Optimization with CasADi and Modelica/FMI

Summer School on Robust Model Predictive Control with CasADi, Freiburg (Germany), September 15-19, 2025

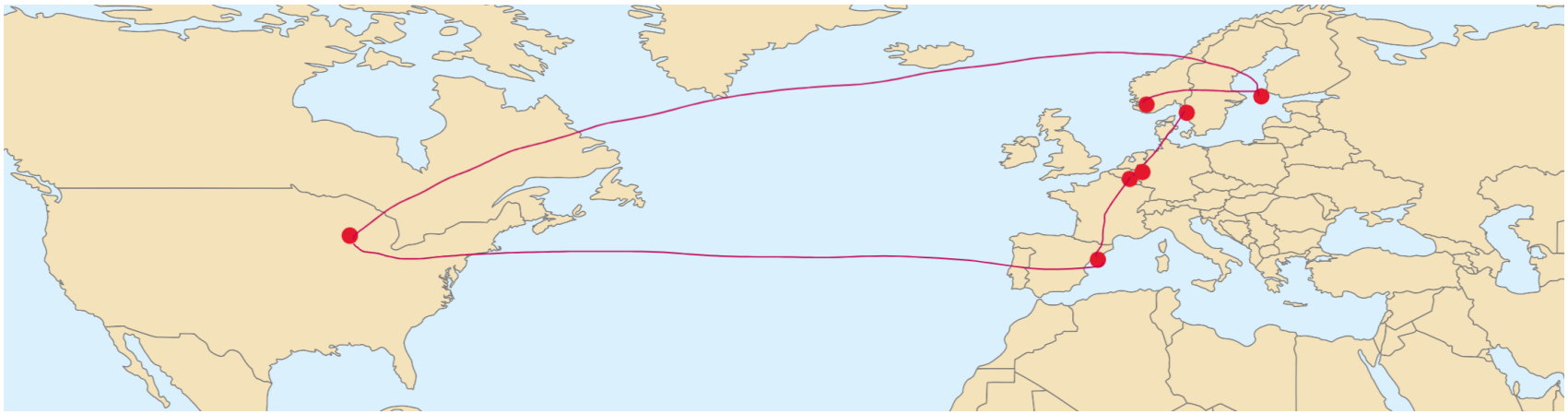
Joel Andersson

FMIOPT AS (Stavanger, Norway)
joel@fmiopt.com

17 September 2025

Bio: Joel Andersson

- 1982: Born in Sweden, undergrad at Chalmers
- 2007–2018: M.Sc. thesis on parameter estimation for batch chromatography at FZ Jülich
- 2008–2013: **Ph.D with Moritz Diehl** at KU Leuven
  - Optimization w/ Modelica
  - Main outcome: **CasADi**
- 2015–2018: **Postdoc with James B. Rawlings** at University of Wisconsin-Madison
- 2018–2020: Optimization algorithms for cancer treatment planning at Philips
- 2020–: Self-employed, mainly Modelica
  - USA → Åland (FI) → Norway
  - New software: **FMIOPT** (2024–)



Topics covered

- What is Modelica?
- What is the Functional Mock-up Interface (FMI)?
- Modelica/FMI interoperability in CasADi
- Tutorial
  - Derivative/sensitivity calculations with imported FMI models
  - Optimization with imported FMI models
  - FMI model export from CasADi

1. Modelica and FMI
2. Calculating derivatives
   - Algorithmic differentiation (AD)
   - Derivative information in FMUs
3. CasADi
   - Modelica/FMI interoperability
4. Outlook
5. Tutorial

Table of Contents

What is Modelica?

- Object-oriented, equation-based, modeling language for dynamical systems
- Enables assembling complex dynamic models from reusable components
- Open standard, supported by multiple software vendors (Dymola, OpenModelica, Wolfram SystemModeler, Modelon Impact, . . .)
- Large ecosystem of free and commercial model libraries
- Multi-domain, especially popular in automotive, process control, building management (heating/ventilation/cooling, HVAC)

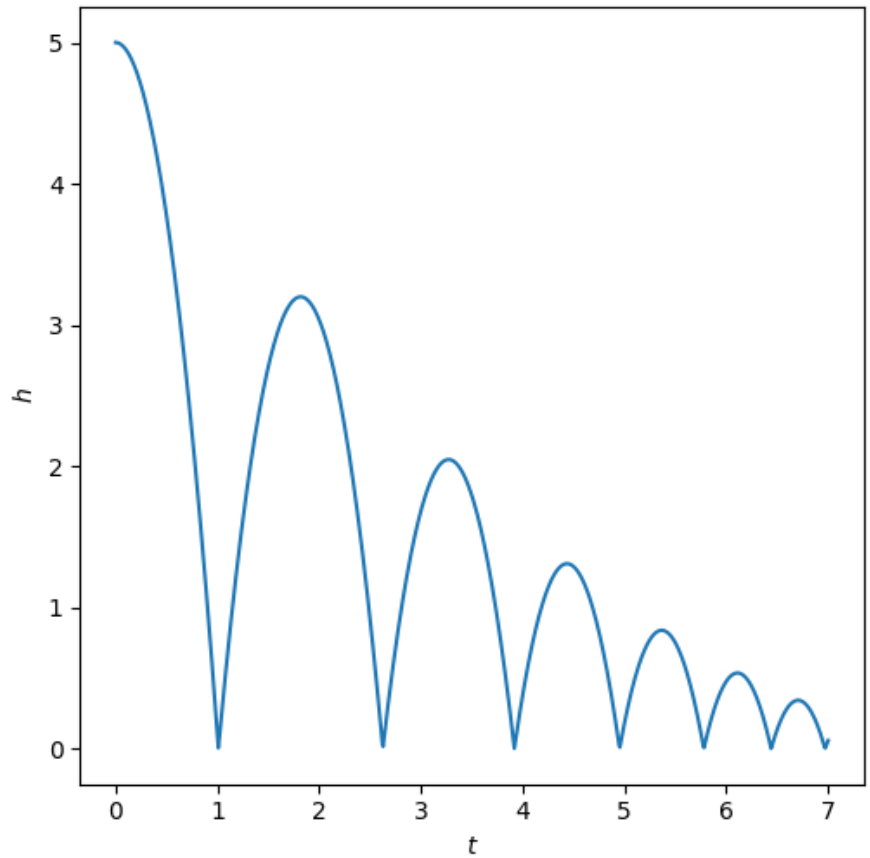
Example of industrial use of Modelica: Formula 1

- Modelica used to model whole system: engine, gears, tires, aero, track, ...
- Design changes multiple times per season
- Software-in-the-loop, driver-in-the-loop simulations

Hybrid system modeling

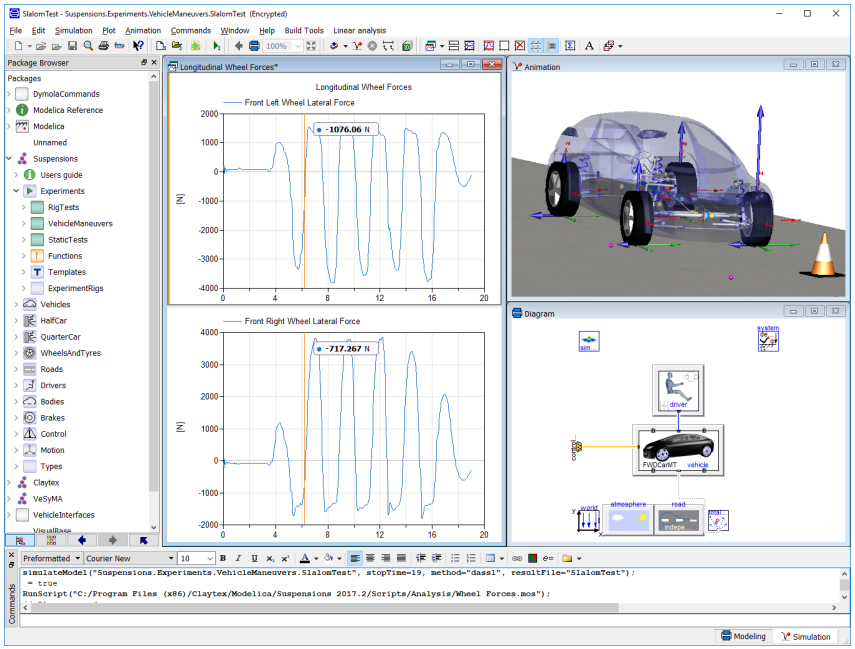
- Modelica makes it easy to model discrete events
- Simulate automatic controllers, model state-machines
- A bouncing ball is the "Hello, world!" in Modelica:

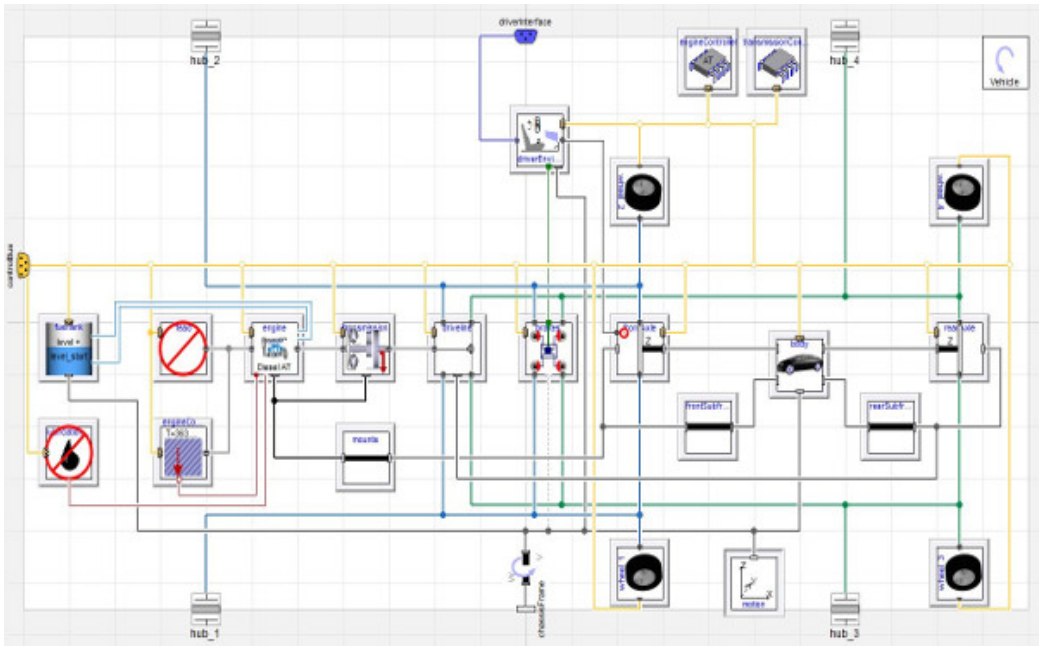
```
model BouncingBall "Bouncing ball"
  type Height=Real(unit="m");
  type Velocity=Real(unit="m/s");
  parameter Real e=0.8
    "Coefficient of restitution";
  parameter Height h0=1.0
    "Initial height";
  Height h;
initial equation
  h = h0;
equation
  v = der(h);
  der(v) = -9.81;
  when h<0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBall;
```



Complex systems can be modeled in GUIs or in code

- Complex dynamic models are created by connecting components in block diagrams or by writing Modelica code
- E.g. in Dymola:





Modelica: Dynamic model

- The Modelica language is expressive:
  - Variables have different units, types (real, integer, strings), *variability* (continuous, discrete), *causality* (input, output, internal), ...
  - Equations: Differential, algebraic, initial, ...
  - Events triggered at certain times, conditionally, periodically, ...
  - Object oriented features: Classes, inheritence, records, connectors, ...
- Can always be *compiled* into a hybrid ODE or DAE
  - Simplified: ODE augmented with output equations, zero-crossing functions and event transition dynamics

$$\begin{cases} x(t) = \lim_{\tau \to t^-} E_j(x(\tau)) & \text{if } \exists j : Z_j(x(t)) < 0 \\ \dot{x}(t) = f(x(t), u(t)) & \text{otherwise} \end{cases} \tag{1}$$
$$y(t) = h(x(t), u(t))$$

  - Process involving sorting, index-reduction, state-selection, etc.

Modelica Association

- Non-profit
  - International meetings every two years (US/Asian meetings off years)
  - Mix of industry and academia/labs
- Large interest in optimization within the Modelica community!
  - CasADi – most attended tutorial (out of 15) at Modelica 2025!
  - Starting to see optimization results in industrial applications
- Maintains Modelica Language and Libraries as well as:
  - **FMI**: Functional Mock-up Interface
  - **SSP**: System Structure and Parameterization
  - DCP: Distributed Co-simulation Protocol
  - eFMI: Functional Mock-up Interface for Embedded Systems

Functional Mock-up Interface (FMI)

- Functional Mock-up Interface (FMI) is a standard for representing model dynamics: Supported by 250+ Modelica and **non-Modelica** tools
- FMI defines Functional Mock-up Unit (FMU), which typically support
  - Co-simulation: Simulate the system from time t_k to t_{k+1}, and/or
  - Model-exchange: Evaluate $\dot{x}$, y, event dynamics
- Especially since FMI 3.0, possible to extract all the model information needed for efficient numerical optimal control, but with *many pitfalls*
  - Model issues, e.g. non-smoothness, non-monotonicity
  - Toolchain issues, e.g. calling overhead, limited accuracy, derivatives
  - Wealth of industry relevant problems to solve

System Structure and Parameterization (SSP)

- Define complete systems, consisting of components and interconnections
- Components can be FMUs or other SSP
- Gaining popularity – opportunity for optimization?
- Often possible to split up monolithic system models into components
  - Expose more model structure than possible with FMI alone
  - More efficient Jacobian and Hessian calculations
  - Analytic derivatives for "most" of the model
  - Better, more granular diagnostics
  - Lifted NLP formulations? (Albersmeyer2010)

Table of Contents

Recall: NLP software

- Direct methods transcribe OCPs into a nonlinear program (NLP):

$$\begin{aligned} \underset{X \in \mathbb{R}^{N_X}}{\text{minimize}} \quad & F(X) \\ \text{subject to} \quad & G(X) = 0, \quad \underline{X} \leq X \leq \overline{X} \end{aligned}$$

- Second-order methods (SQP or nonlinear interior point) typically need:
  1. Undifferentiated functions: $F(X)$ and $G(X)$
  2. First order derivatives: $\nabla_X F(X)$ and $\frac{\partial G}{\partial X}(X)$
  3. Second order derivatives (Λ given):
     $$\nabla_X^2 \left(F(X) + \Lambda^T G(X) \right) = \frac{\partial}{\partial X} \left(\nabla_X F(X) + \Lambda^T \frac{\partial G}{\partial X}(X) \right)$$

Reasonable methods for calculating derivatives

1. Finite differences (FD), sometimes
2. Algorithmic differentiation (AD)

Finite differences

Consider a function $y = f(x) : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ 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
  - One additional evaluation of f for $J(x)\,\hat{x}$
- – Poor accuracy, need to carefully choose t:
  - Small $t \Rightarrow$ *cancellation errors*
  - Large $t \Rightarrow$ *approximation errors*
- \+ Can get higher accuracy with higher-order schemes (e.g. 5-point stencils)
- – No efficient way to calculate $\hat{y}^\intercal\, J(x)$

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

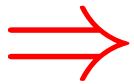
$$y = \sin(\sqrt{x})$$

$z_0 \leftarrow x$
$z_1 = \sqrt{z_0}$
$z_2 = \sin z_1$
$y \leftarrow z_2$
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, $\det(X)$
  - Linear and nonlinear systems of equations
  - Initial-value problems in ODE or DAE $\Rightarrow$ E.g. Shooting methods
  - Nonlinear programs $\Rightarrow$ E.g. calculate confidence intervals
  - **FMUs** (another differentiable building block in optimization?)

Idea: Differentiate the algorithm for $f(x)$ w.r.t. x

$z_0 \leftarrow x$
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



$z_0 \leftarrow x$
$\frac{dz_0}{dx} \leftarrow I$
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

For fixed x, write as a system of linear equations:

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

For fixed x, 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^T from the right: **Reverse (adjoint) 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
    - ★ Potentially large memory use
    - ★ Decrease memory use using *checkpointing*

Checkpointing – trade memory for recalculations

- 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!
  - Small cost if you are calculating multiple derivative directions
- In CasADi: Create nested functions

Calculating Jacobians and Hessians

- Jacobians can be calculated by multiplying with n_{col} unity vectors from the right or n_{row} unity vectors from the left
- Worst-case: $\approx \min(n_{\text{row}}, n_{\text{col}})$ times cost of evaluating f
- *Much cheaper* if J is sparse, e.g. banded. Illustration:

$$J = \begin{bmatrix} * & & & \\ & * & & \\ & & * & \\ & & & * \end{bmatrix} \quad \Rightarrow \hat{x} = [1,1,1,1]$$

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

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

- Good heauristics exist based on graph coloring (cf. Gebremedhin et al, 2005)
- Hessians: Jacobian of reverse mode algorithm
  - Exploit symmetry, especially in graph coloring

Derivative information in FMUs

- Recall: FMU described by ODE state-space model with event dynamics

$$\begin{cases} x(t) = \lim\limits_{\tau \to t^-} E_j(x(\tau)) & \text{if } \exists j : Z_j(x(t)) < 0 \\ \dot{x}(t) = f(x(t), u(t)) & \text{otherwise} \end{cases} \tag{2}$$
$$y(t) = h(x(t), u(t))$$

- FMI 2.0: Have forward derivatives, Jacobian sparsity patterns for f and h
- FMI 3.0: Also adjoint derivatives
- No second order derivatives, but you can do finite differences of adjoint
- (Some) Hessian sparsity information
- No derivative information for events (E_j and Z_j) $\Rightarrow$ Use FD?
  - Event time implicitly defined by $Z_j(x(t)) = 0$
- "Good enough" for dynamic optimization, with many caveats

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
  - CasADi, JAX, Julia, Pyomo, MATLAB, ...
- Can be applied to FMUs $\Rightarrow$ **More in tutorial**

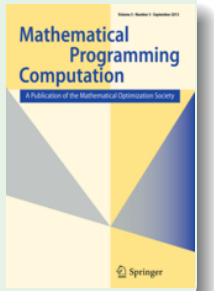
Table of Contents

CasADi

[Andersson et al., 2019]

- A freely available, open-source framework for numerical optimization
- In particular: *Facilitates* OCP→NLP transcription by providing the **building blocks** for efficient optimal control ("100 instead of 10k lines of code")
  - Not an OCP **solver**, but a framework for writing solvers
- Developed since 2009 by Joel Andersson and Joris Gillis, then graduate students with Prof. Moritz Diehl at KU Leuven
- Now widely used in **academic research**, **teaching** and **industry**
  - **5000**+ citations on CasADi implementation papers to date
- Version 3.7.2 (released September 10, 2025) available from `casadi.org`

CasADi – A software framework for nonlinear optimization and optimal control by J Andersson, J Gillis, G Horn, JB Rawlings and M Diehl
Math. Prog. Comp. 11(1):1–36 (2019)
Read it online: `http://rdcu.be/2SS7`



CasADi components

- Symbolic framework written in self-contained C++
- State-of-the-art **algorithmic differentiation (AD)** [Andersson, 2013]
  - **Source code transformation** AD framework supporting **high-level** operations: sparse linear algebra, (non)linear solvers, integrators, ...
  - AD can be used for **gradient-based optimization**, implicit integrator schemes, ODE/DAE **sensitivity analysis**, DAE index reduction, Lagrange-based modeling, Lyapunov differential equations, ...
- In-house solvers and interfaces to numerical software, e.g.
  - **(MI)NLP solvers** (IPOPT, SNOPT, KNITRO, WORHP, FATROP, Bonmin, CasADi's own codes)
  - **(MI)QP solvers** (qpOASES, GUROBI, CPLEX, OOQP, HPMPC, ...)
  - **ODE/DAE integrators** (SUNDIALS, CasADi's own codes)
- Front-ends to C++, Python, MATLAB/Octave

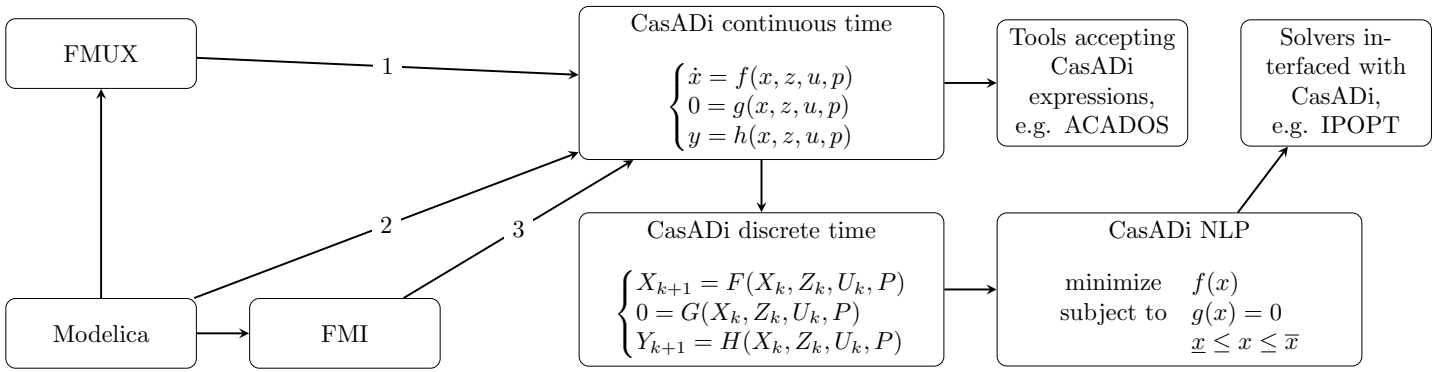
Important feature: C code generation

```
...
/* #1516: output[266][0] = @12 */
casadi_copy(w12, 930, res[266]);
/* #1517: {@14, @15, @16} = f_mx(@0, @1, @2, @3) */
arg1[0]=w0;
arg1[1]=w1;
arg1[2]=w2;
arg1[3]=w3;
res1[0]=w14;
res1[1]=w15;
res1[2]=w16;
if (casadi_f1(arg1, res1, iw, w, 0)) return 1;
/* #1518: @14 = (@14-@8) */
for (i=0, rr=w14, cs=w8; i<930; ++i) (*rr++) -= (*cs++);
/* #1519: @15 = (@15-@9) */
...
```

- Export self-contained C code without dynamic memory allocation
- Why C code generation?
  - Speedup: Typically 3-4x faster than virtual machine code
  - Portability: Embedded optimization
  - Introspection: C code mirrors CasADi's virtual machine code

Modelica interoperability from the start of CasADi

- Original motivation to develop CasADi was to solve optimization problems formulated in Modelica (vICERP project, around 2010)
- Different approaches have been implemented:



1. FMUX: A symbolic extension of FMI 1.0 [JModelica.org, OpenModelica] (incomplete, no longer maintained)
2. Symbolic, tool-specific couplings [Modelon Impact, Pymoca]
3. Via standard model-exchange FMI 2.0 and FMI 3.0 [several tools]

Note: Symbolic interfaces, e.g. in Modelon Impact

Much faster, works with codegen in e.g. acados, but limited to subset of Modelica

New native CasADi interface to standard FMI (2021–) [Andersson, 2023]

- *Flexible* interface to create *twice differentiable* functions that can be embedded into symbolic expressions
- Supports all variable types for parameterizing problems, but only subsets of the real-valued variables can enter in CasADi functions
- Efficient, parallelized calculations of first and second order derivatives:
  - Directional derivatives: Forward, adjoint, forward-over-adjoint
  - Sparse Jacobians and Hessians
  - Diagnostics to check correctness of derivative information
- Supports FMI 2.0 and FMI 3.0

Work in progress

- Continuously improving: Many industrial applications
- Many missing pieces: Events support, C code export, ...

FMI export and reimport in CasADi

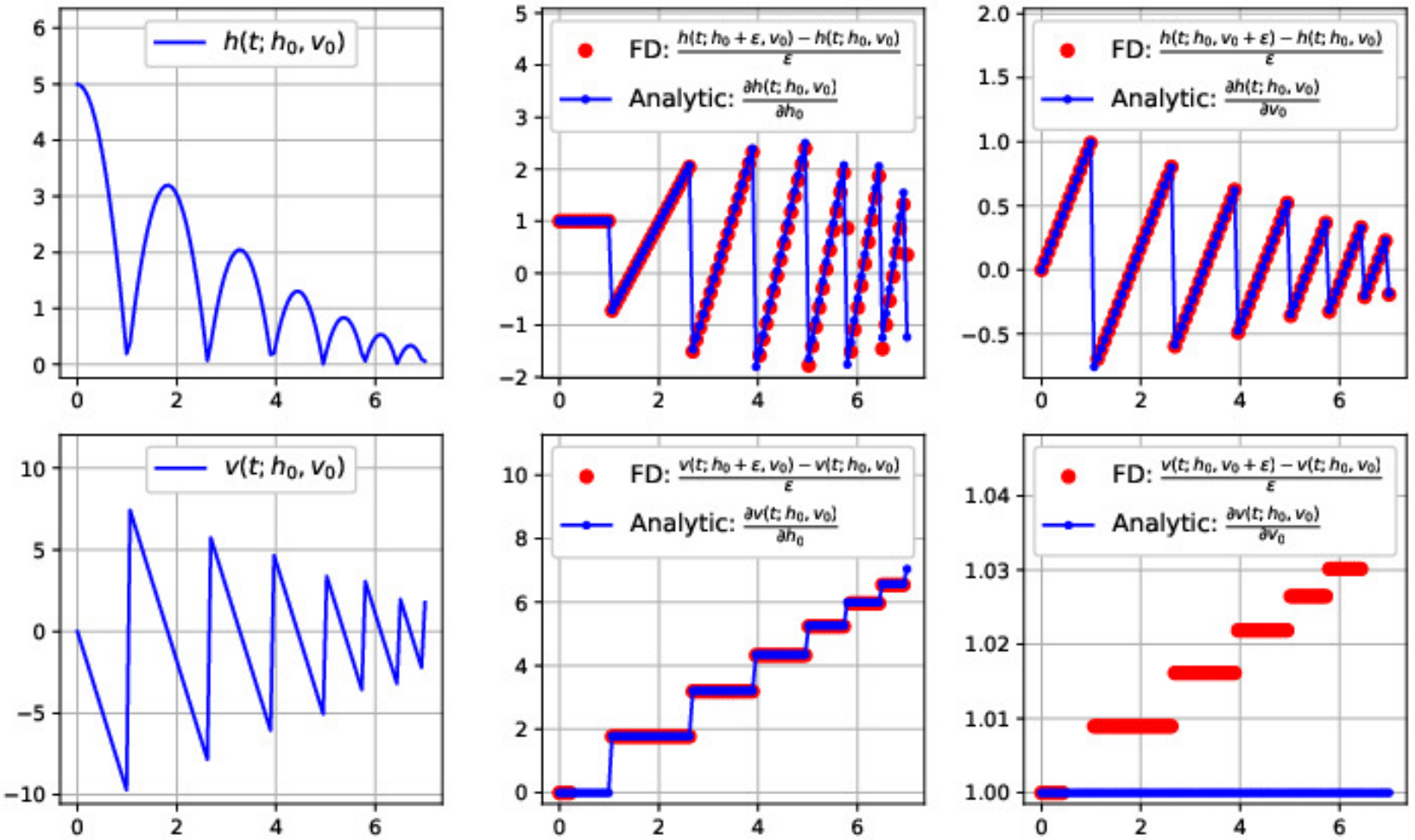
- CasADi can generate FMUs from symbolic expressions
  - Standard FMI 3.0 with efficient derivative calculation
  - Simulate model in e.g. FMPy, Modelica tools, Simulink
- New feature in CasADi, but builds on mature functionality, in particular AD and C code export
- CasADi FMUs contain serialized expressions adhering to a *layered standard*
  - If FMU is imported into CasADi (can be a different installation), get the expressions back – no information is lost

Sensitivity analysis & optimization for systems with events

- Event dynamics are common in realistic Modelica/FMI models
  - How to solve optimal control problems for such systems?
  - Missing in FMI: Derivatives for event triggering, state reset conditions
- In Julia: `https://github.com/ThummeTo/FMISensitivity.jl`
  - Standard FMUs, finite differences for missing derivatives
  - Proposal to supplement the FMI standard [Thummerer et al., 2025]
- In CasADi: Focus on **symbolic** Modelica import
  - Analysis and C code generation
  - Compatibility with e.g. NOSNOC [Nurkanovic and Diehl, 2022]
  - Ongoing work, main challenges:
    1. Missing event support in CasADi, in particular for sensitivity analysis
    2. Missing standardized symbolic model import supporting events

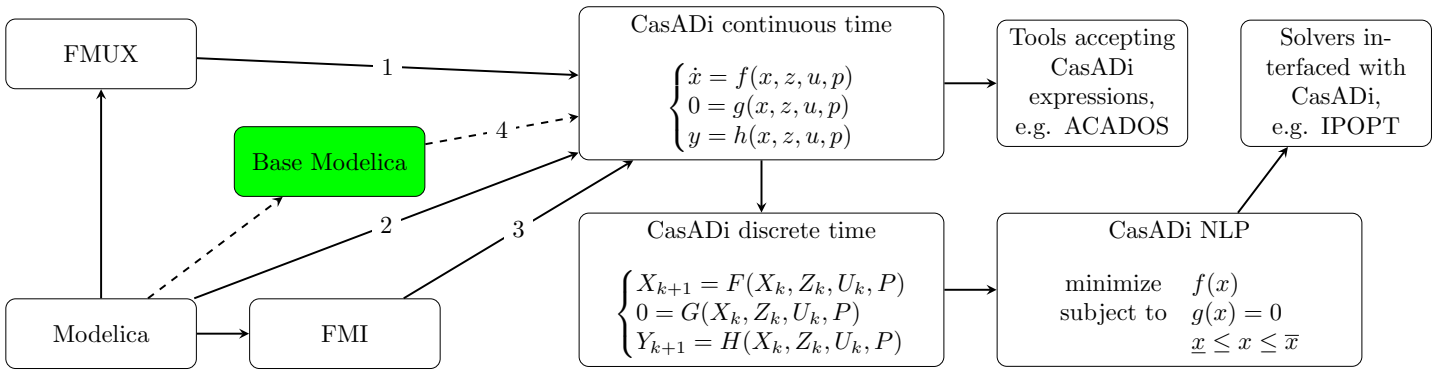
Challenge 1: Event support in CasADi

- Support for analytic sensitivity analysis for systems with events [Andersson and Goppert, 2024]
- Bouncing ball: Sensitivity w.r.t. initial height, initial velocity:



Challenge 2: Symbolic model import supporting events

- Can we find a standardized way to import Modelica models symbolically?
- Before numerical simulation, Modelica tools compile models into a simplified form without constructs such as classes, inheritance, connectors, etc.
- *Base Modelica* [Kurzbach et al., 2023]: Standardize this form across tools
  - Multiple tools, most mature implementation in OpenModelica
- Opportunity: *Much* easier to implement symbolic import



- Open-source project: Rumoca [Condie et al., 2025]
  - Compile Base Modelica to CasADi, sympy, JAX, ...

Table of Contents

Outlook: Optimization with Modelica/FMI

- Maturing FMI export and import (CasADi, Julia, Pyomo), with derivatives
  - Progressed past *simple* problems (where it's better to rewrite model)
- Efforts to make FMI more suitable for optimization
  - Layered standard for sensitivity information for events [Thummerer et al., 2025]
  - Layered standard for index-1 DAEs in FMI?
    - ★ Currently, model-exchange FMUs define ODEs, resulting in overhead and accuracy loss due to Newton iterations inside model equations
  - Layered standard for causalized symbolic expressions in FMI?
    - ★ Should be tool-agnostic (Julia, CasADi, JAX, Pyomo, . . .)
- New standard for specifying optimal control problems in FMI?
- Synnergies between "competing" software tools

Industrial workflows for dynamic optimization

- Modeling environments like Modelica enables industrial users to formulate *thousands* of different, but similar models (e.g. HVAC for building)
- If you want to solve dynamic optimization problems at scale, you need to think about **workflows**
  - "Why is there a NaN in my Jacobian?"
  - "Can I trust the derivatives from tool X?"
  - "Why is dual infeasibiltity not going down?"
  - "Why does the solver converge to a local minimum?"
  - "How should I weigh my objectives?"
  - "How do I initialize my problem?"
- How do you **systematically** solve these problems?
  - Can the process be automated or faciliated?
- Can we standardize the interfaces?
  - Solve same problem with different tools (e.g. CasADi and Julia)
  - Solve different problems for same model

New optimization framework: **FMIOPT** (2024–)

- Higher-level, **scalable** framework, funded by industry
- Robust **workflows** for optimization that start with existing simulation models
- State-of-the-art, parallelized numerics compatible with model-exchange `FMI` (and CasADi)
- Excellent **diagnostics**: When something goes wrong, give **actionable** feedback on how to resolve it
- Clean, method-agnostic REST API separating front end and back end

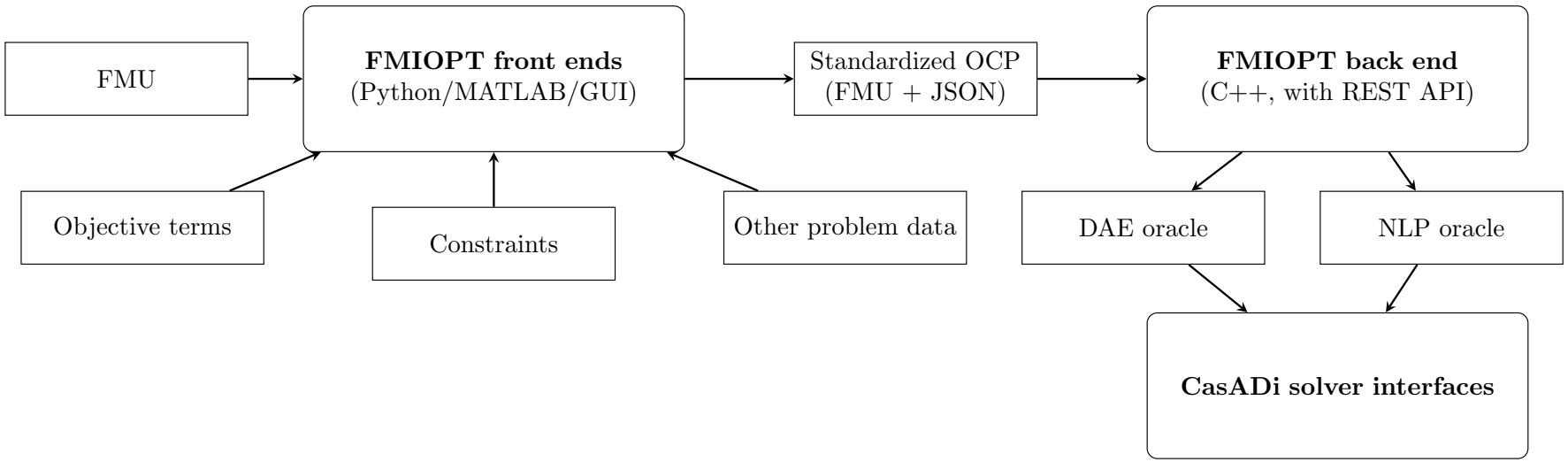


Table of Contents

Tutorial

https://github.com/jaeandersson/rmpc_summer_school_2025

Andersson, J. (2013).

A General-Purpose Software Framework for Dynamic Optimization.

PhD thesis, Arenberg Doctoral School, KU Leuven.

Andersson, J. (2023).

Import and Export of Functional Mockup Units in CasADi.

In *Proceedings of the 15th International Modelica Conference*, volume 2855, pages 321–326.

Andersson, J. and Goppert, J. (2024).

Event support for DAE simulation and sensitivity analysis in CasADi for use with Modelica and FMI.

In *Proceedings of the American Modelica Conference 2024*.

Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. (2019).

CasADi: a software framework for nonlinear optimization and optimal control.

Math. Prog. Comp., 11(1):1 – 36.


Condie, M., Woodbury, A., Goppert, J., and Andersson, J. (2025).

Rumoca: Towards a Translator from Modelica to AlgebraicModeling Languages.

In *Proceedings of the 16th International Modelica & FMI Conference*.


Kurzbach, G., Lenord, O., Olsson, H., Sjölund, M., and Tidefelt, H. (2023).

Design proposal of a standardized Base Modelica language.

In *Proceedings of the 15th International Modelica Conference*, volume 2855, pages 469–478.


Nurkanovic, A. and Diehl, M. (2022).

NOSNOC: A Software Package for Numerical Optimal Control of Nonsmooth Systems.

IEEE Control Systems Letters, 6:3110–3115.


Thummerer, T., Olsson, H., Song, C., Gundermann, J., Blochwitz, T., and Mikelsons, L. (2025).

LS-SA: Developing an FMI layered standard for holistic & efficient sensitivity analysis of FMUs.

In *Proceedings of the 16th International Modelica & FMI Conference*.