Feagin's Order 10, 12, and 14 Methods

Chris Rackauckas

February 26, 2019

DifferentialEquations.jl includes Feagin's explicit Runge-Kutta methods of orders 10/8, 12/10, and 14/12. These methods have such high order that it's pretty much required that one uses numbers with more precision than Float64. As a prerequisite reference on how to use arbitrary number systems (including higher precision) in the numerical solvers, please see the Solving Equations in With Chosen Number Types notebook.

0.1 Investigation of the Method's Error

We can use Feagin's order 16 method as follows. Let's use a two-dimensional linear ODE. Like in the Solving Equations in With Chosen Number Types notebook, we change the initial condition to BigFloats to tell the solver to use BigFloat types.

```
using DifferentialEquations
const linear_bigα = big(1.01)
f(u,p,t) = (linear_bigα*u)

# Add analytical solution so that errors are checked
f_analytic(u0,p,t) = u0*exp(linear_bigα*t)
ff = ODEFunction(f,analytic=f_analytic)
prob = ODEProblem(ff,big(0.5),(0.0,1.0))
sol = solve(prob,Feagin14(),dt=1//16,adaptive=false);
```

```
println(sol.errors)
```

```
Dict(:l∞=>2.19751e-23,:final=>2.19751e-23,:l2=>1.0615e-23)
```

Compare that to machine ϵ for Float64:

```
eps(Float64)
```

```
2.220446049250313e-16
```

The error for Feagin's method when the stepsize is 1/16 is 8 orders of magnitude below machine ϵ! However, that is dependent on the stepsize. If we instead use adaptive timestepping with the default tolerances, we get

```
sol =solve(prob,Feagin14());
println(sol.errors); print("The length was $(length(sol))")
```

```
Dict(:l∞=>1.54574e-09,:final=>1.54574e-09,:l2=>8.92507e-10)
The length was 3
```

Notice that when the stepsize is much higher, the error goes up quickly as well. These super high order methods are best when used to gain really accurate approximations (using still modest timesteps). Some examples of where such precision is necessary is astrodynamics where the many-body problem is highly chaotic and thus sensitive to small errors.

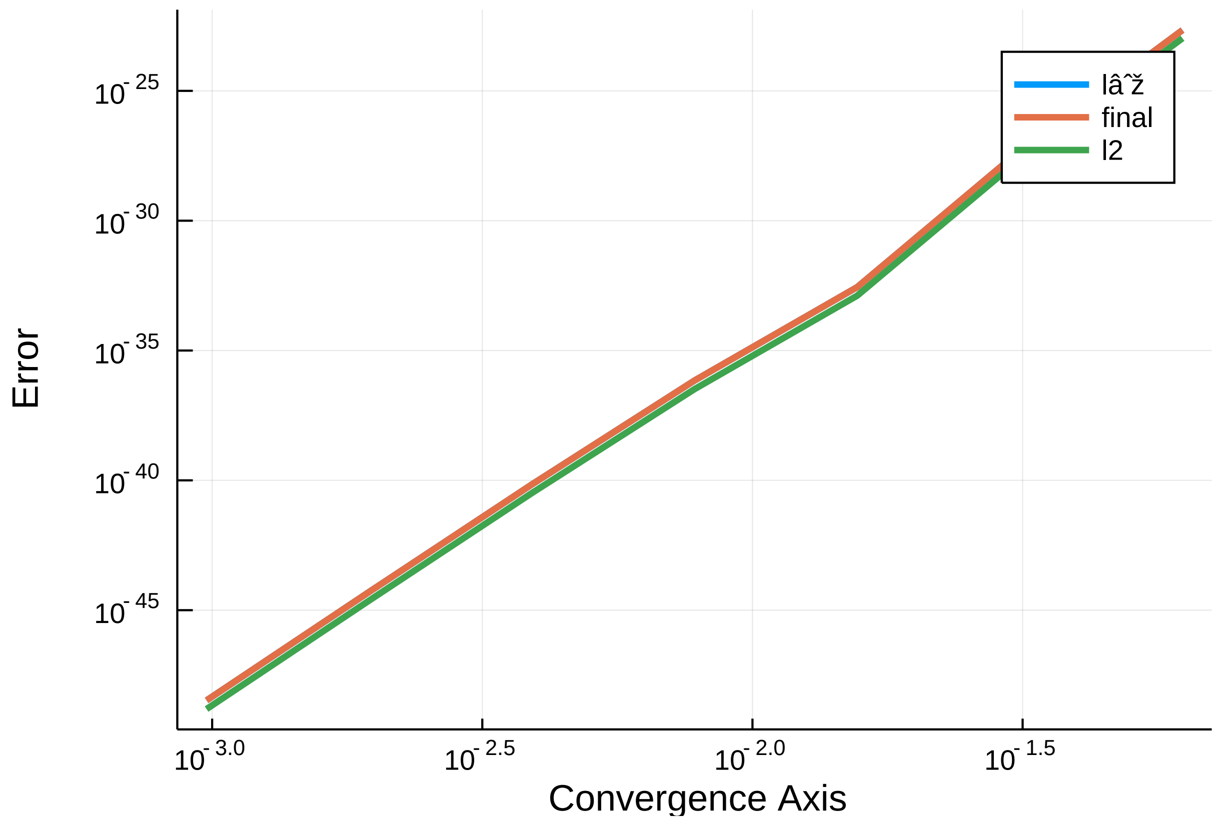
0.2 Convergence Test

The Order 14 method is awesome, but we need to make sure it's really that awesome. The following convergence test is used in the package tests in order to make sure the implementation is correct. Note that all methods have such tests in place.

```
using DiffEqDevTools
dts = 1.0 ./ 2.0 .^(10:-1:4)
sim = test_convergence(dts,prob,Feagin14())
```

For a view of what's going on, let's plot the simulation results.

```
using Plots
gr()
plot(sim)
```



This is a clear trend indicating that the convergence is truly Order 14, which is the estimated slope.

0.3 Appendix

```
using DiffEqTutorials
DiffEqTutorials.tutorial_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])
```

```
These benchmarks are part of the DiffEqTutorials.jl repository, found at:

https://github.com/JuliaDiffEq/DiffEqTutorials.jl

To locally run this tutorial, do the following commands:

using DiffEqTutorials
DiffEqTutorials.weave_file("ode_extras","feagin.jmd")

Computer Information:

Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.34.0\atom.exe" -a
  JULIA_NUM_THREADS = 6

Package Information:

    Status `C:\Users\accou\.julia\environments\v1.1\Project.toml`
  [7e558dbc] ArbNumerics v0.3.6
  [c52e3926] Atom v0.7.14
  [6e4b80f9] BenchmarkTools v0.4.2
  [336ed68f] CSV v0.4.3
  [3895d2a7] CUDAapi v0.5.4
  [be33ccc6] CUDAnative v1.0.1
  [3a865a2d] CuArrays v0.9.1
  [a93c6f00] DataFrames v0.17.1
  [55939f99] DecFP v0.4.8
  [abce61dc] Decimals v0.4.0
  [39dd38d3] Dierckx v0.4.1
  [bb2cbb15] DiffEqBenchmarks v0.0.0 [`C:\Users\accou\.julia\external\DiffE
qBenchmarks.jl`]
  [459566f4] DiffEqCallbacks v2.5.2
  [f3b72e0c] DiffEqDevTools v2.6.1
  [aae7a2af] DiffEqFlux v0.2.0
  [c894b116] DiffEqJump v6.1.0+ [`C:\Users\accou\.julia\dev\DiffEqJump`]
  [1130ab10] DiffEqParamEstim v1.6.0+ [`C:\Users\accou\.julia\dev\DiffEqPar
amEstim`]
  [055956cb] DiffEqPhysics v3.1.0
  [a077e3f3] DiffEqProblemLibrary v4.1.0
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.jl`]
  [0c46a032] DifferentialEquations v6.3.0
  [497a8b3b] DoubleFloats v0.7.5
  [587475ba] Flux v0.7.3
  [f6369f11] ForwardDiff v0.10.3+ [`C:\Users\accou\.julia\dev\ForwardDiff`]
  [28b8d3ca] GR v0.38.1
  [7073ff75] IJulia v1.17.0
  [c601a237] Interact v0.9.1
  [b6b21f68] Ipopt v0.5.4
  [4076af6c] JuMP v0.19.0
  [e5e0dc1b] Juno v0.5.4
  [7f56f5a3] LSODA v0.4.0
  [eff96d63] Measurements v2.0.0
  [76087f3c] NLopt v0.5.1
  [c030b06c] ODE v2.4.0
  [54ca160b] ODEInterface v0.4.5+ [`C:\Users\accou\.julia\dev\ODEInterface`
]
  [09606e27] ODEInterfaceDiffEq v3.0.0
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.2.1+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.1
  [91a5bcdd] Plots v0.23.0
  [71ad9d73] PuMaS v0.0.0 [`C:\Users\accou\.julia\dev\PuMaS`]
  [d330b81b] PyPlot v2.7.0
  [731186ca] RecursiveArrayTools v0.20.0
  [90137ffa] StaticArrays v0.10.2
  [789caeaf] StochasticDiffEq v6.1.1+ [`C:\Users\accou\.julia\dev\Stochasti
cDiffEq`]
  [c3572dad] Sundials v3.0.0
  [1986cc42] Unitful v0.14.0
  [2a06ce6d] UnitfulPlots v0.0.0 #master (https://github.com/ajkeller34/Uni
tfulPlots.jl)
  [44d3d7a6] Weave v0.7.1 [`C:\Users\accou\.julia\dev\Weave`]
```