Three Body Work-Precision Diagrams

Chris Rackauckas

June 12, 2019

```
using OrdinaryDiffEq, ODE, ODEInterfaceDiffEq, LSODA, Sundials, DiffEqDevTools, Plots;
    gr()

## Define the ThreeBody Problem
const threebody_μ = parse(Float64,"0.012277471")
const threebody_μ′ = 1 - threebody_μ

f = (du,u,p,t) -> begin
  @inbounds begin
  # 1 = y_1
  # 2 = y_2
  # 3 = y_1'
  # 4 = y_2'
  D_1 = ((u[1]+threebody_μ)^2 + u[2]^2)^(3/2)
  D_2 = ((u[1]-threebody_μ′)^2 + u[2]^2)^(3/2)
  du[1] = u[3]
  du[2] = u[4]
  du[3] = u[1] + 2u[4] - threebody_μ′*(u[1]+threebody_μ)/D_1 -
    threebody_μ*(u[1]-threebody_μ′)/D_2
  du[4] = u[2] - 2u[3] - threebody_μ′*u[2]/D_1 - threebody_μ*u[2]/D_2
  end
end

t_0 = 0.0; T = parse(Float64,"17.0652165601579625588917206249")
tspan = (t_0,2T)

prob = ODEProblem(f,[0.994, 0.0, 0.0,
    parse(Float64,"-2.00158510637908252240537862224")],tspan)

test_sol = TestSolution(T,[prob.u0])
abstols = 1.0 ./ 10.0 .^ (3:13); reltols = 1.0 ./ 10.0 .^ (0:10);
```

See that it's periodic in the chosen timespan:

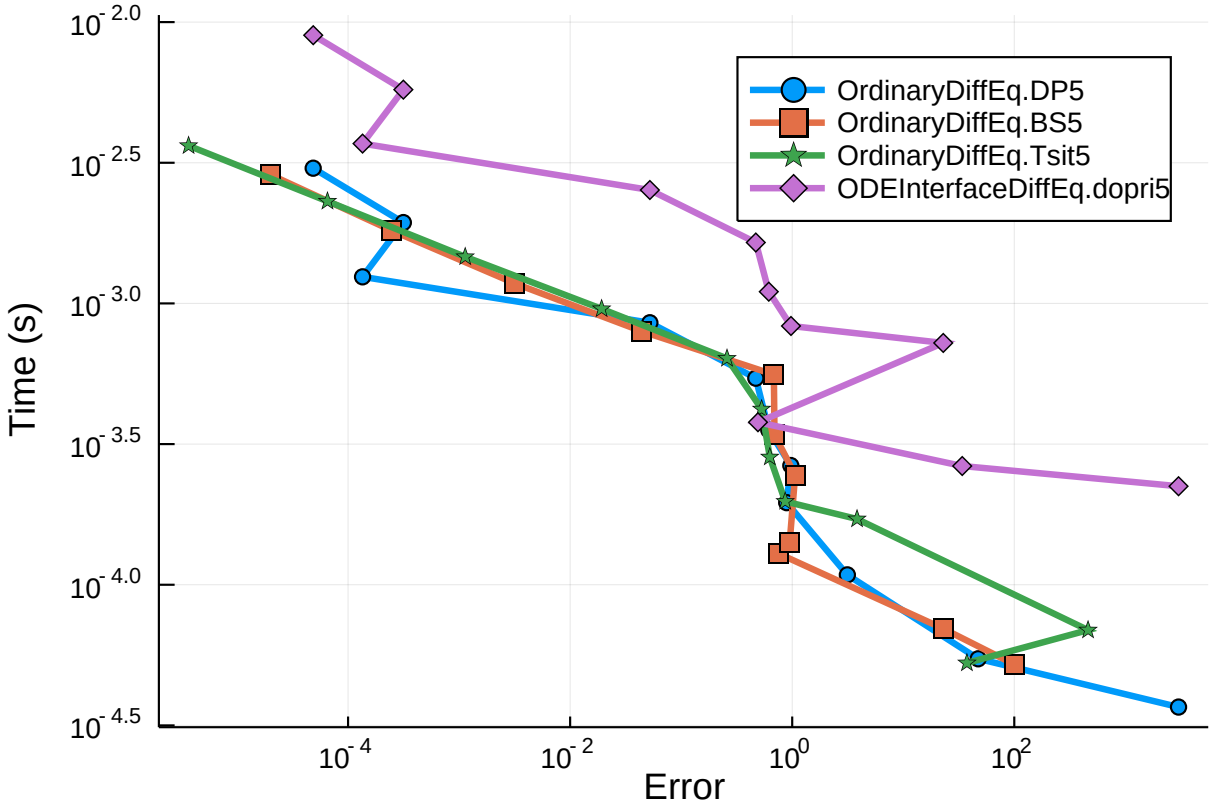
```
sol = solve(prob,Vern9(),abstol=1e-14,reltol=1e-14)
@show sol[1] - sol[end]

sol[1] - sol[end] = [-5.52535e-11, -1.64275e-10, -2.68023e-8, -8.60003e-9]

@show sol[end] - prob.u0;

sol[end] - prob.u0 = [5.52535e-11, 1.64275e-10, 2.68023e-8, 8.60003e-9]

apr = appxtrue(sol,test_sol)
@show sol[end]
```

```
sol[end] = [0.994, 1.64275e-10, 2.68023e-8, -2.00159]

@show apr.u[end]

apr.u[end] = [0.994, 1.64275e-10, 2.68023e-8, -2.00159]

@show apr.errors

apr.errors = Dict(:final=>8.90547e-9)
Dict{Symbol,Float64} with 1 entry:
  :final => 8.90547e-9
```

This three-body problem is known to be a tough problem. Let's see how the algorithms fair at standard tolerances.

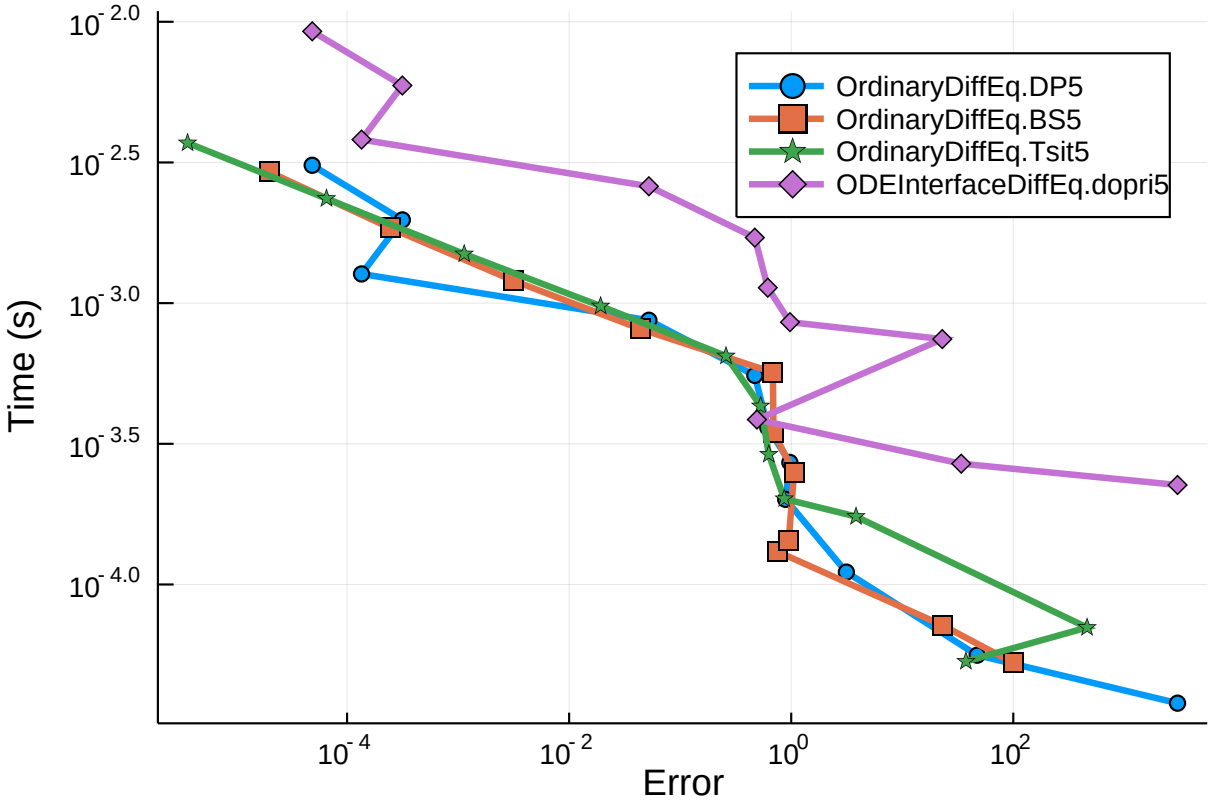
0.0.1 5th Order Runge-Kutta Methods

```
setups = [Dict(:alg=>DP5())
          #Dict(:alg=>ode45()) #fails
          Dict(:alg=>BS5())
          Dict(:alg=>Tsit5())
          Dict(:alg=>dopri5())];
wp =
    WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,save_everystep=false,numruns=100)
plot(wp)
```

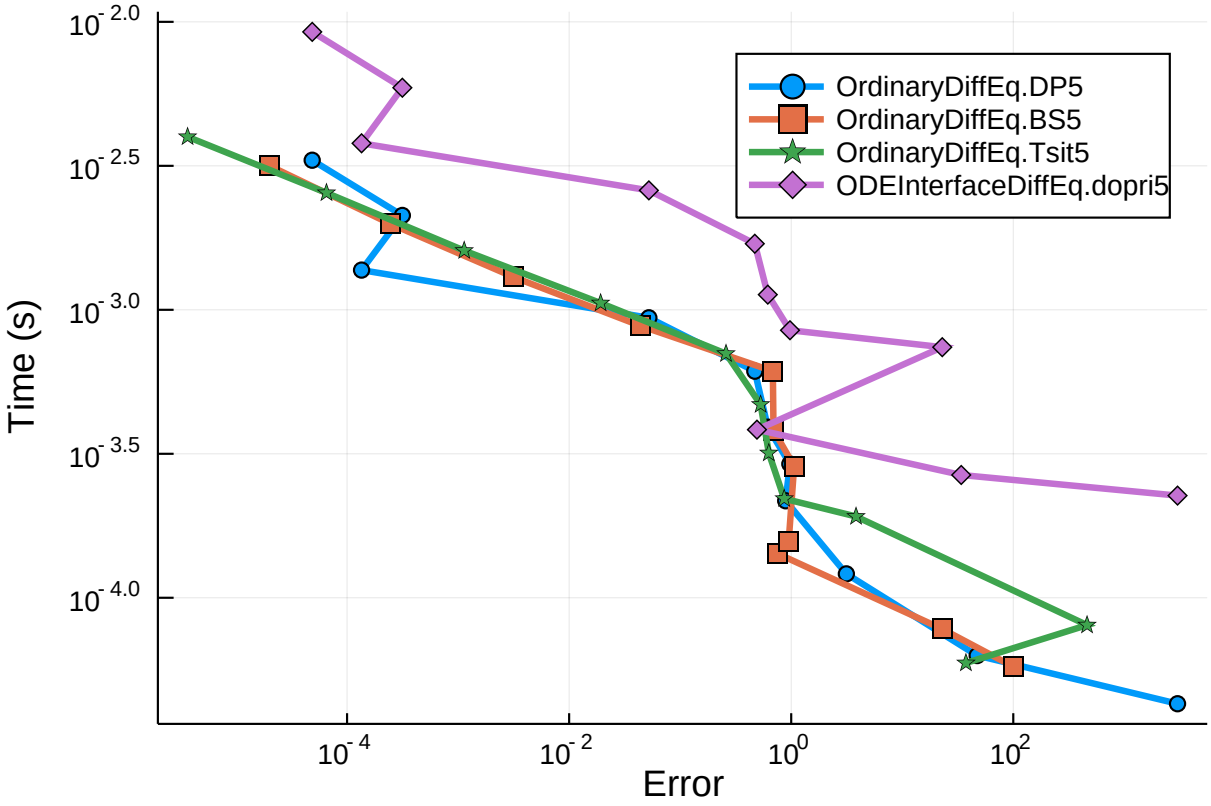



```
setups = [Dict(:alg=>DP5(),:dense=>false)
          #Dict(:alg=>ode45()) # Fails
          Dict(:alg=>BS5(),:dense=>false)
          Dict(:alg=>Tsit5(),:dense=>false)
          Dict(:alg=>dopri5())];
```

```
wp = WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,numruns=100)
plot(wp)
```



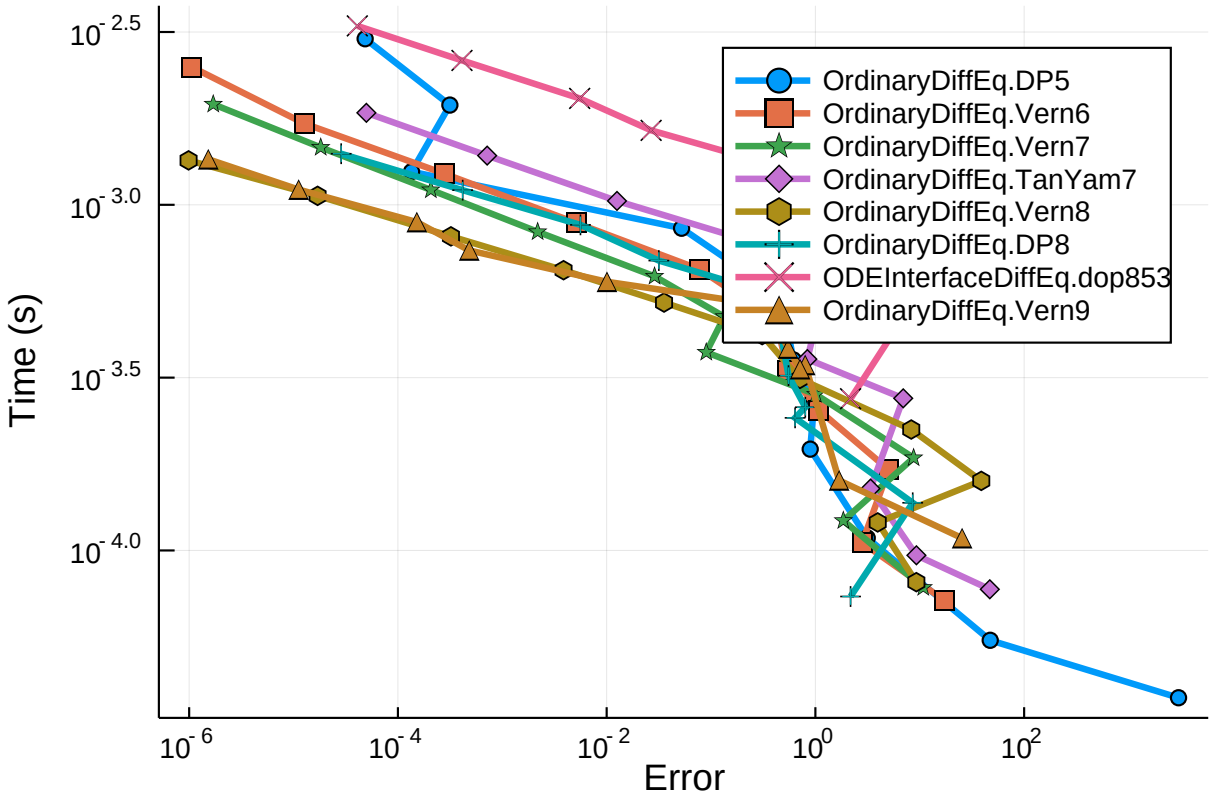

```
setups = [Dict(:alg=>DP5())
          #Dict(:alg=>ode45()) #fails
          Dict(:alg=>BS5())
          Dict(:alg=>Tsit5())
          Dict(:alg=>dopri5())];
wp = WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,numruns=100)
plot(wp)
```



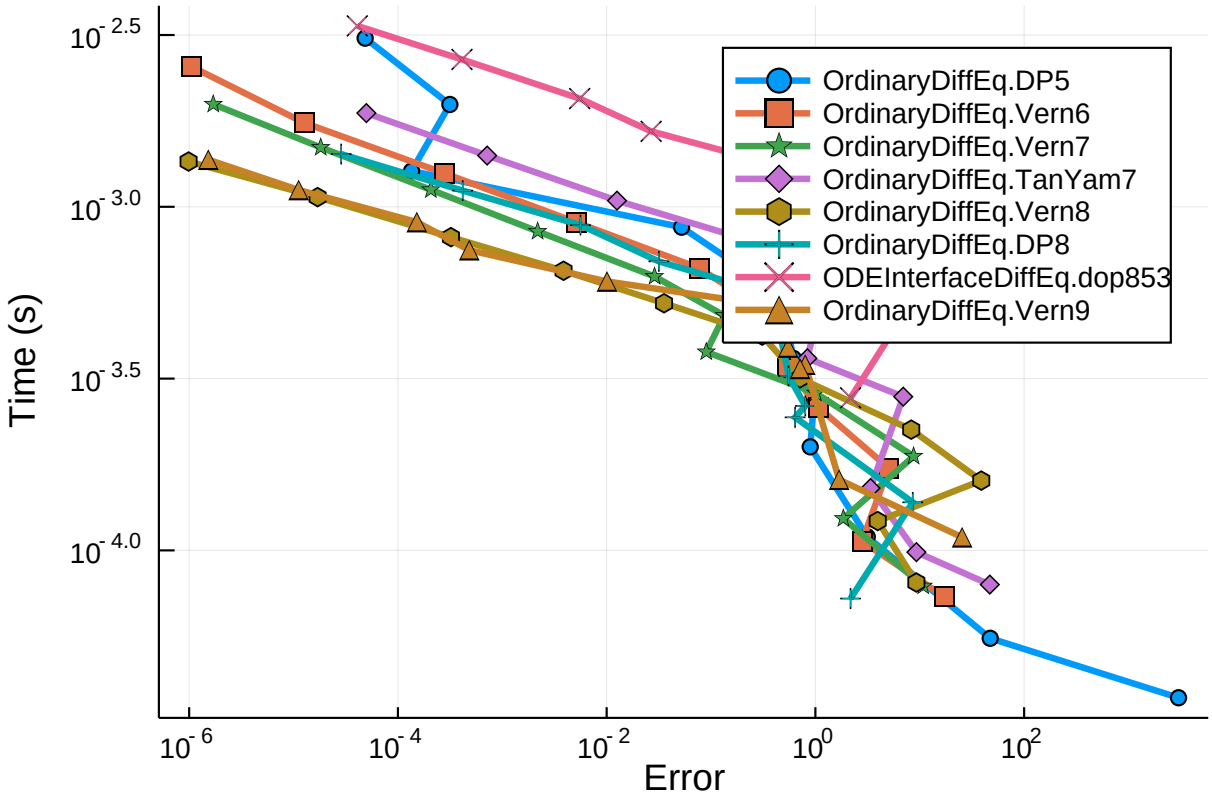
In these tests we see that most of the algorithms are close,with `BS5` and `DP5` showing much better than `Tsit5`. `ode45` errors.

0.0.2 Higher Order Algorithms

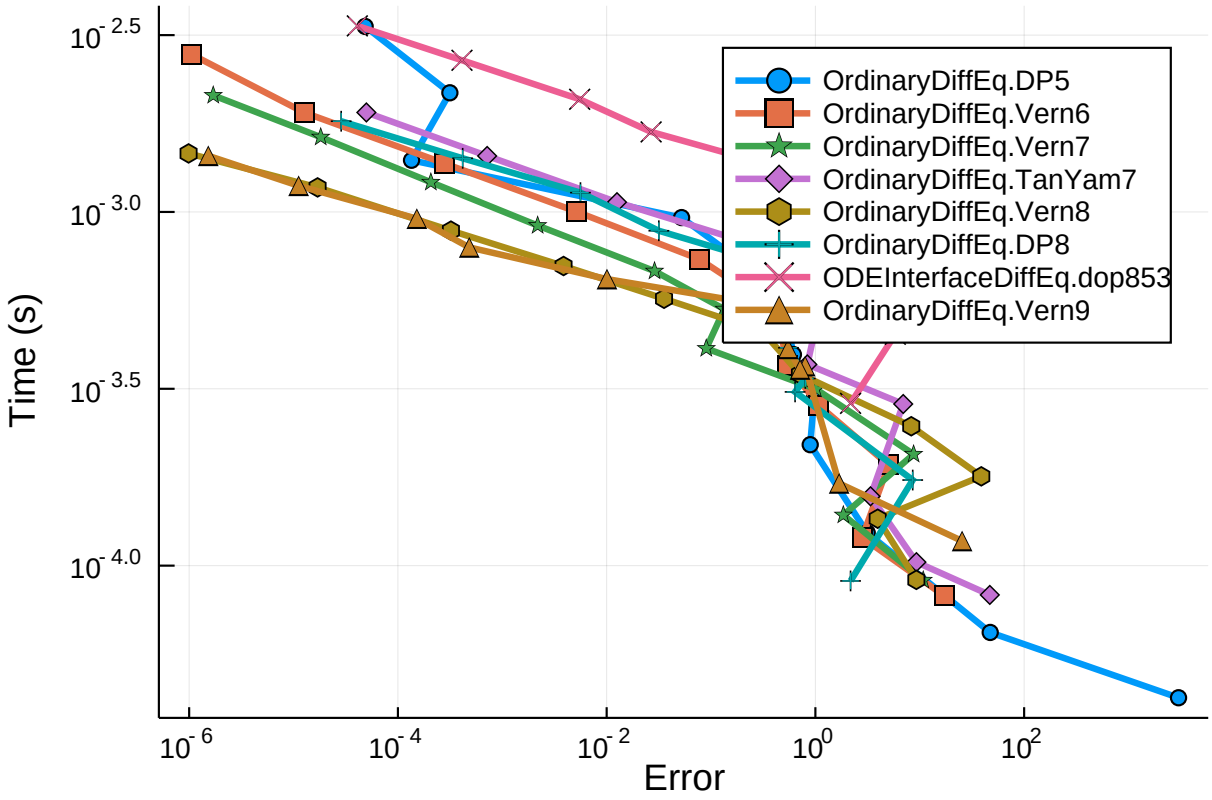
```
setups = [Dict(:alg=>DP5())
          Dict(:alg=>Vern6())
          Dict(:alg=>Vern7())
          Dict(:alg=>TanYam7())
          Dict(:alg=>Vern8())
          Dict(:alg=>DP8())
          Dict(:alg=>dop853())
          Dict(:alg=>Vern9())];
wp =
    WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,save_everystep=false,numruns=100)
plot(wp)
```




```
setups = [Dict(:alg=>DP5())
          Dict(:alg=>Vern6())
          Dict(:alg=>Vern7())
          Dict(:alg=>TanYam7())
          Dict(:alg=>Vern8())
          Dict(:alg=>DP8())
          Dict(:alg=>dop853())
          Dict(:alg=>Vern9())];
wp =
    WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,dense=false,numruns=100,verbose=fals
plot(wp)
```




```
setups = [Dict(:alg=>DP5())
          Dict(:alg=>Vern6())
          Dict(:alg=>Vern7())
          Dict(:alg=>TanYam7())
          Dict(:alg=>Vern8())
          Dict(:alg=>DP8())
          Dict(:alg=>dop853())
          Dict(:alg=>Vern9())];
wp = WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,numruns=100)
plot(wp)
```

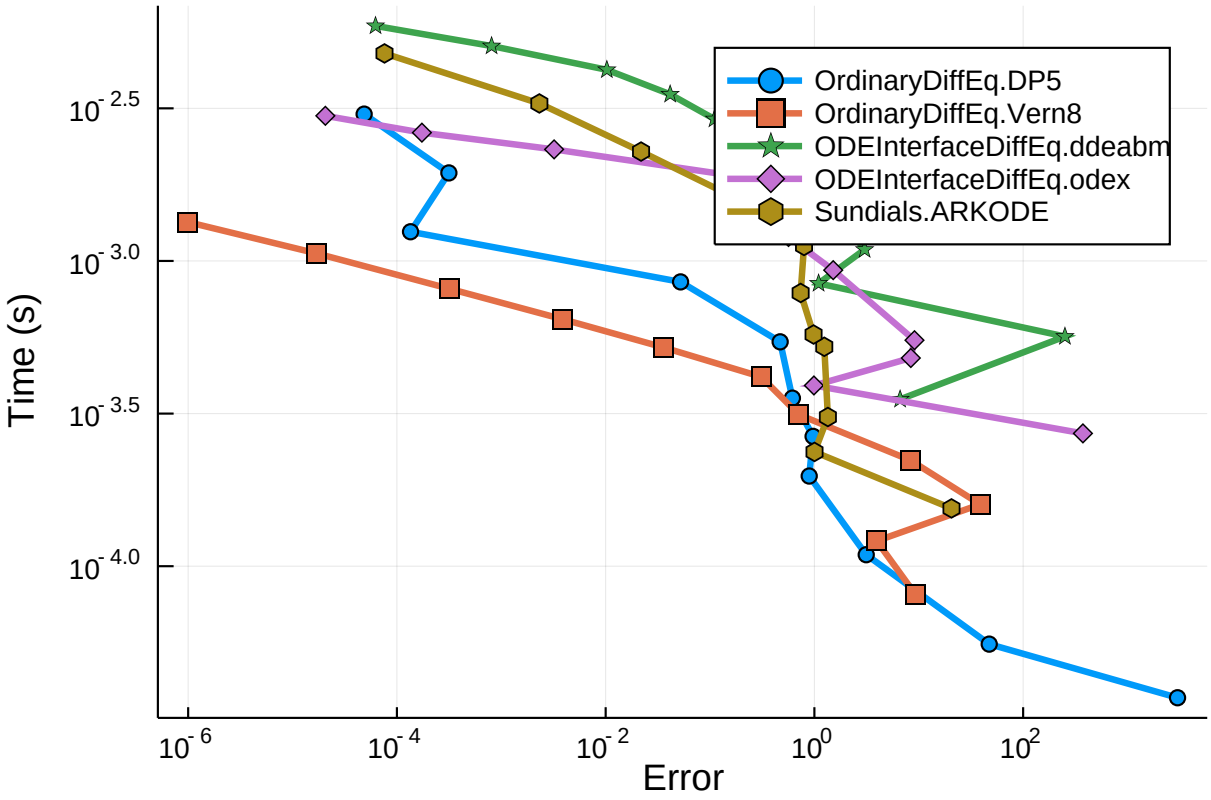


In this test we see `Vern7` and `Vern8` shine.

0.0.3 Other Algorithms

Once again we separate ODE.jl because it fails. We also separate Sundials' `CVODE_Adams` since it fails at high tolerances.

```
#setups = [Dict(:alg=>ode78())
# Dict(:alg=>CVODE_Adams())];
#wp =
    WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,dense=false,numruns=100)

setups = [Dict(:alg=>DP5())
          #Dict(:alg=>lsoda())
          Dict(:alg=>Vern8())
          Dict(:alg=>ddeabm())
          Dict(:alg=>odex())
          Dict(:alg=>ARKODE(Sundials.Explicit(),order=6))
    ];
wp =
    WorkPrecisionSet(prob,abstols,reltols,setups;appxsol=test_sol,save_everystep=false,numruns=100)
plot(wp)
```



Again, on cheap function calculations the Adams methods are shown to not be efficient once the error is sufficiently small. Also, as seen in other places, the extrapolation methods do not fare as well as the Runge-Kutta methods.

0.0.4 Conclusion

As in the other tests, the OrdinaryDiffEq.jl algorithms with the Verner Efficient methods are the most efficient solvers at stringent tolerances for most of the tests, while the order 5 methods do well at cruder tolerances. ODE.jl fails to run the test problems without erroring.

```
using DiffEqBenchmarks
DiffEqBenchmarks.bench_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])
```

0.1 Appendix

These benchmarks are a part of the DiffEqBenchmarks.jl repository, found at: https://github.com/JuliaDi

To locally run this tutorial, do the following commands:

```
using DiffEqBenchmarks
DiffEqBenchmarks.weave_file("NonStiffODE","ThreeBody_wpd.jmd")
```

Computer Information:

```
Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, haswell)
```

Package Information:

```
Status: `/home/crackauckas/.julia/environments/v1.1/Project.toml`
[c52e3926-4ff0-5f6e-af25-54175e0327b1] Atom 0.8.7
[bcd4f6db-9728-5f36-b5f7-82caef46ccdb] DelayDiffEq 5.3.0
[bb2cbb15-79fc-5d1e-9bf1-8ae49c7c1650] DiffEqBenchmarks 0.1.0
[459566f4-90b8-5000-8ac3-15dfb0a30def] DiffEqCallbacks 2.5.2
[f3b72e0c-5b89-59e1-b016-84e28bfd966d] DiffEqDevTools 2.8.0
[aae7a2af-3d4f-5e19-a356-7da93b79d9d0] DiffEqFlux 0.5.0
[78ddff82-25fc-5f2b-89aa-309469cbf16f] DiffEqMonteCarlo 0.14.0
[77a26b50-5914-5dd7-bc55-306e6241c503] DiffEqNoiseProcess 3.3.1
[9fdde737-9c7f-55bf-ade8-46b3f136cc48] DiffEqOperators 3.5.0
[055956cb-9e8b-5191-98cc-73ae4a59e68a] DiffEqPhysics 3.1.0
[a077e3f3-b75c-5d7f-a0c6-6bc4c8ec64a9] DiffEqProblemLibrary 4.1.0
[41bf760c-e81c-5289-8e54-58b1f1f8abe2] DiffEqSensitivity 3.2.2
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.4.0
[b305315f-e792-5b7a-8f41-49f472929428] Elliptic 0.5.0
[587475ba-b771-5e3f-ad9e-33799f191a9c] Flux 0.8.3
[e5e0dc1b-0480-54bc-9374-aad01c23163d] Juno 0.7.0
[7f56f5a3-f504-529b-bc02-0b1fe5e64312] LSODA 0.4.0
[c030b06c-0b6d-57c2-b091-7029874bd033] ODE 2.4.0
[54ca160b-1b9f-5127-a996-1867f4bc2a2c] ODEInterface 0.4.5
[09606e27-ecf5-54fc-bb29-004bd9f985bf] ODEInterfaceDiffEq 3.3.0
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.8.1
[2dcacdae-9679-587a-88bb-8b444fb7085b] ParallelDataTransfer 0.5.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 4.1.1
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 0.25.1
[d330b81b-6aea-500a-939a-2ce795aea3ee] PyPlot 2.8.1
[731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 0.20.0
[295af30f-e4ad-537b-8983-00126c2a3abe] Revise 2.1.6
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.11.0
[789caeaf-c7a9-5a7d-9973-96adeb23e2a0] StochasticDiffEq 6.2.0
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 3.6.0
[92b13dbe-c966-51a2-8445-caca9f8a7d42] TaylorIntegration 0.5.0
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.0
[e88e6eb3-aa80-5325-afca-941959d7151f] Zygote 0.3.1
```