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

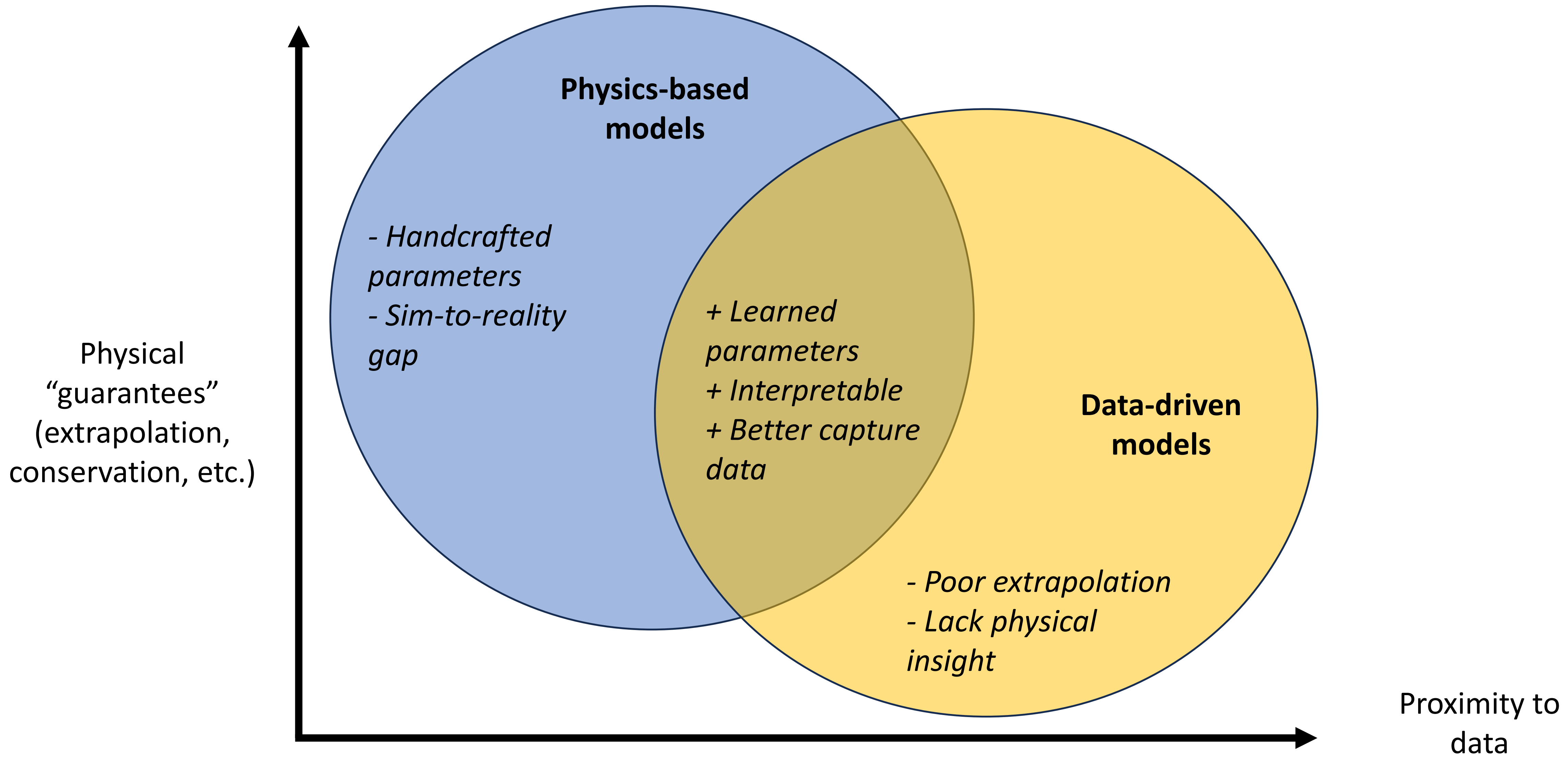
CIFS

Outline

- Differential equations
  - Adjoint state problem
- Differentiable programming
  - For ODEs
  - For PDEs
- Gradient based optimisation
- Neural differential equations

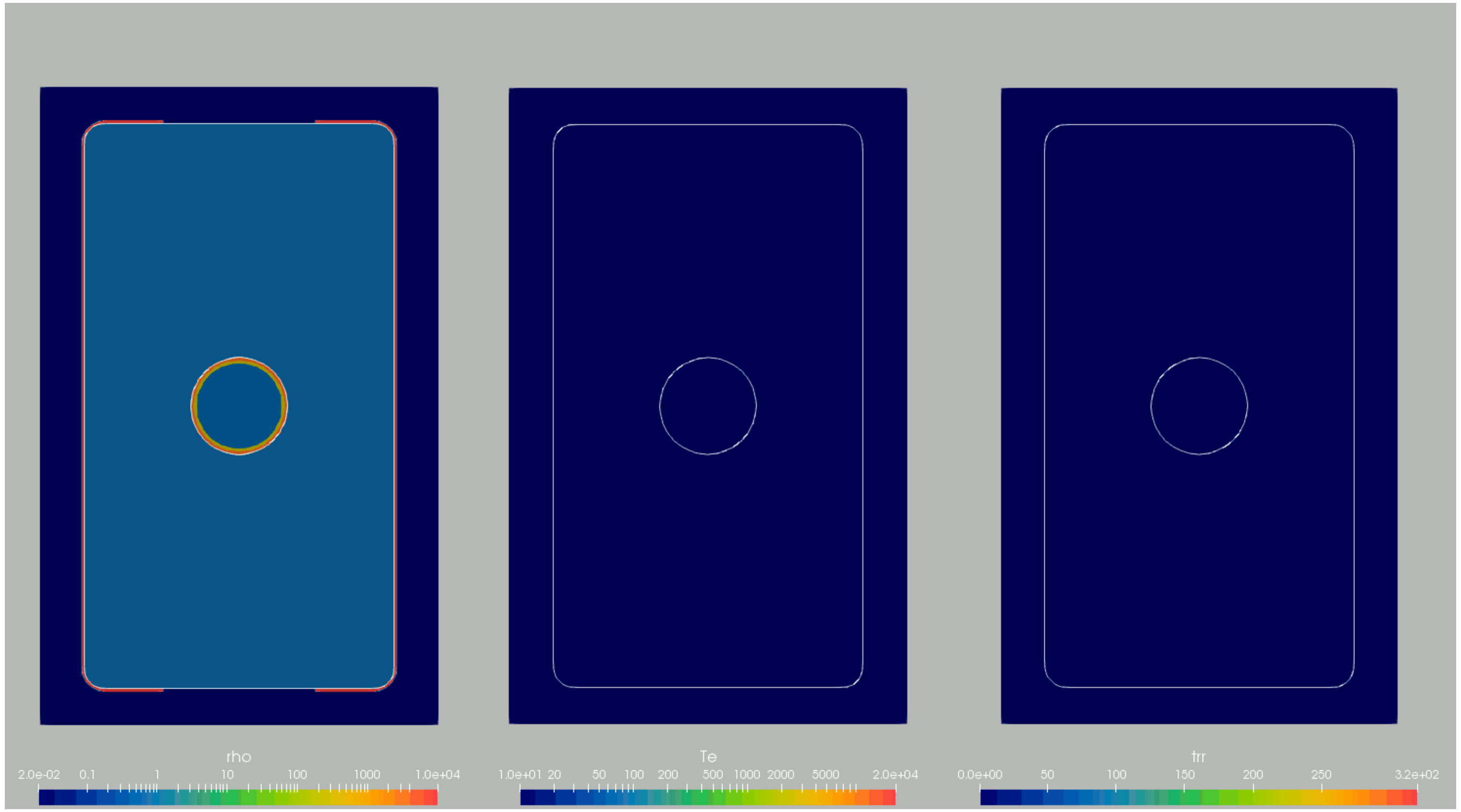
https://github.com/aidancrilly/MiniCourse-DifferentiableSimulation

Data-driven and physics-based models



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…



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…

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

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

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

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

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

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

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

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

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

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

$$\frac{\partial f_\alpha(v_\alpha,t)}{\partial t} = C(f_\alpha(v_\alpha,t)) + \frac{S_0 \delta(v_\alpha - v_b)}{4\pi v_\alpha^2} = \frac{1}{\tau_S v_\alpha^2} \frac{\partial}{\partial v_\alpha} \left\{ \frac{v_\alpha^3 T_e + v_c^3 T_i \Phi(x_i)}{m_\alpha v_\alpha} \frac{\partial f_\alpha}{\partial v_\alpha} + \left[v_\alpha^3 + v_c^3 \Phi(x_i)\right] f_\alpha \right\} + \frac{S_0 \delta(v_\alpha - v_b)}{4\pi v_\alpha^2},$$

And the list goes on…

The transport equation

$$\frac{\partial f}{\partial t} + \boxed{\boldsymbol{v} \cdot \nabla f} + \frac{1}{m}\boxed{\boldsymbol{F} \cdot \nabla_v f} = \boxed{\left(\frac{\partial f}{\partial t}\right)_{coll.}} + \boxed{\left(\frac{\partial f}{\partial t}\right)_{src}}$$

Advection **Forcing** **Collisions** **Sources**

- Encodes conservation laws on particle number, momenta and energy
- 7-dimensional (3-space, 3-velocity, 1-time), $f(x, y, z, v_x, v_y, v_z, t)$

The closure problem

- N^{th} moment equation depends on the $N+1^{th}$ moment (or more)

For example:

Integrating out velocity space:

$$\frac{\partial n}{\partial t} + \frac{\partial}{\partial x}(n\boldsymbol{V}) = 0$$

Multiply by velocity, and integrate out velocity space:

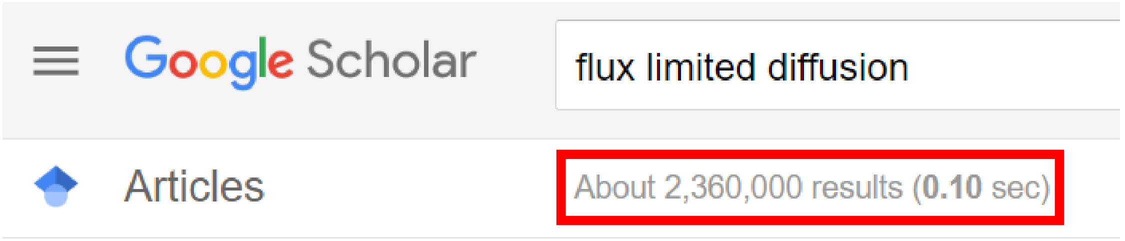
$$\frac{\partial n\boldsymbol{V}}{\partial t} + \frac{\partial}{\partial x}\left(n\boldsymbol{V}\boldsymbol{V} + \frac{1}{m}\underline{\boldsymbol{P}}\right) = 0$$

Traditional approaches

- Moment ‘ordering’ argument
  - “The N+1’th (and higher orders) are negligibly small”
- Proximity to equilibrium
  - “The steady state distribution can be used to compute the closure”
- Limiting behaviour
  - “I expect this limiting behaviour, I invent a closure which respects this”
- Asymptotic expansion
  - “I will expand in terms of a small parameter and neglect higher order terms”

The King of Transport

- Flux limited diffusion is ubiquitous



- In general, this looks like:

Diffusion equation: $$\frac{\partial E}{\partial t} + \nabla \cdot F = Q$$

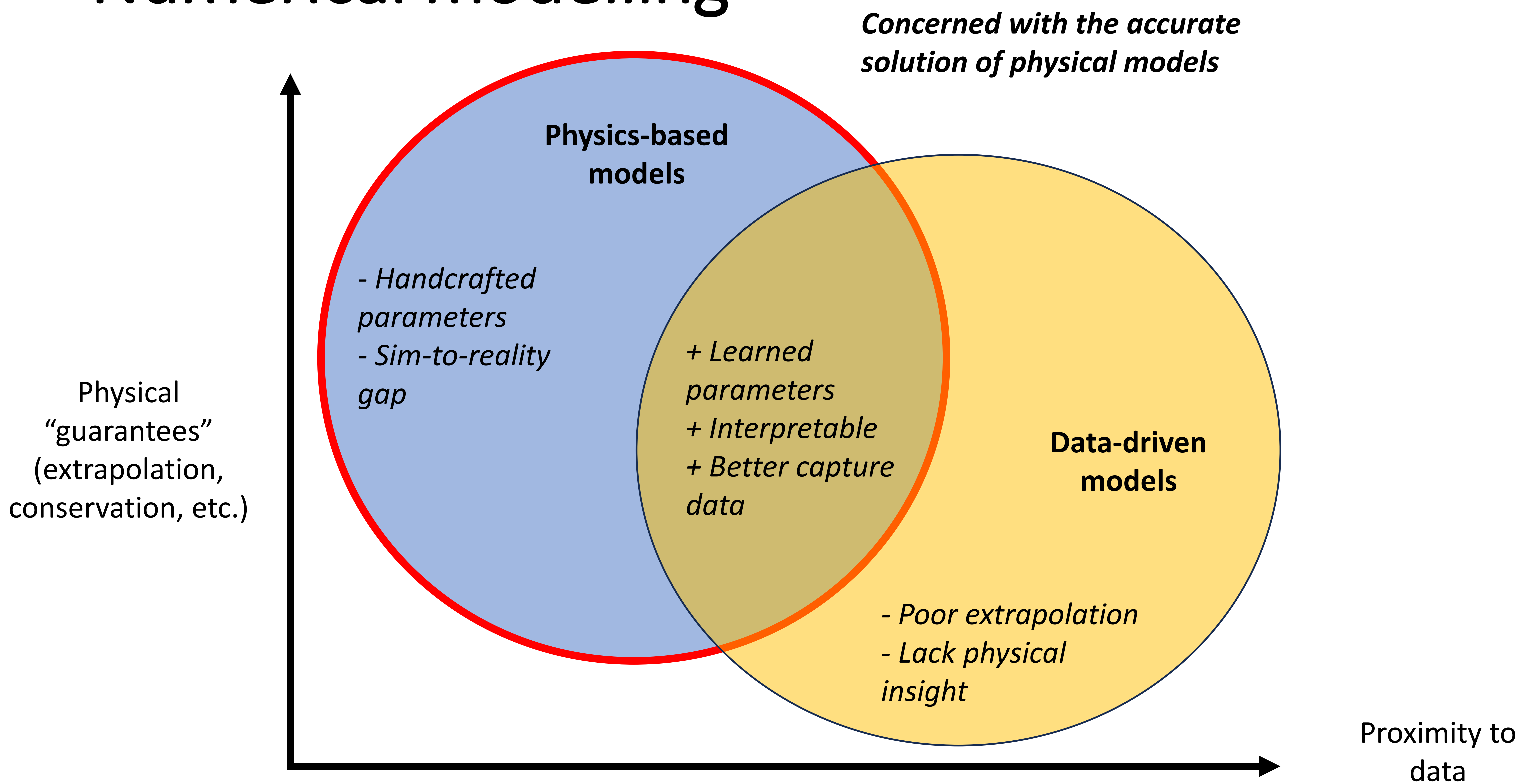
Diffusivity

Flux closure: $$F = -\lambda\, D\, \nabla E$$

Flux limiter

- Flux limiters can be *incredibly* ad-hoc – ruins the predictive power of simulation codes

Numerical modelling



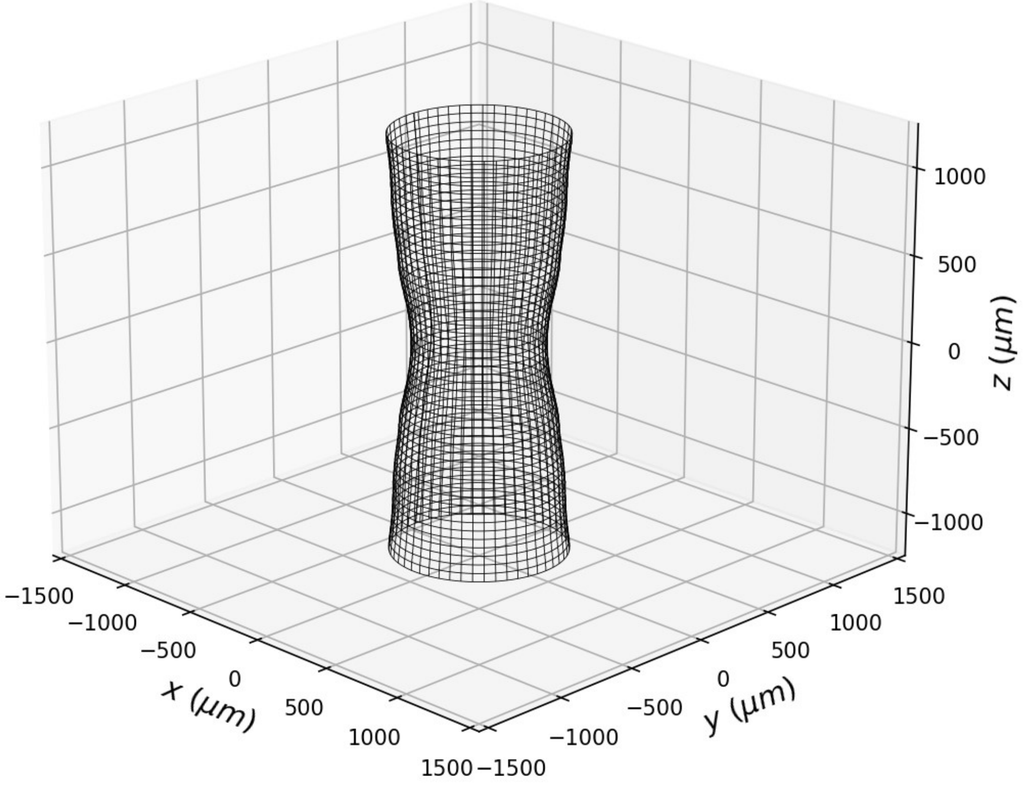
Numerical modelling – ODEs

- 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)

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



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:

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

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

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \\ t_{n+1} &= t_n + h \end{aligned}$$

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\ k_4 &= f(t_n + h, y_n + hk_3). \end{aligned}$$

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

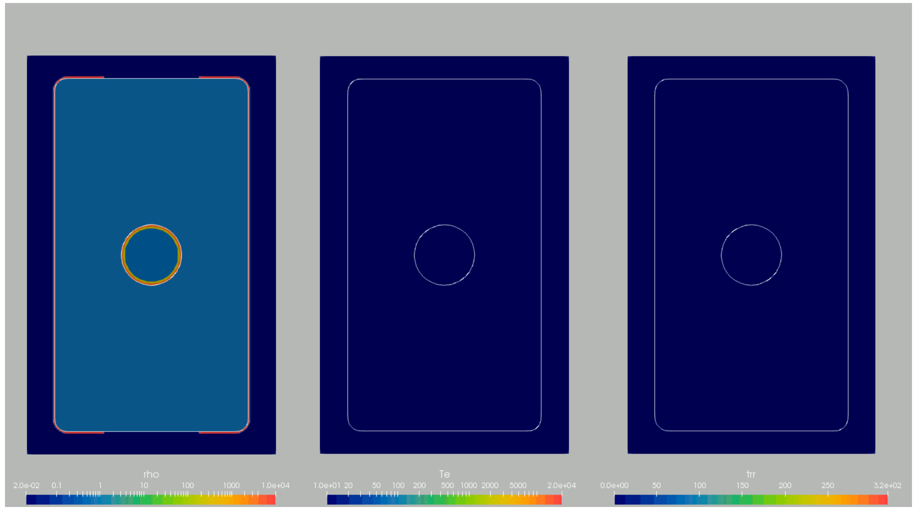
p = parameters

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: $\text{Minimise w.r.t.}\, p\text{:}\;\; f(x, p)$

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*

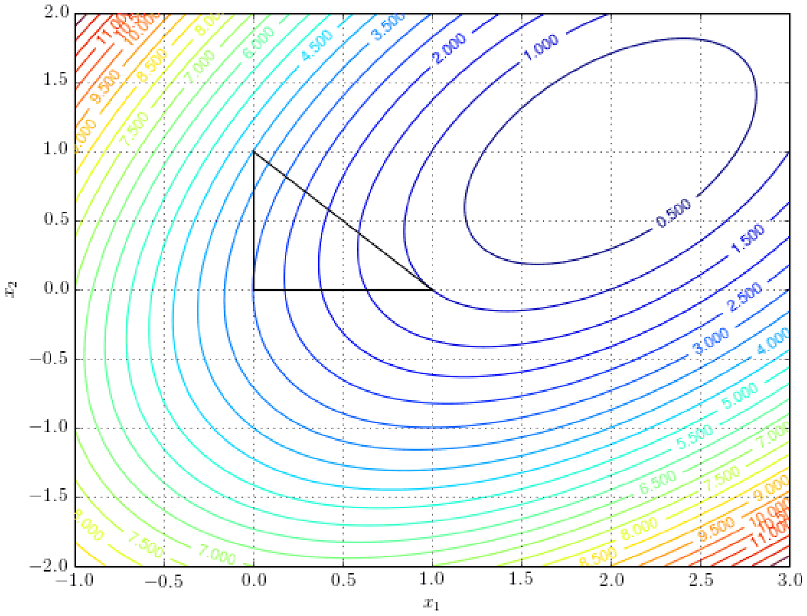
Figure of merit: $\mathrm{Minimise\ w.r.t.}\, p\!:\ f(x, p)$

Optimisation

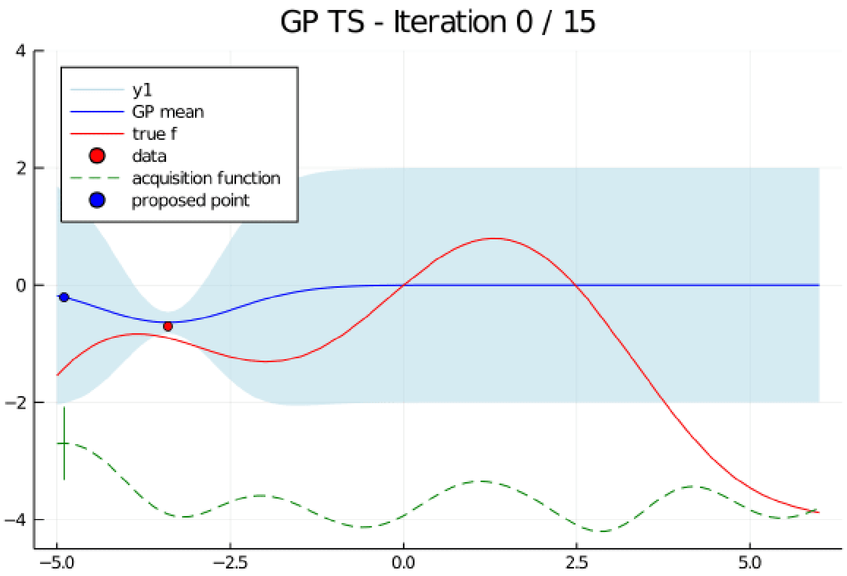
Optimisation

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

Downhill-simplex



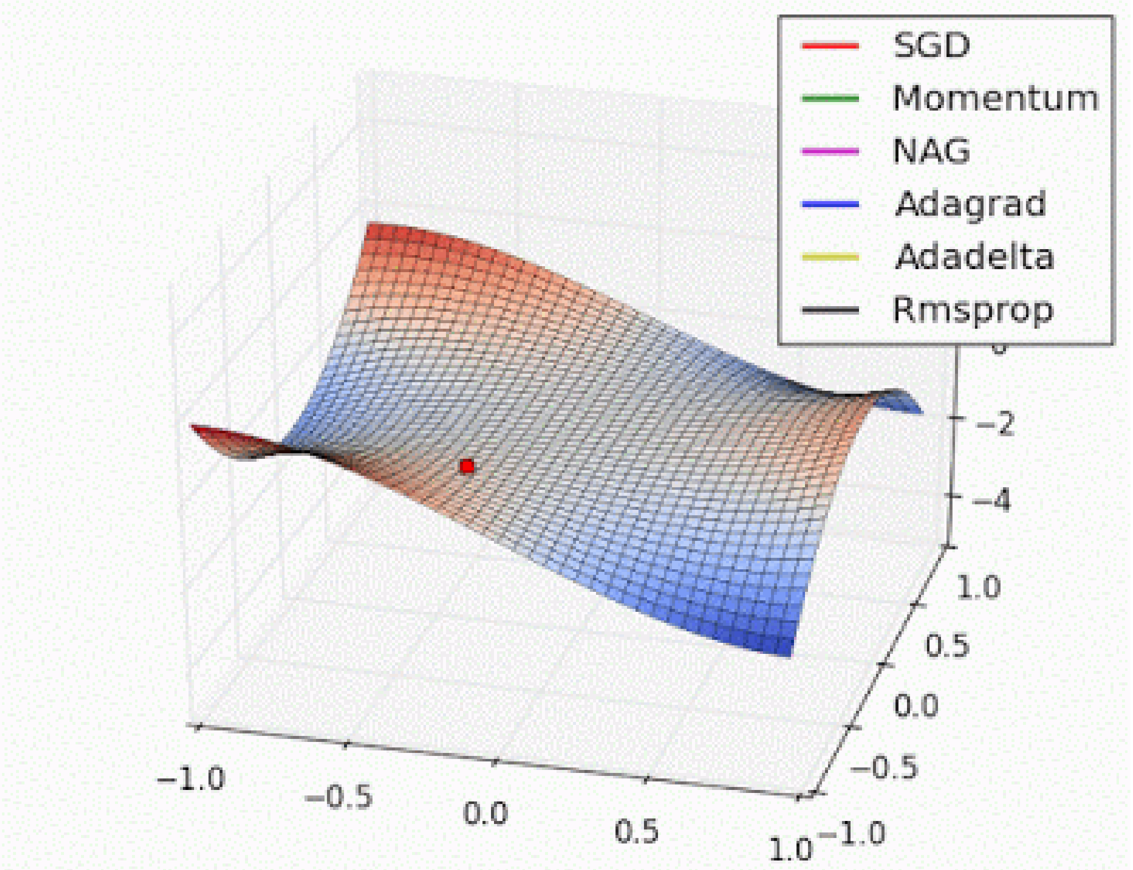
Bayesian optimisation



Optimisation

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

- We will visit the mathematics later…



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*

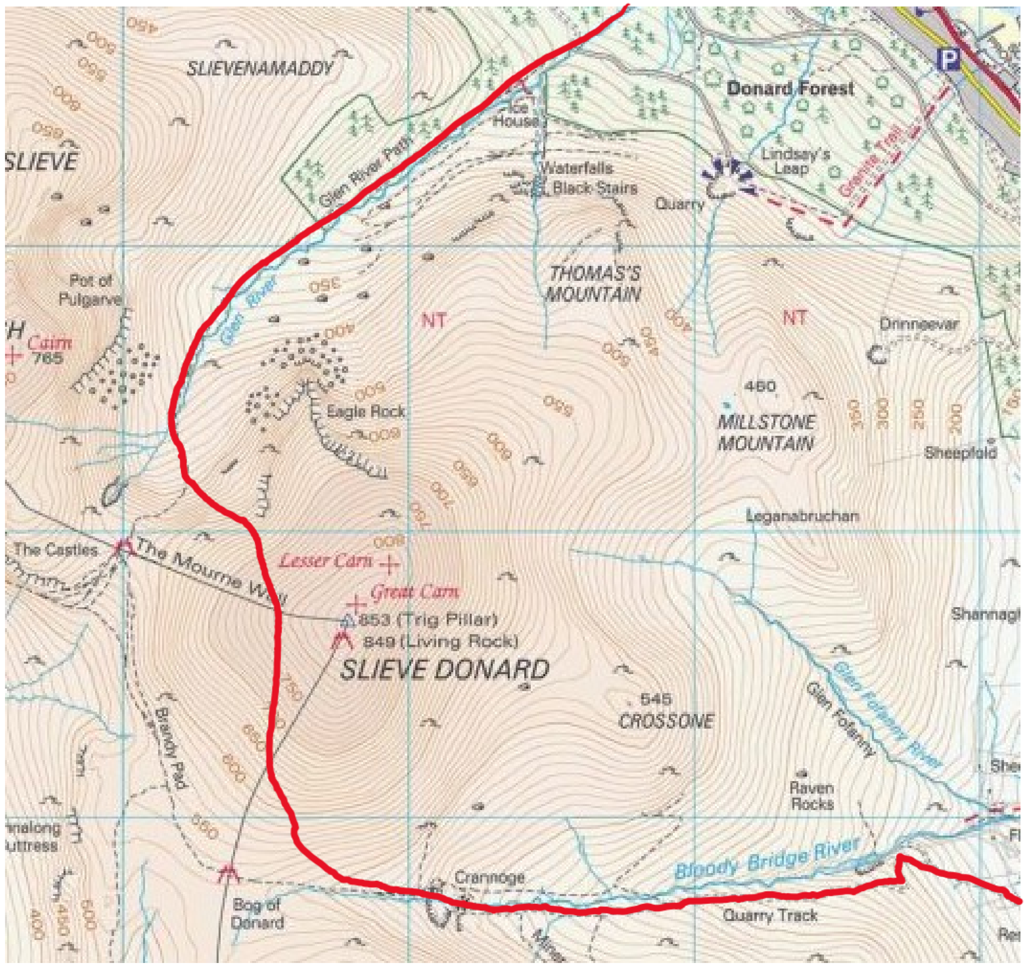
- Instead of predicting how a *single* design change influences *every* aspect of the flow, the adjoint method predicts how *every* design change influences a *single* aspect of the flow.

Constrained Optimisation

- Inverse problem in simplified form:

$$\text{Maximise w.r.t.}\, p\text{:}\;\; f(p)$$
$$\mathrm{g}(p) = 0$$

- *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.
- $\mathrm{g}(p)$ parameterises the path you take, such that if you are on the path $\mathrm{g}(p)$ = 0

Constrained Optimisation – the maths

- Combine target and constraint using "Lagrange multipliers"

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

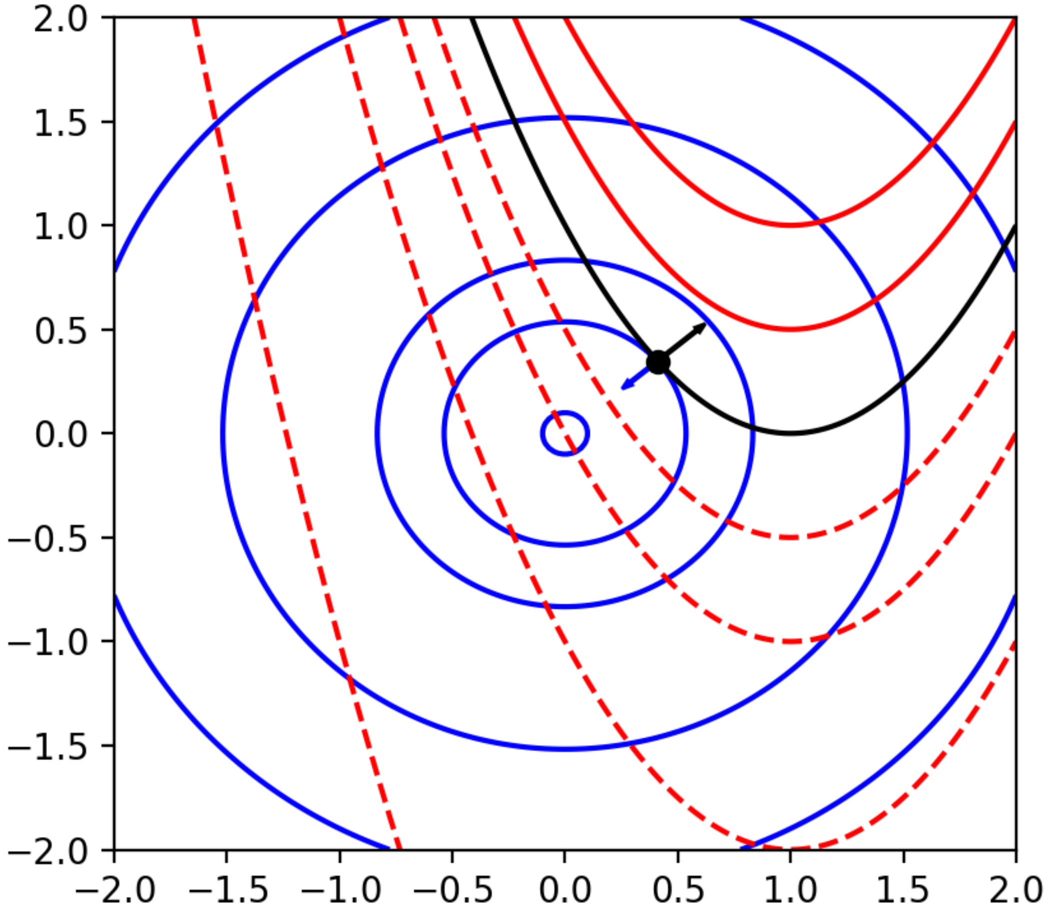
- Find optima of this function:

$$\partial_p L = 0 \rightarrow \partial_p f = -\lambda \partial_p g$$
$$\partial_\lambda L = g(p) = 0$$

- 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:

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

- Constrained optimisation → Lagrange multipliers:

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

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

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

The adjoint state problem – for ODEs

- A simple example ODE, with parameters y_0 and τ

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

- What are the formal *adjoint equations*?

  - There will be differential equations for
    1. *Adjoint process (Lagrange multiplier)*
    2. *Parameter gradients (df/dp)*

The adjoint state problem – for ODEs

- To the whiteboard!

What if our function is a solution to an ODE?

- A simple example ODE, with parameters y_0 and τ

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

- What are the formal *adjoint equations*?

Adjoint process:

$$\frac{d\alpha}{dt} = \frac{1}{\tau}\alpha, \qquad \alpha(T) = \frac{dL}{dy(T)}$$

Exponential growth

Parameter gradients:

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

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

Ref: https://proceedings.mlr.press/v144/ainsworth21a/ainsworth21a.pdf

Automatic differentiation introduction

- Numerical differentiation, finite difference methods:

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

  - Numerical error dependent on the step size used:

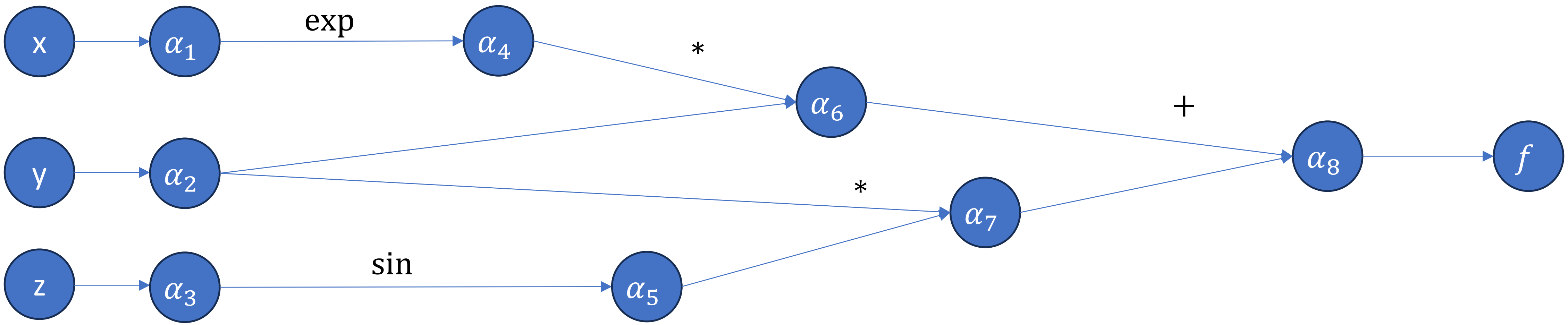
$$\frac{y(t+\varepsilon) - y(t)}{\varepsilon} = \dot{y}(t) + \frac{1}{2}\varepsilon \ddot{y}(t) + \cdots = f(t, y) + \mathrm{Error}(\varepsilon)$$

- Enter automatic differentiation…

Automatic Differentiation Intro

- Every computer program can be written as a computation graph of mathematical 'primitive' functions (*+, x, exp, sin,* etc.)
- For example:

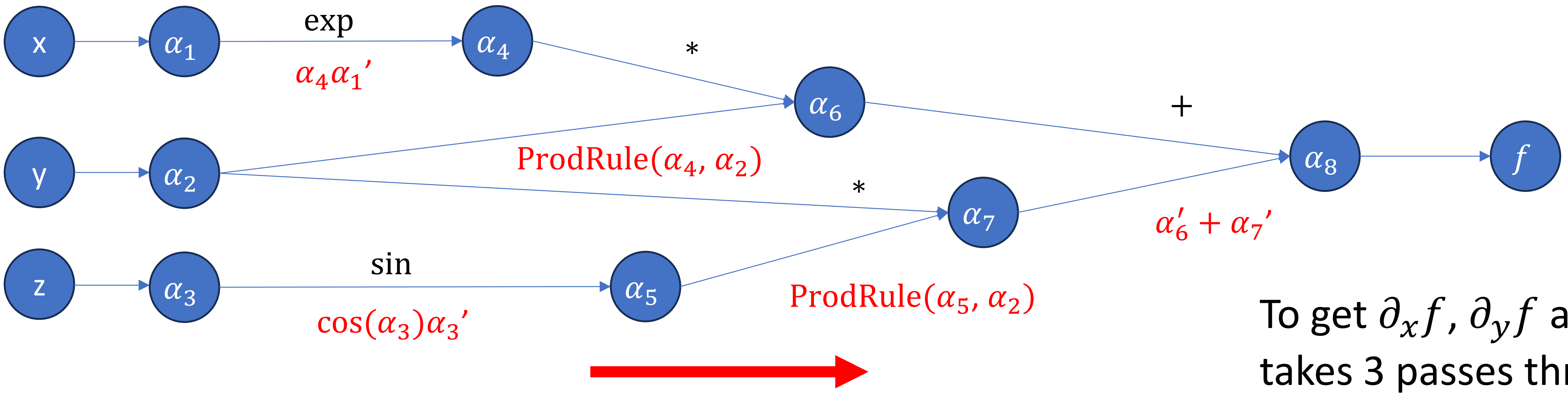
$$f(x,y,z) = y\exp(x) + y\sin(z)$$



Automatic Differentiation Intro

- Every computer program can be written as a computation graph of mathematical 'primitive' functions (*+, x, exp, sin,* etc.)
- For example:

$$f(x,y,z) = y\,\exp(x) + y\,\sin(z)$$



To get $\partial_x f$, $\partial_y f$ and $\partial_z f$ takes 3 passes through graph

"Forward mode" automatic differentiation

Automatic Differentiation Intro

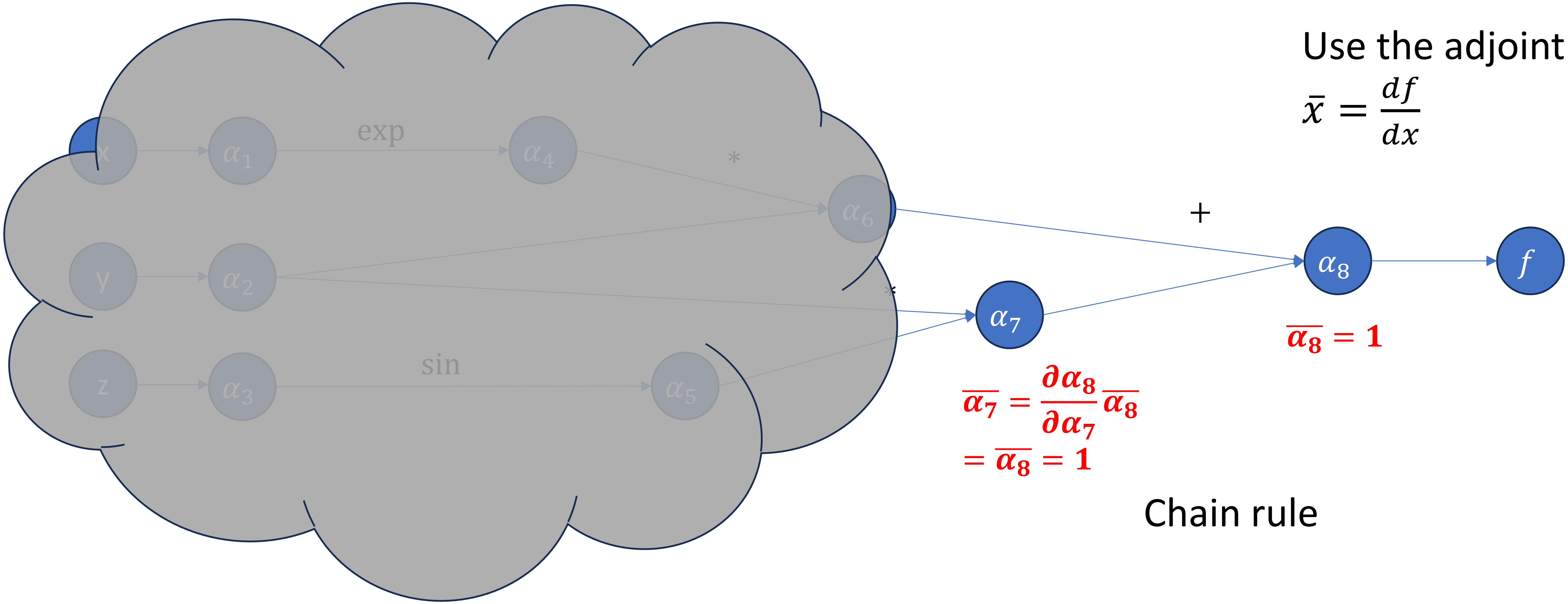
- All optimisation problems have many-to-one functions

- Reverse-mode AD provides most efficient method to compute gradients

- Define the ‘adjoint’ as gradient w.r.t. output: $\bar{x} = \frac{df}{dx}$

Automatic Differentiation Intro

- In our example:



Use the adjoint:

$$\bar{x} = \frac{df}{dx}$$

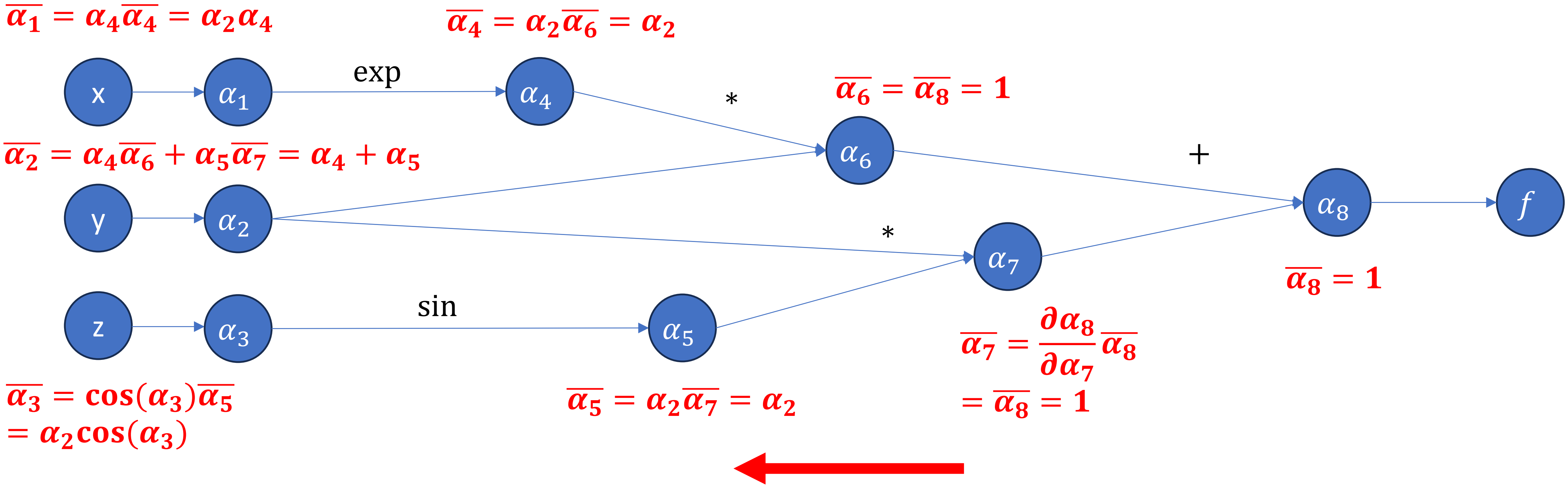
$\overline{\alpha_8} = 1$

$$\overline{\alpha_7} = \frac{\partial \alpha_8}{\partial \alpha_7}\overline{\alpha_8} = \overline{\alpha_8} = 1$$

Chain rule

Automatic Differentiation Intro

- In our example:



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

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

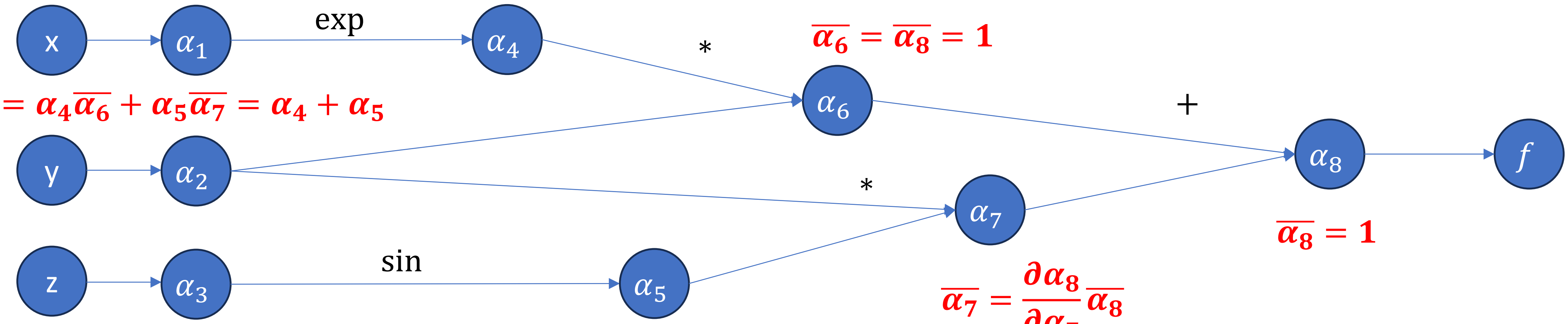
```python
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

def f(x,y,z):
    return y*jnp.exp(x)-y*jnp.sin(z)

dfdx = jax.grad(f,argnums=(0,))
dfdy = jax.grad(f,argnums=(1,))
dfdz = jax.grad(f,argnums=(2,))
```

$\overline{\alpha_1} = \alpha_4\overline{\alpha_4} = \alpha_2\alpha_4$

$\overline{\alpha_4} = \alpha_2\overline{\alpha_6} = \alpha_2$

$\overline{\alpha_6} = \overline{\alpha_8} = 1$

$\overline{\alpha_2} = \alpha_4\overline{\alpha_6} + \alpha_5\overline{\alpha_7} = \alpha_4 + \alpha_5$



$\overline{\alpha_8} = 1$

$\overline{\alpha_7} = \dfrac{\partial\alpha_8}{\partial\alpha_7}\overline{\alpha_8} = \overline{\alpha_8} = 1$

$\overline{\alpha_3} = \cos(\alpha_3)\overline{\alpha_5} = \alpha_2\cos(\alpha_3)$

$\overline{\alpha_5} = \alpha_2\overline{\alpha_7} = \alpha_2$

A single ‘back-propagation’ gives *all* gradients!

Colab exercise 0

- An introduction to JAX

https://github.com/aidancrilly/MiniCourse-DifferentiableSimulation

Back to differential equations…

The adjoint state problem – differentiable programming

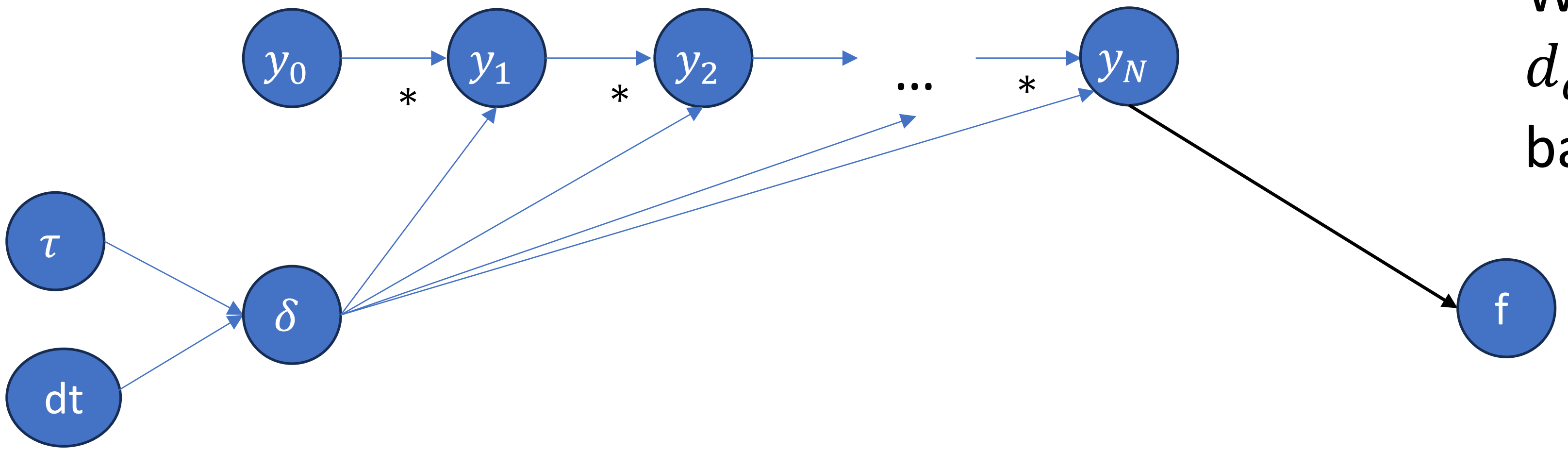
- For differential equations, the *adjoint equation* is itself another differential equation
- Backpropagation of gradients through our differential equation would implicitly solve *adjoint equations*
  - *We will show this…*

What if our function is a solution to an ODE?

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

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

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



We can compute d_τf, 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

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

- What are the formal *adjoint equations*?

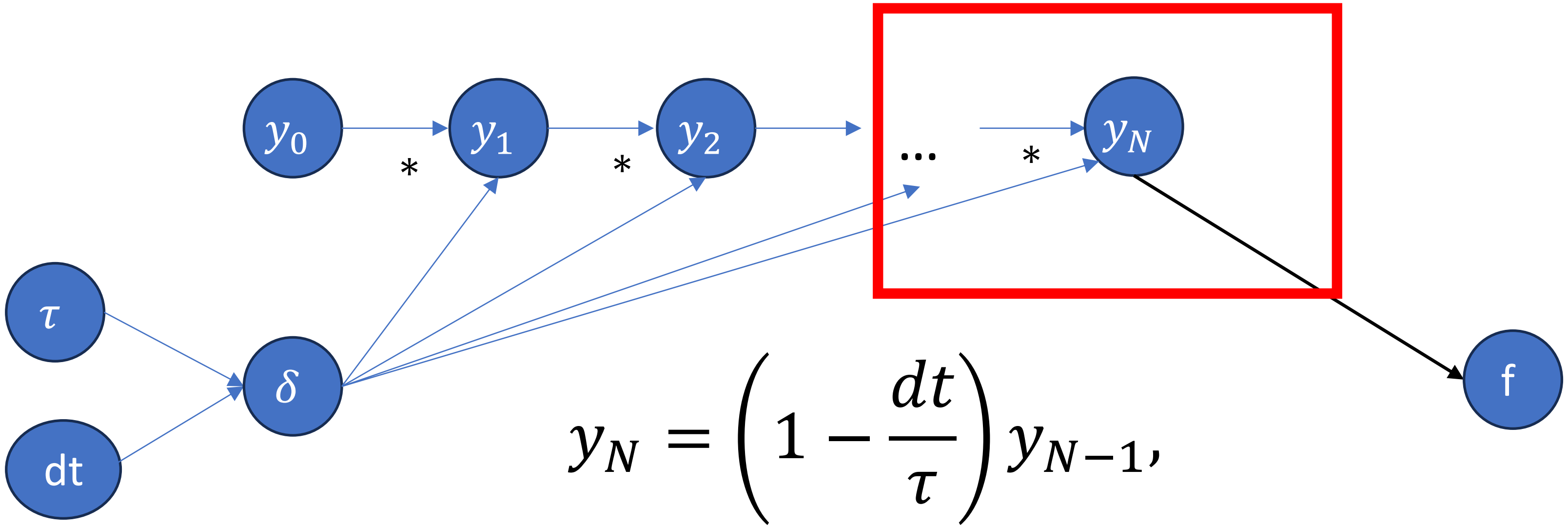
Adjoint process: $$\frac{d\alpha}{dt} = \frac{1}{\tau} \alpha, \qquad \alpha(T) = \frac{dL}{dy(T)}$$ *Exponential growth*

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

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

Ref: https://proceedings.mlr.press/v144/ainsworth21a/ainsworth21a.pdf

What if our function is a solution to an ODE?



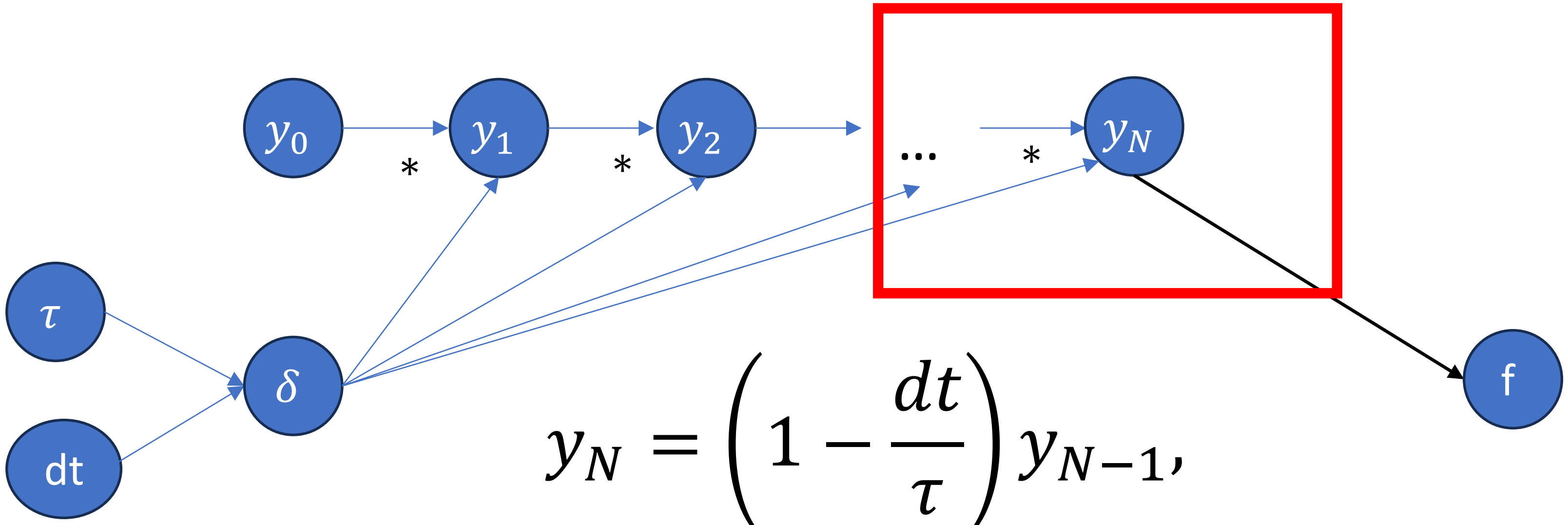
$$y_N = \left(1 - \frac{dt}{\tau}\right) y_{N-1},$$

- 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?

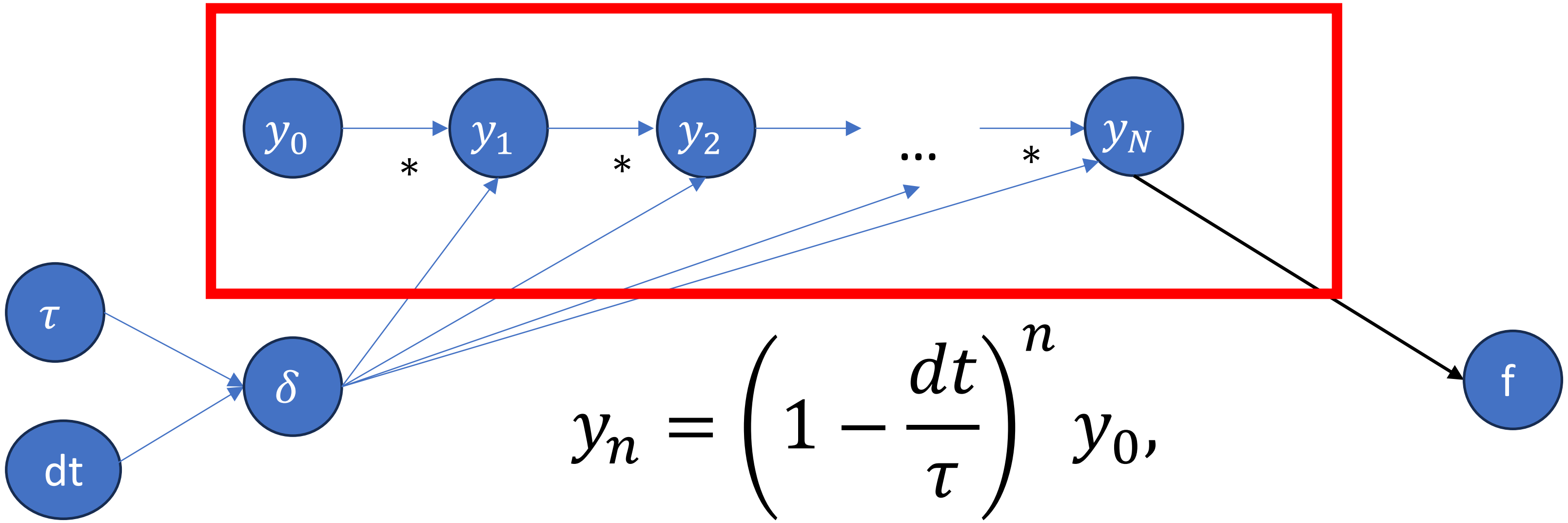


$$y_N = \left(1 - \frac{dt}{\tau}\right) y_{N-1},$$

- 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?



$$y_n = \left(1 - \frac{dt}{\tau}\right)^n y_0,$$

- Backpropagation all the way:

AD + ODE:

$$\frac{df}{d\tau} = \frac{T}{\tau^2}\frac{dL}{dy_N} y_0 \left(1 - \frac{T}{N\tau}\right)^{N-1}, \qquad \mathrm{T} = \mathrm{Ndt}$$

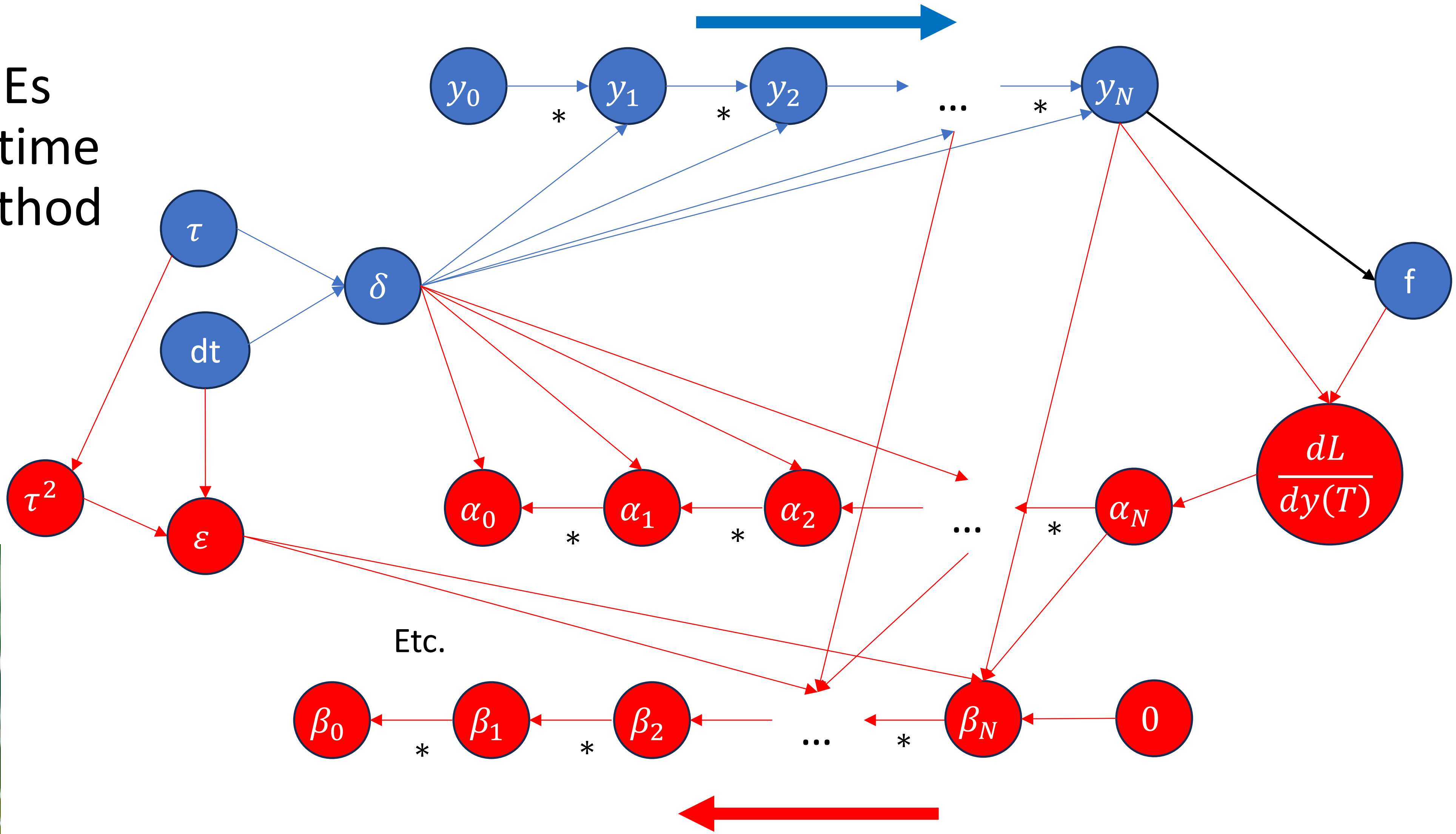
Analytic adjoint state method:

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

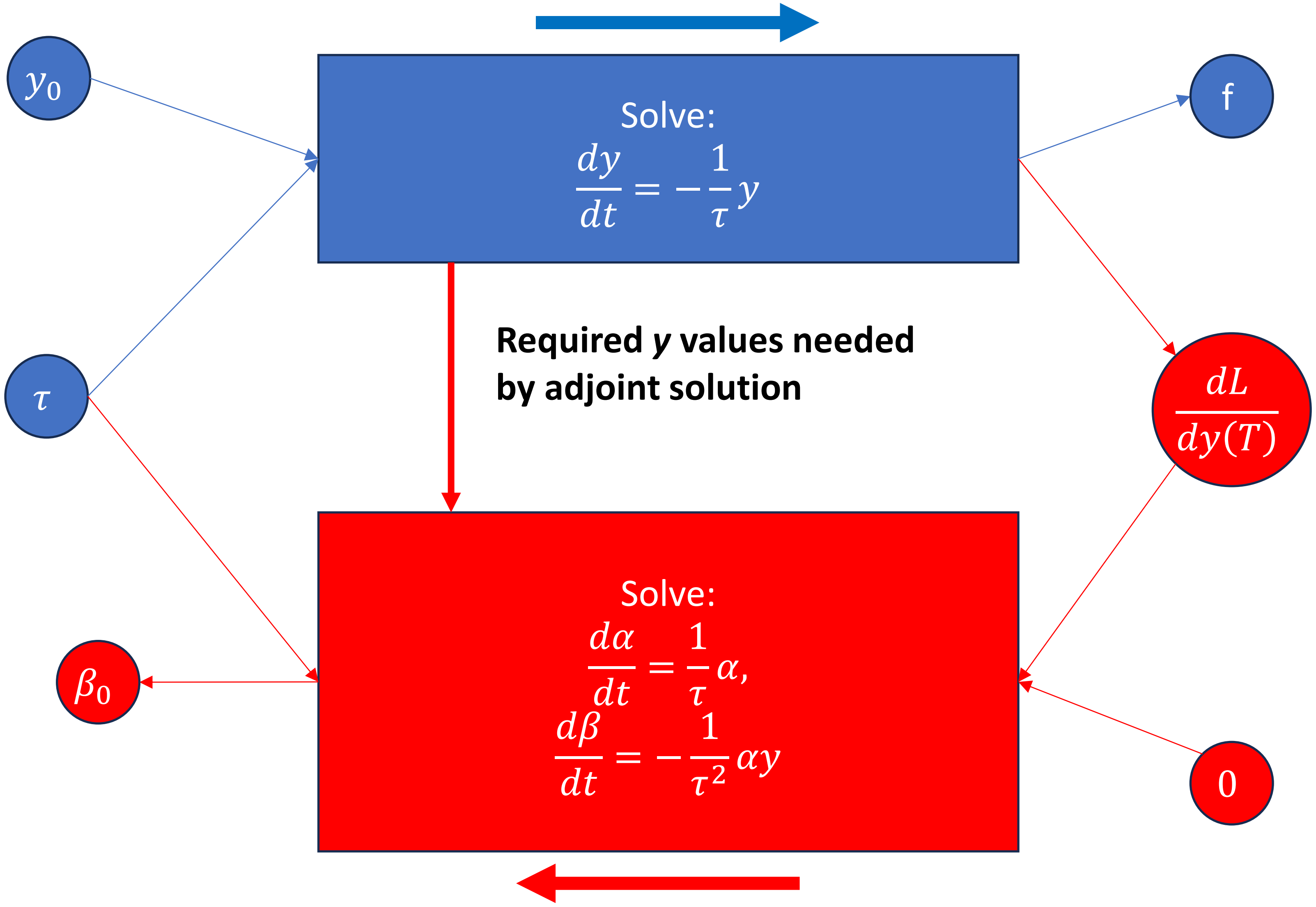
Backprop of ODE solution numerically solves adjoint state method = "discretise-then-optimise"

Extending the computational graph

- Solving 3 ODEs with shared time step and method



An abstraction



Colab Exercise 1

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

https://github.com/aidancrilly/MiniCourse-DifferentiableSimulation

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

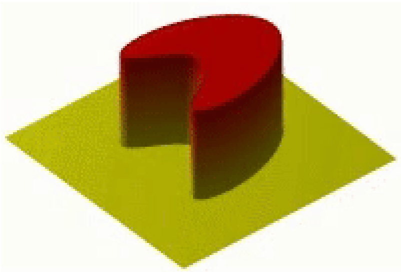
$$\begin{cases} \dfrac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u} = 0 \\ \dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \dfrac{\nabla p}{\rho} = \mathbf{g} \\ \dfrac{\partial e}{\partial t} + \mathbf{u} \cdot \nabla e + \dfrac{p}{\rho}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

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:

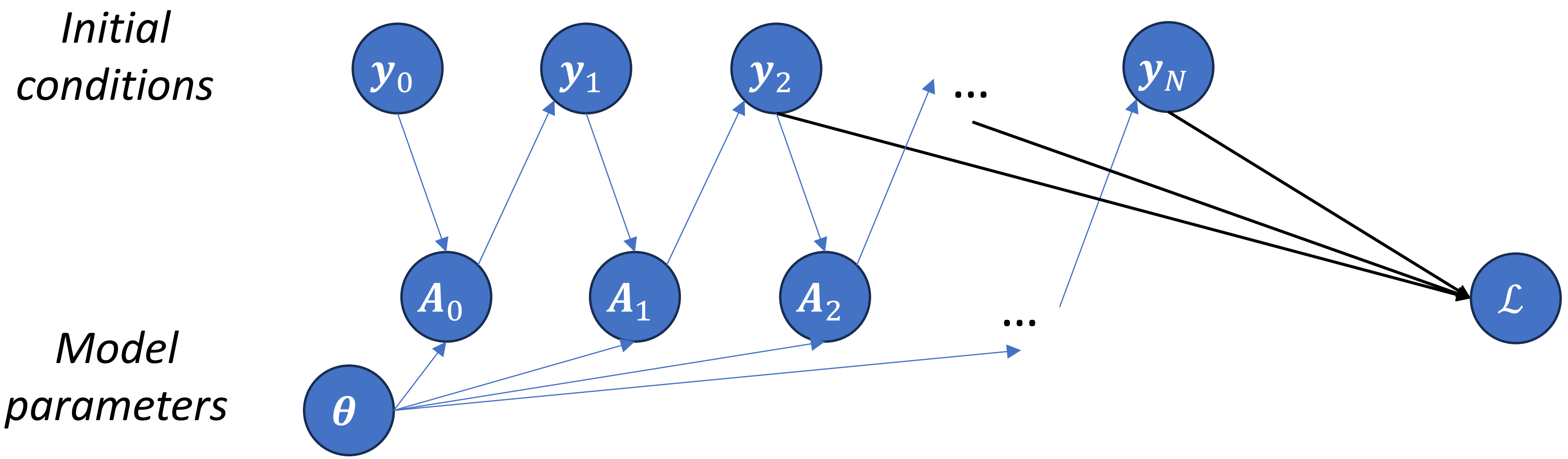
$$\frac{d\boldsymbol{T}(t)}{dt} = k\underline{D}\boldsymbol{T}(t)$$

- Where $\boldsymbol{T}$ 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:

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



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-optimise’ or ‘optimise-then-discretise’ strategies

Gradient-Based Optimisation

- Key idea is gradient descent, move your parameters in the direction of local improvement
- Some other common concepts are:
  - *Momentum,* $\boldsymbol{m}$*:* The history of gradients is used to update parameters, not just current gradient value
  - *Adaptive learning rate,* $\boldsymbol{\alpha}$*:* The step size is adapted for each parameter to penalise parameters which cause oscillations in the gradient
- In general, the parameter updates look like:

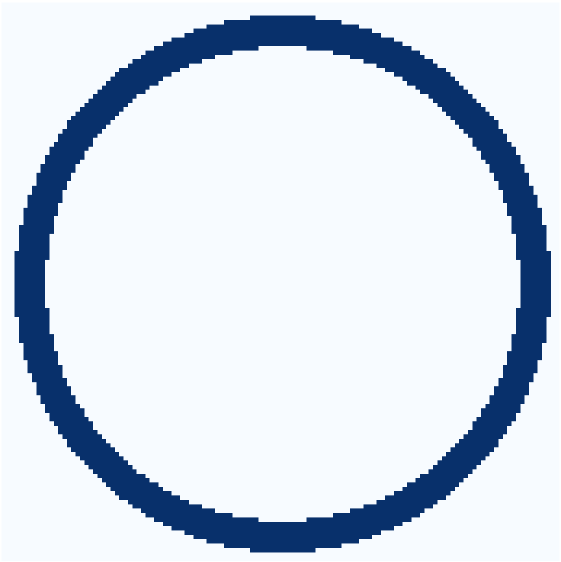
$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \boldsymbol{\alpha}_n \boldsymbol{m}_n$$

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 nx by ny grid
3. Find distance to target image (loss) after N simulation iterations
4. Use Jacobian of loss to update initial velocities (2x nx x ny values)



https://github.com/HIPS/autograd/blob/master/examples/fluidsim/fluidsim.py

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

- **Diffrax – JAX-based library providing numerical differential equation solvers**

- SciMLSensitivity.jl – Julia-based library, AD and adjoints for differential equations solvers (and more…)

Diffrax example

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

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

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

def diffrax_solve(dydt,t0,t1,Nt,rtol=1e-5,atol=1e-5):
  """
  Here we wrap the diffrax 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 diffrax ODETerm
  term = diffrax.ODETerm(dydt)
  # We chose a solver (time-stepping) method from within diffrax library
  # Heun's method (https://en.wikipedia.org/wiki/Heun%27s_method)
  solver = diffrax.Heun()

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

  return lambda y0,tau : diffrax.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 = diffrax_solve(dydt,t0,t1,Nt)

# Solve for specific y0 and tau
y0 = 1.0
tau = 0.5
sol = ODE_solve(y0,tau)
```

Diffrax example

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

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

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

def diffrax_solve(dydt,t0,t1,Nt,rtol=1e-5,atol=1e-5):
  """
  Here we wrap the diffrax 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 diffrax ODETerm
  term = diffrax.ODETerm(dydt)
  # We chose a solver (time-stepping) method from within diffrax library
  # Heun's method (https://en.wikipedia.org/wiki/Heun%27s_method)
  solver = diffrax.Heun()

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

  return lambda y0,tau : diffrax.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 = diffrax_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

https://github.com/aidancrilly/MiniCourse-DifferentiableSimulation