Modelica interoperability with CasADi

Thermal Systems Workshop, Freiburg 2015

Joel Andersson
`joel@casadi.org`

23 March 2015

Outline

Motivation – Large-scale optimal control problems (OCP)

$$\begin{array}{ll}
\underset{x,z,y,u,p}{\text{minimize}} & \displaystyle\int_0^T l(t,x,z,u,p)\,dt + E(x(T),z(T),p) \\
\text{subject to} & \left\{\begin{array}{lll} \dot{x} & = & f(t,x,z,u,p) \\ 0 & = & g(t,x,z,u,p) \\ y & = & h(t,x,z,u,p) \\ x & \in & [x_{\min}, x_{\max}] \\ y & \in & [y_{\min}, y_{\max}] \\ u & \in & [u_{\min}, u_{\max}] \end{array}\right. \qquad t \in [0, T] \\
& r(x(0), x(T), p) = 0 \\
& p \in [p_{\min}, p_{\max}]
\end{array}$$

$x(\cdot) \in \mathbf{R}^{N_x}$ States

$z(\cdot) \in \mathbf{R}^{N_z}$ Algebraic variables

$y(\cdot) \in \mathbf{R}^{N_y}$ Outputs

$u(\cdot) \in \mathbf{R}^{N_u}$ Free control signals

$p \in \mathbf{R}^{N_p}$ Free parameters

- Optimization problem constrained by an initial-value problem in ordinary or differential-algebraic equations (ODE/DAE)
- Dynamic system can be given as Modelica code
  - Original motivation for writing CasADi

Practical methods for solving optimal control problems

- Characterization of the (global) solution of OCP
  - Hamilton-Jacobi-Bellman equations [Bellman, 1957]
  - "Curse of dimensionality" – only works for small problems
- **Indirect methods:** Characterization of local solutions of OCP
  - Pontryagin's maximum principle [Pontryagin, 1962]
  - Efficient for many problems, but inequalities difficult
- **Direct methods:** Approximate OCP with a nonlinear program:

$$\boxed{\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g(x) = 0, \quad h(x) \leq 0 \end{array} \quad \text{(NLP)}}$$

  - Popular approach since the 1980ies thanks to advancement in NLP methods and software

Software for dynamic optimization

Most real-world problems require **numerical solution** ...

- Problem-specific (many approaches possible)
- General-purpose (direct methods)

... which is typically **difficult to implement efficiently** ...

- Thousands of lines of code
- "Researcher-in-the-loop"

... because of

- **No simple standard form OCP**
- Many **different solution strategies**
- Indirect methods require formulation of **optimality conditions**
- Direct methods result in nonlinear programs (NLPs) that
  - are typically very **large** and either **sparse** or **structured**
  - require 1st and (preferably) 2nd order **derivative information**
  - may contain **calls to integrators** of differential equations

CasADi

A general-purpose software framework for quick, yet efficient, implementation of algorithms for numeric optimization

- Outcome of the PhD work of myself and Joris Gillis at KU Leuven, Belgium
- Facilitates the solution of optimal control problems (OCPs)
  - *Facilitates*, not actually *solve* the OCPs
  - Efficient *direct multiple shooting* or *direct collocation* with order of magnitude fewer lines of code compared to pure C/C++
- Use from C++, Python and (in development) MATLAB
- Free & open-source (LGPL), also for commercial use

casadi.org → github.com

Search or type a command | Explore | Gist | Blog | Help | jgillis

PUBLIC casadi / casadi

Pull Request | Unwatch | Unstar 4 | Fork 4

Code | Network | Pull Requests 0 | Issues 120 | Wiki | Graphs | Settings

Home | Pages | Wiki History | Git Access

Home

New Page | Edit Page | Page History

Welcome to the CasADi wiki!

CasADi is a symbolic framework for automatic differentiation and numeric optimization. Using the syntax of computer algebra systems, it implements automatic differentiation in forward and adjoint modes by means of a hybrid symbolic/numeric approach. The main purpose of the tool is to be a low-level tool for quick, yet highly efficient implementation of algorithms for numerical optimization. Of particular interest is dynamic optimization, using either a collocation approach, or a shooting-based approach using embedded ODE/DAE-integrators. In either case, CasADi relieves the user from the work of efficiently calculating the relevant derivative or ODE/DAE sensitivity information to an arbitrary degree, as needed by the NLP solver. This together with full-featured Python and Octave front ends, as well as back ends to state-of-the-art codes such as Sundials (CVODES, IDAS and KINSOL), IPOPT and KNITRO, drastically reduces the effort of implementing the methods compared to a pure C/C++/Fortran approach.

Every feature of CasADi (with very few exceptions) is available in C++, Python and Octave, with little to no difference in performance, so the user has the possibility of working completely in C++, Python or Octave or mixing the languages. We recommend new users to try out the Python version first, since it allows interactivity and is more stable and better documented than the Octave front-end.

CasADi is an open-source tool, written in self-contained C++ code, depending only on the Standard Template Library. It is developed by Joel Andersson and Joris Gillis at the Optimization in Engineering Center, OPTEC of the K.U. Leuven under supervision of Moritz Diehl. CasADi is distributed under the LGPL license, meaning the code can be used royalty-free even in commercial applications.

Outline

CasADi provides the **building blocks** for optimal control

- Facilitate the OCP-to-NLP reformulation in direct methods
- Efficiently solve sparse or structured NLPs
- Simplify ODE/DAE integration and sensitivity analysis (shooting methods)
- (Model and automatically reformulate OCPs)
- Key components
  - **Symbolic core** written in self-contained C++
  - **In-house solvers** and **interfaces** to third party-solvers for (non)linear eqs, NLP, QP, ODE/DAE integration, ...
  - **Front-ends** to C++ (native), Python and (in development) MATLAB
  - (**SymbolicOCP** OCP modeling framework)

Symbolic core

- CasADi allows you to symbolic expressions using syntax similar to e.g. *Symbolic Math Toolbox* for MATLAB: "**everything-is-a-sparse-matrix**"

```
from casadi import *
x = SX.sym("x")          Variable x with display name "x"
f = sqrt(x**2 + 10)
g = sin(x)
```

$f = \sqrt{x^2 + 10}$

$g = \sin(x)$

- These functions are then used to define *functions* ...

```
F = SXFunction([x],[f,g])
F.init()
```

Defines $F : \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \times \mathbb{R} \\ (x) & \mapsto & (f, g) \end{array}$

- ...that can e.g. be automatically differentiated using *algorithmic differentiation* (AD)

```
J = F.jacobian()
```

Defines $J : \begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \times \mathbb{R} \times \mathbb{R} \\ (x) & \mapsto & \left(\frac{\partial f}{\partial x}, f, g\right) \end{array}$

Two symbolic types with (almost) the same syntax

- SX: *Expression graph* with scalar-valued operations
- Low overhead, for simple functions

```
x = SX.sym("x",2,2)
f = sin(x**2 + 10)
print f.shape
print x, f
```

$(2, 2)$

$$\begin{bmatrix} x_0 & x_2 \\ x_1 & x_3 \end{bmatrix}, \begin{bmatrix} \sin x_0^2 + 10 & \sin x_2^2 + 10 \\ \sin x_1^2 + 10 & \sin x_3^2 + 10 \end{bmatrix}$$

- MX: *Expression graph* with matrix-valued operations
- Larger overhead, but more generic

```
x = MX.sym("x",2,3)
f = sin(x**2 + 10)
print f.shape
print f
```

$(2, 2)$

$x, \quad \sin x^2 + 10$

(NB: sin *and power* *elementwise)*

Why?

By mixing, construct expressions (functions) that are both fast and generic

Algorithmic differentiation (AD) [Andersson, 2013]

- CasADi automatically generates derivative information via state-of-the-art algorithmic differentiation (AD):
  - Jacobian-times-vector products ("forward mode")
  - vector-times-Jacobian products ("reverse mode")
  - Complete Jacobians and Hessians (using graph coloring algorithms)
- "Source-to-source": Derivatives to arbitrary order
- AD implementation uses chain rule for high-level operations (matrix operations, ODE integrators, implicitly defined functions).

Illustration: Gradient of the determinant, $\nabla_x \det(x), x \in \mathbb{R}^{3\times 3}$

- Scalar operations (via minor expansion):

$$\begin{bmatrix} x_{2,2}\,x_{3,3} - x_{3,2}\,x_{2,3} & x_{2,3}\,x_{3,1} - x_{3,3}\,x_{2,1} & x_{3,2}\,x_{2,1} - x_{2,2}\,x_{3,1} \\ x_{3,2}\,x_{1,3} - x_{1,2}\,x_{3,3} & x_{3,3}\,x_{1,1} - x_{1,3}\,x_{3,1} & x_{1,2}\,x_{3,1} - x_{3,2}\,x_{1,1} \\ x_{1,2}\,x_{2,3} - x_{2,2}\,x_{1,3} & x_{1,3}\,x_{2,1} - x_{2,3}\,x_{1,1} & x_{2,2}\,x_{1,1} - x_{1,2}\,x_{2,1} \end{bmatrix}$$

- Matrix operations (via chain rule for determinant): $\det(x)\, x^{-\intercal}$

“Smart interfaces“ to numerical codes

- NLP solvers: **IPOPT**, **KNITRO**, **SNOPT**, **WORHP**, *in-house solvers*
  - Automatic generation of derivative information
- ODE/DAE integrators: **CVODES**, **IDAS**, *in-house solvers*
  - Automatic formulation of forward and adjoint sensitivitity equations
  - Generation of Jacobian information for implicit schemes
- QP Solvers: **qpOASES**, **CPLEX**, **OOQP**
- Other tools: Linear solvers, SDP solvers, ...

C-codegen

- Generate C code from CasADi expressions
  - Supported for large subset of CasADi
  - Efficient, self-contained, no static memory
  - For embedded system or for speed-up calculations

C-codegen: Example

```python
from casadi import *
x = MX.sym("x",10,10)
F = MXFunction([x],[2*mul(x,x)-x])
F.init()
F.generateCode('F.c')
```

$$\Downarrow$$

```c
int eval(const double* const* arg, double* const* res, int* iw, double* w) {
  int i, j, k, *ii, *jj, *kk;
  double r, s, t, *rr, *ss, *tt;
  const double *cr, *cs, *ct;
  for (i=0, rr=w+10; i<100; ++i) *rr++=0;
  for (i=0, rr=w+110, cs=arg[0]; i<100; ++i) *rr++=*cs++;
  for (i=0, rr=w+10; i<10; ++i) for (j=0; j<10; ++j, ++rr)
    for (k=0, ss=w+110+j, tt=w+110+i*10; k<10; ++k) *rr += ss[k*10]**tt++;
  for (i=0, rr=w+10, cs=w+10; i<100; ++i) *rr++=(2.* *cs++ );
  for (i=0, rr=w+10, cr=w+10, cs=w+110; i<100; ++i) *rr++=( *cr++ - *cs++ );
  if (res[0]!=0) for (i=0, rr=res[0], cs=w+10; i<100; ++i) *rr++=*cs++;
  return 0;
}
```

Outline

Using Modelica models in CasADi

- CasADi can import models written in Modelica
- Two separate toolchains (both require JModelica.org ⇒ *Toivo's talk*)
  - `SymbolicOCP` class in CasADi ⇒ *Rest of the talk*
  - `CasADi Interface` in JModelica.org ⇒ *Toivo's talk*

Usage example: Start-up of combined cycle power plants

Joint work with P-O Larsson, F Casella, F Magnusson and J Åkesson

`SymbolicOCP` in CasADi

- Original motivation: *Import* and *reformulation* of OCP from Modelica/Optimica
- Why reformulation? For shooting methods:
  - Smaller dimension more important than sparsity
  - Integrator schemes easier to handle for semi-explicit systems
  - Scaling more important
- How it works: Tutorial

Outline

Tutorial

Outline

Summary

- **CasADi** is an open-source tool that drastically simplifies writing efficient optimization algorithms: `http://casadi.org`
- In development since 2009, now relatively mature tool
  - $\approx$ 50 active users in industry and academia
- Can be used with models from Modelica $\Rightarrow$ *more in Toivo's talk*

MATLAB front-end [In development]

- Late 2014, started work to add a MATLAB front-end to CasADi
- Aproach: Extend SWIG (`www.swig.org`) to MATLAB
  - Open-source tool for generating interfaces to C++ code
  - Collaborative effort together with SWIG community

Status (March 2015)

Essentially feature-complete, in testing. Stable version later this year.

MATLAB front-end: Syntax resembles Python front-end

```python
# Load CasADi
from casadi import *

# Create NLP
x = SX.sym('x')
y = SX.sym('y')
z = SX.sym('z')
v = vertcat([x,y,z])
f = x**2 + 100*z**2
g = z + (1-x)**2 - y
nlp_in = nlpIn(x=v)
nlp_out = nlpOut(f=f,g=g)
nlp = SXFunction(nlp_in,nlp_out)

# Create IPOPT solver object
solver = NlpSolver('ipopt', nlp)
solver.init()

# Solve the NLP
solver.setInput([2.5, 3.0, 0.75], 'x0')
solver.setInput(0, 'lbg')
solver.setInput(0, 'ubg')
solver.evaluate()

# Get the solution
f_opt = solver.getOutput('f')
x_opt = solver.getOutput('x')
lam_x_opt = solver.getOutput('lam_x')
lam_g_opt = solver.getOutput('lam_g')
```

```matlab
% Load CasADi
import casadi.*

% Create NLP
x = SX.sym('x');
y = SX.sym('y');
z = SX.sym('z');
v = [x;y;z];
f = x^2 + 100*z^2;
g = z + (1-x)^2 - y;
nlp_in = nlpIn('x',v);
nlp_out = nlpOut('f',f,'g',g);
nlp = SXFunction(nlp_in,nlp_out);

% Create IPOPT solver object
solver = NlpSolver('ipopt', nlp);
solver.init();

% Solve the NLP
solver.setInput([2.5 3.0 0.75], 'x0');
solver.setInput(0, 'lbg');
solver.setInput(0, 'ubg');
solver.evaluate()

% Get the solution
f_opt = solver.getOutput('f')
x_opt = solver.getOutput('x')
lam_x_opt = solver.getOutput('lam_x')
lam_g_opt = solver.getOutput('lam_g')
```

Outlook

- Development of CasADi continues
  - MATLAB interface
  - Continued work on integrators, structure-exploiting NLP
  - Even more code-generation, just-in-time compilation (libclang/OpenCL)
    - Goal: Codegen everything (linear solvers, ODE integrators, Newton solvers, NLP solvers)
  - Modelica/FMI interoperability
  - Nonlinear model-predictive control
  - Robust (periodic) optimal control
- Work on applications