Linear 100x100 Work-Precision Diagrams

Chris Rackauckas

November 17, 2018

For these tests we will solve an 100x100 system of linear differential equations. This will demonstrate the efficiency of the methods for handling large systems. We will be mostly looking at the efficiency of the work-horse Dormand-Prince Order 4/5 Pairs: one from DifferentialEquations.jl (`DP5`, one from ODE.jl `rk45`, and one from ODEInterface: Hairer's famous `dopri5`).

Also included is `Tsit5`. While all other ODE programs have gone with the traditional choice of using the Dormand-Prince 4/5 pair as the default, DifferentialEquations.jl uses `Tsit5` as one of the default algorithms. It's a very new (2011) and not widely known, but the theory and the implimentation shows it's more efficient than DP5. Thus we include it just to show off how re-designing a library from the ground up in a language for rapid code and rapid development has its advantages.

0.1 Setup

```
using DifferentialEquations, DiffEqDevTools, Plots, ODEInterfaceDiffEq, ODE
gr()
# 2D Linear ODE
function f(du,u,p,t)
  for i in 1:length(u)
    du[i] = 1.01*u[i]
  end
end
function f_analytic(u_0,p,t)
  u_0*exp(1.01*t)
end
tspan = (0.0,10.0)
prob = ODEProblem(ODEFunction(f,analytic=f_analytic),rand(100,100),tspan)

abstols = 1.0./10.0.^(3:13)
reltols = 1.0./10.0.^(0:10);
```

0.1.1 Speed Baseline

First a baseline. These are all testing the same Dormand-Prince order 5/4 algorithm of each package. While all the same Runge-Kutta tableau, they exhibit different behavior due to different choices of adaptive timestepping algorithms and tuning. First we will test with all extra saving features are turned off to put DifferentialEquations.jl in "speed mode".

```
setups = [Dict(:alg=>DP5())
          Dict(:alg=>ode45())
```

```
            Dict(:alg=>dopri5())
            Dict(:alg=>ARKODE(Sundials.Explicit(),etable=Sundials.DORMAND_PRINCE_7_4_5))
            Dict(:alg=>Tsit5())]
names = ["OrdinaryDiffEq";"ODE";"ODEInterface";"Sundials ARKODE";"OrdinaryDiffEq Tsit5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;names=names,save_everystep=false)
plot(wp)
```
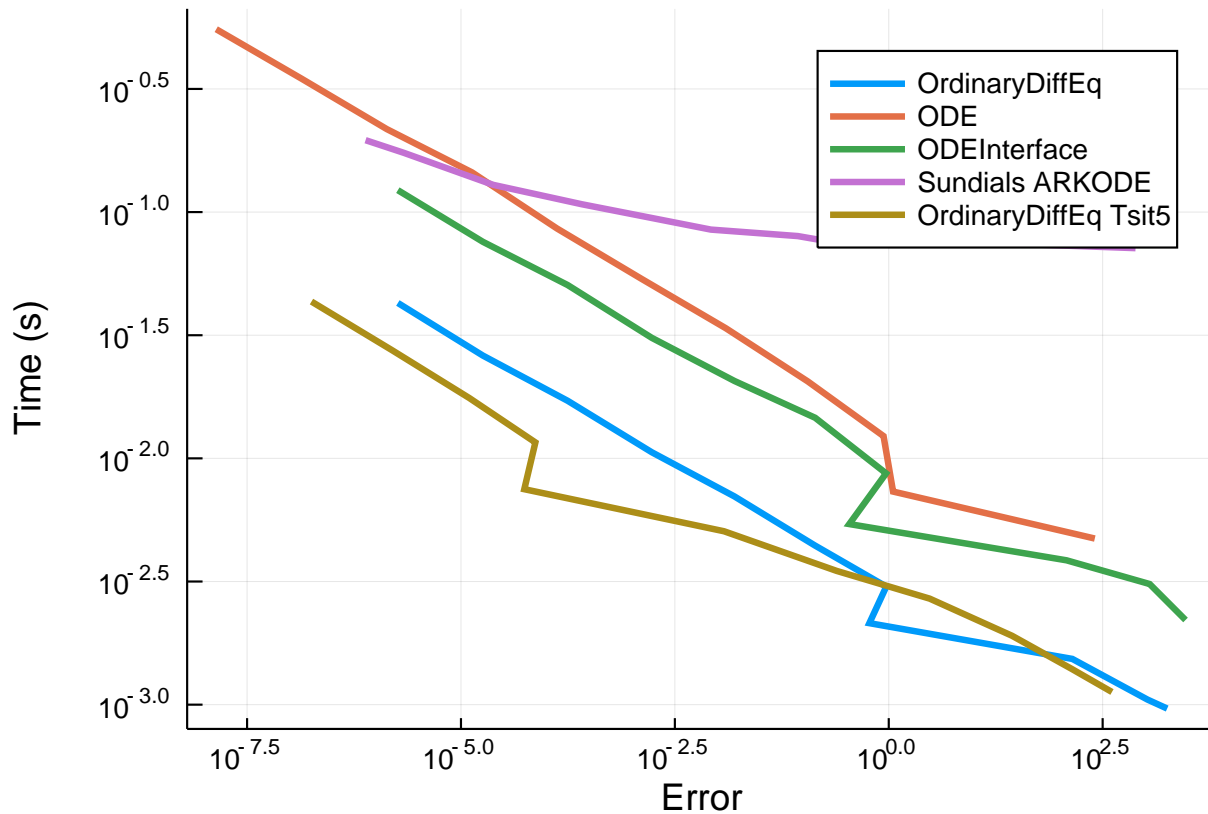

OrdinaryDiffEq.jl is clearly far in the lead, being more than an order of magnitude faster for the same amount of error.

0.1.2   Full Saving

```
setups = [Dict(:alg=>DP5(),:dense=>false)
          Dict(:alg=>ode45(),:dense=>false)
          Dict(:alg=>dopri5()) # dense=false by default:  no nonlinear interpolation

     Dict(:alg=>ARKODE(Sundials.Explicit(),etable=Sundials.DORMAND_PRINCE_7_4_5),:dense=>false)
          Dict(:alg=>Tsit5(),:dense=>false)]
names = ["OrdinaryDiffEq";"ODE";"ODEInterface";"Sundials ARKODE";"OrdinaryDiffEq Tsit5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;names=names)
plot(wp)
```
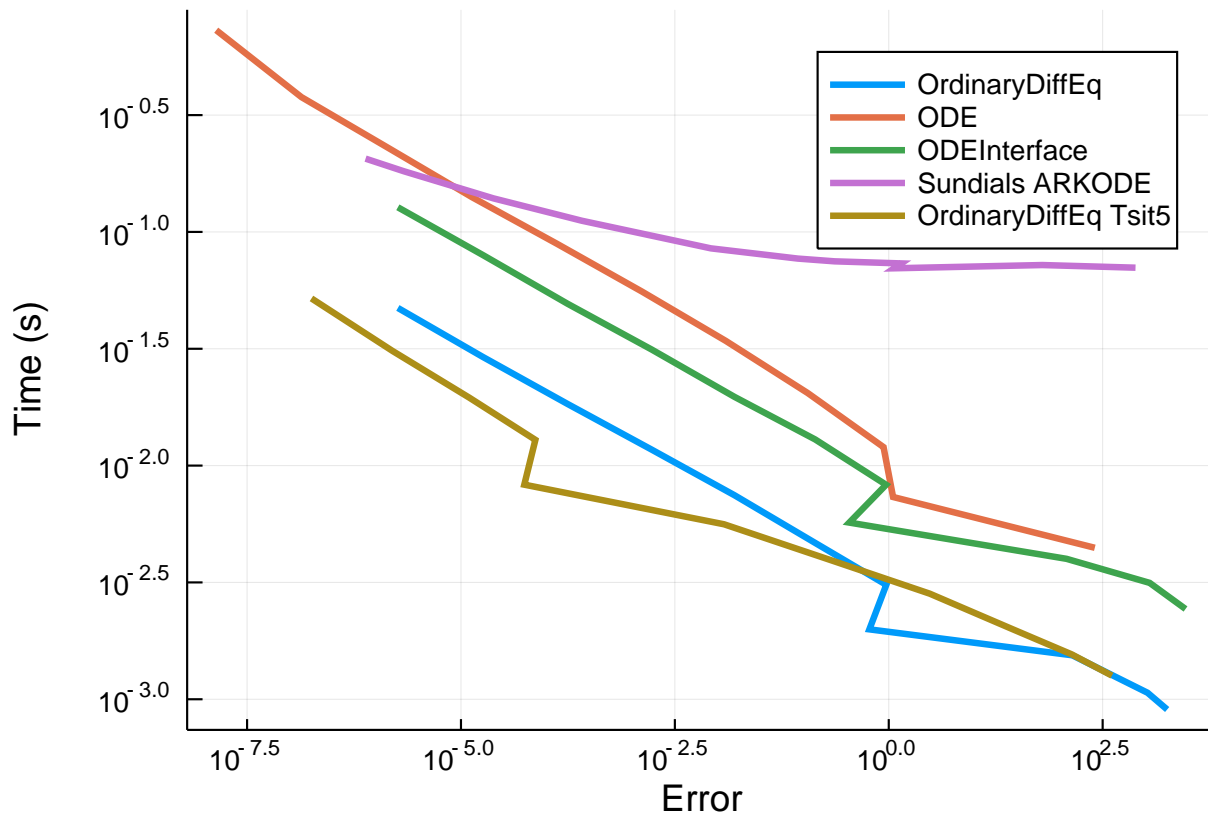
While not as dramatic as before, DifferentialEquations.jl is still far in the lead. Since the times are log scaled, this comes out to be almost a 5x lead over ODEInterface, and about a 10x lead over ODE.jl at default tolerances.

0.1.3 Continuous Output

Now we include continuous output. This has a large overhead because at every timepoint the matrix of rates `k` has to be deep copied.

```julia
setups = [Dict(:alg=>DP5())
          Dict(:alg=>ode45())
          Dict(:alg=>dopri5())

    Dict(:alg=>ARKODE(Sundials.Explicit(),etable=Sundials.DORMAND_PRINCE_7_4_5),:dense=>false)
          Dict(:alg=>Tsit5())]
names = ["OrdinaryDiffEq";"ODE";"ODEInterface";"Sundials ARKODE";"OrdinaryDiffEq Tsit5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;names=names)
plot(wp)
```
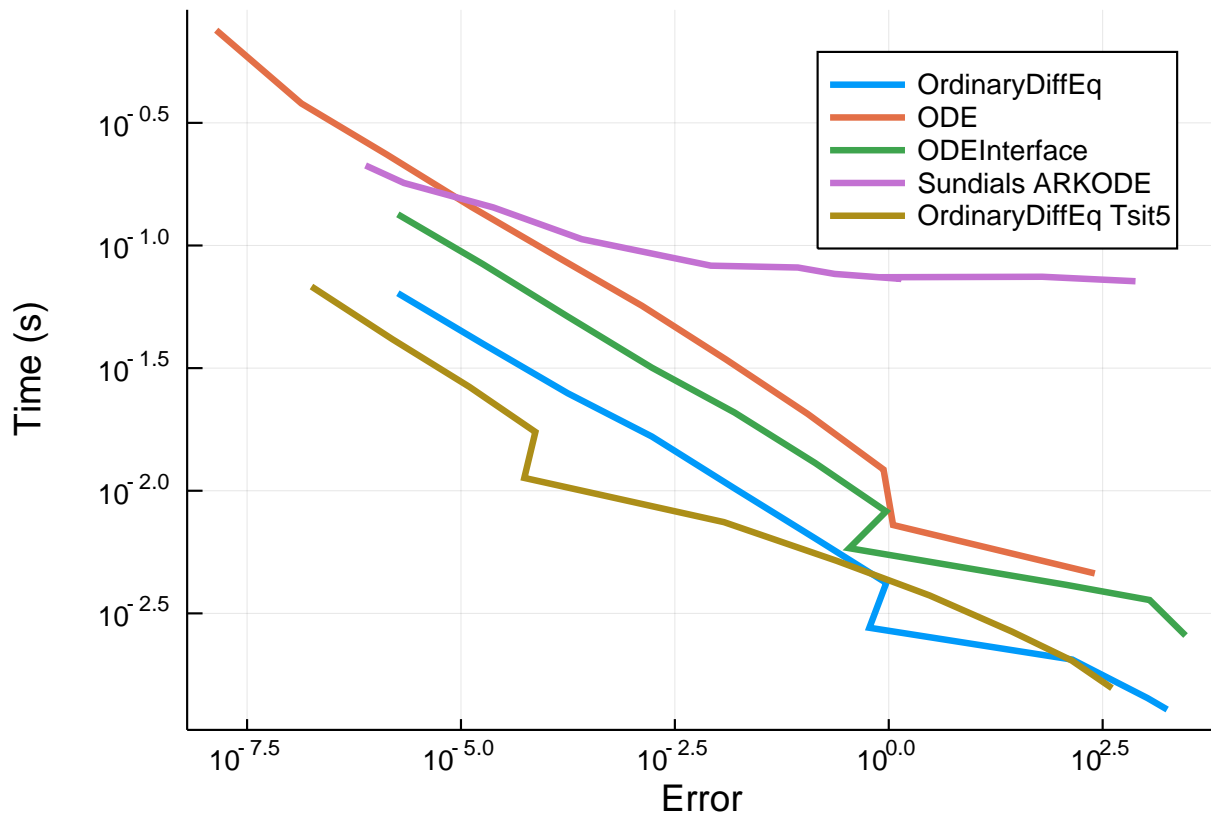
As you can see, even with this large overhead, DifferentialEquations.jl essentially ties with ODEInterface. This shows that the fully featured `DP5` solver holds its own with even the classic "great" methods.
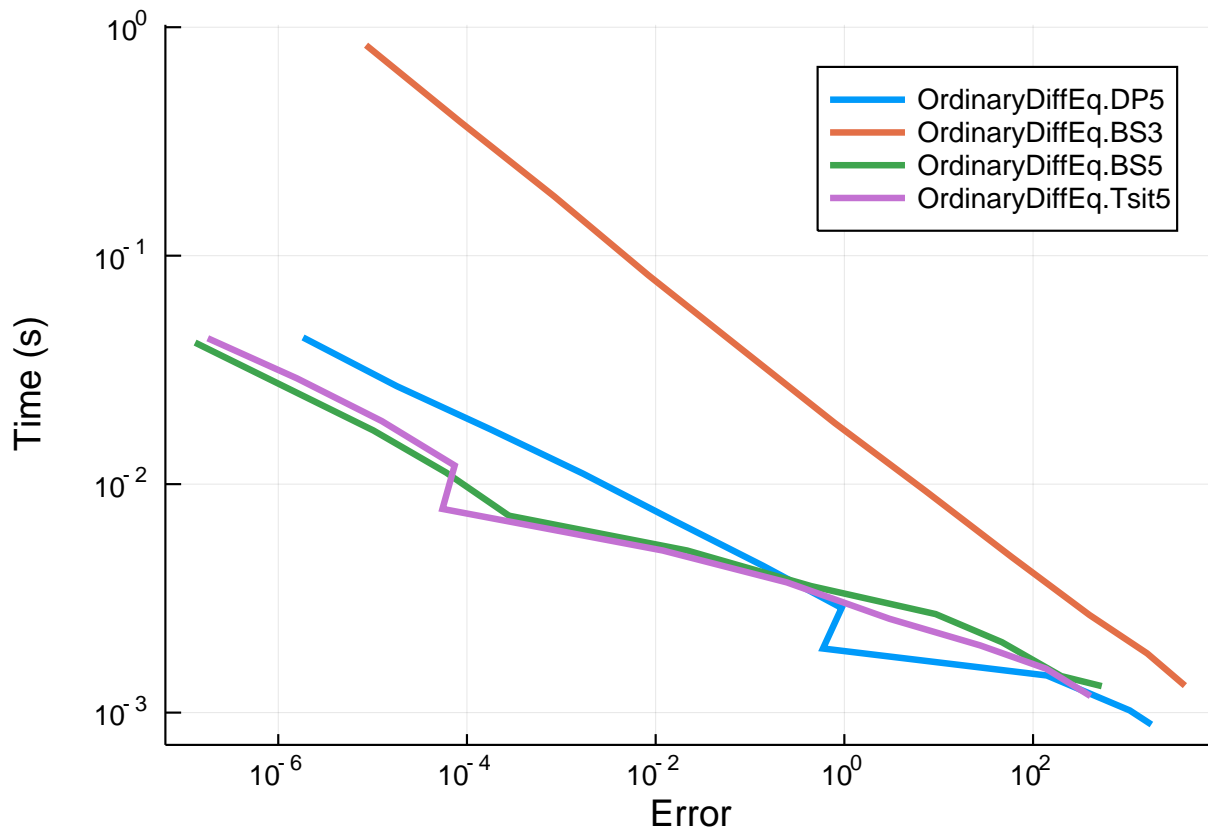
0.1.4 Other Runge-Kutta Algorithms

Now let's test it against a smattering of other Runge-Kutta algorithms. First we will test it with all overheads off. Let's do the Order 5 (and the 2/3 pair) algorithms:

```
setups = [Dict(:alg=>DP5())
          Dict(:alg=>BS3())
          Dict(:alg=>BS5())
          Dict(:alg=>Tsit5())]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;save_everystep=false)
plot(wp)
```

As you can see, the `Tsit5` algorithm is the most efficient, beating `DP5` which is more efficient than the Bogacki-Shampine algorithms. However, you can see that when the tolerance is high, `BS3` could be of use since its slope is so steep.
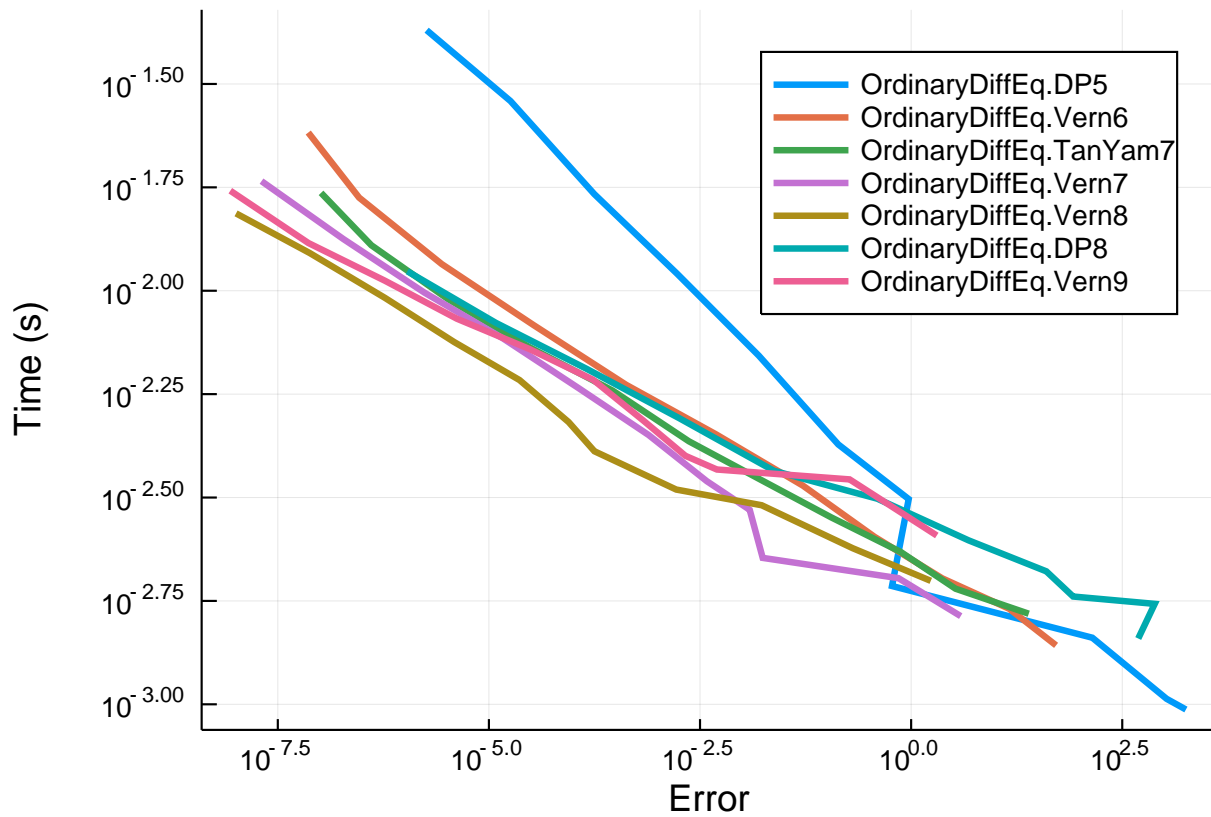
0.2   Higher Order

Now let's see how OrdinaryDiffEq.jl fairs with some higher order algorithms:

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

Vern7 looks to be the winner here, with DP5 doing well at higher tolerances but trailing of when it gets lower as one would expect with lower order algorithms. Some of the higher order methods, such as `Vern9`, would do better at lower tolerances than what's tested (outside of floating point range).

0.3 Higher Order With Many Packages

Now we test OrdinaryDiffEq against the high order methods of the other packages:

```
using LSODA
setups = [Dict(:alg=>DP5())
          Dict(:alg=>Vern7())
          Dict(:alg=>dop853())
          Dict(:alg=>ode78())
          Dict(:alg=>odex())
          Dict(:alg=>lsoda())
          Dict(:alg=>ddeabm())
          Dict(:alg=>ARKODE(Sundials.Explicit(),order=8))
          Dict(:alg=>CVODE_Adams())]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;save_everystep=false)
plot(wp)
```
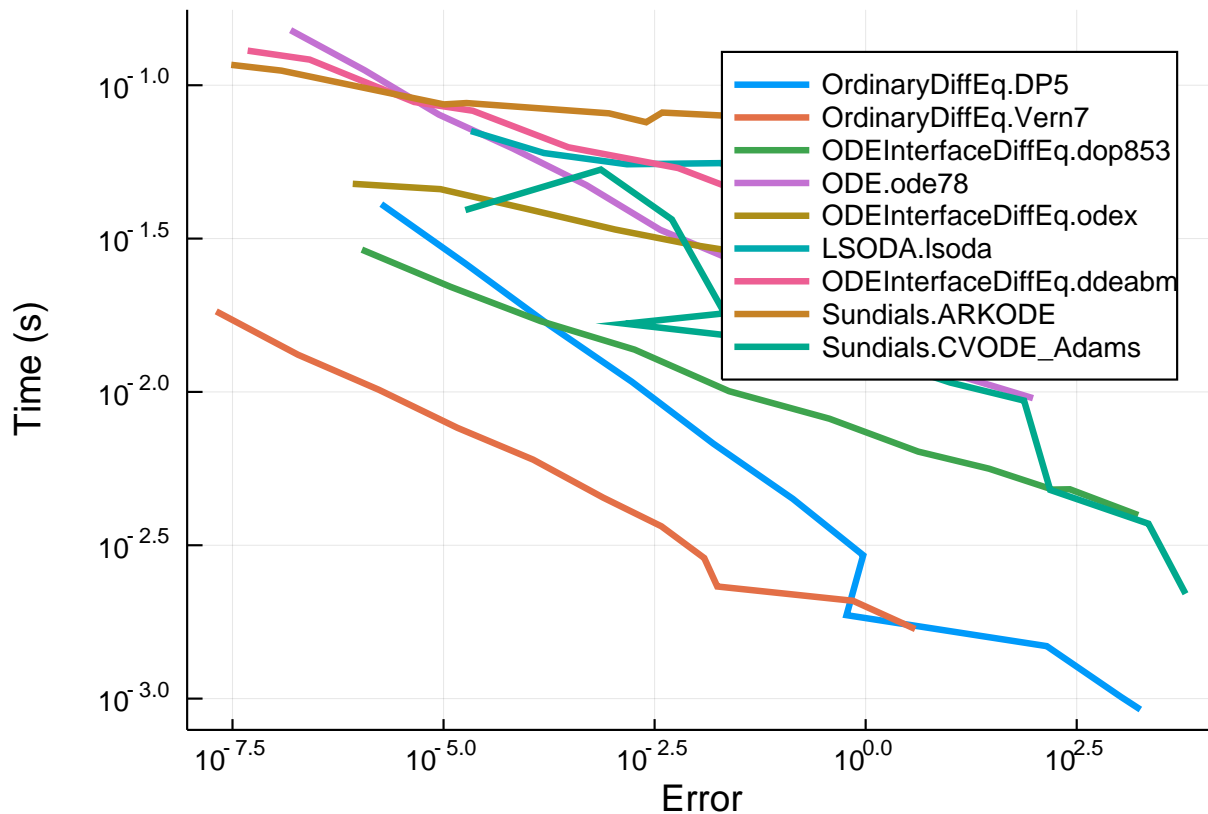
Here you can once again see the DifferentialEquations algorithms far in the lead. It's well known that for cheap function costs Adams methods are inefficient. ODE.jl one again has a bad showing.
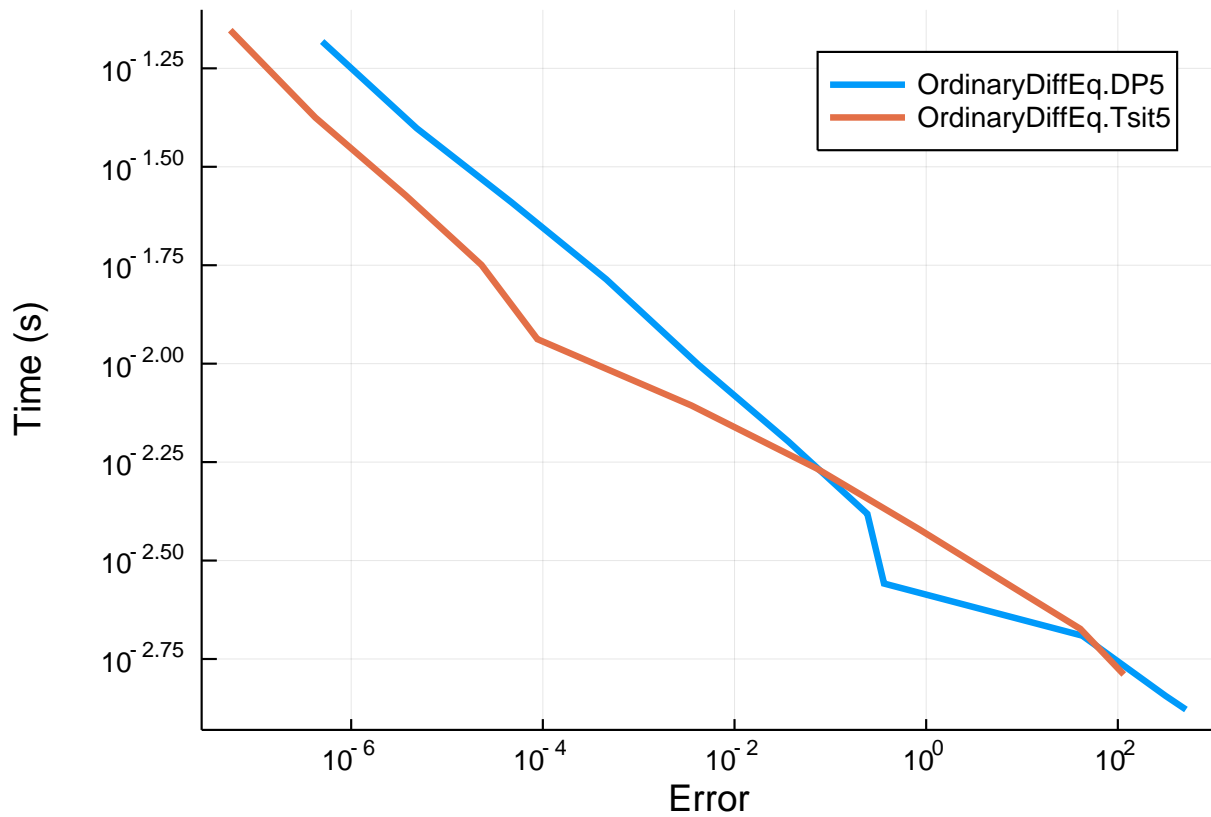
0.4 Interpolation Error

Now we will look at the error using an interpolation measurement instead of at the timestepping points. Since the DifferentialEquations.jl algorithms have higher order interpolants than the ODE.jl algorithms, one would expect this would magnify the difference. First the order 4/5 comparison:

```
setups = [Dict(:alg=>DP5())
          #Dict(:alg=>ode45())
          Dict(:alg=>Tsit5())]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;error_estimate=:L2,dense_errors=true)
plot(wp)
```

Note that all of ODE.jl uses a 3rd order Hermite interpolation, while the DifferentialEquations algorithms interpolations which are specialized to the algorithm. For example, DP5 and `Tsit5` both use "free" order 4 interpolations, which are both as fast as the Hermite interpolation while achieving far less error. At higher order:

```julia
setups = [Dict(:alg=>DP5())
          Dict(:alg=>Vern7())
          #Dict(:alg=>ode78())
          ]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;error_estimate=:L2,dense_errors=true)
plot(wp)
```
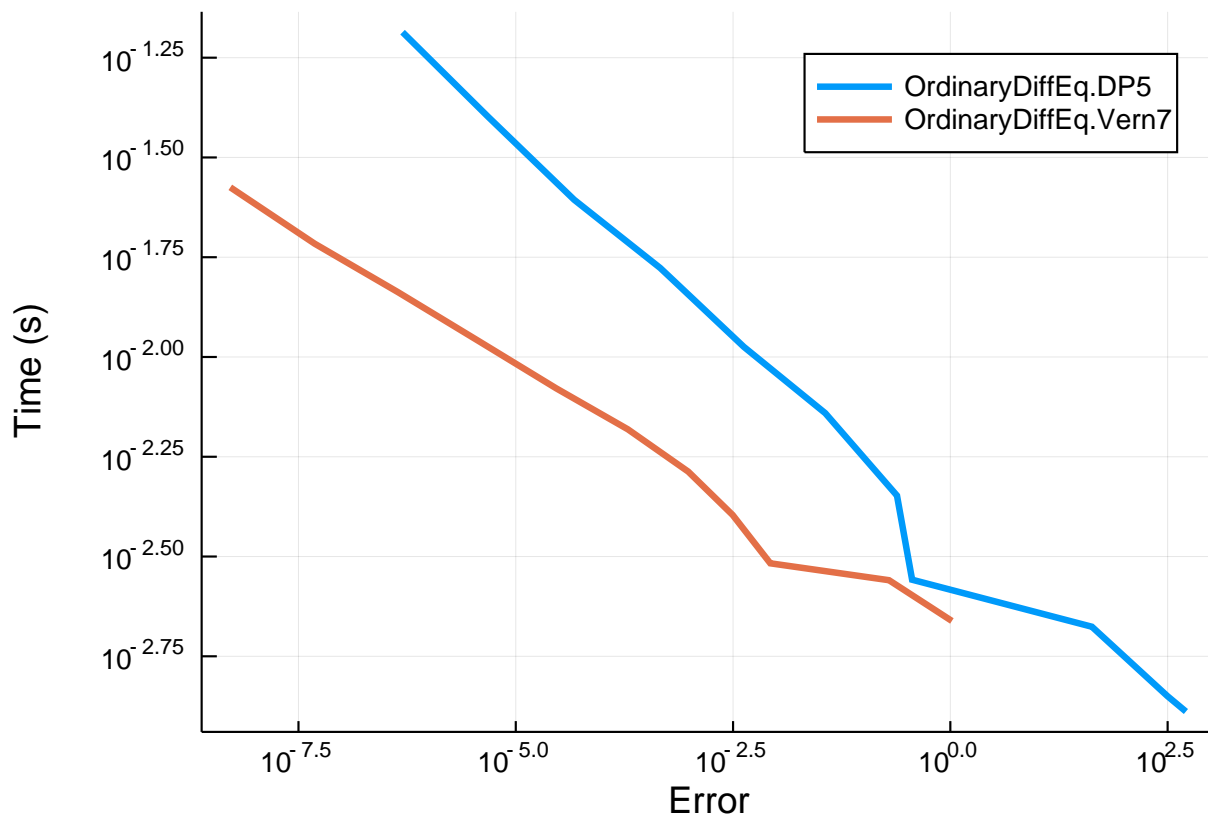
0.5 Comparison with Fixed Timestep RK4

Let's run the first benchmark but add some fixed timestep RK4 methods to see the difference:

```
using GeometricIntegratorsDiffEq
```

```
Error: ArgumentError: Package GeometricIntegratorsDiffEq not found in curre
nt path:
- Run `import Pkg; Pkg.add("GeometricIntegratorsDiffEq")` to install the Ge
ometricIntegratorsDiffEq package.
```

```
abstols = 1.0./10.0.^(3:13)
reltols = 1.0./10.0.^(0:10);
dts = [1,1/2,1/4,1/10,1/20,1/40,1/60,1/80,1/100,1/140,1/240]
setups = [Dict(:alg=>DP5())
          Dict(:alg=>ode45())
          Dict(:alg=>dopri5())
          Dict(:alg=>GIERK4(),:dts=>dts)
          Dict(:alg=>RK4(),:dts=>dts)
          Dict(:alg=>Tsit5())]
```

```
Error: UndefVarError: GIERK4 not defined
```

```
names = ["DifferentialEquations";"ODE";"ODEInterface";"GeometricIntegrators
    RK4";"DifferentialEquations RK4";"DifferentialEquations Tsit5"]
wp = WorkPrecisionSet(prob,abstols,reltols,setups;names=names,
                      save_everystep=false,verbose=false)
plot(wp)
```

```
Error: DimensionMismatch("new dimensions (1, 2) must be consistent with arr
ay size 6")
```

0.6 Conclusion

DifferentialEquations's default choice of `Tsit5` does well for quick and easy solving at normal tolerances. However, at low tolerances the higher order algorithms are faster. In every case, the DifferentialEquations algorithms are far in the lead, many times an order of magnitude faster than the competitors. `Vern7` with its included 7th order interpolation looks to be a good workhorse for scientific computing in floating point range. These along with many other benchmarks are why these algorithms were chosen as part of the defaults.