Numerical Optimal Control, August 2014

Exercise 3: Initial-value problems

Joel Andersson Joris Gillis Greg Horn Rien Quirynen Moritz Diehl

University of Freiburg – IMTEK, August 5th, 2014

Explicit Euler vs. Runge-Kutta 4

Consider a controlled harmonic oscillator described by:

$$\frac{d}{dt}\begin{bmatrix} p(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} p(t) \\ v(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t), \quad t \in [0, T]. \tag{1}$$

We abbreviate this ODE as $\dot{x} = f(x, u)$ with $x = (p, v)^T$. We choose the fixed initial value $x(0) = (10, 0)^T$ and $T = 10$.

Tasks:

The first part of this session, we will just use `numpy` and `matplotlib` utilities, not CasADi. Load both of them with one statement: `from pylab import *`. Note that matrix mulplication in numpy is written with `dot(A,b)`, while for CasADi objects it is `mul(A,b)`.

3.1 We are interested in comparing the simulation results for $u(t) = 0$ that are obtained by two different integration schemes, namely the (explicit) Euler integrator and a Runge-Kutta integrator of 4th order. We regard in particular the value $p(10)$, and as the ODE is explicitly solvable, we know it exactly, which is useful for comparisons. What is the analytical expression for $p(10)$? Evaluate it numerically. This may be helpful:

```
from scipy.linalg import expm
```

3.2 Write a Python function named f using `def f(x,u)` that evaluates the right hand side of the ODE. Then, implement an explicit Euler method with $N = 50$ integrator steps, i.e. with a stepsize of $\Delta t = 10/50 = 0.2$. The central line in the Euler code reads

$$x_{k+1} \quad = \quad x_k + \Delta t \cdot f(x_k, u_k) \tag{2}$$

Plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for $u_k = 0$.

3.2 Now exchange in your Euler simulation code the line that generates the step (2) by the following five lines:

$$k_1 = f(x_k, u_k) \tag{3a}$$
$$k_2 = f(x_k + \frac{1}{2}\Delta t \cdot k_1, u_k) \tag{3b}$$
$$k_3 = f(x_k + \frac{1}{2}\Delta t \cdot k_2, u_k) \tag{3c}$$
$$k_4 = f(x_k + \Delta t \cdot k_3, u_k) \tag{3d}$$
$$x_{k+1} = x_k + \Delta t \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3e}$$

This is the classical Runge Kutta method of order four (RK4). Note that each integrator step is four times as expensive as an Euler step. What is the advantage of this extra effort? To get an idea, plot your trajectories $\{(t_k, x_k)\}_1^{N+1}$ for the same number N of integrator steps. **Beware**: remember to write `1.0/2` to avoid integer division.

3.3 Make both pieces of your integrating code reusable by creating Python functions named `euler` and `rk4` out them. Both should have arguments `x0`, `T` and `N` and return the state at the end point. Test your implementation by comparing with the plots.

3.4 To make the comparison of Euler and RK4 quantitative, regard the different approximations of $p(10)$ that you obtain for different stepsizes, e.g. $\Delta t = 10^{-k}$ with $k = 0, \ldots, 5$. We call these approximations $\tilde{p}(10; \Delta t)$. Compute the errors $|p(10) - \tilde{p}(10; \Delta t)|$ and plot them doubly logarithmic, i.e. using `loglog`. Use `norm` to calculate the norm of vectors. You should see a line for each integrator. Can you explain the different slopes? For easier interpretation, do `grid(True)` after plotting.

Integrators in CasADi

CasADi provides solvers for initial-value problems in differential-algebraic equations (DAE) in the following semi-explicit form:

$$\begin{array}{rcll} \dot{x} & = & f_{\text{ode}}(x, z, p, t), & x(t_0) = x_0, \\ 0 & = & f_{\text{alg}}(x, z, p, t), & \textit{initial guess of } z(t_0) = z_0 \\ \dot{q} & = & f_{\text{quad}}(x, z, p, t), & q(t_0) = 0, \end{array} \tag{4}$$

where x is the differential state, p is a (known) parameter, z is an algebraic variable and q is a quadrature state (i.e. a differential state that does not enter in the right-hand-side).

The DAE consists of three parts: (1) differential equation with known initial conditions, (2) algebraic equations that uniquely determines the algebraic variable z (including an initial guess of $z(t_0)$) and (3) quadrature equations. If well-posed, the initial-value problem (4) uniquely determines x, z and q at a time t_{f}.

To allocate an integrator, we construct a CasADi "DAE" function that takes x, z, p and t as inputs and returns f_{ode}, f_{alg} and f_{quad} (only x and f_{ode} are required), e.g.:

```
x = SX.sym("x",nx)
f_ode = ...
dae = SXFunction(daeIn(x=x),daeOut(ode=f_ode))
```

This function is then used to construct an `Integrator` instance as follows:

```
integrator = Integrator("cvodes",dae)
```

where we use CasADi's interface to the open-source integrator CVODES from the SUNDIALS suite. From the symbolic expressions, the interface will then generate all information needed to solve the problem efficiently, including calculating "sensitivities", as we shall see in Exercise 4.

Integrators are also *functions* in CasADi that are evaluated to get $x(t_{\text{f}})$, $z(t_{\text{f}})$ and $q(t_{\text{f}})$:

```
integrator.setInput(x0,"x0")
integrator.evaluate()   # Solve the initial-value problem
xf = integrator.getOutput("xf")
```

Tasks:

3.5 Solve the above integration using CVODES and compare the results. Note that the end time is passed as an "option" to the solver before the "init" step:

```
integrator.setOption("tf", 10)
```