

Modern Differential Equations Solver Software: Where We Are and Where We're Headed

Chris Rackauckas

Massachusetts Institute of Technology

A lot of people solve differential equations every single day

How has this gotten better, how has it stayed the same?

Non-Stiff Equations

- Non-stiff equations are generally thought to have been “solved”
- Standard methods: Runge-Kutta and Adams-Bashforth-Moulton
 - **ABM is implicit!!!!!!!**
- Tradeoff: ABM minimizes function calls while RK maximizes steps.
- In the end, Runge-Kutta seems to have “won”
 - Optimization of the leading truncation coefficients
 - PI(D)-adaptivity
 - High order (8th , 9th , 14th !)

Simulating ODEs: RK4

$$y' = f(t, y)$$

You know $y(t_n)$ and what to find $y(t_{n+1})$

$$t_{n+1} = t_n + h$$

Euler: $y_{n+1} = y_n + hf(t_n, y_n)$

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right),$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),$$

$$k_4 = f(t_n + h, y_n + hk_3).$$

$$y_{n+1} = y_n + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4),$$

$$t_{n+1} = t_n + h$$

The Structure of a Runge-Kutta Method

$$y_{n+1} = y_n + h \sum_{i=1}^s b_i k_i,$$

where

$$k_1 = f(t_n, y_n),$$

$$k_2 = f(t_n + c_2 h, y_n + h(a_{21} k_1)),$$

$$k_3 = f(t_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)),$$

\vdots

$$k_s = f(t_n + c_s h, y_n + h(a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1})).$$

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
\vdots	\vdots		\ddots		
c_s	a_{s1}	a_{s2}	\cdots	$a_{s,s-1}$	
	b_1	b_2	\cdots	b_{s-1}	b_s

4th Order Runge-Kuttas as Butcher Tableaus

“The Runge-Kutta Method”

0					
1/2		1/2			
1/2		0	1/2		
1		0	0	1	
<hr/>					
		1/6	1/3	1/3	1/6

Runge’s 3/8’s method

0					
1/3		1/3			
2/3		-1/3	1		
1		1	-1	1	
<hr/>					
		1/8	3/8	3/8	1/8

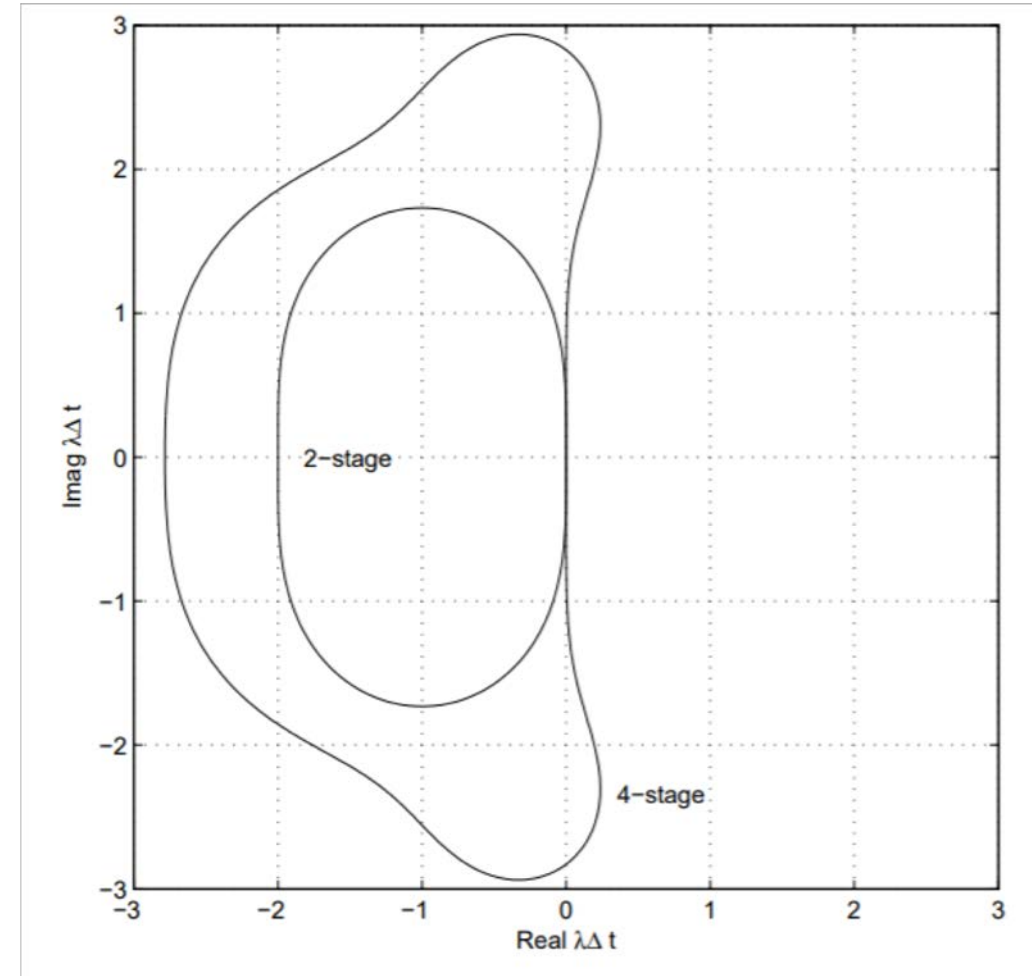
Ways to Judge an RK Method

Optimization of next order coefficients

$$\begin{aligned}
 b_2 a_{21} + b_3 [a_{31} + a_{32}] + b_4 [a_{41} + a_{42} + a_{43}] &= 1/2 \\
 b_2 a_{21}^2 + b_3 [a_{31} + a_{32}]^2 + b_4 [a_{41} + a_{42} + a_{43}]^2 &= 1/3 \\
 b_2 a_{22} + b_3 [a_{21} a_{32} + a_{33}] + b_4 [a_{21} a_{42} + a_{43} (a_{31} + a_{32}) + a_{44}] &= 1/6 \\
 b_2 a_{21}^3 + b_3 [a_{31} + a_{32}]^3 + b_4 [a_{41} + a_{42} + a_{43}]^3 &= 1/4 \\
 b_2 a_{21} a_{22} + b_3 \left[\frac{1}{2} a_{21}^2 a_{32} + (a_{31} + a_{32})(a_{21} a_{32} + a_{33}) \right] + \frac{1}{2} b_4 [a_{21}^2 a_{42} \\
 + a_{43} (a_{31} + a_{32})^2 + 2(a_{41} + a_{42} + a_{43})(a_{21} a_{42} + (a_{31} + a_{32}) a_{43} + a_{44})] &= 1/6 \\
 b_3 a_{22} a_{32} + b_4 [a_{21} a_{32} a_{43} + a_{22} a_{42} + a_{33} a_{43}] &= 1/24 \\
 b_2 a_{21}^4 + b_3 [a_{31} + a_{32}]^4 + b_4 [a_{41} + a_{42} + a_{43}]^4 &= 1/5 \\
 3b_2 a_{21}^2 a_{22} + b_3 [a_{21}^3 a_{32} + 3(a_{31} + a_{32})^2 (a_{21} a_{32} + a_{33})] + b_4 [a_{21}^3 a_{42} \\
 + (a_{31} + a_{32})^3 a_{43} + 3(a_{41} + a_{42} + a_{43})^2 (a_{21} a_{42} + (a_{31} + a_{32}) a_{43} + a_{44})] &= 7/20 \\
 b_3 a_{21}^2 a_{32} (a_{31} + a_{32}) + b_4 [(a_{41} + a_{42} + a_{43})(a_{21}^2 a_{42} + (a_{31} + a_{32})^2 a_{43})] &= 1/15 \\
 \frac{1}{2} b_2 a_{22}^2 + b_3 [a_{21} a_{32} (\frac{1}{2} a_{21} a_{32} + a_{22} + a_{33}) + a_{22} a_{32} (a_{31} + a_{32}) + \frac{1}{2} a_{33}^2] \\
 + b_4 [\frac{1}{2} a_{21}^2 (a_{32} a_{43} + a_{42}^2) + (a_{31} + a_{32})(a_{21} (a_{32} a_{43} + a_{42} a_{43}) + a_{43} (a_{33} + a_{44}) \\
 + \frac{1}{2} (a_{31} + a_{32}) a_{43}^2) + a_{21} a_{42} (a_{22} + a_{44}) + (a_{21} a_{32} a_{43} + a_{22} a_{42} \\
 + a_{33} a_{43})(a_{41} + a_{42} + a_{43}) + \frac{1}{2} a_{44}^2] &= 11/120 \\
 b_4 a_{22} a_{32} a_{43} &= 1/120
 \end{aligned}$$

7

Stability



Dormand-Prince 5th Order (1980)

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40

Adaptivity

- These Runge-Kutta methods have also been tuned for adaptive stepsizes
 - Embedded methods use the same stages k_i in order to get two solutions, u_n and \widetilde{u}_n .
 - The difference is an error estimate: $E_n = \frac{\|u_n - \widetilde{u}_n\|}{abstol + (reltol)|u_n|}$
 - If $E_n < 1$, accept the step, otherwise reject the step
 - Change the timestep. There are many methods!
 - Simplest is akin to proportional control: $\Delta t_{new} = \frac{\Delta t}{E_n}$
 - PI-adaptivity brings in previous errors to smooth out the time steps
 - Changing Δt can decrease stability!

Dense Output

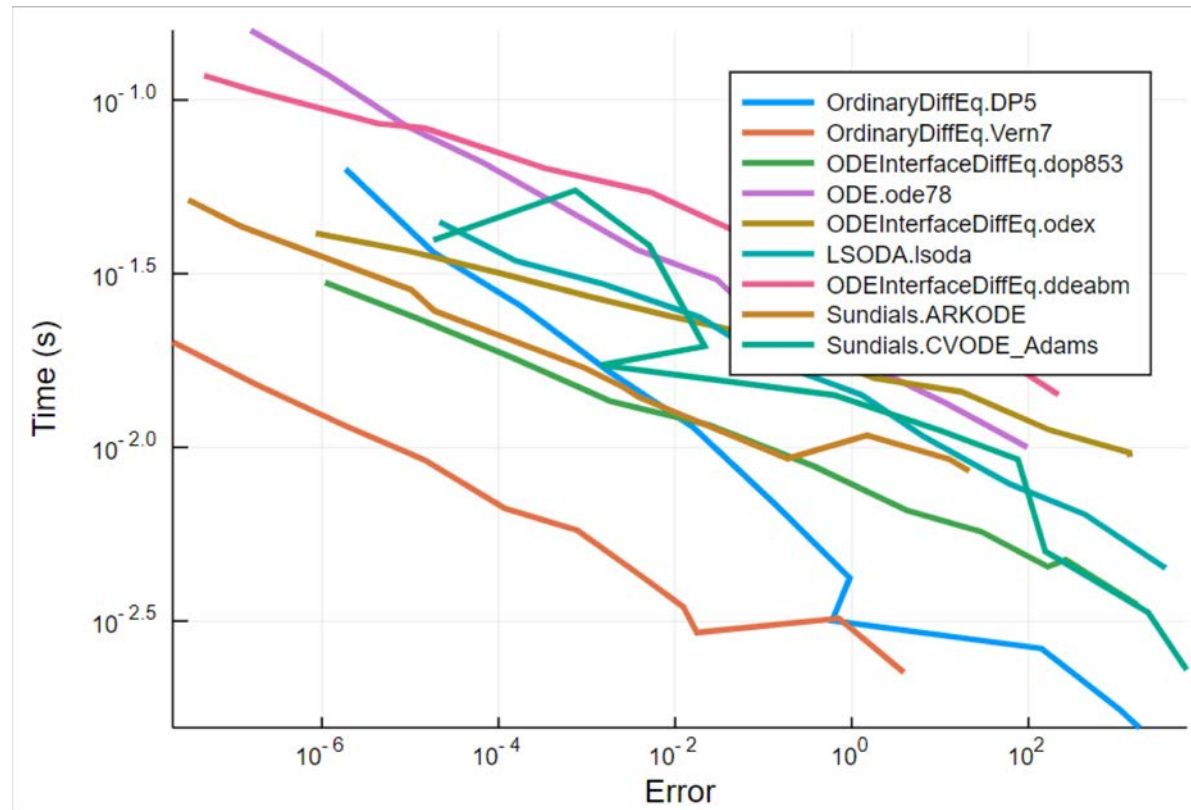
- Dense (continuous) output can also be embedded into the numerical method.
- Simplest method: Hermite interpolation
 - $u_{n+\theta} = (1 - \theta)u_n + \theta u_{n+1} + \theta(\theta - 1)((1 - 2\theta)(u_{n+1} - u_n) + (\theta - 1)\Delta t u'_n + \theta \Delta t u'_{n+1})$
 - Only uses the values and derivatives at the endpoints!

$$\begin{aligned}\tilde{b}_1 &= -1.0530884977290216t(t - 1.3299890189751412)(t^2 - 1.4364028541716351t + 0.7139816917074209) \\ \tilde{b}_2 &= 0.1017t^2(t^2 - 2.1966568338249754t + 1.2949852507374631) \\ \tilde{b}_3 &= 2.490627285651252793t^2(t^2 - 2.38535645472061657t + 1.57803468208092486) \\ \tilde{b}_4 &= -16.54810288924490272(t - 1.21712927295533244)(t - 0.61620406037800089)t^2 \\ \tilde{b}_5 &= 47.37952196281928122(t - 1.203071208372362603)(t - 0.658047292653547382)t^2 \\ \tilde{b}_6 &= -34.87065786149660974(t - 1.2)(t - 0.66666666666666667)t^2 \\ \tilde{b}_7 &= 2.5(t - 1)(t - 0.6)t^2.\end{aligned}$$

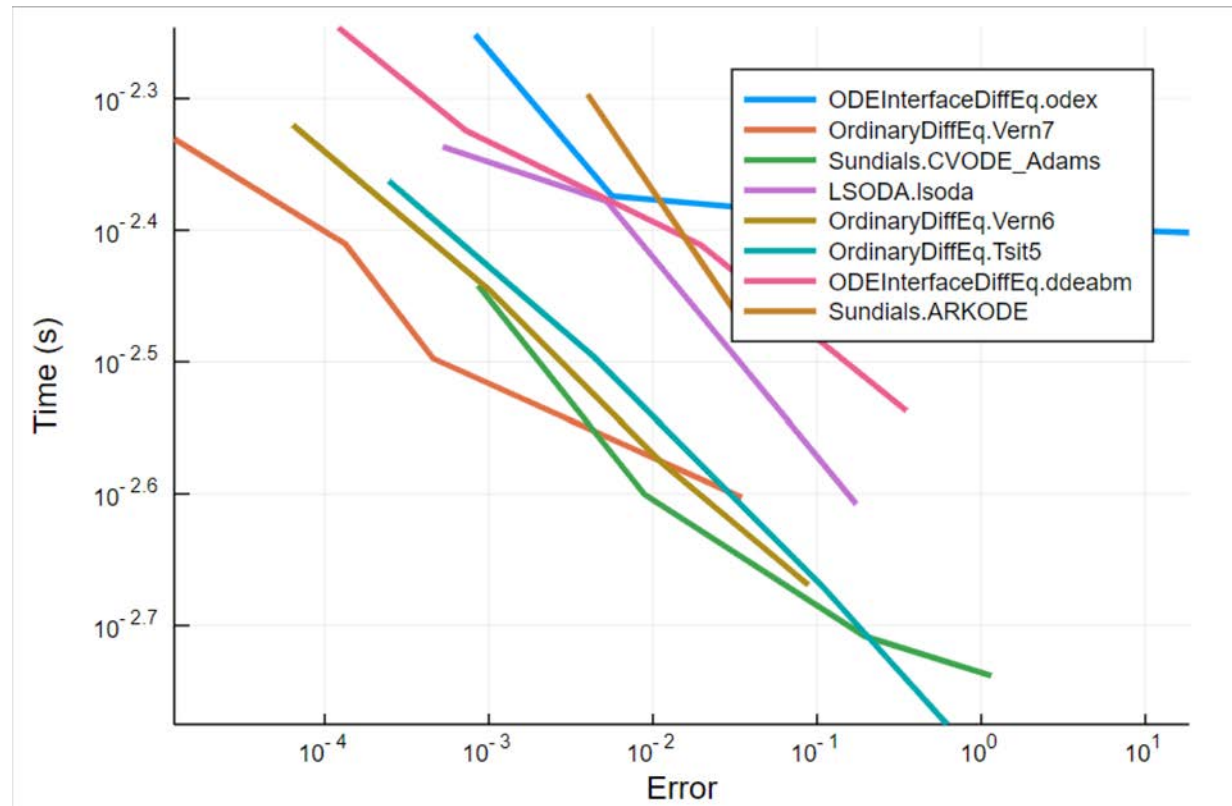
RK methods are still being improved!

- Optimizing coefficients can be done not just in general, but also to applications
 - Recent methods, Tsit5 and Vern#, reduce the number of assumptions made in coefficient optimization, leading to more optimal solutions (>2010)
 - Methods specialized for wave equations, low-dispersion results, extended monotonicity equation for PDEs (SSPRK), etc. are hot topics in new high order Runge-Kutta methods

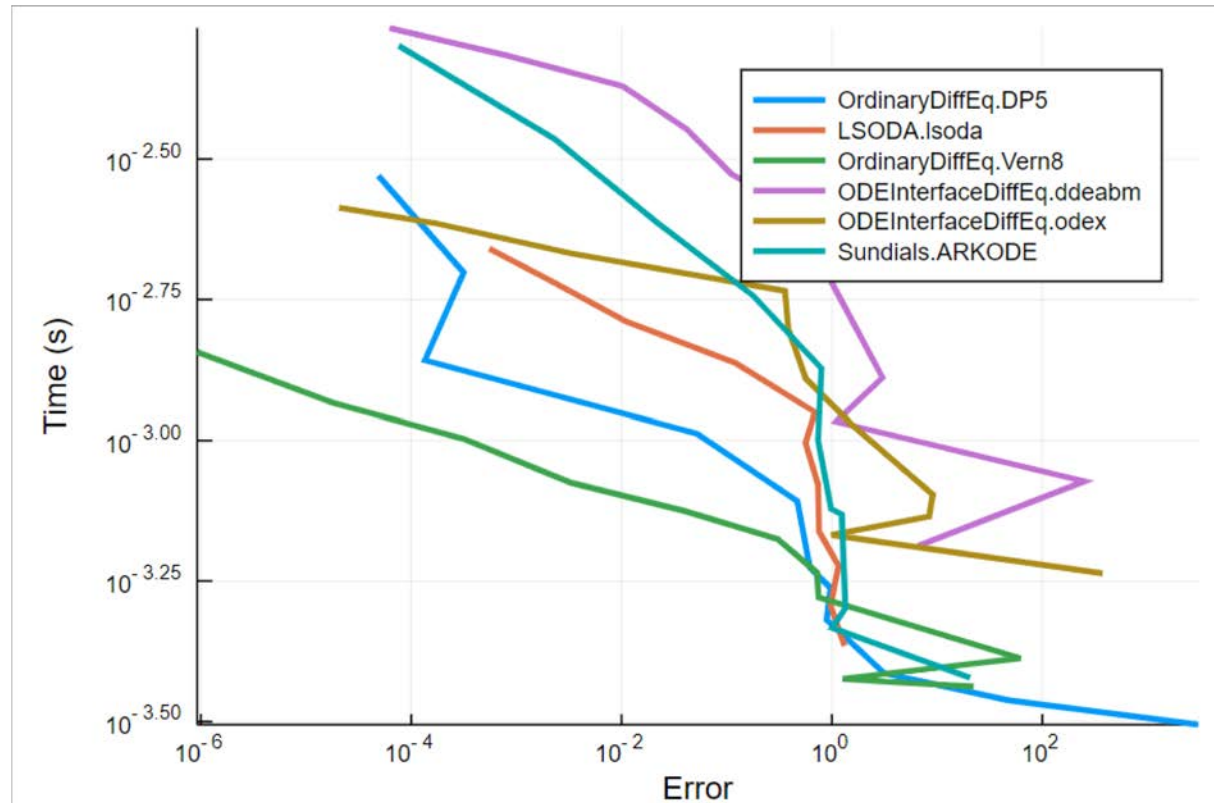
100x100 Linear ODEs



Pleiades Problem



3-Body Problem (CVODE_Adams fails)



Minor improvements in DifferentialEquations.jl

- FMA (fused multiply-add)
- SIMD
- fastmath on adaptivity parameters
- Full inlining of user function

But can we do more?

- Parallelism is not well-exploited.

3 forms of parallelism in diffeqs

- **Within-Method parallelism**
 - Parallelize the operations within the method of a differential equation solver or within the derivative function f
 - Methods can be chosen to have more within-method parallelism
- **Parallelism in time**
 - Parallelize across time, then relax to a solution
 - May be hard to converge! May not be efficient!
- **Parameter Parallelism**
 - If people are solving the same system thousands of times with different initial conditions and parameters, this is a good level to parallelize at!

Pervasive Allowance of Within-Method parallelism through Julia

- Julia's broadcast system allows an array type to define its actions
- If an array chooses to parallelize its elementwise (broadcasted) operations, they will be broadcasted
- If an entire solver is written to never index and always broadcast, then all internal operations will be the user-defined actions
- Result: full parallelism in the ODE solver
 - GPU-based arrays stay on the GPU
 - Distributed arrays stay distributed
 - Multithreaded arrays will auto-multithread the operations of the method

Example of a Broadcast-Based Internal

Zero GPU/Distributed message passing done by the solver!

```
@muladd function perform_step!(integrator, cache::Tsit5Cache, repeat_step=false)
@unpack u, uprev, u5, p = integrator
@unpack c1, c2, c3, c4, c5, c6, a21, a31, a32, a41, a42, a43, a51, a52, a53, a54, a61, a62, a63, a64, a65, a
@unpack k1, k2, k3, k4, k5, k6, k7, utilde, tmp, atmp = cache
a = dt*a21
@. tmp = uprev+a*k1
f(k2, tmp, p, t+c1*dt)
@. tmp = uprev+dt*(a31*k1+a32*k2)
f(k3, tmp, p, t+c2*dt)
@. tmp = uprev+dt*(a41*k1+a42*k2+a43*k3)
f(k4, tmp, p, t+c3*dt)
@. tmp = uprev+dt*(a51*k1+a52*k2+a53*k3+a54*k4)
f(k5, tmp, p, t+c4*dt)
@. tmp = uprev+dt*(a61*k1+a62*k2+a63*k3+a64*k4+a65*k5)
f(k6, tmp, p, t+dt)
@. u = uprev+dt*(a71*k1+a72*k2+a73*k3+a74*k4+a75*k5+a76*k6)
f(k7, u, p, t+dt)
if integrator.opts.adaptive
@. utilde = dt*(btilde1*k1 + btilde2*k2 + btilde3*k3 + btilde4*k4 + btilde5*k5 + btilde6*k6)
calculate_residuals!(atmp, utilde, uprev, u, integrator.opts.abstol, integrator.opts.
integrator.EEst = integrator.opts.internalnorm(atmp,t)
end
end
```

Pros/Cons of “Array-Based Parallelism”

- Pros:
 - It's a style that's already used a lot
 - Big PDE simulations, climate simulations
 - Dead simple to get nearly 100% efficient (in Julia!)
- Cons:
 - Only efficient for LARGE ODE systems

What about changing the method for more within-method parallelism?

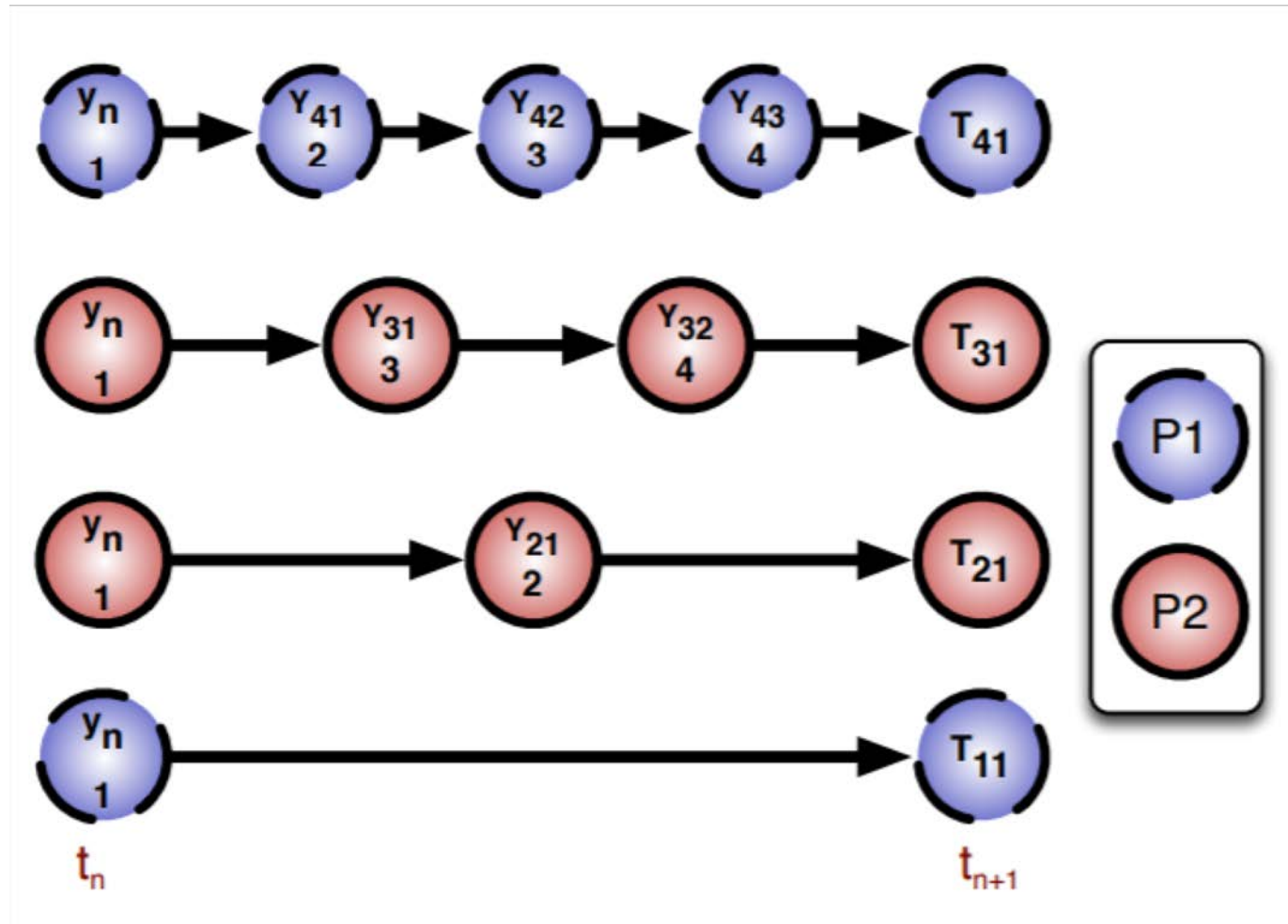
Parallel Runge-Kutta methods

$$\mathbf{A} = \begin{bmatrix} 0 & & & & \\ \times & 0 & & & \\ \times & 0 & 0 & & \\ \times & \times & \times & 0 & \\ \times & \times & \times & 0 & 0 \end{bmatrix}$$

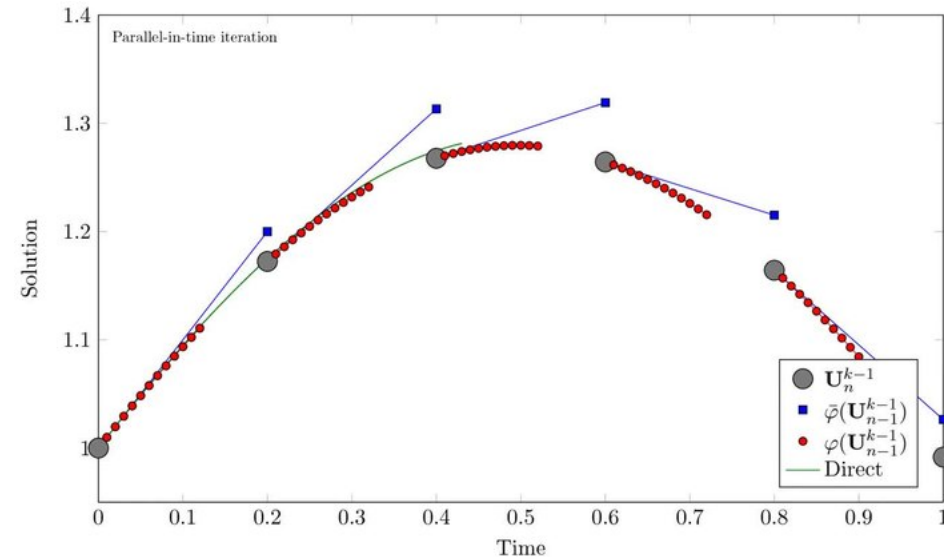
5 stages

But only 3 steps in parallel

Multithreading Extrapolation



Parareal Algorithms – Parallel in Time



Parameter Parallelism

- Naïve: Take the ODE solver and run it in parallel many times
 - This is fairly efficient!
- Next level: compile the ODE solver to a GPU kernel, and then call that GPU kernel on an array of parameters
 - Thousands of ODE solves per computer!
 - Limiting factor: memory

Intermediate Conclusion: That's just non-stiff ODEs (and not even all of it)

Even with non-stiff methods, we have already improved a lot over the older Fortran methods. And there's still a lot more that we can do.

Stiff ODEs: Fall of the BDF

What's coming to get GEAR's method.

Backwards Differentiation Formulae

-
- BDF1: $y_{n+1} - y_n = hf(t_{n+1}, y_{n+1})$ (this is the backward Euler method)
 - BDF2: $y_{n+2} - \frac{4}{3}y_{n+1} + \frac{1}{3}y_n = \frac{2}{3}hf(t_{n+2}, y_{n+2})$
 - BDF3: $y_{n+3} - \frac{18}{11}y_{n+2} + \frac{9}{11}y_{n+1} - \frac{2}{11}y_n = \frac{6}{11}hf(t_{n+3}, y_{n+3})$
 - BDF4: $y_{n+4} - \frac{48}{25}y_{n+3} + \frac{36}{25}y_{n+2} - \frac{16}{25}y_{n+1} + \frac{3}{25}y_n = \frac{12}{25}hf(t_{n+4}, y_{n+4})$
 - BDF5: $y_{n+5} - \frac{300}{137}y_{n+4} + \frac{300}{137}y_{n+3} - \frac{200}{137}y_{n+2} + \frac{75}{137}y_{n+1} - \frac{12}{137}y_n = \frac{60}{137}hf(t_{n+5}, y_{n+5})$
 - BDF6: $y_{n+6} - \frac{360}{147}y_{n+5} + \frac{450}{147}y_{n+4} - \frac{400}{147}y_{n+3} + \frac{225}{147}y_{n+2} - \frac{72}{147}y_{n+1} + \frac{10}{147}y_n = \frac{60}{147}hf(t_{n+6}, y_{n+6})$

Methods with $s > 6$ are not zero-stable so they cannot be used.^[4]

Evolution of Gear's Method

- GEAR: Original code. Adaptive order adaptive time via interpolation
 - Lowers the stability!
- LSODE series: update of GEAR
 - Adds rootfinding, Krylov, etc
- VODE: Variable-coefficient form
 - No interpolation necessary.
- CVODE: VODE rewritten in C++
 - Adds sensitivity analysis

Problems with BDF

BDF is a multistep method

Needs “Startup Steps”

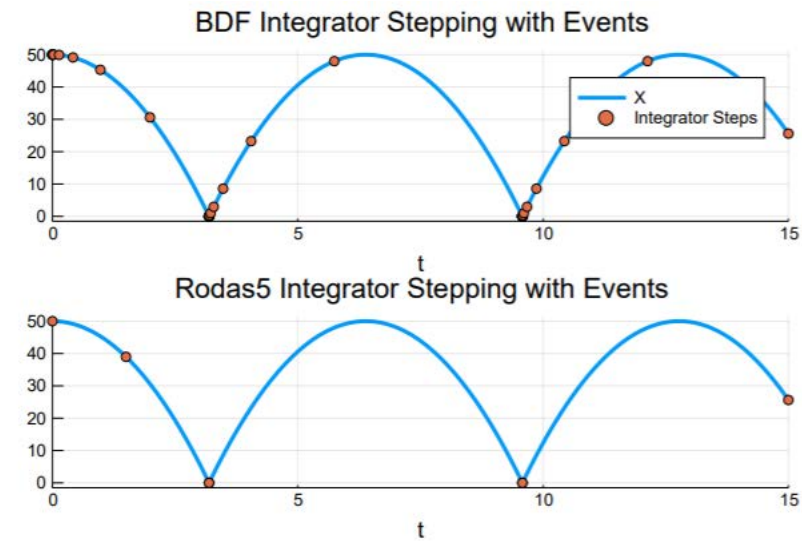
Inefficient with events

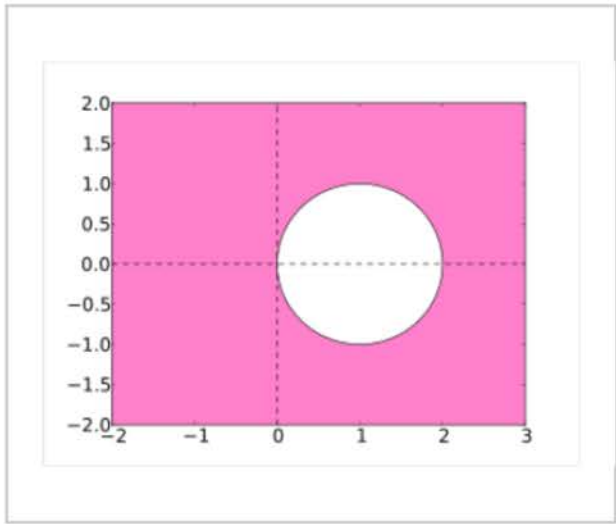
It is only L-stable up to 2nd order

Has high truncation error coefficients

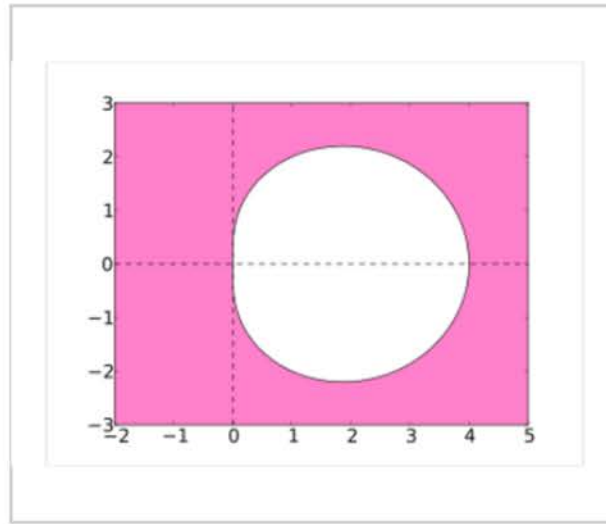
Implicit

Requires good step predictors

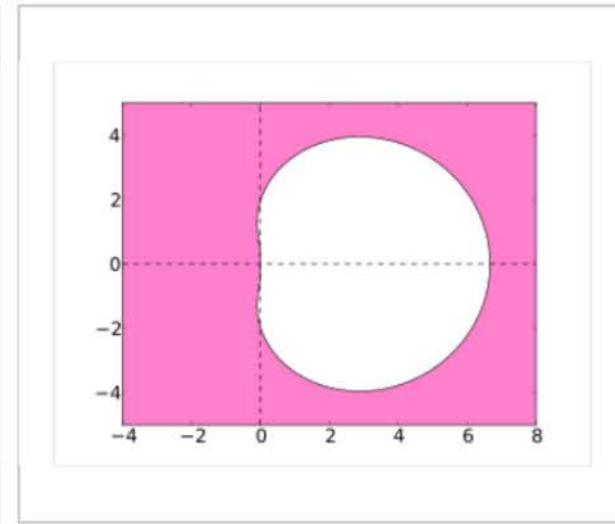




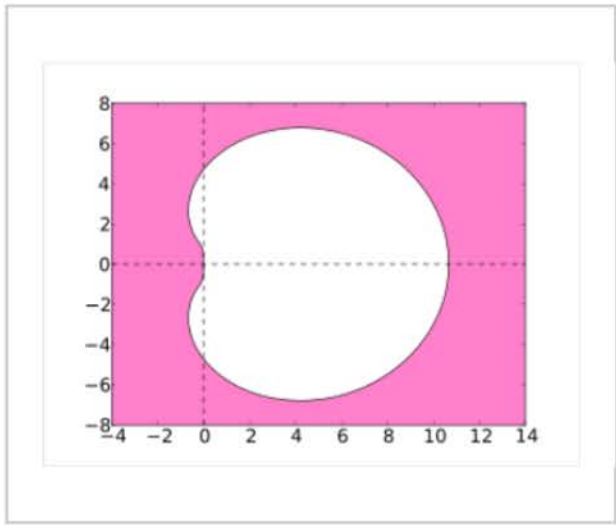
BDF1



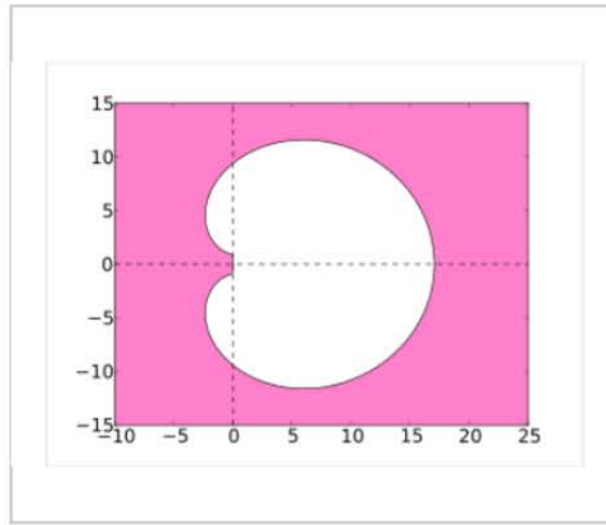
BDF2



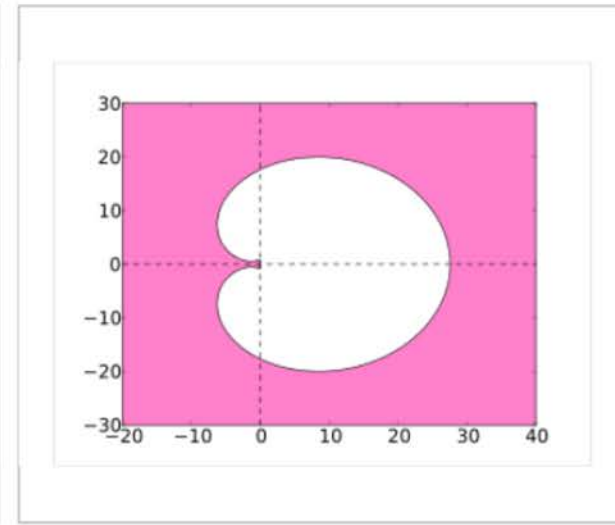
BDF3



BDF4



BDF5

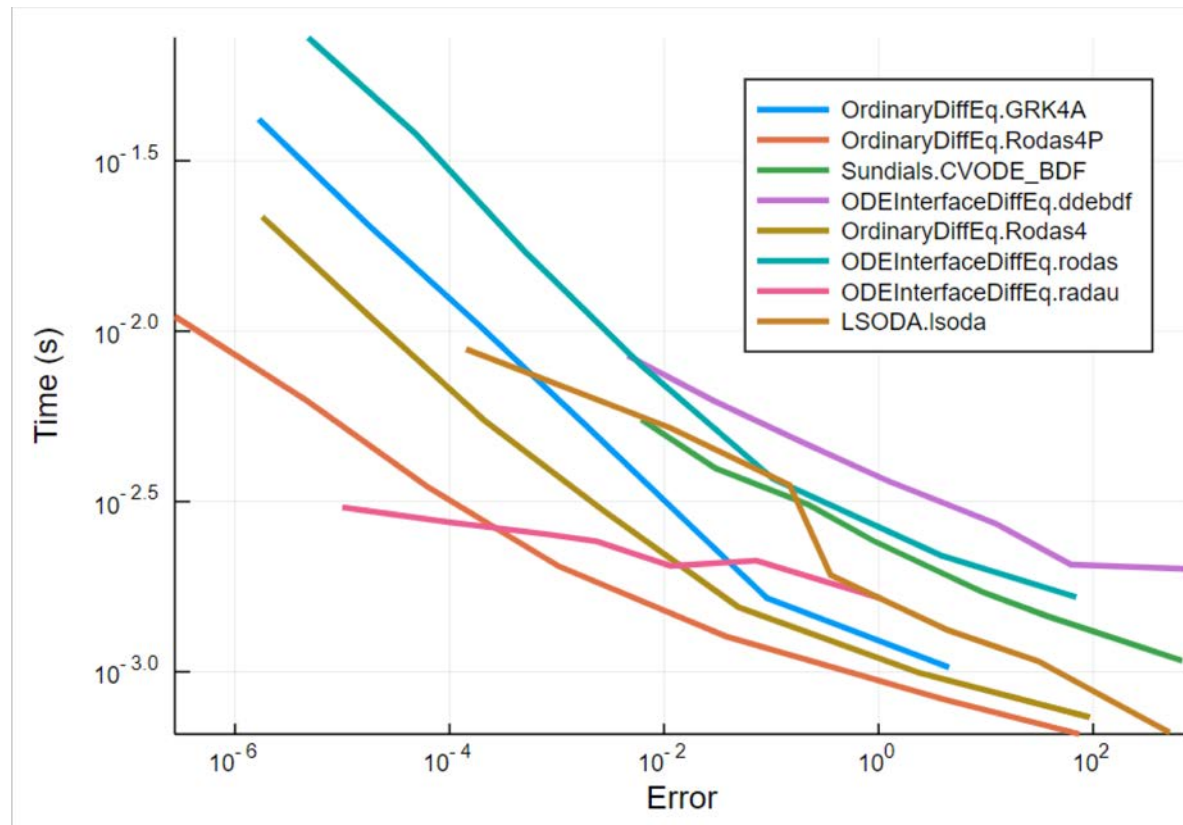


BDF6

But in 2019, what can we exploit?

Sparse factorizations, Krylov exponential linear algebra, IMEX, Approximate Factorization, ETC.

Orego Benchmarks



Rosenbrock Methods

Aren't new! (ode23s)

Can fix a lot of problems:

- Exploit sparse factorization

- No step predictions required

- Can optimize coefficients to high order

Con: Needs accurate Jacobians

$$Wk_1 = F(y_n)$$

$$Wk_2 = F\left(y_n + \frac{2}{3}hk_1\right) - \frac{4}{3}hdJk_1$$

$$y_{n+1} = y_n + \frac{h}{4}(k_1 + 3k_2)$$

Automatic Differentiation in a nutshell

- Numerical differentiation is numerically bad because you're dividing by a small number. Can this be avoided?
- Early idea: instead of using a real-valued difference, when f is real-valued but complex analytic, use the following identity:

$$f'(x) \approx \Im \left\{ \frac{f(x + ih)}{h} \right\}.$$

- Claim: the numerical stability of this algorithm matches that of f
- Automatic differentiation then scales this idea to multiple dimensions
- One implementation: use Dual numbers $x = a + b\epsilon$ where $\epsilon^2 = 0$ (smooth infinitesimal arithmetic). Define $f(x) = f(a) + f'(a)b\epsilon$ (chain rule).

Differentiable Programming

$$(x + x'\varepsilon) + (y + y'\varepsilon) = x + y + (x' + y')\varepsilon$$

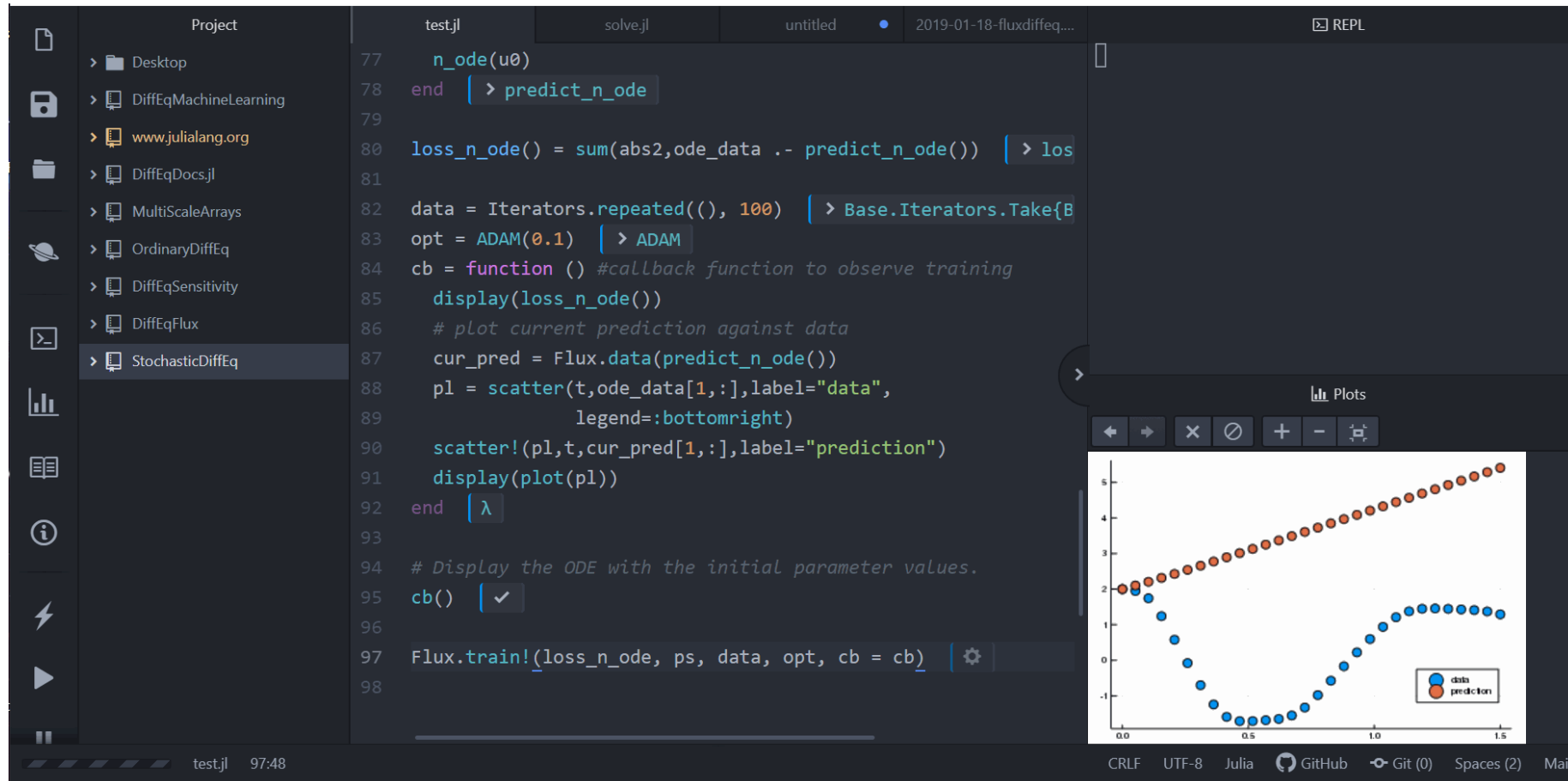
$$(x + x'\varepsilon) \cdot (y + y'\varepsilon) = xy + xy'\varepsilon + yx'\varepsilon + x'y'\varepsilon^2 = xy + (xy' + yx')\varepsilon$$

- Claim: if you recompiled your entire program to do Dual arithmetic, then the output of your program is a Dual number which computes both the original value and derivative simultaneously (to machine accuracy).
- As described, this is known as operator overloading forward-mode automatic differentiation (AD). There are also computational graph and AST-based AD implementations. In addition, there are “adjoint” or reverse-mode automatic differentiation which specifically produce gradients of cost functions with better scaling properties
- “Backpropagation” of neural networks is simple reverse-mode AD on some neural network program.

Differentiable Programming in Julia

- I have defined this implementation of automatic differentiation as “the way you would change every arithmetic operation of a program if you wanted to calculate the derivative.
- The differential equation solvers and PuMaS are all implemented as generic algorithms in Julia which are generic with respect to the Number and AbstractArray types that are used
- ForwardDiff.jl defines a Dual number type for forward-mode automatic differentiation, Flux.jl defines a Tracker number type for reverse-mode automatic differentiation.
- If you put these into these simulation tools, a new algorithm is automatically generated that propagates the solution and its derivatives through every step of the code.

Side note: this same technology let's us fuse with neural networks



The screenshot displays a Julia IDE interface with a code editor on the left and a REPL on the right. The code in the editor defines a function `n_ode(u0)` and a loss function `loss_n_ode()`. It uses `Flux` for training a neural network model `ps` to fit data points. The plot shows the training progress, with data points (blue circles) and predictions (orange circles) plotted against time `t`. The data points form a curve that starts at approximately (0.0, 2.0), dips to a minimum of about -1.5 at `t=0.5`, and then rises to about 1.5 at `t=1.5`. The predictions (orange circles) form a smooth curve that starts at approximately (0.0, 2.0) and increases to about 5.0 at `t=1.5`.

```
77 n_ode(u0)
78 end
79
80 loss_n_ode() = sum(abs2,ode_data .- predict_n_ode())
81
82 data = Iterators.repeated((), 100)
83 opt = ADAM(0.1)
84 cb = function () #callback function to observe training
85     display(loss_n_ode())
86     # plot current prediction against data
87     cur_pred = Flux.data(predict_n_ode())
88     p1 = scatter(t,ode_data[1,:],label="data",
89                 legend=:bottomright)
90     scatter!(p1,t,cur_pred[1,:],label="prediction")
91     display(plot(p1))
92 end
93
94 # Display the ODE with the initial parameter values.
95 cb()
96
97 Flux.train!(loss_n_ode, ps, data, opt, cb = cb)
98
```

ODE Problems can fall into different classes

Physical Modeling

SecondOrderODEProblem(f,u0,tspan,p)

- $u'' = f(u, p, t)$

PartitionedODEProblem(f1,f2,v0,u0,tspan,p)

- $v' = f_1(t, u)$

- $u' = f_2(v)$

HamiltonianODEProblem(H,p0,q0,tspan,p)

PDE Discretizations

SplitODEProblem(f1,f2,u0,tspan,p) (IMEX)

- $u' = f_1(u, p, t) + f_2(u, p, t)$

SemilinearODEProblem(A,f2,u0,tspan,p)

- $u' = Au + f(u, p, t)$

LocalSemilinearODEProblem(A,f2,u0,tspan,p)

$$u' = Au + f(u, p, t)$$

Exponential Runge-Kutta

Explicit methods for stiff equations

Small enough: Build matrix
exponential

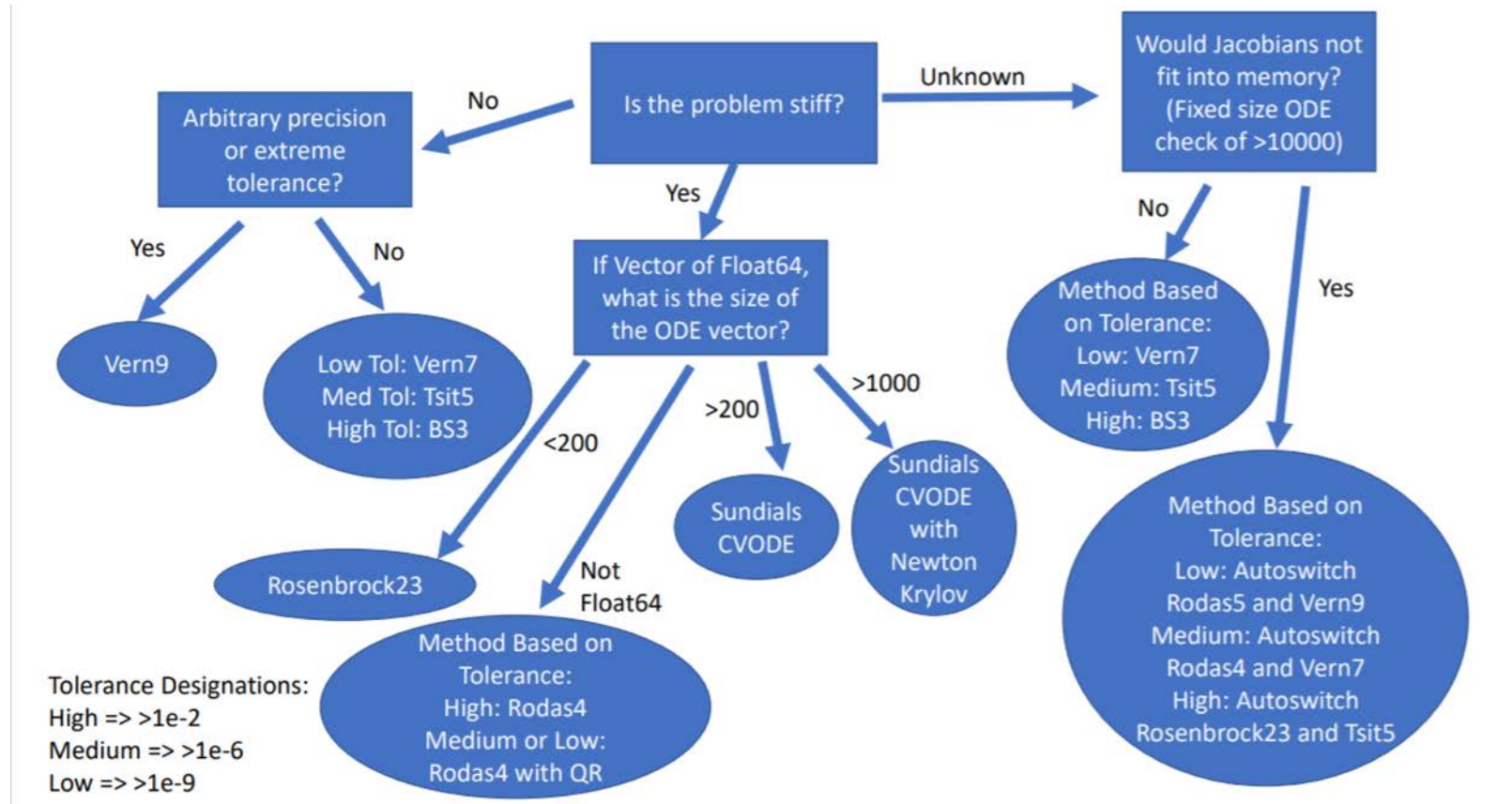
Large enough: Krylov $\exp(t \cdot A) \cdot v$

$$U_{ni} = e^{c_i h_n L_n} u_n + h_n \sum_{j=1}^{i-1} a_{ij}(h_n L_n) N_n(U_{nj}),$$
$$u_{n+1} = e^{h_n L_n} u_n + h_n \sum_{i=1}^s b_i(h_n L_n) N_n(U_{ni})$$

Non-stiff and Stiff ODEs are far from solved if you really need the performance.

Plenty of methods were not mentioned here that are showing promise in research and in the DifferentialEquations.jl software

Putting it together for users: polyalgorithms



Conclusion

- Today you can solve ODEs
- Tomorrow you will likely be able to solve them much faster

Want a paid summer position? Want a paid part time position as a PuMaS/DiffEq developer?

- Contact me for Google Summer of Code or PuMaS development. No Julia experience is required for GSoC. Julia experience is required for PuMaS.
- <https://julialang.org/soc/ideas-page>

MIT OpenCourseWare
<https://ocw.mit.edu>

18.335J Introduction to Numerical Methods
Spring 2019

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.