User Documentation for `CasADi` v3.3.0-194.a1d1a5d78

Joel Andersson Joris Gillis Moritz Diehl

February 11, 2018

Contents

1 Introduction **5**
- 1.1 What CasADi is and what it is *not* . . . 5
- 1.2 Help and support . . . 6
- 1.3 Citing CasADi . . . 6
- 1.4 Reading this document . . . 6

2 Obtaining and installing CasADi **9**

3 Symbolic framework **11**
- 3.1 The SX symbolics . . . 11
- 3.2 DM . . . 13
- 3.3 The MX symbolics . . . 14
- 3.4 Mixing SX and MX . . . 15
- 3.5 The Sparsity class . . . 16
  - 3.5.1 Getting and setting elements in matrices . . . 17
- 3.6 Arithmetic operations . . . 18
- 3.7 Querying properties . . . 20
- 3.8 Linear algebra . . . 21
- 3.9 Calculus – algorithmic differentiation . . . 21

4 Function objects **23**
- 4.1 Calling function objects . . . 24
- 4.2 Converting MX to SX . . . 25
- 4.3 Nonlinear root-finding problems . . . 25
- 4.4 Initial-value problems and sensitivity analysis . . . 26
  - 4.4.1 Creating integrators . . . 26
  - 4.4.2 Sensitivity analysis . . . 27
- 4.5 Nonlinear programming . . . 28
  - 4.5.1 Creating NLP solvers . . . 28
- 4.6 Quadratic programming . . . 29
  - 4.6.1 High-level interface . . . 29
  - 4.6.2 Low-level interface . . . 30

5 Generating C-code **33**
5.1 Syntax for generating code . . . 33
5.2 Using the generated code . . . 35
5.3 API of the generated code . . . 37

6 User-defined function objects **41**
6.1 Subclassing `FunctionInternal` . . . 41
6.2 Subclassing `Callback` . . . 42
6.3 Importing a function with `external` . . . 45
6.4 Just-in-time compile a C language string . . . 46
6.5 Using lookup-tables . . . 47
6.5.1 1D lookup tables . . . 47
6.5.2 2D lookup tables . . . 48
6.6 Derivative calculation using finite differences . . . 49

7 The `DaeBuilder` class **51**
7.1 Mathematical formulation . . . 51
7.2 Constructing a `DaeBuilder` instance . . . 52
7.3 Import of OCPs from Modelica . . . 53
7.4 Symbolic reformulation . . . 54
7.5 Function factory . . . 55

8 Optimal control with `CasADi` **57**
8.1 A simple test problem . . . 57
8.2 Direct single-shooting . . . 58
8.3 Direct multiple-shooting . . . 58
8.4 Direct collocation . . . 59

9 Opti stack **61**
9.1 Problem specification . . . 62
9.2 Problem solving and retrieving . . . 64
9.3 Extras . . . 65

10 Difference in usage from different languages **67**
10.1 General usage . . . 67
10.2 List of operations . . . 67

Chapter 1

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:

```
@PHDTHESIS{Andersson2013b,
  author = {Joel Andersson},
  title = {{A} {G}eneral-{P}urpose {S}oftware {F}ramework for
           {D}ynamic {O}ptimization},
  school = {Arenberg Doctoral School, KU Leuven},
  year = {2013},
  type = {{P}h{D} thesis},
  address = {Department of Electrical Engineering (ESAT/SCD) and
             Optimization in Engineering Center,
             Kasteelpark Arenberg 10, 3001-Heverlee, Belgium},
  month = {October}
}
```

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.

Chapter 2

Obtaining and installing CasADi

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 website: http://casadi.org.

Chapter 3

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 *
```

```
% MATLAB/Octave
import casadi.*
```

Now create a variable `x` using the syntax:

```
# Python
x = MX.sym('x')
```

```
% MATLAB/Octave
x = MX.sym('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:

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

```
% MATLAB/Octave
y = SX.sym('y',5);
Z = SX.sym('Z',4,2);
```

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:

```
# Python
f = x**2 + 10
f = sqrt(f)
print('f:', f)
```

```
% MATLAB/Octave
f = x^2 + 10;
f = sqrt(f);
display(f)
```

```
('f:', MX(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

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:

```
# Python
print('B4:', B4)
```

```
% MATLAB/Octave
display(B4)
```

```
('B4:', SX(@1=1,
[[@1, 00, 00, 00],
 [00, @1, 00, 00],
 [00, 00, @1, 00],
 [00, 00, 00, @1]]))
```

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. $9 * \text{SX.ones}(2,2)$.
- 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]]) .

Note for MATLAB/Octave users

In MATLAB, if the `import` 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.

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:

```
// C++
#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:

```
# Python
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
```

```
% MATLAB/Octave
C = DM(2,3);

C_dense = full(C);

C_sparse = sparse(C);
```

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

3.3 The MX symbolics

Let us perform a simple operation using the `SX` above:

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

```
% MATLAB/Octave
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)
```

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 ($\mathbb{R} \to \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. 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:

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

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

```
((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:

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

```
% MATLAB/Octave
x = MX.sym('x',2,2);
x(1,1)
```

```
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:

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

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

```
('A:', MX((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.

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

% MATLAB/Octave
disp(SX.sym('x',Sparsity.lower(3)))
```

```
[[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.

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

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

```
[[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:

```
# Python
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):

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

```
% MATLAB/Octave
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]]
```

```
print(M[0,0], M[1,0], M[-1,-1])
(SX(3), SX(00), SX(6))
```

```
M(1,1), M(2,1), M(end,end)
```

```
print(M[5], M[-6])
(SX(4), SX(5))
```

```
M(6), M(end-5)
```

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])
```
```
[00, 4, 00, 00]
```
```
disp(M(:,2))
```

```
print(M[1:,1:4:2])
```
```
[[4, 00],
 [00, 00],
 [00, 6]]
```
```
disp(M(2:end,2:2:4))
```

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

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

3.6 Arithmetic operations

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

```
x = SX.sym('x')
y = SX.sym('y',2,2)
print(sin(y)-x)
```
```
[[(sin(y_0)-x), (sin(y_2)-x)],
 [(sin(y_1)-x), (sin(y_3)-x)]]
```
```
x = SX.sym('x');
y = SX.sym('y',2,2);
sin(y)-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 mtimes(A,B):

```
print(y*y, mtimes(y,y))          y.*y, y*y
```

```
(SX(
[[sq(y_0), sq(y_2)],
 [sq(y_1), sq(y_3)]]), SX(
[[(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.T)                       y'
```

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

```
@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 list of input arguments) in Python and C++ and with square brackets in MATLAB:

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

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

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

```
[[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 $\mathit{offset}[i] \leq c < \mathit{offset}[i+1]$. The following demonstrates the syntax:

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

```
(SX([x_0, x_1, x_2, x_3, x_4]), SX([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])               w = vertsplit(x,[0,3,5]);
print(w[0], w[1])                      w{1}, w{2}
```

```
(SX(
[[x_0, x_5],
 [x_1, x_6],
 [x_2, x_7]]), SX(
[[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,:]]               w = {x(1:3,:), x(4:5,:)};
print(w[0], w[1])                      w{1}, w{2}
```

```
(SX(
[[x_0, x_5],
```

```
 [x_1, x_6],
 [x_2, x_7]]), SX(
[[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 $< A, B >:= \mathrm{tr}(A\,B) = \sum_{i,j} A_{i,j}\,B_{i,j}$ are created as follows:

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

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

```
(((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)
```

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);
solve(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 $f : \mathbb{R}^N \to \mathbb{R}^M$:

$$y = f(x), \tag{3.1}$$

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

$$\hat{y} = \frac{\partial f}{\partial x}\,\hat{x}. \tag{3.2}$$

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

$$\bar{x} = \left(\frac{\partial f}{\partial x}\right)^{\mathrm{T}}\,\bar{y}. \tag{3.3}$$

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.

Syntax

An expression for a Jacobian is obtained using the syntax:

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

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)]]
```

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:', SX(@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:

```
v = SX.sym('v',2)
f = mtimes(A,x)
print(jtimes(f,x,v))
```

```
v = SX.sym('v',2);
f = A*x;
jtimes(f,x,v)
```

```
[((A_0*v_0)+(A_3*v_1)), ((A_1*v_0)+(A_4*v_1)), ((A_2*v_0)+(A_5*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)
```

```
[(((A_2*w_2)+(A_1*w_1))+(A_0*w_0)), (((A_5*w_2)+(A_4*w_1))+(A_3*w_0))]
```

Chapter 4

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:

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

```
% MATLAB/Octave
x = SX.sym('x',2);
y = SX.sym('y');
f = Function('f',{x,y},...
             {x,sin(y)*x});
```

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:

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

```
% MATLAB/Octave
x = MX.sym('x',2);
y = MX.sym('y');
f = Function('f',{x,y},...
             {x,sin(y)*x});
```

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

```python
# Python
x = MX.sym('x',2)
y = MX.sym('y')
f = Function('f',[x,y],\
      [x,sin(y)*x],\
      ['x','y'],['r','q'])
```

```
% MATLAB/Octave
x = MX.sym('x',2);
y = MX.sym('y');
f = Function('f',{x,y},...
      {x,sin(y)*x},...
      {'x','y'},{'r','q'});
```

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. E.g. `f.jacobian('x','q')` instead of `f.jacobian(0,1)`.

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:

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

```
% MATLAB/Octave
[r0, q0] = f(1.1,3.3);
display(r0)
display(q0)
```

```
('r0:', DM([1.1, 1.1]))
('q0:', DM([-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++):

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

```
% MATLAB/Octave
res = f('x',1.1,'y',3.3);
display(res)
```

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

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:

```
# Python
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)
```

```
% MATLAB/Octave
arg = {1.1,3.3};
res = f.call(arg);
display(res)
arg = struct('x',1.1,'y',3.3);
res = f.call(arg);
display(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])})
```

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:

$$\begin{aligned} g_0(z, x_1, x_2, \ldots, x_n) &= 0 \\ g_1(z, x_1, x_2, \ldots, x_n) &= y_1 \\ g_2(z, x_1, x_2, \ldots, x_n) &= y_2 \\ &\vdots \\ g_m(z, x_1, x_2, \ldots, x_n) &= y_m, \end{aligned} \tag{4.1}$$

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:

```
# Python
z = SX.sym('x',nz)
x = SX.sym('x',nx)
g0 = (an expression of x, z)
g1 = (an expression of x, z)
g = Function('g',[z,x],[g0,g1])
G = rootfinder('G','newton',g)
```

```
% MATLAB/Octave
z = SX.sym('x',nz);
x = SX.sym('x',nx);
g0 = (an expression of x, z)
g1 = (an expression of x, z)
g = Function('g',{z,x},{g0,g1});
G = rootfinder('G','newton',g);
```

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.

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:

$$\dot{x} = f_{\text{ode}}(t, x, z, p), \qquad x(0) = x_0 \tag{4.2a}$$

$$0 = f_{\text{alg}}(t, x, z, p) \tag{4.2b}$$

$$\dot{q} = f_{\text{quad}}(t, x, z, p), \qquad q(0) = 0 \tag{4.2c}$$

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 a guess for the algebraic variables (only for DAEs) `z0` and returns the state vector `xf`, algebraic variables `zf` and the quadrature state `qf`, all at the final time.

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:

$$\dot{x} = z + p, \tag{4.3a}$$
$$0 = z\cos(z) - x \tag{4.3b}$$

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

```
# Python
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)
```

```
% MATLAB/Octave
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);
```

Integrating this DAE from 0 to 1 with $x(0) = 0$, $p = 0.1$ and using the guess $z(0) = 0$, can be done by evaluating the created function object:

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

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

```
0.1724
```

The time horizon is assumed to be fixed[1] and can be changed from its default [0, 1] by setting the options "t0" and "tf".

[1] for problems with free end time, you can always scale time by introducing an extra parameter and substitute t for a dimensionless time variable that goes from 0 to 1

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

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

$$\begin{array}{ll} \underset{x}{\text{minimize:}} & f(x,p) \\ \text{subject to:} & \begin{array}{rcl} x_{\text{lb}} \leq & x & \leq x_{\text{ub}} \\ g_{\text{lb}} \leq & g(x,p) & \leq g_{\text{ub}} \end{array} \end{array} \tag{4.4}$$

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 ($L(x,\lambda) = f(x) + \lambda^{\mathrm{T}} g(x)$) 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:

$$\begin{array}{ll} \underset{x,y,z}{\text{minimize:}} & x^2 + 100\,z^2 \\ \text{subject to:} & z + (1-x)^2 - y = 0 \end{array} \tag{4.5}$$

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

```
# Python
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)
```

```
% MATLAB/Octave
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)
```

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

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
       For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

This is Ipopt version 3.12.4, running with linear solver ma57.

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  0.0000000e+00 3.19e-01 0.00e+00 -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 3.19e-01  - 1.00e+00 1.00e+00h 1

Number of Iterations....: 10

                                   (scaled)                 (unscaled)
Objective...............: 0.0000000000000000e+00 0.0000000000000000e+00
Dual infeasibility......: 0.0000000000000000e+00 0.0000000000000000e+00
Constraint violation....: 0.0000000000000000e+00 0.0000000000000000e+00
Complementarity.........: 0.0000000000000000e+00 0.0000000000000000e+00
Overall NLP error.......: 0.0000000000000000e+00 0.0000000000000000e+00


Number of objective function evaluations       = 11
Number of objective gradient evaluations       = 11
```

```
Number of equality constraint evaluations       = 11
Number of inequality constraint evaluations     = 0
Number of equality constraint Jacobian evaluations = 11
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations        = 10
Total CPU secs in IPOPT (w/o function evaluations) =  0.000
Total CPU secs in NLP function evaluations      =     0.000

EXIT: Optimal Solution Found.
             t_proc [s]  t_wall [s]   n_eval
          S     0.00358     0.00358       1
       nlp_f       7e-06    5.89e-06      11
       nlp_g     1.1e-05    1.04e-05      11
  nlp_grad_f     1.1e-05    1.02e-05      12
  nlp_hess_l       6e-06    6.81e-06      10
   nlp_jac_g       6e-06    7.26e-06      12
```

```
('x_opt: ', DM([0, 1, 0]))
```

4.6 Quadratic programming

CasADi provides interfaces to solve quadratic programs (QPs). Supported solvers are the open-source solvers qpOASES (distributed with CasADi) and OOQP 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 (4.4), 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:

$$\begin{array}{ll} \text{minimize:} & x^2 + y^2 \\ x, y & \\ \text{subject to:} & x + y - 10 \geq 0 \end{array} \tag{4.6}$$

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:

```
# Python
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)
```

```
% MATLAB/Octave
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)
```

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:

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

```
% MATLAB/Octave
r = S('lbg',0);
x_opt = r.x;
display(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: ', DM([5, 5]))
```

4.6.2 Low-level interface

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

$$\begin{aligned} \underset{x}{\text{minimize:}} \quad & \tfrac{1}{2} x^{\mathrm{T}} H\, x + g^{\mathrm{T}} x \\ \text{subject to:} \quad & x_{\mathrm{lb}} \leq \quad x \quad \leq x_{\mathrm{ub}} \\ & a_{\mathrm{lb}} \leq A\, x \leq a_{\mathrm{ub}} \end{aligned} \tag{4.7}$$

Encoding problem (4.6) in this form, omitting bounds that are infinite, is straightforward:

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

```
% MATLAB/Octave
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:

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

```
% MATLAB/Octave
qp = struct;
qp.h = H.sparsity();
qp.a = A.sparsity();
S = conic('S','qpoases',qp);
```

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:

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

```
% MATLAB/Octave
r = S('h', H, 'g', g,...
      'a', A, 'lba', lba);
x_opt = r.x;
display(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: ', DM([5, 5]))
```

Chapter 5

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
# 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')
```

```
% MATLAB/Octave
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');
```

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 5.2 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:

```python
# Python
f = Function('f',[x],[sin(x)])
g = Function('g',[x],[cos(x)])
C = CodeGenerator('gen.c')
C.add(f)
C.add(g)
C.generate()
```

```
% MATLAB/Octave
f = Function('f',{x},{sin(x)});
g = Function('g',{x},{cos(x)});
C = CodeGenerator('gen.c');
C.add(f);
C.add(g);
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:

```python
# Python
f = Function('f',[x],[sin(x)])
opts = dict(main=True, \
            mex=True)
f.generate('gen.c',opts)
```

```
% MATLAB/Octave
f = Function('f',{x},{sin(x)});
opts = struct('main', true,...
              'mex', true);
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.

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.

`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:

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

```
% MATLAB/Octave
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:

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

```python
# Python
C = Importer('gen.c','clang')
f = external('f',C);
print(f(3.14))
```

```
% MATLAB/Octave
C = Importer('gen.c','clang');
f = external('f',C);
disp(f(3.14))
```

```
[0.00159265, 0.00159265]
```

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

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:

```
% MATLAB/Octave
mex gen.c -largeArrayDims
disp(gen('f', 3.14))
```

```
Building with 'Xcode with Clang'.
MEX completed successfully.
   (1,1)      0.0016
   (2,1)      0.0016
```

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.

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
```

```
0.00159265 0.00159265
```

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.

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.

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.

Number of inputs and outputs

```
int fname_n_in(void);
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.

Names of inputs and outputs

```
const char* fname_name_in(int ind);
const char* fname_name_out(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.

Sparsity patterns of inputs and outputs

```
const int* fname_sparsity_in(int ind);
const int* fname_sparsity_out(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 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.

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;

Work vectors

```
int fname_work(int* sz_arg, int* sz_res, int* sz_iw, 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*`, `int` and `double`, respectively.

The return value of the function is nonzero upon failure.

Numerical evaluation

```
int fname(const double** arg, double** res,
          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.

Chapter 6

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`: 6.1
- Subclassing `Callback`: 6.2
- Importing a function with `external`: 6.3
- Just-in-time compile a C language string: 6.4
- Replace the function call with a lookup table: 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 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 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`.

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

Python

In Python, a custom function class can be defined is as follows:

```
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.

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

MATLAB

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

```
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`.

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

C++

In C++, the syntax is as follows:

```
#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
  int get_n_in() override { return 1;}
  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.

```
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.

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.

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
*/
```

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`. Similary, 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];\n'+\
'while (r[0]<s[0]) {\n'+\
' r[0] *= r[0];\n'+\
'}\n'

f = Function.jit('f',body,\
      ['x','s'],['r'])
```

```
% MATLAB/Octave
body =[...
'r[0] = x[0];\n',...
'while (r[0]<s[0]) {\n',...
' r[0] *= r[0];\n',...
'}\n'];

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

These four arguments of Function.jit are manditory: 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:

```
# Equivalent to the above
sp = Sparsity.scalar()
f = Function.jit('f',body,\
      ['x','s'], ['r'],\
      [sp,sp], [sp])
```

```
% Equivalent to the above
sp = Sparsity.scalar();
f = Function.jit('f',body,...
      {'x','s'}, {'r'});
      {sp,sp}, {sp});
```

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))
```

In MATLAB/Octave, the corresponding code reads:

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

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))
```

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

```
% MATLAB/Octave
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')
```

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:

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

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

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.

Chapter 7

The `DaeBuilder` class

The `DaeBuilder` class in `CasADi` is an auxiliary class intended to facilitate the modeling complex dynamical systems for later use with optimal control algorithms. This class can be seen as a low-level alternative to a physical modeling language such as Modelica (cf. Section 7.3), while still being higher level than working directly with `CasADi` symbolic expressions. Another important usage it to provide an interface to physical modeling languages and software and be a building blocks for developing domain specific modeling environments.

Using the `DaeBuilder` class consists of the following steps:

- Step-by-step constructing a structured system of differential-algebraic equations (DAE) or, alternatively, importing an existing model from Modelica
- Symbolically reformulate the DAE
- Generate a chosen set of `CasADi` functions to be used for e.g. optimal control or C code generation

In the following sections, we describe the mathematical formulation of the class and its intended usage.

7.1 Mathematical formulation

The `DaeBuilder` class uses a relatively rich problem formulation that consists of a set of input expressions and a set of output expressions, each defined by a string identifier. The choice of expressions was inspired by the *functional mock-up interface* (FMI) version 2.0 [1]

Input expressions

't' Time t

[1] FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/`, July 2014. Specification, FMI 2.0. Section 3.1, pp. 7172

'c' Named constants c

'p' Independent parameters p

'd' Dependent parameters d, depends only on p and c and, acyclically, on other d

'x' Differential state x, defined by an explicit ODE

's' Differential state s, defined by an implicit ODE

'sdot' Time derivatives implicitly defined differential state $\dot{s}$

'z' Algebraic variable, defined by an algebraic equation

'q' Quadrature state q. A differential state that may not appear in the right-hand-side and hence can be calculated by quadrature formulas.

'w' Local variables w. Calculated from time and time dependent variables. They may also depend, acyclically, on other w.

'y' Output variables y

Output expressions

The above input expressions are used to define the following output expressions:

'ddef' Explicit expression for calculating d

'wdef' Explicit expression for calculating w

'ode' The explicit ODE right-hand-side: $\dot{x} = \mathrm{ode}(t, w, x, s, z, u, p, d)$

'dae' The implicit ODE right-hand-side: $\mathrm{dae}(t, w, x, s, z, u, p, d, \dot{s}) = 0$

'alg' The algebraic equations: $\mathrm{alg}(t, w, x, s, z, u, p, d) = 0$

'quad' The quadrature equations: $\dot{q} = \mathrm{quad}(t, w, x, s, z, u, p, d)$

'ydef' Explicit expressions for calculating y

7.2 Constructing a DaeBuilder instance

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:

$$\dot{h} = v, \qquad h(0) = 0 \tag{7.1a}$$

$$\dot{v} = (u - a\,v^2)/m - g, \qquad v(0) = 0 \tag{7.1b}$$

$$\dot{m} = -b\,u^2, \qquad m(0) = 1 \tag{7.1c}$$

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 input and output expressions step-by-step as follows.

```python
# Python
dae = DaeBuilder()
# Add input expressions
a = dae.add_p('a')
b = dae.add_p('b')
u = dae.add_u('u')
h = dae.add_x('h')
v = dae.add_x('v')
m = dae.add_x('m')
# Add output expressions
hdot = v
vdot = (u-a*v**2)/m-g
mdot = -b*u**2
dae.add_ode('hdot', hdot)
dae.add_ode('vdot', vdot)
dae.add_ode('mdot', mdot)
# Specify initial conditions
dae.set_start('h', 0)
dae.set_start('v', 0)
dae.set_start('m', 1)
# Add meta information
dae.set_unit('h','m')
dae.set_unit('v','m/s')
dae.set_unit('m','kg')
```

```
% MATLAB/Octave
dae = DaeBuilder;
% Add input expressions
a = dae.add_p('a');
b = dae.add_p('b');
u = dae.add_u('u');
h = dae.add_x('h');
v = dae.add_x('v');
m = dae.add_x('m');
% Add output expressions
hdot = v;
vdot = (u-a*v^2)/m-g;
mdot = -b*u^2;
dae.add_ode('hdot', hdot);
dae.add_ode('vdot', vdot);
dae.add_ode('mdot', mdot);
% Specify initial conditions
dae.set_start('h', 0);
dae.set_start('v', 0);
dae.set_start('m', 1);
% Add meta information
dae.set_unit('h','m');
dae.set_unit('v','m/s');
dae.set_unit('m','kg');
```

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`.

7.3 Import of OCPs from Modelica

An alternative to model directly in `CasADi`, as above, is to use an advanced physical modeling language such as Modelica to specify the model. For this, `CasADi` offers interoperability with the open-source JModelica.org compiler, which is written specifically with optimal control in mind. Model import from JModelica.org is possible in two different ways; using the JModelica.org's `CasadiInterface` or via `DaeBuilder`'s parse_fmi command.

We recommend the former approach, since it is being actively maintained and refer to JModelica.org's user guide for details on how to extract `CasADi` expressions.

In the following, we will outline the legacy approach, using parse_fmi.

Legacy import of a modelDescription.xml file

To see how to use the Modelica import, look at thermodynamics_example.py in CasADi's example collection.

Assuming that the Modelica/Optimica model ModelicaClass.ModelicaModel is defined in the files file1.mo and file2.mop, the Python compile command is:

```
from pymodelica import compile_jmu
jmu_name=compile_jmu('ModelicaClass.ModelicaModel', \
  ['file1.mo','file2.mop'],'auto','ipopt',\
  {'generate_xml_equations':True, 'generate_fmi_me_xml':False})
```

This will generate a jmu-file, which is essentially a zip file containing, among other things, the file modelDescription.xml. This XML-file contains a symbolic representation of the optimal control problem and can be inspected in a standard XML editor.

```
from zipfile import ZipFile
sfile = ZipFile(jmu_name','r')
mfile = sfile.extract('modelDescription.xml','.')
```

Once a modelDescription.xml file is available, it can be imported using the parse_fmi command:

```
dae = DaeBuilder()
ocp.parse_fmi('modelDescription.xml')
```

7.4 Symbolic reformulation

One of the original purposes of the DaeBuilder class was to reformulate a *fully-implicit DAE*, typically coming from Modelica, to a semi-explicit DAE that can be used more readily in optimal control algorithms.

This can be done by the make_implicit command:

```
# Python
ocp.make_explicit()
```

```
% MATLAB/Octave
ocp.make_explicit();
```

Other useful commands available for an instance ocp of DaeBuilder include:

print ocp Print the optimal optimal control problem to screen

ocp.scale_variables() Scale all variables using the *nominal* attribute for each variable

ocp.eliminate_d() Eliminate all independent parameters from the symbolic expressions

For a more detailed description of this class and its functionalities, we again refer to the API documentation.

7.5 Function factory

Once a `DaeBuilder` has been formulated and possibly reformulated to a satisfactory form, we can generate `CasADi` functions corresponding to the input and output expressions outlined in Section 7.1. For example, to create a function for the ODE right-hand-side for the rocket model in Section 7.2, simply provide a display name of the function being created, a list of input expressions and a list of output expressions:

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

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

Using a naming convention, we can also create Jacobians, e.g. for the 'ode' output with respect to 'x':

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

```
% MATLAB/Octave
f = dae.create('f',...
      {'x','u','p'},
      {'jac_ode_x'});
```

Functions with second order information can be extracted by first creating a named linear combination of the output expressions using add_lc and then requesting its Hessian:

```
# Python
dae.add_lc('gamma',['ode'])
hes = dae.create('hes',\
  ['x','u','p','lam_ode'],\
  ['hes_gamma_x_x'])
```

```
% MATLAB/Octave
dae.add_lc('gamma',{'ode'});
hes = dae.create('hes',...
  {'x','u','p','lam_ode'},...
  {'hes_gamma_x_x'});
```

It is also possible to simply extract the symbolic expressions from the `DaeBuilder` instance and manually create `CasADi` functions. For example, dae.x contains all the expressions corresponding to 'x', dae.ode contains the expressions corresponding to 'ode', etc.

Chapter 8

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:

$$\begin{aligned}
&\text{minimize:} \\
&x(\cdot) \in \mathbb{R}^2,\ u(\cdot) \in \mathbb{R} \qquad \int_{t=0}^{T} (x_0^2 + x_1^2 + u^2)\, dt \\
&\text{subject to:} \\
&\begin{cases} \dot{x}_0 = (1 - x_1^2)\, x_0 - x_1 + u \\ \dot{x}_1 = x_0 \\ -1.0 \le u \le 1.0, \quad x_1 \ge -0.25 \end{cases} \qquad \text{for } 0 \le t \le T \\
&x_0(0) = 0, \quad x_1(0) = 1,
\end{aligned} \tag{8.1}$$

with $T = 10$.

[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

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 collocation* and *direct collocation*.

8.2 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.

8.3 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.

[3]You can obtain this collection as an archive named `examples_pack.zip` in CasADi's download area

8.4 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.

Chapter 9

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:

```
# Python
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)
```

```
% MATLAB/Octave
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)
```

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`.

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)
```

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))    )
```

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:

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

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

You may declare double inequalities:

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

% MATLAB/Octave
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 )
```

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:

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

% MATLAB/Octave
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.

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

% MATLAB/Octave
opti.solver('ipopt');
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.

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

% MATLAB/Octave
opti.set_initial(x, 2);
opti.set_initial(10*x(1), 2)
```

9.2 Problem solving and retrieving

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

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

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

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.

```
sol1 = opti.solve()
opti.set_initial(sol1.value_variables())
sol2 = opti.solve()
```

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

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

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.

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.

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

```
% MATLAB/Octave
sol.value(obj,{y==2})
```

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

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

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

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())
```

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:

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

```
% MATLAB/Octave
opti.callback(@(i) plot(opti.debug.value(x)))
```

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

Chapter 10

Difference in usage from different languages

10.1 General usage

	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 = {'opt1':opt1}` or `d = {}; a['opt1'] = opt1`	`a = Dict(); a['opt1'] = opt1;`	`a = struct; a.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	`z=f(x,y)`	`z = f({x,y})`	`z=f(x,y)`
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.

	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`
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` or `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(x, y)`	`atan2(x, y)`	`atan2(x, y)`
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), ...`
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)`	`fmod(x, y)`	`mod(x, y)`
*Absolute value	`fabs(x)`	`fabs(x)`	`abs(x)`
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)`
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)`
*Matrix multiplication	`mtimes(x,y)`	`mtimes(x,y)`	`mtimes(x,y)` or `x*y`
*Transpose	`transpose(A)` or `A.T`	`transpose(A)` or `A.T()`	`transpose(A)` or `A'` or `A.'`
Inner product	`dot(x, y)`	`dot(x, y)`	`dot(x, y)`
*Horizontal/vertical concatenation	`horzcat(x, y)` `vertcat(x, y)`	`horzcat(v)` `vertcat(v)`, (v vector of matrices)	`[x, y]` `[x; y]`
Horizontal/vertical split (inverse of concatenation)	`vertsplit(x)`, `horzsplit(x)`	`vertsplit(x)`, `horzsplit(x)`	`vertsplit(x)`, `horzsplit(x)`
*Element access	`A[i,j]` and `A[i]`, *0-based*	`A(i,j)` and `A(i)`, *0-based*	`A(i,j)` and `A(i)`, *1-based*
*Element assignment	`A[i,j] = b` and `A[i] = b`, *0-based*	`A(i,j) = b` and `A(i) = b`, *0-based*	`A(i,j) = b` and `A(i) = b`, *1-based*
*Nonzero access	`A.nz[k]`, *0-based*	`A.nz(k)`, *0-based*	(currently unsupported)
*Nonzero assignment	`A.nz[k] = b`, *0-based*	`A.nz(k) = b`, *0-based*	(currently unsupported)
Project to a different sparsity	`project(x, s)`	`project(x, s)`	`project(x, s)`