CasADi tutorials, 8 December 2011

Exercise 4: MX and OCP

Joel Andersson

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

December 7, 2011

1 The MX symbolics

Let us perform a simple operation using the SXMatrix type from exercise 1:

```
x = ssym("x",2,2)
y = ssym("y")
f = 3*x + y
print "f = ", f
```

As you can see, the output of this operaton is the 2-by-2 matrix

```
[[((3*x_0_0)+y),  ((3*x_0_1)+y) ]
 [((3*x_1_0)+y),  ((3*x_1_1)+y) ]]
```

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.

This exercise will deal with 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 ($\mathbb{R} \to \mathbb{R}$ or $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. 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: $\mathbb{R}^{n_1 \times m_1} \times \ldots \times \mathbb{R}^{n_N \times m_N} \to \mathbb{R}^{p_1 \times q_1} \times \ldots \times \mathbb{R}^{p_M \times q_M}$. In particular, we allow *calls* to functions of type FX (including calls to ODE integrators and SXFunction instances).

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

```
x = msym("x",2,2)
y = msym("y")
f = 3*x + y
print "f = ", f
```

where msym has replaced ssym. The output of this expression is now simply:

```
((3*x)+y)
```

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). MX is thus more economical when working with operations that are naturally vector or matrix valued. It is also much more general since we allow calls to arbitrary functions that cannot be expanded in terms of elementary operations (like a call to CVodes).

`MX` supports getting and setting elements, using the same syntax as `SXMatrix`, but the way it is implemented is very different. Test, for example, to print the first column of a 2-by-2 symbolic variable:

```
x = msym("x",2,2)
print x[:,0]
# Output: mapping(dense 2-by-1 matrix, dependencies: [x], nonzeros: [0,2])
```

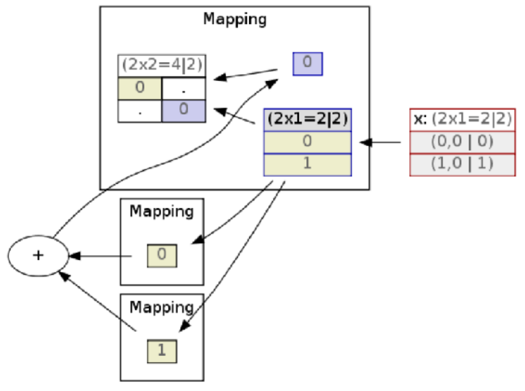
The output should be understood as a (linear) mapping from nonzeros 0 and 2 of the expression `x` to a new expression of dimension 2-by-1.

Similar outputs can be expected when trying to set elements:

```
x = msym("x",2)
A = MX.sparse(2,2)
A[0,0] = x[0]
A[1,1] = x[0]+x[1]
print A
# Output: mapping(diagonal 2-by-2 matrix, dependencies: [x,(x[0]+x[1])], \
#                  nonzeros: [0(0),0(1)]
```

This time expression `A` becomes a mapping from `x` to a sparse 2-by-2 matrix. Linear mappings, of the type you've seen here, is also the the result of several other operations that you might perform on an expression, including (but not limited to) matrix transposes, horizontal and vertical concatenations, resizings and reshapings.

This output tend to become quite hard to understand. An alternative (not always better) way is to represent the function is as a graph:



```
    from casadi import tools
    tools.dotdraw(A)
```

Using this feature, which is still under development, requires that you have installed the Python package *pydot*. We actively work on getting the expressions more easier to grasp.

`MX` expressions may contain calls to CasADi functions. To embed a call to an `FX`-derived function (e.g. `SXFunction` or `CVodesIntegrator`), you use the syntax:

```
[y1,y2,...,yn] = f.call([x1,x2,...,xn])
```

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

Expressions formulated using `MX` expressions can be used to formulate functions of type `MXFunction`:

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

2 Mixing `SX` and `MX`

You can *not* multiply an `SXMatrix` with a `MX`, or perform any other operation to mix the two in the same expression graph. You can, however, as mentioned above, include calls to an `SXFunction` in an `MX` graph. This is often a good idea since `SXFunction` is several times faster than `MXFunction` when dealing with small to medium size expressions (up to some thousands of variables). It is also much more stable and contains fewer bugs.

The `SX` expressions is thus intended to be used for low level operations (for example the DAE right hand side), 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).

An `MXFunction` which only contains built-in operations (e.g. +, *, linear mappings, 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.

3 Automatic differentiation

Like the other classes derived from `FX`, `MXFunction` supports numerical evaluation with forward and adjoint sensitivity calculations etc., using the syntax you learnt in exercises 1 and 2.

Using directional derivatives should mostly work "out of the box".

Just as for the `SX` type, `MX` supports automatic differentiation symbolically, both in forward and adjoint mode, which can be used to create new expressions for e.g. Jacobian and Hessian functions. This feature is still, however, not completely stable (especially for the adjoint mode) and should only be used with caution for the time being.

4 Optimal control with CasADi

CasADi can be used to solve optimal control problem using a variety of methods. As a user, there are two ways to attack the problem:

- Use an existing OCP solver
- Write your own OCP solver

Using an existing OCP solver is attractive for the user who has a relatively simple problem that he wants to test using different methods. In the next exercise, you will get some experience on using such solvers within the JModelica.org framework. Some of these solvers use CasADi.

The option to using an existing solver is to write your own OCP solver. While definitely more involved than using JModelica.org's solver, it is a viable alternative especially when adressing a challing problem, or a problem using a nonstandard problem formulation. Doing this, CasADi will take care of things like Jacobian and Hessian generation, sparsity exploitation and all that is left to the user is to perform the reformulation from the infinite dimensional optimal control problem to a finite dimensional NLP.

5 Optimization of a Van der Pol oscillator

In CasADi's examples collection, you find codes which all try to solve a simple optimal control problem, a problem borrowed from JModelica.org's examples collection, namely that of driving a *Van der Pol* oscillator to the origin, while trying to minimize a quadratic cost:

$$
\begin{aligned}
&\underset{x(\cdot) \in \mathbb{R}^2,\ u(\cdot) \in \mathbb{R}}{\text{minimize:}} \quad \int_{t=0}^{10} (x_0^2 + x_1^2 + u^2)\, dt \\
&\text{subject to:} \\
&\begin{cases} \dot{x}_0 = (1 - x_1^2)\, x_0 - x_1 + u \\ \dot{x}_1 = x_0 \\ -0.75 \le u \le 1.0 \end{cases} \qquad \text{for } 0 \le t \le 10 \\
&x_0(0) = 0, \quad x_0(10) = 0 \\
&x_1(0) = 1, \quad x_1(10) = 0
\end{aligned}
\tag{1}
$$

6 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,Xp] = f_d.call([X,U[k],Xp])
```

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 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 = msym("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,Xp] = f_d.call([Xk,U[k],Xp])

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

8 Exercises

4.1 Recall the simple rocket model from the first exercise: A simple model for the flight of a rocket is the following differential equation:

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

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.

With a time horizon of $T = 10$ s and $n_k = 20$ control segments, solve the optimal control problem of minimizing the control effort[1]:

$$\int_{t=0}^{T} u(t)^2\, dt \tag{3}$$

The rocket starts at rest at $h = 0$ with mass $m = 1$ and must finish in rest at $h = 10$ m.

Use single-shooting to solve the problem, using `vdp_single_shooting.py` as a template.

4.2 Constrain the velocity everywhere along the trajectory: i.e. $v < 1.1$.

4.3 Solve the same problem with direct multiple-shooting, using `vdp_multiple_shooting.py` as a template.

4.4 **Extra:** Read the section on direct collocation below and then solve the rocket flight problem with direct multiple-shooting, using `vdp_collocation.py` as a template.

9 Extra material: 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.

Direct collocation is to take one more step in the same direction: we achieve an even less nonlinear NLP, at the cost of an even larger NLP.

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

[1]Note that this is equivalent to maximizing the final mass of the rocket.

interval) enters in the NLP as variables. An example of a choice of time points are the *Legendre points* of order $d = 3$:

$$\tau = [0, 0.112702, 0.500000, 0.887298] \tag{4}$$

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

$$u(t) = u_k, \quad \text{for } t \in [t_k, t_{k+1}], \quad k = 0, \dots, n_k - 1 \tag{5}$$

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

$$t_{k,j} := t_k + h_k \, \tau_j, \quad \text{for } k = 0, \dots, n_k - 1 \text{ and } j = 0, \dots, d \tag{6}$$

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:

$$L_j(\tau) = \prod_{r=0,\, r \neq j}^{d} \frac{\tau - \tau_j}{\tau_r - \tau_j} \tag{7}$$

Since the Lagrangian basis satisfies:

$$L_j(\tau_r) = \begin{cases} 1, & \text{if } j = r \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

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

$$\tilde{x}_k(t) = \sum_{r=0}^{d} L_r \left(\frac{t - t_k}{h_k} \right) x_{k,r} \tag{9}$$

In particular we get approximations of the state time derivative at each collocation point (not including τ_0):

$$\dot{\tilde{x}}_k(t_{k,j}) = \frac{1}{h_k} \sum_{r=0}^{d} \dot{L}_r(\tau_j) \, x_{k,r} := \frac{1}{h_k} \sum_{r=0}^{d} C_{r,j} \, x_{k,r} \tag{10}$$

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

$$\tilde{x}_{k+1,0} = \sum_{r=0}^{d} L_r(1) \, x_{k,r} := \sum_{r=0}^{d} D_r \, x_{k,r} \tag{11}$$

Plugging in the approximation of the state derivative (10) into the ODE gives us a set of *collocation equations* that needs to be satisfied for every state at every collocation point:

$$h_k \, f(t_{k,j}, x_{k,j}, u_k) - \sum_{r=0}^{d} C_{r,j} \, x_{k,r} = 0, \qquad k = 0, \dots, n_k - 1, \quad j = 1, \dots, d \tag{12}$$

And the approximation of the end state (11) gives us a set of *continuity equations* that must be satisfied for every control interval:

$$x_{k+1,0} - \sum_{r=0}^{d} D_r \, x_{k,r} = 0, \qquad k = 0, \dots, n_k - 1, \tag{13}$$

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`.