An Introduction to Algorithmic Differentiation (AD)

Systems seminar

Joel Andersson

28 October 2016

1. Algorithmic differentiation (AD)
   - Methods for calculating derivatives
   - The forward and reverse modes
   - Checkpointing
   - Jacobians and Hessians
   - AD software
   - Summary

2. CasADi
   - Recall: Optimal control
   - Scope of CasADi
   - Capabilities
   - Summary

Table of Contents

Derivatives play a central role in nonlinear optimization – how compute them?

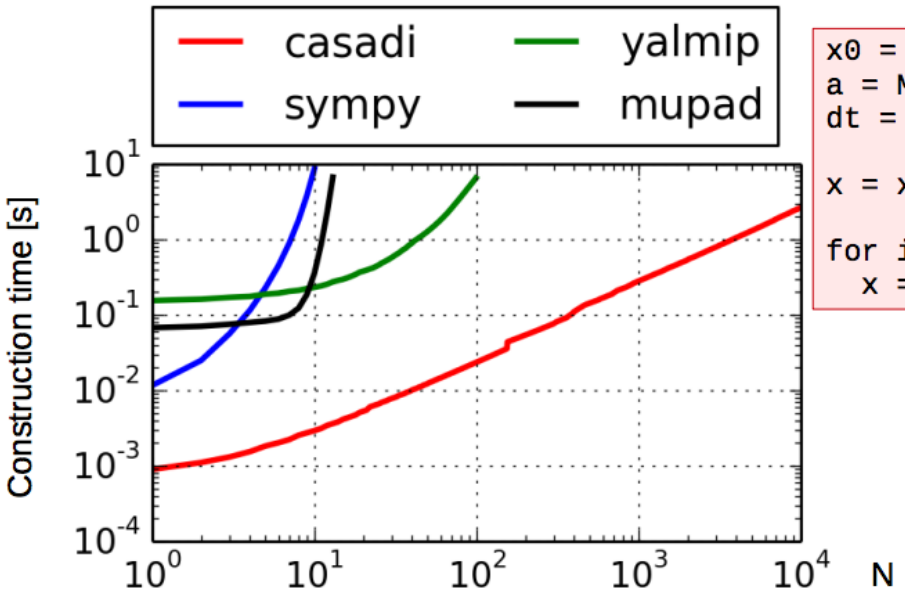
- By hand $\leftarrow$ **Time consuming & error prone!**
- Symbolic differentiation
- Finite difference approximation
- Complex step differentiation ("Imaginary trick")
- Algorithmic differentiation (AD)

Symbolic differentiation

Obtain derivatives with a computer algebra system (CAS):

- Symbolic Toolbox for MATLAB / MuPAD
- SymPy
- ...

Easy to use but often doesn't scale well with number of operations




```
x0 = MX.sym('x0')
a = MX.sym('a')
dt = 1

x = x0

for i in range(N):
  x = x+dt*(a*x+cos(x))
```

Finite differences

Consider a function $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with Jacobian $J(x) = \dfrac{\partial f}{\partial x}$

$$J(x)\,\hat{x} \approx \frac{f(x + t\,\hat{x}) - f(x)}{t}$$

Pros and cons:

- \+ Easy to implement and relatively fast
- – Poor accuracy, need to carefully choose t:
  - Small $t \Rightarrow$ *cancellation errors*
  - Large $t \Rightarrow$ *approximation errors*
- – No efficient way to calculate $\hat{y}^\intercal\, J(x)$

Complex step differentiation ("Imaginary trick")

Finite differences with imaginary perturbation:

$$J(x)\,\hat{x} \approx \Re\left(\frac{f(x + i\,t\,\hat{x}) - f(x)}{i\,t}\right) = -\frac{\Im(f(x + i\,t\,\hat{x}))}{t}$$

Pros and cons:

- \+ Easy to implement in MATLAB/Octave, Python, (Julia?) and relatively fast
- \+ Good accuracy (choose t very small)
- – Error prone, e.g. `x'`
- – Restricted
- – No efficient way to calculate $\hat{y}^\intercal\, J(x)$

Table of Contents

Algorithmic differentiation (AD) (e.g. Griewank & Walther, 2008)

Decomposable function: $y = F(x)$

- $F : \mathbb{R}^{n_0} \to \mathbb{R}^{n_K}$ sufficiently smooth
- Decompose into "atomic operations" with known differentiation rules:

$z_0 \leftarrow x$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
end for
$y \leftarrow z_K$
return y

Such a decomposition is always available if F written as a computer program!

Example

$$y = \sin(\sqrt{x})$$

$z_0 \leftarrow x$
$z_1 = \sqrt{z_0}$
$z_2 = \sin z_1$
$y \leftarrow z_2$
return y

- Decomposition can be with simple scalar operations . . .
  - $x + y$, $x * y$, $\sin(x)$, x^y
- . . . or with higher-level operations for which a chain-rule can be defined
  - $x^\intercal$, $x[i] = y$, XY, e^X
  - E.g. gradient of $\det(X)$: $\det(X)\, X^{-\intercal}$
  - Linear and nonlinear systems of equations (in CasADi!)
  - Initial-value problems in ODE or DAE (in CasADi!)

Differentiate the algorithm!

$z_0 \leftarrow x$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
end for
$y \leftarrow z_K$
return y

$\Rightarrow$

$z_0 \leftarrow x$
$\frac{dz_0}{dx} \leftarrow I$
for $k = 1, \ldots, K$ **do**
  $z_k \leftarrow f_k\left(\{z_i\}_{i \in \mathcal{I}_k}\right)$
  $\frac{dz_k}{dx} \leftarrow \sum_{i \in \mathcal{I}_k} \frac{\partial f_k}{\partial z_i}\left(\{z_i\}_{i \in \mathcal{I}_k}\right) \frac{dz_i}{dx}$
end for
$y \leftarrow z_K$
$J \leftarrow \frac{dz_K}{dx}$
return y, J

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\frac{dz}{dx}, \qquad J = A^\intercal \frac{dz}{dx},$$

Write as a system of linear equations:

$$\frac{dz}{dx} = B + L\,\frac{dz}{dx}, \qquad J = A^\intercal\,\frac{dz}{dx},$$

with

$$z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_K \end{pmatrix}, \quad A = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ I \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} I \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

with I and 0 of appropriate dimensions, as well as the *extended Jacobian*,

$$L = \begin{pmatrix} 0 & \dots & \dots & 0 \\ \frac{\partial f_1}{\partial z_0} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial f_K}{\partial z_0} & \dots & \frac{\partial f_K}{\partial z_{K-1}} & 0 \end{pmatrix},$$

Since $I - L$ is invertible, we can solve for J:

$$J = A^\intercal\,(I - L)^{-1}\,B$$

- Have $J = A^\intercal (I - L)^{-1} B$
- Multiply J from the right: **Forward mode of AD**
  - $\hat{y} := J \hat{x} = A^\intercal (I - L)^{-1} B \hat{x}$
  - Cheap with *forward substitution* of lower triangular $(I - L)$
  - Computational cost: $\approx$ cost of evaluating F
  - Small memory requirements (no storage of L needed)
- Multiply J from the left: **Reverse mode of AD**
  - $\bar{x} := J^\intercal \bar{y} = B^\intercal (I - L)^{-\intercal} A \bar{y}$
  - Cheap with *backward substitution* of upper triangular $(I - L)^\intercal$
  - Computational cost: $\approx$ cost of evaluating F
  - If $F(x)$ is scalar, $\bar{y} = 1$ gives $\nabla_x F(x)$
  - Intermediate operations (or their linearization) must be stored
  - Remedy: Checkpointing

Table of Contents

Checkpointing

- E.g.

$y = f(x)$:
$z_0 \leftarrow x$
for $k = 1, \ldots, 1000$ **do**
    $z_k \leftarrow \sin(z_{k-1})$
end for
$y \leftarrow z_{1000}$

- Divide up!:

$y = f(x)$:
$z_0 \leftarrow x$
for $k = 1, \ldots, 20$ **do**
    $z_k \leftarrow g(z_{k-1})$
end for
$y \leftarrow z_{20}$

$y = g(x)$:
$z_0 \leftarrow x$
for $k = 1, \ldots, 50$ **do**
    $z_k \leftarrow \sin(z_{k-1})$
end for
$y \leftarrow z_{50}$

- Memory for reverse mode decreases from $50 * 20$ to $50 + 20$
- Price: Need to reevaluate $g(x)$ – twice as many sin calls!
- Checkpointing trades memory for reevaluation!

Table of Contents

Calculating complete Jacobians and Hessians

- Jacobians can be calculated by multiplying with n_{col} vectors from the right or n_{row} vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating F
- Hessians can be calculated as Jacobian-of-gradient – exploit symmetry!
- *Much cheaper* if J is sparse, e.g. banded

Exploiting sparsity: illustration

- $J = \begin{bmatrix} * & & & \\ & * & & \\ & & * & \\ & & & * \end{bmatrix}$ $\Rightarrow \hat{x} = [1, 1, 1, 1]$
- $J = \begin{bmatrix} * & & & \\ * & * & & \\ * & & * & \\ * & & & * \end{bmatrix}$ $\Rightarrow \hat{x}_1 = [0, 1, 1, 1], \hat{x}_2 = [1, 0, 0, 0]$
- $J = \begin{bmatrix} * & * & * & * \\ * & & & \\ * & & & \\ * & & & \end{bmatrix}$ $\Rightarrow \hat{x}_1 = [1, 0, 0, 0], \bar{y}_2 = [1, 0, 0, 0]$

Finding a small set of vectors

- NP hard combinatorial problem!
- Graph coloring techniques usually work well (cf. Gebremedhin et al, 2005)

Table of Contents

Software

- Tools are generally divided into:
  - Operator-overloading (OO): Calculate derivatives "piggyback" during numerical evaluation
  - Source-code-transformation (SCT): Generate new source code for derivatives
- Generic tools to differentiate "black-box" code
  - Language-specific: `www.autodiff.org`, Wikipedia
  - ADOL-C (OO), ADIC (SCT), CppAD (OO) for C/C++
  - ADIFOR (SCT), TAPENADE (SCT) for FORTRAN
- AD implemented for domain specific languages (DMS)
  - CasADi (SCT), AMPL (OO), GAMS (OO), JuliaOpt (OO)

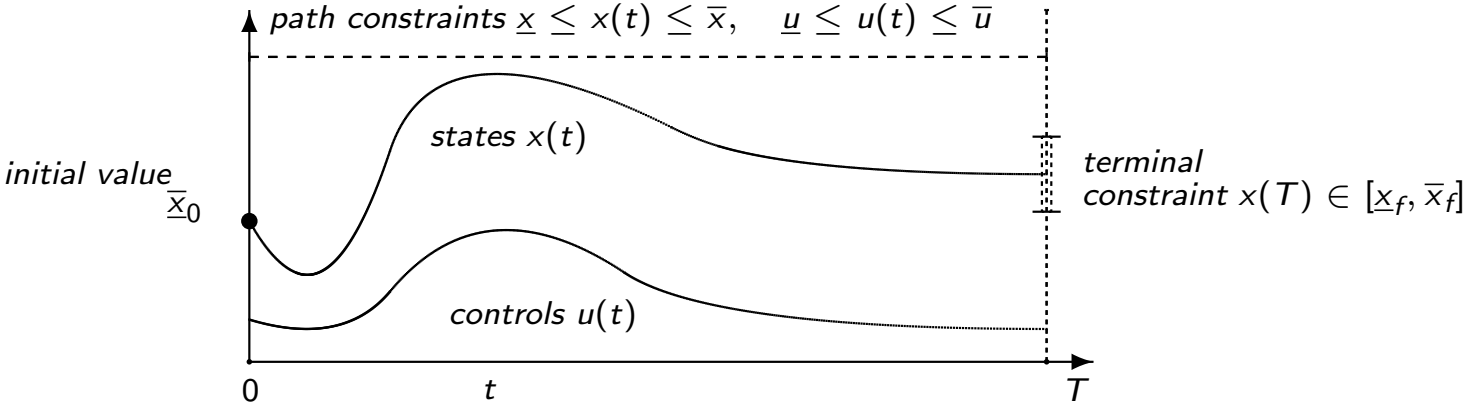
Table of Contents

Summary: AD

- Jacobian-times-vector products cheap using *forward mode AD*
- Vector-times-Jacobian products cheap using *reverse mode AD*
  - *Checkpointing* can avoid excessive memory use
- Complete Jacobians and Hessians: depends on sparsity pattern
  - Worse case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating F
  - Good heauristics exist for complete sparse Jacobians and Hessians
- Software exists for many languages and domain specific languages

Table of Contents

Simplified optimal control problem (OCP)[1]



$$\underset{x(\cdot),u(\cdot)}{\text{minimize}} \quad \int_0^T L(x(t),u(t))\,dt \;+\; E\left(x(T)\right)$$

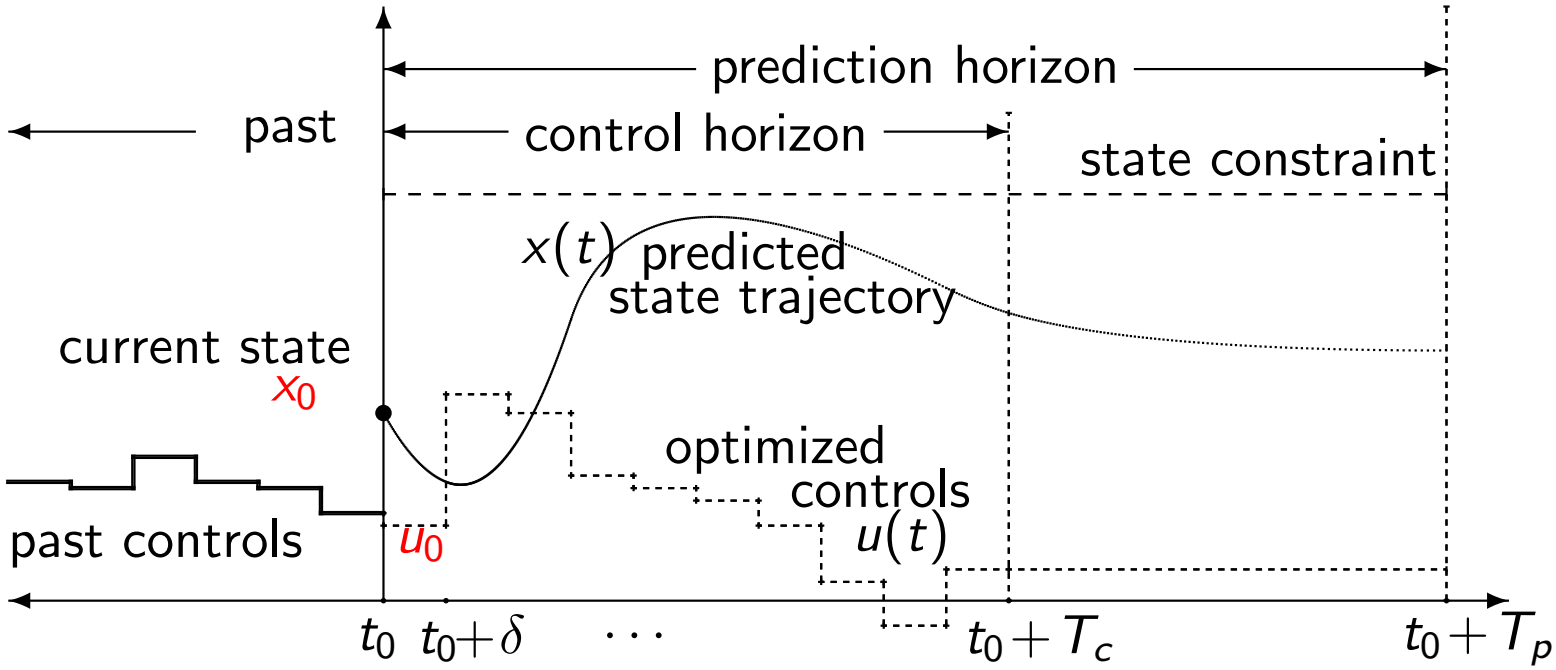
subject to

$$x(0) = \overline{x}_0, \qquad \text{(Initial condition)}$$
$$\dot{x}(t) = f(x(t),u(t)), \quad t \in [0,T], \qquad \text{(System model)}$$
$$\underline{x} \leq x(t) \leq \overline{x}, \quad t \in [0,T], \qquad \text{(Path constraints)}$$
$$\underline{u} \leq u(t) \leq \overline{u}, \quad t \in [0,T], \qquad \text{(Control bounds)}$$
$$\underline{x}_\mathrm{f} \leq x(T) \leq \overline{x}_\mathrm{f} \qquad \text{(Terminal constraints)}$$

[1] A.k.a. "dynamic optimization problem"

Model predictive control (MPC)

- Solve a sequence of OCPs to determine the optimal control action:



- Give first control move u_0 back to real-world system.
  Move horizon.
- Result: **Feedback law** $u_0(x_0)$.
  Allows to react to disturbances and modeling errors.

Slightly more realistic optimal control problem

Optimal control problem in differential-algebraic equations:

$$\begin{aligned}
\text{minimize} \quad & \int_0^T L(x(t), z(t), u(t), p, t)\, dt + E(x(T), p) \\
\text{subject to} \quad & \begin{cases} \dot{x}(t) &= f(x(t), z(t), u(t), p, t) \\ 0 &= g(x(t), z(t), u(t), p, t) \end{cases} \qquad t \in [0, T] \\
& \underline{x} \le x(t) \le \overline{x}, \quad t \in (0, T) \\
& \underline{u} \le u(t) \le \overline{u}, \quad t \in [0, T] \\
& x(0) = \overline{x}_0, \quad \underline{x}_f \le x(T) \le \overline{x}_f, \quad \underline{p} \le p \le \overline{p},
\end{aligned}$$

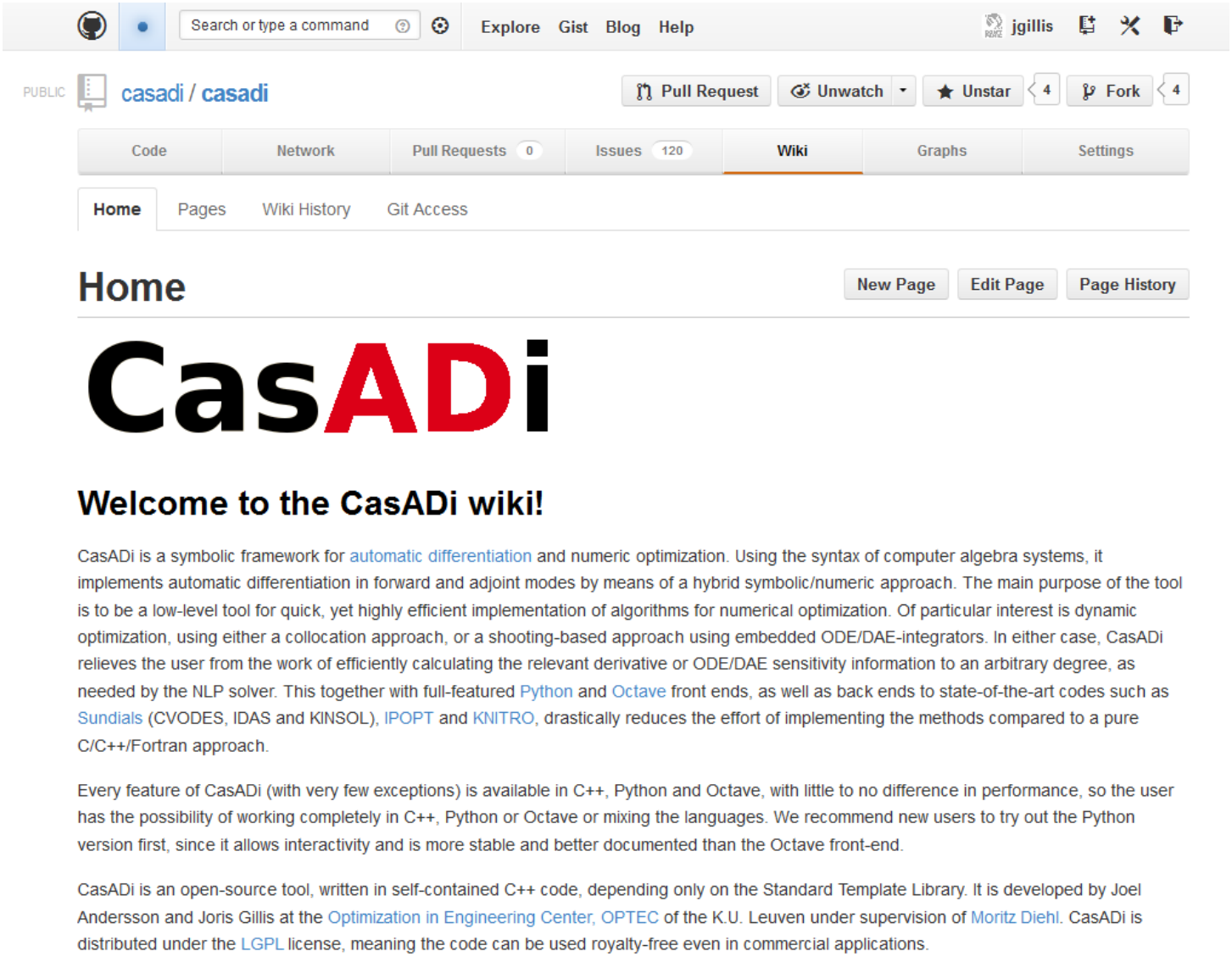
$x(\cdot) \in \mathbb{R}^{n_x}$	Differential states	$L(\ldots)$	Lagrange cost term
$z(\cdot) \in \mathbb{R}^{n_z}$	Algebraic variables	$E(\ldots)$	Mayer cost term
$u(\cdot) \in \mathbb{R}^{n_u}$	Controls	$f(\ldots)$	Differential equations
$p \in \mathbb{R}^{n_p}$	Parameters	$g(\ldots)$	Algebraic constraints

Table of Contents

CasADi

- Started as an implementation of AD using CAS-like syntax
- Current scope: Numerical optimization general (or just AD!)
- In particular: Facilitates the solution of optimal control problems (OCPs)
  - *Facilitates*, not actually *solves* the OCPs
  - Write state-of-the-art OCP algorithms with very little code!
- Available for C++, Python, MATLAB/Octave
- Free & open-source (LGPL), also for commercial use
- Project started in December 2009, now (almost) at version 3.0
- Main developers: Joel Andersson and Joris Gillis

casadi.org → github.com



Algorithmic differentiation (AD) in CasADi (Andersson, 2013)

- Decomposes algorithms into a sequence of either **scalar** or **sparse matrix-valued atomic operations**
- New symbolic expressions generated for derivatives: "Source code transformation" approach
  - Forward mode: Jacobian-times-vector products
  - Reverse mode: vector-times-Jacobian products
  - Sparse Jacobians and Hessians via:
    - Automatic detection of sparsity pattern (nontrivial!)
    - Graph coloring techniques to exploit sparsity & symmetry
  - Arbitrary order
- Supports **high-level operations**: matrix-operations, implicit functions, calls to DAE integrators

Table of Contents

Quadratic programs (QPs)

- User needs to write the problem in the following standard form:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & \underline{g} \leq g(x) \leq \overline{g} \\ & \underline{x} \leq x \leq \overline{x} \end{array}$$

  where $f(x)$ is a convex quadratic function and $g(x)$ is a linear function.
- QP solvers available: qpOASES, OOQP, CPLEX, GUROBI
- Solver "plugins" can be added post-installation
- CasADi automatically generates matrix sparsities
- GUROBI & CPLEX: a subset of x can be integer-valued!

Nonlinear programs (NLPs)

- User needs to write the problem in the following standard form:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & \underline{g} \leq g(x) \leq \overline{g} \\ & \underline{x} \leq x \leq \overline{x} \end{array}$$

  where $f(x)$ and $g(x)$ twice continuously differentiable functions

- NLP solvers available: IPOPT, SNOPT, KNITRO, WORHP, blockSQP, Bonmin, CasADi's own
- Solver "plugins" can be added post-installation
- CasADi automatically generates derivative information
- Bonmin & KNITRO: a subset of x can be integer-valued!

Integrators

- Solves initial-value problems in ordinary or differential-algebraic equations (ODE/DAE)
- Given a DAE with fixed initial values coupled to another (linear) DAE with fixed terminal value (both with quadratures):

$$
\begin{cases}
\dot{x} &= f_x(x, z, p, t) \\
0 &= f_z(x, z, p, t) \\
\dot{q} &= f_q(x, z, p, t) \\
 & \\
-\dot{\tilde{x}} &= \tilde{f}_x(\tilde{x}, \tilde{z}, \tilde{p}, x, z, p, t) \\
0 &= \tilde{f}_z(\tilde{x}, \tilde{z}, \tilde{p}, x, z, p, t) \\
-\dot{\tilde{q}} &= \tilde{f}_q(\tilde{x}, \tilde{z}, \tilde{p}, x, z, p, t)
\end{cases}
\qquad t \in [0, T]
\qquad
\begin{cases}
x(0) &= x_0 \\
q(0) &= 0 \\
 & \\
\tilde{x}(T) &= \tilde{x}_0 \\
\tilde{q}(T) &= 0
\end{cases}
$$

- An *integrator in CasADi* is a mapping from $\{x_0, p, \tilde{x}_0, \tilde{p}\}$ to $\{x(T), q(T), \tilde{x}(0), \tilde{q}(0)\}$
- Enables **automatic forward and adjoint sensitivity analysis**

Table of Contents

Summary: CasADi

- Open-source framework for numerical optimization
- Central feature I: general-purpose implementation of AD
- Central feature II: solve standard problems conveniently
  - QPs
  - NLPs
  - Initial-value problems in ODE/DAE
  - Everything you need for optimal control!
- Video tutorial: `http://tutorial.casadi.org/`