CasADi tutorials Lund, December 2011

Exercise 2: ODE/DAE integration

Joel Andersson

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

December 5, 2011

1 ODE/DAE formulation

The integrators interfaced with CasADi assumes a DAE residual function of the fully implicit form:

$$f(t, x, p, \dot{x}) = 0 \tag{1}$$

Solvers for *ordinary* differential equations will make the additional assumption that the structure of the function is:

$$f_{\text{ode}}(t, x, p) - \dot{x} = 0 \tag{2}$$

the explicit equation for $\dot{x}$ can thus be retrieved by simply evaluating the DAE function with $\dot{x} = 0$.

It is important that the arguments of the function uses the order above, and for this reason, we strongly recommend users to work with a set of constants defining the required input and output schemes of the function. For the DAE residual function, these constants are `DAE_NUM_IN=4`, `DAE_T=0`, `DAE_Y=1`, `DAE_P=2` and `DAE_YDOT=3` for the inputs and `DAE_NUM_OUT=1` and `DAE_RES=0` for the outputs.

An integrator in CasADi is a function that take the state at the initial time, guesses for the algebraic states and state derivatives (only important for DAEs) and evaluates the state vector at the final time. The time horizon is assumed to be fixed[1] and can be set with the option:

```
integrator.setOption("tf",integration_end_time)
```

2 Sundials integrators

The Sundials suite contains the two popular integrators CVodes and IDAS for ODEs and DAEs respectively. These two integrators supports forward and adjoint sensitivities and when used via CasADi's Sundials interface, CasADi will automatically formulate the Jacobian information, which is needed by the backward differentiation formula (BDF) that CVodes and IDAS use. Also automatically formulated will be the forward and adjoint sensitivitiy equations. This means that the only information that the user needs to provide is the DAE residual function:

```
integrator = CVodesIntegrator(f)   or  integrator = IdasIntegrator(f)
```

[1] for problems with free end time, you can always scale time by introducing an extra parameter and substitute t for a dimensionless time variable that goes from 0 to 1

for CVodes and IDAS respectively.

For a list of options for the integrators, as well as the input and output schemes of this *function*, check the documentation directly from Python:

```
CVodesIntegrator?
```

or by consulting the online C++ API docs on the website.

3 Sensitivity analysis

From evaluation point of view, an integrator behaves just like the `SXFunction` introduced in the previous session. You set inputs, forward/adjoint seeds, evaluate and obtain the outputs and forward/adjoint sensitivities.

4 The `Simulator` class

As already mentioned, integrators in CasADi are functions that calculates the state at the final time. Often, however, a user is interested in obtaining the solution at multiple time points. This can often be done more efficiently than by repeatedly calling `integrator.evaluate()`. The easiest way to use this functionality is to use the `Simulator` class.

A `Simulator` can be created using the syntax:

```
# Import numpy
import numpy as NP

# Allocate an integrator instance
integrator = ...
integrator.setOption("...",...)

# Choose a time grid
tgrid = NP.linspace(0,end_time,num_time_steps)

# Create a simulator
simulator = Simulator(integrator, time_grid)
```

A `Simulator` can be used just like an integrator, and its input scheme is the same. Its output is now matrix valued, with the the columns corresponding to different time points. The class can also be used to evaluate a particular function of the state at a set of time points. See the API documentation for more information.

5 Exercises

2.1 Rewrite the rocket ODE from the previous exercise in the fully implicit form required by CasADi's integrators. Create a `CVodesIntegrator` instance and integrate from $t = 0$ to $t = 10.0$ with $u = 1$ and $x(0) = [0, 0, 1]^{\mathrm{T}}$. Calculate, using forward sensitivity analysis how the final state depends on the control. Also calculate, using adjoint sensitivity analysis how the m at the final time depends on the initial state and the control.

2.2 Integrate again by creating a `Simulator`, and stop att 100 intermediate times. Visualize the trajectory using a `matplotlib` plot. Note that the `Simulator` class currently does not support adjoint sensitivitiy analysis.

2.3 **Extra:** Add a *quadrature state* defined by the equation $\dot{q} = \|u - 3\,sin(t)\|_2^2$ without augmenting the ODE. Check CVodes documentation. What are the advantages/disadvantages of using quadrature states in general?

2.4 **Extra:** A Runge-Kutta 4 step can be written:

$$k_1 = f(x_k, u_k) \tag{3}$$

$$k_2 = f(x_k + \frac{1}{2}\,\Delta t\,k_1, u_k) \tag{4}$$

$$k_3 = f(x_k + \frac{1}{2}\,\Delta t\,k_2, u_k) \tag{5}$$

$$k_4 = f(x_k + \Delta t\,k_3, u_k) \tag{6}$$

$$x_{k+1} = x_k + \frac{1}{6}\,\Delta t\,(k_1 + 2\,k_2 + 2\,k_3 + k_4) \tag{7}$$

Use this formula to create an expression for this state at the final time using 100 RK4 steps. To evaluate your ODE/DAE function (which is of type `SXFunction`) symbolically, you can use the function *eval*(). Create a new `SXFunction` instance with the same input and output scheme as your CVodes integrator. Then repeat exercise 2.1 using this integrator and and compare the results that you got then.