Choosing an ODE Algorithm

Chris Rackauckas

December 29, 2019

While the default algorithms, along with `alg_hints = [:stiff]`, will suffice in most cases, there are times when you may need to exert more control. The purpose of this part of the tutorial is to introduce you to some of the most widely used algorithm choices and when they should be used. The corresponding page of the documentation is the ODE Solvers page which goes into more depth.

0.1 Diagnosing Stiffness

One of the key things to know for algorithm choices is whether your problem is stiff. Let's take for example the driven Van Der Pol equation:

```
using DifferentialEquations, ParameterizedFunctions
van! = @ode_def VanDerPol begin
  dy = μ*((1-x^2)*y - x)
  dx = 1*y
end μ

prob = ODEProblem(van!,[0.0,2.0],(0.0,6.3),1e6)
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 6.3)
u0: [0.0, 2.0]
```

One indicating factor that should alert you to the fact that this model may be stiff is the fact that the parameter is `1e6`: large parameters generally mean stiff models. If we try to solve this with the default method:

```
sol = solve(prob,Tsit5())
```

```
retcode: MaxIters
Interpolation: specialized 4th order "free" interpolation
t: 999977-element Array{Float64,1}:
 0.0
 4.997501249375313e-10
 5.4972513743128435e-9
 3.289919594544218e-8
 9.055581394883546e-8
 1.7309428803584187e-7
 2.793753933394586e-7
 4.149527171475212e-7
 5.807919390815544e-7
 7.81280701490125e-7
 ⋮
 1.8457012081010522
 1.845702696026691
 1.8457041839548325
 1.8457056718857727
 1.845707159819413
 1.8457086477557534
 1.8457101356946952
 1.8457116236362385
 1.8457131115805805
u: 999977-element Array{Array{Float64,1},1}:
 [0.0, 2.0]
 [-0.0009987513736106552, 1.9999999999997504]
 [-0.010904339759596433, 1.9999999999699458]
 [-0.06265556194129239, 1.9999999989523902]
 [-0.1585948892562767, 1.9999999924944207]
 [-0.2700352862461109, 1.9999999746155703]
 [-0.3783197963325601, 1.9999999398563364]
 [-0.47467864703912216, 1.9999998815910678]
 [-0.5499302545937235, 1.999999796115446]
 [-0.6026934372089534, 1.9999996800439757]
 ⋮
 [-0.7770871866226842, 1.8321769350351387]
 [-0.7770880934309836, 1.8321757783565626]
 [-0.7770890004563554, 1.832174621674691]
 [-0.7770899073362528, 1.832173464989294]
 [-0.7770908141915421, 1.832172308300448]
 [-0.777091721022237, 1.8321711516081531]
 [-0.7770926279492066, 1.832169994912486]
 [-0.7770935349724621, 1.8321688382134467]
 [-0.7770944418503102, 1.8321676815108816]
```

Here it shows that maximum iterations were reached. Another thing that can happen is that the solution can return that the solver was unstable (exploded to infinity) or that `dt` became too small. If these happen, the first thing to do is to check that your model is correct. It could very well be that you made an error that causes the model to be unstable!

If the model is the problem, then stiffness could be the reason. We can thus hint to the solver to use an appropriate method:

```
sol = solve(prob,alg_hints = [:stiff])
```

```
retcode: Success
Interpolation: specialized 3rd order "free" stiffness-aware interpolation
t: 694-element Array{Float64,1}:
 0.0
 4.997501249375313e-10
 5.454105825317844e-9
 1.8954286226539402e-8
 4.149674379723465e-8
 7.308080698498873e-8
 1.1714649583268228e-7
 1.7481330012839592e-7
 2.4862371241875205e-7
 3.4025555832660864e-7
 ⋮
 5.69767949075004
 5.748994165486137
 5.811844321155623
 5.886853430367259
 5.969502584336209
 6.05645855489726
 6.143414525458311
 6.230370496019361
 6.3
u: 694-element Array{Array{Float64,1},1}:
 [0.0, 2.0]
 [-0.0009987513736106515, 1.9999999999997504]
 [-0.010819454588930516, 1.9999999999704143]
 [-0.036850919195836614, 1.999999999647449]
 [-0.07803540833511657, 1.999999998347307]
 [-0.13124866317048456, 1.9999999950290166]
 [-0.19755036285072544, 1.9999999877524572]
 [-0.272075415352352, 1.999999974149605]
 [-0.35045254633113243, 1.9999999510683737]
 [-0.4264538643666248, 1.9999999153142478]
 ⋮
 [0.6849948021041035, -1.9679959070285558]
 [0.7068255882516246, -1.9322949901761672]
 [0.7369247908644185, -1.8869463160135977]
 [0.7789756893010558, -1.8301403490903476]
 [0.8358041460218815, -1.7634992825515126]
 [0.9131711695745722, -1.6876171241799416]
 [1.0200095610067244, -1.6038403486733988]
 [1.182122272069454, -1.5086434776790882]
 [1.3982811580024197, -1.4194614700844543]
```

Or we can use the default algorithm. By default, DifferentialEquations.jl uses algorithms like `AutoTsit5(Rodas5())` which automatically detect stiffness and switch to an appropriate method once stiffness is known.

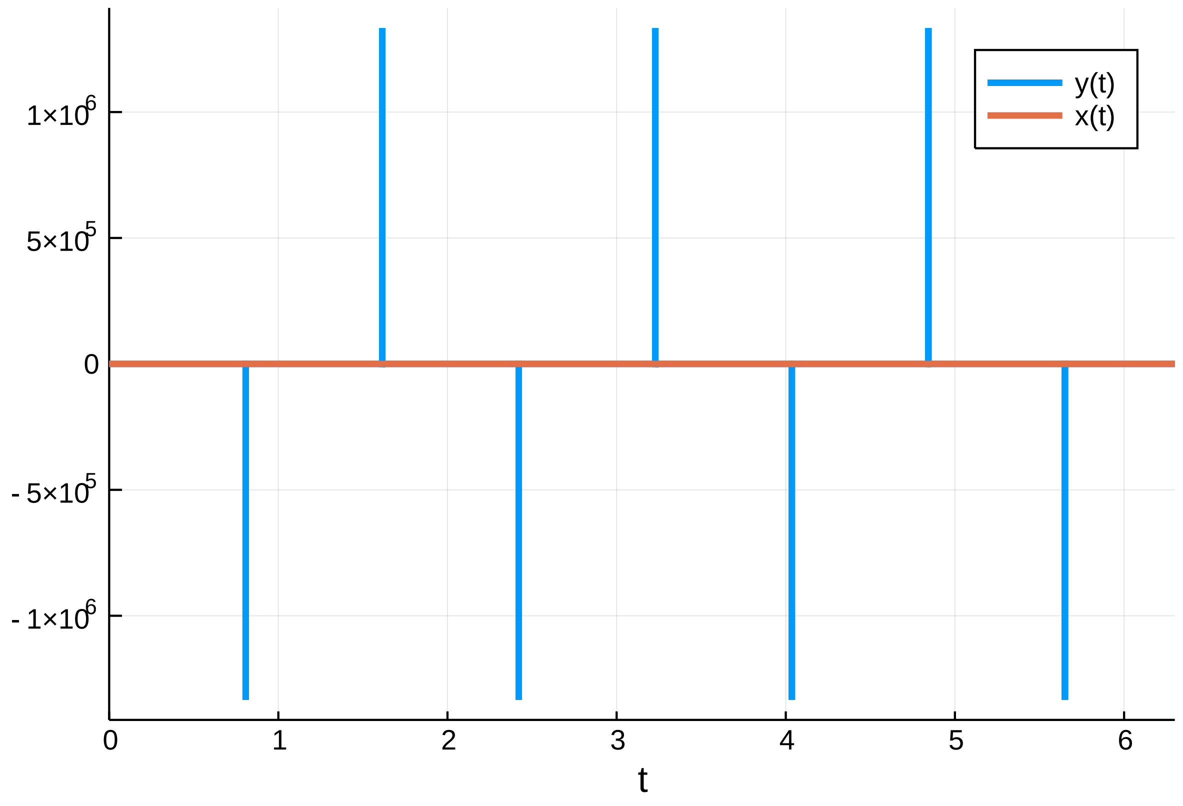
```
sol = solve(prob)
```

```
retcode: Success
Interpolation: Automatic order switching interpolation
t: 1927-element Array{Float64,1}:
 0.0
 4.997501249375313e-10
 5.4972513743128435e-9
 3.289919594544218e-8
 9.055581394883546e-8
 1.7309428803584187e-7
 2.79375393394586e-7
 4.149527171475212e-7
 5.807919390815544e-7
 7.81280701490125e-7
 ⋮
 6.204648226459174
 6.219556296846657
 6.233842035405188
 6.247504825256131
 6.260547615587699
 6.2729765212211905
 6.2848005774657025
 6.296031197826328
 6.3
```

```
u: 1927-element Array{Array{Float64,1},1}:
 [0.0, 2.0]
 [-0.0009987513736106552, 1.9999999999997504]
 [-0.010904339759596433, 1.9999999999699458]
 [-0.06265556194129239, 1.9999999989523902]
 [-0.1585948892562767, 1.9999999924944207]
 [-0.2700352862461109, 1.9999999746155703]
 [-0.3783197963325601, 1.9999999398563364]
 [-0.47467864703912216, 1.9999998815910678]
 [-0.5499302545937235, 1.999999796115446]
 [-0.6026934372089534, 1.9999996800439757]
 ⋮
 [1.1173199781760204, -1.5429755449491689]
 [1.1481789818936488, -1.526090225258629]
 [1.1805062564165951, -1.5094585545821166]
 [1.2143604506793437, -1.4931001271860491]
 [1.2498033047017658, -1.477032218841621]
 [1.286899775928202, -1.461269894062796]
 [1.3257186497126714, -1.4458259322647566]
 [1.3663322749653013, -1.4307111536240857]
 [1.3818948926590964, -1.4252572688140688]
```

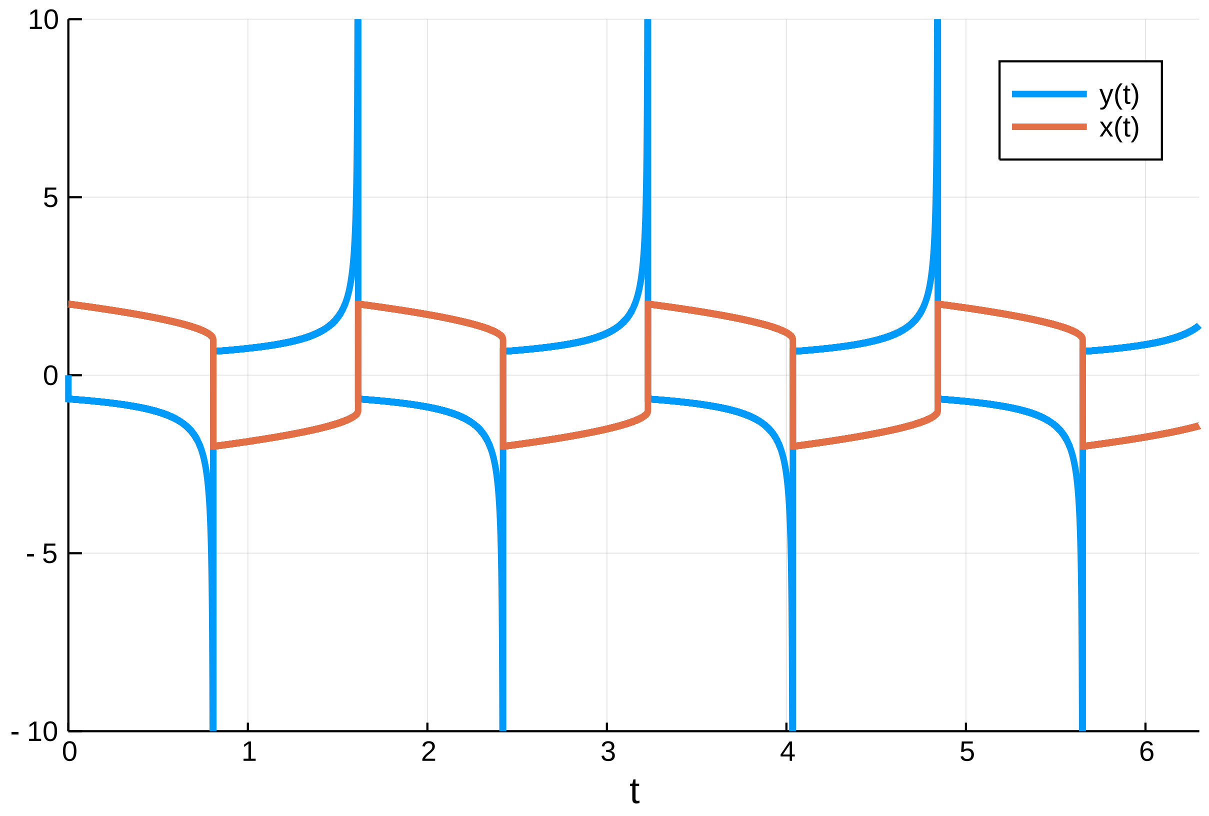
Another way to understand stiffness is to look at the solution.

```
using Plots; gr()
sol = solve(prob,alg_hints = [:stiff],reltol=1e-6)
plot(sol,denseplot=false)
```



Let's zoom in on the y-axis to see what's going on:

```
plot(sol,ylims = (-10.0,10.0))
```



Notice how there are some extreme vertical shifts that occur. These vertical shifts are places where the derivative term is very large, and this is indicative of stiffness. This is an extreme example to highlight the behavior, but this general idea can be carried over to your problem. When in doubt, simply try timing using both a stiff solver and a non-stiff solver and see which is more efficient.

To try this out, let's use BenchmarkTools, a package that let's us relatively reliably time code blocks.

```
function lorenz!(du,u,p,t)
    σ,ρ,β = p
    du[1] = σ*(u[2]-u[1])
    du[2] = u[1]*(ρ-u[3]) - u[2]
    du[3] = u[1]*u[2] - β*u[3]
end
u0 = [1.0,0.0,0.0]
p = (10,28,8/3)
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan,p)
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 100.0)
u0: [1.0, 0.0, 0.0]
```

And now, let's use the `@btime` macro from benchmark tools to compare the use of non-stiff and stiff solvers on this problem.

```
using BenchmarkTools
@btime solve(prob);
```

```
798.200 μs (13116 allocations: 1.42 MiB)
```

```
@btime solve(prob,alg_hints = [:stiff]);
```

```
10.689 ms (74459 allocations: 3.11 MiB)
```

In this particular case, we can see that non-stiff solvers get us to the solution much more quickly.

0.2 The Recommended Methods

When picking a method, the general rules are as follows:

- Higher order is more efficient at lower tolerances, lower order is more efficient at higher tolerances
- Adaptivity is essential in most real-world scenarios
- Runge-Kutta methods do well with non-stiff equations, Rosenbrock methods do well with small stiff equations, BDF methods do well with large stiff equations

While there are always exceptions to the rule, those are good guiding principles. Based on those, a simple way to choose methods is:

- The default is `Tsit5()`, a non-stiff Runge-Kutta method of Order 5
- If you use low tolerances (`1e-8`), try `Vern7()` or `Vern9()`
- If you use high tolerances, try `BS3()`
- If the problem is stiff, try `Rosenbrock23()`, `Rodas5()`, or `CVODE_BDF()`
- If you don't know, use `AutoTsit5(Rosenbrock23())` or `AutoVern9(Rodas5())`.

(This is a simplified version of the default algorithm chooser)

0.3 Comparison to other Software

If you are familiar with MATLAB, SciPy, or R's DESolve, here's a quick translation start to have transfer your knowledge over.

- `ode23` -> `BS3()`
- `ode45`/`dopri5` -> `DP5()`, though in most cases `Tsit5()` is more efficient
- `ode23s` -> `Rosenbrock23()`, though in most cases `Rodas4()` is more efficient
- `ode113` -> `VCABM()`, though in many cases `Vern7()` is more efficient
- `dop853` -> `DP8()`, though in most cases `Vern7()` is more efficient
- `ode15s`/`vode` -> `QNDF()`, though in many cases `CVODE_BDF()`, `Rodas4()` or `radau()` are more efficient
- `ode23t` -> `Trapezoid()` for efficiency and `GenericTrapezoid()` for robustness

- `ode23tb` -> `TRBDF2`
- `lsoda` -> `lsoda()` (requires `]add LSODA; using LSODA`)
- `ode15i` -> `IDA()`, though in many cases `Rodas4()` can handle the DAE and is significantly more efficient

0.4 Appendix

This tutorial is part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("introduction","02-choosing_algs.jmd")
```

Computer Information:

```
Julia Version 1.3.0
Commit 46ce4d7933 (2019-11-26 06:09 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.42.0\atom.exe"  -a
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `~\.julia\dev\DiffEqTutorials\Project.toml`
```