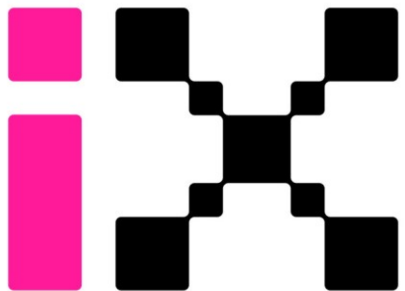


Differentiable simulators: a bridge between machine learning and scientific computing

Dr Aidan Crilly

Eric and Wendy Schmidt AI in Science postdoctoral fellow

Centre for Inertial Fusion Studies



Imperial-X

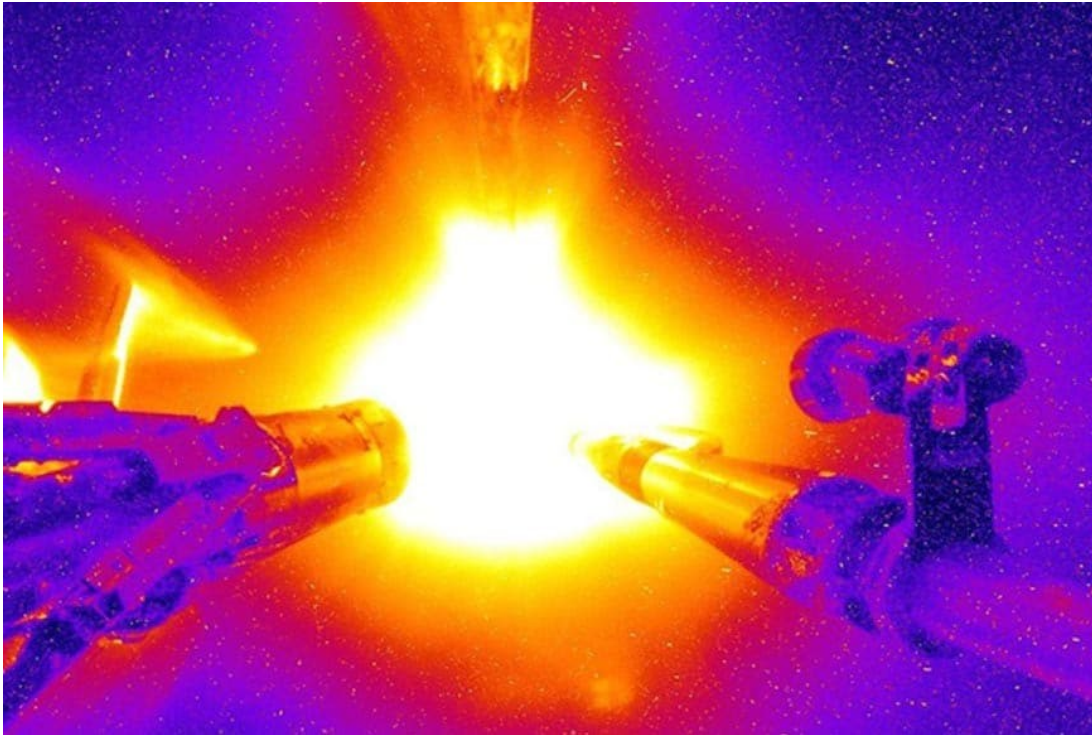
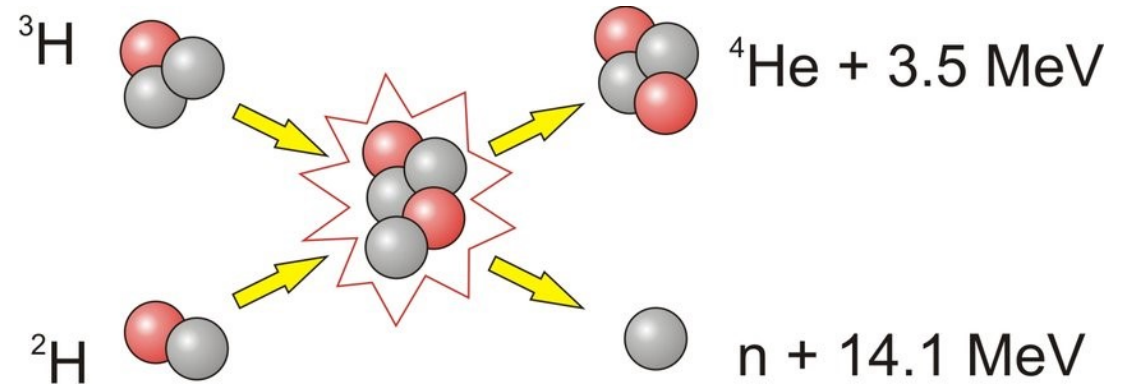


Outline

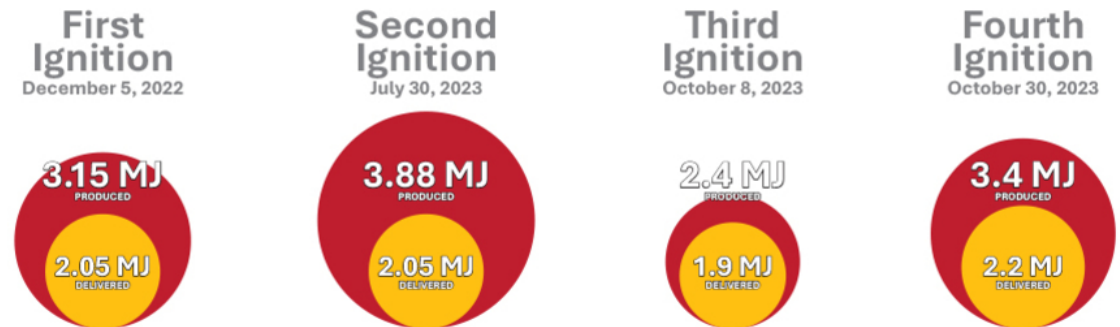
- Context
- Differential equations
 - Adjoint state problem
- Differentiable programming
 - For differential equations
- Neural differential equations
- Exercises and Assessment

My context

- Nuclear fusion – “inertial confinement”

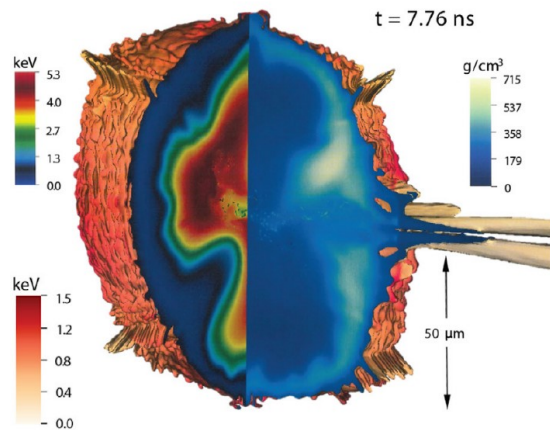
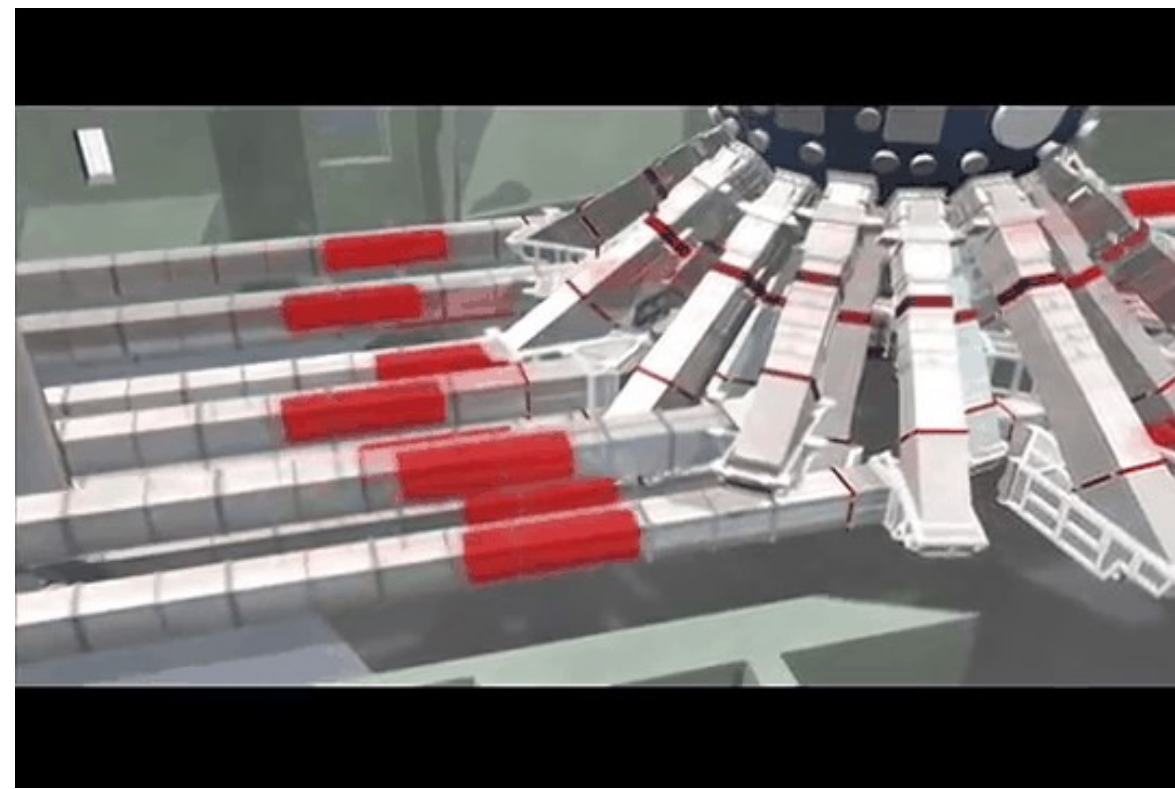


Charting the First Year of Ignition



My context

- Nuclear fusion – “inertial confinement”
- Predictive modelling of plasma behaviour and observable signals

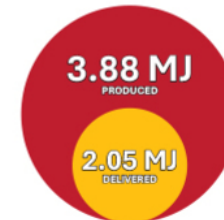


Charting the First Year of Ignition

First Ignition
December 5, 2022



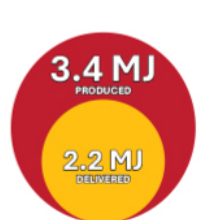
Second Ignition
July 30, 2023



Third Ignition
October 8, 2023



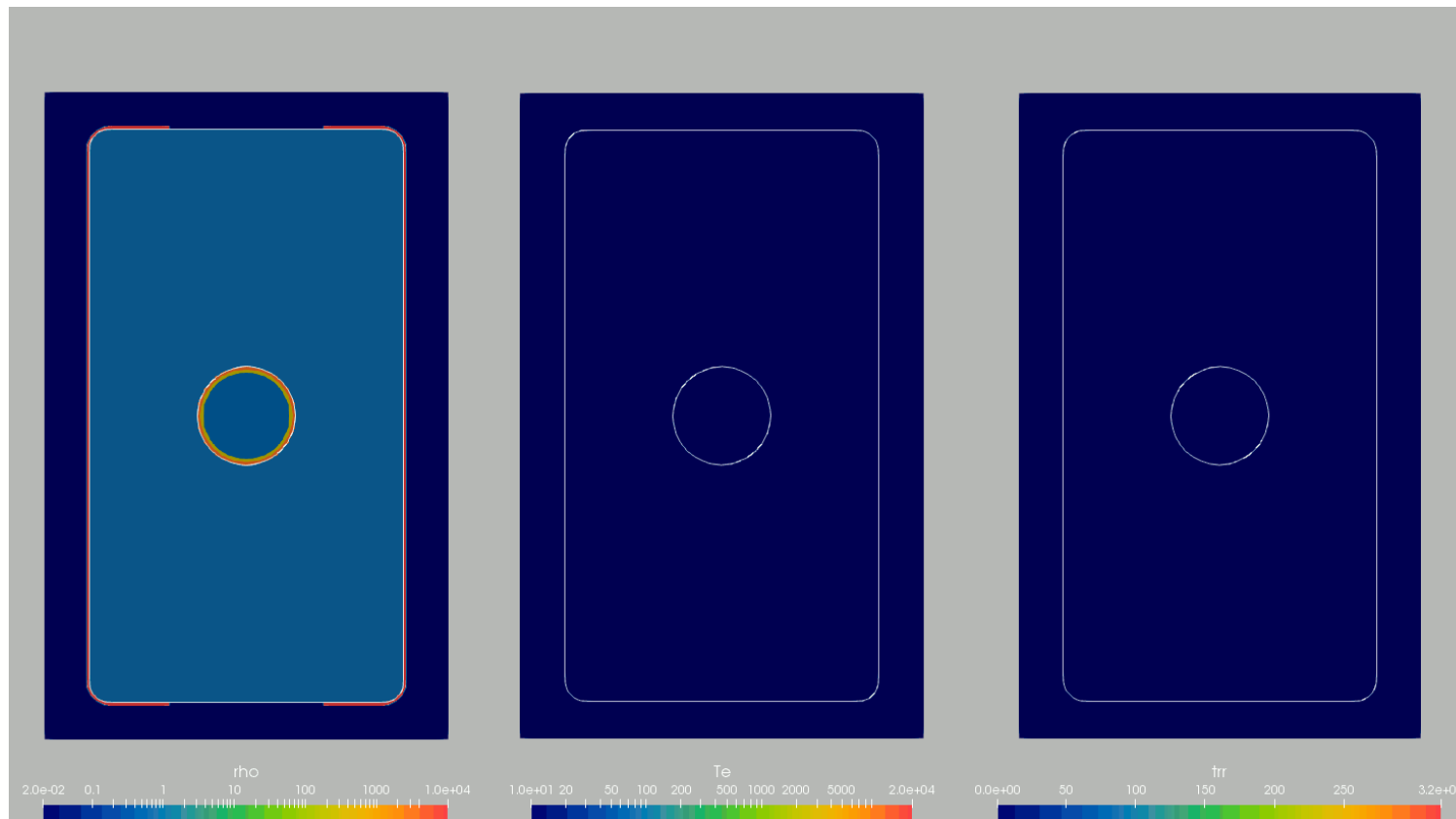
Fourth Ignition
October 30, 2023



My context



- Multi-physics code in 3D:
 - Magneto-hydrodynamics, radiation transport, thermal conduction, split electron-ion energy equations, laser ray trace, alpha particle transport, non-ideal equation of state, extended Ohm's law, material strength...



My context



- Multi-physics code in 3D:
 - Magneto-hydrodynamics, radiation transport, thermal conduction, split electron-ion energy equations, laser ray trace, alpha particle transport, non-ideal equation of state, extended Ohm's law, material strength...

$$\partial_t \rho + \nabla \cdot (\rho \vec{u}) = 0$$

$$\partial_t (\rho \vec{u}) + \nabla \cdot (\rho \vec{u} \vec{u}) = -\nabla P + J \times B$$

$$\partial_t (\rho e_{tot}) + \nabla \cdot (\rho e_{tot} \vec{u}) = -\nabla \cdot (P \vec{u}) + \text{Sources}$$

$$\frac{\partial \underline{B}}{\partial t} = \nabla \times \underline{v}_B \times \underline{B} - \nabla \times \eta_{\parallel} \nabla \times \underline{B} + \nabla \times \frac{\nabla P_e}{en_e}$$

$$\left[\frac{\partial}{\partial t} + \kappa_{\nu} c \right] E_{\nu} = 4\pi j_{\nu} - \nabla \cdot \vec{F}_{\nu} ,$$

$$\left[\frac{\partial}{\partial t} + \kappa_{\nu} c \right] \vec{F}_{\nu} = -c^2 \nabla \cdot \mathbf{P}_{\nu} ,$$

$$C_e \frac{\partial T_e}{\partial t} = \kappa_e \nabla^2 T_e + S_e + \omega_{ie} (T_i - T_e)$$

$$C_i \frac{\partial T_i}{\partial t} = \kappa_i \nabla^2 T_i + S_i + \omega_{ie} (T_e - T_i)$$

$$\frac{\partial v_{ray}}{\partial t} = \nabla \cdot \left(\frac{-c^2}{2} \frac{n_e}{n_c} \right)$$

$$\begin{aligned} & \left[\frac{1}{v} \frac{\partial}{\partial t} + \hat{\Omega} \cdot \vec{\nabla} + n(\vec{r}, t) \sigma(E) \right] \psi(\vec{r}, \hat{\Omega}, E, t) \\ &= S_{ex}(\vec{r}, \hat{\Omega}, E, t) \\ &+ \int_0^{\infty} dE' \int d\hat{\Omega}' n(\vec{r}, t) \sigma_s(\hat{\Omega}' \cdot \hat{\Omega}, E' \rightarrow E) \psi(\vec{r}, \hat{\Omega}', E', t), \end{aligned}$$

$$\begin{aligned} \frac{\partial f_{\alpha}(v_{\alpha}, t)}{\partial t} &= C(f_{\alpha}(v_{\alpha}, t)) + \frac{S_0 \delta(v_{\alpha} - v_b)}{4\pi v_{\alpha}^2} \\ &= \frac{1}{\tau_S v_{\alpha}^2} \frac{\partial}{\partial v_{\alpha}} \left\{ \frac{v_{\alpha}^3 T_e + v_c^3 T_i \Phi(x_i)}{m_{\alpha} v_{\alpha}} \frac{\partial f_{\alpha}}{\partial v_{\alpha}} \right. \\ &\quad \left. + [v_{\alpha}^3 + v_c^3 \Phi(x_i)] f_{\alpha} \right\} + \frac{S_0 \delta(v_{\alpha} - v_b)}{4\pi v_{\alpha}^2}, \quad \text{And the list goes on...} \end{aligned}$$

Nature and differential equations

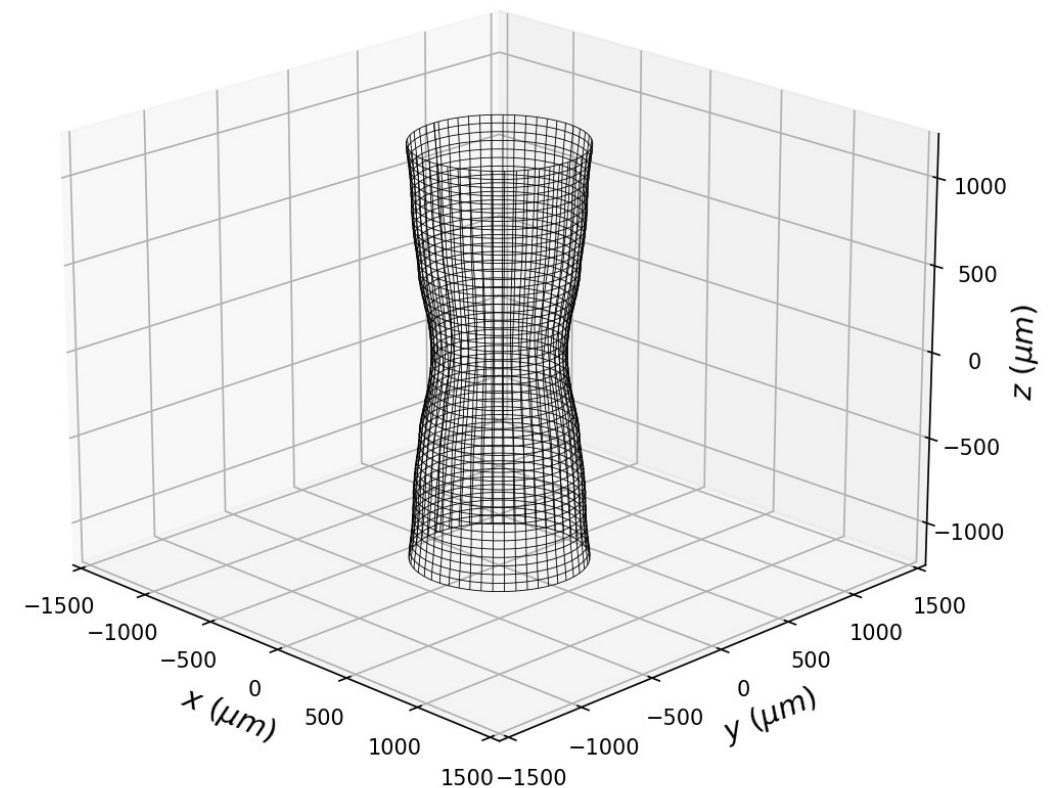
- Differential equations have “unreasonable effectiveness” in describing nature*
- Newton noted that the circular orbit of the moon and parabolic trajectory of a thrown rock were special cases of an ellipse
 - 2nd order derivative
- Can we use ML/AI to augment differential equation models?

* *c.f. Wigner, Comms. In Pure and Applied Mathematics (1960)*

Numerical modelling – ODEs

- Ordinary differential equations (ODEs) – dependent on only a single independent variable
 - For the most part, this variable is time
- Example: the path of a laser in a plasma (geometric optics)

$$\frac{d\underline{v}_{ray}}{dt} = \nabla \left(\frac{-c^2 n_e}{2 n_c} \right)$$



Numerical modelling – ODEs

- Numerical solutions to ODEs use finite steps (h or dt) to approximate derivatives

$$\frac{dy}{dt} = f(t, y)$$

Numerical modelling – ODEs

- We use finite differencing to approximate derivatives

Forward differencing:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y^{n+1} - y^n}{x^{n+1} - x^n}$$

Backward differencing:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y^n - y^{n-1}}{x^n - x^{n-1}}$$

First order in accuracy

Centred differencing:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y^{n+1} - y^{n-1}}{x^{n+1} - x^{n-1}} \longrightarrow$$

Second order in accuracy

Numerical modelling – ODEs

- Numerical solutions to ODEs use finite steps (h or dt) to approximate derivatives

$$\frac{dy}{dt} = f(t, y)$$

- Simplest = Forward Euler:

$$y_{n+1} = y_n + h f(t_n, y_n)$$

Numerical modelling – ODEs

- Numerical solutions to ODEs use finite steps (h or dt) to approximate derivatives

$$\frac{dy}{dt} = f(t, y)$$

- Simplest = Forward Euler:

$$y_{n+1} = y_n + h f(t_n, y_n)$$

- Very common = 4th order Runge-Kutta (RK4) with adaptive stepping

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

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

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

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

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

$$k_4 = f(t_n + h, y_n + h k_3).$$

Forward vs Inverse problems

- Solutions to differential equations are often concerned with the *forward problem*

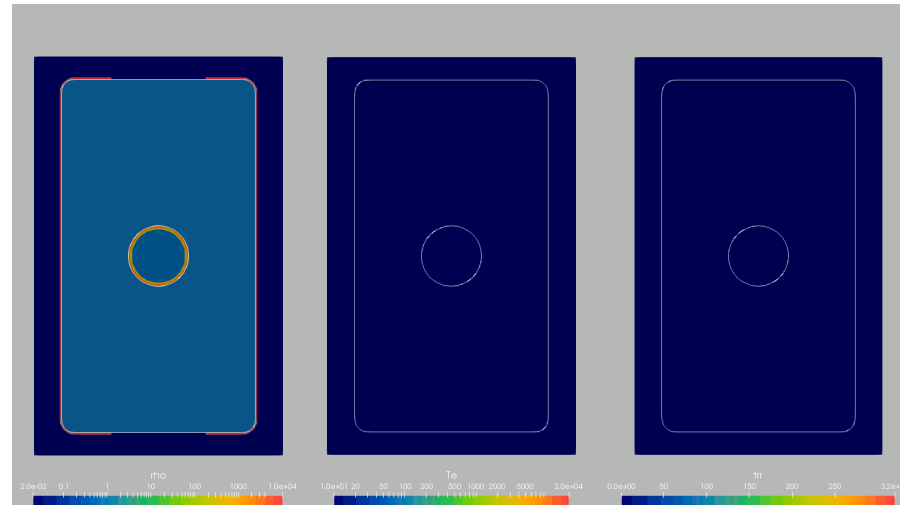
Differential equation: $h(x, p, t) = 0$
Initial condition: $g(x(0), p) = 0$

x = state variables
 p = parameters
 t = time

Initial conditions:
Densities, temperatures, etc.

Differential equations:
Hydrodynamics ++

Other parameters:
Laser power vs time



Forward vs Inverse problems

- Solutions to differential equations are often concerned with the *forward problem*

Differential equation: $h(x, p, t) = 0$

Initial condition: $g(x(0), p) = 0$

x = state variables

p = parameters

t = time

- However, if we want to minimize some other scalar function at the same time = *inverse problem*

Figure of merit: Minimise w. r. t. p : $f(x, p)$

Forward vs Inverse problems

- Solutions to differential equations are often concerned with the *forward problem*

Differential equation: $h(x, p, t) = 0$

Initial condition: $g(x(0), p) = 0$

x = state variables

p = parameters

t = time

- However, if we want to minimize some other scalar function at the same time = *inverse problem*

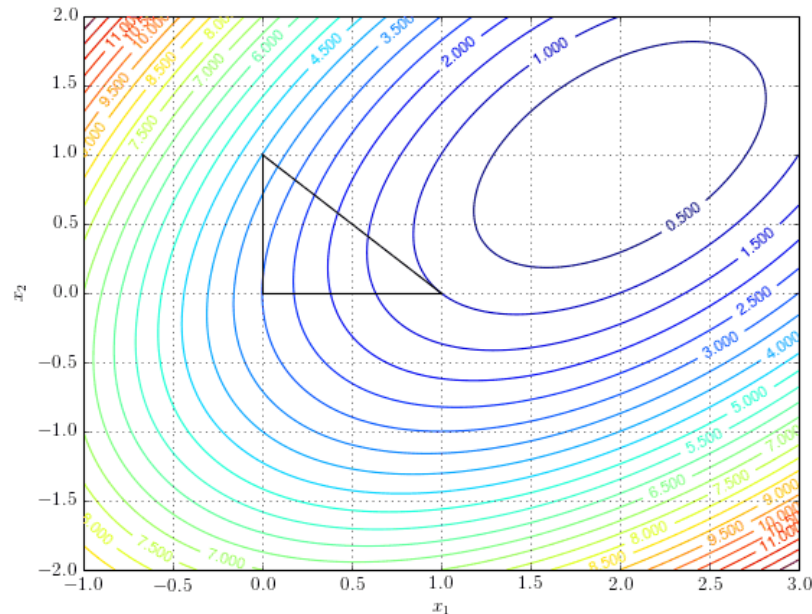
Figure of merit: **Minimise w. r. t. p : $f(x, p)$**

Optimisation

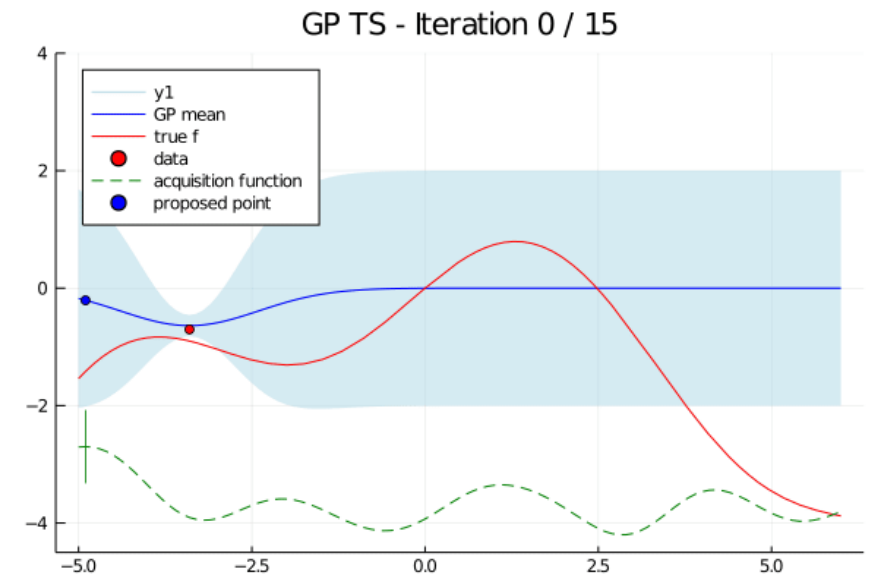
Optimisation

- Minimisation = optimisation = root-finding of gradient
- Gradient-based vs gradient-free optimisation
- Gradient-free:

Downhill-simplex

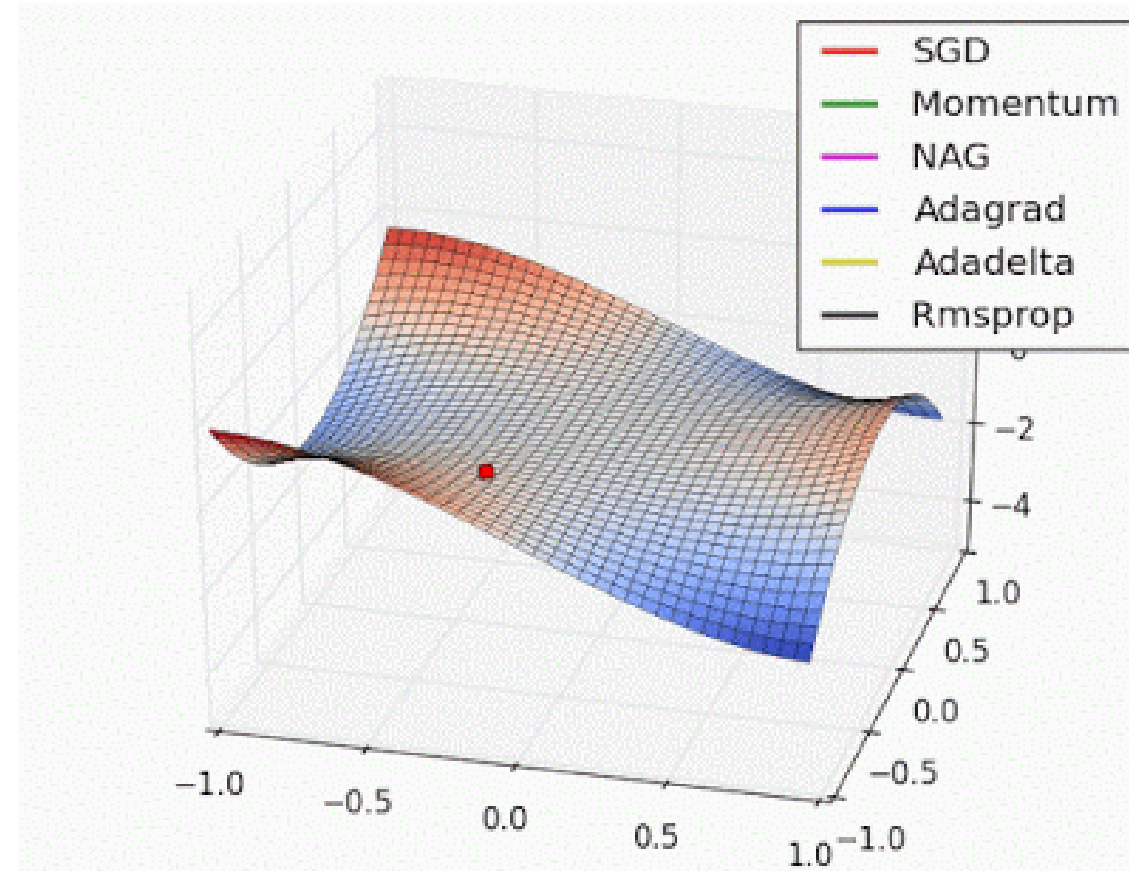


Bayesian optimisation



Optimisation

- Minimisation = optimisation = root-finding of gradient
- Gradient-based:
 - Simple idea, roll down-hill (to minimize)

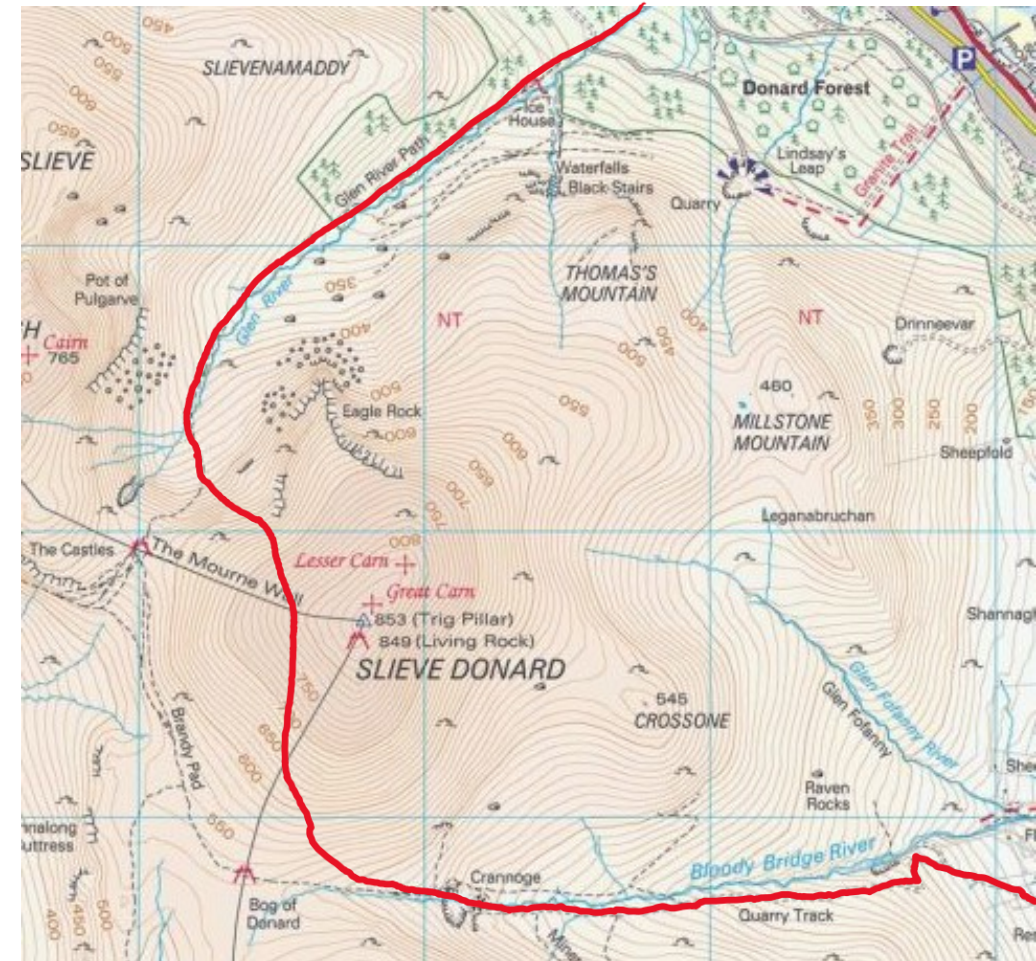


The adjoint state problem

- How do we solve the inverse problem in differential equations efficiently?
- We could run additional forward solutions to get finite difference gradients – very inefficient
- Instead of predicting how a *single* design change influences *every* aspect of the flow, the adjoint method predicts how *every* design change influences a *single* aspect of the flow.

Constrained Optimisation

- Inverse problem in simplified form:
Maximise w.r.t. p : $f(p)$
 $g(p) = 0$
- *Intuitive example*: Reach highest point (f) while sticking to path (g)
 - When path falls below you both in front and behind you
 - Or the path is tangent with contour
 - Or the path of steepest ascent (gradient) is 90 degree to path



In this example:

- $f(p)$ gives the height given your position, p , i.e. x and y coordinates.
- $g(p)$ parameterises the path you take, such that if you are on the path $g(p) = 0$

Constrained Optimisation – the maths

- Combine target and constraint using “Lagrange multipliers”



$$L(p, \lambda) = f(p) + \lambda g(p)$$

- Find optima of this function:

$$\partial_p L = 0 \rightarrow \partial_p f = -\lambda \partial_p g$$

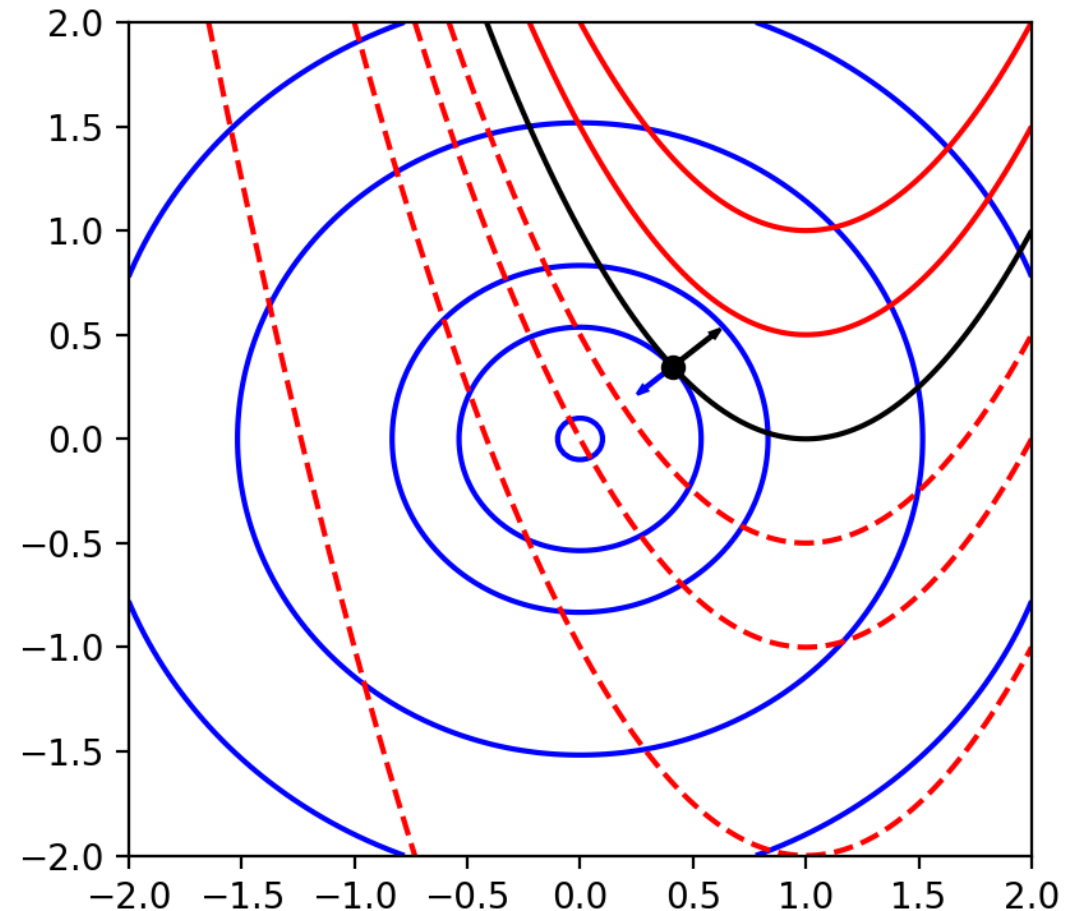
$$\partial_\lambda L = g(p) = 0$$

- In other words, at the optima the gradients are parallel

Let's consider $p = [x, y]$ and:

$$f(x, y) = e^{-x^2 - y^2}$$

$$g(x, y) = y - (x - 1)^2 = 0$$



The adjoint state problem

- Inverse problem in simplified form:

$$\begin{aligned} &\text{Minimise w.r.t. } p: f(x, p) \\ &g(x, p) = 0 \end{aligned}$$

- Constrained optimisation \rightarrow Lagrange multipliers:

$$L(x, p, \lambda) = f(x, p) + \lambda^T g(x, p)$$

- Gives *adjoint equation* and gradient w.r.t. parameters, p :

$$\frac{\partial f}{\partial x} + \lambda^T \frac{\partial g}{\partial x} = 0, \quad \frac{df}{dp} = \frac{dL}{dp} = \frac{\partial f}{\partial p} + \lambda^T \frac{\partial g}{\partial p}$$

The adjoint state problem – differentiable programming

- For differential equations, the *adjoint equation* is itself another differential equation
- Backpropagation of gradients through our differential equation would implicitly solve *adjoint equations*
 - *We will show this later...*
- How do we do get gradients of numerical solutions to differential equations?

Automatic differentiation introduction

- Numerical differentiation, finite difference methods:

$$\frac{dy}{dt} = \lim_{\varepsilon \rightarrow 0} \frac{y(t + \varepsilon) - y(t)}{\varepsilon} = f(t, y)$$

- Numerical error dependent on the step size used:

$$\frac{y(t + \varepsilon) - y(t)}{\varepsilon} = \dot{y}(t) + \frac{1}{2} \varepsilon \ddot{y}(t) + \dots = f(t, y) + \text{Error}(\varepsilon)$$

- Enter automatic differentiation...

Differentiable Programming?



Yann LeCun:

“Deep Learning est mort. Vive Differentiable Programming!

...

*An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them. **It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable.** Dynamic networks have become increasingly popular...”*

Automatic-differentiation/differentiable-programming frameworks

- Python: PyTorch – autograd
- Python: Tensorflow – GradientTape
- Python: JAX
- C++: autodiff
- Check out autodiff.org for other languages/libraries

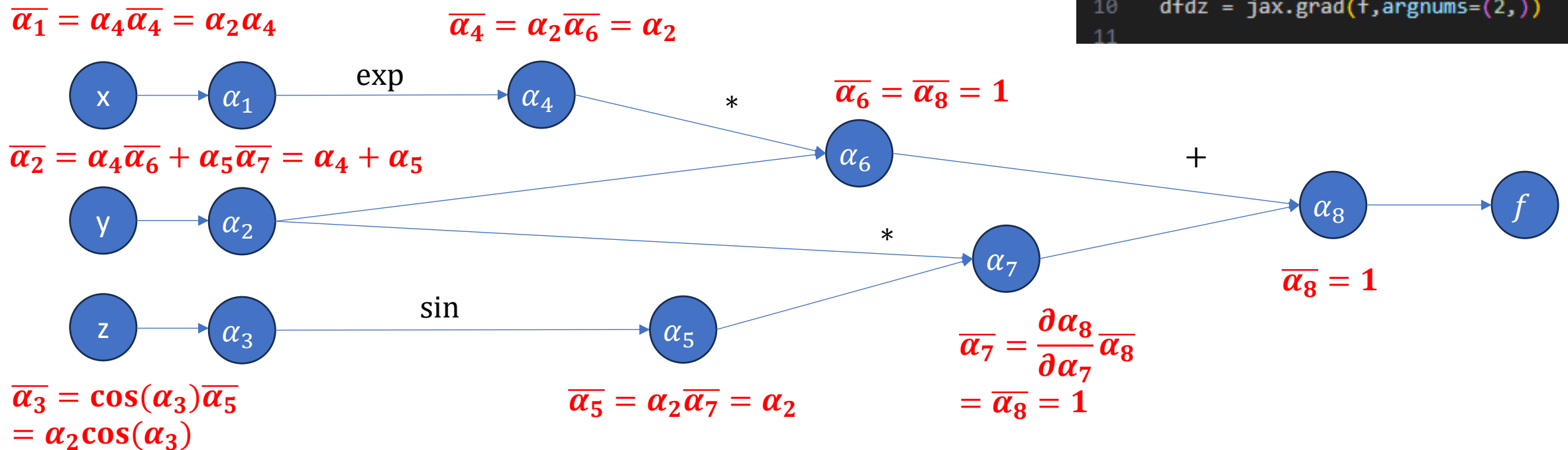
Automatic-differentiation/differentiable-programming frameworks

- Python: PyTorch – autograd
- Python: Tensorflow – GradientTape
- **Python: JAX**
- C++: autodiff
- Check out autodiff.org for other languages/libraries

PythonJAX simple example

- In example:
 - $f(x, y, z) = y \exp(x) - y \sin(z)$

```
1 import jax
2 import jax.numpy as jnp
3 import matplotlib.pyplot as plt
4
5 def f(x,y,z):
6     return y*jnp.exp(x)-y*jnp.sin(z)
7
8 dfdx = jax.grad(f,argnums=(0,))
9 dfdy = jax.grad(f,argnums=(1,))
10 dfdz = jax.grad(f,argnums=(2,))
11
```



A single 'back-propagation' gives *all* gradients!

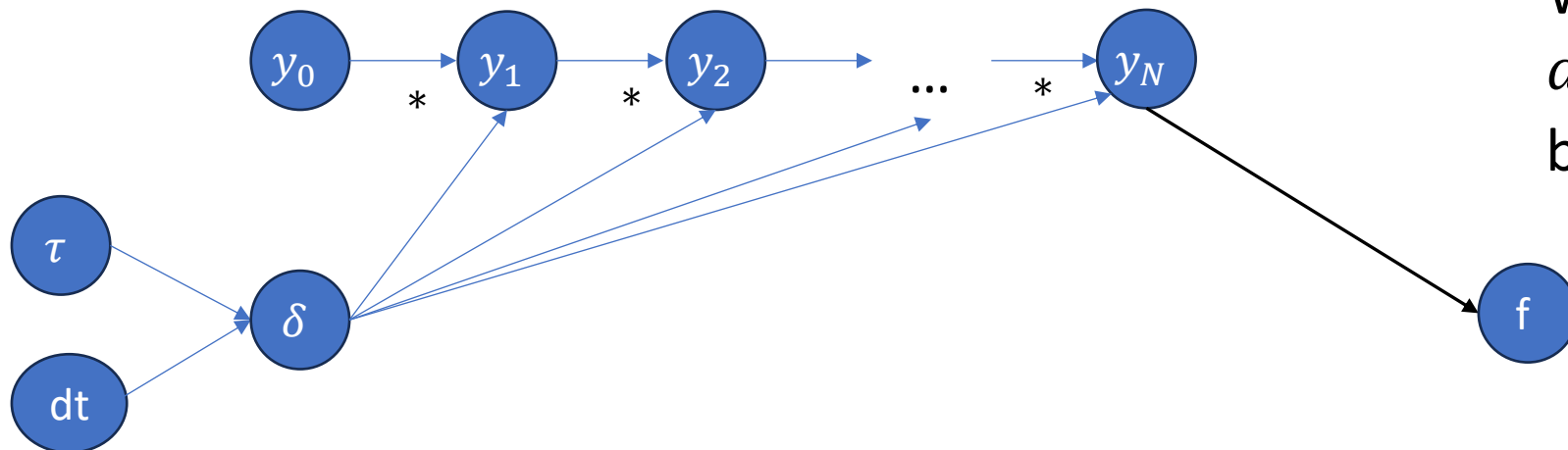
Back to differential equations...

What if our function is a solution to an ODE?

- A simple example ODE, solved using Euler's method

$$\frac{dy}{dt} = -\frac{1}{\tau}y, \quad y_{n+1} = \left(1 - \frac{dt}{\tau}\right)y_n = \delta y_n$$

- Define some loss, f , which uses values of y
- It's just a graph



We can compute $d_\tau f$,
 $d_{dt} f$ and $d_{y_0} f$ using
back-propagation

What if our function is a solution to an ODE?

- A simple example ODE, with parameters y_0 and τ

$$\frac{dy}{dt} = -\frac{1}{\tau}y, \quad f = L(y(T))$$

- What are the formal *adjoint equations*?
 - There will be differential equations for
 1. *Adjoint process (Lagrange multiplier)*
 2. *Parameter gradients (df/dp)*

What if our function is a solution to an ODE?

- A simple example ODE, with parameters y_0 and τ

$$\frac{dy}{dt} = -\frac{1}{\tau}y, \quad f = L(y(T))$$

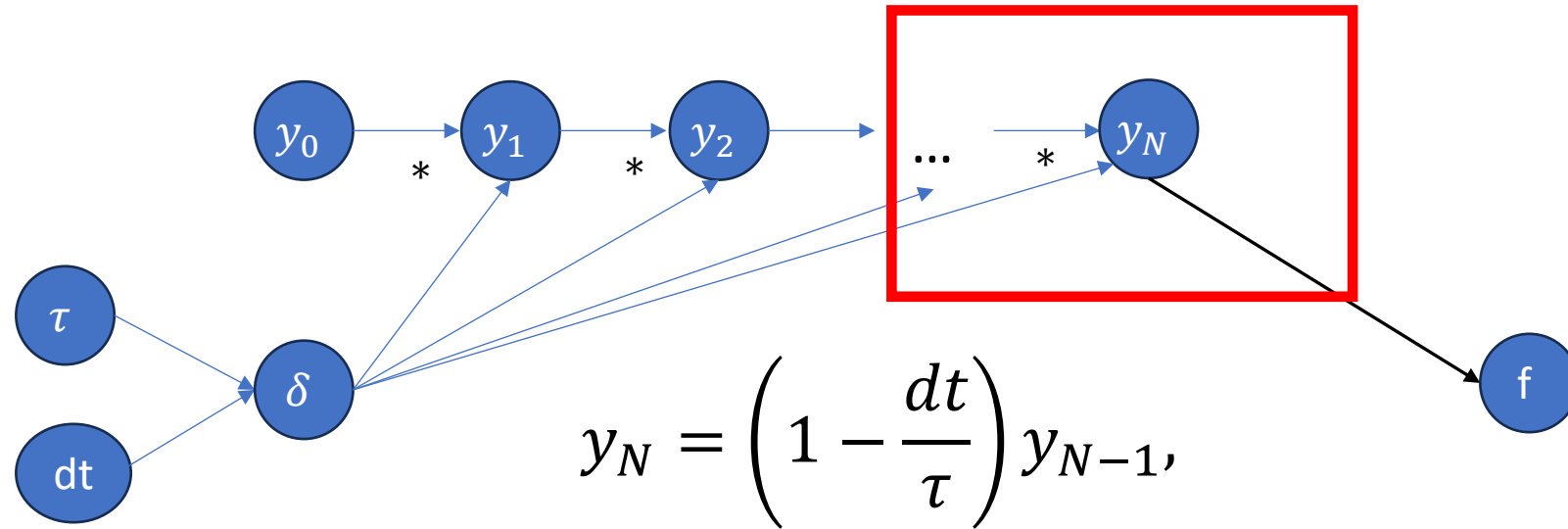
- What are the formal *adjoint equations*?

Adjoint process: $\frac{d\alpha}{dt} = \frac{1}{\tau}\alpha, \quad \alpha(T) = \frac{dL}{dy(T)}$ Exponential growth

Parameter gradients: $\frac{d\beta}{dt} = -\frac{1}{\tau^2}\alpha y, \quad \beta(T) = 0$

- Solved backwards in time, $\beta(0) = d_{\tau}f = \left(\frac{T}{\tau^2}\right) \left(\frac{dL}{dy(T)}\right) y_0 \exp\left[-\frac{T}{\tau}\right]$

What if our function is a solution to an ODE?

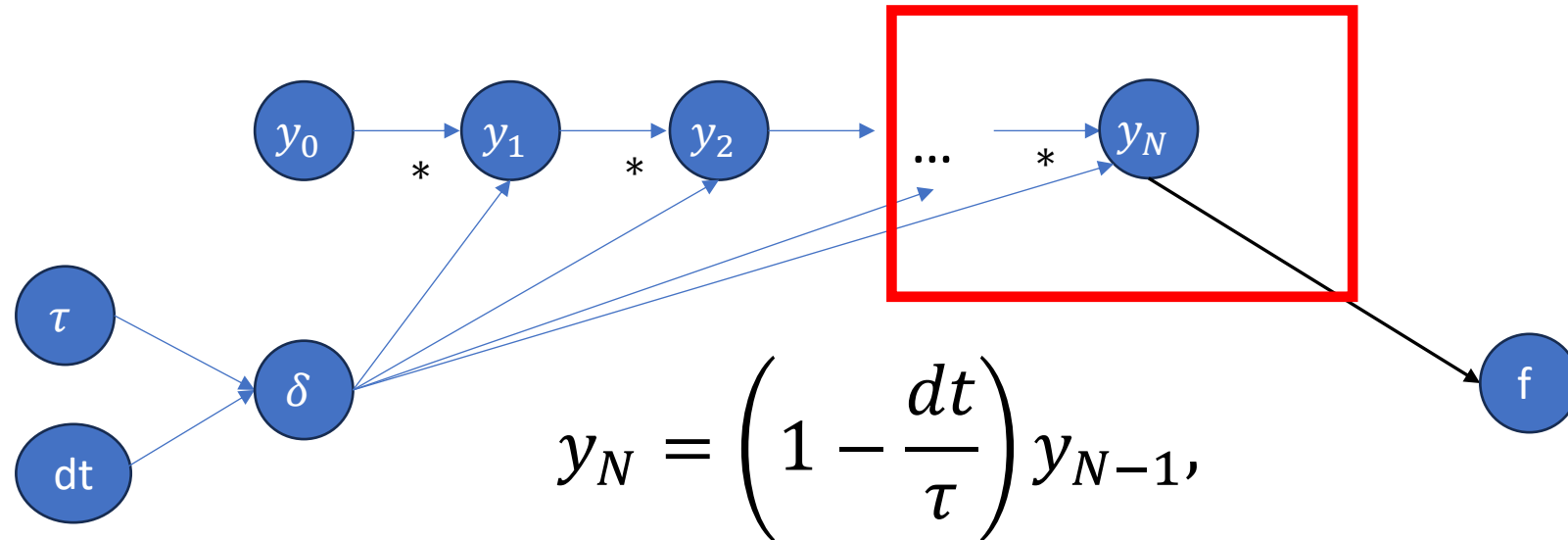


- Backpropagation a single step:

$$\frac{df}{d\tau} = \frac{dL}{dy_N} \cdot \frac{dy_N}{d\tau}$$

$$\frac{dy_N}{d\tau} = \frac{d}{d\tau} \left[\left(1 - \frac{dt}{\tau}\right) y_{N-1} \right]$$

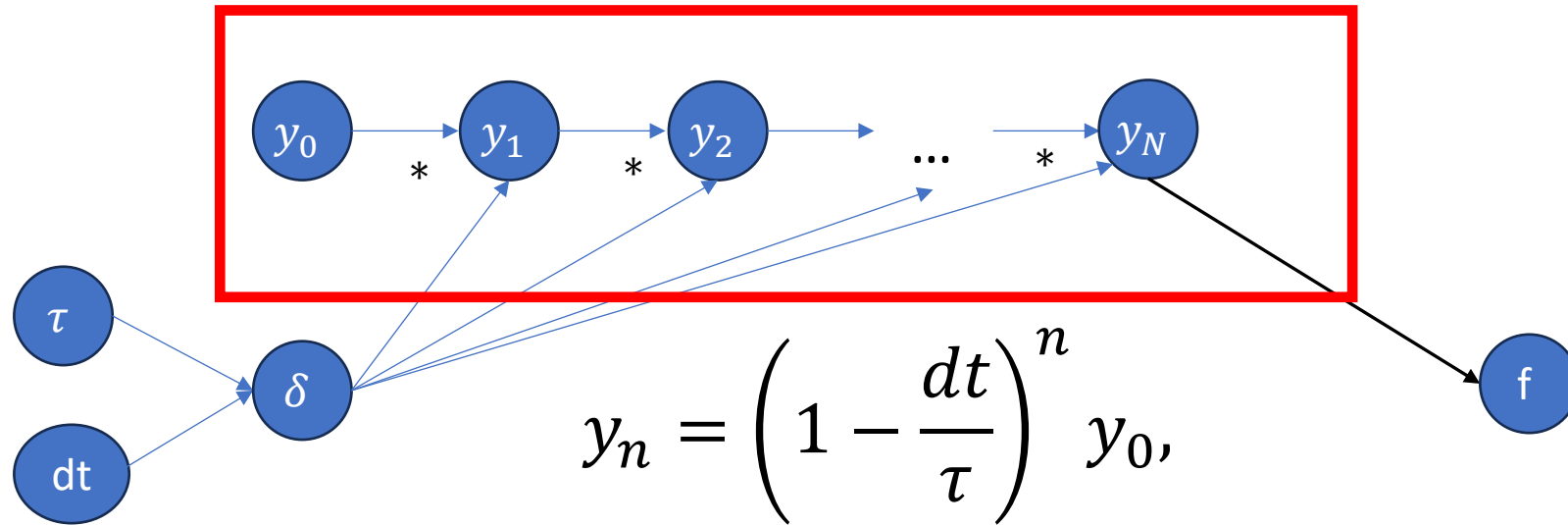
What if our function is a solution to an ODE?



- Backpropagation a single step:

$$\frac{df}{d\tau} = \frac{dL}{dy_N} \cdot \frac{dy_N}{d\tau} = \frac{dL}{dy_N} \cdot \left(\frac{dt}{\tau^2} y_{N-1} + \left(1 - \frac{dt}{\tau}\right) \frac{dy_{N-1}}{d\tau} \right)$$

What if our function is a solution to an ODE?



- Backpropagation all the way:

$$AD + ODE: \quad \frac{df}{d\tau} = \frac{T}{\tau^2} \frac{dL}{dy_N} y_0 \left(1 - \frac{T}{N\tau}\right)^{N-1}, \quad T = Ndt$$

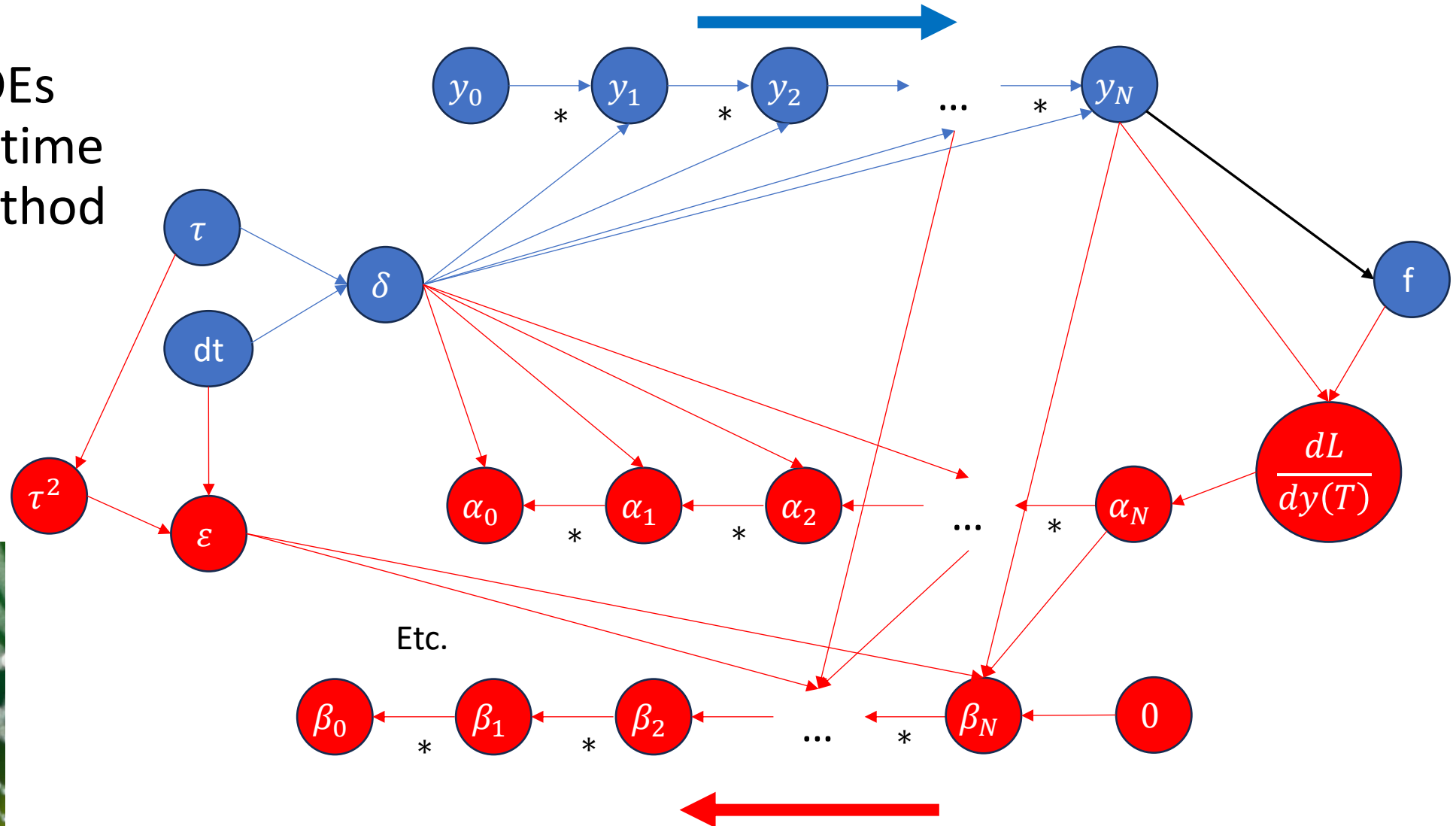
Analytic adjoint state method:

$$\beta(0) = \frac{T}{\tau^2} \frac{dL}{dy(T)} y_0 \exp \left[-\frac{T}{\tau} \right]$$

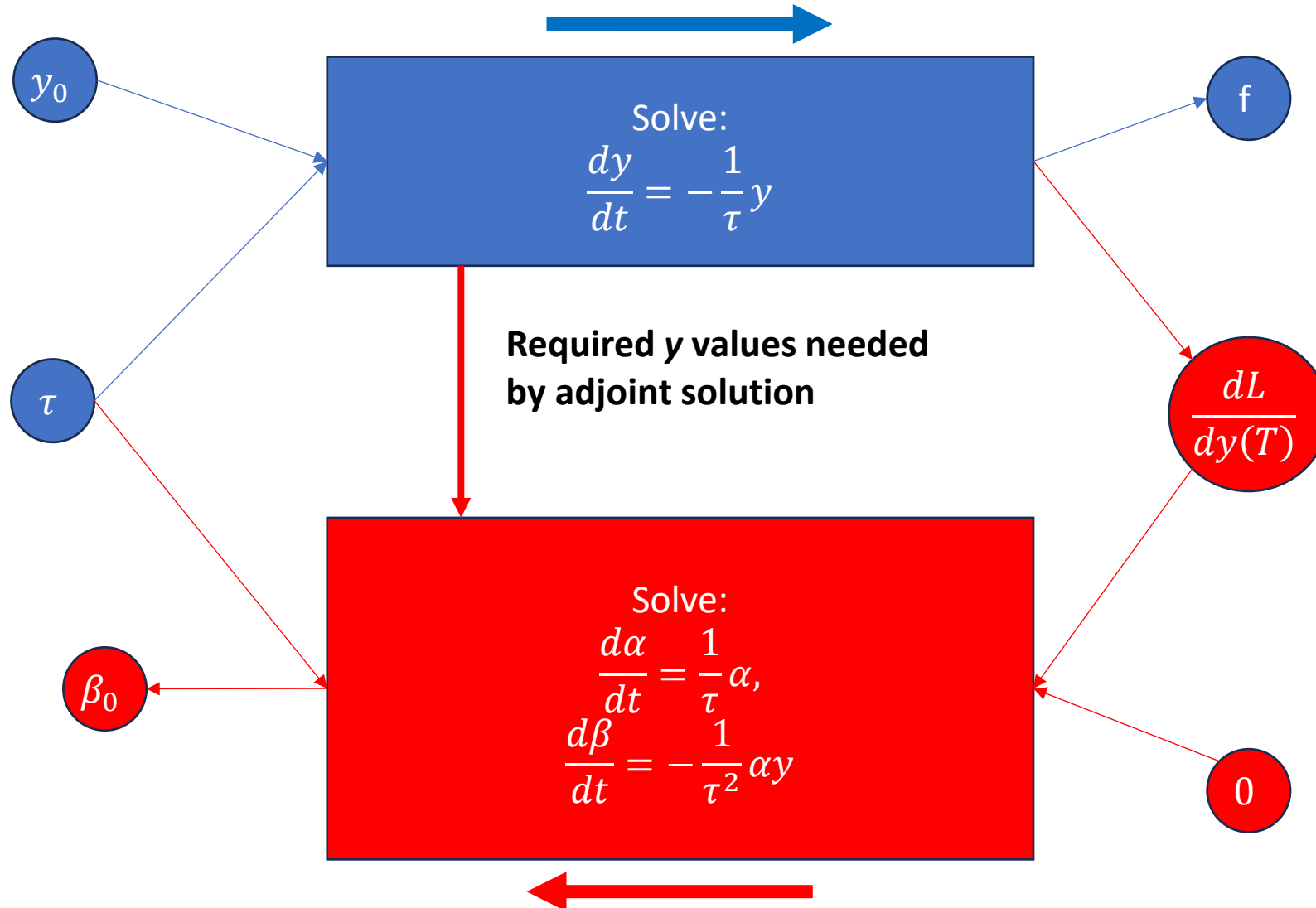
Backprop of ODE solution
numerically solves adjoint
state method
= “discretise-then-optimize”

Extending the computational graph

- Solving 3 ODEs with shared time step and method



An abstraction



Numerical modelling – PDEs

- Partial differential equations (PDEs) – dependent on only many independent variables
 - For the most part, these variables are space and time
- Total derivative vs. partial derivative

$$\frac{d}{dt}[f(x, t)] = \frac{\partial f}{\partial t} + \frac{\partial x}{\partial t} \cdot \frac{\partial f}{\partial x} = \frac{\partial f}{\partial t} + v_x \cdot \frac{\partial f}{\partial x}$$

- Example: Hydrodynamics

$$\begin{cases} \frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u} = 0 \\ \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \frac{\nabla p}{\rho} = \mathbf{g} \\ \frac{\partial e}{\partial t} + \mathbf{u} \cdot \nabla e + \frac{p}{\rho} \nabla \cdot \mathbf{u} = 0 \end{cases}$$



Numerical modelling – PDEs

- *Method of lines* for time dependent PDEs
 - Discretising all but the time dimension turns PDE → system of ODEs

- For example, the heat equation:

$$\frac{\partial T(x, t)}{\partial t} = k \frac{\partial^2 T(x, t)}{\partial x^2}$$

- Becomes (centred-space differencing):

$$\frac{dT(x_i, t)}{dt} = k \left(\frac{T(x_{i+1}, t) - 2T(x_i, t) - T(x_{i-1}, t)}{\Delta x^2} \right)$$

Numerical modelling – PDEs

- *Method of lines* for time dependent PDEs
 - Discretising all but the time dimension turns PDE → system of ODEs

- For example, the heat equation:

$$\frac{\partial T(x, t)}{\partial t} = k \frac{\partial^2 T(x, t)}{\partial x^2}$$

- Becomes:

$$\frac{d\mathbf{T}(t)}{dt} = k \underline{D} \mathbf{T}(t)$$

- Where \mathbf{T} is a vector of $T(x_i, t)$ at discrete points x_i

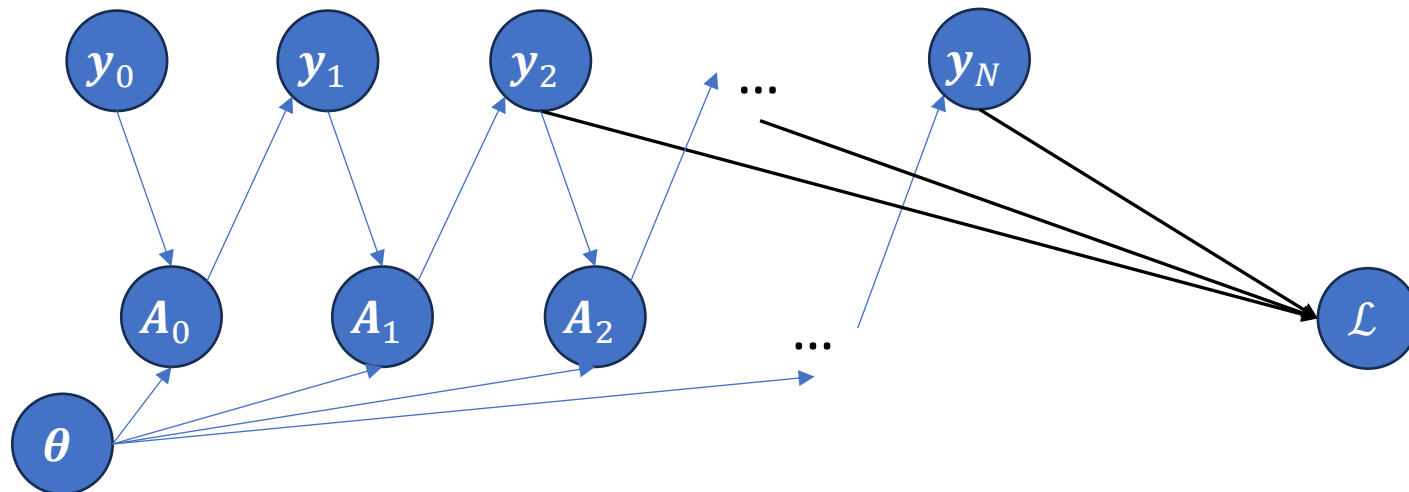
What if our function is a solution to a PDE?

- Spatial discretisation makes time-space PDEs into time ODEs
- Explicit time-stepping easiest:

$$\frac{d\mathbf{y}}{dt} = L(\mathbf{y}, \boldsymbol{\theta}) \xrightarrow{\text{discretise}} \mathbf{y}_{n+1} = (\mathbf{I} - \underline{L}(\mathbf{y}_n, \boldsymbol{\theta})dt)\mathbf{y}_n = \underline{A}_n\mathbf{y}_n$$

*Initial
conditions*

*Model
parameters*



We can modify our initial conditions or our model parameters to minimise the loss

Recipe for differentiable solver

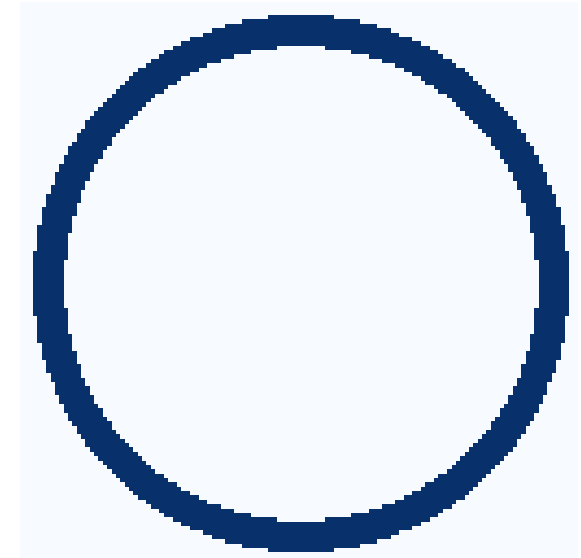
1. Specify your ODE/PDE
2. Choose spatial (and temporal) differencing scheme
3. Define loss term for optimisation purposes
4. Compute adjoint system via 'discretise-then-optimize' or 'optimize-then-discretise' strategies

Backpropagating through a fluid simulation

- Modify the initial conditions (velocities) to match some input image after a set simulation time

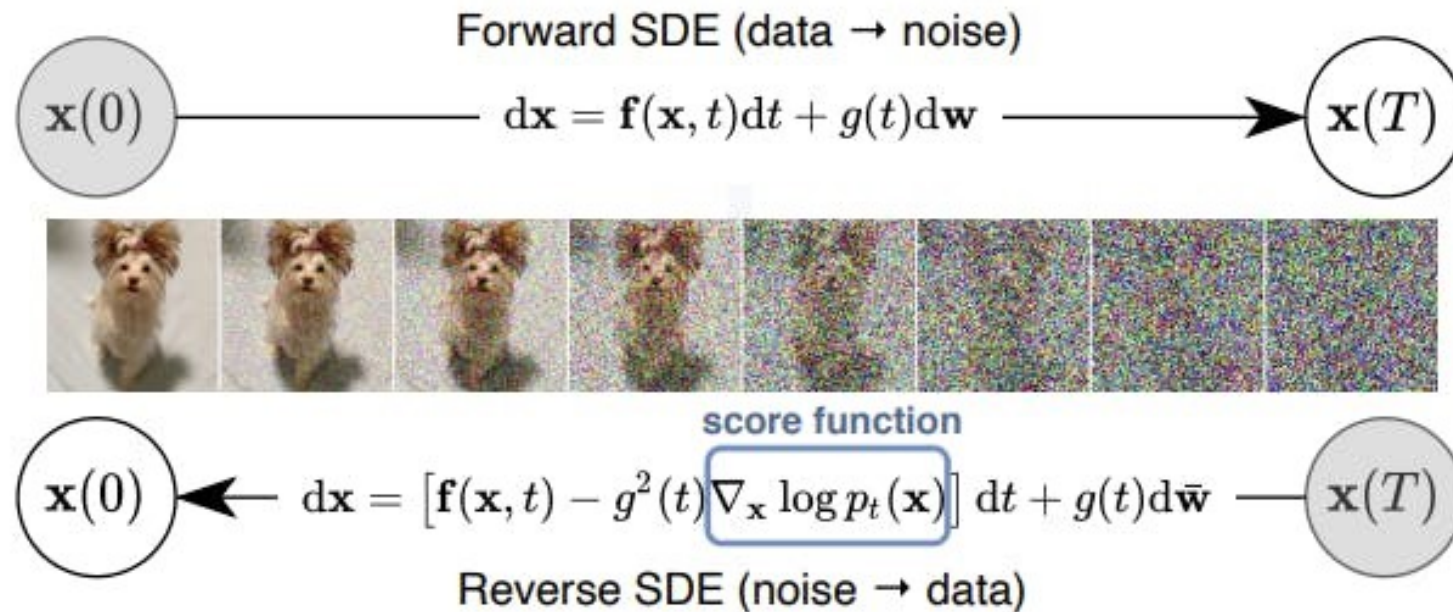
Steps:

1. Write a fluid simulator in a differentiable programming framework – in this case AutoGrad
2. Initialise some densities and x,y velocities on n_x by n_y grid
3. Find distance to target image (loss) after N simulation iterations
4. Use Jacobian of loss to update initial velocities ($2 \times n_x \times n_y$ values)



Differential equations in pure ML

- Diffusion models are based on stochastic differential equations (SDEs)



- SDEs solve for a single trajectory, subject to deterministic and white noise forces

Fokker-Planck equation

- One can instead solve for the PDF of an ensemble of trajectories
SDE \rightarrow Fokker-Planck equation

$$dX_t = \mu(X_t, t) dt + \sigma(X_t, t) dW_t \qquad \frac{\partial}{\partial t} p(x, t) = -\frac{\partial}{\partial x} [\mu(x, t)p(x, t)] + \frac{\partial^2}{\partial x^2} [D(x, t)p(x, t)] .$$

- Fokker-Planck describes the probability flow in diffusion models
- Fokker-Planck is used to describe the motion of particles in a plasma
- In exercise 2, you will encounter the various simplified forms of the Fokker-Planck equation

Colab Exercise 1

- Computational graphs and AD
- Euler's method for simple ODEs
- The adjoint method for ODEs

ODE differentiable programming libraries

- Diffrax – JAX-based library providing numerical differential equation solvers
- SciMLSensitivity.jl – Julia-based library, AD and adjoints for differential equations solvers (and more...)

ODE differentiable programming libraries

- **Difffrax – JAX-based library providing numerical differential equation solvers**
- SciMLSensitivity.jl – Julia-based library, AD and adjoints for differential equations solvers (and more...)

Difffrax example

- Difffrax has 4 key features (for our purposes):

1. Terms
2. Solvers
3. Adaptive stepping
4. Adjoint

```
def dydt(t,y,args):  
    return -y/args['tau']  
  
def difffrax_solve(dydt,t0,t1,Nt,rtol=1e-5,atol=1e-5):  
    """  
    Here we wrap the difffrax diffeqsolve function such that we can run with  
    different y0s and taus over the same time interval easily  
    """  
  
    # We convert our python function to a difffrax ODETerm  
    term = difffrax.ODETerm(dydt)  
    # We chose a solver (time-stepping) method from within difffrax library  
    # Heun's method (https://en.wikipedia.org/wiki/Heun%27s\_method)  
    solver = difffrax.Heun()  
  
    # At what time points you want to save the solution  
    saveat = difffrax.SaveAt(ts=jnp.linspace(t0,t1,Nt))  
    # Difffrax uses adaptive time stepping to gain accuracy within certain tolerances  
    stepsize_controller = difffrax.PIDController(rtol=rtol, atol=atol)  
  
    return lambda y0,tau : difffrax.diffeqsolve(term, solver,  
                                                y0=y0, args = {'tau' : tau},  
                                                t0=t0, t1=t1, dt0=(t1-t0)/Nt,  
                                                saveat=saveat, stepsize_controller=stepsize_controller)  
  
t0 = 0.0  
t1 = 1.0  
Nt = 100  
  
ODE_solve = difffrax_solve(dydt,t0,t1,Nt)  
  
# Solve for specific y0 and tau  
y0 = 1.0  
tau = 0.5  
sol = ODE_solve(y0,tau)
```


Difffrax example

- Difffrax has 4 key features (for our purposes):

1. Terms
2. Solvers
3. Adaptive stepping
4. Adjoint

```
def dydt(t,y,args):
    return -y/args['tau']

def difffrax_solve(dydt,t0,t1,Nt,rtol=1e-5,atol=1e-5):
    """
    Here we wrap the difffrax diffeqsolve function such that we can run with
    different y0s and taus over the same time interval easily
    """
    # We convert our python function to a difffrax ODETerm
    term = difffrax.ODETerm(dydt)
    # We chose a solver (time-stepping) method from within difffrax library
    # Heun's method (https://en.wikipedia.org/wiki/Heun%27s\_method)
    solver = difffrax.Heun()

    # At what time points you want to save the solution
    saveat = difffrax.SaveAt(ts=jnp.linspace(t0,t1,Nt))
    # Difffrax uses adaptive time stepping to gain accuracy within certain tolerances
    stepsize_controller = difffrax.PIDController(rtol=rtol, atol=atol)

    return lambda y0,tau : difffrax.diffeqsolve(term, solver,
                                                y0=y0, args = {'tau' : tau},
                                                t0=t0, t1=t1, dt0=(t1-t0)/Nt,
                                                saveat=saveat, stepsize_controller=stepsize_controller)

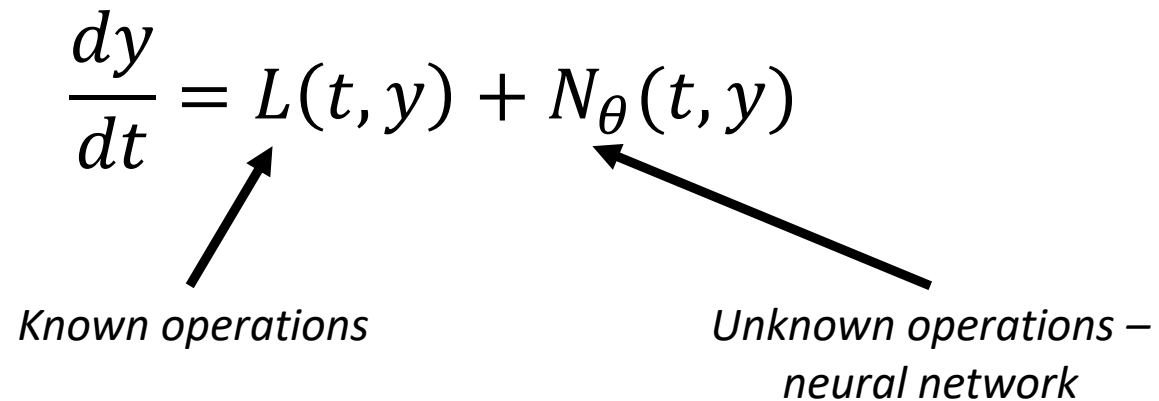
t0 = 0.0
t1 = 1.0
Nt = 100

ODE_solve = difffrax_solve(dydt,t0,t1,Nt)
def loss(inputs):
    y0 = inputs['y0']
    tau = inputs['tau']
    sol = ODE_solve(y0,tau)
    return sol.ys[-1]

inputs = {'y0' : y0, 'tau' : tau}
# Returns gradient of loss with respect to all inputs, i.e. dLdtau and dLdy0
jax.grad(loss)(inputs)
```

Neural Differential Equations

- What if what to describe a dynamical system but don't know the form of (some of) the terms?

$$\frac{dy}{dt} = L(t, y) + N_{\theta}(t, y)$$


Known operations

*Unknown operations –
neural network*

- Neural network parameters learnt using adjoint state method

Colab Exercise 2

- Diffrax library for solving ODEs
- Numerical solution to PDEs
- Differentiable PDE solvers
- Neural Differential Equations

Feedback



Assessment

- Quiz questions
- *Refer to slides and previous exercises*
- Coding exercise
- *Training data on the github:
<https://github.com/aidancrilly/AIMSLecture/data>*

Feedback

