Import and Export of Functional Mock-up Units in CasADi

Joel A. E. Andersson[1]

[1]Freelance software developer and consultant, Madison, Wisconsin (USA) `joel@jaeandersson.com`

Abstract

This paper presents the recently added support for import and export of functional mock-up units (FMUs) in CasADi, an open-source software framework for numerical optimization. Of particular interest is the efficient calculation of derivatives, especially in the context of sensitivity analysis and dynamic optimization. We show how the import interface allows for both first and second derivatives can be efficiently and accurately calculated and – importantly – validated for correctness. We also outline the FMU export interface, which leverages CasADi mature and efficient support for forward and adjoint derivative calculation and C code generation. Finally, potential future developments of the support are discussed.

Keywords: CasADi, FMI, FMU, Modelica, optimal control

1 Introduction

The work presented here is intended to be general-purpose, but will typically include some sort of optimization formulation with a high-fidelity simulation model adhering to the FMI standard, cf. Section 1.2 below. Examples of such applications include:

- Parameter estimation applications, which may use parametric sensitivty approaches to obtain estimates of confidence intervals for estimated parameters
- A wide range of different optimal control formulations, i.e. problems with free control trajectories to be determined by the optimization algorithm
- Optimization-based control techniques such as (nonlinear) model predictive control (MPC), including their deployment on embedded systems
- Steady-state optimization formulations, i.e. problems that lack time derivatives or have the time derivatives fixed to some value (typically zero)

For more details on the implementation of such algorithms and on possible applications, we refer to the various textbooks on the topic, including (Biegler 2010) and (Rawlings, Mayne, and Diehl 2020).

1.1 CasADi

CasADi (J. A. E. Andersson et al. 2019) is an open-source software package for C++, Python, MATLAB and Octave. CasADi offers a flexible approach to solve numerical optimization problems in general and numerical optimal control in particular. At the lowest level, it offers all the building blocks needed to efficiency address all the problem formulations listed above. At the core of the package is a symbolic expression framework implementing *algorithmic differentiation* (AD) in forward and reverse (adjoint) modes. CasADi's symbolic expressions can contain embedded *function objects*, which offer a standard interface to generic, differentiable functions. These function objects can be defined in a number of ways, including by other symbolic expressions, by systems of nonlinear equations, by initial-value problems in differential-algebraic equations or by external function calls.

1.2 FMI

The *functional mock-up interface* (FMI) (Modelica Association 2020; Modelica Association 2023) is an open standard for exchanging information about dynamic system models between tools. The format specifies the structure of self-contained zip archives called *functional mock-up units* (FMUs), The FMUs contain, in particular, an XML file with static meta information and a C library for evaluating model equations and their derivatives. The C library is designed for either static linking or dynamic linking and can be distributed either in source form or as a compiled dynamically linked library (DLL), available in the FMU.

The FMI standard describes connections either at the dynamic equation level, referred to as *model exchange*, or on an input-output level at discrete communication time points, referred to as *co-simulation*. In this work, we are mainly concerned with the connections at the level of dynamic equations and references to "FMI" implicitly implies FMI for model exchange.

Derivative information in FMUs

FMI 2.0 (Modelica Association 2020) specifies two types of derivative information:

- Firstly, the FMI XML API contains information about which output variables depend on what input variables (`dependencies` attribute) and whether this dependency is linear (`dependenciesKind` attribute).
- Secondly, the FMI C API includes the routine for calculating (forward) directional derivatives i.e. Jacobian-times-vector products, for selected subsets of the FMU inputs and outputs

(`fmi2GetDirectionalDerivative` function).

Together, these two pieces of information can be used to obtain sparse Jacobians, as outlined in Section 3.2.

In FMI 3.0 (Modelica Association 2023), where forward directional derivatives are calculated using the routine `fmi3GetDirectionalDerivative`, the C API is extended with the ability to calculate adjoint directional derivatives, i.e. Jacobian-transposed times vector products, via the routine `fmi3GetAdjointDerivative`. Furthermore, the variable dependency information in the XML API can now be refined at runtime, with the addition of `fmi3GetVariableDependencies` in the C API.

2 Other integration efforts of CasADi and Modelica

This work does not represent the first time Modelica models has been integrated into CasADi. Indeed, it was a Modelica application that motivated the start of the CasADi project in 2009 (Ahlbrink et al. 2009). The first integration effort, between JModelica.org and CasADi, was described in (J. Andersson et al. 2011). That approach was based on the FMI version 1.0, where the XML model description had been extended with a symbolic representation of the model equations. Later, support for export of the same format was also added to OpenModelica (Shitahun et al. 2013).

Eventually, JModelica.org replaced the XML-based format with a tighter integration based on a Java interface generated using SWIG (Beazley 2003), an approach which is also used in OCT, JModelica.org's closed-source successor code from AB Modelon. As of this writing, the CasADi-backend of OCT represents the most mature *symbolic* coupling between generic Modelica models and CasADi.

Two additional interfaces between Modelica and CasADi are available via the open-source Pymoca[1] and Cymoca[2] packages on Github. Both these packages include native CasADi-based backends, using CasADi's Python and C++ APIs, respectively.

3 Importing FMUs into CasADi

CasADi 3.6 introduces the ability to import standard FMUs, adhering to FMI version 2.0, which can be generated from Modelica or non-Modelica models. Unlike the other efforts described in Section 2, this approach uses the standard C API, as defined by the FMI standard, for evaluating model equations and any derivative information.

3.1 Postponing the creation of CasADi function objects

A fundamental difficulty with all the integration efforts described in Section 2 is that not all Modelica expressions can be efficiently represented as CasADi expressions. In particular, CasADi does not support arithmetics involving string-valued expressions and expressions where the dimensions or sparsity patterns are unknown. There are also fundamental differences in how flow control can be implemented, e.g. if-then-else statements and for/while loops.

In the new FMI support of CasADi, we are able to overcome these limitations by *postponing the creation of CasADi functions objects*. Rather than creating a CasADi function directly from the FMU – which is how CasADi's other foreign function interfaces work – the FMU is imported in the form of a *mutable* representation of the physical model. This mutable representation, in the form of instances of the `DaeBuilder` class in CasADi, allows the user to change properties, set values and perform certain manipulations before an immutable (stateless) CasADi function object is finally created. The corresponding function objects are instances of the newly added `FmuFunction` class, and upon creation saves a snapshot of the current `DaeBuilder` state. Each FMU function object can have multiple vector-valued inputs and multiple vector-valued outputs, where the user defines the composition of each input or output. We typically only include the real-valued, differentable model variables that are manipulated by the various simulation and optimization algorithms available in CasADi. Inputs that are non-differentiable – including string-valued and integer-valued variables – or known to be fixed are assumed to be set prior to the creation of `FmuFunction` instance. Calculated quantitites of interest that are non-differentiable, and hence cannot be used in simulation or optimization algorithms in CasADi, can be obtained via the statistics-functionality of the FMU function objects.

3.2 Derivative calculation

The typical use cases for FMI and/or Modelica models within CasADi involve calculation of derivatives. This includes optimial control formulations solved with gradient-based optimization algorithms as well as dynamic simulation with sensitivity analysis.

To acommodate such use cases, the interface has been designed to:

- Be as efficient as possible, by leveraging parallelization and all available analytic derivative and sparsity information
- Support for both first and second order derivatives (even through the FMI standard only includes first order derivatives)
- Ensure that any calculated derivative quantities can be validated for correctness
- Ensure that the derivative calculation is *predictable* and customizable from a user standpoint

In the following sections, we briefly summarize the kinds of supported derivative information and how they

[1]`https://github.com/pymoca/pymoca`
[2]`https://github.com/jgoppert/cymoca`

are calculated by the interface. We will not discuss user syntax for calculation of derivative information as it is the same as for any other function object in CasADi. For illustration, we will consider a set of FMU model equations that can be represented as a differentiable function $y = f(x)$, where $x \in \mathbb{R}^{n_x}$ is a set of inputs or state vector components and $y \in \mathbb{R}^{n_y}$ is a set of calculated outputs or state derivatives.

Forward derivatives

Forward derivatives, i.e. Jacobian-times-vector products $\frac{\partial f(x)}{\partial x} v$ for some x and v, can be calculated using `fmi2GetDirectionalDerivative` in the FMI C API (Modelica Association 2020). Alternatively, we can estimate the same information using one of three finite difference schemes:

- Forward differences, i.e. $(f(x+hv)-f(x))/h$ for some small h, with approximation error $O(h)$
- Central differences, i.e. $(f(x+hv)-f(x-hv))/(2h)$ for some small h, with approximation error $O(h^2)$
- A generalized, smoothness seeking, scheme using 5-point stencils, $f(x-2hv), f(x-hv), f(x), f(x+hhv), f(x+2hv)$, with approximation error $O(h^4)$

The above schemes represent different tradeoffs between accuracy and computational overhead, requiring 1, 2 and 4 additional function evaluations, respectively. For all of the schemes, we will select a fixed, and predictable, perturbation size h, by default 10^{-6}. This necessitates that the directional derivative seed v is properly scaled.

The intended purpose of the finite difference implementation is not to serve as an alternative to analytic derivatives, but to validate that the provided analytic derivatives are correct. This is achieved by, optionally, allowing a selected finite difference implementation to run in a “shadow mode”, ensuring that the two derivative estimates agree up to some absolute and relative tolerance. This validation also helps ensuring that the finite difference perturbation is correctly chosen, which is important for the calculation of second order derivatives.

Jacobians

We use CasADi’s *greedy*, *distance-2*, *unidirectional* algorithm (Gebremedhin, Manne, and Pothen 2005) to calculate large-and-sparse Jacobians, i.e. $\frac{\partial f(x)}{\partial x}$ for the above example. This approach exploits a priori knowledge of the Jacobian sparsity pattern, which can be derived from the variable dependency information provided in the FMI standard. In most use cases, this technique reduces the problem of calculating the sparse Jacobian to one of calculating a reasonably small set of forward directional derivatives.

We allow this directional derivative calculation to be performed in parallel, using either `std::thread` in the C++ standard or OpenMP. We also scale derivative seeds with nominal values of the FMU and adjust the sign of the perturbation to respect variable bounds, when necessary. By using a fixed step size scaled by the nominal value for the derivative seeds, we ensure that the calculation becomes predictable and customizable as the user can adjust the individual nominal values.

Adjoint derivatives

Support for adjoint derivatives, i.e. Jacobian-transpose-times-vector products $\left[\frac{\partial f(x)}{\partial x}\right]^T w$ for some x and w, was added in FMI 3. As the import code, as of this writing, was limited to FMI 2, we instead use the above Jacobian calculation to calculate adjoint derivatives, i.e. we multiply the transpose of the Jacobian, which may not be formed explicitly, with the vector w from the right.

Note that such an approach may be significantly less efficient than using `fmi3GetAdjointDerivative`, assuming a reverse mode algorithmic differentiation is used for the calculation. As the CasADi FMI import transitions to FMI 3, the intention is for the existing implementation to be used as an optional *validation* of the adjoint directional derivatives, provided by the FMI C API. We predict that such validation will prove important when using the interface for complex physical systems.

Forward-over-adjoint derivatives

The FMI standard, whether FMI 2 or FMI 3, does not include an API for second order derivatives, i.e Hessian-times-vector products $\frac{\partial^2 [w^T f(x)]}{\partial x^2} v$, for some v and w. Nevertheless, we can calculate this information with acceptable efficiency and accuracy using finite difference perturbations of the (analytical) adjoint derivatives. For example, can we approximate the second derivative using central differences:

$$\frac{1}{2h}\left(\left[\frac{\partial f}{\partial x}(x+hv)\right]^T w - \left[\frac{\partial f}{\partial x}(x-hv)\right]^T w\right), \qquad (1)$$

where we calculate the adjoint derivatives as described above.

Hessians

Our main intrest for calculating forward-over-adjoint derivatives is to obtain a large-and-sparse Hessian, i.e. $\frac{\partial^2 [w^T f(x)]}{\partial x^2}$ for the above example. In large-scale gradient-based optimization, knowledge of the exact Hessian can be used to get faster and more robust convergence.

We use CasADi’s *greedy*, *distance-2*, *star-coloring* algorithm (Gebremedhin, Manne, and Pothen 2005) to calculate sparse Hessians. For this calculation, we use the (incomplete) knowledge of the Hessian sparsity pattern that can be extracted from the FMI XML API. In particular, we know that variables that enter linearly will be absent from the Hessian sparsity pattern, which occurs whenever the `dependenciesKind` field is set to something other than *dependent*.

As for the Jacobian calculation, we allow the different directional derivatives to be calculated in parallel, using `std::thread` in the C++ standard or OpenMP.

Since the Hessian calculation relies on approximations, it is especially important to validate the results. We do this validation by comparing each Hessian entry with a reference value:

- For diagonal entries of the Hessian, we compare the result against the corresponding second order finite difference formula, i.e. $(f(x+hv)-2f(x)+f(x-hv))/h^2$.
- For off-diagonal entries of the Hessian, we compare the result against the mirror element, which will be calculated by a finite difference perturbation of a different variable.

In both cases, the validation can be done with little additional overhead and can thus be used as an on-the-fly diagnostics check. The main additional overhead comes from having to disable the star-coloring algorithm to get every Hessian element validated – avoiding to calculate mirror elements in Hessians is a fundamental property of star-coloring, cf. (Gebremedhin, Manne, and Pothen 2005). Whenever the calculated value deviates significantly from the reference value, a warning is issued, helping the user to either resolve non-smoothness issues in the model, detect toolchain bugs or adjust the nominal values.

4 Generalized support for ODE/DAE integration and sensitivity analysis in CasADi

The FMU import described in Section 3 has multiple potential use cases, including simply being used for validated Jacobians and Hessians of the model equations. Another use case is for dynamic simulation with forward and/or adjoint sensitivity analysis. Such an analysis is particularly important if we use *shooting method* to reformulate a dynamic optimization problem into a nonlinear program (NLP). How this type of reformulations can be implemented using CasADi was detailed in (J. A. E. Andersson et al. 2019).

A key ability of CasADi is to embed solvers of initial-value problems (IVPs) in ordinary differential equations (ODE) or differential-algebraic equations (DAE) – which we will refer to as *integrators* - into symbolic expressions and have the framework calculate forward and adjoint sensitivity analysis, including higher order, automatically and efficiently. This support is relatively mature and has been used in numerous applications. However, in order to use this feature with models defined by FMUs, a number of challenges had to be overcome:

- There was previously no support for *controls* in CasADi, i.e. external inputs that change at certain time points. While such problems could still be solved by constructing multiple calls to integrator instances with parametric inputs, this solution is particularly inefficient for FMUs as it would cause the FMUs to be reinitialized at every control point.
- While there was already support for outputting a solution at multiple time points (as opposed to just the end time), this feature was never made to work together with the automatic sensitivity analysis. So as in the case for controls, the solver would need to be called repeatedly, for each segment, again causing excessive reinitilizations.
- The implementation of the automatic forward and adjoint sensitivity analysis only worked well for models given as symbolic expressions. When the model equation was a function object as is the case here, a more limited range of derivative information is efficiently available.
- The ODE/DAE integrators in CasADi did not scale very well to large dimension. In particular, the structure of the forward and adjoint sensitivity equations were insufficiently exploited.

All the above points were addressed in the major refactoring of the ODE/DAE integrator in CasADi 3.6. In particular, the integrators now explicitly exploits forward sensitivity equation structure, adjoint sensitivity equation structure and forward-over-over adjoint sensitivity equation structure. While there may certainly be bottlenecks left in the code, there is – to the best knowledge of the author – no longer any fundamental limitation in CasADi for large-scale ODE/DAE sensitivity analysis, including for FMU models.

5 Exporting FMUs from CasADi

Another addition to CasADi 3.6 is support for exporting FMUs from CasADi. The FMU export is done from instances of `DaeBuilder`, a class which originates from the original (symbolic) coupling between CasADi and JModelica.org as described in Section 2. The FMU export thus reuses the data structures used for the FMU import descibed in Section 3.

As of this writing, a proof-of-concept implementation of FMUs adhering to FMI 3.0 exists in the framework. The implementation is based on the comprehensive support export of self-contained C code from symbolic expressions CasADi that exists in in CasADi. The generated FMUs contain support for both forward and adjoint derivative calculation.

6 Examples and Tutorial

While parts of the FMI support in CasADi are still rudimentary, the framework has been used successfully for real-world applications, including for parameter estimation with Modelica building models (Cañas et al. 2023).

In the CasADi Github repository, a step-by-step Jupyter Notebook tutorial (`fmu_demo.ipynb`) can be found that demonstrates the main capabilities of the FMU import described in Section 3, including:

- Compilation of FMUs from Modelica
- Loading FMUs into CasADi and creating function objects
- Calculation of Jacobians and Hessians
- Integration and forward/adjoint sensitivity analysis
- Dynamic optimization using a direct collocation approach

For up-to-date information about this and other examples, we refer to the CasADi user guide and website.

7 Conclusions and Outlook

The intention of the FMI support in CasADi is to provide numerically efficient and mature interfaces to FMUs, both for import and for export. In particular, such interfaces can enable the implementation of efficient simulation and optimization formulations, for existing physical models available as FMUs. These formulations can include forward, adjoint and forward-over-adjoint sensitivity analysis for the simulation problems and as well as sensitivity calculation for the optimization formulations. To enable such applications, special care has been taken to provide validation and diagnostics of provided derivative information as well as the efficient calculation of second derivatives.

The FMU interfaces are intended to be general-purpose and can be used for both static (steady-state) and dynamic problem formulations. In the dynamic case, both open-loop and closed-loop formulations are of interest.

As of this writing, the interface was still in active development and future additions to the support will be driven mainly by industrial and academic interest.

In the following, we list some of the main future developments the framework.

7.1 Support for FMI 3 import

FMI 3.0 is a natural fit with CasADi as it adds features that are important to many use cases of CasADi. These features especially include the added support for adjoint derivatives and better Jacobian sparsity information, as discussed in Section 1.2. The addded the support for vector-valued variables is also important as all expressions in CasADi are all matrix-valued.

At the time of this writing, only FMU 2 was supported for the FMI import.

7.2 C code generation for imported FMUs

A common use case of CasADi – especially for industrial applications – is to use the code generation support to generate self-contained C code, which can then be run on an embedded system. This code can represent just the evaluation of a function and its derivatives or a higher-level operation, including the solution of an optimal control problem.

A useful extension of the FMI import would be to allow for C code export of imported FMUs. It would for example allow CasADi symbolic expressions to be exported to an embedded system with static or dynamic linking to the FMU shared libary.

It would be possible to use the C code generation of imported FMU together with the FMU export described in Section 5. For example, we could use the CasADi framework to import multiple FMUs, connect them together into an aggregated system model and then export the aggregated model as a new FMU.

7.3 Support for hybrid systems

FMI – and the Modelica modeling language – provides a flexible modeling and execution paradigm for hybrid systems, i.e. systems with event dynamics. To allow such models to be used within CasADi, an extension of the framework would be needed. In particular, we may want extend the simulation and sensitivity analysis support described in Section 4 to also handle hybrid systems. While handling hybrid systems in the context of dynamic optimization – and in CasADi – is often not possible with the same generality as in the context of system simulation, several interesting problems could be addressed this way.

No explicit support for hybrid systems exists in CasADi as of this writing, although many hybrid systems can be reformulated and solved as multi-stage formulations. In addition, integer valued decision variables can be handled by using one of the interfaced solvers for mixed-integer quadratic programs (MIQP) or mixed-integer nonlinear programs (MINLP).

References

Ahlbrink, N. et al. (2009). “vICERP – The virtual Institute of Central Receiver Power Plants: Modeling and Simulation of an Open Volumetric Air Receiver Power Plant”. In: *Proceedings MATHMOD09 Vienna*. Vol. 263.

Andersson, J. et al. (2011). “Integration of CasADi and JModelica.org”. In: *8th International Modelica Conference*.

Andersson, Joel A E et al. (2019). “CasADi – A software framework for nonlinear optimization and optimal control”. In: *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

Beazley, D. M. (2003). “Automated scientific software scripting with SWIG”. In: *Future Gener. Comput. Syst.* 19, pp. 599–609.

Biegler, Lorenz T. (2010). *Nonlinear Programming*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719383. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719383. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898719383.

Cañas, Carlos Durán et al. (2023). “Parameter Estimation of Modelica Building Models Using CasADi”. In: *15th International Modelica Conference*.

Gebremedhin, Assefaw Hadish, Fredrik Manne, and Alex Pothen (2005). “What color is your Jacobian? Graph coloring for computing derivatives”. In: *SIAM Review* 47, pp. 629–705.

Modelica Association (2020-12). *Functional Mock-up Interface for Model Exchange and Co-Simulation Version 2.0.2*. Tech. rep. Linköping: Modelica Association. URL: https://fmi-standard.org.

Modelica Association (2023-12). *Functional Mock-up Interface for Model Exchange and Co-Simulation Version 3.0*. Tech. rep. Linköping: Modelica Association. URL: https://fmi-standard.org.

Rawlings, J.B., D.Q. Mayne, and M. Diehl (2020). *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing. ISBN: 9780975937754. URL: https://books.google.com/books?id=0IJezgEACAAJ.

Shitahun, Alachew et al. (2013). “Model-Based Dynamic Optimization with OpenModelica and CasADi”. In: *IFAC Proceedings Volumes* 46.21. 7th IFAC Symposium on Advances in Automotive Control, pp. 446–451.