VanDerPol Work-Precision Diagrams

Chris Rackauckas

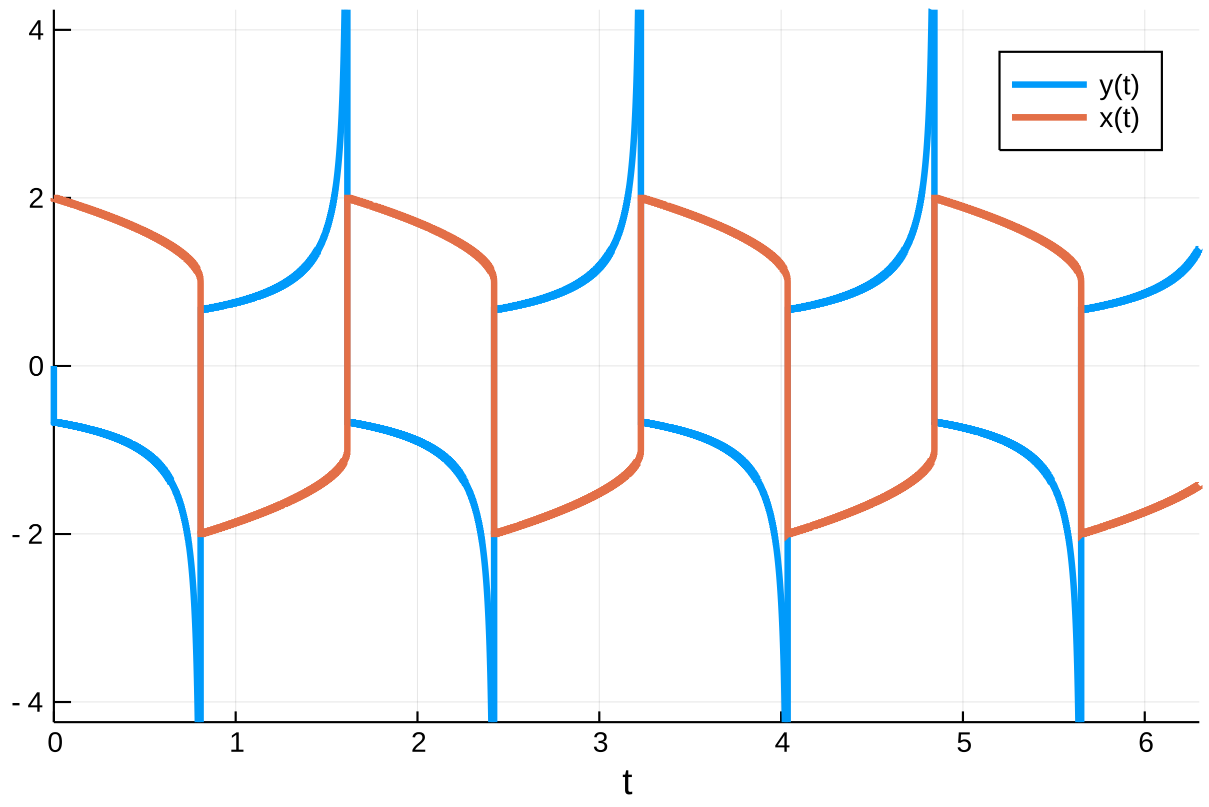
July 9, 2019

```
using OrdinaryDiffEq, DiffEqDevTools, Sundials, ParameterizedFunctions, Plots, ODE,
    ODEInterfaceDiffEq, ODEInterface, LSODA
gr()
using LinearAlgebra
LinearAlgebra.BLAS.set_num_threads(1)

van = @ode_def begin
  dy = μ*((1-x^2)*y - x)
  dx = 1*y
end μ

prob = ODEProblem(van,[0;2.],(0.0,6.3),1e6)
abstols = 1.0 ./ 10.0 .^ (5:9)
reltols = 1.0 ./ 10.0 .^ (2:6)

sol = solve(prob,CVODE_BDF(),abstol=1/10^14,reltol=1/10^14)
test_sol = TestSolution(sol)
```

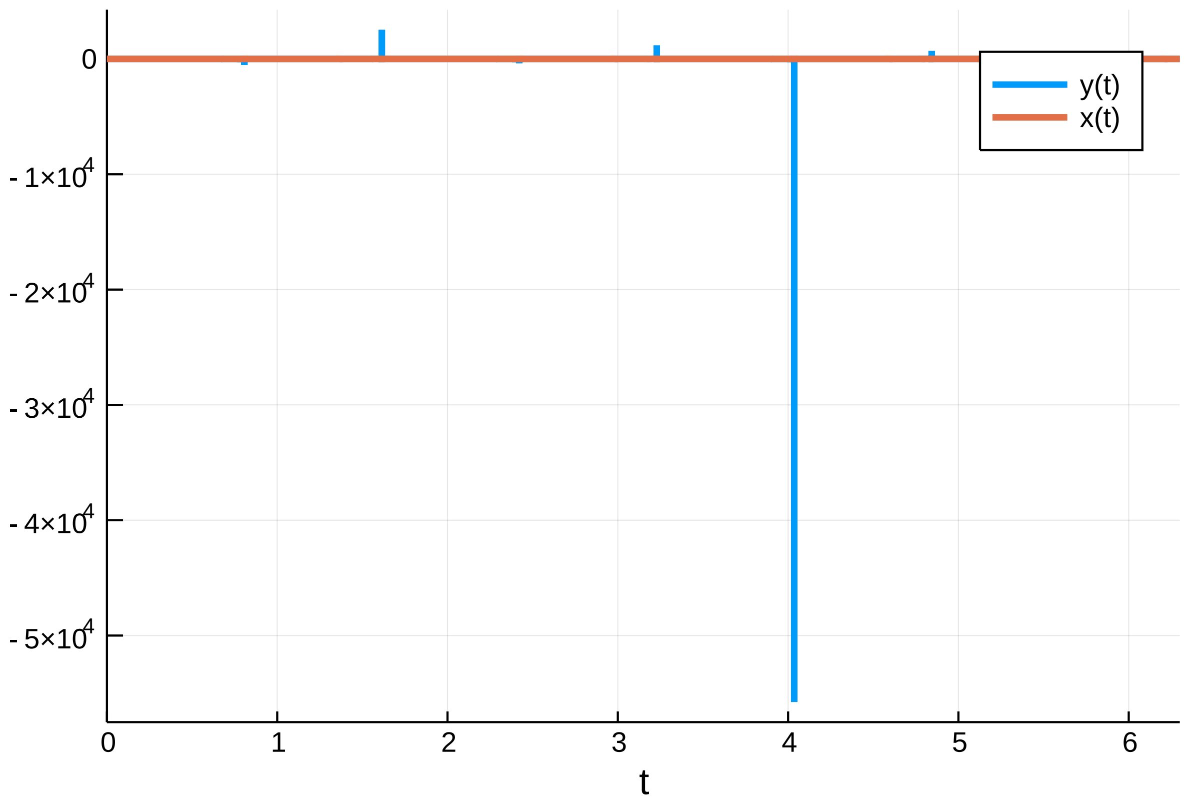
```
retcode: Success
Interpolation: 3rd order Hermite
t: nothing
u: nothing
```

0.0.1 Plot Test

```
plot(sol,ylim=[-4;4])
```



```
plot(sol)
```



0.1 Omissions And Tweaking

The following were omitted from the tests due to convergence failures. ODE.jl's adaptivity is not able to stabilize its algorithms, while GeometricIntegratorsDiffEq has not upgraded to Julia 1.0. GeometricIntegrators.jl's methods used to be either fail to converge at comparable dts (or on some computers errors due to type conversions).

```
#sol = solve(prob,ode23s()); println("Total ODE.jl steps:  $(length(sol))")
#using GeometricIntegratorsDiffEq
#try
# sol = solve(prob,GIRadIIA3(),dt=1/1000)
#catch e
# println(e)
#end
```

`ARKODE` needs a lower `nonlinear_convergence_coefficient` in order to not diverge.

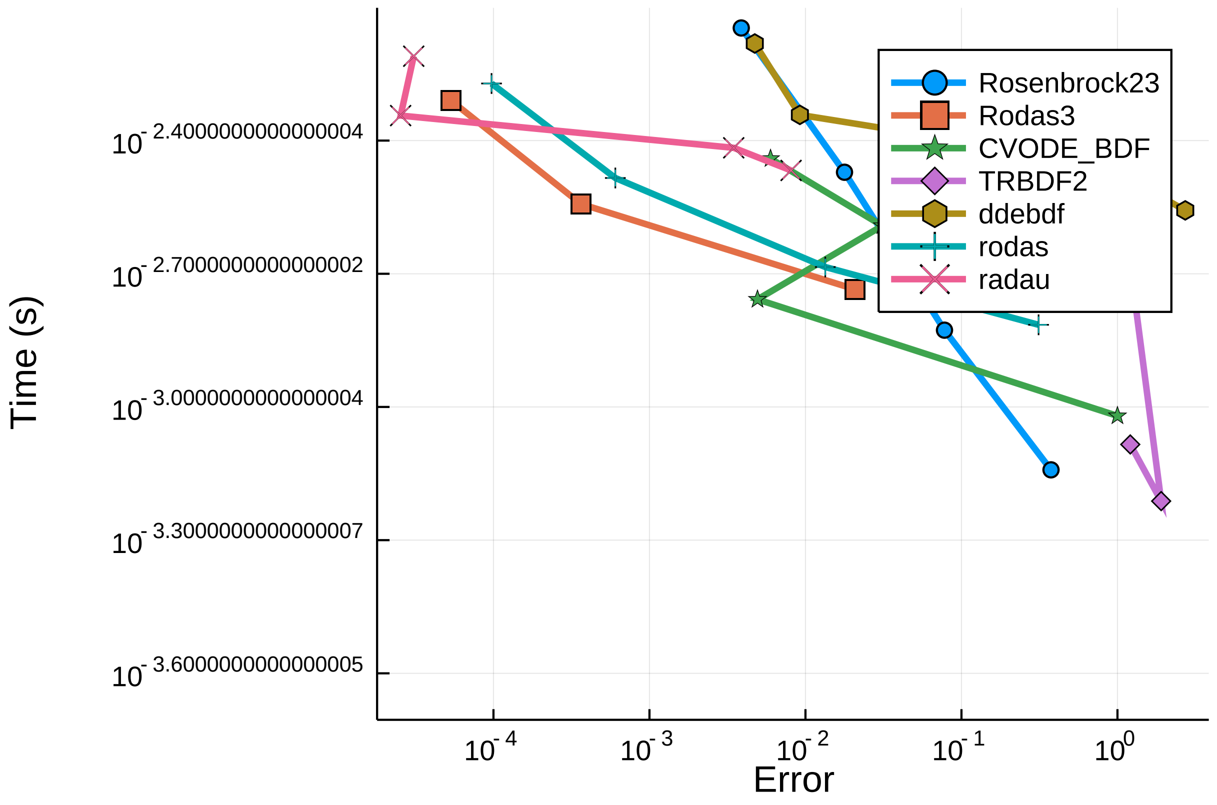
```
sol = solve(prob,ARKODE(),abstol=1e-4,reltol=1e-2);

sol = solve(prob,ARKODE(nonlinear_convergence_coefficient =
    1e-6),abstol=1e-4,reltol=1e-1);

sol = solve(prob,ARKODE(order=3),abstol=1e-4,reltol=1e-1);

sol = solve(prob,ARKODE(nonlinear_convergence_coefficient =
    1e-6,order=3),abstol=1e-4,reltol=1e-1);

sol = solve(prob,ARKODE(order=5,nonlinear_convergence_coefficient =
    1e-3),abstol=1e-4,reltol=1e-1);

sol = solve(prob,ARKODE(order=5,nonlinear_convergence_coefficient =
    1e-4),abstol=1e-4,reltol=1e-1);
```

0.2 Low Order and High Tolerance

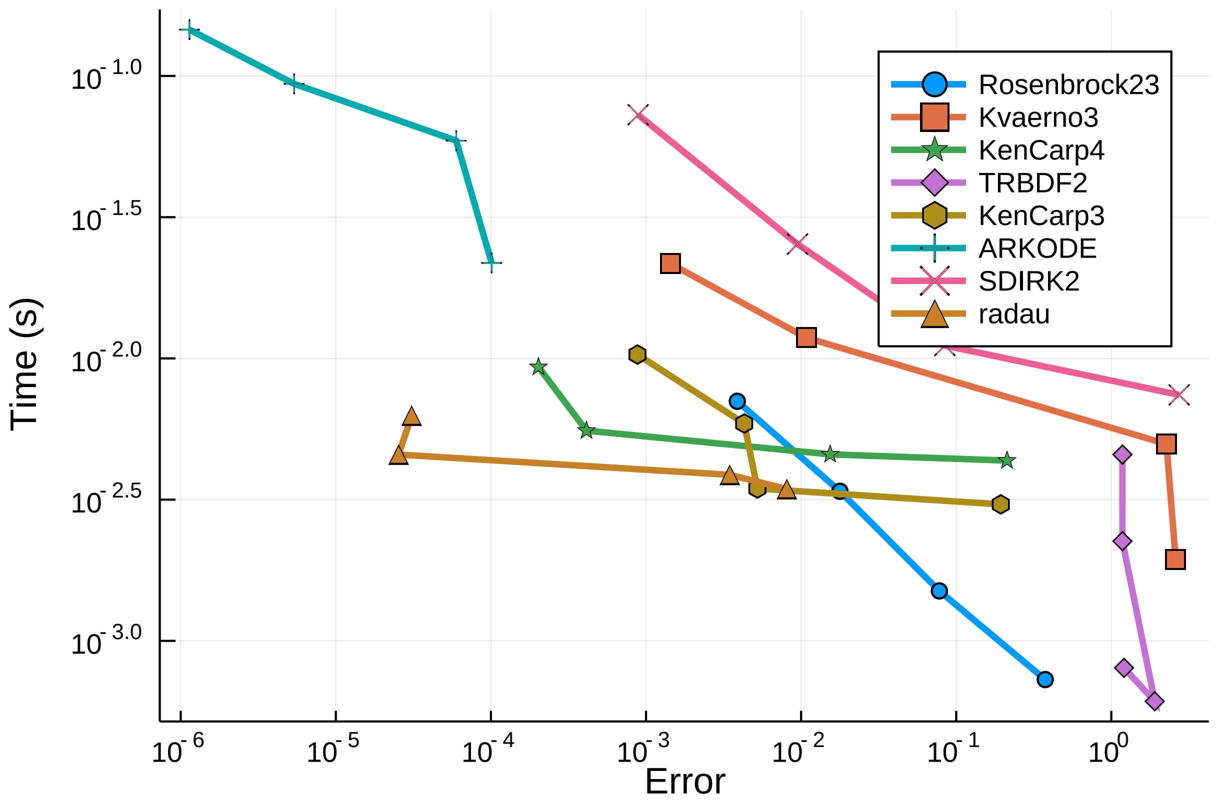
This tests the case where accuracy is not needed as much and quick robust solutions are necessary. Note that `ARKODE`'s convergence coefficient must be lowered to `1e-7` in order to converge.

Final timepoint error This measures the efficiency to get the value at the endpoint correct.

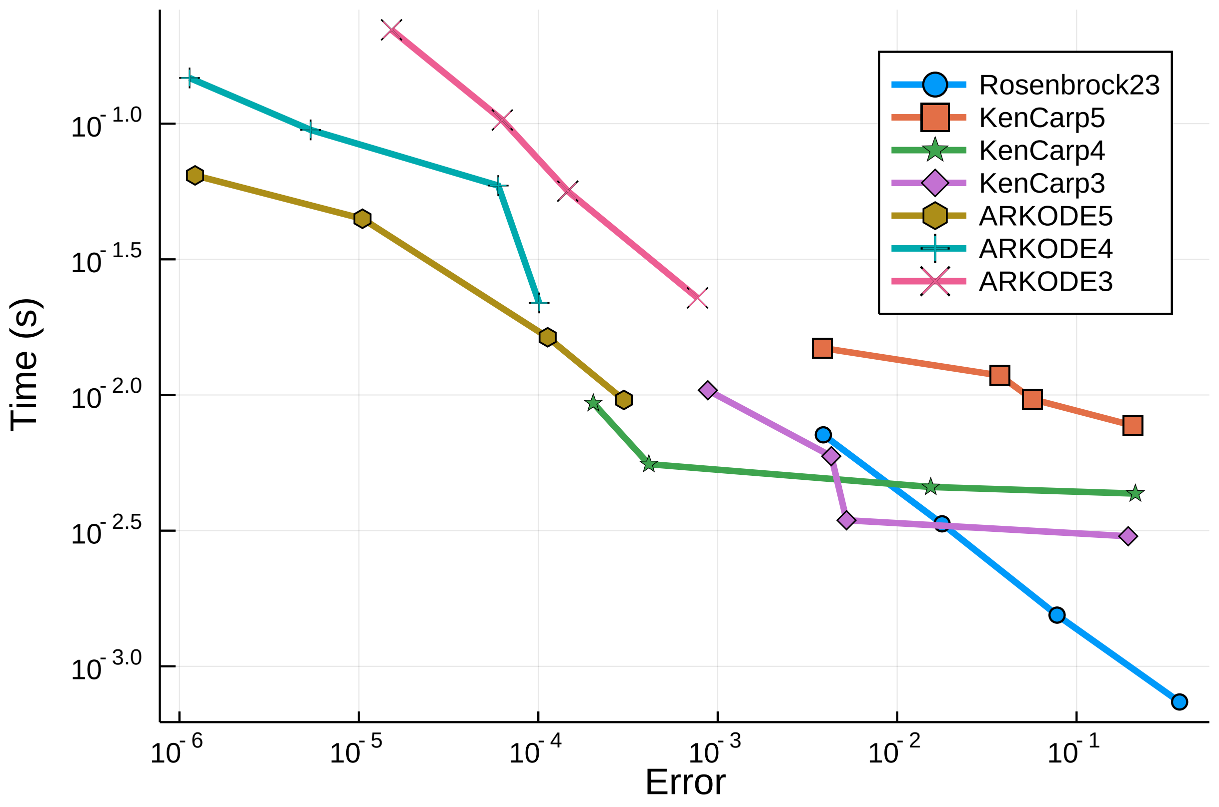
```
abstols = 1.0 ./ 10.0 .^ (4:7)
reltols = 1.0 ./ 10.0 .^ (1:4)

solve(prob, ddebdf())
solve(prob, rodas())
solve(prob, radau())
setups = [Dict(:alg=>Rosenbrock23()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>TRBDF2()),
          Dict(:alg=>ddebdf()),
          Dict(:alg=>rodas()),
          #Dict(:alg=>lsoda()),
          Dict(:alg=>radau())]
names = ["Rosenbrock23" "Rodas3" "CVODE_BDF" "TRBDF2" "ddebdf" "rodas" "radau"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e5),seconds=5)
plot(wp)
```



```
setups = [Dict(:alg=>Rosenbrock23()),
          Dict(:alg=>Kvaerno3()),
          Dict(:alg=>KenCarp4()),
          Dict(:alg=>TRBDF2()),
          Dict(:alg=>KenCarp3()),
          Dict(:alg=>ARKODE(nonlinear_convergence_coefficient = 1e-6)),
          Dict(:alg=>SDIRK2()),
          Dict(:alg=>radau())]
names = ["Rosenbrock23" "Kvaerno3" "KenCarp4" "TRBDF2" "KenCarp3" "ARKODE" "SDIRK2"
    "radau"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e5),seconds=5)
plot(wp)
```



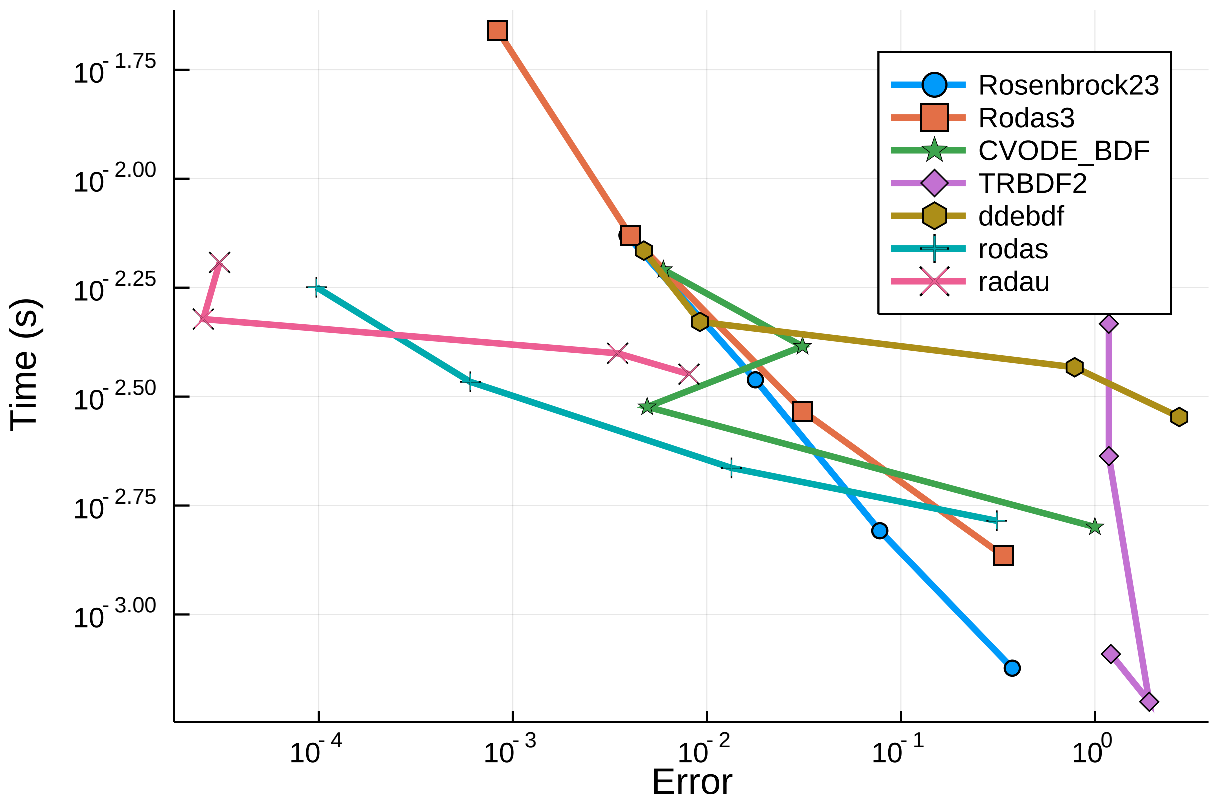
```
setups = [Dict(:alg=>Rosenbrock23()),
          Dict(:alg=>KenCarp5()),
          Dict(:alg=>KenCarp4()),
          Dict(:alg=>KenCarp3()),
          Dict(:alg=>ARKODE(order=5,nonlinear_convergence_coefficient = 1e-4)),
          Dict(:alg=>ARKODE(nonlinear_convergence_coefficient = 1e-6)),
          Dict(:alg=>ARKODE(nonlinear_convergence_coefficient = 1e-6,order=3))]
names = ["Rosenbrock23" "KenCarp5" "KenCarp4" "KenCarp3" "ARKODE5" "ARKODE4" "ARKODE3"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e5),seconds=5)
plot(wp)
```



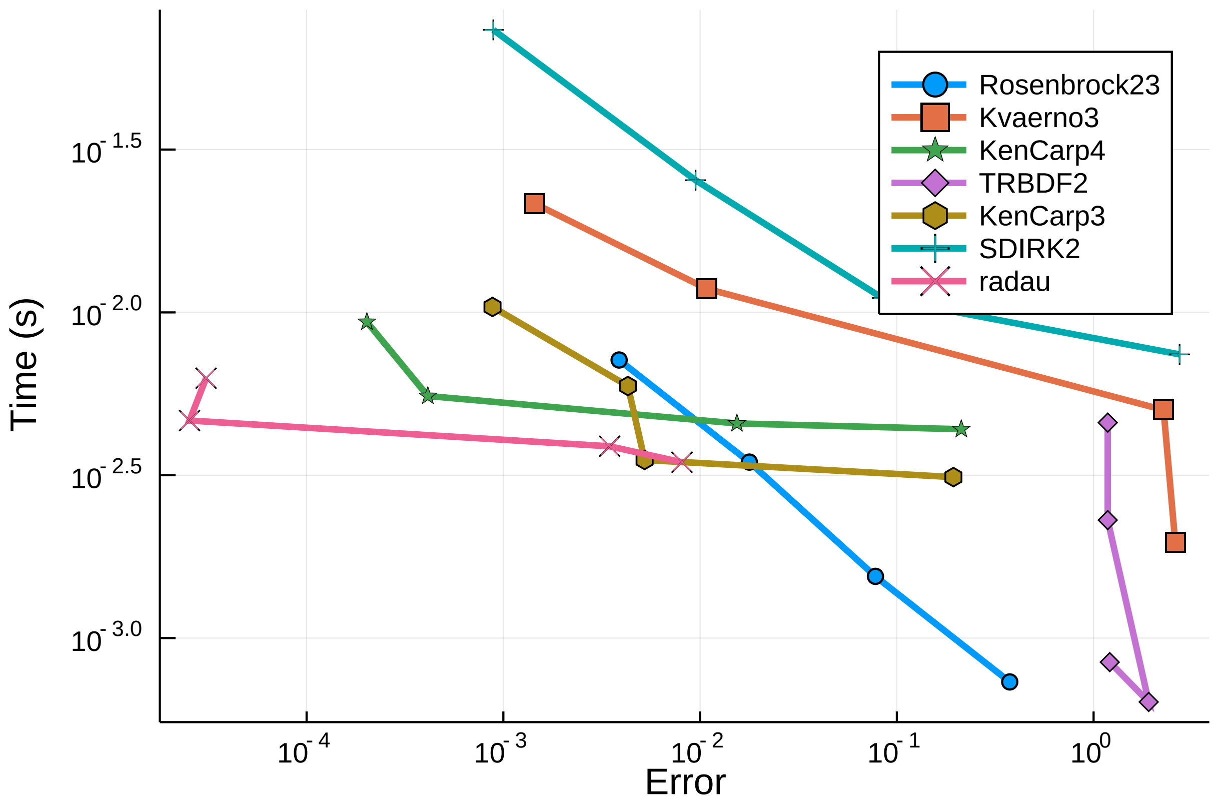
Notice that `KenCarp4` is the same overarching algorithm as `ARKODE` here (with major differences to stage predictors and adaptivity though). In this case, `KenCarp4` is more robust and more efficient than `ARKODE`. `CVODE_BDF` does quite well here, which is unusual for it on small equations. You can see that the low-order Rosenbrock methods `Rosenbrock23` and `Rodas3` dominate this test.

Timeseries error Now we measure the average error of the timeseries.

```
abstols = 1.0 ./ 10.0 .^ (4:7)
reltols = 1.0 ./ 10.0 .^ (1:4)

setups = [Dict(:alg=>Rosenbrock23()),
          Dict(:alg=>Rodas3()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>TRBDF2()),
          Dict(:alg=>ddebdf()),
          Dict(:alg=>rodas()),
          #Dict(:alg=>lsoda()),
          Dict(:alg=>radau())]
names = ["Rosenbrock23" "Rodas3" "CVODE_BDF" "TRBDF2" "ddebdf" "rodas" "radau"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,error_estimator=:l2,appxsol=test_sol,maxiters=Int(1e5),seconds=5)
plot(wp)
```



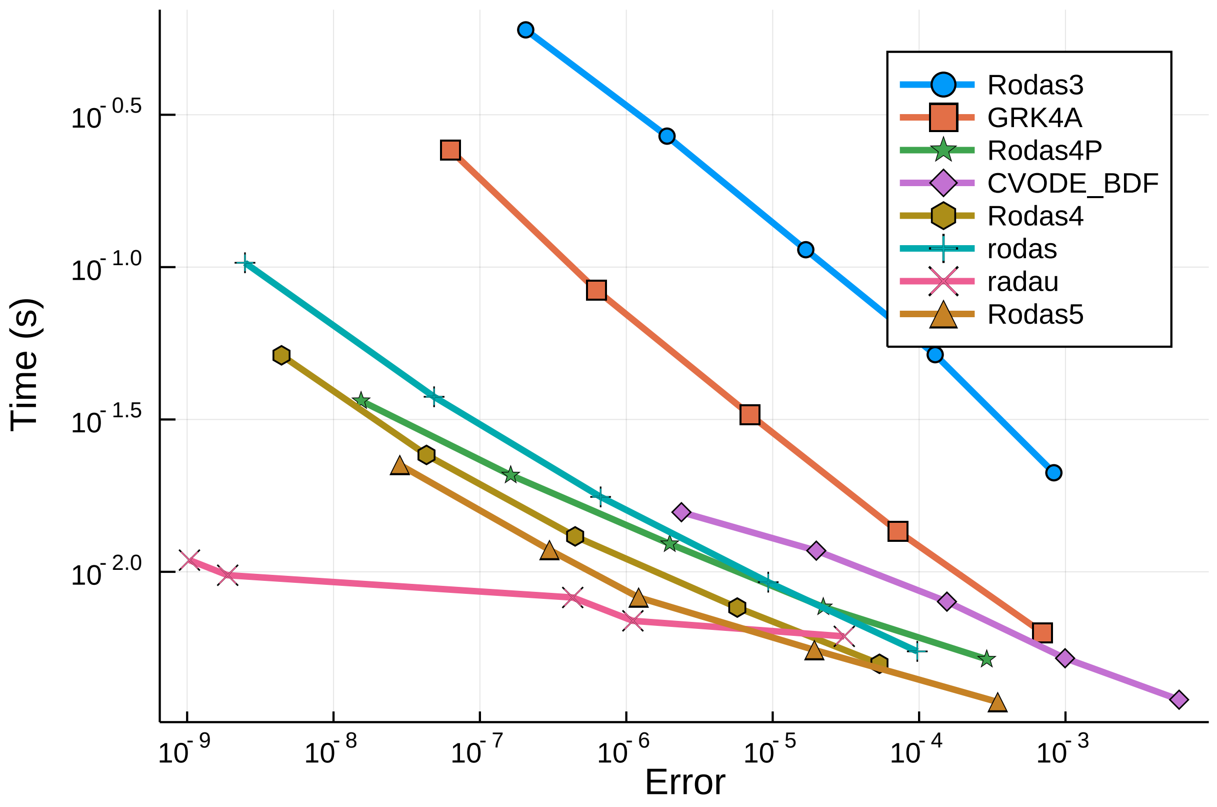
```
setups = [Dict(:alg=>Rosenbrock23(),:dense=>false),
          Dict(:alg=>Kvaerno3(),:dense=>false),
          Dict(:alg=>KenCarp4(),:dense=>false),
          Dict(:alg=>TRBDF2(),:dense=>false),
          Dict(:alg=>KenCarp3(),:dense=>false),
          Dict(:alg=>SDIRK2(),:dense=>false),
          Dict(:alg=>radau())]
names = ["Rosenbrock23" "Kvaerno3" "KenCarp4" "TRBDF2" "KenCarp3" "SDIRK2" "radau"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,appxsol=test_sol,maxiters=Int(1e5),error_estimator=:l2,seconds=5)
plot(wp)
```



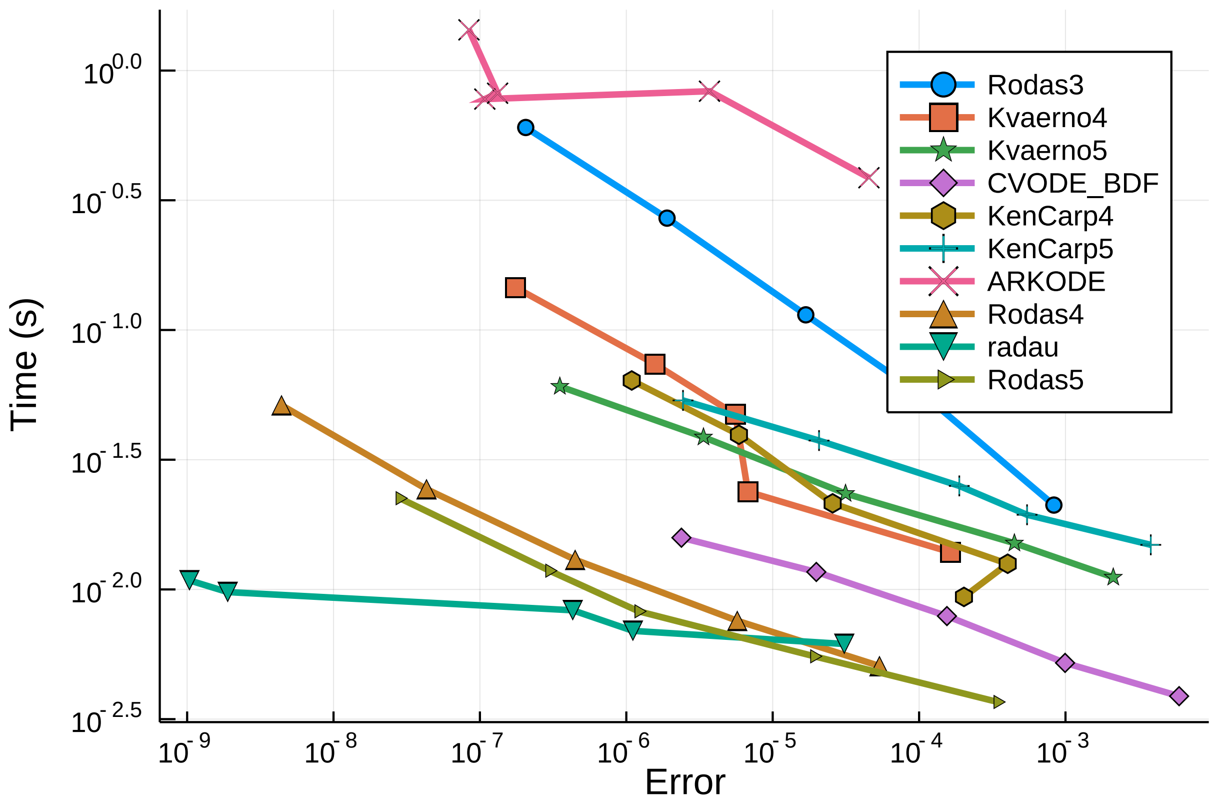
0.2.1 Higher accuracy tests

Now we transition to higher accracy tests. In this domain higher order methods are stable and much more efficient.

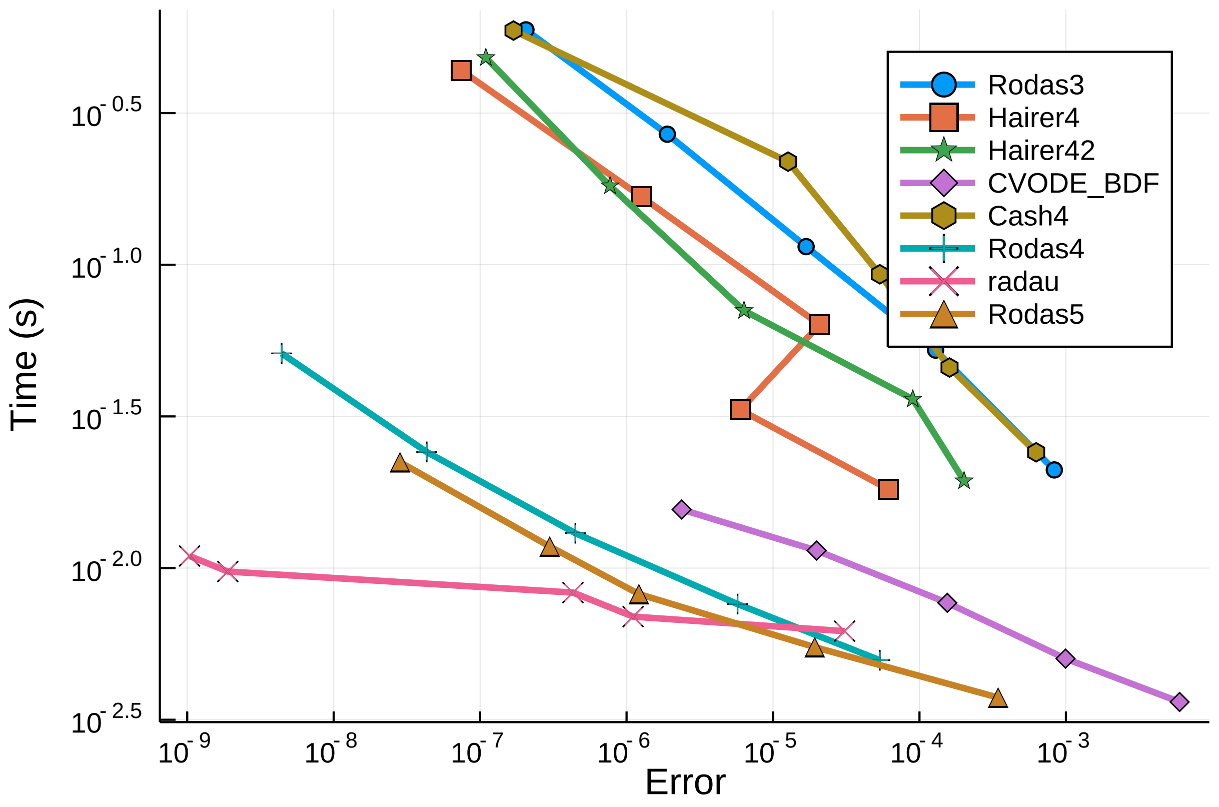
```
abstols = 1.0 ./ 10.0 .^ (7:11)
reltols = 1.0 ./ 10.0 .^ (4:8)
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>GRK4A()),
          Dict(:alg=>Rodas4P()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>rodas()),
          Dict(:alg=>radau()),
          #Dict(:alg=>lsoda()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "GRK4A" "Rodas4P" "CVODE_BDF" "Rodas4" "rodas" "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e6),seconds=5)
plot(wp)
```



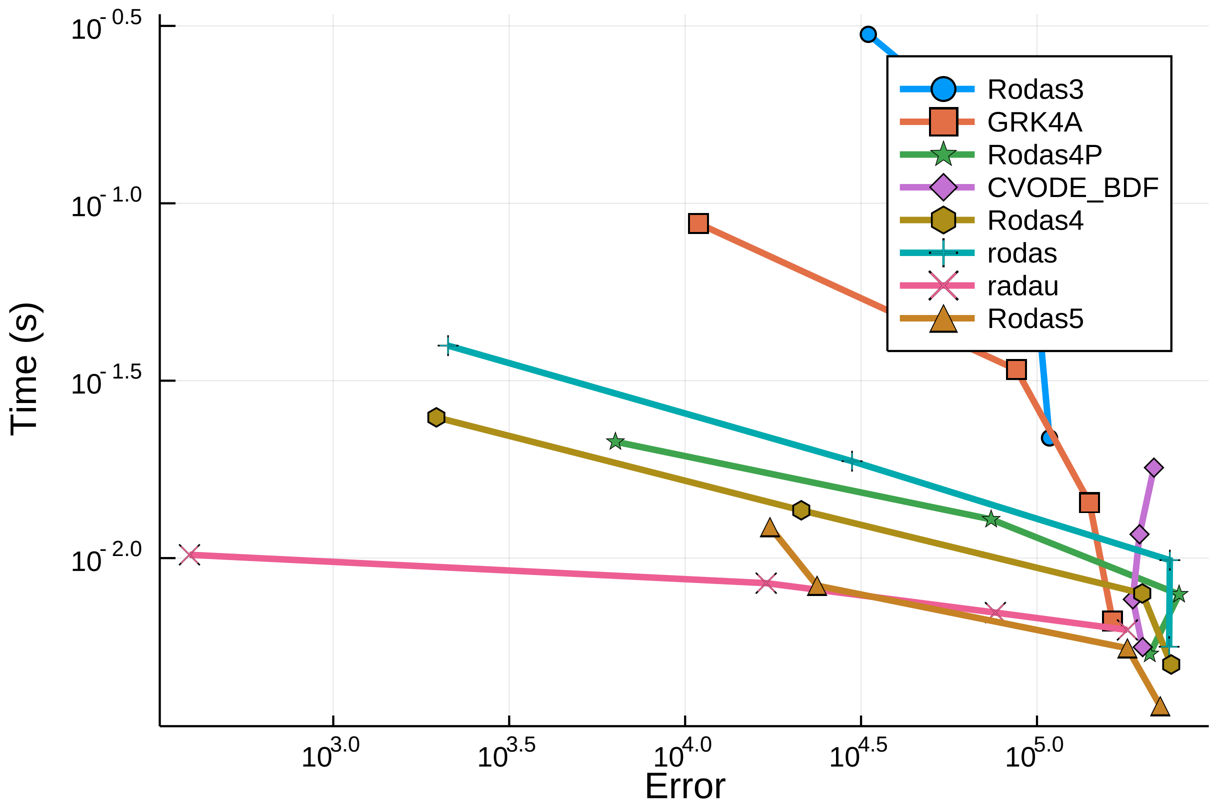
```
abstols = 1.0 ./ 10.0 .^ (7:11)
reltols = 1.0 ./ 10.0 .^ (4:8)
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>Kvaerno4()),
          Dict(:alg=>Kvaerno5()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>KenCarp4()),
          Dict(:alg=>KenCarp5()),
          Dict(:alg=>ARKODE()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>radau()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "Kvaerno4" "Kvaerno5" "CVODE_BDF" "KenCarp4" "KenCarp5" "ARKODE"
    "Rodas4" "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e6),seconds=5)
plot(wp)
```



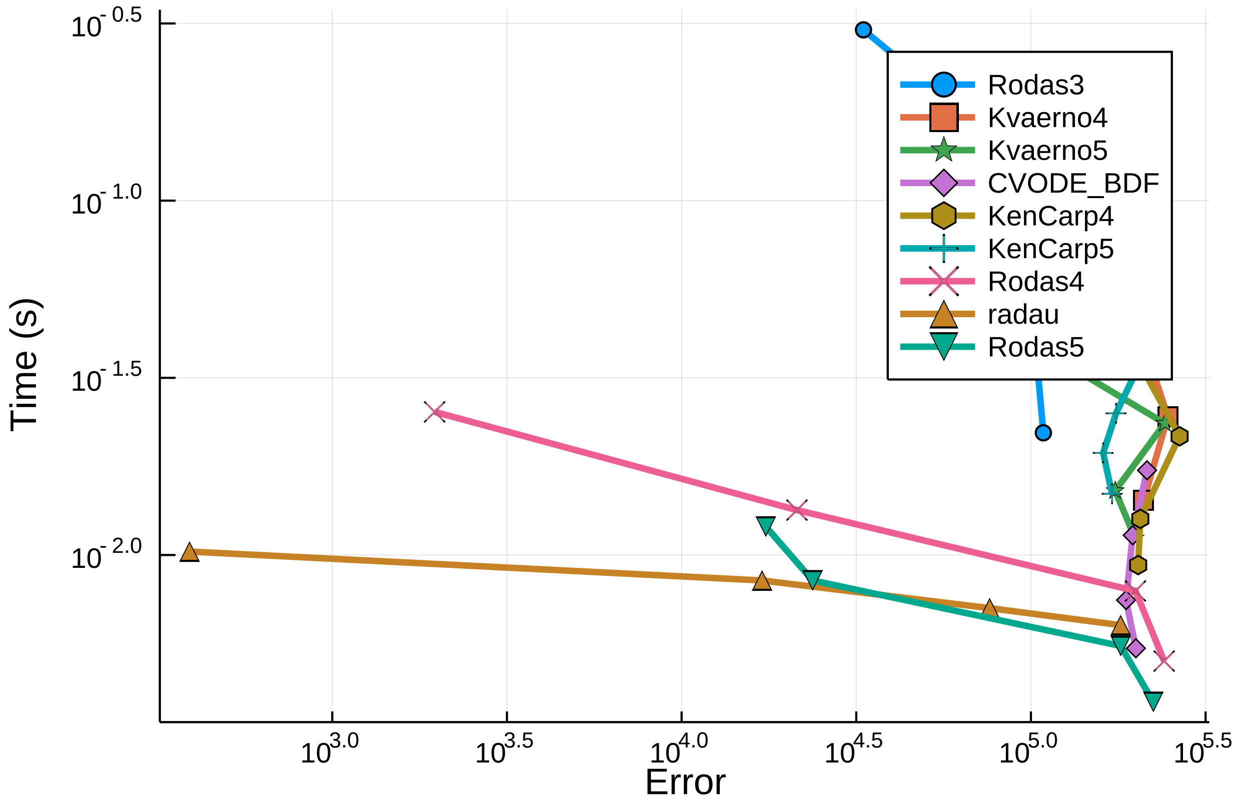
```
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>Hairer4()),
          Dict(:alg=>Hairer42()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>Cash4()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>radau()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "Hairer4" "Hairer42" "CVODE_BDF" "Cash4" "Rodas4" "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,save_everystep=false,appxsol=test_sol,maxiters=Int(1e6),seconds=5)
plot(wp)
```



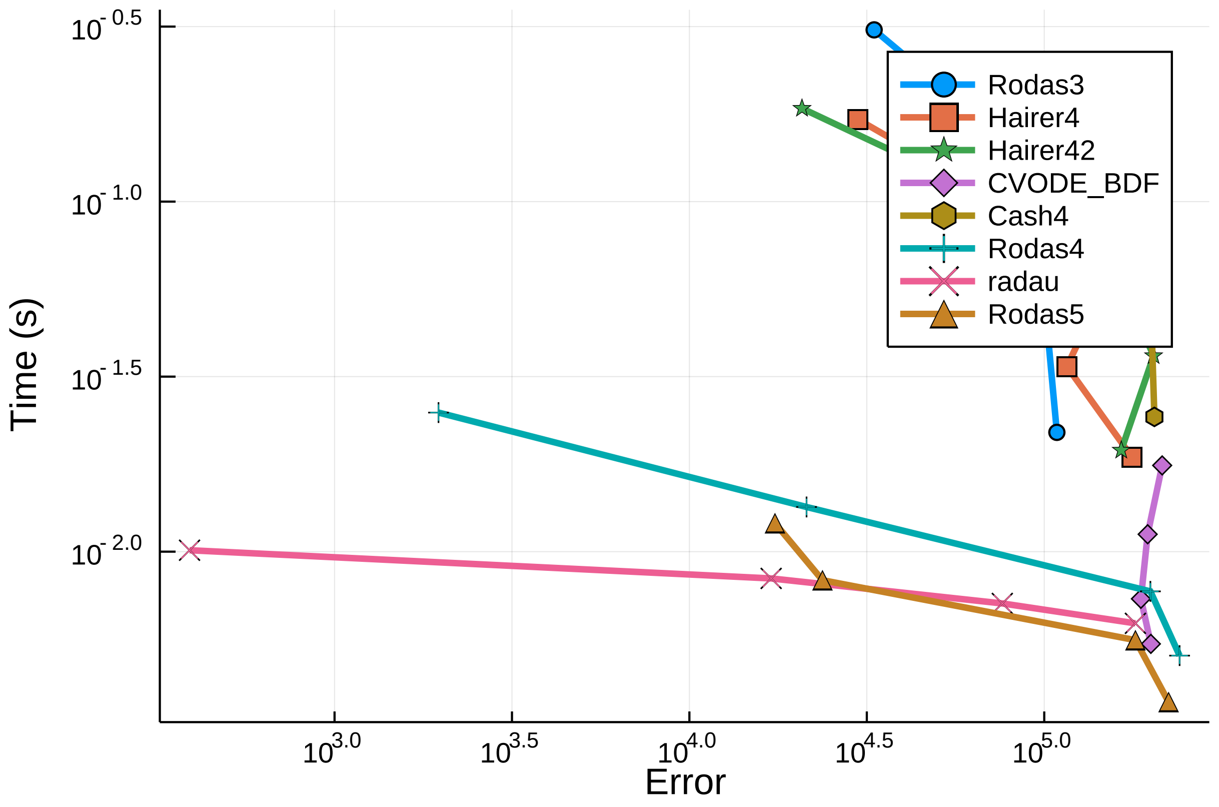
```
abstols = 1.0 ./ 10.0 .^ (7:10)
reltols = 1.0 ./ 10.0 .^ (4:7)
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>GRK4A()),
          Dict(:alg=>Rodas4P()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>rodas()),
          Dict(:alg=>radau()),
          #Dict(:alg=>lsoda()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "GRK4A" "Rodas4P" "CVODE_BDF" "Rodas4" "rodas" "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,appxsol=test_sol,maxiters=Int(1e6),error_estimate=:l2,seconds=5)
plot(wp)
```



```
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>Kvaerno4()),
          Dict(:alg=>Kvaerno5()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>KenCarp4()),
          Dict(:alg=>KenCarp5()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>radau()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "Kvaerno4" "Kvaerno5" "CVODE_BDF" "KenCarp4" "KenCarp5" "Rodas4"
    "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,appxsol=test_sol,maxiters=Int(1e6),error_estimate=:l2,seconds=5)
plot(wp)
```



```
setups = [Dict(:alg=>Rodas3()),
          Dict(:alg=>Hairer4()),
          Dict(:alg=>Hairer42()),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>Cash4()),
          Dict(:alg=>Rodas4()),
          Dict(:alg=>radau()),
          Dict(:alg=>Rodas5())]
names = ["Rodas3" "Hairer4" "Hairer42" "CVODE_BDF" "Cash4" "Rodas4" "radau" "Rodas5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;

    names=names,appxsol=test_sol,maxiters=Int(1e6),error_estimate=:l2,seconds=5)
plot(wp)
```



The timeseries test is a little odd here because of the high peaks in the VanDerPol oscillator. At a certain accuracy, the steps try to resolve those peaks and so the error becomes higher.

While the higher order order Julia-based Rodas methods (`Rodas4` and `Rodas4P`) Rosenbrock methods are not viable at higher tolerances, they dominate for a large portion of this benchmark. When the tolerance gets low enough, `radau` adaptive high order (up to order 13) takes the lead.

0.2.2 Conclusion

`Rosenbrock23` and `Rodas3` do well when tolerances are higher. In most standard tolerances, `Rodas4` and `Rodas4P` do extremely well. Only when the tolerances get very low does `radau` do well. The Julia Rosenbrock methods vastly outperform their Fortran counterparts. `CVODE_BDF` is a top performer in the final timepoint errors with low accuracy, but take that with a grain of salt because the problem is periodic which means it's getting the spikes wrong but the low parts correct. `ARKODE` does poorly in these tests. `lsoda` does quite well in both low and high accuracy domains, but is never the top.

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

0.3 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("StiffODE","VanDerPol.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 5.4.1
[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 0.15.1
[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
[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 3.3.1
[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 0.25.2
[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 3.6.1
[92b13dbe-c966-51a2-8445-caca9f8a7d42] TaylorIntegration 0.5.0
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.0
[e88e6eb3-aa80-5325-afca-941959d7151f] Zygote 0.3.2
```