CasADi tutorial 1

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

This tutorial file explains the use of CasADi's SX in a python context. Let's start with the import statements to load CasADi.

```
from casadi import *
from numpy import *
```

Contructors & printing

Always provide a name to your symbols. This name is not used for identification, only for printing.

```
a = SX.sym("z")
print type(a)
```

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

```
print a
```

```
z
```

You can explicitely create constant SX objects as follows

```
c = SX(5)
print c
```

```
5
```

Scalar algebra

Any operations on SX objects return SX objects.

```
x = SX.sym("x")
y = SX.sym("y")
c = x+y
print type(c)
```

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

```
print c
```

```
(x+y)
```

While you construct ever complex expressions, a graph of SX objects is created.

```
d = c*2 + x
print d
```

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

Note that, by itself, CasADi does very little expansions or simplifications of expressions. Only simplifications that do not to introduce new nodes to the graph are allowed.

```
print d-x
```

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

```
print simplify(d-x)
```

```
@1=2, ((@1*x)+(@1*y))
```

```
print SX(5) + SX(7)
```

```
12
```

```
print 0*x + 0*y
```

```
0
```

```
print 1*x
```

```
x
```

SX objects are immutable entities. The assignment and update operators are really creating new object instead of modifying the old ones.

```
print "object address before: %d" % id(d)
```

```
object address before: 47933456
```

```
d = d - x
print d
```

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

```
print "object address after: %d" % id(d)
```

```
object address after: 37372432
```

Consequently, updates and assignements don't have side effects for other SX objects

```
f = x + y
x *= 2
print x
```

```
(2*x)
```

```
print f
```

```
(x+y)
```

```
print x+y
```

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

Expression substitution

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

y=x*x
print y
```

```
sq(x)
```

```
print substitute(y,x,SX.sym("w"))
```

```
sq(w)
```

```
print y
```

```
sq(x)
```

More operators

Some familiar mathematical operations are supported that mimic the standard numpy functions: sqrt sin cos tan arctan arcsin arccos exp log pow fabs floor ceil erf fmin fmax. Using these functions require numpy to be imported.

```
y = sin(x**x)

x=SX.sym("x")
print type(x>0)
```

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

```
t = if_else(x>0,-10,10)
print t
```

```
@1=(0<x), ((@1?-10:0)+((!@1)?10:0))
```

Note that 'a<b' is treated as '!(a>=b)'

Conclusion

We have seen how SX objects behave like symbolic objects. They can be used to contruct expression trees.

To see how we can efficiently evaluate and differentiate these objects, jump on to the sxfunction tutorial...