Integrating equation solvers with probabilistic programming through differentiable programming

Chris Rackauckas
Director of Modeling and Simulation, Julia Computing
Research Affiliate, Co-PI of Julia Lab, Massachusetts Institute of Technology, CSAIL
Director of Scientific Research, Pumas-AI
Turing.jl arguably has both the most extensive differential equation solving support, and no support for differential equations at all.

Let me explain.
Mixing equation discovery into epidemic modeling workflows will revolutionize the field

1. What could “extensive” differential equation support even mean?

2. Does Turing.jl have it? And what does that mean about its developer community?
Problem: Lorenz equation on t in (0,100)

```plaintext
using OrdinaryDiffEq
function lorenz!(du, u, p, t)
    du[1] = 10.0(u[2] - u[1])
end
u0 = [1.0; 0.0; 0.0]
tspan = (0.0, 100.0)
prob = ODEProblem(lorenz!, u0, tspan)
sol = solve(prob, Tsit5())
using Plots;
plot(sol, vars=(1, 2, 3));
```
But there are lots of little optimizations that can be done
DifferentialEquations.jl is generally:

1. 50x faster than SciPy
2. 50x faster than MATLAB
3. 100x faster than R’s deSolve

When optimally JIT compiling Py/Mat/R

Foundation: Fast Differential Equation Solvers

### 1. Speed
### 2. Stability
### 3. Stochasticity
### 4. Adoints and Inference
### 5. Parallelism

DifferentialEquations.jl is generally:
- 50x faster than SciPy
- 50x faster than MATLAB
- 100x faster than R’s deSolve

When optimally JIT compiling Py/Mat/R


https://github.com/SciML/SciMLBenchmarks.jl
DifferentialEquations.jl is:

- Faster than C codes like CVODE and Fortran codes like LSODE/LSODA on stiff equations
- Has symbolic compilers to automatically improve numerical stability and performance of user code

This excludes the extra 2x from symbolics and 2x from sparse parallel compilation!

https://github.com/SciML/SciMLBenchmarks.jl


Speed alone does not give good robust differential equation solving.

There’s a lot more to the algorithms.
Multiscale Behavior

$$S = \frac{\max(|Re(\lambda)|)}{\min(|Re(\lambda)|)} (t_{final} - t_0)$$

“Stiff equations are problems for which explicit methods don't work.”

- Ernst Hairer
So you just use a stiff ODE solver? 2 Solvers and we’re done?

Okay, maybe more than two just so you can optimize the performance?
3 Solvers?

Magnetic Dipole PDE
Neither stiff nor non-stiff
Performance winner: ROCK2
Some problems need special integrators

- Symplectic integrators for long time Hamiltonian systems
- Magnus for $u' = A(t)u$
- Munthe-Kaas methods for $u' = A(u)u$
- Nystrom specializations for 2\textsuperscript{nd} order
- Exponential integrators for semilinear ODEs $u' = Au + f(u)$
- Implicit-Explicit (IMEX) methods for partly stiff equations
- Runge-Kutta-Chebyshev methods for semi-stiff equations

There are lots of special properties
There are lots of special properties

Some problems need special integrators

- Symplectic integrators for long time Hamiltonian systems
- Magnus for $u' = A(t)u$
- Munthe-Kaas methods for $u' = A(u)u$
- Nystrom specializations for 2nd order
- Exponential integrators for semilinear ODEs $u' = Au + f(u)$
- Implicit-Explicit (IMEX) methods for partly stiff equations
- Runge-Kutta-Chebyshev methods for semi-stiff equations

...
Julia’s DifferentialEquations.jl is at over 300 integrators, and there’s still more to do.

But that’s just adding more solvers, right?
Differential Equations Go Beyond ODEs

- Discrete equations (function maps, discrete stochastic (Gillespie/Markov) simulations)
- Ordinary differential equations (ODEs)
- Split and Partitioned ODEs (Symplectic integrators, IMEX Methods)
- Stochastic ordinary differential equations (SODEs or SDEs)
- Stochastic differential-algebraic equations (SDAEs)
- Random differential equations (RODEs or RDEs)
- Differential algebraic equations (DAEs)
- Delay differential equations (DDEs)
- Neutral, retarded, and algebraic delay differential equations (NDDEs, RDDEs, and DDAEs)
- Stochastic delay differential equations (SDDEs)
- Experimental support for stochastic neutral, retarded, and algebraic delay differential equations (SNDDDEs, SRDDDEs, and SDDAEs)
- Mixed discrete and continuous equations (Hybrid Equations, Jump Diffusions)
- (Stochastic) partial differential equations ((S)PDEs) (with both finite difference and finite element methods)

But if you keep adding solver choices, then you’re okay?
Eye-balling Index Issues: any algebraic equation should be dependent on some algebraic variable

\[ x' = v_x \]
\[ v_x' = T x \]
\[ y' = v_y \]
\[ v_y' = T y - g \]
\[ 0 = x^2 + y^2 - L^2 \]

Not solvable by standard numerical solvers!

\[ x' = v_x \]
\[ v_x' = x T \]
\[ y' = v_y \]
\[ v_y' = y T - g \]
\[ 0 = 2 (v_x^2 + v_y^2 + y(y T - g) + T x^2) \]

Easy to solve!

Differentiate the last equation twice, do a few substitutions...

States: \( x, v_x, y, v_y, \) and \( T \). Algebraic equation states: \( x \) and \( y \) (no \( T \)).

If you don’t know the details about why this makes a better numerical simulation, then you should be using ModelingToolkit.jl
DAE Index Reduction is Automatic with ModelingToolkit.jl

Let me fix that for you...

```julia
using DifferentialEquations, ModelingToolkit
using LinearAlgebra, Plots

function pendulum!(du, u, p, t)
    x, dx, y, dy, T = u
    g, L = p
du[1] = dx
du[2] = T*x
du[3] = dy
du[4] = T*y - g
du[5] = x^2 + y^2 - L^2
return nothing
end

pendulum_fun! = ODEFunction(pendulum!,
    mass_matrix=Diagonal([1,1,1,1,0]))

uθ = [1.8, 0, 0, 0, 0]
p = [9.8, 1]
tspan = (0, 10.0)
pendulum_prob = ODEProblem(pendulum_fun!, uθ, tspan, p)

@named traced_sys = modelingtoolkitize(pendulum_prob)

pendulum_sys = structural_simplify(traced_sys)
prob = ODAEProblem(pendulum_sys, [], tspan)
sol = solve(prob, Tsit5(), abstol=1e-8, reltol=1e-8)

plot(sol, vars=states(traced_sys))
```

structural_simplify: The God of Transforms

Standard equation solvers (IDA, DASKR, all of the Julia DAE solvers, etc.) cannot solve even a Cartesian pendulum without symbolic modification!
And when you finally get there, you have to keep customizing

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!

Some giant ODE

```
using DifferentialEquations, LinearAlgebra, SparseArrays

const N = 32
const xyd.bruusselator = range(0, stop=1, length=N)
brusselator_f(x, y, t) = (((x-0.3)^2 + (y-0.6)^2 <= 0.1^2) * (t >= 1.1) * 5.
limit(x, N) = a == N+1 ? 1 : a == 0 ? N : a
function brusselator_2d_loop(dx, u, p)
    A, B, alpha, dx = p
    alpha = alpha/dx^2
    @inbounds for I in CartesianIndices((N, N))
        i, j = Tuple(I)
        x, y = xyd.bruusselator[I[1]], xyd.bruusselator[I[2]]
        ip1, im1, ipj1, jm1 = limit(i+1, N), limit(i-1, N), limit(j+1, N), limit(j-1, N)
        dx[i,j,1] = alphaw(u[i,1,1] + u[ip1,1,1] + u[j,1,1] + u[jm1,1,1] - 4u[i,j,1]) + B + u[i,j,1]^2u[i,j,2] - (A + 1)u[i,j,1] + brusselator_f(x, y, t)
        dx[i,j,2] = alphaw(u[i,md1,1] + u[ip1,1,1] + u[j,1,1] + u[jm1,1,1] - 4u[i,j,2]) + A + u[i,j,1]^2u[i,j,2] - 2u[i,j,2]
    end

    p = (3.4, 1.0, step(xyd.bruusselator))

    function init.bruusselator_2d(xyd)
        N = length(xyd)
        u = zeros(N, N, 2)
        for I in CartesianIndices((N, N))
            x = xyd[I[1]]
            y = xyd[I[2]]
            u[i,1,1] = 22x(y(1-x))^3/2
            u[i,1,2] = 27x(x(1-x))^3/2
        end
        return u
    end

    u0 = init.bruusselator_2d(xyd.bruusselator)
    prob.ode.bruusselator_2d = ODEProblem(brusselator_2d_loop, u0,(0,(0.11,5),p)
```

https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/
And when you finally get there, you have to keep customizing
And when you finally get there, you have to keep customizing

Baseline with sparse Jacobian, pretty decent

Just pass different LinearSolve.jl algorithms to try different internal solvers. Non-trivial differences!

https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!
And when you finally get there, you have to keep customizing

Use the preconditioner interface for iLU with GMRES, chunked it down a few notches

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!

https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/
What About Partial Differential Equations (and Beyond?)

PDEs need lots of discretizers

- Physics-Informed NNs: NeuralPDE.jl
- Finite Difference: MethodOfLines.jl
- Neural Operators: NeuralOperators.jl
- Finite Volume: Trixi.jl
- Finite Element: Gridap.jl
- Pseudospectral: ApproxFun.jl
- High Dimension: HighDimPDE.jl

Etc. a bunch more issues to address...

```julia
using ModelingToolkit
import ModelingToolkit: Interval, infimum, supremum

@parameters x y
@variables u(..)
Dxx = Differential(x)^2
Dyy = Differential(y)^2

# 2D PDE
eq  = Dxx(u(x,y)) + Dyy(u(x,y)) ~ -sin(pi*x)*sin(pi*y)

# Boundary conditions
bcs = [u(0,y) ~ 0.0f0, u(1,y) ~ -sin(pi*1)*sin(pi*y),
      u(x,0) ~ 0.0f0, u(x,1) ~ -sin(pi*x)*sin(pi*1)]

# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```
Okay, now I’m just ranting.

But the point is, equation solving is a huge topic. As big as probabilistic programming.

So how do you integrate that with PPLs?
Mixing equation discovery into epidemic modeling workflows will revolutionize the field.

1. What could “extensive” differential equation support even mean?
2. Does Turing.jl have it? And what does that mean about its developer community?
Stan(dard) PPL DSLs: The Top-Down Approach

- **rk45**: a fourth and fifth order Runge-Kutta method for non-stiff systems (Dormand and Prince 1980; Ahnert and Mulansky 2011). **rk45** is the most generic solver and should be tried first.

- **bdf**: a variable-step, variable-order, backward-differentiation formula implementation for stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). **bdf** is often useful for ODEs modeling chemical reactions.

- **adams**: a variable-step, variable-order, Adams-Moulton formula implementation for non-stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). The method has order up to 12, hence is commonly used when high-accuracy is desired for a very smooth solution, such as in modeling celestial mechanics and orbital dynamics (Montenbruck and Gill 2000).

- **ckrk**: a fourth and fifth order explicit Runge-Kutta method for non-stiff and semi-stiff systems (Cash and Karp 1990; Mazzia, Cash, and Soetaert 2012).
16.3. Index of DAEs

The index along a DAE solution $y(t)$ is the minimum number of differentiations of some of the components of the system required to solve for $y'$ uniquely in terms of $y$ and $t$, so that the DAE is converted into an ODE for $y$. Thus an ODE system is of index 0. The above chemical kinetics DAE is of index 1, as we can perform differentiation of the third equation followed by introducing the first two equations in order to obtain the ODE for $y_3$.

Most DAE solvers, including the one in Stan, support only index-1 DAEs. For a high index DAE problem the user must first convert it to a lower index system. This often can be done by carrying out differentiations analytically (Ascher and Petzold 1998).

No DAE automation

No SDEs, DDEs, PDEs, ...
Symplectic, Munthe-Kaas, ...
No KLU, iLU preconditioning, algebraic multigrid...
Okay, I can poke fun at Stan.

But the “why” is more important.
Good PPL methods (Hamiltonian Monte Carlo, ADVI, etc.) Requires Good Derivatives of Every Operation
PPLs Need Derivatives

Good PPL methods (Hamiltonian Monte Carlo, ADVI, etc.) Requires Good Derivatives of Every Operation

You cannot just stick an ODE solver into a PPL and expect it to work!
Julia’s Pervasive Differentiable Programming

Julia has a pervasive language-wide system for differentiable programming

No DSL required: directly support Julia code in any code that requires differentiation!
Just do it!

```julia
@model function fitlv(data, prob)
    # Prior distributions.
    σ ~ InverseGamma(2, 3)
    α ~ truncated(Normal(1.5, 0.5), 0.5, 2.5)
    β ~ truncated(Normal(1.2, 0.5), 0, 2)
    γ ~ truncated(Normal(3.0, 0.5), 1, 4)
    δ ~ truncated(Normal(1.0, 0.5), 0, 2)

    # Simulate Lotka-Volterra model.
    p = [α, β, γ, δ]
    predicted = solve(prob, Tsit5(); p=p, saveat=0.1)

    # Observations.
    for i in 1:length(predicted)
        data[:, i] ~ MVNormal(predicted[i], σ^2 * I)
    end

    return nothing
end

model = fitlv(odedata, prob)

# Sample 3 independent chains with forward-mode automatic differentiation (the default)
chain = sample(model, NUTS(0.65), MCMCSerial(), 1000, 3; progress=false)
```

Turing.jl + DifferentialEquations.jl: Just use the ODE solver inside of Turing.
Improving Coverage of Automatic Differentiation over Solvers

**LinearSolve.jl**: Unified Linear Solver Interface

\[ A(p)x = b \]

**NonlinearSolve.jl**: Unified Nonlinear Solver Interface

\[ f(u, p) = 0 \]

**DifferentialEquations.jl**: Unified Interface for all Differential Equations

\[ u' = f(u, p, t) \]

\[ du = f(u, p, t)dt + g(u, p, t)dW_t \]

**Optimization.jl**: Unified Optimization Interface

\[
\begin{align*}
\text{minimize } & f(u, p) \\
\text{subject to } & g(u, p) \leq 0, h(u, p) = 0
\end{align*}
\]

**Integrals.jl**: Unified Quadrature Interface

\[
\int_{lb}^{ub} f(t, p)dt
\]

**Unified Partial Differential Equation Interface**

\[ u_t = u_{xx} + f(u) \]

\[ u_{tt} = u_{xx} + f(u) \]

---

*The SciML Common Interface for Julia Equation Solvers*

[https://scimlbase.sciml.ai/dev/](https://scimlbase.sciml.ai/dev/)
Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

### Probabilistic Model Discovery with Turing.jl

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>Number of Active terms</th>
<th>Dominant terms</th>
<th>Error</th>
<th>Mean AIC score</th>
<th>% sampled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>9</td>
<td>$u_1^2, u_2^2, u_1 u_2$</td>
<td>0.765</td>
<td>40.4</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_1^2 u_2, u_1^2 u_2, u_2^2 u_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_1 u_2$, const</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>9</td>
<td>$u_1^2, u_2^2, u_1 u_2$</td>
<td>0.764</td>
<td>35</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_1^2 u_2, u_1^2 u_2, u_2^2 u_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_1 u_2$, const</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>$u_1^2, u_2^2, u_1 u_2$</td>
<td>0.764</td>
<td>21.6</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$u_1 u_2, u_1 u_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$u_1^2, u_1 u_2$</td>
<td>0.634</td>
<td>7.2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>$u_1 u_2$</td>
<td>0.7</td>
<td>4.1</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>$u_1^2, u_2^2$</td>
<td>2.49</td>
<td>-1</td>
<td>100</td>
</tr>
</tbody>
</table>

```julia
function lotka_volterra!(du, u, p, t)
    x, y = u
    α, β, δ, γ = p
    du[1] = dx = α*x - α*x*y - β*y
    du[2] = dy = -δ*y
end
```

Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

<table>
<thead>
<tr>
<th>$\lambda_{cr}$</th>
<th>Number of Active terms</th>
<th>Dominant terms</th>
<th>% of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>2</td>
<td>$\rho, \rho^2$</td>
<td>73</td>
</tr>
<tr>
<td>0.5</td>
<td>3</td>
<td>$\rho, \rho^2, \rho^3$</td>
<td>27</td>
</tr>
</tbody>
</table>
The only mention of differential equations is in a tutorial! No API docs!
Upside: Lots of Features!

Also includes stochastic differential equations, delay differential equations, etc. all just from slapping DifferentialEquations.jl inside!

```julia
@model function fitlv_dde(data, prob)
    # Prior distributions.
    σ ~ InverseGamma(2, 3)
    α ~ Truncated(Normal(1.5, 0.5), 0.5, 2.5)
    β ~ Truncated(Normal(1.2, 0.5), 0, 2)
    γ ~ Truncated(Normal(3.0, 0.5), 1, 4)
    δ ~ Truncated(Normal(1.0, 0.5), 0, 2)

    # Simulate Lotka-Volterra model.
    p = [α, β, γ, δ]
    predicted = solve(prob, MethodOfSteps(Tsit5()); p=p, saveat=0.1)

    # Observations.
    for i in 1:length(predicted)
        data[:, i] ~ MvNormal(predicted[i], σ^2 * I)
    end
end

model_dde = fitlv_dde(ddedata, prob_dde)

# Sample 3 independent chains.
chain_dde = sample(model_dde, NUTS(0.65), MCMCSerial(), 300, 3; progress=false)
```
How the adjoint is calculated also matters!

Gradient calculations on a stiff PDE, varying dt


For more details on the performance of the adjoint methods, see Accurate and Efficient Physics-Informed Learning Through Differentiable Simulation
Enzyme is fast!!!

Static analysis & optimization => very, very fast scalar AD

```python
def taylor_jax(x, N):
    sum = 0 * x
    for i in range(1, N):
        sum += x**i / i
    return sum

def taylor_lax(x, N):
    return lax.fori_loop(1, N, lambda i, cur:
        cur + x**i / i, 0)
```

```plaintext
@btime Enzyme.autodiff(Forward, taylor, Duplicated(0.5, 1.0), 10^6)
# 30 ms (0 bytes)

@btime Enzyme.autodiff(Reverse, taylor, Active(0.5), 10^6).
# 30 ms (0 bytes)

@btime ForwardDiff.derivative(x -> taylor(x, 10^6), 0.5)
# 60 ms (0 bytes)

@btime Zygote.gradient(taylor, 0.5, 10^6)
# 993 ms (663.56 MiB)

@btime Diffractor.gradient(taylor, 0.5, 10^6)
# 96665 ms (96.37 GiB)

@pytime jax.grad(taylor_jax)(0.5, 10^5)
# >183993 ms

@pytime jax.grad(taylor_lax)(0.5, 10^6)
# 95 ms

Is Jax close enough?
Enzyme is fast!!!

- Static analysis & optimization => very, very fast scalar AD

```python
function taylor(x, N)
    sum = 0 * x
    for i = 1:N
        sum += x^i / i
    end
    return sum
end
```

```python
def taylor_jax(x, N):
    sum = 0 * x
    for i in range(1, N):
        sum += x**i / i
    return sum
```

```python
@btime Enzyme.autodiff(Forward, taylor, Duplicated(0.5, 1.0, 10^6))
# 30 ms (0 bytes)
```

```python
@btime Enzyme.autodiff(Reverse, taylor, Active(0.5), 10^6).
# 30 ms (0 bytes)
```

```python
@btime ForwardDiff.derivative(x -> taylor(x, 10^6), 0.5)
# 60 ms (0 bytes)
```

```python
@btime Zygote.gradient(taylor, 0.5, 10^6)
# 993 ms (663.56 MiB)
```

```python
@btime DiffTractor.gradient(taylor, 0.5, 10^6)
# 96665 ms (96.37 GiB)
```

```python
@btime jax.grad(taylor_jax)(0.5, 10^5)
# >183993 ms
```

```python
@btime jax.grad(taylor_lax)(0.5, 10^6)
# 95 ms
```

<table>
<thead>
<tr>
<th>construct</th>
<th>jit</th>
<th>grad</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>for</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>while</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lax.cond</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lax.while_loop</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lax.fori_loop</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lax.scan</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

If forward mode only.
Differentiating after optimization can create asymptotically faster gradients!
Downside: Documentation

If the composition of two packages automatically constructs the functionality, who documents it?
Pervasive Differentiable Programming greatly enlarges the developer pool and accelerates development.

But relying on package composability creates a documentation and “ownership” problem.

We are looking for nice solutions to the latter issues with our automatically constructed feature sets.
SciML Open Source Software Organization
sciml.ai

- DifferentialEquations.jl: 2x-10x Sundials, Hairer, ...
- DiffEqFlux.jl: adjoints outperforming Sundials and PETSc-TS
- ModelingToolkit.jl: 15,000x Simulink
- Catalyst.jl: >100x SimBiology, gillespy, Copasi
- DataDrivenDiffEq.jl: >10x pySindy
- NeuralPDE.jl: ~2x DeepXDE* (more optimizations to be done)
- NeuralOperators.jl: ~3x original papers (more optimizations required)
- ReservoirComputing.jl: 2x-10x pytorch-esn, ReservoirPy, PyRCN
- SimpleChains.jl: 5x PyTorch GPU with CPU, 10x Jax (small only!)
- DiffEqGPU.jl: Some wild GPU ODE solve speedups coming soon

And 100 more libraries to mention...

If you work in SciML and think optimized and maintained implementations of your method would be valuable, please let us know and we can add it to the queue.

Democratizing SciML via pedantic code optimization
Because we believe full-scale open benchmarks matter
There are many different ways, all with engineering trade-offs

<table>
<thead>
<tr>
<th>Method</th>
<th>Stability</th>
<th>Stiff Performance Scaling</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BacksolveAdjoint</td>
<td>Poor</td>
<td>$O((s + p)^3)$</td>
<td>Low. $O(1)$</td>
</tr>
<tr>
<td>InterpolatingAdjoint</td>
<td>Good</td>
<td>$O((s + p)^3)$</td>
<td>High. Requires full continuous solution of forward</td>
</tr>
<tr>
<td>QuadratureAdjoint</td>
<td>Good</td>
<td>$O(s^3 + p)$</td>
<td>Higher. Requires full continuous solution of forward and Lagrange multiplier</td>
</tr>
<tr>
<td>BacksolveAdjoint (Checkpointed)</td>
<td>Okay</td>
<td>$O((s + p)^3) + C$</td>
<td>Medium. $O(c)$ where $c$ is the number of checkpoints</td>
</tr>
<tr>
<td>InterpolatingAdjoint (Checkpointed)</td>
<td>Good</td>
<td>$O((s + p)^3) + C$</td>
<td>Medium. $O(c)$ where $c$ is the number of checkpoints</td>
</tr>
<tr>
<td>ReverseDiffAdjoint</td>
<td>Best</td>
<td>$O(s^3 + p) + C$</td>
<td>Highest. Requires full forward and reverse AD of solve</td>
</tr>
<tr>
<td>TrackerAdjoint</td>
<td>Best</td>
<td>$O(s^3 + p) + C$</td>
<td>Highest. Requires full forward and reverse AD of solve</td>
</tr>
<tr>
<td>ForwardLSS/AdjointLSS/NILSS</td>
<td>Chaos</td>
<td>Not even comparable: expensive.</td>
<td>Super duper high OMG.</td>
</tr>
</tbody>
</table>
We wish to solve for some cost function $G(u, p)$ evaluated throughout the differential equation, i.e.:

\[ G(u, p) = G(u(p)) = \int_{t_0}^{T} g(u(t, p)) dt \]

To derive this adjoint, introduce the Lagrange multiplier $\lambda$ to form:

\[ I(p) = G(p) - \int_{t_0}^{T} \lambda^* (u' - f(u, p, t)) dt \]

Since $u' = f(u, p, t)$, this is the mathematician's trick of adding zero, so then we have that

\[ s = \frac{du}{dp}, \quad \frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^{T} (g_p + g_u s) dt - \int_{t_0}^{T} \lambda^* (s' - f_u s - f_p) dt \]
Differentiating Ordinary Differential Equations: Integration By Parts

for $s$ being the sensitivity, $s = \frac{du}{dp}$. After applying integration by parts to $\lambda^* s'$, we get that:

$$\int_{t_0}^{T} \lambda^* (s' - f_u s - f_p) \, dt = \int_{t_0}^{T} \lambda^* s' \, dt - \int_{t_0}^{T} \lambda^* (f_u s - f_p) \, dt$$

$$= |\lambda^* (t) s(t)|_{t_0}^{T} - \int_{t_0}^{T} \lambda^{*'} s \, dt - \int_{t_0}^{T} \lambda^* (f_u s - f_p) \, dt$$

To see where we ended up, let's re-arrange the full expression now:

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p + g_u s) \, dt + |\lambda^* (t) s(t)|_{t_0}^{T} - \int_{t_0}^{T} \lambda^{*'} s \, dt - \int_{t_0}^{T} \lambda^* (f_u s - f_p) \, dt$$

$$= \int_{t_0}^{T} (g_p + \lambda^* f_p) \, dt + |\lambda^* (t) s(t)|_{t_0}^{T} - \int_{t_0}^{T} (\lambda^{*'} + \lambda^* f_u - g_u) s \, dt$$
Differentiating Ordinary Differential Equations: The Final Form

\[
\frac{dG}{dp} = \int_{t_0}^{T} (g_p + \lambda^* f_p) dt + |\lambda^*(t)s(t)|_{t_0}^{T} - \int_{t_0}^{T} (\lambda^* t + \lambda^* f_u - g_u) s dt
\]

That was just a re-arrangement. Now, let's require that

\[
\lambda' = -\frac{df^*}{du} \lambda - \left( \frac{dg}{du} \right)^*
\]

\[
\lambda(T) = 0
\]

This means that the boundary term of the integration by parts is zero, and also one of those integral terms are perfectly zero. Thus, if \( \lambda \) satisfies that equation, then we get:

\[
\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p) dt
\]
Summary:

1. Solve \( u' = f(u, p, t) \)

2. Solve \( \lambda' = -\frac{df^*}{du} \lambda - \left( \frac{dg}{du} \right)^* \)

\( \lambda(T) = 0 \)

3. Solve \( \frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p) \, dt \)
2. Solve

\[ \lambda(t) = -\frac{df}{du(t)} \lambda(t) \left( \frac{dg}{du(t)} \right)^* \]

\[ \lambda(T) = 0 \]

How do you get \( u(t) \) while solving backwards?

3 options!

1. \( u' = f(z, t) \) forwards, then
   \( u' = -f(z, -t) \) backwards!

2. Store \( u(t) \) while solving forwards (dense output)

3. Checkpointing
How the gradient (adjoint) is calculated also matters!

This term is traditionally computed via differentiation and then multiplied to lambda.
Reverse-mode embedded implementation: push-forward $f(u)$ pullback lambda.
Computational cost $O(n) \rightarrow O(1)$ $f$ evaluations and automatically uses optimized backpropagation.

$$M^* \lambda' = -\frac{df}{du}^* \lambda - \left( \frac{dg}{du} \right)^*$$

$$\lambda(T) = 0,$$

**Adjoint Differential Equation**

**Six choices for this computation:**
- Numerical
- Forward-mode
- Reverse-mode traced compiled graph (ReverseDiffVJP(true))
  - Fast method for scalarized nonlinear equations
  - Requires CPU and no branching (generally used in SciML)
- Reverse-mode static
  - Fastest method when applicable
- Reverse-mode traced
  - Fast but not GPU compatible
- Reverse-mode vector source-to-source
  - Best for embedded neural networks
Differentiating Ordinary Differential Equations: Step 3 Details

3. Solve

$$\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^{T} (g_p + \lambda^*(t)f_p) \, dt$$

How do you calculate the integral?

1. Store $\lambda(t)$ while solving backwards (dense output)

2. $\mu' = -\lambda^* f_p + g_p$ where $\mu(T') = 0$

What’s the trade-off between these ideas?
Some methods are “mathematically correct”, but “numerically incorrect”

SciML is a software problem.
The adjoint equation is an ODE!

\[
\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z}
\]

Timeseries is not stored, therefore $O(1)$ in memory!

How do you get $z(t)$? One suggestion: Reverse the ODE
"Adjoint by reversing" also is unconditionally unstable on some problems!

Advection Equation:
\[ \frac{\partial u}{\partial t} + a \frac{\partial u}{\partial x} = 0 \]

Approximating the derivative in \( x \) has two choices: forwards or backwards

If you discretize in the wrong direction you get unconditional instability

You need to understand the engineering principles and the numerical simulation properties of domain to make ML stable on it.
Differentiation of Chaotic Systems: Shadow Adoints

chaotic systems: trajectories diverge to o(1) error ... but shadowing lemma guarantees that the solution lies on the attractor

\[ \frac{d}{d\rho} \langle z \rangle_\infty \neq \lim_{T \to \infty} \frac{\partial}{\partial \rho} \langle z \rangle_T \]

- AD and finite differencing fails!

\[ \left. \frac{d\langle z \rangle_\infty}{d\rho} \right|_{\rho=28} \approx -49899 \text{ (ForwardDiff)} \]

\[ \left. \frac{d\langle z \rangle_\infty}{d\rho} \right|_{\rho=28} \approx 472 \text{ (Calculus)} \]

- Shadowing methods in DiffEqSensitivity.jl

\[ \left. \frac{d\langle z \rangle_\infty}{d\rho} \right|_{\rho=28} \approx 1.028 \text{ (LSS/AdjointLSS)} \]

\[ \left. \frac{d\langle z \rangle_\infty}{d\rho} \right|_{\rho=28} \approx 0.997 \text{ (NILSS)} \]

https://frankschae.github.io/post/shadowing/
Problems With Naïve Adjoint Approaches On Stiff Equations

Error grows exponentially...

$u'(t) = \lambda u(t)$, plot the error in the reverse solve:

How do you get $u(t)$ while solving backwards?

3 options!

1. $u' = f(z, t)$ forwards, then $u' = -f(z, -t)$ backwards! \textcolor{red}{Unstable}

2. Store $u(t)$ while solving forwards (dense output) \textcolor{red}{High memory}

3. Checkpointing \textcolor{red}{More Compute}

Each choices has an engineering trade-off!

Problems With Naïve Adjoint Approaches On Stiff Equations

Error grows exponentially...

\[ u'(t) = \lambda u(t), \] plot the error in the reverse solve:

Compute cost is cubic with parameter size when stiff

Size of reverse ODE system is:

\[ 2\text{states} + \text{parameters} \]

Linear solves inside of stiff ODE solvers, ~cubic

Thus, adjoint cost:

\[ O((\text{states} + \text{parameters})^3) \]

Problems With Naïve Adjoint Approaches On Stiff Equations

\[
\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^{T} (g_p + \lambda^* f_p) \, dt
\]

How do you calculate the integral?

1. Store \(\lambda(t)\) while solving backwards (dense output)

2. \(\mu' = -\lambda^* f_p + g_p\) where \(\mu(T) = 0\)  
   Size = Number of Parameters

3. Use an IMEX integrator and solve \(\mu' = -\lambda^* f_p + g_p\) explicitly

4. Our paper describes a 4th way!


Compute cost is cubic with parameter size when stiff

Size of reverse ODE system is:

\(2\text{states} + \text{parameters}\)

Linear solves inside of stiff ODE solvers, \(~\text{cubic}\)

Thus, adjoint cost:

\(O((\text{states} + \text{parameters})^3)\)

Thus, adjoint cost without extra memory:

\(O(\text{states}^3 + \text{parameters})\)
The math has >20 ways to implement.

Every choice makes engineering trade-offs.

SciML is a software problem.
### DiffEqSensitivity.jl: Every adjoint is optimized for a different case

<table>
<thead>
<tr>
<th>Method</th>
<th>Stability</th>
<th>Stiff Performance Scaling</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BacksolveAdjoint</td>
<td>Poor</td>
<td>$O((s + p)^3)$</td>
<td>Low. $O(1)$</td>
</tr>
<tr>
<td>InterpolatingAdjoint</td>
<td>Good</td>
<td>$O((s + p)^3)$</td>
<td>High. Requires full continuous solution of forward</td>
</tr>
<tr>
<td>QuadratureAdjoint</td>
<td>Good</td>
<td>$O(s^3 + p)$</td>
<td>Higher. Requires full continuous solution of forward and Lagrange multiplier</td>
</tr>
<tr>
<td>BacksolveAdjoint (Checkpointed)</td>
<td>Okay</td>
<td>$O((s + p)^3) + C$</td>
<td>Medium. $O(c)$ where $c$ is the number of checkpoints</td>
</tr>
<tr>
<td>InterpolatingAdjoint (Checkpointed)</td>
<td>Good</td>
<td>$O((s + p)^3) + C$</td>
<td>Medium. $O(c)$ where $c$ is the number of checkpoints</td>
</tr>
<tr>
<td>ReverseDiffAdjoint</td>
<td>Best</td>
<td>$O(s^3 + p) + C$</td>
<td>Highest. Requires full forward and reverse AD of solve</td>
</tr>
<tr>
<td>TrackerAdjoint</td>
<td>Best</td>
<td>$O(s^3 + p) + C$</td>
<td>Highest. Requires full forward and reverse AD of solve</td>
</tr>
<tr>
<td>ForwardLSS/AdjointLSS/NILSS</td>
<td>Chaos</td>
<td>Not even comparable: expensive.</td>
<td>Super duper high OMG.</td>
</tr>
</tbody>
</table>
How the adjoint is calculated also matters!

Gradient calculations on a stiff PDE, varying dt


Methods with Reverse-mode vjp seeding + new adjoints give 3 orders of magnitude improvement!
The SciML ecosystem is the only one with fully-featured Universal Differential Equations

<table>
<thead>
<tr>
<th>Feature</th>
<th>SciML (Julia)</th>
<th>Sundials (C++)</th>
<th>PETSc TS (C++)</th>
<th>torchdiffeq</th>
<th>Jax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stiff ODEs and DAEs</td>
<td>Hundreds of methods tested and tuned on hundreds of problems</td>
<td>Yes (CVODE_BDF and IDA)</td>
<td>Yes (Rosenbrock-W methods, BDFs, etc.)</td>
<td>None</td>
<td>None (one in progress, ~200 times slower than SciPy according to the author!)</td>
</tr>
<tr>
<td>Adjoint Methods</td>
<td>11 choices tuned for different scenarios, including stabilized checkpointing, differentiate the solver, reversing adjoint</td>
<td>Stabilized checkpointing, no AD integration, no chaos compatibility</td>
<td>Discrete sensitivity analysis, no AD integration, no chaos compatibility</td>
<td>Requires reversing the ODE or differentiate the solver (tracing)</td>
<td>Requires reversing the ODE</td>
</tr>
<tr>
<td>Parallelism</td>
<td>GPU, MPI, multithreading</td>
<td>GPU, MPI, multithreading</td>
<td>GPU, MPI, and multithreading</td>
<td>GPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Event handling</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>SDEs</td>
<td>Lots of methods, including stabilized, methods for stiff equations, high strong order, high weak order</td>
<td>None</td>
<td>None</td>
<td>torchsde, only diagonal noise (or order 0.5), requires reversing the SDE</td>
<td>None</td>
</tr>
<tr>
<td>Delays</td>
<td>All ODE methods</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
These ODEs are non-stiff ODEs from astrodynamics, chemical kinetics, numerical weather prediction, etc. and include scalarized operations.

<table>
<thead>
<tr>
<th>Number of ODEs</th>
<th>3</th>
<th>28</th>
<th>768</th>
<th>3,072</th>
<th>12,288</th>
<th>49,152</th>
<th>196,608</th>
<th>786,432</th>
</tr>
</thead>
<tbody>
<tr>
<td>DifferentialEquations.jl</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>DifferentialEquations.jl dopri5</td>
<td>1.0x</td>
<td>1.6x</td>
<td>2.8x</td>
<td>2.7x</td>
<td>3.0x</td>
<td>3.0x</td>
<td>3.9x</td>
<td>2.8x</td>
</tr>
<tr>
<td>torchdiffeq dopri5</td>
<td>4,900x</td>
<td>190x</td>
<td>840x</td>
<td>220x</td>
<td>82x</td>
<td>31x</td>
<td>24x</td>
<td>17x</td>
</tr>
</tbody>
</table>

**Spiral Neural ODE (from original Neural ODE paper)**

- DiffEqFlux defaults: 7.4 seconds
- DiffEqFlux optimized: 2.7 seconds
- torchdiffeq: 288.965871299999 seconds

**Relative time to solve**

**Geometric Brownian Motion of size 4**

The SDE is solved 100 times. The summary of the results is as follows:

- torchsde: 1.87 seconds
- DifferentialEquations.jl: 0.00115 seconds

*Note: performance is not necessarily indicative of large “pure” neural equations*