CasADi tutorial 2

```
#
#
#
#
#
#
#
```

This tutorial file explains the use of CasADi's Function in a python context. We assume you have read trough the SX tutorial.

Introduction

Let's start with creating a simple expression tree z.

```
from casadi import *
from numpy import *
x = SX.sym("x")
y = x**2
z = sin(y) + y
print z
```

```
@1=sq(x), (sin(@1)+@1)
```

The printout value of z may give you the false impression that the evaluation of z will involve two multiplications of x. This is not the case. This is what's going on under the hood:

The expression tree of z does not contain two subexpressions x*x, rather it contains two pointers to a signle subexpression x*x. In fact, in the C++ implementation, an S object is really not more than a collection of pointers. It are SXNode objects which really contain the data associated with subexpressions.

CasADi generates SXnodes at a very fine-grained level. Even 'sin(y)' is an SXNode, even though we have not ourselves declared a variable to point to it.

When evaluating z for a particular numerical value of x, the product is computed only once.

Functions

CasADi's Function has powerful input/output behaviour. The following input/output primitives are supported: A function that uses one primitive as input/output is said to be 'single input'/'single output'.

In general, an SXfunction can map from-and-to list/tuples of these primitives.

Functions with scalar valued input

The following code creates and evaluates a single input (scalar valued), single output (scalar valued) function.

```
f = Function('f', [x], [z]) # z = f(x)

print "%d -> %d" % (f.n_in(),f.n_out())
```

```
1 -> 1
```

```
f_in = f.sx_in()
print f_in, type(f_in)
```

```
[SX(x)] <type 'list'>
```

```
f_out = f(*f_in)
print f_out, type(f_out)
```

```
@1=sq(x), (sin(@1)+@1) <class 'casadi.casadi.SX'>
```

```
z0 = f(2)
print z0
```

```
3.2432
```

```
print type(z0)
```

```
<class 'casadi.casadi.DM'>
```

```
z0 = f(3)
print z0
```

```
9.41212
```

We can evaluate symbolically, too:

```
print f(y)
```

```
@1=sq(sq(x)), (sin(@1)+@1)
```

Since numbers get cast to SXConstant object, you can also write the following non-efficient code:

```
print f(2)
```

```
3.2432
```

We can do symbolic derivatives: f' = dz/dx .

```
print SX.grad(f)
```

```
((x+x)*(1+cos(sq(x))))
```

The following code creates and evaluates a multi input (scalar valued), multi output (scalar valued) function.

```
x = SX.sym("x") # 1 by 1 matrix serves as scalar

y = SX.sym("y") # 1 by 1 matrix serves as scalar

f = Function('f', [x , y ], [x*y, x+y])
print "%d -> %d" % (f.n_in(),f.n_out())
```

```
2 -> 2
```

```
r = f(2, 3)

print [r[i] for i in range(2)]
```

```
[DM(6), DM(5)]
```

```
print [[SX.grad(f,i,j) for i in range(2)] for j in range(2)]
```

```
[[SX(y), SX(x)], [SX(1), SX(1)]]
```

Symbolic function manipulation

```
x=SX.sym("x")
a=SX.sym("a")
b=SX.sym("b")
```

```
f = Function('f', [x,vertcat(a,b)],[a*x + b])

print f(x,vertcat(a,b))
```

```
((a*x)+b)
```

```
print f(SX(1.0),vertcat(a,b))
```

```
(a+b)
```

```
print f(x,vertcat(SX.sym("c"),SX.sym("d")))
```

```
((c*x)+d)
```

```
print f(SX(),vertcat(SX.sym("c"),SX.sym("d")))
```

```
d
```

```


k = SX(a)
print f(x,vertcat(k[0],b))
```

```
((a*x)+b)
```

```
print f(x,vertcat(SX.sym("c"),SX.sym("d")))
```

```
((c*x)+d)
```

Functions with vector valued input

The following code creates and evaluates a single input (vector valued), single output (vector valued) function.

```

x = SX.sym("x")
y = SX.sym("y")
f = Function('f', [vertcat(x, y)], [vertcat(x*y, x+y)])
print "%d -> %d" % (f.n_in(),f.n_out())
```

```
1 -> 1
```

```
r = f([2, 3])
print z
```

```
@1=sq(x), (sin(@1)+@1)
```

```
G=SX.jac(f).T
print G
```

```
@1=1,
[[y, @1],
 [x, @1]]
```

The evaluation of v can be efficiently achieved by automatic differentiation as follows:

```
df = f.derivative(1,0)
res = df([2,3], [7,6])
print res[1] # v
```

```
[33, 13]
```

Functions with matrix valued input

```
x = SX.sym("x",2,2)
y = SX.sym("y",2,2)
print x*y # Not a dot product
```

```
[[(x_0*y_0), (x_2*y_2)],
 [(x_1*y_1), (x_3*y_3)]]
```

```
f = Function('f', [x,y], [x*y])
print "%d -> %d" % (f.n_in(),f.n_out())
```

```
2 -> 1
```

```
print f(x,y)
```

```
[[(x_0*y_0), (x_2*y_2)],
 [(x_1*y_1), (x_3*y_3)]]
```

```
r = f(DM([[1,2],[3,4]]), DM([[4,5],[6,7]]))
print r
```

```
[[4, 10],
 [18, 28]]
```

```
print SX.jac(f,0).T
```

```
[[y_0, 00, 00, 00],
 [00, y_1, 00, 00],
 [00, 00, y_2, 00],
 [00, 00, 00, y_3]]
```

```
print SX.jac(f,1).T
```

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

```

print 12
```

```
12
```

```

f = Function('f', [x,y], [x*y,x+y])
print type(x)
```

```
<class 'casadi.casadi.SX'>
```

```
print f(x,y)
```

```
(SX(
[[(x_0*y_0), (x_2*y_2)],
 [(x_1*y_1), (x_3*y_3)]]), SX(
```

```
[[(x_0+y_0), (x_2+y_2)],
 [(x_1+y_1), (x_3+y_3)]]))
```

```python
print type(f(x,y))
```

```
<type 'tuple'>
```

```python
print type(f(x,y)[0])
```

```
<class 'casadi.casadi.SX'>
```

```python
print type(f(x,y)[0][0,0])
```

```
<class 'casadi.casadi.SX'>
```

```python

f = Function('f', [x], [x+y])
print type(x)
```

```
<class 'casadi.casadi.SX'>
```

```python
print f(x)
```

```

[[(x_0+y_0), (x_2+y_2)],
 [(x_1+y_1), (x_3+y_3)]]
```

```python
print type(f(x))
```

```
<class 'casadi.casadi.SX'>
```

A current limitation is that matrix valued input/ouput is handled through flattened vectors Note the peculiar form of the gradient.

Conclusion

This tutorial showed how Function allows for symbolic or numeric evaluation and differentiation.