CasADi Documentation

Release 3.7.2

Joel Andersson, Joris Gillis

Sep 10, 2025

CONTENTS:

INTRODUCTION

CasADi is an open-source software tool for numerical optimization in general and optimal control (i.e. optimization involving differential equations) in particular. The project was started by Joel Andersson and Joris Gillis while PhD students at the Optimization in Engineering Center (OPTEC) of the KU Leuven under supervision of Moritz Diehl.

This document aims at giving a condensed introduction to CasADi. After reading it, you should be able to formulate and manipulate expressions in CasADi's symbolic framework, generate derivative information efficiently using *algorithmic differentiation*, to set up, solve and perform forward and adjoint sensitivity analysis for systems of ordinary differential equations (ODE) or differential-algebraic equations (DAE) as well as to formulate and solve nonlinear programs (NLP) problems and optimal control problems (OCP).

CasADi is available for C++, Python and MATLAB/Octave with little or no difference in performance. In general, the Python API is the best documented and is slightly more stable than the MATLAB API. The C++ API is stable, but is not ideal for getting started with CasADi since there is limited documentation and since it lacks the interactivity of interpreted languages like MATLAB and Python. The MATLAB module has been tested successfully for Octave (version 4.0.2 or later).

1.1 What CasADi is and what it is *not*

CasADi started out as a tool for algorithmic differentiation (AD) using a syntax borrowed from computer algebra systems (CAS), which explains its name. While AD still forms one of the core functionalities of the tool, the scope of the tool has since been considerably broadened, with the addition of support for ODE/DAE integration and sensitivity analysis, nonlinear programming and interfaces to other numerical tools. In its current form, it is a general-purpose tool for gradient-based numerical optimization – with a strong focus on optimal control – and CasADi is just a name without any particular meaning.

It is important to point out that CasADi is *not* a conventional AD tool, that can be used to calculate derivative information from existing user code with little to no modification. If you have an existing model written in C++, Python or MATLAB/Octave, you need to be prepared to reimplement the model using CasADi syntax.

Secondly, CasADi is *not* a computer algebra system. While the symbolic core does include an increasing set of tools for manipulate symbolic expressions, these capabilities are very limited compared to a proper CAS tool.

Finally, CasADi is not an "optimal control problem solver", that allows the user to enter an OCP and then gives the solution back. Instead, it tries to provide the user with a set of "building blocks" that can be used to implement general-purpose or specific-purpose OCP solvers efficiently with a modest programming effort.

1.2 Help and support

If you find simple bugs or lack some feature that you think would be relatively easy for us to add, the simplest thing is simply to write to the forum, located at http://forum.casadi.org/. We check the forum regularly and try to respond as quickly as possible. The only thing we expect for this kind of support is that you cite us, cf. Section 1.3, whenever you use CasADi in scientific work.

If you want more help, we are always open for academic or industrial cooperation. An academic cooperation usually take the form of a co-authorship of a peer reviewed paper, and an industrial cooperation involves a negotiated consulting contract. Please contact us directly if you are interested in this.

1.3 Citing CasADi

If you use CasADi in published scientific work, please cite the following:

```
@Article{Andersson2018,
  Author = {Joel A E Andersson and Joris Gillis and Greg Horn
            and James B Rawlings and Moritz Diehl},
  Title = {{CasADi} -- {A} software framework for nonlinear optimization
            and optimal control},
  Journal = {Mathematical Programming Computation},
  Year = {2018},
}
```

1.4 Reading this document

The goal of this document is to make the reader familiar with the syntax of CasADi and provide easily available building blocks to build numerical optimization and dynamic optimization software. Our explanation is mostly program code driven and provides little mathematical background knowledge. We assume that the reader already has a fair knowledge of theory of optimization theory, solution of initial-value problems in differential equations and the programming language in question (C++, Python or MATLAB/Octave).

We will use Python and MATLAB/Octave syntax side-by-side in this guide, noting that the Python interface is more stable and better documented. Unless otherwise noted, the MATLAB/Octave syntax also applies to Octave. We try to point out the instances where has a diverging syntax. To facilitate switching between the programming languages, we also list the major differences in Chapter 10.

TWO

OBTAINING AND INSTALLING

CasADi is an open-source tool, available under LGPL license, which is a permissive license that allows the tool to be used royalty-free also in commercial closed-source applications. The main restriction of LGPL is that if you decide to modify CasADi's source code as opposed to just using the tool for your application, these changes (a "derivative-work" of CasADi) must be released under LGPL as well.

The source code is hosted on GitHub and has a core written in self-contained C++ code, relying on nothing but the C++ Standard Library. Its front-ends to Python and MATLAB/Octave are full-featured and auto-generated using the tool SWIG. These front-ends are unlikely to result in noticeable loss of efficiency. CasADi can be used on Linux, OS X and Windows.

For up-to-date installation instructions, visit CasADi's install section: http://install.casadi.org/

SYMBOLIC FRAMEWORK

At the core of CasADi is a self-contained symbolic framework that allows the user to construct symbolic expressions using a MATLAB inspired everything-is-a-matrix syntax, i.e. vectors are treated as n-by-1 matrices and scalars as 1-by-1 matrices. All matrices are *sparse* and use a general sparse format – *compressed column storage* (CCS) – to store matrices. In the following, we introduce the most fundamental classes of this framework.

3.1 The SX symbolics

The SX data type is used to represent matrices whose elements consist of symbolic expressions made up by a sequence of unary and binary operations. To see how it works in practice, start an interactive Python shell (e.g. by typing `ipython` from a Linux terminal or inside a integrated development environment such as Spyder) or launch MATLAB's or Octave's graphical user interface. Assuming CasADi has been installed correctly, you can import the symbols into the workspace as follows:

```python
from casadi import *
```
```
import casadi.*
```

Now create a variable `x` using the syntax:

```python
x = SX.sym("x")
```
```
x = SX.sym('x')
```

```
x
```
```
x =

x
```

This creates a 1-by-1 matrix, i.e. a scalar containing a symbolic primitive called `x`. This is just the display name, not the identifier. Multiple variables can have the same name, but still be different. The identifier is the return value. You can also create vector- or matrix-valued symbolic variables by supplying additional arguments to `SX.sym`:

```
y = SX.sym('y',5)
Z = SX.sym('Z',4,2)
```

```
y = SX.sym('y',5)
Z = SX.sym('Z',4,2)
```

```
[y_0, y_1, y_2, y_3, y_4]
[[Z_0, Z_4],
 [Z_1, Z_5],
 [Z_2, Z_6],
 [Z_3, Z_7]]
```

```
y =

[y_0, y_1, y_2, y_3, y_4]

Z =


[[Z_0, Z_4],
 [Z_1, Z_5],
 [Z_2, Z_6],
 [Z_3, Z_7]]
```

which creates a 5-by-1 matrix, i.e. a vector, and a 4-by-2 matrix with symbolic primitives, respectively.

SX.sym is a (static) function which returns an SX instance. When variables have been declared, expressions can now be formed in an intuitive way:

```
f = x**2 + 10
f = sqrt(f)
print(f)
```

```
f = x^2 + 10;
f = sqrt(f);
display(f)
```

```
sqrt((10+sq(x)))
```

```
f =

sqrt((10+sq(x)))
```

You can also create constant SX instances *without* any symbolic primitives:

- B1 = SX.zeros(4,5): A dense 4-by-5 empty matrix with all zeros

- B2 = SX(4,5): A sparse 4-by-5 empty matrix with all zeros

- B4 = SX.eye(4): A sparse 4-by-4 matrix with ones on the diagonal

```
B1: @1=0,
[[@1, @1, @1, @1, @1],
 [@1, @1, @1, @1, @1],
 [@1, @1, @1, @1, @1],
 [@1, @1, @1, @1, @1]]
B2:
[[00, 00, 00, 00, 00],
 [00, 00, 00, 00, 00],
 [00, 00, 00, 00, 00],
 [00, 00, 00, 00, 00]]
B4: @1=1,
[[@1, 00, 00, 00],
 [00, @1, 00, 00],
 [00, 00, @1, 00],
 [00, 00, 00, @1]]
```

Note the difference between a sparse matrix with *structural* zeros and a dense matrix with *actual* zeros. When printing an expression with structural zeros, these will be represented as `00` to distinguish them from actual zeros `0`.

The following list summarizes the most commonly used ways of constructing new `SX` expressions:

- `SX.sym(name,n,m)`: Create an n-by-m symbolic primitive

- `SX.zeros(n,m)`: Create an n-by-m dense matrix with all zeros

- `SX(n,m)`: Create an n-by-m sparse matrix with all *structural* zeros

- `SX.ones(n,m)`: Create an n-by-m dense matrix with all ones

- `SX.eye(n)`: Create an n-by-n diagonal matrix with ones on the diagonal and structural zeros elsewhere.

- `SX(scalar_type)`: Create a scalar (1-by-1 matrix) with value given by the argument. This method can be used explicitly, e.g. `SX(9)`, or implicitly, e.g. `9 * SX.ones(2,2)`.

- `SX(matrix_type)`: Create a matrix given a numerical matrix given as a NumPy or SciPy matrix (in Python) or as a dense or sparse matrix (in MATLAB/Octave). In MATLAB/Octave e.g. `SX([1,2,3,4])` for a row vector, `SX([1;2;3;4])` for a column vector and `SX([1,2;3,4])` for a 2-by-2 matrix. This method can be used explicitly or implicitly.

- `repmat(v,n,m)`: Repeat expression v n times vertically and m times horizontally. `repmat(SX(3),2,1)` will create a 2-by-1 matrix with all elements 3.

- (*Python only*) `SX(list)`: Create a column vector (n-by-1 matrix) with the elements in the list, e.g. `SX([1,2,3,4])` (note the difference between Python lists and MATLAB/Octave horizontal concatenation, which both uses square bracket syntax)

- (*Python only*) `SX(list of list)`: Create a dense matrix with the elements in the lists, e.g. `SX([[1,2],[3,4]])` or a row vector (1-by-n matrix) using `SX([[1,2,3,4]])`.

3.1.1 Note on namespaces

In MATLAB, if the `import casadi.*` command is omitted, you can still use CasADi by prefixing all the symbols with the package name, e.g. `casadi.SX` instead of `SX`, provided the `casadi` package is in the path. We will not do this in the following for typographical reasons, but note that it is often preferable in user code. In Python, this usage corresponds to issuing `import casadi` instead of `from casadi import *`.

Unfortunately, Octave (version 4.0.3) does not implement MATLAB's `import` command. To work around this issue, we provide a simple function `import.m` that can be placed in Octave's path enabling the compact syntax used in this guide.

3.1.2 Note for C++ users

In C++, all public symbols are defined in the `casadi` namespace and require the inclusion of the `casadi/casadi.hpp` header file. The commands above would be equivalent to:

```cpp
#include <casadi/casadi.hpp>
using namespace casadi;
int main() {
  SX x = SX::sym("x");
  SX y = SX::sym("y",5);
  SX Z = SX::sym("Z",4,2)
  SX f = pow(x,2) + 10;
  f = sqrt(f);
  std::cout << "f: " << f << std::endl;
```

```
    return 0;
}
```

3.2 DM

DM is very similar to SX, but with the difference that the nonzero elements are numerical values and not symbolic expressions. The syntax is also the same, except for functions such as SX.sym, which have no equivalents.

DM is mainly used for storing matrices in CasADi and as inputs and outputs of functions. It is *not* intended to be used for computationally intensive calculations. For this purpose, use the builtin dense or sparse data types in MATLAB, NumPy or SciPy matrices in Python or an expression template based library such as eigen, ublas or MTL in C++. Conversion between the types is usually straightforward:

```
C = DM(2,3)

C_dense = C.full()
from numpy import array
C_dense = array(C) # equivalent

C_sparse = C.sparse()
from scipy.sparse import csc_matrix
C_sparse = csc_matrix(C) # equivalent
```

```
C = DM(2,3);
C_dense = full(C);
C_sparse = sparse(C);
```

More usage examples for SX can be found in the example pack at http://install.casadi.org/. For documentation of particular functions of this class (and others), find the "C++ API" on http://docs.casadi.org/ and search for information about casadi::Matrix.

3.3 The MX symbolics

Let us perform a simple operation using the SX above:

```
x = SX.sym('x',2,2)
y = SX.sym('y')
f = 3*x + y
print(f)
print(f.shape)
```

```
x = SX.sym('x',2,2);
y = SX.sym('y');
f = 3*x + y;
disp(f)
disp(size(f))
```

```
@1=3,
[[((@1*x_0)+y), ((@1*x_2)+y)],
 [((@1*x_1)+y), ((@1*x_3)+y)]]
(2, 2)
```

```
@1=3,
[[((@1*x_0)+y), ((@1*x_2)+y)],
 [((@1*x_1)+y), ((@1*x_3)+y)]]
   2    2
```

As you can see, the output of this operation is a 2-by-2 matrix. Note how the multiplication and the addition were performed element-wise and new expressions (of type SX) were created for each entry of the result matrix.

We shall now introduce a second, more general *matrix expression* type MX. The MX type allows, like SX, to build up expressions consisting of a sequence of elementary operations. But unlike SX, these elementary operations are not restricted to be scalar unary or binary operations ($\mathbb{R} \rightarrow \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$). Instead, the elementary operations that

are used to form `MX` expressions are allowed to be general *multiple sparse-matrix valued* input, *multiple sparse-matrix valued* output functions: $\mathbb{R}^{n_1 \times m_1} \times \ldots \times \mathbb{R}^{n_N \times m_N} \to \mathbb{R}^{p_1 \times q_1} \times \ldots \times \mathbb{R}^{p_M \times q_M}$.

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

```
x = MX.sym('x',2,2)
y = MX.sym('y')
f = 3*x + y
print(f)
print(f.shape)
```

```
x = MX.sym('x',2,2);
y = MX.sym('y');
f = 3*x + y;
disp(f)
disp(size(f))
```

```
((3*x)+y)
(2, 2)
```

```
((3*x)+y)
  2   2
```

Note how the result consists of only two operations (one multiplication and one addition) using `MX` symbolics, whereas the `SX` equivalent has eight (two for each element of the resulting matrix). As a consequence, `MX` can be more economical when working with operations that are naturally vector or matrix valued with many elements. As we shall see in Chapter 4, it is also much more general since we allow calls to arbitrary functions that cannot be expanded in terms of elementary operations.

`MX` supports getting and setting elements, using the same syntax as `SX`, but the way it is implemented is very different. Test, for example, to print the element in the upper-left corner of a 2-by-2 symbolic variable:

```
x = MX.sym('x',2,2)
print(x[0,0])
```

```
x = MX.sym('x',2,2);
x(1,1)
```

```
x[0]
```

```
ans =

x[0]
```

The output should be understood as an expression that is equal to the first (i.e. index 0 in C++) structurally non-zero element of `x`, unlike `x_0` in the `SX` case above, which is the name of a symbolic primitive in the first (index 0) location of the matrix.

Similar results can be expected when trying to set elements:

```
x = MX.sym('x',2)
A = MX(2,2)
A[0,0] = x[0]
A[1,1] = x[0]+x[1]
print('A:', A)
```

```
x = MX.sym('x',2);
A = MX(2,2);
A(1,1) = x(1);
A(2,2) = x(1)+x(2);
display(A)
```

```
A: (project((zeros(2x2,1nz)[0] = x[0]))[1]␣
↪= (x[0]+x[1]))
```

```
A =

(project((zeros(2x2,1nz)[0] = x[0]))[1] =␣
↪(x[0]+x[1]))
```

The interpretation of the (admittedly cryptic) output is that starting with an all zero sparse matrix, an element is assigned to `x_0`. It is then projected to a matrix of different sparsity and an another element is assigned to `x_0+x_1`.

Element access and assignment, of the type you have just seen, are examples of operations that can be used to construct expressions. Other examples of operations are matrix multiplications, transposes, concatenations, resizings, reshapings

and function calls.

3.4 Mixing `SX` and `MX`

You can *not* multiply an `SX` object with an `MX` object, or perform any other operation to mix the two in the same expression graph. You can, however, in an `MX` graph include calls to a `Function` defined by `SX` expressions. This will be demonstrated in Chapter 4. Mixing `SX` and `MX` is often a good idea since functions defined by `SX` expressions have a much lower overhead per operation making it much faster for operations that are naturally written as a sequence of scalar operations. The `SX` expressions are thus intended to be used for low level operations (for example the DAE right hand side in Section 4.4), whereas the `MX` expressions act as a glue and enables the formulation of e.g. the constraint function of an NLP (which might contain calls to ODE/DAE integrators, or might simply be too large to expand as one big expression).

3.5 The `Sparsity` class

As mentioned above, matrices in CasADi are stored using the *compressed column storage* (CCS) format. This is a standard format for sparse matrices that allows linear algebra operations such as element-wise operations, matrix multiplication and transposes to be performed efficiently. In the CCS format, the sparsity pattern is decoded using the dimensions – the number of rows and number of columns – and two vectors. The first vector contains the index of the first structurally nonzero element of each column and the second vector contains the row index for every nonzero element. For more details on the CCS format, see e.g. Templates for the Solution of Linear Systems on Netlib. Note that CasADi uses the CCS format for sparse as well as dense matrices.

Sparsity patterns in CasADi are stored as instances of the `Sparsity` class, which is *reference-counted*, meaning that multiple matrices can share the same sparsity pattern, including `MX` expression graphs and instances of `SX` and `DM`. The `Sparsity` class is also *cached*, meaning that the creation of multiple instances of the same sparsity patterns is always avoided.

The following list summarizes the most commonly used ways of constructing new sparsity patterns:

- `Sparsity.dense(n,m)`: Create a dense n-by-m sparsity pattern
- `Sparsity(n,m)`: Create a sparse n-by-m sparsity pattern
- `Sparsity.diag(n)`: Create a diagonal n-by-n sparsity pattern
- `Sparsity.upper(n)`: Create an upper triangular n-by-n sparsity pattern
- `Sparsity.lower(n)`: Create a lower triangular n-by-n sparsity pattern

The `Sparsity` class can be used to create non-standard matrices, e.g.

```
print(SX.sym('x',Sparsity.lower(3)))
```

```
disp(SX.sym('x',Sparsity.lower(3)))
```

```
[[x_0, 00, 00],
 [x_1, x_3, 00],
 [x_2, x_4, x_5]]
```

```
[[x_0, 00, 00],
 [x_1, x_3, 00],
 [x_2, x_4, x_5]]
```

3.5.1 Getting and setting elements in matrices

To get or set an element or a set of elements in CasADi's matrix types (`SX`, `MX` and `DM`), we use square brackets in Python and round brackets in C++ and MATLAB. As is conventional in these languages, indexing starts from zero in C++ and Python but from one in MATLAB. In Python and C++, we allow negative indices to specify an index counted from the end. In MATLAB, use the `end` keyword for indexing from the end.

Indexing can be done with one index or two indices. With two indices, you reference a particular row (or set or rows) and a particular column (or set of columns). With one index, you reference an element (or set of elements) starting from the upper left corner and column-wise to the lower right corner. All elements are counted regardless of whether they are structurally zero or not.

```
M = SX([[3,7],[4,5]])
print(M[0,:])
M[0,:] = 1
print(M)
```

```
M = SX([3,7;4,5]);
disp(M(1,:))
M(1,:) = 1;
disp(M)
```

```
[[3, 7]]
@1=1,
[[@1, @1],
 [4, 5]]
```

```
[[3, 7]]
@1=1,
[[@1, @1],
 [4, 5]]
```

Unlike Python's NumPy, CasADi slices are not views into the data of the left hand side; rather, a slice access copies the data. As a result, the matrix m is not changed at all in the following example:

```
M = SX([[3,7],[4,5]])
M[0,:][0,0] = 1
print(M)
```

```
[[3, 7],
 [4, 5]]
```

The getting and setting matrix elements is elaborated in the following. The discussion applies to all of CasADi's matrix types.

Single element access is getting or setting by providing a row-column pair or its flattened index (column-wise starting in the upper left corner of the matrix):

```
M = diag(SX([3,4,5,6]))
print(M)
```

```
M = diag(SX([3,4,5,6]));
disp(M)
```

```
[[3, 00, 00, 00],
 [00, 4, 00, 00],
 [00, 00, 5, 00],
 [00, 00, 00, 6]]
```

```
[[3, 00, 00, 00],
 [00, 4, 00, 00],
 [00, 00, 5, 00],
 [00, 00, 00, 6]]
```

```
print(M[0,0])                      disp(M(1,1))
print(M[1,0])                      disp(M(2,1))
print(M[-1,-1])                    disp(M(end,end))
```

```
3                                  3
00                                 00
6                                  6
```

Slice access means setting multiple elements at once. This is significantly more efficient than setting the elements one at a time. You get or set a slice by providing a (*start* , *stop* , *step*) triple. In Python and MATLAB, CasADi uses standard syntax:

```
print(M[:,1])                      disp(M(:,2))
print(M[1:,1:4:2])                 disp(M(2:end,2:2:4))
```

```
[00, 4, 00, 00]                    [00, 4, 00, 00]

[[4, 00],                          [[4, 00],
 [00, 00],                          [00, 00],
 [00, 6]]                           [00, 6]]
```

In C++, CasADi's `Slice` helper class can be used. For the example above, this means `M(Slice(),1)` and `M(Slice(1, -1),Slice(1,4,2))`, respectively.

List access is similar to (but potentially less efficient than) slice access:

```
M = SX([[3,7,8,9],[4,5,6,1]])      M = SX([3 7 8 9; 4 5 6 1]);
print(M)                           disp(M)
print(M[0,[0,3]], M[[5,-6]])       disp(M(1,[1,4]))
                                   disp(M([6,numel(M)-5]))
```

```
[[3, 7, 8, 9],
 [4, 5, 6, 1]]                     [[3, 7, 8, 9],
[[3, 9]] [6, 7]                     [4, 5, 6, 1]]
                                   [[3, 9]]
                                   [[6, 7]]
```

3.6 Arithmetic operations

CasADi supports most standard arithmetic operations such as addition, multiplications, powers, trigonometric functions etc:

```
x = SX.sym('x')                    x = SX.sym('x');
y = SX.sym('y',2,2)                y = SX.sym('y',2,2);
print(sin(y)-x)                    disp(sin(y)-x)
```

```
[[(sin(y_0)-x), (sin(y_2)-x)],     [[(sin(y_0)-x), (sin(y_2)-x)],
 [(sin(y_1)-x), (sin(y_3)-x)]]      [(sin(y_1)-x), (sin(y_3)-x)]]
```

In C++ and Python (but not in MATLAB), the standard multiplication operation (using *) is reserved for element-wise multiplication (in MATLAB `.*`). For **matrix multiplication**, use `A @ B` or (`mtimes(A,B)` in Python 3.4+):

```
print(y*y, y@y)
```

```
disp(y.*y)
disp(y*y)
```

```
[[sq(y_0), sq(y_2)],
 [sq(y_1), sq(y_3)]]
[[(sq(y_0)+(y_2*y_1)), ((y_0*y_2)+(y_2*y_
→3))],
 [((y_1*y_0)+(y_3*y_1)), ((y_1*y_2)+sq(y_
→3))]]
```

```
[[sq(y_0), sq(y_2)],
 [sq(y_1), sq(y_3)]]

[[(sq(y_0)+(y_2*y_1)), ((y_0*y_2)+(y_2*y_
→3))],
 [((y_1*y_0)+(y_3*y_1)), ((y_1*y_2)+sq(y_
→3))]]
```

As is customary in MATLAB, multiplication using `*` and `.*` are equivalent when either of the arguments is a scalar.

Transposes are formed using the syntax `A.T` in Python, `A.T()` in C++ and with `A'` or `A.'` in MATLAB:

```
print(y)
print(y.T)
```

```
disp(y)
disp(y')
```

```
[[y_0, y_2],
 [y_1, y_3]]

[[y_0, y_1],
 [y_2, y_3]]
```

```
[[y_0, y_2],
 [y_1, y_3]]

[[y_0, y_1],
 [y_2, y_3]]
```

Reshaping means changing the number of rows and columns but retaining the number of elements and the relative location of the nonzeros. This is a computationally very cheap operation which is performed using the syntax:

```
x = SX.eye(4)
print(reshape(x,2,8))
```

```
x = SX.eye(4);
reshape(x,2,8)
```

```
@1=1,
[[@1, 00, 00, 00, 00, @1, 00, 00],
 [00, 00, @1, 00, 00, 00, 00, @1]]
```

```
ans =

@1=1,
[[@1, 00, 00, 00, 00, @1, 00, 00],
 [00, 00, @1, 00, 00, 00, 00, @1]]
```

Concatenation means stacking matrices horizontally or vertically. Due to the column-major way of storing elements in CasADi, it is most efficient to stack matrices horizontally. Matrices that are in fact column vectors (i.e. consisting of a single column), can also be stacked efficiently vertically. Vertical and horizontal concatenation is performed using the functions `vertcat` and `horzcat` (that take a variable amount of input arguments) in Python and C++ and with square brackets in MATLAB:

```
x = SX.sym('x',5)
y = SX.sym('y',5)
print(vertcat(x,y))
```

```
x = SX.sym('x',5);
y = SX.sym('y',5);
disp([x;y])
```

```
[x_0, x_1, x_2, x_3, x_4, y_0, y_1, y_2, y_
↪3, y_4]
```

```
[x_0, x_1, x_2, x_3, x_4, y_0, y_1, y_2, y_
↪3, y_4]
```

```
print(horzcat(x,y))
```

```
disp([x,y])
```

```
[[x_0, y_0],
 [x_1, y_1],
 [x_2, y_2],
 [x_3, y_3],
 [x_4, y_4]]
```

```
[[x_0, y_0],
 [x_1, y_1],
 [x_2, y_2],
 [x_3, y_3],
 [x_4, y_4]]
```

There are also variants of these functions that take a list (in Python) or a cell array (in Matlab) as inputs:

```
L = [x,y]
print(hcat(L))
```

```
L = {x,y};
disp([L{:}])
```

```
[[x_0, y_0],
 [x_1, y_1],
 [x_2, y_2],
 [x_3, y_3],
 [x_4, y_4]]
```

```
[[x_0, y_0],
 [x_1, y_1],
 [x_2, y_2],
 [x_3, y_3],
 [x_4, y_4]]
```

Horizontal and vertical split are the inverse operations of the above introduced horizontal and vertical concatenation. To split up an expression horizontally into n smaller expressions, you need to provide, in addition to the expression being split, a vector *offset* of length $n + 1$. The first element of the *offset* vector must be 0 and the last element must be the number of columns. Remaining elements must follow in a non-decreasing order. The output i of the split operation then contains the columns c with $offset[i] \leq c < offset[i + 1]$. The following demonstrates the syntax:

```
x = SX.sym('x',5,2)
w = horzsplit(x,[0,1,2])
print(w[0], w[1])
```

```
x = SX.sym('x',5,2);
w = horzsplit(x,[0,1,2]);
disp(w{1}), disp(w{2})
```

```
[x_0, x_1, x_2, x_3, x_4] [x_5, x_6, x_7, x_
↪8, x_9]
```

```
[x_0, x_1, x_2, x_3, x_4]
[x_5, x_6, x_7, x_8, x_9]
```

The vertsplit operation works analogously, but with the *offset* vector referring to rows:

```
w = vertsplit(x,[0,3,5])
print(w[0], w[1])
```

```
w = vertsplit(x,[0,3,5]);
disp(w{1}), disp(w{2})
```

```
[[x_0, x_5],
 [x_1, x_6],
 [x_2, x_7]]
[[x_3, x_8],
 [x_4, x_9]]
```

```
[[x_0, x_5],
 [x_1, x_6],
 [x_2, x_7]]

[[x_3, x_8],
 [x_4, x_9]]
```

Note that it is always possible to use slice element access instead of horizontal and vertical split, for the above vertical split:

```
w = [x[0:3,:], x[3:5,:]]
print(w[0], w[1])
```

```
w = {x(1:3,:), x(4:5,:)};
disp(w{1}), disp(w{2})
```

```
[[x_0, x_5],
 [x_1, x_6],
 [x_2, x_7]]
[[x_3, x_8],
 [x_4, x_9]]
```

```
[[x_0, x_5],
 [x_1, x_6],
 [x_2, x_7]]

[[x_3, x_8],
 [x_4, x_9]]
```

For SX graphs, this alternative way is completely equivalent, but for MX graphs using horzsplit/vertsplit is *significantly more efficient when all the split expressions are needed*.

Inner product, defined as $< A, B >:= \text{tr}(A\,B) = \sum_{i,j} A_{i,j}\,B_{i,j}$ are created as follows:

```
x = SX.sym('x',2,2)
print(dot(x,x))
```

```
x = SX.sym('x',2,2);
disp(dot(x,x))
```

```
(((sq(x_0)+sq(x_1))+sq(x_2))+sq(x_3))
```

```
(((sq(x_0)+sq(x_1))+sq(x_2))+sq(x_3))
```

Many of the above operations are also defined for the Sparsity class (Section 3.5), e.g. vertcat, horzsplit, transposing, addition (which returns the *union* of two sparsity patterns) and multiplication (which returns the *intersection* of two sparsity patterns).

3.7 Querying properties

You can check if a matrix or sparsity pattern has a certain property by calling an appropriate member function. e.g.

```
y = SX.sym('y',10,1)
print(y.shape)
```

```
y = SX.sym('y',10,1);
size(y)
```

```
(10, 1)
```

```
ans =

    10    1
```

Note that in MATLAB, `obj.myfcn(arg)` and `myfcn(obj, arg)` are both valid ways of calling a member function `myfcn`. The latter variant is probably preferable from a style viewpoint.

Some commonly used properties for a matrix `A` are:

- `A.size1()` The number of rows

- `A.size2()` The number of columns

- `A.shape` (in MATLAB "size") The shape, i.e. the pair (*nrow*,*ncol*)

- `A.numel()` The number of elements, i.e $nrow * ncol$

- `A.nnz()` The number of structurally nonzero elements, equal to `A.numel()` if *dense*.

- `A.sparsity()` Retrieve a reference to the sparsity pattern

- `A.is_dense()` Is a matrix dense, i.e. having no structural zeros

- `A.is_scalar()` Is the matrix a scalar, i.e. having dimensions 1-by-1?

- `A.is_column()` Is the matrix a vector, i.e. having dimensions n-by-1?

- `A.is_square()` Is the matrix square?

- `A.is_triu()` Is the matrix upper triangular?

- `A.is_constant()` Are the matrix entries all constant?

- `A.is_integer()` Are the matrix entries all integer-valued?

The last queries are examples of queries for which *false negative* returns are allowed. A matrix for which `A.is_constant()` is *true* is guaranteed to be constant, but is *not* guaranteed to be non-constant if `A.is_constant()` is *false*. We recommend you to check the API documentation for a particular function before using it for the first time.

3.8 Linear algebra

CasADi supports a limited number of linear algebra operations, e.g. for solution of linear systems of equations:

```
A = MX.sym('A',3,3)
b = MX.sym('b',3)
print(solve(A,b))
```

```
A = MX.sym('A',3,3);
b = MX.sym('b',3);
disp(A\b)
```

```
(A\b)
```

```
(A\b)
```

3.9 Calculus – algorithmic differentiation

The single most central functionality of CasADi is *algorithmic (or automatic) differentiation* (AD). For a function $f : \mathbb{R}^N \to \mathbb{R}^M$:

$$y = f(x),$$

forward mode directional derivatives can be used to calculate Jacobian-times-vector products:

$$\hat{y} = \frac{\partial f}{\partial x} \, \hat{x}.$$

Similarly, *reverse mode* directional derivatives can be used to calculate Jacobian-transposed-times-vector products:

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

Both forward and reverse mode directional derivatives are calculated at a cost proportional to evaluating $f(x)$, *regardless of the dimension* of x.

CasADi is also able to generate complete, *sparse* Jacobians efficiently. The algorithm for this is very complex, but essentially consists of the following steps:

- Automatically detect the sparsity pattern of the Jacobian

- Use graph coloring techniques to find a few forward and/or directional derivatives needed to construct the complete Jacobian

- Calculate the directional derivatives numerically or symbolically

- Assemble the complete Jacobian

Hessians are calculated by first calculating the gradient and then performing the same steps as above to calculate the Jacobian of the gradient in the same way as above, while exploiting symmetry.

3.9.1 Syntax

An expression for a Jacobian is obtained using the syntax:

```
A = SX.sym('A',3,2)
x = SX.sym('x',2)
print(jacobian(A@x,x))
```

```
A = SX.sym('A',3,2);
x = SX.sym('x',2);
jacobian(A*x,x)
```

```
[[A_0, A_3],
 [A_1, A_4],
 [A_2, A_5]]
```

```
ans =

[[A_0, A_3],
 [A_1, A_4],
 [A_2, A_5]]
```

When the differentiated expression is a scalar, you can also calculate the gradient in the matrix sense:

```
print(gradient(dot(A,A),A))
```

```
gradient(dot(A,A),A)
```

```
[[(A_0+A_0), (A_3+A_3)],
 [(A_1+A_1), (A_4+A_4)],
 [(A_2+A_2), (A_5+A_5)]]
```

```
ans =

[[(A_0+A_0), (A_3+A_3)],
 [(A_1+A_1), (A_4+A_4)],
 [(A_2+A_2), (A_5+A_5)]]
```

Note that, unlike `jacobian`, `gradient` always returns a dense vector.

Hessians, and as a by-product gradients, are obtained as follows:

```
[H,g] = hessian(dot(x,x),x)
print('H:', H)
```

```
[H,g] = hessian(dot(x,x),x);
display(H)
```

```
H: @1=2,
[[@1, 00],
 [00, @1]]
```

```
H =

@1=2,
[[@1, 00],
 [00, @1]]
```

For calculating a Jacobian-times-vector product, the `jtimes` function – performing forward mode AD – is often more efficient than creating the full Jacobian and performing a matrix-vector multiplication:

```
A = DM([[1,3],[4,7],[2,8]])
x = SX.sym('x',2)
v = SX.sym('v',2)
f = mtimes(A,x)
print(jtimes(f,x,v))
```

```
A = [1 3;4 7;2 8];
x = SX.sym('x',2);
v = SX.sym('v',2);
f = A*x;
jtimes(f,x,v)
```

```
[(v_0+(3*v_1)), ((4*v_0)+(7*v_1)), ((2*v_
↪0)+(8*v_1))]
```

```
ans =

[(v_0+(3*v_1)), ((4*v_0)+(7*v_1)), ((2*v_
↪0)+(8*v_1))]
```

The `jtimes` function optionally calculates the transposed-Jacobian-times-vector product, i.e. reverse mode AD:

```
w = SX.sym('w',3)
f = mtimes(A,x)
print(jtimes(f,x,w,True))
```

```
w = SX.sym('w',3);
f = A*x;
jtimes(f,x,w,true)
```

```
[(((2*w_2)+(4*w_1))+w_0), (((8*w_2)+(7*w_
↪1))+(3*w_0))]
```

```
ans =

[(((2*w_2)+(4*w_1))+w_0), (((8*w_2)+(7*w_
↪1))+(3*w_0))]
```

FUNCTION OBJECTS

CasADi allows the user to create function objects, in C++ terminology often referred to as *functors*. This includes functions that are defined by a symbolic expression, ODE/DAE integrators, QP solvers, NLP solvers etc.

Function objects are typically created with the syntax:

```
f = functionname(name, arguments, ..., [options])
```

The name is mainly a display name that will show up in e.g. error messages or as comments in generated C code. This is followed by a set of arguments, which is class dependent. Finally, the user can pass an options structure for customizing the behavior of the class. The options structure is a dictionary type in Python, a struct in MATLAB or CasADi's `Dict` type in C++.

A `Function` can be constructed by passing a list of input expressions and a list of output expressions:

```
x = SX.sym('x',2)
y = SX.sym('y')
f = Function('f',[x,y],\
        [x,sin(y)*x])
print(f)
```

```
x = SX.sym('x',2);
y = SX.sym('y');
f = Function('f',{x,y},...
        {x,sin(y)*x});
disp(f)
```

```
f:(i0[2],i1)->(o0[2],o1[2]) SXFunction
```

```
f:(i0[2],i1)->(o0[2],o1[2]) SXFunction
```

which defines a function $f : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2 \times \mathbb{R}^2, \quad (x, y) \mapsto (x, \sin(y)x)$. Note that all function objects in CasADi, including the above, are multiple matrix-valued input, multiple, matrix-valued output.

`MX` expression graphs work the same way:

```
x = MX.sym('x',2)
y = MX.sym('y')
f = Function('f',[x,y],\
        [x,sin(y)*x])
print(f)
```

```
x = MX.sym('x',2);
y = MX.sym('y');
f = Function('f',{x,y},...
        {x,sin(y)*x});
disp(f)
```

```
f:(i0[2],i1)->(o0[2],o1[2]) MXFunction
```

```
f:(i0[2],i1)->(o0[2],o1[2]) MXFunction
```

When creating a `Function` from expressions like that, it is always advisory to *name* the inputs and outputs as follows:

```
f = Function('f',[x,y],\
    [x,sin(y)*x],\
    ['x','y'],['r','q'])
print(f)
```

```
f = Function('f',{x,y},...
    {x,sin(y)*x},...
    {'x','y'},{'r','q'});
disp(f)
```

```
f:(x[2],y)->(r[2],q[2]) MXFunction
```

```
f:(x[2],y)->(r[2],q[2]) MXFunction
```

Naming inputs and outputs is preferred for a number of reasons:

- No need to remember the number or order of arguments

- Inputs or outputs that are absent can be left unset

- More readable and less error prone syntax.

For `Function` instances – to be encountered later – that are *not* created directly from expressions, the inputs and outputs are named automatically.

4.1 Calling function objects

`MX` expressions may contain calls to `Function`-derived functions. Calling a function object is both done for the numerical evaluation and, by passing symbolic arguments, for embedding a *call* to the function object into an expression graph (cf. also Section 4.4).

To call a function object, you either pass the argument in the correct order:

```
r0, q0 = f(1.1,3.3)
print('r0:',r0)
print('q0:',q0)
```

```
[r0, q0] = f(1.1,3.3);
disp(r0)
disp(q0)
```

```
r0: [1.1, 1.1]
q0: [-0.17352, -0.17352]
```

```
[1.1, 1.1]
[-0.17352, -0.17352]
```

or the arguments and their names as follows, which will result in a dictionary (`dict` in Python, `struct` in MATLAB and `std::map<std::string, MatrixType>` in C++):

```
res = f(x=1.1, y=3.3)
print('res:', res)
```

```
res = f('x',1.1,'y',3.3);
disp(res)
```

```
res: {'q': DM([-0.17352, -0.17352]), 'r':
↪DM([1.1, 1.1])}
```

```
    scalar structure containing the fields:

        q =

            casadi.DM object with properties:


        r =

            casadi.DM object with properties:
```

When calling a function object, the dimensions (but not necessarily the sparsity patterns) of the evaluation arguments have to match those of the function inputs, with two exceptions:

- A row vector can be passed instead of a column vector and vice versa.

- A scalar argument can always be passed, regardless of the input dimension. This has the meaning of setting all elements of the input matrix to that value.

When the number of inputs to a function object is large or changing, an alternative syntax to the above is to use the *call* function which takes a Python list / MATLAB cell array or, alternatively, a Python dict / MATLAB struct. The return value will have the same type:

```
arg = [1.1,3.3]
res = f.call(arg)
print('res:', res)
arg = {'x':1.1,'y':3.3}
res = f.call(arg)
print('res:', res)
```

```
arg = {1.1,3.3};
res = f.call(arg);
disp(res)
arg = struct('x',1.1,'y',3.3);
res = f.call(arg);
disp(res)
```

```
res: [DM([1.1, 1.1]), DM([-0.17352, -0.
→17352])]
res: {'q': DM([-0.17352, -0.17352]), 'r':␣
→DM([1.1, 1.1])}
```

```
{
  [1,1] =

    casadi.DM object with properties:


  [1,2] =

    casadi.DM object with properties:


}
  scalar structure containing the fields:

    q =

      casadi.DM object with properties:


    r =

      casadi.DM object with properties:
```

4.2 Converting `MX` to `SX`

A function object defined by an `MX` graph that only contains built-in operations (e.g. element-wise operations such as addition, square root, matrix multiplications and calls to `SX` functions), can be converted into a function defined purely by an `SX` graph using the syntax:

```
sx_function = mx_function.expand()
```

This might speed up the calculations significantly, but might also cause extra memory overhead.

4.3 Nonlinear root-finding problems

Consider the following system of equations:

$$
\begin{aligned}
g_0(z, x_1, x_2, \ldots, x_n) &= 0 \\
g_1(z, x_1, x_2, \ldots, x_n) &= y_1 \\
g_2(z, x_1, x_2, \ldots, x_n) &= y_2 \\
&\vdots \\
g_m(z, x_1, x_2, \ldots, x_n) &= y_m,
\end{aligned}
$$

where the first equation uniquely defines z as a function of x_1, ldots, x_n by the *implicit function theorem* and the remaining equations define the auxiliary outputs y_1, ldots, y_m.

Given a function g for evaluating g_0, ldots, g_m, we can use CasADi to automatically formulate a function G : $\{z_{\text{guess}}, x_1, x_2, \ldots, x_n\} \rightarrow \{z, y_1, y_2, \ldots, y_m\}$. This function includes a guess for z to handle the case when the solution is non-unique. The syntax for this, assuming $n = m = 1$ for simplicity, is:

```
z = SX.sym('x',nz)
x = SX.sym('x',nx)
g0 = sin(x+z)
g1 = cos(x-z)
g = Function('g',[z,x],[g0,g1])
G = rootfinder('G','newton',g)
print(G)
```

```
z = SX.sym('x',nz);
x = SX.sym('x',nx);
g0 = sin(x+z);
g1 = cos(x-z);
g = Function('g',{z,x},{g0,g1});
G = rootfinder('G','newton',g);
disp(G)
```

```
G:(i00,i1)->(i0,o1) Newton
```

```
G:(i00,i1)->(i0,o1) Newton
```

where the `rootfinder` function expects a display name, the name of a solver plugin (here a simple full-step Newton method) and the residual function.

Rootfinding objects in CasADi are differential objects and derivatives can be calculated exactly to arbitrary order.

See also:

API of rootfinder

4.4 Initial-value problems and sensitivity analysis

CasADi can be used to solve initial-value problems in ODE or DAE. The problem formulation used is a DAE of semi-explicit form with quadratures:

$$\dot{x} = f_{\text{ode}}(t, x, z, p, u), \qquad x(0) = x_0$$
$$0 = f_{\text{alg}}(t, x, z, p, u)$$
$$\dot{q} = f_{\text{quad}}(t, x, z, p, u), \qquad q(0) = 0$$

For solvers of *ordinary* differential equations, the second equation and the algebraic variables z must be absent.

An integrator in CasADi is a function that takes the state at the initial time `x0`, a set of parameters `p` and controls `u`, and a guess for the algebraic variables (only for DAEs) `z0` and returns the state vector `xf`, algebraic variables `zf` and the quadrature state `qf` at a number of output times. The control vector `u` is assumed to be piecewise constant and has the same grid discretization as the output grid.

The freely available SUNDIALS suite (distributed along with CasADi) contains the two popular integrators CVodes and IDAS for ODEs and DAEs respectively. These integrators have support for forward and adjoint sensitivity analysis and when used via CasADi's Sundials interface, CasADi will automatically formulate the Jacobian information, which is needed by the backward differentiation formula (BDF) that CVodes and IDAS use. Also automatically formulated will be the forward and adjoint sensitivity equations.

4.4.1 Creating integrators

Integrators are created using CasADi's `integrator` function. Different integrators schemes and interfaces are implemented as *plugins*, essentially shared libraries that are loaded at runtime.

Consider for example the DAE:

$$\dot{x} = z + p,$$
$$0 = z \cos(z) - x$$

An integrator, using the "idas" plugin, can be created using the syntax:

```
x = SX.sym('x'); z = SX.sym('z'); p = SX.sym('p')
dae = {'x':x, 'z':z, 'p':p, 'ode':z+p, 'alg':z*cos(z)-x}
F = integrator('F', 'idas', dae)
print(F)
```

```
F:(x0,z0,p,u[0],adj_xf[],adj_zf[],adj_qf[])->(xf,zf,qf[0],adj_x0[],adj_z0[],adj_p[],adj_
↪u[]) IdasInterface
```

```
x = SX.sym('x'); z = SX.sym('z'); p = SX.sym('p');
dae = struct('x',x,'z',z,'p',p,'ode',z+p,'alg',z*cos(z)-x);
F = integrator('F', 'idas', dae);
disp(F)
```

```
F:(x0,z0,p,u[0],adj_xf[],adj_zf[],adj_qf[])->(xf,zf,qf[0],adj_x0[],adj_z0[],adj_p[],adj_
↪u[]) IdasInterface
```

This will result in an integration from $t_0 = 0$ until $t_f = 1$, i.e. a single output time. We can evaluate the function object using the initial condition $x(0) = 0$, parameter $p = 0.1$ and the guess for the algebraic variable at the initial time $z(0) = 0$ as follows:

```
r = F(x0=0, z0=0, p=0.1)
print(r['xf'])
```

```
r = F('x0',0,'z0',0,'p',0.1);
disp(r.xf)
```

```
0.1724
```

```
0.1724
```

Note that the time horizon is always fixed. It can be changed to its default values $t_0 = 0$ and $t_f = 1$ by adding two additional argument to the constructor, after the DAE. t_f can either be a single value or a vector of values. It may include the initial time.

4.4.2 Sensitivity analysis

From a usage point of view, an integrator behaves just like the function objects created from expressions earlier in the chapter. You can use member functions in the Function class to generate new function objects corresponding to directional derivatives (forward or reverse mode) or complete Jacobians. Then evaluate these function objects numerically to obtain sensitivity information. The documented example "sensitivity_analysis" (available in CasADi's example collection for Python, MATLAB and C++) demonstrate how CasADi can be used to calculate first and second order derivative information (forward-over-forward, forward-over-adjoint, adjoint-over-adjoint) for a simple DAE.

4.5 Nonlinear programming

Note: This section assumes familiarity with much of the material that comes above. There is also a higher-level interface in Chapter 9. That interface can be learned stand-alone.

The NLP solvers distributed with or interfaced to CasADi solves parametric NLPs of the following form:

$$
\begin{aligned}
\underset{x}{\text{minimize:}} \quad & f(x,p) \\
\text{subject to:} \quad & x_{\text{lb}} \leq x \leq x_{\text{ub}} \\
& g_{\text{lb}} \leq g(x,p) \leq g_{\text{ub}}
\end{aligned}
\tag{4.5.1}
$$

where $x \in \mathbb{R}^{nx}$ is the decision variable and $p \in \mathbb{R}^{np}$ is a known parameter vector.

An NLP solver in CasADi is a function that takes the parameter value (p), the bounds (lbx, ubx, lbg, ubg) and a guess for the primal-dual solution (x0, lam_x0, lam_g0) and returns the optimal solution. Unlike integrator objects, NLP solver functions are currently not differentiable functions in CasADi.

There are several NLP solvers interfaced with CasADi. The most popular one is IPOPT, an open-source primal-dual interior point method which is included in CasADi installations. Others, that require the installation of third-party software, include SNOPT, WORHP and KNITRO. Whatever the NLP solver used, the interface will automatically generate the information that it needs to solve the NLP, which may be solver and option dependent. Typically an NLP solver will need a function that gives the Jacobian of the constraint function and a Hessian of the Lagrangian function $(L(x,\lambda) = f(x) + \lambda^T g(x))$ with respect to x.

4.5.1 Creating NLP solvers

NLP solvers are created using CasADi's `nlpsol` function. Different solvers and interfaces are implemented as *plugins*. Consider the following form of the so-called Rosenbrock problem:

$$\begin{array}{ll} \underset{x, y, z}{\text{minimize:}} & x^2 + 100\, z^2 \\ \text{subject to:} & z + (1 - x)^2 - y = 0 \end{array}$$

A solver for this problem, using the "ipopt" plugin, can be created using the syntax:

```
x = SX.sym('x'); y = SX.sym('y'); z = SX.sym('z')
nlp = {'x':vertcat(x,y,z), 'f':x**2+100*z**2, 'g':z+(1-x)**2-y}
S = nlpsol('S', 'ipopt', nlp)
print(S)
```

```
S:(x0[3],p[],lbx[3],ubx[3],lbg,ubg,lam_x0[3],lam_g0)->(x[3],f,g,lam_x[3],lam_g,lam_p[])␣
↪IpoptInterface
```

```
x = SX.sym('x'); y = SX.sym('y'); z = SX.sym('z');
nlp = struct('x',[x;y;z], 'f',x^2+100*z^2, 'g',z+(1-x)^2-y);
S = nlpsol('S', 'ipopt', nlp);
disp(S)
```

```
S:(x0[3],p[],lbx[3],ubx[3],lbg,ubg,lam_x0[3],lam_g0)->(x[3],f,g,lam_x[3],lam_g,lam_p[])␣
↪IpoptInterface
```

Once the solver has been created, we can solve the NLP, using `[2.5,3.0,0.75]` as an initial guess, by evaluating the function S:

```python
r = S(x0=[2.5,3.0,0.75],\
      lbg=0, ubg=0)
x_opt = r['x']
print('x_opt: ', x_opt)
```

```matlab
r = S('x0',[2.5,3.0,0.75],...
      'lbg',0,'ubg',0);
x_opt = r.x;
disp(x_opt)
```

```
This is Ipopt version 3.14.11, running with␣
→linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint␣
→Jacobian...:        3
Number of nonzeros in inequality constraint␣
→Jacobian.:        0
Number of nonzeros in Lagrangian Hessian....
→.........:        2

Total number of variables..................
→.........:        3
                 variables with only␣
→lower bounds:        0
            variables with lower and␣
→upper bounds:        0
                 variables with only␣
→upper bounds:        0
Total number of equality constraints........
→.........:        1
Total number of inequality constraints......
→.........:        0
        inequality constraints with only␣
→lower bounds:        0
   inequality constraints with lower and␣
→upper bounds:        0
        inequality constraints with only␣
→upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
   0  6.2500000e+01 0.00e+00 9.00e+01  -1.0␣
→0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.8457621e+01 1.48e-02 4.10e+01  -1.0␣
→4.10e-01   2.0 1.00e+00 1.00e+00f  1
   2  7.8031530e+00 3.84e-03 8.76e+00  -1.0␣
→2.63e-01   1.5 1.00e+00 1.00e+00f  1
   3  7.1678278e+00 9.42e-05 1.04e+00  -1.0␣
→9.32e-02   1.0 1.00e+00 1.00e+00f  1
   4  6.7419924e+00 6.18e-03 9.95e-01  -1.0␣
→2.69e-01   0.6 1.00e+00 1.00e+00f  1
   5  5.4363330e+00 7.03e-02 1.04e+00  -1.7␣
→8.40e-01   0.1 1.00e+00 1.00e+00f  1
   6  1.2144815e+00 1.52e+00 1.32e+00  -1.7␣
→3.21e+00  -0.4 1.00e+00 1.00e+00f  1
   7  1.0214718e+00 2.51e-01 1.17e+01  -1.7␣
→1.33e+00   0.9 1.00e+00 1.00e+00h  1
   8  3.1864085e-01 1.04e-03 7.53e-01  -1.7␣
→3.58e-01    -  1.00e+00 1.00e+00f  1
   9  3.7092062e-66 3.19e-01 2.57e-32  -1.7␣
→5.64e-01    -  1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
  10  0.0000000e+00 0.00e+00 0.00e+00  -1.7
```

4.6 Quadratic programming

CasADi provides interfaces to solve quadratic programs (QPs). Supported solvers are the open-source solvers qpOASES (distributed with CasADi), OOQP, OSQP and PROXQP as well as the commercial solvers CPLEX and GUROBI.

There are two different ways to solve QPs in CasADi, using a high-level interface and a low-level interface. They are described in the following.

4.6.1 High-level interface

The high-level interface for quadratic programming mirrors that of nonlinear programming, i.e. expects a problem of the form (4.5.1), with the restriction that objective function $f(x, p)$ must be a convex quadratic function in x and the constraint function $g(x, p)$ must be linear in x. If the functions are not quadratic and linear, respectively, the solution is done at the current linearization point, given by the "initial guess" for x.

If the objective function is not convex, the solver may or may not fail to find a solution or the solution may not be unique.

To illustrate the syntax, we consider the following convex QP:

$$
\begin{aligned}
\text{minimize:} \quad & x^2 + y^2 \\
x, y \quad & \\
\text{subject to:} \quad & x + y - 10 \geq 0
\end{aligned}
\tag{4.6.1}
$$

To solve this problem with the high-level interface, we simply replace `nlpsol` with `qpsol` and use a QP solver plugin such as the with CasADi distributed qpOASES:

```
x = SX.sym('x'); y = SX.sym('y')
qp = {'x':vertcat(x,y), 'f':x**2+y**2, 'g':x+y-10}
S = qpsol('S', 'qpoases', qp)
print(S)
```

```
S:(x0[2],p[],lbx[2],ubx[2],lbg,ubg,lam_x0[2],lam_g0)->(x[2],f,g,lam_x[2],lam_g,lam_p[])␣
↪MXFunction
```

```
x = SX.sym('x'); y = SX.sym('y');
qp = struct('x',[x;y], 'f',x^2+y^2, 'g',x+y-10);
S = qpsol('S', 'qpoases', qp);
disp(S)
```

```
S:(x0[2],p[],lbx[2],ubx[2],lbg,ubg,lam_x0[2],lam_g0)->(x[2],f,g,lam_x[2],lam_g,lam_p[])␣
↪MXFunction
```

The created solver object S will have the same input and output signature as the solver objects created with `nlpsol`. Since the solution is unique, it is less important to provide an initial guess:

```
r = S(lbg=0)
x_opt = r['x']
print('x_opt: ', x_opt)
```

```
r = S('lbg',0);
x_opt = r.x;
disp(x_opt)
```

```
################### qpOASES -- QP NO.
↪ 1   ###################

    Iter  |   StepLength   |      Info
↪      |   nFX  |   nAC
---------+----------------+-------------
↪-----+---------+---------
      0 |   1.866661e-07 |   ADD CON
↪ 0 |     1  |     1
      1 |   8.333622e-10 |   REM BND
↪ 1 |     0  |     1
      2 |   1.000000e+00 |   QP
↪SOLVED     |     0  |     1
x_opt:  [5, 5]
```

```
################### qpOASES -- QP NO.
↪ 1   ###################

    Iter  |   StepLength   |      Info
↪      |   nFX  |   nAC
---------+----------------+-------------
↪-----+---------+---------
      0 |   1.866661e-07 |   ADD CON
↪ 0 |     1  |     1
      1 |   8.333622e-10 |   REM BND
↪ 1 |     0  |     1
      2 |   1.000000e+00 |   QP
↪SOLVED     |     0  |     1
[5, 5]
```

4.6.2 Low-level interface

The low-level interface, on the other hand, solves QPs of the following form:

$$
\begin{aligned}
\text{minimize:} &\quad \tfrac{1}{2} x^T H\, x + g^T x \\
x & \\
\text{subject to:} &\quad
\begin{aligned}
x_{\text{lb}} \leq &\quad x &\leq x_{\text{ub}} \\
a_{\text{lb}} \leq &\quad A\,x &\leq a_{\text{ub}}
\end{aligned}
\end{aligned}
\tag{4.6.2}
$$

Encoding problem (4.6.1) in this form, omitting bounds that are infinite, is straightforward:

```
H = 2*DM.eye(2)
A = DM.ones(1,2)
g = DM.zeros(2)
lba = 10.
```

```
H = 2*DM.eye(2);
A = DM.ones(1,2);
g = DM.zeros(2);
lba = 10;
```

To create a solver instance, instead of passing symbolic expressions for the QP, we now pass the sparsity patterns of the matrices H and A. Since we used CasADi's DM-type above, we can simply query the sparsity patterns:

```
qp = {}
qp['h'] = H.sparsity()
qp['a'] = A.sparsity()
S = conic('S','qpoases',qp)
print(S)
```

```
S:(h[2x2,2nz],g[2],a[1x2],lba,uba,lbx[2],ubx[2],x0[2],lam_x0[2],lam_a0,q[],p[])->(x[2],
↪cost,lam_a,lam_x[2]) QpoasesInterface
```

```
qp = struct;
qp.h = H.sparsity();
qp.a = A.sparsity();
S = conic('S','qpoases',qp);
disp(S)
```

```
S:(h[2x2,2nz],g[2],a[1x2],lba,uba,lbx[2],ubx[2],x0[2],lam_x0[2],lam_a0,q[],p[])->(x[2],
↪cost,lam_a,lam_x[2]) QpoasesInterface
```

The returned `Function` instance will have a *different* input/output signature compared to the high-level interface, one that includes the matrices H and A:

```
r = S(h=H, g=g, \
      a=A, lba=lba)
x_opt = r['x']
print('x_opt: ', x_opt)
```

```
r = S('h', H, 'g', g,...
      'a', A, 'lba', lba);
x_opt = r.x;
disp(x_opt)
```

```
#################### qpOASES -- QP NO.
↪ 1    ####################

   Iter   |    StepLength    |      Info
↪      |   nFX   |   nAC
  ----------+-----------------+-------------
↪-----+---------+---------
        0  |   1.866661e-07   |   ADD CON
↪ 0   |     1    |     1
        1  |   8.333622e-10   |   REM BND
↪ 1   |     0    |     1
        2  |   1.000000e+00   |    QP
↪SOLVED     |     0    |     1
x_opt:  [5, 5]
```

```
#################### qpOASES -- QP NO.
↪ 1    ####################

   Iter   |    StepLength    |      Info
↪      |   nFX   |   nAC
  ----------+-----------------+-------------
↪-----+---------+---------
        0  |   1.866661e-07   |   ADD CON
↪ 0   |     1    |     1
        1  |   8.333622e-10   |   REM BND
↪ 1   |     0    |     1
        2  |   1.000000e+00   |    QP
↪SOLVED     |     0    |     1
[5, 5]
```

4.7 For-loop equivalents

When modeling using expression graphs in CasADi, it is a common pattern to use of for-loop constructs of the host language (C++/Python/Matlab).

The graph size will grow linearly with the loop size n, and so will the construction time of the expression graph and the initialization time of functions using that expression.

We offer some special constructs that improve on this complexity.

4.7.1 Map

Suppose you are interested in computing a function $f : \mathbb{R}^n \to \mathbb{R}^m$ repeatedly on all columns of a matrix $X \in \mathbb{R}^{n \times N}$, and aggregating all results in a result matrix $Y \in \mathbb{R}^{m \times N}$:

```
N = 4
X = MX.sym("X",1,N)

print(f)

ys = []
for i in range(N):
  ys.append(f(X[:,i]))

Y = hcat(ys)
F = Function('F',[X],[Y])
print(F)
```

```
N = 4;
X = MX.sym('X',1,N);

disp(f)

ys = {};
for i=1:N
  ys{end+1} = f(X(:,i));
end
Y = [ys{:}];
F = Function('F',{X},{Y});
disp(F)
```

```
f:(i0)->(o0) SXFunction
F:(i0[1x4])->(o0[1x4]) MXFunction
```

```
f:(i0)->(o0) SXFunction
F:(i0[1x4])->(o0[1x4]) MXFunction
```

The aggregate function $F : \mathbb{R}^{n \times N} \to \mathbb{R}^{m \times N}$ can be obtained with the `map` construct:

```
F = f.map(N)
print(F)
```

```
F = f.map(N);
disp(F)
```

```
map4_f:(i0[1x4])->(o0[1x4]) Map
```

```
map4_f:(i0[1x4])->(o0[1x4]) Map
```

CasADi can be instructed to parallelize when F gets evaluated. In the following example, we dedicate 2 threads for the map task.

```
F = f.map(N,"thread",2)
print(F)
```

```
F = f.map(N,'thread',2);
disp(F)
```

```
threadmap2_map2_f:(i0[1x4])->(o0[1x4])␣
→ThreadMap
```

```
threadmap2_map2_f:(i0[1x4])->(o0[1x4])␣
→ThreadMap
```

The `map` operation supports primitive functions f with multiple inputs/outputs which can also be matrices. Aggregation will always happen horizontally.

The `map` operation exhibits constant graph size and initialization time.

4.7.2 Fold

In case each for-loop iteration depends on the result from the previous iteration, the `fold` construct applies. In the following, the `x` variable acts as an accumulater that is initialized by $x_0 \in \mathbb{R}^n$:

```
x = x0
for i in range(N):
    x = f(x)

F = Function('F',[x0],[x])
print(F)
```

```
x = x0;
for i=1:N
    x = f(x);
end

F = Function('F',{x0},{x});
disp(F)
```

```
F:(i0)->(o0) MXFunction
```

```
F:(i0)->(o0) MXFunction
```

For a given function $f : \mathbb{R}^n \to \mathbb{R}^n$, the result function $F : \mathbb{R}^n \to \mathbb{R}^n$ can be obtained with the `fold` construct:

```
F = f.fold(N)
print(F)
```

```
F = f.fold(N);
disp(F)
```

```
fold_f:(i0)->(o0) MXFunction
```

```
fold_f:(i0)->(o0) MXFunction
```

In case intermediate accumulator values are desired as output ($\mathbb{R}^n \to \mathbb{R}^{n \times N}$), use `mapaccum` instead of `fold`.

The `fold`/`mapaccum` operation supports primitive functions f with multiple inputs/outputs which can also be matrices. The first input and output are used for accumulating, while the remainder inputs are read column-wise over the iterations.

The `map`/`mapaccum` operation exhibits a graph size and initialization time that scales logarithmically with n.

4.7.3 Conditional evaluation

It is possible to include conditional evaluation of expressions in CasADi expression graphs by constucting `conditional` function instances. This function takes a number of existing `Function` instances, f_1, f_2, f_n as well as a "default" function f_{def}. All these functions must have the same input and output signatures, i.e. the same number of inputs and outputs with the same dimensions:

```
x = SX.sym("x")
f0 = Function("f0",[x],[sin(x)])
f1 = Function("f1",[x],[cos(x)])
f2 = Function("f2",[x],[tan(x)])
f_cond = Function.conditional('f_cond', [f0, f1], f2)
print(f_cond)
```

```
f_cond:(i0,i1)->(o0) Switch
```

```
x = SX.sym('x');
f0 = Function('f0',{x},{sin(x)});
f1 = Function('f1',{x},{cos(x)});
f2 = Function('f2',{x},{tan(x)});
f_cond = Function.conditional('f_cond', {f0, f1}, f2);
disp(f_cond);
```

```
f_cond:(i0,i1)->(o0) Switch
```

The result is a new function instance with the same input/output signature, but with one additional input corresponding to an index. Its evaluation corresponds to:

$$f_{\text{cond}}(c, x_1, x_2, \ldots, x_m) = \begin{cases} f_0(x_1, x_2, \ldots, x_m) & \text{if } c = 0, \\ f_1(x_1, x_2, \ldots, x_m) & \text{if } c = 1, \\ f_2(x_1, x_2, \ldots, x_m) & \text{otherwise} \end{cases}$$

A function above can be missing (i.e. be a null pointer `Function()`), in which case all outputs will be evaluated to NaN. Note that the evaluation is "short-circuiting", i.e. only the relevant function is evaluated. This also applies to any derivative calculation.

A common special case is when there is only a single case in addition to the default case. This is equivalent to an if-then-else statement, which can be written with the shorthand:

```
x = SX.sym("x")
f_true = Function("f_true",[x],[cos(x)])
f_false = Function("f_false",[x],[sin(x)])
f_cond = Function.if_else('f_cond', f_true, f_false)
print(f_cond)
```

```
f_cond:(i0,i1)->(o0) Switch
```

```
x = SX.sym('x');
f_true = Function('f_true',{x},{cos(x)});
f_false = Function('f_false',{x},{sin(x)});
f_cond = Function.if_else('f_cond', f_true, f_false);
disp(f_cond);
```

```
f_cond:(i0,i1)->(o0) Switch
```

Note that conditional expressions such as these can result in non-smooth expressions that may not converge if used if used in a gradient-based optimization algorithm.

GENERATING C-CODE

The numerical evaluation of function objects in CasADi normally takes place in *virtual machines*, implemented as part of CasADi's symbolic framework. But CasADi also supports the generation of self-contained C-code for a large subset of function objects.

C-code generation is interesting for a number of reasons:

- Speeding up the evaluation time. As a rule of thumb, the numerical evaluation of autogenerated code, compiled with code optimization flags, can be between 4 and 10 times faster than the same code executed in CasADi's virtual machines.

- Allowing code to be compiled on a system where CasADi is not installed, such as an embedded system. All that is needed to compile the generated code is a C compiler.

- Debugging and profiling functions. The generated code is essentially a mirror of the evaluation that takes place in the virtual machines and if a particular operation is slow, this is likely to show up when analyzing the generated code with a profiling tool such as `gprof`. By looking at the code, it is also possible to detect what is potentially done in a suboptimal way. If the code is very long and takes a long time to compile, it is an indication that some functions need to be broken up in smaller, but nested functions.

5.1 Syntax for generating code

Generated C code can be as simple as calling the `generate` member function for a `Function` instance.

```python
x = MX.sym('x',2)
y = MX.sym('y')
f = Function('f',[x,y],\
      [x,sin(y)*x],\
      ['x','y'],['r','q'])
f.generate('gen.c')
print(open('gen.c','r').read())
```

```matlab
x = MX.sym('x',2);
y = MX.sym('y');
f = Function('f',{x,y},...
      {x,sin(y)*x},...
      {'x','y'},{'r','q'});
f.generate('gen.c');
```

```c
/* This file was automatically generated by␣
→CasADi 3.7.2.
 *  It consists of:
 *    1) content generated by CasADi␣
→runtime: not copyrighted
 *    2) template code copied from CasADi␣
→source: permissively licensed (MIT-0)
 *    3) user code: owned by the user
 *
 */
#ifdef __cplusplus
extern "C" {
#endif

/* How to prefix internal symbols */
#ifdef CASADI_CODEGEN_PREFIX
  #define CASADI_NAMESPACE_CONCAT(NS, ID) _
→CASADI_NAMESPACE_CONCAT(NS, ID)
  #define _CASADI_NAMESPACE_CONCAT(NS, ID)␣
→NS ## ID
  #define CASADI_PREFIX(ID) CASADI_
→NAMESPACE_CONCAT(CODEGEN_PREFIX, ID)
#else
  #define CASADI_PREFIX(ID) gen_ ## ID
#endif

#include <math.h>

#ifndef casadi_real
#define casadi_real double
#endif

#ifndef casadi_int
#define casadi_int long long int
#endif

/* Add prefix to internal symbols */
#define casadi_copy CASADI_PREFIX(copy)
#define casadi_f0 CASADI_PREFIX(f0)
#define casadi_s0 CASADI_PREFIX(s0)
#define casadi_s1 CASADI_PREFIX(s1)

/* Symbol visibility in DLLs */
#ifndef CASADI_SYMBOL_EXPORT
  #if defined(_WIN32) || defined(__WIN32__)␣
→|| defined(__CYGWIN__)
    #if defined(STATIC_LINKED)
      #define CASADI_SYMBOL_EXPORT
    #else
      #define CASADI_SYMBOL_EXPORT __
→declspec(dllexport)
```

This will create a C file `gen.c` containing the function `f` and all its dependencies and required helper functions. We will return to how this file can be used in Section 5.1 and the structure of the generated code is described in Section 5.3 below.

You can generate a C file containing multiple CasADi functions by working with CasADi's `CodeGenerator` class:

```
f = Function('f',[x],[sin(x)])       f = Function('f',{x},{sin(x)});
g = Function('g',[x],[cos(x)])       g = Function('g',{x},{cos(x)});
C = CodeGenerator('gen.c')           C = CodeGenerator('gen.c');
C.add(f)                             C.add(f);
C.add(g)                             C.add(g);
C.generate()                         C.generate();
```

Both the `generate` function and the `CodeGenerator` constructor take an optional options dictionary as an argument, allowing customization of the code generation. Two useful options are `main`, which generates a *main* entry point, and `mex`, which generates a *mexFunction* entry point:

```
f = Function('f',[x],[sin(x)])       f = Function('f',{x},{sin(x)});
opts = dict(main=True, \             opts = struct('main', true,...
        mex=True)                             'mex', true);
f.generate('gen.c',opts)             f.generate('gen.c',opts);
```

This enables executing the function from the command line and MATLAB, respectively, as described in Section 5.2 below.

If you plan to link directly against the generated code in some C/C++ application, a useful option is `with_header`, which controls the creation of a header file containing declarations of the functions with external linkage, i.e. the API of the generated code, described in Section 5.3 below.

Here is a list of available options for the `CodeGenerator` class:

Option	Default value	Description
`verbose`	true	Include comments in generated code
`mex`	false	Generate an MATLAB/Octave MEX entry point
`cpp`	false	Generated code is C++ instead of C
`main`	false	Generate a command line interface
`casadi_real`	double	Floating point type
`casadi_int`	long long int	Integer type
`with_header`	false	Generate a header file
`with_mem`	false	Generate a simplified C API
`indent`	2	Number of spaces per indentation level

5.2 Using the generated code

The generated C code can be used in a number of different ways:

- The code can be compiled into a dynamically linked library (DLL), from which a `Function` instance can be created using CasADi's `external` function. Optionally, the user can rely on CasADi to carry out the compilation *just-in-time*.

- The generated code can be compiled into MEX function and executed from MATLAB.

- The generated code can be executed from the command line.

- The user can link, statically or dynamically, the generated code to his or her C/C++ application, accessing the C API of the generated code.

- The code can be compiled into a dynamically linked library and the user can then manually access the C API using `dlopen` on Linux/OS X or `LoadLibrary` on Windows.

This is elaborated in the following.

5.2.1 CasADi's `external` function

The `external` command allows the user to create a `Function` instance from a dynamically linked library with the entry points described by the C API described in Section 5.3. Since the autogenerated files are self-contained[1], the compilation – on Linux/OSX – can be as easy as issuing:

```
gcc -fPIC -shared gen.c -o gen.so
```

from the command line. Or, equivalently using MATLAB's `system` command or Python's `os.system` command. Assuming `gen.c` was created as described in the previous section, we can then create a `Function` f as follows:

```
f = external('f', './gen.so')
print(f(3.14))
```

```
f = external('f', './gen.so');
disp(f(3.14))
```

```
0.00159265
```

```
0.00159265
```

We can also rely on CasADi performing the compilation *just-in-time* using CasADi's `Importer` class. This is a plugin class, which at the time of writing had two supported plugins, namely `'clang'`, which invokes the *LLVM/Clang* compiler framework (distributed with CasADi), and `'shell'`, which invokes the system compiler via the command line:

```
C = Importer('gen.c','shell')
f = external('f',C)
print(f(3.14))
```

```
C = Importer('gen.c','shell');
f = external('f',C);
disp(f(3.14))
```

```
0.00159265
```

```
0.00159265
CasADi - 2025-09-10 17:20:50 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shellQUBlnf.so") [.../casadi/solvers/
↪shell_compiler.cpp:66]
CasADi - 2025-09-10 17:20:50 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shellQUBlnf.o") [.../casadi/solvers/shell_
↪compiler.cpp:67]
```

We will return to the `external` function in Section 6.3.

---

[1] An exception is when code is generated for a function that in turn contains calls to external functions.

5.2.2 Calling generated code from MATLAB

An alternative way of executing generated code is to compile the code into a MATLAB MEX function and call from MATLAB. This assumes that the `mex` option was set to "true" during the code generation, cf. Section 5.1. The generated MEX function takes the function name as its first argument, followed by the function inputs:

```
%mex gen.c -largeArrayDims  % Matlab
mex gen.c -DMATLAB_MEX_FILE % Octave

disp(gen('f', 3.14))
```

```
1.5927e-03
```

Note that the result of the execution is always a MATLAB sparse matrix by default. Compiler flags `-DCASASI_MEX_ALWAYS_DENSE` and `-DCASASI_MEX_ALLOW_DENSE` may be set to influence this behaviour.

5.2.3 Calling generated code from the command line

Another option is to execute the generated code from the Linux/OSX command line. This is possible if the `main` option was set to "true" during the code generation, cf. Section 5.1. This is useful if you e.g. want to profile the generated with a tool such as `gprof`.

When executing the generated code, the function name is passed as a command line argument. The nonzero entries of all the inputs need to be passed via standard input and the function will return the output nonzeros for all the outputs via standard output:

```
# Command line
echo 3.14 3.14 > gen_in.txt
gcc gen.c -o gen
./gen f < gen_in.txt > gen_out.txt
cat gen_out.txt # returns 0.00159265 0.00159265
```

5.2.4 Linking against generated code from a C/C++ application

The generated code is written so that it can be linked with directly from a C/C++ application. If the `with_header` option was set to "true" during the code generation, a header file with declarations of all the exposed entry points of the file. Using this header file requires an understanding of CasADi's codegen API, as described in Section 5.3 below. Symbols that are *not* exposed are prefixed with a file-specific prefix, allowing an application to link against multiple generated functions without risking symbol conflicts.

5.2.5 Dynamically loading generated code from a C/C++ application

A variant of above is to compile the generated code into a shared library, but directly accessing the exposed symbols rather than relying on CasADi's `external` function. This also requires an understanding of the structure of the generated code.

In CasADi's example collection, `codegen_usage.cpp` demonstrates how this can be done.

5.3 API of the generated code

The API of the generated code consists of a number of functions with external linkage. In addition to the actual execution, there are functions for memory management as well as meta information about the inputs and outputs. These functions are described in the following. Below, assume that the name of function we want to access is `fname`. To see what these functions actually look like in code and when they are called, we refer to the `codegen_usage.cpp` example.

5.3.1 Reference counting

```
void fname_incref(void);
void fname_decref(void);
```

A generated function may need to e.g. read in some data or initialize some data structures before first call. This is typically not needed for functions generated from CasADi expressions, but may be required e.g. when the generated code contains calls to external functions. Similarly, memory might need to be deallocated after usage.

To keep track of the ownership, the generated code contains two functions for increasing and decreasing a reference counter. They are named `fname_incref` and `fname_decref`, respectively. These functions have no input argument and return void.

Typically, some initialization may take place upon the first call to `fname_incref` and subsequent calls will only increase some internal counter. The `fname_decref`, on the other hand, decreases the internal counter and when the counter hits zero, a deallocation – if any – takes place.

5.3.2 Number of inputs and outputs

```
casadi_int fname_n_in(void);
casadi_int fname_n_out(void);
```

The number of function inputs and outputs can be obtained by calling the `fname_n_in` and `fname_n_out` functions, respectively. These functions take no inputs and return the number of input or outputs (`casadi_int` is an alias for `long long int`).

5.3.3 Names of inputs and outputs

```
const char* fname_name_in(casadi_int ind);
const char* fname_name_out(casadi_int ind);
```

The functions `fname_name_in` and `fname_name_out` return the name of a particular input or output. They take the index of the input or output, starting with index 0, and return a `const char*` with the name as a null-terminated C string. Upon failure, these functions will return a null pointer.

5.3.4 Sparsity patterns of inputs and outputs

```
const casadi_int* fname_sparsity_in(casadi_int ind);
const casadi_int* fname_sparsity_out(casadi_int ind);
```

The sparsity pattern for a given input or output is obtained by calling `fname_sparsity_in` and `fname_sparsity_out`, respectively. These functions take the input or output index and return a pointer to a field of constant integers (`const casadi_int*`). This is a compact representation of the *compressed column storage* (CCS) format that CasADi uses, cf. Section 3.5. The integer field pointed to is structured as follows:

- The first two entries are the number of rows and columns, respectively. In the following referred to as `nrow` and `ncol`.

- If the third entry is 1, the pattern is dense and any remaining entries are discarded.

- If the third entry is `0`, that entry plus subsequent `ncol` entries form the nonzero offsets for each column, `colind` in the following. E.g. column i will consist of the nonzero indices ranging from `colind[i]` to `colind[i+1]`. The last entry, `colind[ncol]`, will be equal to the number of nonzeros, `nnz`.

- Finally, *if* the sparsity pattern is *not dense*, i.e. if `nnz` $\neq$ `nrow * ncol`, then the last `nnz` entries will contain the row indices.

Upon failure, these functions will return a null pointer.

5.3.5 Memory objects

A function may contain some mutable memory, e.g. for caching the latest factorization or keeping track of evaluation statistics. When multiple functions need to call the same function without conflicting, they each need to work with a different memory object. This is especially important for evaluation in parallel on a shared memory architecture, in which case each thread should access a different memory object.

```
void* fname_alloc_mem(void);
```

Allocates a memory object which will be passed to the numerical evaluation.

```
int fname_init_mem(void* mem);
```

(Re)initializes a memory object. Returns 0 upon successful return;

```
int fname_free_mem(void* mem);
```

Frees a memory object. Returns 0 upon successful return;

5.3.6 Work vectors

```
int fname_work(casadi_int* sz_arg, casadi_int* sz_res, casadi_int* sz_iw, casadi_int* sz_
↪w);
```

To allow the evaluation to be performed efficiently with a small memory footprint, the user is expected to pass four work arrays. The function `fname_work` returns the length of these arrays, which have entries of type `const double*`, `double*`, `casadi_int` and `double`, respectively.

The return value of the function is nonzero upon failure.

5.3.7 Numerical evaluation

```
int fname(const double** arg, double** res,
          casadi_int* iw, double* w, void* mem);
```

Finally, the function fname, performs the actual evaluation. It takes as input arguments the four work vectors and a memory object created using fname_alloc_mem (or NULL if absent). The length of the work vectors must be at least the lengths provided by the fname_work command and the index of the memory object must be strictly smaller than the value returned by fname_n_mem.

The nonzeros of the function inputs are pointed to by the first entries of the arg work vector and are unchanged by the evaluation. Similarly, the output nonzeros are pointed to by the first entries of the res work vector and are also unchanged (i.e. the pointers are unchanged, not the actual values).

The return value of the function is nonzero upon failure.

USER-DEFINED FUNCTION OBJECTS

There are situations when rewriting user-functions using CasADi symbolics is not possible or practical. To tackle this, CasADi provides a number of ways to embed a call to a "black box" function defined in the language CasADi is being used from (C++, MATLAB or Python) or in C. That being said, the recommendation is always to try to avoid this when possible, even if it means investing a lot of time reimplementing existing code. Functions defined using CasADi symbolics are almost always more efficient, especially when derivative calculation is involved, since a lot more structure can typically be exploited.

Depending on the circumstances, the user can implement custom `Function` objects in a number of different ways, which will be elaborated on in the following sections:

- Subclassing `FunctionInternal`: Section 6.1

- Subclassing `Callback`: Section 6.2

- Importing a function with `external`: Section 6.3

- Just-in-time compile a C language string: Section 6.4

- Replace the function call with a lookup table: Section 6.5

6.1 Subclassing `FunctionInternal`

All function objects presented in Chapter 4 are implemented in CasADi as C++ classes inheriting from the `FunctionInternal` abstract base class. In principle, a user with familiarity with C++ programming, can implement a class inheriting from `FunctionInternal`, overloading the virtual methods of this class. The best reference for doing so is the C++ API documentation, choosing "switch to internal" to expose the internal API.

Since `FunctionInternal` is not considered part of the stable, public API, we advice against this in general, unless the plan is to contribution to CasADi's source.

6.2 Subclassing `Callback`

The `Callback` class provides a public API to `FunctionInternal` and inheriting from this class has the same effect as inheriting directly from `FunctionInternal`. Thanks to *cross-language polymorphism*, it is possible to implement the exposed methods of `Callback` from either Python, MATLAB/Octave or C++.

The derived class consists of the following parts:

- A constructor or a static function replacing the constructor

- A number of *virtual* functions, all optional, that can be overloaded in order to get the desired behavior. This includes the number of inputs and outputs using `get_n_in` and `get_n_out`, their names using `get_name_in` and `get_name_out` and their sparsity patterns `get_sparsity_in` and `get_sparsity_out`.

- An optional `init` function called duing the object construction.

- A function for numerical evaluation.

- Optional functions for derivatives. You can choose to work with finite differences (`enable_fd`), supply a full Jacobian (`has_jacobian`, `get_jacobian`), or choose to supply forward/reverse sensitivities (`has_forward`, `get_forward`, `has_reverse`, `get_reverse`).

For a complete list of functions, see the C++ API documentation for `Callback`. Also see the `callback.py` example.

The usage from the different languages are described in the following.

6.2.1 Python

```python
class MyCallback(Callback):
  def __init__(self, name, d, opts={}):
    Callback.__init__(self)
    self.d = d
    self.construct(name, opts)

  # Number of inputs and outputs
  def get_n_in(self): return 1
  def get_n_out(self): return 1

  # Initialize the object
  def init(self):
     print('initializing object')

  # Evaluate numerically
  def eval(self, arg):
    x = arg[0]
    f = sin(self.d*x)
    return [f]
```

The implementation should include a constructor, which should begin with a call to the base class constructor using `Callback.__init__(self)` and end with a call to initialize object construction using `self.construct(name, opts)`.

This function can be used as any built-in CasADi function with the important caveat that when embedded in graphs, the ownership of the class will *not* be shared between all references. So it is important that the user does not allow the Python class to go out of scope while it is still needed in calculations.

```python
# Use the function
f = MyCallback('f', 0.5)
print(f(2))

x = MX.sym("x")
print(f(x))
```

```
initializing object
0.841471
f(x){0}
```

6.2.2 MATLAB

In MATLAB, a custom function class can be defined as follows, in a file `MyCallback.m`:

```matlab
classdef MyCallback < casadi.Callback
  properties
    d
  end
  methods
    function self = MyCallback(name, d)
      self@casadi.Callback();
      self.d = d;
      construct(self, name);
    end

    % Number of inputs and outputs
    function v=get_n_in(self)
      v=1;
    end
    function v=get_n_out(self)
      v=1;
    end

    % Initialize the object
    function init(self)
      disp('initializing object')
    end

    % Evaluate numerically
    function arg = eval(self, arg)
      x = arg{1};
      f = sin(self.d * x);
      arg = {f};
    end
  end
end
```

This function can be used as any built-in CasADi function, but as for Python, the ownership of the class will *not* be shared between all references. So the user must not allow a class instance to get deleted while it is still in use, e.g. by making it `persistent`.

```matlab
% Use the function
f = MyCallback('f', 0.5);
res = f(2);
disp(res)

x = MX.sym('x');
disp(f(x))
```

6.2.3 C++

In C++, the syntax is as follows:

```cpp
#include "casadi/casadi.hpp"
using namespace casadi;
class MyCallback : public Callback {
  // Data members
  double d;
public:
  // Constructor
  MyCallback(const std::string& name, double d,
             const Dict& opts=Dict()) : d(d) {
    construct(name, opts);
  }

  // Destructor
  ~MyCallback() override {}

  // Number of inputs and outputs
  casadi_int get_n_in() override { return 1;}
  casadi_int get_n_out() override { return 1;}

  // Initialize the object
  void init() override() {
    std::cout << "initializing object" << std::endl;
  }

  // Evaluate numerically
  std::vector<DM> eval(const std::vector<DM>& arg) const override {
    DM x = arg.at(0);
    DM f = sin(d*x);
    return {f};
  }
};
```

A class created this way can be used as any other `Function` instance, but with the important difference that the user is responsible to managing the memory of this class.

```cpp
int main() {
  MyCallback f("f", 0.5);
  std::vector<DM> arg = {2};
  std::vector<DM> res = f(arg);
  std::cout << res << std::endl;
  return 0;
}
```

6.3 Importing a function with `external`

The basic usage of CasADi's `external` function was demonstrated in Section 5.2 in the context of using autogenerated code. The same function can also be used for importing a user-defined function, as long as it also uses the C API described in Section 5.3.

The following sections expands on this.

6.3.1 Default functions

It is usually *not* necessary to define all the functions defined in Section 5.3. If `fname_incref` and `fname_decref` are absent, it is assumed that no memory management is needed. If no names of inputs and outputs are provided, they will be given default names. Sparsity patterns are in general assumed to be scalar by default, unless the function corresponds to a derivative of another function (see below), in which case they are assumed to be dense and of the correct dimension.

Furthermore, work vectors are assumed not to be needed if `fname_work` has not been implemented.

6.3.2 Meta information as comments

If you rely on CasADi's just-in-time compiler, you can provide meta information as a comment in the C code instead of implementing the actual callback function.

The structure of such meta information should be as follows:

```
/*CASADIMETA
:fname_N_IN 1
:fname_N_OUT 2
:fname_NAME_IN[0] x
:fname_NAME_OUT[0] r
:fname_NAME_OUT[1] s
:fname_SPARSITY_IN[0] 2 1 0 2
*/
```

6.3.3 Derivatives

The external function can be made differentiable by providing functions for calculating derivatives. During derivative calculations, CasADi will look for symbols in the same file/shared library that follows a certain *naming convention*. For example, you can specify a Jacobian for all the outputs with respect to all inputs for a function named `fname` by implementing a function named `jac_fname`. Similarly, you can specify a function for calculating one forward directional derivative by providing a function named `fwd1_fname`, where 1 can be replaced by 2, 4, 8, 16, 32 or 64 for calculating multiple forward directional derivatives at once. For reverse mode directional derivatives, replace `fwd` with `adj`.

This is an experimental feature.

6.4 Just-in-time compile a C language string

In the previous section we showed how to specify a C file with functions for numerical evaluation and meta information. As was shown, this file can be just-in-time compiled by CasADi's interface to Clang. There exists a shorthand for this approach, where the user simply specifies the source code as a C language string.

```python
body =\
'r[0] = x[0];'+\
'while (r[0]<s[0]) {'+\
' r[0] *= r[0];'+\
'}'

f = Function.jit('f',body,\
    ['x','s'],['r'],\
    {"compiler":"shell"})
print(f)
```

```matlab
body =[...
'r[0] = x[0];',...
'while (r[0]<s[0]) {',...
' r[0] *= r[0];',...
'}'];

f = Function.jit('f',body,...
    {'x','s'},{'r'},...
    struct('compiler','shell'));
disp(f)
```

```
f:(x,s)->(r) JitFunction
```

```
f:(x,s)->(r) JitFunction
CasADi - 2025-09-10 17:20:51 WARNING(
↪"Failed to remove jit_tmp698lq1.c") [.../
↪casadi/core/function_internal.cpp:134]
CasADi - 2025-09-10 17:20:51 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shellQqTwwm.so") [.../casadi/solvers/
↪shell_compiler.cpp:66]
CasADi - 2025-09-10 17:20:51 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shellQqTwwm.o") [.../casadi/solvers/shell_
↪compiler.cpp:67]
```

These four arguments of `Function.jit` are mandatory: The function name, the C source as a string and the names of inputs and outputs. In the C source, the input/output names correspond to arrays of type `casadi_real_t` containing the nonzero elements of the function inputs and outputs. By default, all inputs and outputs are scalars (i.e. 1-by-1 and dense). To specify a different sparsity pattern, provide two additional function arguments containing vectors/lists of the sparsity patterns:

```
sp = Sparsity.scalar()
f = Function.jit('f',body,\
    ['x','s'], ['r'],\
    [sp,sp], [sp],\
    {"compiler":"shell"})
print(f)
```

```
sp = Sparsity.scalar();
f = Function.jit('f',body,...
    {'x','s'}, {'r'},...
    {sp,sp}, {sp},...
    struct('compiler','shell'));
disp(f)
```

```
f:(x,s)->(r) JitFunction
```

```
f:(x,s)->(r) JitFunction
CasADi - 2025-09-10 17:20:45 WARNING(
↪"Failed to remove jit_tmpawZ5yx.c") [.../
↪casadi/core/function_internal.cpp:134]
CasADi - 2025-09-10 17:20:45 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shell1cAbfa.so") [.../casadi/solvers/
↪shell_compiler.cpp:66]
CasADi - 2025-09-10 17:20:45 WARNING(
↪"Failed to remove ./tmp_casadi_compiler_
↪shell1cAbfa.o") [.../casadi/solvers/shell_
↪compiler.cpp:67]
```

Both variants accept an optional 5th (or 7th) argument in the form of an options dictionary.

6.5 Using lookup-tables

Lookup-tables can be created using CasADi's `interpolant` function. Different interpolating schemes are implemented as *plugins*, similar to `nlpsol` or `integrator` objects. In addition to the identifier name and plugin, the `interpolant` function expects a set of grid points with the corresponding numerical values.

The result of an `interpolant` call is a CasADi Function object that is differentiable, and can be embedded into CasADi computational graphs by calling with MX arguments. Furthermore, C code generation is fully supported for such graphs.

Currently, two plugins exist for `interpolant`: `'linear'` and `'bspline'`. They are intended to behave simiarly to MATLAB/Octave's `interpn` with the method set to `'linear'` or `'spline'` – corresponding to a multilinear interpolation and a (by default cubic) spline interpolation with not-a-knot boundary conditions.

In the case of `bspline`, coefficients will be sought at construction time that fit the provided data. Alternatively, you may also use the more low-level `Function.bspline` to supply the coefficients yourself. The default degree of the bspline is 3 in each dimension. You may deviate from this default by passing a `degree` option.

We will walk through the syntax of `interpolant` for the 1D and 2D versions, but the syntax in fact generalizes to an arbitrary number of dimensions.

6.5.1 1D lookup tables

A 1D spline fit can be done in CasADi/Python as follows, compared with the corresponding method in SciPy:

```python
import casadi as ca
import numpy as np
xgrid = np.linspace(1,6,6)
V = [-1,-1,-2,-3,0,2]
lut = ca.interpolant('LUT','bspline',[xgrid],V)
print(lut(2.5))
# Using SciPy
import scipy.interpolate as ip
interp = ip.InterpolatedUnivariateSpline(xgrid, V)
print(interp(2.5))
```

```
-1.35
-1.350000000000005
```

In MATLAB/Octave, the corresponding code reads:

```
xgrid = 1:6;
V = [-1 -1 -2 -3 0 2];
lut = casadi.interpolant('LUT','bspline',{xgrid},V);
lut(2.5)
% Using MATLAB/Octave builtin
interp1(xgrid,V,2.5,'spline')
```

```
ans =

-1.35

ans = -1.3500
```

Note in particular that the `grid` and `values` arguments to `interpolant` must be numerical in nature.

6.5.2 2D lookup tables

In two dimensions, we get the following in Python, also compared to SciPy for reference:

```python
xgrid = np.linspace(-5,5,11)
ygrid = np.linspace(-4,4,9)
X,Y = np.meshgrid(xgrid,ygrid,indexing='ij')
R = np.sqrt(5*X**2 + Y**2)+ 1
data = np.sin(R)/R
data_flat = data.ravel(order='F')
lut = ca.interpolant('name','bspline',[xgrid,ygrid],data_flat)
print(lut([0.5,1]))
# Using Scipy

interp = ip.RectBivariateSpline(xgrid, ygrid, data)
print(interp.ev(0.5,1))
```

```
0.245507
0.24550661674668917
```

or, in MATLAB/Octave compared to the built-in functions:

```
xgrid = -5:1:5;
ygrid = -4:1:4;
[X,Y] = ndgrid(xgrid, ygrid);
R = sqrt(5*X.^2 + Y.^2)+ 1;
V = sin(R)./R;
lut = interpolant('LUT','bspline',{xgrid, ygrid},V(:));
lut([0.5 1])
% Using Matlab builtin
interpn(X,Y,V,0.5,1,'spline')
```

```
ans =

0.245507

ans = 0.2455
```

In particular note how the `values` argument had to be flatten to a one-dimensional array.

6.6 Derivative calculation using finite differences

CasADi 3.3 introduced support for finite difference calculation for all function objects, in particular including external functions defined as outlined in Section 6.2, Section 6.3 or Section 6.4 (for lookup tables, Section 6.5, analytical derivatives are available).

Finite difference derivative are disabled by default, with the exception of `Function.jit`, and to enable it, you must set the option `'enable_fd'` to `True`/`true`:

```
f = external('f', './gen.so',\
    dict(enable_fd=True))

e = jacobian(f(x),x)
D = Function('d',[x],[e])
print(D(0))
```

```
f = external('f', './gen.so',...
    struct('enable_fd',true));

e = jacobian(f(x),x);
D = Function('d',{x},{e});
disp(D(0))
```

```
1
```

```
1
```

cf. Section 5.1.

The `'enable_fd'` options enables CasADi to use finite differences, *if* analytical derivatives are not available. To force CasADi to use finite differences, you can set `'enable_forward'`, `'enable_reverse'` and `'enable_jacobian'` to `False`/`false`, corresponding to the three types of analytical derivative information that CasADi works with.

The default method is central differences with a step size determined by estimates of roundoff errors and truncation errors of the function. You can change the method by setting the option `'fd_method'` to `'forward'` (corresponding to first order forward differences), `'backward'` (corresponding to first order backward differences) and `'smoothing'` for a second-order accurate discontinuity avoiding scheme, suitable when derivatives need to be calculated at the edges of a domain. Additional algorithmic options for the finite differences are available by setting `'fd_options'` option.

THE `DAEBUILDER` CLASS

The `DaeBuilder` class in CasADi is an auxiliary class intended to facilitate the modeling as well as import and export of complex dynamical systems. This class has lower level of abstraction than a physical modeling language such as Modelica (cf. Section 7.4), while still being higher level than working directly with CasADi symbolic expressions. In particular, it can be used to import or export physical models defined using the Functional Mock-up Interface (FMI) format. It can also be used as a building block for constructing a domain specific modeling environments.

The syntax and scope of the class has changed significantly since its initial inception. In the following, we provide an overview of the syntax introduced in CasADi 3.7.

7.1 `DaeBuilder` model variables and their categorization

Each `DaeBuilder` instance maintains a list of model variables. The variables must be uniquely named and hold import meta information about the variable such as its type (e.g. floating point, integer or sting-valued), minimum and maximum values, nominal values, dimension, description, unit etc. The format for model variables closely mimics that of the FMI standard, version 3.0, and to be able to use the `DaeBuilder` class efficently, you should probably be somewhat familar with the format. As in FMI, variables are classified according to their *causality* and *variability* (consult the FMI specification[3] if you are unfamilar with these terms). They are also given a CasADi-specific categorization into the following categories:

t

    The independent variable, usually time

c

    Constants, calculated explicitly using $c = f_{\text{cdef}}(c)$, where the dependency on c is always acyclic

p

    Independent parameters

d

    Dependent parameters, calculated explicitly using $d = f_{\text{ddef}}(p, d; c)$, where the dependency on d is always acyclic

u

    Input variables

w

    Dependent variables, calculated explicitly using $w = f_{\text{wdef}}(t, p, d, u, w, x, z; c)$, where the dependency on w is always acyclic

x

    Differential states, each defined by an ordinary differential equation (ODE), $\dot{x} = f_{\text{ode}}(t, p, d, u, w, x, z; c)$

---

[3] Modelica Association, Functional Mock-up Interface Specification, version 3.0.2

q
> Quadrature states, i.e. a differential states that do not appear in the right-hand-sides, $\dot{q} = f_{\text{quad}}(t, p, d, u, w, x, z; c)$

z
> Algebraic variables, defined implicitly using the algebraic equations, $0 = f_{\text{alg}}(t, p, d, u, w, x, z; c)$

There is also a category for *calculated* model variables, which represent different quantities that can be calculated using the above variables but cannot be used as function inputs. Calculated model variables include output variables, y, calculated explicitly using the output equations, $y = f_y(t, p, d, u, w, x, z; c)$. These quantities are referenced via the corresponding model equation, cf. Section 7.2 below. Note that in the functions above, the dependency on c is always parametric to the function definition, i.e. a new function needs to be defined if the value of c changes.

The category for a variable is determined automatically from its causality and variability according to the following table, where the rows correspond to the causality and the columns to the variability (cf. Table 18 in FMI specification, 3.0.2[Page 51, 3]):

	parameter	calculated Parameter	input	output (also in y)	local	independent
constant	n/a	n/a	n/a	c	c	n/a
fixed	c	d	n/a	n/a	d	n/a
tunable	p	d	n/a	n/a	d	n/a
discrete	n/a	n/a	u	x or q	x or q	n/a
continuous	n/a	n/a	u	x, q, z, or w	x, q, z, or w	t

Not all combinations of causality and variability are permitted, as explained in the FMI specification[Page 51, 3]. These combinations are marked "n/a" in the table. There may also be multiple combinations mapping to the same variable category.

Variables of continuous variability are given a category depending on its defining equation, which we will return to in Section 7.2 below. If the defining equation is an ODE, the variable is normally given the category x but can also be category q *if* detection of quadrature states is enabled and the variable does not appears in the right-hand-side of any other model equation. If there is a defining equation for the variable itself, i.e. if it appears in the left-hand-side of an equation, and if it has not previously appeared in any equation, it becomes part of w. If none of the above applies, the variable is given the category z and assumed implicitly defined by a residual equation. A variable with discrete variability, including integer/boolean-valued variables, is treated as a continuous variable with a trivial ODE defined, i.e. $\dot{x} = 0$ or $\dot{q} = 0$. After the user generates functions to calculate the ODE right hand sides as described in Section 7.8, this structure can easily be detected (e.g. by looking for rows in the Jacobian matrices with only structural zeros).

The set of variables with output causality also define the calculated model outputs y with associated output equation.

The category of a variable may change during symbolic construction, cf. Section 7.3, in particular when adding equations. It may also change by explicitly changing the variability or causality of a variable, when such an operation is permitted. A continuous model variable also changes from category z to x when it appears differentiated in a model equation.

7.2 `DaeBuilder` model equations

At the time of this writing, `DaeBuilder` instances supported the following types of equations, leaving out c, d and w for simplicity:

- Ordinary differential equation, e.g.: $\dot{x} = f_{\mathrm{ode}}(t, x, z, u, p)$

- Output equation, e.g.: $y = f_{\mathrm{y}}(t, x, z, u, p)$

- Algebraic equation, e.g.: $0 = f_{\mathrm{alg}}(t, x, z, u, p)$

- Quadrature equation, e.g.: $\dot{q} = f_{\mathrm{quad}}(t, x, z, u, p)$

- When equation, e.g.: when $f_{\mathrm{zero}}(t, x, z, u, p) < 0$ then $x := f_{\mathrm{transition}}(t, x, z, u, p)$

Future versions of `DaeBuilder` may support more types of equations, in particular related to proper handling of initial equations and other types of events, guided by the Modelica standard[4].

The functions in the list above are associated with additional, internal, variable categories, in addition to the categories listed in Section 7.1. For example ordinary differential equations and quadrature equations are associated with derivative variables and algebraic equations are associated with residual variables. When equations are associated with zero-crossing conditions and a set of assignment operations. We will return to when equations in Section 7.6 below.

Model equations may be given either as symbolic expressions, as described in Section 7.3, from Modelica source code as described in Section 7.4 or via standard Functional Mock-up Unit (FMU) shared libraries, as described in Section 7.5.

7.3 Constructing a `DaeBuilder` instance symbolically

We can create `DaeBuilder` instances using CasADi symbolic expressions. The current ambition is to support an acausal model approach taking elements from the Functional Mock-up Interface (FMI) standard as well as the proposed Base Modelica standard[5]. Specifically, model variables (cf. Section 7.1) are defined using the conventions defined by the FMI standard, version 3.0. The model equations (cf. Section 7.2), on the other hand, where standard FMI relies on a black-box C API, we try to follow Base Modelica. We will discuss in Section 7.4 how this support can be used to enable symbolic import of models formulated in actual Modelica.

To illustrate the usage, consider the following simple DAE corresponding to a controlled rocket subject to quadratic air friction term and gravity, which loses mass as it uses up fuel:

$$\begin{aligned}
\dot{h} &= v, & h(0) &= 0 \\
\dot{v} &= (u - a\,v^2)/m - g, & v(0) &= 0 \\
\dot{m} &= -b\,u^2, & m(0) &= 1
\end{aligned}$$

where the three states correspond to height, velocity and mass, respectively. u is the thrust of the rocket and (a, b) are parameters.

To construct a DAE formulation for this problem, start with an empty `DaeBuilder` instance and add the model variables, including any meta information, and model equations step-by-step as follows.

---

[4] Modelica Association, Modelica(R) - A Unified Object-Oriented Language for Systems Modeling Language Specification, Version 3.6

[5] Peter Harman, Werther Kai, Gerd Kurzbach, Oliver Lenord, Hans Olsson, Michael Schellenberger, Martin Sjölund, Henrik Tidefelt, Modelica Change Proposal MCP-0031 Base Modelica and MLS modularization

```
dae = DaeBuilder('rocket')
# Add model variables
a = dae.add('a', 'parameter', 'tunable')
b = dae.add('b', 'parameter', 'tunable')
u = dae.add('u', 'input')
h = dae.add('h', dict(start=0))
v = dae.add('v', dict(start=0))
m = dae.add('m', dict(start=1))
# Change some meta information
dae.set_unit('h', 'm')
dae.set_unit('v', 'm/s')
dae.set_unit('m', 'kg')
# Add model equations
dae.eq(dae.der(h), v)
dae.eq(dae.der(v), (u-a*v**2)/m-9.81)
dae.eq(dae.der(m), -b*u**2)
```

```
dae = DaeBuilder('rocket')
% Add model variables
a = dae.add('a', 'parameter', 'tunable');
b = dae.add('b', 'parameter', 'tunable');
u = dae.add('u', 'input');
h = dae.add('h', struct('start', 0));
v = dae.add('v', struct('start', 0));
m = dae.add('m', struct('start', 1));
% Change some meta information
dae.set_unit('h','m');
dae.set_unit('v','m/s');
dae.set_unit('m','kg');
% Add model equations
dae.eq(dae.der(h), v);
dae.eq(dae.der(v), (u-a*v^2)/m-9.81);
dae.eq(dae.der(m), -b*u^2);
```

```
dae =

nx = 0, nz = 0, nq = 0, ny = 0, np = 0, nc␣
→= 0, nd = 0, nw = 0, nu = 0
```

Other input and output expressions can be added in an analogous way. For a full list of functions, see the C++ API documentation for `DaeBuilder`.

In Section 7.8 below, we will show how to evaluate the model equations, which is same as if the equations had been created a different way, e.g. from an FMU, cf. Section 7.5.

7.4 Creating a `DaeBuilder` instance from Modelica, symbolically

The original goal of the `DaeBuilder` class was to enable symbolic import of models formulated in Modelica. This was made possible by supporting the import models in the "FMUX" format, an XML format derived from FMI 1.0. In this format, the model variables (cf. Section 7.1) are defined using standard FMI 1.0 XML syntax, while the model equations (c.f. Section 7.2) are provided in the same XML instead of the C API defined by the standard. The support for FMUX import in CasADi never reached a mature state, in part because there was never a mature way to generate FMUX files using available Modelica tools. While originally supported by the JModelica.org tool, this support was later dropped in favor of a tighter, SWIG-based CasADi interface. FMUX export has also been implemented in the OpenModelica compiler.

There is still legacy support for FMUX import in the `DaeBuilder` class, but since it is not actively maintained and very incomplete, it may require diving into the C++ source code to get it to work for nontrivial models.

We also note that JModelica.org's closed-source successor code, OCT, does retain CasADi interoperability. Details of how to use OCT to generate CasADi expressions is best described in OCT's user guide. As of this writing, this support did not use the DaeBuilder class in CasADi, although it is possible to create DaeBuilder instances from OCT-generated CasADi expressions, using the standard symbolic API described in Section 7.3.

In future versions of CasADi, we hope to provide a more mature support for symbolic import of Modelica models based on the emerging Base Modelica standard. Base Modelica[Page 53, 5] is intended to be a subset of the full Modelica language, which can be generated from any Modelica models, but removing many of the high-level features associated to object oriented physical modeling.

Unlike previous attempts at symbolic import, this support will also include support for hybrid (event-driven) modeling, while still retaining the differentiability needed by gradient-based optimization algorithms. Specifically, we try to ensure that the zero-crossing functions used to trigger events are differentiable and, secondly, that the state transitions are provided as differentiable functions. CasADi's Base Modelica effort currently does not include a Modelica parser, instead we are working with the open-source Rumoca[6] translator, which can be used to convert Base Modelica source code into Pythons scripts that construct DaeBuilder instances symbolically.

7.5 Constructing a `DaeBuilder` instance from an FMU

The DaeBuilder class can be used to import standard FMUs, adhering to the FMI standard version 2.0 or 3.0[2]. This is a dedicated, native, foreign function interface (FFI) of CasADi that communicates directly with the FMI C API. The evaluation takes place in CasADi function objects created using the function factory described in Section 7.8. Importantly, the binary FMI interface enables the efficient calculation of first and second derivatives of the model equations using a hybrid symbolic-numeric approach, relying on analytical derivatives for the first order derivatives and efficient finite differences implementations for the second order derivatives. Sparsity and parallelization is also exploited, whenever possible.

Not all model exchange FMUs are suitable for import into CasADi, although we are hoping to gradually support more and more parts of the standard. In general, for derivative calculations, the FMUs should support analytical derivatives. Although CasADi does support finite differencing for calculating first order derivatives, this should been seen mainly as a debugging and diagnostics feature.

Event-driven dynamics, cf. Section 7.6, have not yet been demonstrated for standard FMUs and may require some manual reformulation such as explicitly defining zero crossing function and event transitions as additional outputs. In section Section 7.8, we will discuss how to evaluate the model equations numerically.

7.6 Hybrid modeling with `DaeBuilder`

CasADi 3.7 introduces initial support for hybrid modeling and sensitivity analysis. In `DaeBuilder`, state events can be formulated with when equations, mirroring the syntax in (Base) Modelica[Page 53, 4]. Whenever a when equation is added, a differential zero crossing condition is created automatically, using a form compatible with an extension of the CasADi integrator class to enable automatic sensitivity analysis of hybrid systems. While the event support is still at an early stage, it has been successfully used to model simple hybrid systems, such as the following bouncing ball example:

---

[6] https://github.com/CogniPilot/rumoca_parser

[2] Joel Andersson, Import and Export of Functional Mockup Units in CasADi , Proceedings of the 15th International Modelica Conference 2023, Aachen, October 9-11

```
dae = DaeBuilder('bouncing_ball')
# Add model variables
h = dae.add('h', dict(start=5))
v = dae.add('v', dict(start=0))
# Add model equations
dae.eq(dae.der(h), v)
dae.eq(dae.der(v), -9.81)
dae.when(h < 0, \
  [dae.reinit('v', -0.8*dae.pre(v))])
```

```
dae = DaeBuilder('bouncing_ball')
% Add model variables
h = dae.add('h', struct('start', 5));
v = dae.add('v', struct('start', 0));
% Add model equations
dae.eq(dae.der(h), v);
dae.eq(dae.der(v), -9.81);
dae.when(h < 0, ...
  {dae.reinit('v', -0.8*dae.pre(v))});
```

```
CasADi - 2025-09-10 17:20:29 WARNING(
→"DaeBuilder::pre has not been
→implemented: Returning identity mapping")
→[.../casadi/core/dae_builder.cpp:832]
```

```
dae =

nx = 0, nz = 0, nq = 0, ny = 0, np = 0, nc
→= 0, nd = 0, nw = 0, nu = 0

CasADi - 2025-09-10 17:20:46 WARNING(
→"DaeBuilder::pre has not been
→implemented: Returning identity mapping")
→[.../casadi/core/dae_builder.cpp:832]
```

For more information about the hybrid support in CasADi, we refer to the implementation paper:[1].

7.7 Reformulating a model

Instances of the `DaeBuilder` class are mutable and it is possible, with several restrictions, to change the formulation after creation. In particular, it is possible to change the causality or variability of a variable, as long as the change is possible with the current set of model equations. In particular, it is possible to remove an output variable y by changing its causality to local. An input variable u can be treated as a parameter p or a constant c by changing the variability to tunable or fixed, respectively (the causality will be automatically updated to "parameter" in this case). In addition, it is always possible to reorder the variables in a category in an arbitrary way – by default the ordering within a category will match the ordering of the model variables.

If a `DaeBuilder` instance has been created symbolically (as opposed as from a standard FMU), additional manipulations such as the elimination of dependent variables, BLT reordering, index reduction, etc. is possible. Some of these features have not been actively maintained or continuously tested, but may be revived with limited work of the C++ source code.

7.8 Evaluating model equations, function factory

The evaluation of model equations in a `DaeBuilder` instance follows a somewhat different paradigm than other tools capable of evaluating FMUs such as FMPy or PyFMI. In general, we only use setters (`DaeBuilder.set`) to set constants (c) or *initial/default* values for other variables. For the evaluation, `DaeBuilder` relies on a *function factory* where the user creates differentiable CasADi function objects by providing a function name as well as a list of inputs and a list of outputs. The following example shows how to create a function named `f` with three (vector-valued) inputs, x, u and p, and one (vector-valued) output, f_{ode}:

```
f = dae.create('f',['x','u','p'],['ode'])
```

```
f = dae.create('f',{'x','u','p'},{'ode'});
```

---

[1] Joel Andersson, James Goppert, Event Support for Simulation and Sensitivity Analysis in CasADi for use with Modelica and FMI , Proceedings of the American Modelica Conference 2024, Storrs, Connecticut, USA, October 14-16, 2024

Similarly, we can create an output function named `h` as follows:

```
h = dae.create('h',['x','u','p'],['y'])
```

```
h = dae.create('h',{'x','u','p'},{'y'});
```

The names of inputs and outputs correspond to the categories outlined in Section 7.1 and Section 7.2, respectively. Constants (c) are never allowed as mentioned above. Dependent parameters (d) and dependent variables (w) can be *either* inputs or outputs (but not both). If they are absent from inputs (whether or not they are defined as outputs), they will be substituted out from the expressions during function construction. Outputs (y) and other purely dependent variables are only allowed as outputs. During creation, the function factory will save the current state of the (mutable) `DaeBuilder` instance and the created function will not be impacted by any subsequent changes to or even deletion of the `DaeBuilder` instance. The created functions are thus immutable (with very few exceptions), similar to all other functions in CasADi.

We may also use CasADi's naming conventions for derivative functions to include e.g. Jacobian blocks in the function output. The following example creates a function named `J` with three (vector-valued) inputs, x, u and p, and one (matrix-valued) output corresponding to the Jacobian of f_ode with respect to x:

```
J = dae.create('J',\
    ['x','u','p'], ['jac_ode_x'])
```

```
J = dae.create('J',...
    {'x','u','p'} {'jac_ode_x'});
```

We refer to the notebook *fmu_demo.ipynb* for more examples on how to use the function factory, including how to define and obtain (non-differentiable) auxiliary outputs during evaluation.

OPTIMAL CONTROL WITH CASADI

CasADi can be used to solve *optimal control problems* (OCP) using a variety of methods, including direct (a.k.a. *discretize-then-optimize*) and indirect (a.k.a. *optimize-then-discretize*) methods, all-at-once (e.g. collocation) methods and shooting-methods requiring embedded solvers of initial value problems in ODE or DAE. As a user, you are in general expected to *write your own OCP solver* and CasADi aims as making this as easy as possible by providing powerful high-level building blocks. Since you are writing the solver yourself (rather than calling an existing "black-box" solver), a basic understanding of how to solve OCPs is indispensable. Good, self-contained introductions to numerical optimal control can be found in the recent textbooks by Biegler[1] or Betts[2] or Moritz Diehl's lecture notes on numerical optimal control.

8.1 A simple test problem

To illustrate some of the methods, we will consider the following test problem, namely driving a *Van der Pol* oscillator to the origin, while trying to minimize a quadratic cost:

$$\text{minimize:} \quad \int_{t=0}^{T} (x_0^2 + x_1^2 + u^2)\, dt$$
$$x(\cdot) \in \mathbb{R}^2,\ u(\cdot) \in \mathbb{R}$$

subject to:

(8.1.1)

$$\begin{cases} \dot{x}_0 = (1 - x_1^2)\, x_0 - x_1 + u \\ \dot{x}_1 = x_0 \qquad\qquad\qquad\qquad \text{for } 0 \le t \le T \\ -1.0 \le u \le 1.0, \quad x_1 \ge -0.25 \end{cases}$$
$$x_0(0) = 0, \quad x_1(0) = 1,$$

with $T = 10$.

In CasADi's examples collection[3], you find codes for solving optimal control problems using a variety of different methods.

In the following, we will discuss three of the most important methods, namely *direct single shooting*, *direct multiple shooting* and *direct collocation*.

---

[1] Lorenz T. Biegler, Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes , SIAM 2010
[2] John T. Betts, Practical Methods for Optimal Control Using Nonlinear Programming , SIAM 2001
[3] You can obtain this collection as an archive named `examples_pack.zip` in CasADi's download area

8.1.1 Direct single-shooting

In the direct single shooting method, the control trajectory is parameterized using some piecewise smooth approximation, typically piecewise constant.

Using an explicit expression for the controls, we can then eliminate the whole state trajectory from the optimization problem, ending up with an NLP in only the discretized controls.

In CasADi's examples collection, you will find the codes `direct_single_shooting.py` and `direct_single_shooting.m` for Python and MATLAB/Octave, respectively. These codes implement the direct single shooting method and solves it with IPOPT, relying on CasADi to calculate derivatives. To obtain the discrete time dynamics from the continuous time dynamics, a simple fixed-step Runge-Kutta 4 (RK4) integrator is implemented using CasADi symbolics. Simple integrator codes like these are often useful in the context of optimal control, but care must be taken so that they accurately solve the initial-value problem.

The code also shows how the RK4 scheme can be replaced by a more advanced integrator, namely the CVODES integrator from the SUNDIALS suite, which implements a variable stepsize, variable order backward differentiation formula (BDF) scheme. An advanced integrator like this is useful for larger systems, systems with stiff dynamics, for DAEs and for checking a simpler scheme for consistency.

8.1.2 Direct multiple-shooting

The `direct_multiple_shooting.py` and `direct_multiple_shooting.m` codes, also in CasADi's examples collection, implement the direct multiple shooting method. This is very similar to the direct single shooting method, but includes the state at certain *shooting nodes* as decision variables in the NLP and includes equality constraints to ensure continuity of the trajectory.

The direct multiple shooting method is often superior to the direct single shooting method, since "lifting" the problem to a higher dimension is known to often improve convergence. The user is also able to initialize with a known guess for the state trajectory.

The drawback is that the NLP solved gets much larger, although this is often compensated by the fact that it is also much sparser.

8.1.3 Direct collocation

Finally, the `direct_collocation.py` and `direct_collocation.m` codes implement the direct collocation method. In this case, a parameterization of the entire state trajectory, as piecewise low-order polynomials, are included as decision variables in the NLP. This removes the need for the formulation of the discrete time dynamics completely.

The NLP in direct collocation is even larger than that in direct multiple shooting, but is also even sparser.

OPTI STACK

The Opti stack is a collection of CasADi helper classes that provides a close correspondence between mathematical NLP notation, e.g.

$$\begin{array}{ll} \underset{x,y}{\text{minimize}} & (y - x^2)^2 \\ \text{subject to} & x^2 + y^2 = 1 \\ & x + y \geq 1, \end{array} \tag{9.1}$$

and computer code:

```
opti = casadi.Opti()

x = opti.variable()
y = opti.variable()

opti.minimize(  (y-x**2)**2   )
opti.subject_to( x**2+y**2==1 )
opti.subject_to(       x+y>=1 )

opti.solver('ipopt')


sol = opti.solve()

print(sol.value(x))
print(sol.value(y))
```

```
opti = casadi.Opti();

x = opti.variable();
y = opti.variable();

opti.minimize(  (y-x^2)^2   );
opti.subject_to( x^2+y^2==1 );
opti.subject_to(       x+y>=1 );

opti.solver('ipopt');

sol = opti.solve();

sol.value(x)
sol.value(y)
```

```
0.7861513776531158
0.6180339888825889
```

```
ans = 0.7862
ans = 0.6180
```

The main characteristics of the Opti stack are:

- Allows *natural* syntax for constraints.

- Indexing/bookkeeping of decision variables is hidden.

- Closer mapping of numerical data-type to the host language: no encounter with `DM`.

By default, Opti will assume a nonlinear program. To solve quadratic programs, supply the string 'conic' as argument to the Opti constructor.

Note that Opti is designed to be user-friendly and there is quite some overhead with respect to using **|nlpsol|**. However, there is a **|to_function|** method (documented below) that eliminates the bulk of that overhead.

9.1 Problem specification

Variables: Declare any amount of decision variables:

- `x = opti.variable()`: scalar

- `x = opti.variable(5)`: column vector

- `x = opti.variable(5,3)`: matrix

- `x = opti.variable(5,5,'symmetric')`: symmetric matrix

The order in which you declare the variables is respected by the solver. Note that the variables are in fact plain MX symbols. You may perform any CasADi MX operations on them, e.g. embedding integrator calls.

Parameters: Declare any amount of parameters. You must fix them to a specific numerical value before solving, and you may overwrite this value at any time.

```
p = opti.parameter()
opti.set_value(p, 3)
```

```
p = opti.parameter();
opti.set_value(p, 3)
```

Objective: Declare an objective using an expression that may involve all variables or parameters. Calling the command again discards the old objective.

```
opti.minimize(   sin(x*(y-p))   )
```

```
opti.minimize(   sin(x*(y-p))   )
```

Constraints: Declare any amount of equality/inequality constraints:

- `opti.subject_to( sqrt(x+y) >= 1)`: inequality

- `opti.subject_to( sqrt(x+y) > 1)`: same as above

- `opti.subject_to( 1<= sqrt(x+y) )`: same as above

- `opti.subject_to( 5*x+y==1 )`: equality

You may also throw in several constraints at once:

```
opti.subject_to([x*y>=1,x==3])
```

```
opti.subject_to({x*y>=1,x==3});
```

You may declare double inequalities:

```
opti.subject_to( opti.bounded(0,x,1) )
```

```
opti.subject_to( 0<=x<=1 );
```

When the bounds of the double inequalities are free of variables, the constraint will be passed on efficiently to solvers that support them (notably IPOPT).

You may make element-wise (in)equalities with vectors:

```
x = opti.variable(5,1)

opti.subject_to( x*p<=3 )
```

```
x = opti.variable(5,1);

opti.subject_to( x*p<=3 )
```

Elementwise (in)equalities for matrices are not supported with a natural syntax, since there is an ambiguity with semi-definiteness constraints. The workaround is to vectorize first:

```
A = opti.variable(5,5)
opti.subject_to( vec(A)<=3 )
```

```
A = opti.variable(5,5);
opti.subject_to( A(:)<=3 );
```

Each `subject_to` command adds to the set of constraints in the problem specification. Use `subject_to()` to empty this set and start over.

Solver: You must always declare the `solver` (numerical back-end). An optional dictionary of CasADi plugin options can be given as second argument. An optional dictionary of `solver` options can be given as third argument.

```
p_opts = {"expand":True}
s_opts = {"max_iter": 100}
opti.solver("ipopt",p_opts,
                    s_opts)
```

```
p_opts = struct('expand',true);
s_opts = struct('max_iter',100);
opti.solver('ipopt',p_opts,
                    s_opts);
```

Initial guess: You may provide initial guesses for decision variables (or simple mappings of decision variables). When no initial guess is provided, numerical zero is assumed.

```
opti.set_initial(x, 2)
opti.set_initial(10*x[0], 2)
```

```
opti.set_initial(x, 2);
opti.set_initial(10*x(1), 2)
```

9.2 Problem solving and retrieving

9.2.1 Solving

After setting up the problem, you may call the solve method, which constructs a CasADi `nlpsol` and calls it.

```
sol = opti.solve()
```

```
sol = opti.solve();
```

```
This is Ipopt version 3.14.11, running with␣
→linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint␣
→Jacobian...:        2
Number of nonzeros in inequality constraint␣
→Jacobian.:        2
Number of nonzeros in Lagrangian Hessian....
→.........:        3

Total number of variables..................
→.........:        2
                     variables with only␣
→lower bounds:        0
               variables with lower and␣
→upper bounds:        0
                     variables with only␣
→upper bounds:        0
Total number of equality constraints.......
→.........:        1
Total number of inequality constraints......
→.........:        1
        inequality constraints with only␣
→lower bounds:        1
    inequality constraints with lower and␣
→upper bounds:        0
        inequality constraints with only␣
→upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
   0  0.0000000e+00 1.00e+00 1.00e+00  -1.0␣
→0.00e+00    - 0.00e+00 0.00e+00    0
   1  1.5180703e+00 2.32e-01 3.95e+08  -1.0␣
→1.11e+00    - 9.90e-01 1.00e+00h  1
   2  9.1720733e-01 1.38e-02 8.65e+08  -1.0␣
→1.05e-01  10.0 1.00e+00 1.00e+00f  1
   3  8.9187743e-01 4.67e-05 9.29e+06  -1.0␣
→7.19e-03    - 1.00e+00 1.00e+00h  1
   4  8.9179090e-01 5.46e-10 2.17e+02  -1.0␣
→2.42e-05    - 1.00e+00 1.00e+00h  1
   5  8.5961118e-01 2.43e-04 3.39e+00  -1.0␣
→1.56e-02    - 1.00e+00 1.00e+00f  1
   6  5.0768568e-01 3.58e-02 2.18e+00  -1.0␣
→1.89e-01    - 3.86e-01 1.00e+00f  1
   7  4.4751644e-01 4.12e-04 6.30e+07  -1.0␣
→1.96e-02   9.5 1.00e+00 1.00e+00h  1
   8  4.4707538e-01 4.25e-08 2.27e+04  -1.0␣
→2.53e-04    - 1.00e+00 1.00e+00h  1
   9  4.4688522e-01 9.34e-09 2.20e+00  -1.0␣
→9.32e-05    - 1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
  10  2.9995484e-02 3.46e-01 1.19e+00  -1.0␣
→5.68e-01    - 5.45e-01 1.00e+00f  1
  11  3.9964883e-04 3.56e-02 5.81e-02  -1.0␣
→2.09e-01    - 1.00e+00 1.00e+00h  1
```

```
This is Ipopt version 3.14.11, running with␣
→linear solver MUMPS 5.4.1.

Number of nonzeros in equality constraint␣
→Jacobian...:        2
Number of nonzeros in inequality constraint␣
→Jacobian.:        2
Number of nonzeros in Lagrangian Hessian....
→.........:        3

Total number of variables..................
→.........:        2
                     variables with only␣
→lower bounds:        0
               variables with lower and␣
→upper bounds:        0
                     variables with only␣
→upper bounds:        0
Total number of equality constraints.......
→.........:        1
Total number of inequality constraints......
→.........:        1
        inequality constraints with only␣
→lower bounds:        1
    inequality constraints with lower and␣
→upper bounds:        0
        inequality constraints with only␣
→upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
   0  0.0000000e+00 1.00e+00 1.00e+00  -1.0␣
→0.00e+00    - 0.00e+00 0.00e+00    0
   1  1.5180703e+00 2.32e-01 3.95e+08  -1.0␣
→1.11e+00    - 9.90e-01 1.00e+00h  1
   2  9.1720733e-01 1.38e-02 8.65e+08  -1.0␣
→1.05e-01  10.0 1.00e+00 1.00e+00f  1
   3  8.9187743e-01 4.67e-05 9.29e+06  -1.0␣
→7.19e-03    - 1.00e+00 1.00e+00h  1
   4  8.9179090e-01 5.46e-10 2.17e+02  -1.0␣
→2.42e-05    - 1.00e+00 1.00e+00h  1
   5  8.5961118e-01 2.43e-04 3.39e+00  -1.0␣
→1.56e-02    - 1.00e+00 1.00e+00f  1
   6  5.0768568e-01 3.58e-02 2.18e+00  -1.0␣
→1.89e-01    - 3.86e-01 1.00e+00f  1
   7  4.4751644e-01 4.12e-04 6.30e+07  -1.0␣
→1.96e-02   9.5 1.00e+00 1.00e+00h  1
   8  4.4707538e-01 4.25e-08 2.27e+04  -1.0␣
→2.53e-04    - 1.00e+00 1.00e+00h  1
   9  4.4688522e-01 9.34e-09 2.20e+00  -1.0␣
→9.32e-05    - 1.00e+00 1.00e+00h  1
iter    objective    inf_pr   inf_du lg(mu)␣
→ ||d||  lg(rg) alpha_du alpha_pr  ls
  10  2.9995484e-02 3.46e-01 1.19e+00  -1.0␣
→5.68e-01    - 5.45e-01 1.00e+00f  1
  11  3.9964883e-04 3.56e-02 5.81e-02  -1.0␣
→2.09e-01    - 1.00e+00 1.00e+00h  1
```

The call will fail with an error if the solver fails to convergence. You may still inspect the non-converged solution (see Section 9.3).

You may call `solve` any number of times. You will always get an immutable copy of the problem specification and its solution. Consecutively calling `solve` will not help the convergence of the problem.

To warm start a solver, you need to explicitly transfer the solution of one problem to the initial value of the next.

```python
sol1 = opti.solve()
print(sol1.stats()["iter_count"])

# Solving again makes no difference
sol1 = opti.solve()
print(sol1.stats()["iter_count"])

# Passing initial makes a difference
opti.set_initial(sol1.value_variables())
sol2 = opti.solve()
print(sol2.stats()["iter_count"])
```

```
15
15
4
```

```matlab
sol1 = opti.solve();
sol1.stats.iter_count

% Solving again makes no difference
sol1 = opti.solve();
sol1.stats.iter_count

% Passing initial makes a difference
opti.set_initial(sol1.value_variables());
sol2 = opti.solve();
sol2.stats.iter_count
```

```
ans = 15
ans = 15
ans = 4
```

In order to initialize the dual variables, e.g. when solving a set of similar optimization problems, you can use the following syntax:

```python
sol = opti.solve()
lam_g0 = sol.value(opti.lam_g)
opti.set_initial(opti.lam_g, lam_g0)
```

```matlab
sol = opti.solve();
lam_g0 = sol.value(opti.lam_g);
opti.set_initial(opti.lam_g, lam_g0);
```

9.2.2 Numerical value at the solution

Afterwards, you may retrieve the numerical values of variables (or expressions of those variables) at the solution:

- `sol.value(x)`: value of a decision variable
- `sol.value(p)`: value of a parameter
- `sol.value(sin(x+p))`: value of an expression
- `sol.value(jacobian(opti.g,opti.x))`: value of constraint jacobian

Note that the return type of `value` is sparse when applicable.

9.2.3 Numerical value at other points

You may pass a list of overruling assignment expressions to `value`. In the following code, we are asking for the value of the objective, using all optimal values at the solution, except for `y`, which we set equal to 2. Note that such statement does not modify the actual optimal value of `y` in a permanent way.

```
obj = (y-x**2)**2

opti.minimize(obj)


print(sol.value(obj,[y==2]))
```

```
1.909830056703819
```

```
obj = (y-x^2)^2;

opti.minimize(obj);



sol.value(obj,{y==2})
```

```
ans = 1.9098
```

A related usage pattern is to evaluate an expression at the initial guess:

```
print(sol.value(x**2+y,opti.initial()))
```

```
0.0
```

```
sol.value(x**2+y,opti.initial())
```

```
ans = 0
```

9.2.4 Dual variables

In order to obtain dual variables (Lagrange multipliers) of constraints, make sure you save the constraint expression first:

```
con = sin(x+y)>=1
opti.subject_to(con)
sol = opti.solve()

print(sol.value(opti.dual(con)))
```

```
0.258679501736367
```

```
con = sin(x+y)>=1;
opti.subject_to(con);
sol = opti.solve();

sol.value(opti.dual(con))
```

```
ans = 0.2587
```

9.3 Extras

It may well happen that the solver does not find an optimal solution. In such cases, you may still access the non-converged solution through debug mode:

```
opti.debug.value(x)
```

Related, you may inspect the value of an expression, at the initial guess that you supplied to the solver:

```
opti.debug.value(x,opti.initial())
```

In case the solver stops due to problem infeasibility, you may identify the problematic constraints with:

```
opti.debug.show_infeasibilities()
```

In case the solver reports NaN/Inf at a certain location, you may find out which constraint or variable is to blame by looking at its description:

```
opti.debug.x_describe(index)
opti.debug.g_describe(index)
```

You may specify a callback function; it will be called at each iteration of the solver, with the current iteration number as argument. To plot the progress of the solver, you may access the non-converged solution through debug mode:

```
opti.callback(lambda i: plot(opti.debug.value(x)))
```

```
opti.callback(@(i) plot(opti.debug.value(x)))
```

The callback may be cleared from the Opti stack by calling the `Callback` function without arguments.

You may construct a regular CasADi Function out of an Opti object, by calling the `to_function` function. The supplied list of input arguments may contain parameters, decision variables and dual variables. Inputs corresponding

to variables are taken as initial guesses. If you desire this Function to throw a runtime error when the solver fails, you may pass the `error_on_fail` as solver option. Using the resulting CasADi `Function` will be quite a bit faster than `opti.set_initial` + `opti.solve` + `sol.value`.

```
f = opti.to_function("f",[x,y],[x**2+y])
print(f(0, 0))
```

```
1.23607
```

```
f = opti.to_function('f',{x,y},{x^2+y});
disp(f(0, 0))
```

```
1.23607
```

DIFFERENCE IN USAGE FROM DIFFERENT LANGUAGES

10.1 General usage

Example:

Table 10.1.1: General usage

Task	Python	C++	MATLAB/Octave
Starting CasADi	`from casadi import *`	`#include <casadi/` `→casadi.hpp>` `using namespace` `→casadi;`	`import casadi.*`
Printing object	`print(A)`	`std::cout << A;`	`disp(A)`
Printing with type information	A <ENTER> (interactive), `print(repr(A))`	`std::cout <<` `→repr(A);`	A <ENTER> (interactive), `disp(repr(A))`
Get (extended) representation, `more=false` by default	`A.str(more)`	`str(A, more)`	`str(A, more)`
Calling a class function	`SX.zeros(3,4)`	`SX::zeros(3,4)`	`SX.zeros(3,4)`
Creating a dictionary (e.g. for options)	`d = dict(opt1=opt1)` `#or` `d = {'opt1':opt1}` `#or` `d = {}; a['opt1'] =` `→opt1`	`Dict a;` `a["opt1"] = opt1;`	`a = struct;` `a.opt1 = opt1;` `#or` `a = struct('opt1',` `→opt1);`
Creating a symbol	`MX.sym("x",2,2)`	`MX::sym("x",2,2)`	`MX.sym('x',2,2)`
Creating a function	`Function("f",[x,y],` `→[x+y])`	`Function("f",{x,y},` `→{x+y})`	`Function('f',{x,y},` `→{x+y})`
Calling a function (form 1)	`z=f(x,y)`	`z = f({x,y})`	`z=f(x,y)`
Calling a function (form 2)	`res = f(x=x,y=y)` `res["z"]`	`auto res =\` `f({{"x",x},{"y",y}}` `→);` `res["z"]`	`res=f('x',x,'y',y)` `res.z`
Create an NLP solver	`nlp = {"x":x,"f":f}` `nlpsol("S","ipopt",` `→nlp)`	`MXDict nlp =\` `{{"x",x},{"f",f}};` `nlpsol("S","ipopt",` `→nlp);`	`nlp=struct('x',x,'f` `→',f);` `nlpsol('S','ipopt',` `→nlp);`

10.2 List of operations

The following is a list of the most important operations. Operations that differ between the different languages are marked with a star (*). This list is neither complete, nor does it show all the variants of each operation. Further information is available in the API documentation.

Table 10.2.1: General usage

Task	Python	C++	MATLAB/Octave
Addition, subtraction	`x+y, x-y, -x`	`x+y, x-y, -x`	`x+y, x-y, -x`
Elementwise multiplication, division	`x*y, x/y`	`x*y, x/y`	`x.*y, x./y`

Matrix multiplication	`x @ y`
	`# or before Py3.4:`
	`mtimes(x,y)`
		`mtimes(x,y)`	`x*y`

Linear system solve	`solve(A,b)`	`solve(A,b)`	`A\b`
Natural exponential function and logarithm	`exp(x), log(x)`	`exp(x), log(x)`	`exp(x), log(x)`
Exponentiation	`x**y`	`pow(x,y)`	`x^y, x.^y`
Square root	`sqrt(x)`	`sqrt(x)`	`sqrt(x)`
Trigonometric functions	`sin(x), cos(x), tan(x)`	`sin(x), cos(x), tan(x)`	`sin(x), cos(x), tan(x)`
Inverse trigonometric	`asin(x), acos(x)`	`asin(x), acos(x)`	`asin(x), acos(x)`
Two argument arctangent	`atan2(y, x)`	`atan2(y, x)`	`atan2(y, x)`
Hyperbolic functions	`sinh(x), cosh(x), tanh(x)`	`sinh(x), cosh(x), tanh(x)`	`sinh(x), cosh(x), tanh(x)`
Inverse hyperbolic	`asinh(x), acosh(x)`	`asinh(x), acosh(x)`	`asinh(x), acosh(x)`

continues on next page

Table 10.2.1 – continued from previous page

Task	Python	C++	MATLAB/Octave
Inequalities	`a<b, a<=b, a>b, a>=b`	`a<b, a<=b, a>b, a>=b`	`a<b, a<=b, a>b, a>=b`
(*Not*) equal to	`a==b, a!=b`	`a==b, a!=b`	`a==b, a~=b`
Logical and	`logic_and(a, b)`	`a && b`	`a & b`
Logical or	`logic_or(a, b)`	`a || b`	`a | b`
Logical not	`logic_not(a)`	`!a`	`~a`
Round to integer	`floor(x), ceil(x)`	`floor(x), ceil(x)`	`floor(x), ceil(x)`
Modulus after division	`fmod(x, y),␣↪remainder(x,y)`	`fmod(x, y),␣↪remainder(x,y)`	`rem(x, y),␣↪remainder(x,y)`
Absolute value	`fabs(x)`	`fabs(x)`	`abs(x)`
Norm	`norm_1(x)`	`norm_1(x)`	`norm(x,1)`
Sign function	`sign(x)`	`sign(x)`	`sign(x)`
(Inverse) error function	`erf(x), erfinv(x)`	`erf(x), erfinv(x)`	`erf(x), erfinv(x)`
Elementwise min and max	`fmin(x, y), fmax(x,␣↪y)`	`fmin(x, y), fmax(x,␣↪y)`	`min(x, y), max(x, y)`
Global min and max	`mmin(x), mmax(x)`	`mmin(x), mmax(x)`	`mmin(x), mmax(x)`
Index of first nonzero	`find(x)`	`find(x)`	`find(x)`
If-then-else	`if_else(c, x, y)`	`if_else(c, x, y)`	`if_else(c, x, y)`

CasADi Documentation, Release 3.7.2

Table 10.2.1 – continued from previous page

Task	Python	C++	MATLAB/Octave
Transpose	`A.T`	`A.T()`	`A',A.'`
Inner product	`dot(x, y)`	`dot(x, y)`	`dot(x, y)`

Horizontal/vertical concatenation	`horzcat(x, y)`
	`vertcat(x, y)`
	`hcat([x, y])`
	`vcat([x, y])`
		`horzcat(x, y)`
		`vertcat(x, y)`
		`MX::horzcat({x,y})`
		`MX::vertcat({x,y})`
			`[x,y]`
			`[x; y]`
			`c = {x,y};`
			`[c{:}]`
			`vertcat(c{:})`

Horizontal/vertical split (inverse of concatenation)	`vertsplit(x)` `horzsplit(x)`	`vertsplit(x)` `horzsplit(x)`	`vertsplit(x)` `horzsplit(x)`

Element access	`# 0-based`
	`A[i,j]`
	`A[i]`
		`// 0-based`
		`A(i,j)`
		`A(i)`
			`%  1-based`
			`A(i,j)`
			`A(i)`

Element assignment	`# 0-based`
	`A[i,j] = b`
	`A[i] = b`
		`// 0-based`
		`A(i,j) = b`
		`A(i) = b`
			`%  1-based`
			`A(i,j) = b`
			`A(i) = b`

Nonzero access	`# 0-based` `A.nz[k]`	`// 0-based` `A.nz(k)`	`(currently␣` `↪unsupported)`
Nonzero assignment	`# 0-based` `A.nz[k] = b`	`// 0-based` `A.nz(k) = b`	`(currently␣` `↪unsupported)`
Project to a different sparsity	`project(x, s)`	`project(x, s)`	`project(x, s)`