## 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 DMatrix .......................... 13  
   3.3 The MX symbolics .................. 13  
   3.4 Mixing SX and MX ................. 14  
   3.5 The Sparsity class ............... 15  
   3.6 Getting and setting elements in matrices 15  
      3.6.1 Getting and setting .......... 16  
   3.7 Arithmetic operations ............. 17  
   3.8 Querying properties .............. 18  
   3.9 Linear algebra ................... 19  
   3.10 Calculus ........................ 20  

4 Function objects ..................... 21  
   4.1 Function and derived classes ...... 21  
   4.2 Input-output schemes for function objects 22  
   4.3 Calling function objects .......... 22  
   4.4 Algorithmic differentiation ...... 23  
   4.5 Generating Jacobian and Hessian function objects 24  
   4.6 Converting MXFunction to SXFunction 24  

5 ODE/DAE integration and sensitivity analysis 25  
   5.1 ODE/DAE formulation ............. 25  
   5.2 Sundials integrators .............. 26  
   5.3 Sensitivity analysis .............. 26  

3
5.4 The Simulator class ................................................. 26

6 Nonlinear Programming ........................................... 29

7 Optimal control with CasADi ................................. 31
   7.1 Optimization of a Van der Pol oscillator ................. 31
   7.2 Direct single-shooting ..................................... 32
   7.3 Direct multiple-shooting .................................. 32
   7.4 Direct collocation .......................................... 33

8 Import and reformulation of OCPs from Modelica ....... 35
   8.1 Compiling the Modelica code .............................. 35
   8.2 The SymbolicOCP class .................................... 36

9 Syntax differences between CasADi in Python, C++ and MATLAB (in development) .......................... 37
Chapter 1

Introduction

This document aims to give an introduction to CasADi, an open-source software framework for numerical optimization in general and optimal control (i.e. optimization involving differential equations) in particular. CasADi was developed 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.

After reading this document, the reader 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).

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.

CasADi is written in C++ but also has a full-featured front-end to Python\footnote{In addition, a MATLAB front-end is in development}. 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 say C++ or Python, 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 forum.casadi.org. We read the forum post regularly and try to respond as soon 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 journal paper, and an industrial cooperation involves a negotiated consulting contract. Please contact the 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 Dynamic {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
1.4. READING THIS DOCUMENT

mathematical background knowledge. We assume that the reader already has a fair knowledge of either Python or C++, theory of differential equations and optimization theory.

We will mainly use Python syntax in the discussion, which is the language we recommend using for getting familiar with CasADi, even if the intention is to use another language later. We try to point out the instances where the C++ syntax diverges from the Python syntax. To facilitate switching between the programming languages, we also list the major differences in Chapter 9.
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 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-end to Python is full-featured and auto-generated using SWIG and can be used with little to no loss of efficiency. CasADi has been compiled and used successfully on Linux, OS X and Windows.

For up-to-date installation instructions, we refer to the CasADi’s website, casadi.org.
CHAPTER 2. OBTAINING AND INSTALLING CASADI
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 data type, 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\(^1\). 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) and import CasADi using the command:

```python
from casadi import *
```

Now create the variable `x` using the syntax (cf. the function `sym` in Matlab’s Symbolic Toolbox)\(^3\):

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

which 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. You can also create vector- or matrix-valued symbolic variables by supplying additional arguments to `SX.sym`:

---

\(^1\)Before v1.9.0, *compressed row storage* (CRS) was used.

\(^2\)In C++, include the core header files through "\#include "casadi/casadi.hpp"". Classes and functions are defined in the namespace `casadi`.

\(^3\)In C++, since it is statically typed, you need to specify the return type, here `SX`. To find the return types for a CasADi function, use the C++ API docs on the website
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:
f = x**2 + 10 # "x**y" corresponds to "pow(x,y)" in C++
f = sqrt(f)
print "f = ", f

|f = sqrt((sq(x)+10))|

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.sparse(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:

print "B4 = ", B4

```
B4 =
[[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 a n-by-m symbolic primitive

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

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

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

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

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

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

---

- The text is a chapter discussing the symbolic framework in a programming context.
- It introduces the `SX.sym` function for creating symbolic variables.
- It demonstrates how to create matrices and perform operations like `sqrt` on expressions.
- It also shows how to create constant matrices without symbolic primitives.
- The document emphasizes the difference between structural zeros and actual zeros in printing expressions.
- The list at the end summarizes the common ways to construct symbolic expressions.

---
3.2. DMATRIX

**SX(list)**: Create a column vector (n-by-1 matrix) with the elements in the list, e.g. SX([1, 2, 3])

**SX(list of lists)**: Create a dense matrix with the elements in the lists, e.g. SX([[1, 2], [3, 4]]).

### 3.2 DMATRIX

DMatrix is very similar to SX (in fact, they are just two template instantiations of the same C++ class casadi::Matrix<T>), 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.

DMatrix 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 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 = DMatrix.sparse(2, 3)
C[:,2] = 5
print type(C),
print C
```

```
<class 'casadi.casadi.DMatrix'>
[[0, 0, 5],
 [0, 0, 5]]
```

```python
from numpy import array
C_numpy = array(C)
print type(C_numpy)
print C_numpy
```

```
<type 'numpy.ndarray'>
[[ 0., 0., 5.],
 [ 0., 0., 5.]]
```

```python
from scipy.sparse import csc_matrix
C_scipy = csc_matrix(C)
print type(C_scipy)
print C_scipy
```

```
<class 'scipy.sparse.csc.csc_matrix'>
(0, 2) 5.0
(1, 2) 5.0
```

More usage examples for SX can be found in the tutorials at 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<T>.
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 = ", f
print "f.shape = ", f.shape
```

```
f = 
[[(3*x_0)+y], ((3*x_2)+y)],
[[(3*x_1)+y], ((3*x_3)+y)]
f.shape = (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 elementwise and new expressions (of type SX) were created for each entry of the result matrix.

We shall now introduce a second, more general matrix expression type MX. The MX type allows, like SX, to build up expressions consisting of a sequence of elementary operations. But unlike SX, these elementary operations are not restricted to be scalar unary or binary operations ($\mathbb{R} \rightarrow \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$). Instead the elementary operations that are used to form MX expressions are allowed to be general multiple sparse-matrix valued input, multiple sparse-matrix valued output functions: $\mathbb{R}^{n_1 \times m_1} \times \cdots \times \mathbb{R}^{n_N \times m_N} \rightarrow \mathbb{R}^{p_1 \times q_1} \times \cdots \times \mathbb{R}^{p_M \times q_M}$.

The syntax of MX is intended to mirror that of SX:

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

```
f = ((3*x)+y)
f.shape = (2, 2)
```

Note that this operation required only 2 elementary operations (one multiplication and one addition) using MX symbolics, whereas the SX symbolics required 8 (2 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]
```
3.4. MIXING SX AND MX

\[ |x[0]| \]

The output should be understood as an expression that is equal to the first (i.e. index 0) nonzero 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 (cf. Section 3.6):

\[
\begin{align*}
x &= \text{MX.sym}("x",2) \\
A &= \text{MX.sparse}(2,2) \\
A[0,0] &= x[0] \\
A[1,1] &= x[0]+x[1] \\
\text{print } A
\end{align*}
\]

\[
\left(\text{set_sparse((zeros(2x2, 1 nnz)[0] = x[0]))}[1] = (x[0]+x[1])\right)
\]

The interpretation of the (admittedly cryptic) output is that to a diagonal 2-by-2 matrix with structural zeros off the diagonal and actual zeros on the diagonal ("Const<0>(2x2: diagonal)"), the diagonal entries are set to \( x[0] \) and \( x[0]+x[1] \), respectively.

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 and reshapings.

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 expression. 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 is thus intended to be used for low level operations (for example the DAE right hand side, cf. Chapter 5), 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 elementwise operations, matrix multiplication and transposes to be performed efficiently. In the CCS, 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 reference-counted, meaning that multiple matrices can share the same sparsity pattern, including MX expression graphs and instances of SX and DMatrix. The Sparsity class is also cached, meaning that the creation of multiple instances of the same sparsity patterns are (usually) avoided.

Like the matrix classes of CasADi, the sparsity patterns are mutable objects. This means e.g. that the patterns get updated when assigning a value to a structurally zero location of a matrix.

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.sparse(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))
```

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

### 3.6 Getting and setting elements in matrices

CasADi offers a variety of different ways to get or set specific elements or blocks of elements in matrix types. There are two different categories for this:

**Element access and assignment** Access or set elements of a matrix – which may well be structural zeros – by their (row, col) pairs. This is done with the syntax `A = B[i,j]`, `A[i,j] = B` in Python, and `A = B(i,j), A(i,j) = B` in C++ and MATLAB.

**Nonzero access and assignment** Access or set a particular nonzero of the (sparse) matrices. Entries are ordered by columns using the compressed column storage format, cf. Section 3.5. The syntax for this is `A = B.el[k]`, `A.el[k] = B` in Python, `A = B[k], A[k] = B` in C++ and `A = B{k}, A{k} = B` in MATLAB.

In Python and C++, indexing starts from zero, while in MATLAB indexing starts from one. In Python, negative indices to specify an index counted from the end of a list.

The discussion in the following applies to all matrix types: SX, MX, DMatrix.
3.6. GETTING AND SETTING ELEMENTS IN MATRICES

3.6.1 Getting and setting

Each type of slicing comes with a get or set variant.

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

Unlike 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 in the following example:

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

**Single element access** is getting or setting by providing a $(row, column)$ pair:

```python
M = diag([3,4,5,6])
print M
[[3, 00, 00, 00],
 [00, 4, 00, 00],
 [00, 00, 5, 00],
 [00, 00, 00, 6]]
print M[0,0]
3
print M[1,0]
00
print M[-1,-1]
6
```

Nonzero access works the same way:
print M.nz[3]
|6

print M.nz[-2]
|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 \((\text{start}, \text{stop}, \text{step})\) triple. In Python, *CasADi* uses the standard syntax:

```python
print M[:,1]
| [00, 4, 00, 00]
```

which is equivalent to \(M(:,2)\) in MATLAB and \(M(\text{Slice()},1)\) in C++.

```python
print M[1:,1:4:2]
```

```python
| [[4, 00],
  [00, 00],
  [00, 6]]
```

which is equivalent to \(M(1:\text{end},2:2:5)\) in MATLAB and \(M(\text{Slice}(1,-1),\text{Slice}(1,4,2))\) in C++.

**List access** are similar to (but not as efficient as) slice access:

```python
M = DMatrix([[3,7,8,9],[4,5,6,1]])
```

```python
print M
```

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

```python
print M[0,[0,3]]
```

```python
| [[3, 9]]
```

```python
print M[[1,0],[0,3]]
```

```python
| [[4, 1],
  [3, 9]]
```

```python
print M.nz[[6,1,3]]
```

```python
| [9, 4, 5]
```
3.7 Arithmetic operations

CasADi supports basic such as addition, multiplications, powers, trigonometric functions etc:

```plaintext
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)]
```

In C++ and Python, the standard multiplication operation (using *) is reserved for elementwise multiplication. For matrix multiplication, use `mul(A,B)`:

```plaintext
print mul(y,y)
```

```
[[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))]]
```

**Transposes** are formed using the syntax `A.T` in Python or `A.T()` in C++:

```plaintext
print y.T
```

```
[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 cheap operation which is performed using the syntax:

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

```
[[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 efficient to stack matrices horizontally. Matrices that are in fact 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`, respectively:

```plaintext
x = SX.sym("x",5,1)
y = SX.sym("y",5,1)
print vertcat((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_0, y_0],
 [x_1, y_1],
 [x_2, y_2],
 [x_3, y_3],
 [x_4, y_4]]

**Horizontal and vertical splitting** 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 splitted, a vector \( \text{offset} \) of length \( n + 1 \). The first element of the \( \text{offset} \) 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 splitting operation then contains the columns \( c \) with \( \text{offset}[i] \leq c < \text{offset}[i + 1] \). The following demonstrates the syntax:

```python
x = SX.sym("x",5,2)
offset = [0,1,2]
x1,x2 = horzsplit(x,offset)
print "x1 = ", x1, " , x2 = ", x2

| x1 = [x_0, x_1, x_2, x_3, x_4] , x2 = [x_5, x_6, x_7, x_8, x_9] |
```

The vertsplit operation works analogously, but with the \( \text{offset} \) vector referring to rows:

```python
y = SX.sym("y",10,1)
offset = [0,5,10]
y1,y2 = vertsplit(y,offset)
print "y1 = ", y1, " , y2 = ", y2

| y1 = [y_0, y_1, y_2, y_3, y_4] , y2 = [y_5, y_6, y_7, y_8, y_9] |
```

It is always possible to use slice element access (cf. Section \[3.6\]) instead of horizontal and vertical splitting:

```python
y = SX.sym("y",10,1)
offset = [0,5,10]
y1,y2 = [y[offset[i]:offset[i+1],:] for i in range(len(offset)-1)]
print "y1 = ", y1, " , y2 = ", y2

| y1 = [y_0, y_1, y_2, y_3, y_4] , y2 = [y_5, y_6, y_7, y_8, y_9] |
```

For SX graphs, this alternative way is completely equivalent, but for MX graphs using `horzsplit/vertsplit` is **significantly more efficient** and should be preferred when possible.

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

```python
x = SX.sym("x",2,2)
print inner_prod(x,x)

| (((sq(x_0)+sq(x_1))+sq(x_2))+sq(x_3)) |
3.8. QUERYING PROPERTIES

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).

3.8 Querying properties

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

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

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

- $A\text{.size1()}$ The number of rows
- $A\text{.size2()}$ The number of columns
- $A\text{.shape}$ The shape, i.e. the pair $(nrow,ncol)$
- $A\text{.numel()}$ The number of elements, i.e. $nrow \ast ncol$
- $A\text{.size()}$ The number of structurally nonzero elements, equal to $A\text{.numel()}$ if dense.
- $A\text{.sparsity()}$ Retrieve a reference to the sparsity pattern
- $A\text{.isDense()}$ Is a matrix dense, i.e. having no structural zeros
- $A\text{.isScalar()}$ Is the matrix a scalar, i.e. having dimensions 1-by-1?
- $A\text{.isVector()}$ Is the matrix a vector, i.e. having dimensions $n$-by-1?
- $A\text{.isSquare()}$ Is the matrix square?
- $A\text{.isTriu()}$ Is the matrix upper triangular?
- $A\text{.isConstant()}$ Are the matrix entries all constant?
- $A\text{.isInteger()}$ 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 $A\text{.isConstant()}$ is true is guaranteed to be constant, but is not guaranteed to be non-constant if $A\text{.isConstant()}$ is false. We recommend you to check the API documentation for a particular function before using it for the first time.
3.9 Linear algebra

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

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

3.10 Calculus

Basic calculus operations are also possibly and implemented very efficiently using *algorithmic differentiation* (AD) as outlined in Chapter 4.

```python
A = SX.sym("A", 3, 3)
x = SX.sym("x", 3)
print jacobian(mul(A, x), x)
```

```plaintext
[[A_0, A_3, A_6],
 [A_1, A_4, A_7],
 [A_2, A_5, A_8]]
```
Chapter 4

Function objects

4.1 Function and derived classes

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

The basic usage skeleton of all these function objects are:

\[
\begin{align*}
& f = \text{ClassName}(\text{arguments}) \quad \# \text{Call the constructor} \\
& f.\text{setOption}("\text{option}_\text{name}",\text{option}_\text{value}) \quad \# \text{Set options} \\
& f.\text{init}() \quad \# \text{Initialize the function} \\
& f.\text{setInput}(\text{value},\text{input}_\text{index}) \quad \# \text{Pass inputs} \\
& f.\text{evaluate}() \quad \# \text{Evaluate numerically} \\
& \ldots = f.\text{getOutput}(\text{output}_\text{index}) \quad \# \text{Get outputs}
\end{align*}
\]

Note that all functions are multiple (sparse, matrix-valued) input, multiple (sparse, matrix-valued) output.

One important member of the Function derived classes is SXFunction, which is constructed with the syntax SXFunction(list of inputs,list of outputs):

\[
\begin{align*}
& x = \text{SX.sym}(\"x\",2,2) \\
& y = \text{SX.sym}(\"y\") \\
& f = \text{SXFunction}([x,y],[x,\sin(y)*x])
\end{align*}
\]

which defines a function \( f : \mathbb{R}^{2 \times 2} \times \mathbb{R}, (x,y) \mapsto (x, \sin(y)x). \)

MXFunction works the same way, but for the MX expression graphs:

\[
\begin{align*}
& x = \text{MX.sym}(\"x\",2,2) \\
& y = \text{MX.sym}(\"y\") \\
& f = \text{MXFunction}([x,y],[x,\sin(y)*x])
\end{align*}
\]

\footnote{In C++, getOutput is return-by-reference}
All derived classes of Function come with a range of options that can be set. The API documentation shows the list of available options for each class, along with their default values and descriptions.

4.2 Input-output schemes for function objects

Most Function-derived classes have a particular input/output signature with them and/or take as arguments in the constructor other function objects (callback functions) with a particular input/output signature. For example, as we shall see in Chapter 6 a nonlinear program (NLP) is defined by two inputs (the decision variable and a set of parameters) and two outputs (the objective and constraint functions). To make it easier to remember the number and order of the input/output arguments, CasADi provides helper functions of the form:

\[
x = \text{SX}\text{.sym}("x")
p = \text{SX}\text{.sym}("p")
f = x^2x\]
\[g = \log(x) - p\]
\[\text{nlp} = \text{SX}\text{Function}(\text{nlpIn}(x=x, p=p), \text{nlpOut}(f=f, g=g))\]

The last line is essentially equivalent to \[\text{nlp} = \text{SX}\text{Function}([x, p], [f, g])\], but 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
- The helper functions provide meta-information that allows inputs to be referred to by their names instead of their index. E.g. \[\text{nlp.jacobian("x","g")}\] instead of \[\text{nlp.jacobian(0,1)}\].

4.3 Calling function objects

MX expressions may contain calls to Function-derived functions. To embed a call to function (e.g. SXFunction or Integrator, see Chapter 5), you use the syntax:

\[[y_1, y_2, \ldots, y_n] = f([x_1, x_2, \ldots, x_n])\]

That is, you provide a list of argument expressions (which may consist of a single element) and gets back a list of expressions corresponding to the result of the call. Here is a code example:
4.4. ALGORITHMIC DIFFERENTIATION

\[ A = \text{SX.sym("A",}3,3) \]
\[ x = \text{SX.sym("x",}3) \]
\[ y = \text{mul}(A,x) \]
\[ f = \text{SXFunction}([A,x],[y]) \]
\[ f.\text{init()} \] # Must be called before symbolic or numeric evaluation

\[ A2 = \text{MX.sym("A",}3,3) \]
\[ x2 = \text{MX.sym("x",}3) \]
\[ [y2] = f([A2,x2]) \] # Call f with argument (A2,x2)

Remember to put the brackets around the output even if the function only has a single output.

4.4 Algorithmic differentiation

The single most central functionality of CasADi is algorithmic (or automatic) differentiation (AD).

Assume a function \( \mathbb{R}^N \rightarrow \mathbb{R}^M \):
\[ y = f(x) \] (4.1)

Forward mode directional derivatives can be used to calculate Jacobian-times-vector products:
\[ y_{\text{sens}} = \frac{\partial f}{\partial x} x_{\text{seed}} \] (4.2)

We will refer to the multiplying vector as the seed vector and the result as the sensitivity vector.

Similarly, reverse mode (or adjoint mode) directional derivatives can be used to calculate Jacobian-transposed-times-vector products:
\[ x_{\text{asens}} = \left( \frac{\partial f}{\partial x} \right)^T y_{\text{aseed}} \] (4.3)

Both forward and adjoint 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. Internally, the algorithm it will use for this depends on the particular Function class but in most cases 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.

4.5 Generating Jacobian and Hessian function objects

You can generate a new function for calculating the Jacobian by calling the `Function::jacobian` member function:

\[
J = f.jacobian(iind,oind) \quad \text{# function corresponding to the Jacobian}
\]
\[
\text{# of the oind-th output w.r.t. the iind-th input}
\]

This will generate a function with the same input scheme as the function \( f \) but with the (sparse) Jacobian as well as all the original function outputs as outputs.

Similarly, the gradient of a scalar output can be called by calling the `Function::gradient` member function:

\[
G = f.gradient(iind,oind) \quad \text{# function corresponding to the gradient}
\]
\[
\text{# of the oind-th output w.r.t. the iind-th input}
\]

which will generate a function with the same input scheme as the function \( f \) but with the (sparse) gradient and all the original function outputs as outputs.

Lastly, Hessians of a scalar output can be called by calling the `Function::hessian` member function:

\[
H = f.hessian(iind,oind) \quad \text{# function corresponding to the Hessian}
\]
\[
\text{# of the oind-th output w.r.t. the iind-th input}
\]

which will generate a function with the same input scheme as the function \( f \) but with the (sparse) Hessian, the gradient and all the original function outputs as outputs.

4.6 Converting MXFunction to SXFunction

An `MXFunction` which only contains built-in operations (e.g. elementwise operations such as addition, square root, matrix multiplications and calls to `SXFunction` instances, can be converted into an `SXFunction` using the syntax:

\[
sx_function = SXFunction(mx_function)
\]

This might speed up the calculations significantly, but might also cause extra memory overhead.
Chapter 5

ODE/DAE integration and sensitivity analysis

5.1 ODE/DAE formulation

The integrators interfaced with CasADi assumes a DAE residual function of semi-explicit form with quadratures:

\[
\dot{x} = f_{\text{ode}}(t, x, z, p), \quad x(0) = x_0 \quad (5.1a)
\]

\[
0 = f_{\text{alg}}(t, x, z, p) \quad (5.1b)
\]

\[
\dot{q} = f_{\text{quad}}(t, x, z, p), \quad q(0) = 0 \quad (5.1c)
\]

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

DAEs of form can be formed using the \texttt{daeIn} and \texttt{daeOut} helper functions:

\begin{verbatim}
x = SX.sym("x")
z = SX.sym("z")
f = -z*x
g = z*z-2
dae = SXFunction(daeIn(x=x,z=z),daeOut(ode=f,alg=g))
print dae
\end{verbatim}

Input (DAEInput: 4):
0. (DAE_X aka 'x') 1-by-1 (dense)
1. (DAE_Z aka 'z') 1-by-1 (dense)
2. (DAE_P aka 'p') 0-by-0 (dense)
3. (DAE_T aka 't') 0-by-0 (dense)

Outputs (DAEOutput: 3):
0. (DAE_ODE aka 'ode') 1-by-1 (dense)
1. (DAE_ALG aka 'alg') 1-by-1 (dense)
2. (DAE_QUAD aka 'quad') 0-by-0 (dense)

Function not initialized
An integrator in CasADi is a function that takes the state at the initial time, a set of parameters, the state vector at the initial time and a guess for the algebraic variables (only for DAEs) and returns the state vector and algebraic variables at the final time.

\[
\text{integrator} = \text{Integrator}("solver\_name", \text{dae})
\]

The time horizon is assumed to be fixed\(^1\) and can be set with the option:

\[
\text{integrator.setOption\("tf\", \text{integration\_end\_time})
\]

### 5.2 Sundials integrators

The Sundials suite contains the two popular integrators CVodes and IDAS for ODEs and DAEs respectively. These two integrators support forward and adjoint sensitivities 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. This means that the only information that the user needs to provide is the DAE residual function:

\[
\text{integrator} = \text{Integrator}("cvodes", \text{dae}) \quad \# \text{ For ODEs} \\
\text{integrator} = \text{Integrator}("idas", \text{dae}) \quad \# \text{ For DAEs}
\]

for CVodes and IDAS respectively.

For a list of options for the integrators, as well as the input and output schemes of this function, check the documentation directly from Python:

\[
\text{print Integrator.doc("cvodes")}
\]

or by consulting the online C++ API docs on the website.

### 5.3 Sensitivity analysis

From evaluation point of view, an integrator behaves just like the SXFunction introduced in the previous session. You set inputs, forward/adjoint seeds, evaluate and obtain the outputs and forward/adjoint sensitivities.

\(^1\) 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
5.4 The Simulator class

As already mentioned, integrators in CasADi are functions that calculates the state at the final time. Often, however, a user is interested in obtaining the solution at multiple time points. This can often be done more efficiently than by repeatedly calling `integrator.evaluate()`. The easiest way to use this functionality is to use the Simulator class.

A Simulator can be created using the syntax:

```python
# Import numpy
import numpy as NP

# Allocate an integrator instance
integrator = ...
integrator.setOption("...", ...)

# Choose a time grid
tgrid = NP.linspace(0, end_time, num_time_steps)

# Create a simulator
simulator = Simulator(integrator, time_grid)
```

A Simulator can be used just like an integrator, and its input scheme is the same. Its output is now matrix valued, with the columns corresponding to different time points. The class can also be used to evaluate a particular function of the state at a set of time points. See the API documentation for more information.
Chapter 6

Nonlinear Programming

The NLP solvers contained in or interfaced with CasADi solves parametric NLPs of the following form:

\[
\begin{align*}
\text{minimize:} & \quad f(x, p) \\
\text{subject to:} & \quad x_{\min} \leq x \leq x_{\max} \\
& \quad g_{\min} \leq g(x, p) \leq g_{\max}
\end{align*}
\]

(6.1)

With a function \text{nlp}, which given \text{x} and \text{p}, gives \text{f} and \text{g}, an NLP solver instance can be allocated by:

\[
nlp\_solver = \text{NlpSolver("solver\_name",nlp)}
\]

The interface will then automatically generate the information that it might need to solve the NLP, which may be solver and option dependent. Typically an NLP solver will need a function that gives the Jacobian of the constraint function and a Hessian of the Lagrangian function \(L(x, \lambda) = f(x) + \lambda^T g(x)\) with respect to \text{x}. The interface will generate this kind of information automatically and provide to the solver.

NLP solvers, like ODE/DAE integrators are functions in CasADi (though taking derivatives of them is currently not supported). You will find the input and output schemes in the CasADi API documentation on the website or by using the question mark in Python.
Chapter 7

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\textsuperscript{1} or Betts\textsuperscript{2} or Moritz Diehl’s lecture notes on numerical optimal control.

7.1 Optimization of a Van der Pol oscillator

In CasADi’s examples collection\textsuperscript{3} you find codes for solving simple optimal control problem using a variety of different methods. The problem is driving a Van der Pol oscillator to

\textsuperscript{1}Lorenz T. Biegler, \textit{Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes}, SIAM 2010
\textsuperscript{3}You can obtain this collection as an archive named \texttt{examples.pack.zip} in CasADi’s download area
the origin, while trying to minimize a quadratic cost:

\[
\begin{align*}
\text{minimize:} & \quad \int_{t=0}^{10} (x_0^2 + x_1^2 + u^2) \, dt \\
\text{subject to:} & \quad \begin{cases}
\dot{x}_0 = (1 - x_1^2)x_0 - x_1 + u \\
\dot{x}_1 = x_0 \\
-0.75 \leq u \leq 1.0 \\
x_0(0) = 0, \quad x_0(10) = 0 \\
x_1(0) = 1, \quad x_1(10) = 0
\end{cases}
\end{align*}
\]

(7.1)

7.2 Direct single-shooting

The simplest OCP method to implement in CasADi is single-shooting. In fact, it can be implemented in just 30 lines of code, which the script `vdp_single_shooting.py` in CasADi’s examples collection demonstrates. Please go through the code carefully and make sure that you understand the principle. The most important part of the code are the lines:

```
# Build a graph of integrator calls
for k in range(nk):
    X, = integratorOut(integrator(integratorIn(x0=X,p=U[k])),"xf")
```

Here, we recursively construct a symbolic expression for the state at the final time. This expression is then used in the formulation of the NLP objective and constraint functions.

7.3 Direct multiple-shooting

The second optimal control method we shall consider is multiple shooting. In multiple shooting, the states at each ”shooting node” are degrees of freedom in the NLP. The script `vdp_multiple_shooting.py` demonstrates how the VDP problem can be solved using direct multiple shooting in CasADi.

Note how the symbolic variable in the NLP functions now contains both the control and states for each of the \( nk \) intervals:

```
# Total number of variables
nv = 1*nk + 3*(nk+1)

# Declare variable vector
V = MX.sym("V", nv)
```

Also note how the recursive overwriting of \( X \) has been replaced with:
for k in range(nk):
    # Local state vector
    Xk = vertcat((X0[k], X1[k], X2[k]))
    Xk_next = vertcat((X0[k+1], X1[k+1], X2[k+1]))

    # Call the integrator
    Xk_end, = integratorOut(integrator(integratorIn(x0=Xk, p=U[k]), "xf")

    # append continuity constraints
    g.append(Xk_next - Xk_end)
    g_min.append(NP.zeros(Xk.size()))
    g_max.append(NP.zeros(Xk.size()))

7.4 Direct collocation

When we went from direct single shooting to direct multiple shooting we essentially traded nonlinearity for problem size. The NLP in single shooting is small, but often highly nonlinear, whereas the NLP for multiple-shooting is larger, but less nonlinear and with a sparsity structure that can be exploited efficiently. Direct collocation is to take one more step in the same direction, adding even more degrees of freedom. The resulting NLP is even larger, but has even more structure that can be exploited.

While multiple shooting only includes the state at the beginning of each control interval as a degree of freedom in the NLP (in addition to the discretized control and the parameters), in direct collocation the state at a set of collocation points (in addition to the beginning of the interval) enters in the NLP as variables. An example of a choice of time points are the Legendre points of order \( d = 3 \):

\[
\tau = [0, 0.112702, 0.500000, 0.887298]
\] (7.2)

Keeping the same control discretization as in single and multiple shooting:

\[
u(t) = u_k, \quad \text{for } t \in [t_k, t_{k+1}], \quad k = 0, \ldots, n_k - 1
\] (7.3)

the complete list of time points, with \( h_k := t_{k+1} - t_k \), is:

\[
t_{k,j} := t_k + h_k \tau_j, \quad \text{for } k = 0, \ldots, n_k - 1 \text{ and } j = 0, \ldots, d
\] (7.4)

as well as the final time \( t_{n_k,0} \). Also let \( x_{k,j} \) denote the states at these time points.

On each control interval, we shall define a Lagrangian polynomial basis:

\[
L_j(\tau) = \prod_{r=0, r\neq j}^d \frac{\tau - \tau_r}{\tau_j - \tau_r}
\] (7.5)
Since the Lagrangian basis satisfies:

\[
L_j(\tau_r) = \begin{cases} 
1, & \text{if } j = r \\
0, & \text{otherwise}
\end{cases} \quad (7.6)
\]

we can approximate the state trajectory approximation as a linear combination of these basis functions:

\[
\tilde{x}_k(t) = \sum_{r=0}^{d} L_r \left( \frac{t - t_k}{h_k} \right) x_{k,r} \quad (7.7)
\]

In particular we get approximations of the state time derivative at each collocation point (not including \(\tau_0\)):

\[
\tilde{\dot{x}}_k(t_{k,j}) = \frac{1}{h_k} \sum_{r=0}^{d} \dot{L}_r(\tau_j) x_{k,r} := \frac{1}{h_k} \sum_{r=0}^{d} C_{r,j} x_{k,r} \quad (7.8)
\]

as well as an approximation of the state at the end of the control interval:

\[
\tilde{x}_{k+1,0} = \sum_{r=0}^{d} L_r(1) x_{k,r} := \sum_{r=0}^{d} D_r x_{k,r} \quad (7.9)
\]

Plugging in the approximation of the state derivative (7.8) into the ODE gives us a set of collocation equations that needs to be satisfied for every state at every collocation point:

\[
 h_k f(t_{k,j}, x_{k,j}, u_k) - \sum_{r=0}^{d} C_{r,j} x_{k,r} = 0, \quad k = 0, \ldots, n_k - 1, \quad j = 1, \ldots, d \quad (7.10)
\]

And the approximation of the end state (7.9) gives us a set of continuity equations that must be satisfied for every control interval:

\[
x_{k+1,0} - \sum_{r=0}^{d} D_r x_{k,r} = 0, \quad k = 0, \ldots, n_k - 1, \quad (7.11)
\]

These two sets of equations take the place of the continuity equation (represented by the integrator call) in direct multiple shooting. The above equations have been implemented in the script vdp_collocation.py.
Chapter 8

Import and reformulation of OCPs from Modelica

CasADi supports importing OCPs formulated in the Modelica modeling language as well as in the Optimica extension of Modelica to optimal control. The import facility currently works with the open-source JModelica.org compiler.

For instructions on how to install and use JModelica.org, we refer to their website. Note that a CasADi build (which may not be the latest) is contained in the pre-built JModelica.org binaries.

NOTE: The functionality to import and reformulation of OCPs from Modelica is still under development and to be considered unstable.

8.1 Compiling the Modelica code

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

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

```python
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.

```python
from zipfile import ZipFile
sfile = ZipFile(jmu_name', 'r')
mfile = sfile.extract('modelDescription.xml', '.')
```
8.2 The SymbolicOCP class

The symbolic optimal control problem representation (or just model description) contained in `modelDescription.xml` can be imported into CasADi in the form of the SymbolicOCP class:

```python
ocp = SymbolicOCP()
```

The `SymbolicOCP` class contains symbolic representation of the optimal control problem designed to be general and allow manipulation. The basic usage of this class is to call the member function `parseFMI` with the XML file name:

```python
ocp.parseFMI('modelDescription.xml')
```

Symbolic expressions corresponding to the states, controls parameters etc. are directly accessible as member variables of the `SymbolicOCP` object. E.g. the free controls can be accessed as `ocp.u`. Symbolic expressions for the OCP are either accessible in the same way (e.g. `ocp.dae` will return the equations for the fully-implicit differential-algebraic equation formulation used by Modelica. Other equations are tied to a particular variable. E.g. the `binding equations` corresponding to the set of `independent parameters` can be retrieved as `ocp.beq(ocp.pi)`.

Other useful commands available for an instance `ocp` of `SymbolicOCP` include:

```python
print ocp  # Print the optimal optimal control problem to screen
ocp.makeExplicit()  # Reformulate the OCP in fully-explicit form
ocp.scaleVariable()  # Scale all variables using the `nominal` attribute for each variable
ocp.eliminateIndependentParameters()  # Eliminate all independent parameters from the symbolic expressions
```

For a more detailed description of this class and its functionalities, we refer to the API documentation. We also recommend potential users to have a look at the `modelica_ms.py` tutorial example.
Chapter 9

Syntax differences between CasADi in Python, C++ and MATLAB (in development)

<table>
<thead>
<tr>
<th>Python</th>
<th>C++</th>
<th>MATLAB (in development)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Starting CasADi</strong></td>
<td>from casadi import *</td>
<td>#include &quot;casadi/casadi.hpp&quot; using namespace casadi;</td>
</tr>
<tr>
<td><strong>Printing the representation</strong>&lt;br&gt; (string representation intended to be unambiguous)</td>
<td>A &lt;ENTER&gt; (interactive), print repr(A) (in scripts)</td>
<td>std::cout &lt;&lt; A;</td>
</tr>
<tr>
<td><strong>Printing the description</strong>&lt;br&gt; (string representation intended to be readable)</td>
<td>print A</td>
<td>A.print(); or A.print(stream);</td>
</tr>
<tr>
<td><strong>Calling a class function</strong></td>
<td>SX.zeros(3,4)</td>
<td>SX::zeros(3,4)</td>
</tr>
<tr>
<td><strong>Element access</strong></td>
<td>A[i,j], 0-based</td>
<td>A(i,j) and A(i), 0-based</td>
</tr>
<tr>
<td><strong>Element assignment</strong></td>
<td>A[i,j] = b, 0-based</td>
<td>A(i,j) = b and A(i) = b, 0-based</td>
</tr>
<tr>
<td><strong>Nonzero access</strong></td>
<td>A.el[k], 0-based</td>
<td>A[k], 0-based</td>
</tr>
<tr>
<td><strong>Nonzero assignment</strong></td>
<td>A.el[k] = b, 0-based</td>
<td>A[k] = b, 0-based</td>
</tr>
<tr>
<td><strong>Matrix multiplication</strong></td>
<td>mul(x,y)</td>
<td>mul(x,y)</td>
</tr>
<tr>
<td><strong>Elementwise multiplication</strong></td>
<td>x*y</td>
<td>x*y</td>
</tr>
<tr>
<td><strong>Elementwise division</strong></td>
<td>x/y</td>
<td>x/y</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>x**y</td>
<td>pow(x,y)</td>
</tr>
<tr>
<td><strong>Transpose</strong></td>
<td>transpose(A) or A.T</td>
<td>transpose(A) or A.T()</td>
</tr>
<tr>
<td><strong>Creating a dictionary (e.g. for options)</strong></td>
<td>d = {'opt1': opt1} or d = {}; a['opt1'] = opt1</td>
<td>a = Dictionary(); a['opt1'] = opt1;</td>
</tr>
<tr>
<td><strong>Getting function output by value</strong></td>
<td>out = f.getOutput()</td>
<td>DMatrix out; f.setOutput(out);</td>
</tr>
</tbody>
</table>