CasADi tutorial

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

This tutorial file explains the use of structures Structures are a python-only feature from the tools library:

```

from casadi.tools import *
```

The struct tools offer a way to structure your symbols and data It allows you to make abstraction of ordering and indices.

Put simply, the goal is to eleminate code with 'magic numbers' numbers such as:

> f = Function('f', [V],[V[214]]) # time ... x_opt = solver.getOutput()[::5] # Obtain all optimized x's

and replace it with f = Function('f', [V],[V["T"]]) ... shooting(solver.getOutput())["x",:]

Introduction

Create a structured SX.sym

```
states = struct_symSX(["x","y","z"])

print states
```

```
symbolic SX with following structure:
Structure with total size 3.
Structure holding 3 entries.
  Order: ['x', 'y', 'z']
  y = 1-by-1 (dense)
  x = 1-by-1 (dense)
  z = 1-by-1 (dense)
```

Superficially, states behaves like a dictionary

```
print states["y"]
```

```
y
```

To obtain aliases, use the Ellipsis index:

```
x,y,z = states[...]
```

The cat attribute will return the concatenated version of the struct. This will always be a column vector

```
print states.cat
```

```
[x, y, z]
```

This structure is of size:

```
print states.size, "=", states.cat.shape
```

```
3 = (3, 1)
```

```


f = Function('f', [states.cat],[x*y*z])
```

In many cases, states will be auto-cast to SX:

```
f = Function('f', [states],[x*y*z])
```

Expanded structure syntax and ordering

The structure defnition above can also be written in expanded syntax:

```
simplestates = struct_symSX([
    entry("x"),
    entry("y"),
    entry("z")
  ])
```

More information can be attached to the entries shape argument : specify sparsity/shape

```
states = struct_symSX([
    entry("x",shape=3),
    entry("y",shape=(2,2)),
    entry("z",shape=Sparsity.lower(2))
  ])

print states["x"]
```

```
[x_0, x_1, x_2]
```

```
print states["y"]
```

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

```
print states["z"]
```

```
[[z_0, 00],
 [z_1, z_2]]
```

Note that the cat version of this structure does only contain the nonzeros

```
print states.cat
```

```
[x_0, x_1, x_2, y_0, y_1, y_2, y_3, z_0, z_1, z_2]
```

repeat argument : specify nested lists

```
states = struct_symSX([
    entry("w",repeat=2),
    entry("v",repeat=[2,3]),
  ])

print states["w"]
```

```
[SX(w_0), SX(w_1)]
```

```
print states["v"]
```

```
[[SX(v_0_0), SX(v_0_1), SX(v_0_2)], [SX(v_1_0), SX(v_1_1), SX(v_1_2)]]
```

Notice that all v variables come before the v entries:

```
for i,s in enumerate(states.labels()):
  print i, s
```

```
0 [w,0,0]
1 [w,1,0]
2 [v,0,0,0]
3 [v,0,1,0]
4 [v,0,2,0]
5 [v,1,0,0]
6 [v,1,1,0]
7 [v,1,2,0]
```

We can influency this order by introducing a grouping bracket:

```

states = struct_symSX([
    "a",
    ( entry("w",repeat=2),
      entry("v",repeat=[2,3])
    ),
    "b"
  ])
```

Notice how the w and v variables are now interleaved:

```
for i,s in enumerate(states.labels()):
  print i, s
```

```
0 [a,0]
1 [w,0,0]
2 [v,0,0,0]
3 [v,0,1,0]
4 [v,0,2,0]
5 [w,1,0]
6 [v,1,0,0]
7 [v,1,1,0]
8 [v,1,2,0]
9 [b,0]
```

Nesting, Values and PowerIndex

Structures can be nested. For example consider a statespace of two cartesian coordinates and a quaternion

```
states = struct_symSX(["x","y",entry("q",shape=4)])

shooting = struct_symSX([
  entry("X",repeat=[5,3],struct=states),
  entry("U",repeat=4,shape=1),
])

print shooting.size
```

```
94
```

The canonicalIndex is the combination of strings and numbers that uniquely defines the entries of a structure:

```
print shooting["X",0,0,"x"]
```

```
X_0_0_x
```

If we use more exoctic indices, we call this a powerIndex

```
print shooting["X",:,0,"x"]
```

```
[SX(X_0_0_x), SX(X_1_0_x), SX(X_2_0_x), SX(X_3_0_x), SX(X_4_0_x)]
```

Having structured symbolics is one thing. The numeric structures can be derived: The following line allocates a DM of correct size, initialised with zeros

```
init = shooting(0)

print init.cat
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    ]
```

```

init["X",0,-1,"y"] = 12
```

The corresponding numerical value has changed now:

```
print init.cat
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    ]
```

The entry that changed is in fact this one:

```
print init.f["X",0,-1,"y"]
```

```
[13]
```

```
print init.cat[13]
```

```
12
```

One can lookup the meaning of the 13th entry in the cat version as such: Note that the canonicalIndex does not contain negative numbers

```
print shooting.getCanonicalIndex(13)
```

```
('X', 0, 2, 'y', 0)
```

```

print shooting.labels()[13]
```

```
[X,0,2,y,0]
```

Other datatypes

A symbolic structure is immutable

```

try:
  states["x"] = states["x"]**2
except Exception as e:
  print "Oops:", e
```

```
Oops: 'ssymStruct' object does not support item assignment
```

If you want to have a mutable variant, for example to contian the right hand side of an ode, use struct_SX:

```
rhs = struct_SX(states)

rhs["x"] = states["x"]**2
rhs["y"] = states["y"]*states["x"]
rhs["q"] = -states["q"]

print rhs.cat
```

```
[sq(x), (y*x), (-q_0), (-q_1), (-q_2), (-q_3)]
```

Alternatively, you can supply the expressions at defintion time:

```
x,y,q = states[...]
rhs = struct_SX([
    entry("x",expr=x**2),
    entry("y",expr=x*y),
    entry("q",expr=-q)
  ])

print rhs.cat
```

```
[sq(x), (x*y), (-q_0), (-q_1), (-q_2), (-q_3)]
```

One can also construct symbolic MX structures

```
V = struct_symMX(shooting)

print V
```

```
MX.sym with following structure:
Structure with total size 94.
Structure holding 2 entries.
  Order: ['X', 'U']
  X = repeated([5, 3]): {y: 1-by-1 (dense),x: 1-by-1 (dense),q: 4-by-1 (
      dense)}
  U = repeated([4]): 1-by-1 (dense)
```

The catted version is one single MX from which all entries are derived:

```
print V.cat
```

```
V
```

```
print V.shape
```

```
(94, 1)
```

```
print V["X",0,-1,"y"]
```

```
vertsplit(vertsplit(V){2}){1}
```

Similar to struct_SX, we have struct_MX:

```
V = struct_MX([
    (
    entry("X",expr=[[ MX.sym("x",6)**2 for j in range(3)] for i in range(5)])
    ,
    entry("U",expr=[ -MX.sym("u") for i in range(4)])
    )
  ])
```

By default SX.sym structure constructor will create new SX.syms. To recycle one that is already available, use the 'sym' argument:

```
qsym = SX.sym("quaternion",4)
states = struct_symSX(["x","y",entry("q",sym=qsym)])
print states.cat
```

```
[x, y, quaternion_0, quaternion_1, quaternion_2, quaternion_3]
```

The 'sym' feature is not available for struct_MX, since it will construct one parent MX.

More powerIndex

As illustrated before, powerIndex allows slicing

```
print init["X",:,:,"x"]
```

```
[[DM(0), DM(0), DM(0)], [DM(0), DM(0), DM(0)], [DM(0), DM(0), DM(0)], [DM
    (0), DM(0), DM(0)], [DM(0), DM(0), DM(0)]]
```

The repeated method duplicates its argument a number of times such that it matches the length that is needed at the lhs

```
init["X",:,:,"x"] = repeated(range(3))

print init["X",:,:,"x"]
```

```
[[DM(0), DM(1), DM(2)], [DM(0), DM(1), DM(2)], [DM(0), DM(1), DM(2)], [DM
    (0), DM(1), DM(2)], [DM(0), DM(1), DM(2)]]
```

Callables/functions can be thrown in in the powerIndex at any location. They operate on subresults obtain from resolving the remainder of the powerIndex

```

print init["X",:,lambda v: horzcat(*v),:,"x"]
```

```
[DM([[0, 1, 2]]), DM([[0, 1, 2]]), DM([[0, 1, 2]]), DM([[0, 1, 2]]), DM([[
    0, 1, 2]])]
```

```
print init["X",lambda v: vertcat(*v),:,lambda v: horzcat(*v),:,"x"]
```

```
[[0, 1, 2],
 [0, 1, 2],
 [0, 1, 2],
 [0, 1, 2],
 [0, 1, 2]]
```

```
print init["X",blockcat,:,:,"x"]
```

```
[[0, 1, 2],
 [0, 1, 2],
 [0, 1, 2],
 [0, 1, 2],
 [0, 1, 2]]
```

Set all quaternions to 1,0,0,0

```
init["X",:,:,"q"] = repeated(repeated(DM([1,0,0,0])))
```

{} can be used in the powerIndex to expand into a dictionary once

```
init["X",:,0,{}] = repeated({"y": 9})
```

```
print init["X",:,0,{}]
```

```
[{'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0])}, {'y': DM(9), 'x': DM(0),
    'q': DM([1, 0, 0, 0])}, {'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0]
    )}, {'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0])}, {'y': DM(9), 'x':
    DM(0), 'q': DM([1, 0, 0, 0])}]
```

lists can be used in powerIndex in both list context or dict context:

```
print shooting["X",[0,1],[0,1],"x"]
```

```
[[SX(X_0_0_x), SX(X_0_1_x)], [SX(X_1_0_x), SX(X_1_1_x)]]
```

```
print shooting["X",[0,1],0,["x","y"]]
```

```
[[SX(X_0_0_x), SX(X_0_0_y)], [SX(X_1_0_x), SX(X_1_0_y)]]
```

nesteddict can be used to expand into a dictionary recursively

```
print init[nesteddict]
```

```
{'X': [[{'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0])}, {'y': DM(0), 'x':
    DM(1), 'q': DM([1, 0, 0, 0])}, {'y': DM(12), 'x': DM(2), 'q': DM([1,
    0, 0, 0])}], [{'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0])}, {'y':
    DM(0), 'x': DM(1), 'q': DM([1, 0, 0, 0])}, {'y': DM(0), 'x': DM(2), 'q
    ': DM([1, 0, 0, 0])}], [{'y': DM(9), 'x': DM(0), 'q': DM([1, 0, 0, 0])
    }, {'y': DM(0), 'x': DM(1), 'q': DM([1, 0, 0, 0])}, {'y': DM(0), 'x':
    DM(2), 'q': DM([1, 0, 0, 0])}], [{'y': DM(9), 'x': DM(0), 'q': DM([1,
    0, 0, 0])}, {'y': DM(0), 'x': DM(1), 'q': DM([1, 0, 0, 0])}, {'y': DM
    (0), 'x': DM(2), 'q': DM([1, 0, 0, 0])}], [{'y': DM(9), 'x': DM(0), 'q
    ': DM([1, 0, 0, 0])}, {'y': DM(0), 'x': DM(1), 'q': DM([1, 0, 0, 0])},
     {'y': DM(0), 'x': DM(2), 'q': DM([1, 0, 0, 0])}]], 'U': [DM(0), DM(0)
    , DM(0), DM(0)]}
```

... will expand entries as an ordered list

```
print init["X",:,0,...]
```

```
[[DM(0), DM(9), DM([1, 0, 0, 0])], [DM(0), DM(9), DM([1, 0, 0, 0])], [DM
    (0), DM(9), DM([1, 0, 0, 0])], [DM(0), DM(9), DM([1, 0, 0, 0])], [DM
    (0), DM(9), DM([1, 0, 0, 0])]]
```

If the powerIndex ends at the boundary of a structure, it's catted version is returned:

```
print init["X",0,0]
```

```
[0, 9, 1, 0, 0, 0]
```

If the powerIndex is longer than what could be resolved as structure, the remainder, extraIndex, is passed onto the resulting Casadi-matrix-type

```
print init["X",blockcat,:,:,"q",0]
```

```
[[1, 1, 1],
 [1, 1, 1],
 [1, 1, 1],
 [1, 1, 1],
 [1, 1, 1]]
```

```
print init["X",blockcat,:,:,"q",0,0]
```

```
[[1, 1, 1],
 [1, 1, 1],
 [1, 1, 1],
 [1, 1, 1],
 [1, 1, 1]]
```

shapeStruct and delegated indexing

When working with covariance matrices, both the rows and columns relate to states

```
states = struct(["x","y",entry("q",repeat=2)])
V = struct_symSX([
      entry("X",repeat=5,struct=states),
      entry("P",repeat=5,shapestruct=(states,states))
   ])
```

P has a 4x4 shape

```
print V["P",0]
```

```
[[P_0_0, P_0_4, P_0_8, P_0_12],
 [P_0_1, P_0_5, P_0_9, P_0_13],
 [P_0_2, P_0_6, P_0_10, P_0_14],
 [P_0_3, P_0_7, P_0_11, P_0_15]]
```

Now we can use powerIndex-style in the extraIndex:

```
print V["P",0,["x","y"],["x","y"]]
```

```
[[P_0_0, P_0_4],
 [P_0_1, P_0_5]]
```

There is a problem when we wich to use the full potential of powerIndex in these extraIndices: The following is in fact invalid python syntax:

We resolve this by using delegater objects index/indexf:

```
print V["P",0,indexf["q",:],indexf["q",:]]
```

```
[[P_0_10, P_0_14],
 [P_0_11, P_0_15]]
```

Of course, in this basic example, also the following would be allowed

```
print V["P",0,"q","q"]
```

```
[[P_0_10, P_0_14],
 [P_0_11, P_0_15]]
```

Prefixing

The prefix attribute allows you to create shorthands for long powerIndices

```
states = struct(["x","y","z"])
V = struct_symSX([
```

```
        entry("X",repeat=[4,5],struct=states)
])

num = V()
```

Consider the following statements:

```

num["X",0,0,"x"] = 1
num["X",0,0,"y"] = 2
num["X",0,0,"z"] = 3
```

Note the common part [“X”,0,0]. We can pull this apart with prefix:

```

initial = num.prefix["X",0,0]

initial["x"] = 1
initial["y"] = 2
initial["z"] = 3
```

This is equivalent to the longer statements above

Helper constructors

If you work with Simulator, ControlSimulator, you typically end up with wanting to index a DM that is n x N with n the size of a statespace and N an arbitrary integer

```

states = struct(["x","y","z"])
```

We artificially construct here a DM that could be a Simulator output.

```
output = DM.zeros(states.size,8)
```

The helper construct is ’repeated’ here. Instead of “states(output)”, we have

```
outputs = states.repeated(output)
```

Now we have an object that supports powerIndexing:

```
outputs[-1] = DM([1,2,3])
outputs[:,"x"] = range(8)

print output
```

```
[[0, 1, 2, 3, 4, 5, 6, 7],
 [0, 0, 0, 0, 0, 0, 0, 2],
 [0, 0, 0, 0, 0, 0, 0, 3]]
```

```
print outputs[5,{}]
```

```
{'y': DM(0), 'x': DM(5), 'z': DM(0)}
```

Next we represent the ’squared’ helper construct Imagine we somehow obtain a matrix that represents covariance

```
P0 = DM.zeros(states.size,states.size)
```

We can conveniently access it as follows:

```
P = states.squared(P0)
P["x","y"] = 2
P["y","x"] = 3

print P0
```

```
[[0, 2, 0],
 [3, 0, 0],
 [0, 0, 0]]
```

P itself is a rather queer object

```
print P
```

```
prefix( ('t',),Mutable DM ({t: 3-by-3 (dense)}))
```

You can access its concents with a call:

```
print P()
```

```
[[0, 2, 0],
 [3, 0, 0],
 [0, 0, 0]]
```

But often, it will behave like a DM transparantly:

```
P0 + P
```

Next we represent the ’squared_repeated’ helper construct Imagine we somehow obtain a matrix that represents a horizontal concatenation of covariance

```
P0 = horzcat(DM.zeros(states.size,states.size),DM.ones(states.size,states.
   size))
```

We can conveniently access it as follows:

```
P = states.squared_repeated(P0)
P[0,"x","y"] = 2
P[:,"y","x"] = 3

print P0
```

```
[[0, 2, 0, 1, 1, 1],
 [3, 0, 0, 3, 1, 1],
 [0, 0, 0, 1, 1, 1]]
```

Finally, we present the ’product’ helper construct

```
controls = struct(["u","v"])

J0 = DM.zeros(states.size,controls.size)

J = states.product(controls,J0)

J[:,"u"] = 3
J[["x","z"],:] = 2

print J()
```

```
[[2, 2],
 [3, 0],
 [2, 2]]
```

Saving and loading

It is possible to save and load some types of structures. Supported types are pure structures (the ones created with 'struct') and numeric structures.

Saving: mystructure.save("myfilename")

Loading:

struct_load("myfilename")

or pickle.load(file('myfilename','r'))