution, where the Lagrange multipliers are chosen to be consistent with the following definition of the Lagrangian function:

$$\mathscr{L}(x,p,\lambda_x,\lambda_p,\lambda_g) := f(x,p) + \lambda_x^{\mathrm{T}} x + \lambda_p^{\mathrm{T}} p + \lambda_g^{\mathrm{T}} g(x,p). \tag{8}$$

This formulation drops all terms that do not depend on x or p and uses the same multipliers (but with different signs) for the inequality constraints according to

$$\lambda_{\overline{x}}(x-\overline{x}) + \lambda_{\underline{x}}(\underline{x}-x) = \underbrace{(\lambda_{\overline{x}} - \lambda_{\underline{x}})}_{:=\lambda_x}{}^{\mathrm{T}} x \underbrace{-\lambda_{\overline{x}}^{\mathrm{T}}\overline{x} + \lambda_{\underline{x}}^{\mathrm{T}}\underline{x}}_{\text{ignored}}$$

and equivalently for $g(x,p)$. Note that $\lambda_{\overline{x}}(i)$ and $\lambda_{\underline{x}}(i)$ cannot both be positive and that the dropped terms do not appear in the KKT conditions of (7).

In this NLP formulation, a strictly positive multiplier signals that the corresponding *upper* bound is active and vice versa. Furthermore, λ_p is the *parametric sensitivity* of the objective with respect to the parameter vector p.

Table 2 lists the available NLP solver plugins at the time of this writing. The list includes both open-source solvers that are typically distributed along with CasADi and commercial solvers that require separate installation. The table attempts to make a rough division of the plugins between solvers suitable for very large but very sparse NLPs, e.g., arising from direct collocation, and large and structured NLPs, e.g. arising from direct multiple shooting, cf. Section 4.3. The plugin `'sqpmethod'` corresponds to a "vanilla" SQP method, mainly intended to serve as a boilerplate code for users who intend to develop their own solver codes. A subset of the solvers supports mixed-integer nonlinear programming (MINLP) formulations, where a subset of the decision variables are restricted to take integer values.

Table 2 NLP solver plugins in CasADi 3.1

Solver	Plugin name	License	Method	Problem
IPOPT [134]	`'ipopt'`	EPL	Nonlinear interior point [b]	NLP
BONMIN [1]	`'bonmin'`	EPL	Nonlinear interior point [b]	MINLP
KNITRO [28]	`'knitro'`	Commercial	Multiple [b, c]	MINLP
WORHP [27]	`'worhp'`	Commercial [a]	SQP [b]	NLP
SNOPT [60]	`'snopt'`	Commercial	SQP [c]	NLP
`casadi::Blocksqp` [d]	`'blocksqp'`	zlib	SQP [c]	NLP
`casadi::Sqpmethod`	`'sqpmethod'`	LGPL	SQP boilerplate code	NLP
`casadi::Scpgen`	`'scpgen'`	LGPL	SQP [c]	NLP

[a] Free for academic use
[b] Suitable for large and sparse NLPs
[c] Suitable for structured NLPs
[d] Fork of blockSQP [77]

4.2 Conic optimization

When the objective function $f(x,p)$ is quadratic in x and the constraint function $g(x,p)$ is linear in x, (7) can be posed as a quadratic program (QP) of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \tfrac{1}{2}x^{\mathrm{T}}Hx + g^{\mathrm{T}}x \\ \text{subject to} \quad & \underline{x} \le x \le \overline{x}, \quad \underline{a} \le Ax \le \overline{a}. \end{aligned} \tag{9}$$

CasADi supports solving QPs, formulated either as (7) or as (9). In the former case, AD is used to reformulate the problem in form (9), automatically identifying the sparse matrices H and A as well as the vectors g, $\underline{a}$ and $\overline{a}$.

As with NLP solvers, the solution of optimization problems takes place in one of CasADi's *conic* solver plugins, listed in Table 3. Some plugins impose additional restrictions on (9); *linear* programming solvers require that the H term is zero and most interfaced QP solvers require H to be positive semi-definite. A subset of the solvers supports mixed-integer formulations. Note that mixed integer quadratic programming (MIQP) is a superset of QP and that QP is a superset of LP. Future versions of CasADi may allow more generic conic constraints such as SOCP and SDP [25].

4.3 Sparsity and structure exploitation

NLPs and QPs arising from transcription of optimal control problems are often either *sparse* or *block-sparse*. A sparse QP in this context is one where matrices – i.e. A and H in (9) – have few enough nonzero entries per row or column to be handled efficiently by general sparse linear algebra routines. For a sparse NLP, the same applies to the matrices that arrise from the linearization of the KKT conditions, i.e. to $\frac{\partial g}{\partial x}$ and $\nabla^2_x \mathscr{L}$ in (7) and (8). A direct collocation type OCP transcription will typically result in sparse NLPs or QPs, provided that the Jacobian of the ODE right-hand-side function (or DAE residual function) is sufficiently sparse. Several tools exist that can

Table 3 Conic solver plugins in CasADi 3.1

Solver	Plugin name	License	Method	Problem
qpOASES [47]	`'qpoases'`	LGPL	Active-set	QP [b]
CPLEX [76]	`'cplex'`	Commercial	Multiple [d]	MIQP
GUROBI [5]	`'gurobi'`	Commercial	Multiple [d]	MIQP
CLP [3]	`'clp'`	CPL	Simplex	LP
OOQP [56]	`'ooqp'`	Open-source	Interior point [d]	QP
HPMPC [53]	`'hpmpc'`	LGPL	Interior point	QP[e]

[a] Free for academic use
[b] Supports non-convex QPs
[c] Supports integer decision variables
[d] For large and sparse problems
[e] For structured problems

handle these problems efficiently, cf. Table 2 and Table 3. These solvers generally rely on sparse direct linear algebra routines, e.g. from the HSL Library [6].

A direct multiple shooting type transcription, on the other hand, will typically result in NLPs or QPs that have dense sub-blocks and hence an overall block-sparse pattern, with too many nonzero entries to be handled efficiently by general sparse linear algebra routines. *Condensing* [84] in combination with a dense solver such as qpOASES [47] is known to work well for certain structured QPs, in particular when the time horizon is short and the control dimension small relative to the state dimension. Other QPs can be solved efficiently with tools such as FORCES [38], qpDUNES [51] and HPMPC [53]. The lifted Newton method [9] is a generalization of the condensing approach to NLPs and nonlinear root-finding problems. We refer to [78] for a comprehensive treatment of structured QPs and NLPs.

At the time of this writing, CasADi supported one structured QP solver, HPMPC [53], and two structured NLP solvers, Scpgen and blockSQP. Scpgen [13] is an implementation of the lifted Newton method using CasADi's AD framework and block-SQP [77], incorporated into CasADi in modified form, can handle NLPs with a block-diagonal Hessian matrices.

5 Tutorial examples

In the following, we showcase the CasADi syntax and usage paradigm through a series of tutorial examples of increasing complexity. The first two examples introduce the optimization modeling approach in CasADi, which differs from that of conventional algebraic modeling languages such as AMPL, GAMS, JuMP or Pyomo. In Section 5.3, we demonstrate the automatic ODE/DAE sensitivity analysis in CasADi and finally, in Section 5.4, we combine the tools introduced in the previous examples in order to implement the *direct multiple shooting* method.

We will use a syntax corresponding to CasADi version 3.1 in both MATLAB/Octave and Python, side-by-side. The presentation attempts to convey a basic understanding of what modeling in CasADi entails. For a more comprehensive and up-to-date introduction to CasADi, needed to understand each line of the example scripts, we refer to the user guide [14].

5.1 An unconstrained optimization problem

Let us start out by finding the minimum of Rosenbrock's banana-valley function:

$$\underset{x,y}{\text{minimize}} \quad x^2 + 100\,(y-(1-x)^2)^2 \tag{10}$$

By inspection, we can see that its unique solution is $(0,1)$. The problem can be formulated and solved with CasADi as follows:

```
% MATLAB/Octave
import casadi.*

% Symbolic representation
x=SX.sym('x');
y=SX.sym('y');
z=y-(1-x)^2;
f=x^2+100*z^2;
P=struct('x',[x;y],'f',f);

% Create solver instance
F=nlpsol('F','ipopt',P);

% Solve the problem
r=F('x0',[2.5 3.0])
disp(r.x)
```

```
# Python
from casadi import *

# Symbolic representation
x=SX.sym('x')
y=SX.sym('y')
z=y-(1-x)**2
f=x**2+100*z**2
P=dict(x=vertcat(x,y),f=f)

# Create solver instance
F=nlpsol('F','ipopt',P)

# Solve the problem
r=F(x0=[2.5,3.0])
print(r['x'])
```

The solution consists of three parts. Firstly, constructing a symbolic representation of the problem in the form of a MATLAB/Octave `struct` or a Python `dict`. A naming scheme, consistent with (7), is used for the different fields. The variable `z` is an intermediate expression; more complex models will contains a large number of such expressions. Secondly, a solver instance is created, here using IPOPT. During this step, the AD framework is invoked to generate a set of solver-specific functions for numerical evaluation, here corresponding to the cost function, its gradient, and its Hessian. Finally, the solver instance is evaluated numerically in order to obtain the optimal solution. We pass the initial guess $(2.5, 3.0)$ as an input argument. Other inputs are left at their default values, e.g. the bounds on x are $\underline{x} = -\infty$ and $\overline{x} = \infty$.

The above script converges to the optimal solution in 26 iterations. The total solution time on a MacBook Pro is in the order of 0.02 s.

5.2 Nonlinear programming example

Let us reformulate (10) as a constrained optimization problem, introducing a decision variable corresponding to `z` above:

$$\begin{aligned} &\underset{x,y,z}{\text{minimize}} && x^2+100z^2 \\ &\text{subject to} && z+(1-x)^2-y=0. \end{aligned} \tag{11}$$

The problem can be formulated and solved with CasADi as follows:

```
% MATLAB/Octave
import casadi.*

% Formulate the NLP
x=SX.sym('x');
y=SX.sym('y');
z=SX.sym('z');
f=x^2+100*z^2;
g=z+(1-x)^2-y;
P=struct('f',f,'g',g,...
         'x',[x;y;z]);

% Create solver instance
F=nlpsol('F','ipopt',P);

% Solve the problem
r=F('x0',[2.5 3.0 0.75],...
    'ubg',0,'lbg',0);
disp(r.x)
```

```
# Python
from casadi import *

# Formulate the NLP
x=SX.sym('x')
y=SX.sym('y')
z=SX.sym('z')
f=x**2+100*z**2
g=z+(1-x)**2-y
P=dict(f=f,g=g,\
       x=vertcat(x,y,z))

# Create solver instance
F=nlpsol('F','ipopt',P)

# Solve the problem
r=F(x0=[2.5,3.0,0.75],\
    ubg=0, lbg=0)
print(r['x'])
```

We impose the equality constraint by setting the upper and lower bound of g to 0 and use the $(2.5, 3.0, 0.75)$ as an initial value, consistent with the initial guess for the unconstrained formulation. The above script converges to the optimal solution in 10 iterations taking around 0.01 s.

Notice how *lifting* the optimization problem to a higher dimension like this resulted in faster local convergence of IPOPT's Newton-type method. This behavior can often be observed for structurally complex nonlinear problems as discussed in e.g. [9]. This faster local convergence is one of the advantages of the direct multiple shooting method, which we will return to in Section 5.4.

5.3 Automatic sensitivity analysis example

We now shift our attention to simulation and sensitivity analysis using the CasADi's `integrator` objects introduced in Section 3.3. Consider the following initial-value problem in ODE corresponding to a Van der Pol oscillator:

$$\begin{cases} \dot{x}_1 = (1-x_2^2)x_1 - x_2 + p, & x_1(0) = 0 \\ \dot{x}_2 = x_1, & x_2(0) = 1 \end{cases} \tag{12}$$

With p fixed to 0.1, we wish to solve for $x_f := x(1)$. This can be solved as follows:

```
% MATLAB/Octave
import casadi.*

% Formulate the ODE
x=SX.sym('x',2);
p=SX.sym('p');
z=1-x(2)^2;
f=[z*x(1)-x(2)+p;x(1)];
dae=struct('x',x,'p',p,...
           'ode',f);

% Create integrator
op=struct('t0',0,'tf',1);
F=integrator('F',...
      'cvodes',dae,op);

% Integrate
r=F('x0',[0,1],'p',0.1);
disp(r.xf)
```

```
# Python
from casadi import *

# Formulate the ODE
x=SX.sym('x',2)
p=SX.sym('p')
z=1-x[1]**2
f=vertcat(z*x[0]-x[1]+p,\
          x[0])
dae=dict(x=x,p=p,ode=f)

# Create integrator
op=dict(t0=0,tf=1)
F=integrator('F',\
      'cvodes',dae,op)

# Integrate
r=F(x0=[0,1],p=0.1)
print(r['xf'])
```

As for the optimization examples above, the solution consists of three parts; construction of a symbolic representation of the problem, creating a solver instance, and evaluating this solver instance in order to obtain the solution. In the scripts above, we used CVODES from the SUNDIALS suite [72] to solve the initial value problem, which implements a variable step-size, variable-order backward differentiation formula (BDF) method.

Since `F` in the above scripts is a differentiable CasADi function, as described in Section 3.3, we can automatically generate derivative information to any order. For example, the Jacobian of $x(1)$ with respect to $x(0)$ can be calculated as follows:

```
% Create Jacobian function
D=F.factory('D',...
  {'x0','p'},{'jac:xf:x0'});

% Sensitivity analysis
r=D('x0',[0,1],'p',0.1);
disp(r.jac_xf_x0)
```

```
# Create Jacobian function
D=F.factory('D',\
  ['x0','p'],['jac:xf:x0'])

# Sensitivity analysis
r=D(x0=[0,1],p=0.1)
print(r['jac_xf_x0'])
```

The automatic sensitivity analysis is often invoked indirectly, when using a gradient-based optimization solver, as the next example shows.

5.4 The direct multiple shooting method

By combining the nonlinear programing example in Section 5.2 with the embeddable integrator in Section 5.3, we can implement the direct multiple shooting method by Bock and Plitt [23,22]. We will consider a simple OCP with the same IVP as in (12), but reformulated as a DAE and with p replaced by a time-varying control u:

$$\underset{x(\cdot),z(\cdot),u(\cdot)}{\text{minimize}} \quad \int_0^T \left(x_1(t)^2 + x_2(t)^2 + u(t)^2\right) dt \tag{13}$$

$$\text{subject to} \quad \begin{cases} \dot{x}_1(t) = z(t)\,x_1(t) - x_2(t) + u(t) \\ \dot{x}_2(t) = x_1(t) \\ 0 = x_2(t)^2 + z(t) - 1 \\ -1.0 \le u(t) \le 1.0, \quad x_1(t) \ge -0.25 \end{cases} \qquad t \in [0,T] \tag{14}$$

$$x_1(0) = 0, \quad x_2(0) = 1 \tag{15}$$

where $x(\cdot) \in \mathbb{R}^2$ is the (differential) state, $z(\cdot) \in \mathbb{R}$ is the algebraic variable and $u(\cdot) \in \mathbb{R}$ is the control. We let $T = 10$.

Our goal is to transcribe the OCP (15) to a problem of form (7). In the direct approach, the first step in this process is a parameterization of the control trajectory. For simplicity, we assume a uniformly spaced, piecewise constant control trajectory:

$$u(t) := u_k \quad \text{for } t \in [t_k, t_{k+1}), \quad k = 0,\ldots,N-1 \quad \text{with } t_k := kT/N.$$

With the control fixed over one interval, we can use an integrator to reformulate the problem from continuous time to discrete time. This can be done as in Section 5.3. Since we now have a DAE, we introduce an algebraic variable `z` and the corresponding algebraic equation `g` and use IDAS instead of CVODES. We also introduce a quadrature for calculating the contributions to the cost function.

```
% MATLAB/Octave
import casadi.*

% Formulate the DAE
x=SX.sym('x',2);
z=SX.sym('z');
u=SX.sym('u');
f=[z*x(1)-x(2)+u;x(1)];
g=x(2)^2+z-1;
h=x(1)^2+x(2)^2+u^2;
dae=struct('x',x,'p',u,...
   'ode',f,...
   'z',z,'alg',g,'quad',h);

% Create solver instance
T = 10; % end time
N = 20; % discretization
op=struct('t0',0,'tf',T/N);
F=integrator('F',...
       'idas',dae,op);
```

```
# Python
from casadi import *

# Formulate the DAE
x=SX.sym('x',2)
z=SX.sym('z')
u=SX.sym('u')
f=vertcat(z*x[0]-x[1]+u,\
          x[0])
g=x[1]**2+z-1
h=x[0]**2+x[1]**2+u**2
dae=dict(x=x,p=u,ode=f,\
    z=z,alg=g,quad=h)

# Create solver instance
T = 10. # end time
N = 20 # discretization
op=dict(t0=0,tf=T/N)
F=integrator('F',\
       'idas',dae,op)
```

Our next step is to construct a symbolic representation of the NLP. We will use the following formulation:

$$\begin{aligned} &\text{minimize } J(w) \\ &\text{subject to } G(w) = 0, \quad \underline{w} \leq w \leq \overline{w} \end{aligned} \tag{16}$$

For this we start with an empty NLP and add a decision variable corresponding to the initial conditions:

```
% Empty NLP
w={}; lbw=[]; ubw=[];
G={}; J=0;

% Initial conditions
Xk=MX.sym('X0',2);
w{end+1}=Xk;
lbw=[lbw;0;1];
ubw=[ubw;0;1];
```

```
# Empty NLP
w=[]; lbw=[]; ubw=[]
G=[]; J=0

# Initial conditions
Xk=MX.sym('X0',2)
w+=[Xk]
lbw+=[0,1]
ubw+=[0,1]
```

Inside a for loop, we introduce decision variables corresponding to each control interval and the state at the end of each interval:

```
for k=1:N
  % Local control
  Uname=['U' num2str(k-1)];
  Uk=MX.sym(Uname);
  w{end+1}=Uk;
  lbw=[lbw;-1];
  ubw=[ubw; 1];

  % Call integrator
  Fk=F('x0',Xk,'p',Uk);
  J=J+Fk.qf;

  % New local state
  Xname=['X' num2str(k)];
  Xk=MX.sym(Xname,2);
  w{end+1}=Xk;
  lbw=[lbw;-.25;-inf];
  ubw=[ubw; inf; inf];

  % Continuity constraint
  G{end+1}=Fk.xf-Xk;
end
```

```
for k in range(1,N+1):
  # Local control
  Uname='U'+str(k-1)
  Uk=MX.sym(Uname)
  w+=[Uk]
  lbw+=[-1]
  ubw+=[ 1]

  # Call integrator
  Fk=F(x0=Xk,p=Uk)
  J+=Fk['qf']

  # New local state
  Xname='X'+str(k)
  Xk=MX.sym(Xname,2)
  w+=[Xk]
  lbw+=[-.25,-inf]
  ubw+=[ inf, inf]

  # Continuity constraint
  G+=[Fk['xf']-Xk]
```

With symbolic expressions for (16), we can use CasADi's fork of blockSQP [77] to solve this block structured NLP, as in Section 5.2:

```
% Create NLP solver
nlp=struct('f',J,...
      'g',vertcat(G{:}),...
      'x',vertcat(w{:}));
S=nlpsol('S',...
      'blocksqp',nlp);

% Solve NLP
r=S('lbx',lbw,'ubx',ubw,...
    'x0',0,'lbg',0,'ubg',0);
disp(r.x);
```

```
# Create NLP solver
nlp=dict(f=J,\
      g=vertcat(*G),\
      x=vertcat(*w))
S=nlpsol('S',\
      'blocksqp',nlp)

# Solve NLP
r=S(lbx=lbw,ubx=ubw,\
     x0=0,lbg=0,ubg=0)
print(r['x'])
```

5.5 Further examples

More examples on CasADi usage can be found in CasADi's example collection. These examples include other OCP methods such as direct single shooting, direct collocation as well as two indirect methods and a dynamic programming method for comparison.

6 Applications of CasADi

Since the first release of CasADi in 2011, the tool has been used to teach optimal control in graduate level courses, to solve optimization problems in science and engineering as well as to implement new algorithms and software. An overview of applied research using CasADi, as of early 2017, is presented in the following.

6.1 Optimization and simulation in science and engineering

In energy research, applications include the exploitation [79,30,102] and transport [130] of fossil fuels, power-to-gas systems [26], control of combined-cycle [80] and steam [17] power plants, solar thermal power plants [114], production models [87] and control [65] of classical wind-turbines, design and control of airborne wind energy systems [75,66,85,43,89,104], MEMS energy harvesters [82], design of geothermal heat pumps [135], thermal control of buildings [105,33], electrical grids [45, 118], electrical grid balancing [44], and price arbitrage on the energy market [34].

In the automotive industry, applications include design and operation of drivetrains [20,108,109], electrical power systems [121], control of combustion engines [120], research towards self-driving cars [95,39,71,19], traffic control [119], and operation of a driving simulator [129].

In the process industries, applications include control [88,73,92], optimal experimental design [90,106], and parameter estimation [74] of (bio-)chemical reactors.

In robotics research, applications include control of agricultural robots [125], remote sensing of icebergs with UAVs [15,70], time-optimal control of robots [132], motion templates for robot-human interaction [133], motion planning of robotic systems with contacts [8,99], and multi-objective control of complex robots [86].

Further assorted applications include estimation in systems biology [124,122], biomechanics [18], optimal control of bodily processes [24,57], signal processing [117], and machine learning [113].

6.2 Nonstandard optimization problem formulations

Most applications above deal with either system design, parameter estimation, model predictive control (MPC) or moving horizon estimation [55] (MHE) formulations. For some of these problems, it is the nonstandard formulation of the optimization problem that poses the main challenge.

Some applications transcend the classical subdivisions; dual control combines control and learning [123,68,46], codesign of optimal trajectory and reference follower [61], multi-objective design [100,126], the use of different transcription methods on different parts of the system statespace [10].

Concerning robustness, formulations include the use of scenario trees [88], ellipsoidal calculus [91], spline-relaxations [127], stochastic control using linearizations [116,61], using sigma-points [111], and using polynomial chaos expansion [111, 110]. In [61], stochastic optimal control was efficiently implemented by embedding a discrete periodic Lyapunov solver in the CasADi expression graphs.

In MPC research, formulations include multi-level iterations [52], offset-free design [112], Lyapunov-based MPC [42,41], multi-objective MPC [101], distributed MPC [128,103], time-optimal path following [36] and tube following [37].

Further formulations include hybrid OCP [49], treatment of systems with invariants [115,67], an improved Gauss-Newton method for OCP [131].

6.3 Software packages using CasADi

Software packages that rely on CasADi for algorithmic differentiation and optimization include the `JModelica.org` package for simulation and optimization [16,98], the `Greybox` tool for constructing thermal building models [35], the `do-mpc` environment for efficient testing and implementation of robust nonlinear MPC [93,94], `mpc-tools-casadi` for nonlinear MPC [7], the `casiopeia` toolbox for parameter estimation and optimum experimental design [2], the `RTC-Tools 2` package for control of hydraulic networks, the `omgtools` package for real-time motion planning in the presence of moving obstacles, the `Pomodoro` toolbox for multi-objective optimal control [29], the spline toolbox for robust optimal control [127], and a MATLAB optimal control toolbox [83].

7 Discussion and outlook

Since the release of CasADi 3.0 in early 2016, the scope and syntax can be considered mature and no more major non-backwards compatible changes are foreseen. Current development focusses on making existing features more efficient, by addressing speed and memory bottlenecks, and adding new functionality.

An area of special interest are mixed integer optimal control problems (MIOCPs). Problems of this class appear naturally across engineering fields, e.g. in the form of discrete actuators in model predictive control (MPC) formulations. At the time of this writing, CasADi included support for mixed-integer QP and NLP problems, as explained in Section 4, but MIOCPs are largely unexplored.

Another ongoing development is to enable automatic sensitivity analysis for NLPs. Assuming the optimal NLP solution meets regularity criteria as described in e.g. [48], directional derivatives of an NLP solver object are guarenteed to exist and can be calculated using information that can be extracted from the expression graphs. Parametric sensitivity information is useful in a range of applications, including the estimation of covariances in parameter estimation.

Acknowledgements The authors would like to thank for the generous support over the years that has made this work possible. In particular; the K.U. Leuven Research Council via CoE EF/05/006 Optimization in Engineering (OPTEC); the Flemish Government via FWO; the Belgian State via Science Policy programming (IAP VII, DYSCO); the European Union via HDMPC (223854), EMBOCON (248940), HIGHWIND (259166), TEMPO (607957), AWESCO (642682); the Helmholtz Association of German Research Centres via vICERP; the German Federal Ministry for Economic Affairs and Energy (BMWi) via projects eco4wind and DyConPV, the German Research Foundation (DFG) via Research Unit FOR 2401; Flanders Make via MBSE4M, Drivetrain Co-design, Conceptdesign. We also thank our industrial partners, including General Electric Global Research and Johnson Controls International Inc.