ODE solvers in Julia

Gabriel Ingesson

October 2, 2015

Motivation

- General: Numerical methods for solving ODE's is important for system simulations. Simulation is important for controller design.
- Personal: In my research project it is interesting to solve ODE's fast, in the control loop. For this purpose, Julia seems to be an interesting alternative.

This Presentation Covers

- ODE solver packages in Julia.
- How to use them, what functionality they contain.
- Code examples and a comparison to Matlab solvers w.r.t. speed and functionality.

Essentially two packages available

- ODE.jl - Various basic Ordinary Differential Equation solvers implemented in Julia, used to be a part of Base. Supports fixed step, adaptive and stiff solvers.
- Sundials.jl - package that interfaces to the Sundials C library. SUite of Nonlinear and DIfferential/Algebraic equation Solvers.

Supports the following solvers

- ode23: 2nd order adaptive solver with 3rd order error control, using the Bogacki–Shampine coefficients.
- ode45: 4th order adaptive solver with 5th order error control, using the Dormand Prince coefficients. Fehlberg and Cash-Karp coefficients are also available.
- ode78: 7th order adaptive solver with 8th order error control, using the Fehlberg coefficients.
- ode23s: 2nd/3rd order adaptive solver for stiff problems, using a modified Rosenbrock triple.

ODE.jl

Supports the following solvers

- ode23: 2nd order adaptive solver with 3rd order error control, using the Bogacki–Shampine coefficients.
- ode45: 4th order adaptive solver with 5th order error control, using the Dormand Prince coefficients. Fehlberg and Cash-Karp coefficients are also available.
- ode78: 7th order adaptive solver with 8th order error control, using the Fehlberg coefficients.
- ode23s: 2nd/3rd order adaptive solver for stiff problems, using a modified Rosenbrock triple.

All of which have the following basic API:

tout, yout = odeXX(F, y0, tspan; keywords...)

For solving the following ODE at the instants of tspan

$$\frac{dy}{dt} = f(t,y),\ y(0) = y_0 \tag{1}$$

ODE.jl - Keywords

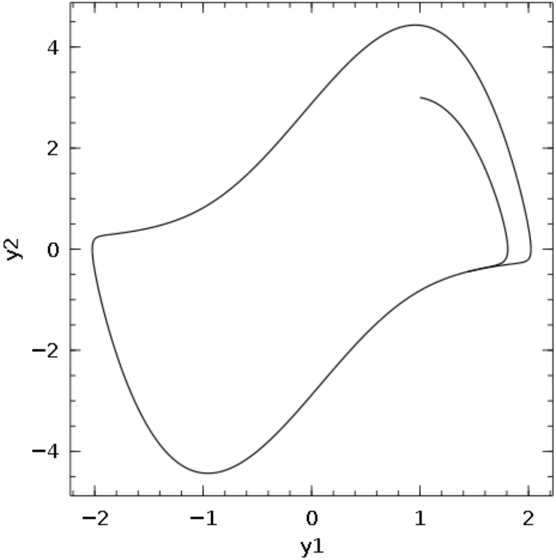
- norm: user-supplied norm for determining the error E (default Base.vecnorm),
- abstol and/or reltol: an integration step is accepted if $E <= \text{abstol} || E <= \text{reltol} * abs(y)$
- maxstep, minstep and initstep: determine the maximal, minimal and initial integration step.
- points=:all (default): output is given for each value in tspan as well as for each intermediate point the solver used. points=:specified: output is given only for each value in tspan.
- Additionally, ode23s solver supports jacobian $= G(t, y)$: user-supplied Jacobian $G(t, y) = dF(t, y)/dy$.
- Note: There are currently discussions about how the Julian API for ODE solvers should look like, and the current documentation is more like a wishlist than a documentation.

Example: Van Der Pool Oscillator

```
using ODE
using Winston
 function f(t, y)
       mu = 2.5
       ydot = similar(y)
       ydot[1] = y[2]
       ydot[2] = mu*(1-y[1]^2)*y[2]-y[1]
       ydot
 end

 t = [0:.1:10.0;]
 y0 = [1.0, 3.0]
 t,y=ODE.ode23s(f, y0, t)
 y1 = [ a[1] for a in y] # Rearranging the output,
 y2 = [ a[2] for a in y] # more convenient
 plot(float(y1),float(y2))
```

Example: Van Der Pool Oscillator



Sundials.jl

Containts:

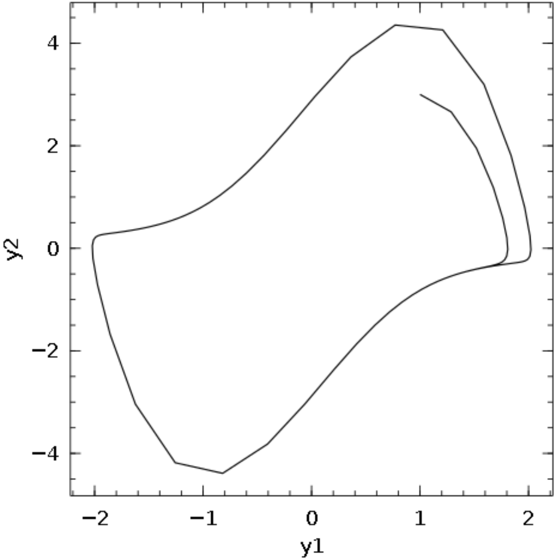
- CVODES - for integration and sensitivity analysis of ODEs. CVODES treats stiff and nonstiff ODE systems of the form $y' = f(t, y, p), y(t0) = y0(p)$, where p is a set of parameters.
- IDAS - for integration and sensitivity analysis of DAEs. IDAS treats DAE systems of the form $F(t, y, y', p) = 0, y(t0) = y0(p), y'(t0) = y0'(p)$.
- KINSOL - for solution of nonlinear algebraic systems. KINSOL treats nonlinear systems of the form $F(u) = 0$.

CVODES

$$\frac{dy}{dt} = f(t,y),\ y(0) = y_0$$

```
using Sundials
using Winston
function f(t,y,ydot)
   mu = 2.5
       ydot[1] = y[2]
       ydot[2] = mu*(1-y[1]^2)*y[2]-y[1]
       ydot
end
t = [0:.1:10.0;]
y0 = [1.0, 3.0]
res = Sundials.cvode(f, y0, t)
plot(res[:,1],res[:,2])
```

Example: Van Der Pool Oscillator

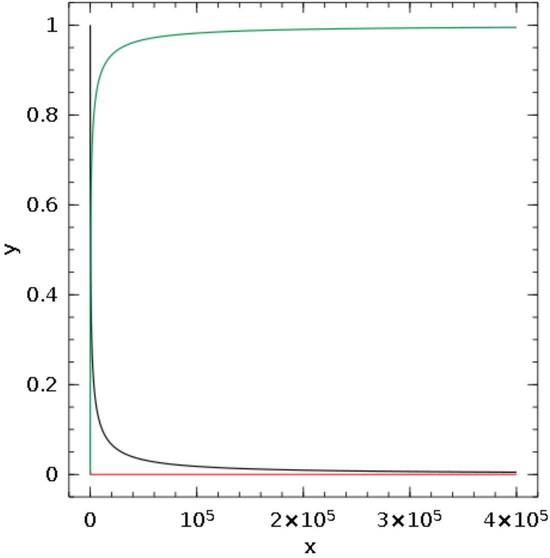


IDAS

$$F(dy/dt, y, t) = 0 \tag{2}$$

```
using Sundials
function resrob(tres, y, yp, r)
    r[1]  = -0.04*y[1] + 1.0e4*y[2]*y[3]
    r[2]  = -r[1] - 3.0e7*y[2]*y[2] - yp[2]
    r[1] -=  yp[1]
    r[3]  =  y[1] + y[2] + y[3] - 1.0
end
# yp is here the derivative vector.
t = [0.0, 4 * logspace(-1., 5., 100)]
yout, ypout =
Sundials.idasol(resrob,[1.0,0,0],[-0.04,0.04,0.0],t)
```

IDAS Example



KINSOL

$$F(y) = 0 \tag{3}$$

```
import Sundials
function sysfn(y, fy)
    fy[1] = y[1]^2 + y[2]^2 - 1.0
    fy[2] = y[2] - y[1]^2
end

sol = Sundials.kinsol(sysfn, ones(2))

julia> sol
2-element Ar
 0.786153
 0.618035
```

More code examples can be found at

https://github.com/JuliaLang/Sundials.jl
https://github.com/JuliaLang/Ode.jl

ODE.jl ode23s, Sundials CVODE and Matlabs ode23s

With the conditions

```
t = [0:.01:10.0;]
y0 = [1.0, 3.0]
abstol=1e-8; reltol=1e-8;

function f(t, y)  # Van Der Pool Oscillator
       mu = 3.0
       ydot = similar(y)
       ydot[1] = y[2]
       ydot[2] = mu*(1-y[1]^2)*y[2]-y[1]
       ydot
end
```

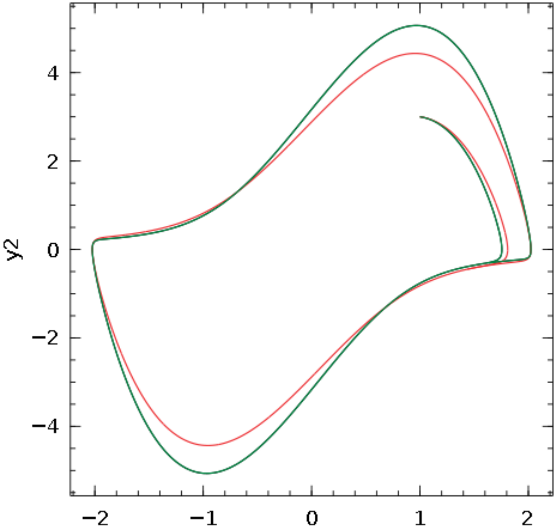
Results

ODE.jl ode23s
elapsed time: 3.539e-6 seconds (80 bytes allocated)

Sundials CVODEs
elapsed time: 4.954e-6 seconds (80 bytes allocated)

Matlab ode23s
Elapsed time is 1.099103 seconds.

Results, ODE.jl ode23s (blue), Sundials CVODEs (red), Matlabs ode23s (green)



Summary

- ODE.jl – is a work in progress, will probably be the main choice in the future.
- Sundials.jl – has a lot more functionality, might be a better short term solution.

Homework (Not Mandatory)

Check out the Three-Body Problem Sundials.jl Julia Code at

http://nbviewer.ipython.org/github/pjpmarques/Julia-Modeling-the-World/blob/master/Three-Body%20Problem.ipynb

and generate your own plots.