BCR Work-Precision Diagrams

Samuel Isaacson and Chris Rackauckas

May 13, 2021

The following benchmark is of 1122 ODEs with 24388 terms that describe a stiff chemical reaction network modeling the BCR signaling network from Barua et al.. We use `ReactionNetworkImporters` to load the BioNetGen model files as a Catalyst model, and then use ModelingToolkit to convert the Catalyst network model to ODEs.

```
using DiffEqBase, OrdinaryDiffEq, Catalyst, ReactionNetworkImporters,
      Sundials, Plots, DiffEqDevTools, ODEInterface, ODEInterfaceDiffEq,
      LSODA, TimerOutputs, LinearAlgebra, ModelingToolkit

gr()
datadir  = joinpath(dirname(pathof(ReactionNetworkImporters)),"../data/bcr")
const to = TimerOutput()
tf       = 100000.0

# generate ModelingToolkit ODEs
@timeit to "Parse Network" prnbng = loadrxnetwork(BNGNetwork(), joinpath(datadir,
"bcr.net"))
rn    = prnbng.rn
@timeit to "Create ODESys" osys = convert(ODESystem, rn)

u_0    = prnbng.u_0
p     = prnbng.p
tspan = (0.,tf)
@timeit to "ODEProb No Jac" oprob = ODEProblem(osys, u_0, tspan, p)
@timeit to "ODEProb DenseJac" densejacprob = ODEProblem(osys, u_0, tspan, p, jac=true)
```

```
Parsing parameters...done
Adding parameters...done
Parsing species...done
Adding species...done
Creating ModelingToolkit versions of species and parameters...done
Parsing and adding reactions...done
Parsing groups...done
ODEProblem with uType Vector{Float64} and tType Float64. In-place: true
timespan: (0.0, 100000.0)
u0: 1122-element Vector{Float64}:
 299717.8348854
  47149.15480798
  46979.01102231
 290771.2428252
 299980.7396749
 300000.0
    141.3151575495
      0.1256496403614
      0.4048783555301
```

```
 140.8052338618
   ⋮
   1.005585387399e-24
   6.724953378237e-17
   3.395560698281e-16
   1.787990228838e-5
   8.761844379939e-13
   0.0002517949074779
   0.0005539124513976
   2.281251822741e-14
   1.78232055967e-8
```

```
@timeit to "ODEProb SparseJac" sparsejacprob = ODEProblem(osys, u_0, tspan, p, jac=true,
sparse=true)
show(to)
```

```
                                         Time                    Allocations


      Tot / % measured:              313s / 100%             48.6GiB / 100%


 Section                  ncalls     time    %tot     avg     alloc    %tot      a
vg


 ODEProb DenseJac              1     284s   90.7%    284s   41.4GiB   85.2%  41.4G
iB
 ODEProb SparseJac             1    18.8s   6.00%   18.8s   4.20GiB   8.66%  4.20G
iB
 ODEProb No Jac                1    8.71s   2.78%   8.71s   2.33GiB   4.81%  2.33G
iB
 Parse Network                 1    971ms   0.31%   971ms    150MiB   0.30%   150M
iB
 Create ODESys                 1    709ms   0.23%   709ms    506MiB   1.02%   506M
iB
```

```
@show numspecies(rn) # Number of ODEs
@show numreactions(rn) # Apprx. number of terms in the ODE
@show numparams(rn) # Number of Parameters
```

```
numspecies(rn) = 1122
numreactions(rn) = 24388
numparams(rn) = 128
128
```

0.1 Time ODE derivative function compilation

As compiling the ODE derivative functions has in the past taken longer than running a simulation, we first force compilation by evaluating these functions one time.

```
u  = copy(u_0)
```

```
du = similar(u)
@timeit to "ODERHS Eval1" oprob.f(du,u,p,0.)
@timeit to "ODERHS Eval2" oprob.f(du,u,p,0.)

# force compilation for dense and sparse problem rhs
densejacprob.f(du,u,p,0.)
sparsejacprob.f(du,u,p,0.)

J = zeros(length(u),length(u))
@timeit to "DenseJac Eval1" densejacprob.f.jac(J,u,p,0.)
@timeit to "DenseJac Eval2" densejacprob.f.jac(J,u,p,0.)
```

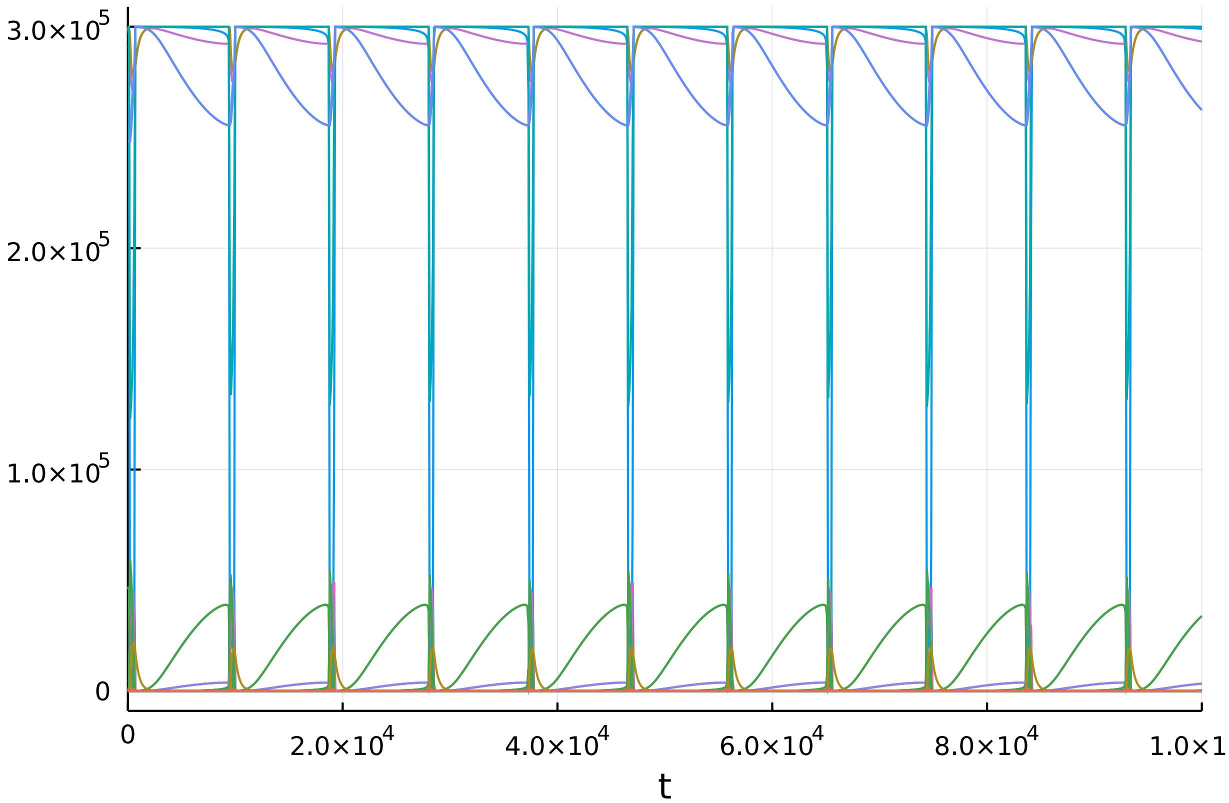
```
Error: syntax: expression too large
```

```
Js = similar(sparsejacprob.f.jac_prototype)
@timeit to "SparseJac Eval1" sparsejacprob.f.jac(Js,u,p,0.)
@timeit to "SparseJac Eval2" sparsejacprob.f.jac(Js,u,p,0.)
show(to)
```

```
Error: syntax: expression too large
```

0.2 Picture of the solution

```
sol = solve(oprob, CVODE_BDF(), saveat=tf/1000., reltol=1e-5, abstol=1e-5)
plot(sol, legend=false, fmt=:png)
```



For these benchmarks we will be using the time-series error with these saving points since the final time point is not well-indicative of the solution behavior (capturing the oscillation is the key!).

0.3 Generate Test Solution

```
@time sol = solve(oprob, CVODE_BDF(), abstol=1/10^12, reltol=1/10^12)
test_sol  = TestSolution(sol)
```

```
641.600334 seconds (4.71 M allocations: 2.204 GiB, 0.40% gc time, 0.09% com
pilation time)
retcode: Success
Interpolation: 3rd order Hermite
t: nothing
u: nothing
```

0.4 Setups

```
abstols = 1.0 ./ 10.0 .^ (5:8)
reltols = 1.0 ./ 10.0 .^ (5:8);
setups = [
          #Dict(:alg=>Rosenbrock23(autodiff=false)),
          Dict(:alg=>TRBDF2(autodiff=false)),
          Dict(:alg=>CVODE_BDF()),
          Dict(:alg=>CVODE_BDF(linear_solver=:LapackDense)),
          #Dict(:alg=>rodas()),
          #Dict(:alg=>radau()),
          #Dict(:alg=>Rodas4(autodiff=false)),
          #Dict(:alg=>Rodas5(autodiff=false)),
          Dict(:alg=>KenCarp4(autodiff=false)),
          #Dict(:alg=>RadauIIA5(autodiff=false)),
          #Dict(:alg=>lsoda()),
          ]
```

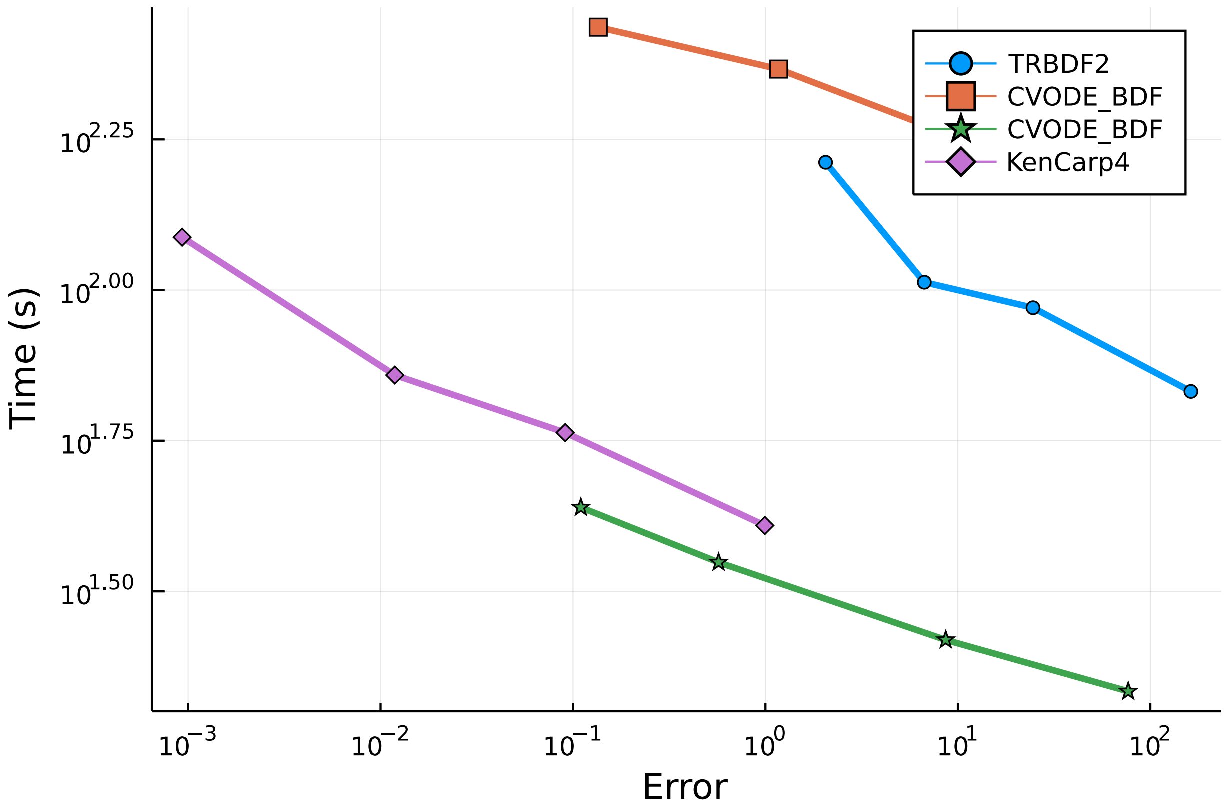
```
4-element Vector{Dict{Symbol, V} where V}:
 Dict{Symbol, OrdinaryDiffEq.TRBDF2{0, false, DiffEqBase.DefaultLinSolve, D
iffEqBase.NLNewton{Rational{Int64}, Rational{Int64}, Rational{Int64}}, Data
Type}}(:alg => OrdinaryDiffEq.TRBDF2{0, false, DiffEqBase.DefaultLinSolve,
DiffEqBase.NLNewton{Rational{Int64}, Rational{Int64}, Rational{Int64}}, Dat
aType}(DiffEqBase.DefaultLinSolve(nothing, nothing), DiffEqBase.NLNewton{Ra
tional{Int64}, Rational{Int64}, Rational{Int64}}(1//100, 10, 1//5, 1//5), V
al{:forward}, true, :linear, :PI))
 Dict{Symbol, Sundials.CVODE_BDF{:Newton, :Dense, Nothing, Nothing}}(:alg =
> Sundials.CVODE_BDF{:Newton, :Dense, Nothing, Nothing}(0, 0, 0, false, 10,
 5, 7, 3, 10, nothing, nothing, 0))
 Dict{Symbol, Sundials.CVODE_BDF{:Newton, :LapackDense, Nothing, Nothing}}(
:alg => Sundials.CVODE_BDF{:Newton, :LapackDense, Nothing, Nothing}(0, 0, 0
, false, 10, 5, 7, 3, 10, nothing, nothing, 0))
 Dict{Symbol, OrdinaryDiffEq.KenCarp4{0, false, DiffEqBase.DefaultLinSolve,
 DiffEqBase.NLNewton{Rational{Int64}, Rational{Int64}, Rational{Int64}}, Da
taType}}(:alg => OrdinaryDiffEq.KenCarp4{0, false, DiffEqBase.DefaultLinSol
ve, DiffEqBase.NLNewton{Rational{Int64}, Rational{Int64}, Rational{Int64}},
 DataType}(DiffEqBase.DefaultLinSolve(nothing, nothing), DiffEqBase.NLNewto
n{Rational{Int64}, Rational{Int64}, Rational{Int64}}(1//100, 10, 1//5, 1//5
), Val{:forward}, true, :linear, :PI))
```

0.5 Automatic Jacobian Solves

Due to the computational cost of the problem, we are only going to focus on the methods which demonstrated computational efficiency on the smaller biochemical benchmark problems. This excludes the exponential integrator, stabilized explicit, and extrapolation classes of methods.

First we test using auto-generated Jacobians (finite difference)

```
wp = WorkPrecisionSet(oprob,abstols,reltols,setups;error_estimate=:l2,
                      saveat=tf/10000.,appxsol=test_sol,maxiters=Int(1e5),numruns=1)
plot(wp)
```



0.6 Analytical Jacobian

Now we test using the generated analytic Jacobian function.

```
wp = WorkPrecisionSet(densejacprob,abstols,reltols,setups;error_estimate=:l2,
                      saveat=tf/10000.,appxsol=test_sol,maxiters=Int(1e5),numruns=1)
plot(wp)
```

```
Error: syntax: expression too large
```

0.7 Sparse Jacobian

Finally we test using the generated sparse analytic Jacobian function.

```
setups = [
          #Dict(:alg=>Rosenbrock23(autodiff=false)),
          Dict(:alg=>TRBDF2(autodiff=false)),
          #Dict(:alg=>CVODE_BDF(linear_solver=:KLU)),
          #Dict(:alg=>rodas()),
          #Dict(:alg=>radau()),
          #Dict(:alg=>Rodas4(autodiff=false)),
```

```
          #Dict(:alg=>Rodas5(autodiff=false)),
          Dict(:alg=>KenCarp4(autodiff=false)),
          #Dict(:alg=>RadauIIA5(autodiff=false)),
          #Dict(:alg=>lsoda()),
          ]
wp = WorkPrecisionSet(sparsejacprob,abstols,reltols,setups;error_estimate=:l2,
                      saveat=tf/10000.,appxsol=test_sol,maxiters=Int(1e5),numruns=1)
plot(wp)
```

```
Error: syntax: expression too large
```