CasADi tutorials, 23-24 November 2011

Exercise 1: Getting started with CasADi

Joel Andersson

Optimization in Engineering Center (OPTEC) and ESAT-SCD, K.U. Leuven,
Kasteelpark Arenberg 10, B-3001 Leuven, Belgium

December 2, 2011

1 CasADi structure

The backbone of CasADi is centered around some fundamental classes:

- `SX` – *scalar symbolic type*
- `SXMatrix` – *sparse matrix with elements of type* ***`SX`***
- `DMatrix` – *same as* ***`SXMatrix`*** *but with elements of floating point type*
- `FX` and derived classes – *functions*
- `MX` – *matrix symbolic type*

This session covers the first four classes in the list. The `MX` class will be presented in separate session tomorrow.

1.1 `SX` and `SXMatrix`

The `SX` type is used to represent symbolic expressions made up by a sequence of unary and binary operations. It uses a syntax similar to a numeric type such as `float` in Python (which corresponds to `double` in C/C++), but instead of calculating the expressions numerically, the `SX` will build up an expression. The type overloads most common operations and can also be used instead of Python's built-in types in container classes such as `numpy.array`.

Though it is possible and somtimes benificial, we shall not work with the `SX` type directly in this tutorial. Instead we shall work with a *sparse matrix type* called `SXMatrix`, whose elements are `SX` instances. `SXMatrix` uses an everything-is-a-matrix type syntax that should be familiar to Matlab users, which means that scalars can be represented as 1-by-1 matrices and vectors as n-by-1 matrices. To store its element, it uses a general sparse matrix format comparible to that used for sparse matrices in Matlab[1].

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:

```
from casadi import *
```

[1]To be more precise, the storage format used is *compressed row storage*

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

```
x = ssym("x")
```

Note that the string passed is only the display name, not the identifier. Multiple variables can have the same name, but still be different. You can also create vectors or matrices of variables using by supplying additional arguments to `ssym`:

```
y = ssym("y",5)   # A 5-by-1 matrix, i.e. a vector, with symbolic variables
Z = ssym("Z",4,2) # A 4-by-2 matrix with symbolic variables
```

Note that `ssym` is a function which returns an `SXMatrix` instance. When variables have been declared, expressions can now be formed in an intuitive way:

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

You can also create `SXMatrix` instances *without* any symbolic variables:

```
B1 = SXMatrix.zeros(4,5)  # a dense 4-by-5 empty matrix with all zeros
B2 = SXMatrix.sparse(4,5) # a sparse 4-by-5 empty matrix with all zeros
B3 = SXMatrix.ones(4,5)   # a dense 4-by-5 matrix filled with ones
B4 = SXMatrix.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.

Elements are accessed and set using a bracket syntax, which also updates the sparsity pattern. Note that in C++ and Python (and thus also in CasADi) indices start with zero:

```
print B1[0,0]   # print elements [0,0] (Python is zero-based!)
print B2[-1,-1] # print the last element of the last row
print B3[:,2]   # print all elements in the 2nd column
B4[2:4,1]=[7,3] # set [2,1] to 7 and [3,1] to 3
```

Remember to pass two arguments if you wish to access a matrix entry. If you call the same function using *one* index, is interpreted as accesssing the (structural) nonzeros of the matrix:

```
B4[:] = 5   # set all nonzeros to 5
print B4[6] # error: there are only six nonzeros!
```

There is a growing set of operations that can be performed with the `SXMatrix`, including:

- Calculus: E.g. `Jx = jacobian(f,x)`: Jacobian of expression `f` with respect to expression `x`, see also the `FX` member *function* `jacobian` below.
- Algebra: E.g. `x = solve(A,b)`: Can be used to give a symbolic expression for $x = A^{-1}\, b$.

It is typically uncomplicated to add new such functions whenever they appear in applications.

1.2 DMatrix

DMatrix is very similar to SXMatrix (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 SX expressions. Except for the function ssym above (which

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:

```
C = 4*DMatrix.ones(2,3)
C[:,2] = 5
C_numpy = NP.array(C)

import scipy as SP
C_scipy = SP.sparse.csr_matrix(C)
```

More usage examples for SX can be found in the tutorials at www.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>.

1.3 FX and derived classes

CasADi contains a number of functions that all derive from the FX base class. This includes functions that are defined by a symbolic expression, ODE/DAE integrators, QP solvers, NLP solvers etc.

The usage skeleton of all these functions are:

```
# Call the constructor
f = ClassName(arguments)

# Set options
f.setOption("option_name",option_value)

# Initialize the function
f.init()

# Set inputs, forward and adjoint derivative seeds
f.setInput(value,input_index)
f.setFwdSeed(value,input_index)
f.setAdjSeed(value,output_index)

# Evaluate
f.evaluate(num_fwd_sensitivities, num_adj_sensitivities)

# Get outputs, forward and adjoint derivative sensitivities
f.getOutput(value,output_index)
f.getFwdSens(value,output_index)
f.getAdjSens(value,input_index)
```

As an alternative to calling getOutput, setFwdSeed etc., you can also directly access the internal matrices (which are of type DMatrix). This can decrease overhead and shorten the

code, but must be used with caution, since changing the *structure* of these matrices can cause CasADi to crash.

```
print "output is ", f.output()
print "forward seed is ", f.fwdSeed()
```

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

One of the `FX` derived classes is `SXFunction`, which defines a function given a symbolic expression. Its constructor is:

```
f = SXFunction(list_of_inputs,list_of_outputs)
```

2 Automatic differentiation

The most important functionality of CasADi is *automatic differentiation* or AD. CasADi contains a total of 8 different AD algorithms that are suitable in different situations.

Given a function $\mathbf{R}^N \rightarrow \mathbf{R}^M$:

$$y = f(x) \tag{1}$$

forward directional derivatives (Jacobian-times-vector products) can be calculated by giving a *seed* in that direction:

$$y_{\text{fsens}} = \frac{\partial f}{\partial x}\, x_{\text{fseed}} \tag{2}$$

Jacobian-transposed-times-vector products can be calculated in a similar way, using adjoint mode AD:

$$x_{\text{asens}} = \left(\frac{\partial f}{\partial x}\right)^{\text{T}} y_{\text{aseed}} \tag{3}$$

In CasADi, forward/adjoint seeds are set together with the function inputs and forward/adjoint sensitivities (or directional derivatives) are collected together with the outputs. Multiple forward and/or adjoint derivative directions can be handled simultaneously, that is you can simultaneously multiply the Jacobian from the left or right with multiple vectors. The number of vectors are passed as arguemnts to `f.evaluate(num_fwd_sensitivities, num_adj_sensitivities)`. In this turorial we shall restrict ourselves to at most one forward and one adjoint derivative direction, meaning that `num_fwd_sensitivities` and `num_adj_sensitivities` will either be 0 or 1.

CasADi is also able to generate complete, *sparse* Jacobians efficiently by calling the `jacobian` member function. The algorithm it will use for this depends on the particular class[2].

```
Jf = f.jacobian(0,0) # function corresponding to the jacobian
                     # of the 0-th output w.r.t. the 0-th input
```

3 Exercises

1.1 A simple model for the flight of a rocket is the following differential equation:

$$\begin{aligned} \dot{h} &= v \\ \dot{v} &= (u - \alpha\, v^2)/m \\ \dot{m} &= -\beta\, u^2 \end{aligned} \tag{4}$$

[2]For *SXFunction*, the default algorithm is to first determine the sparsity pattern, then use a graph coloring algorithm to determine which rows or columns can be calculated simultaneously, performing the AD algorithm symbolically and then assemble a new *SXFunction* corresponding the the Jacobian

where h is height, v is velocity, m is mass and u is the thrust. α is the friction coefficient taken to be 0.05 and β is the fuel consumption rate, here taken to be 0.1. Create an `SXFunction` f which calculates $\dot{x}$ given x and u where $x := [h, v, m]^{\mathrm{T}}$, i.e. $\dot{x} = f(x, u)$. Evaluate the function numerically with $x = [0.2, 0.3, 1.0]^{\mathrm{T}}$ and $u = 0.4$. Check the results.

1.2 For the same numerical values of x and u, calculate forward sensitivities using the forward seeds $x_{\mathrm{fseed}} = [0, 1, 0]^{\mathrm{T}}$, $u_{\mathrm{fseed}} = 0.0$. How do you interpret the result?

1.3 For the same numerical values of x and u, calculate adjoint sensitivities using the adjoint seeds $\dot{x}_{\mathrm{aseed}} = [0, 0, 1]^{\mathrm{T}}$. How do you interpret the result?

1.4 Create a function $J_x = \frac{\partial f}{\partial x}(x, u)$ using the syntax `f.jacobian()` where `f` is the ODE right hand side function. Evaluate the function with the above values and compare the results with the already calculated directional derivatives. Do they agree?

1.5 Get a *symbolic expression* for $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial u}$ using the syntax: `J = jacobian(0,0)` and `J = jacobian(1,0)`. How many nonzero entries do the Jacobians contain?