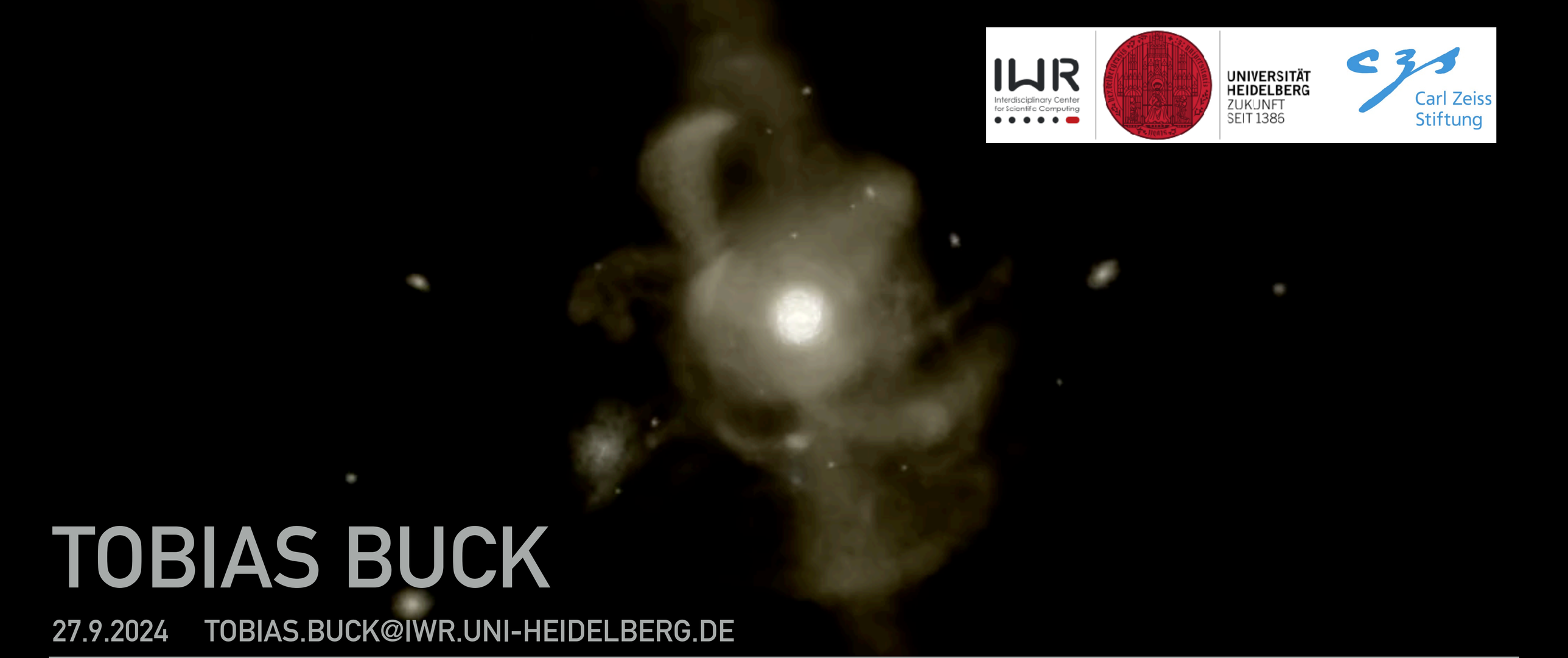


TOBIAS BUCK

27.9.2024 TOBIAS.BUCK@IWR.UNI-HEIDELBERG.DE

SCIENTIFIC ML

OUTLINE OF TODAY

1. Scientific ML
   A. Neural ODEs
   B. Symbolic Regression
   C. Operator Learning
   D. Physics Informed Neural Networks
   E. Hamiltonian Neural Networks

WHERE TO FIND THE LECTURE MATERIAL

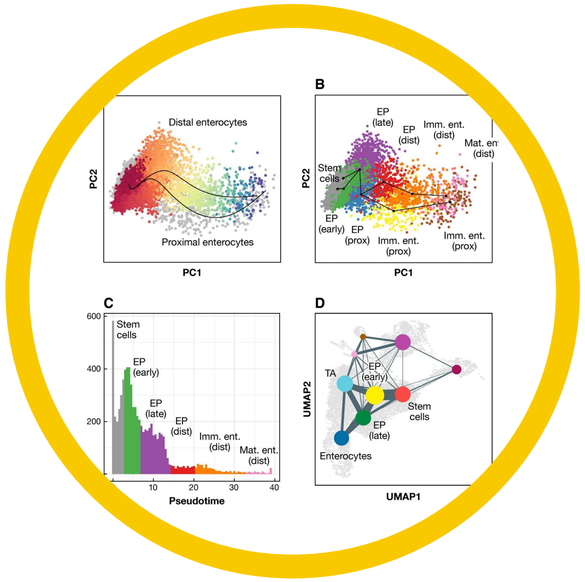
https://github.com/TobiBu/graddays

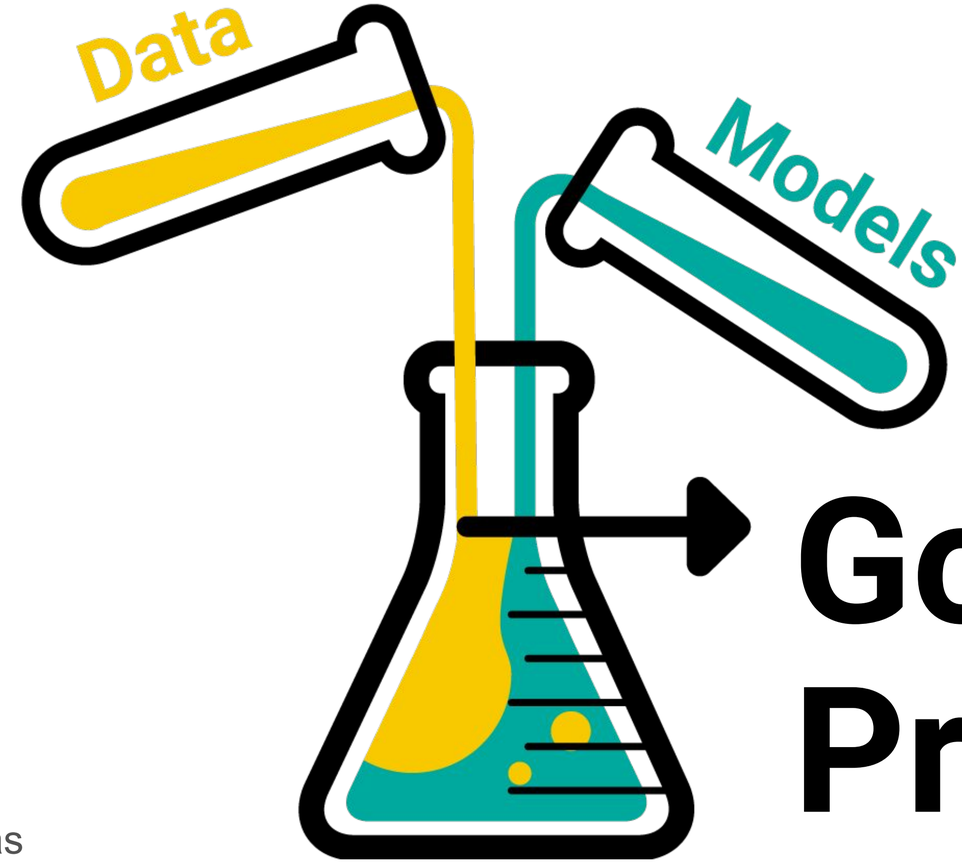
SCIENTIFIC MACHINE LEARNING

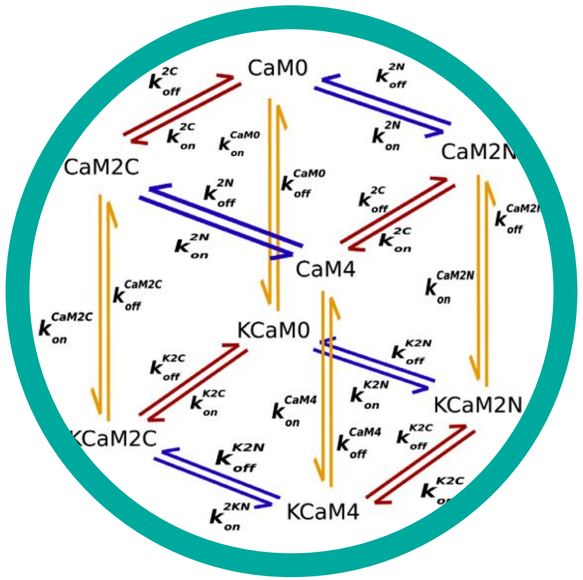
»Scientific machine learning is the burgeoning field combining techniques of machine learning into traditional scientific computing and mechanistic modeling.« [e. g. Zubov et al, 2021]

SCIENTIFIC MACHINE LEARNING IS MODEL-BASED DATA-EFFICIENT ML

How do we simultaneously use both sources of knowledge?

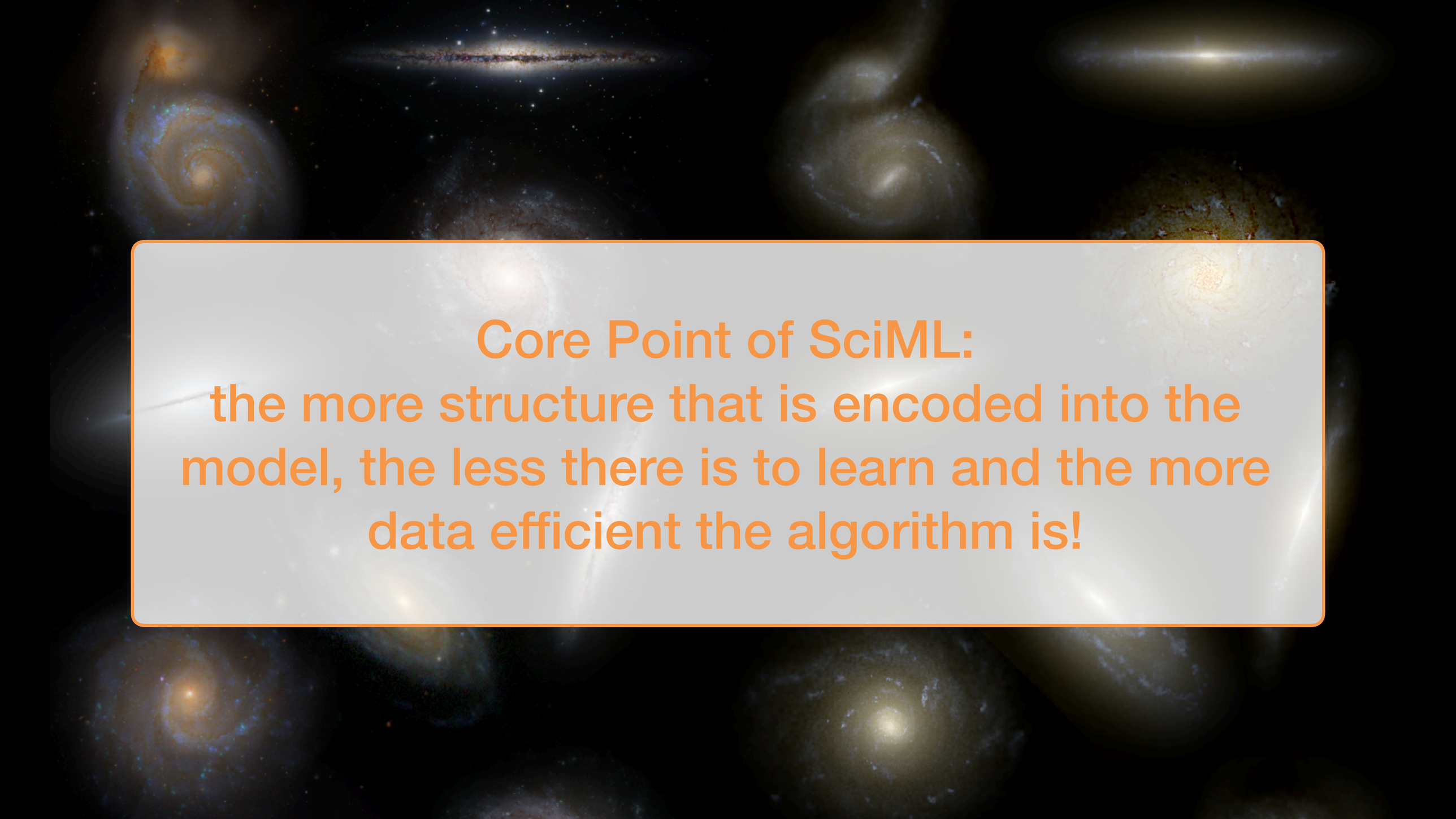






Good Predictions

slide inspired by Chris Rackauckas

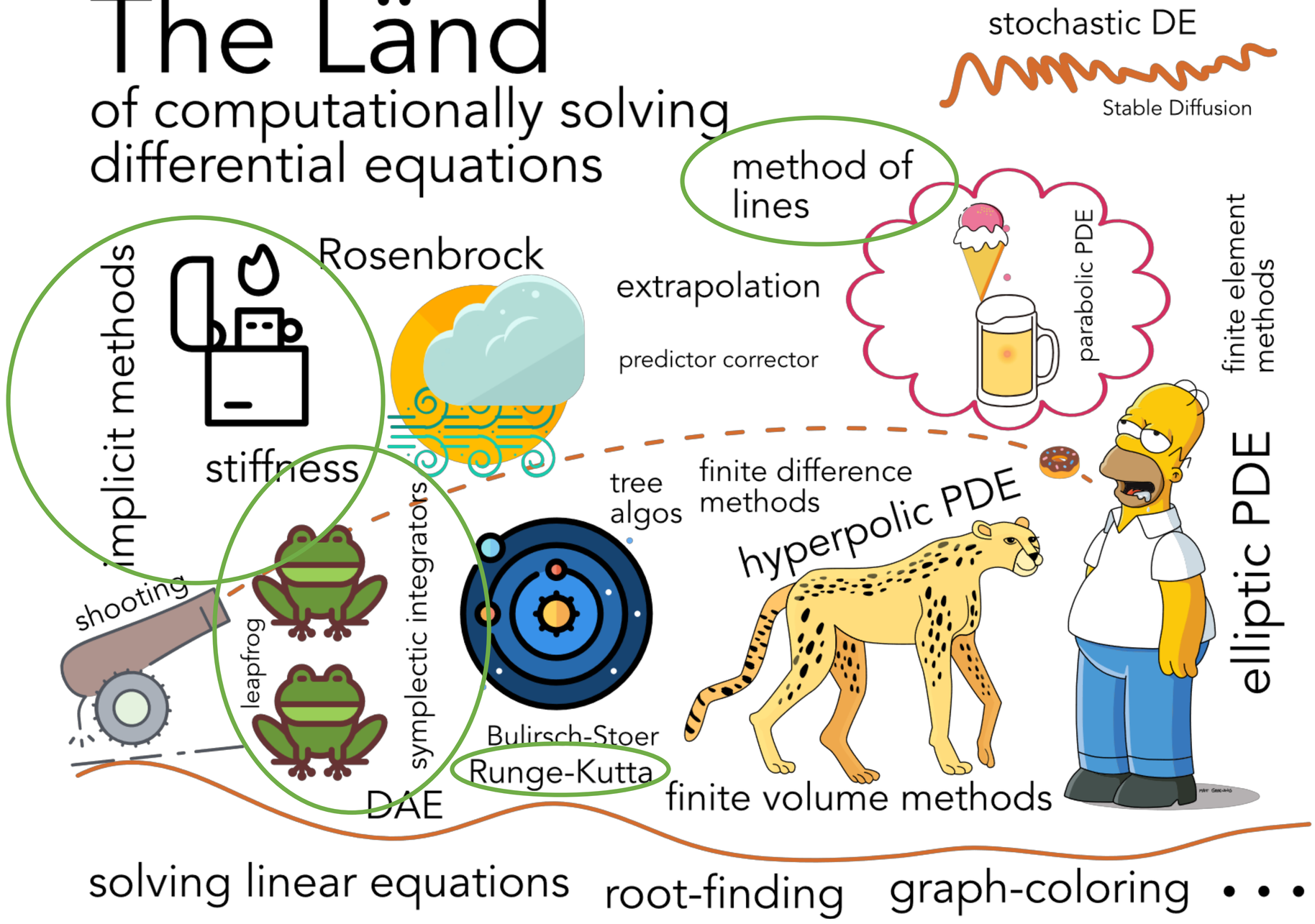

Core Point of SciML:
the more structure that is encoded into the model, the less there is to learn and the more data efficient the algorithm is!

Traditional methods for solving DEs

It's big.

The Länd

of computationally solving differential equations



Slide taken from Leonard Storcks

THE SOLUTION OF ODES

Explicit Euler Method:

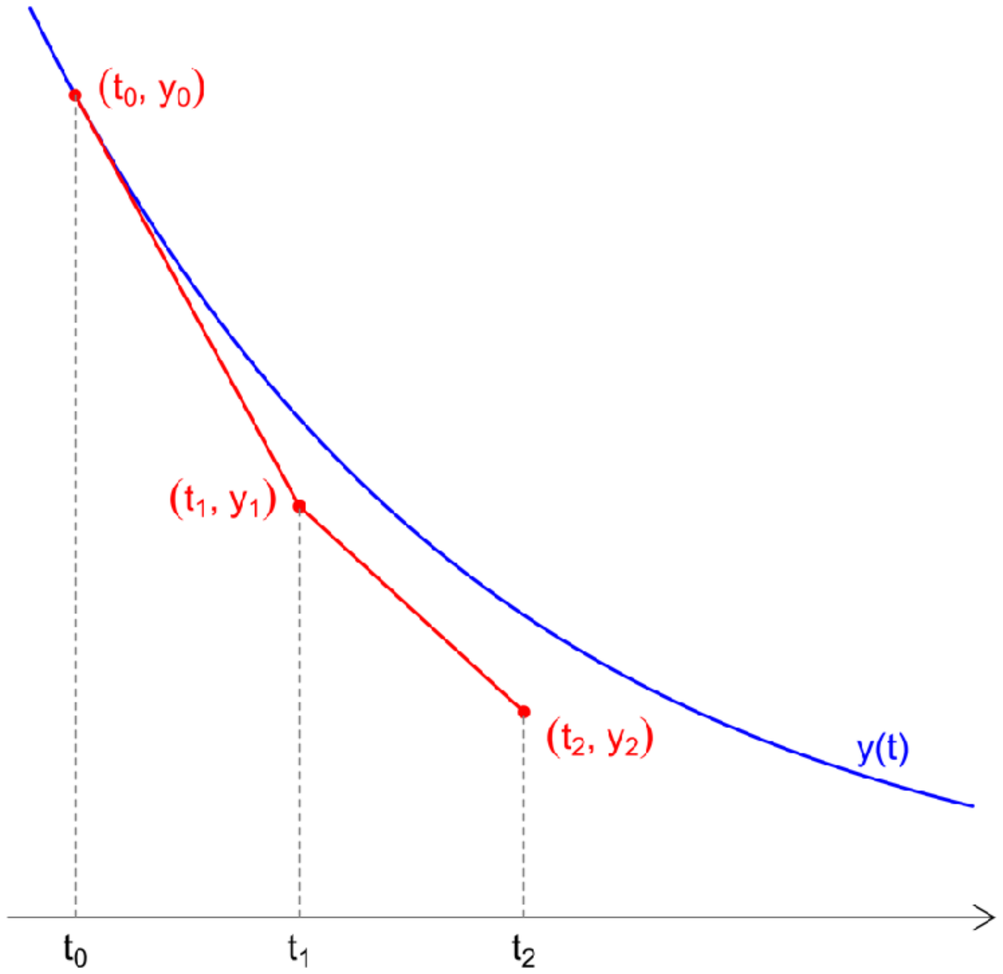
$$y^{n+1} = y^n + \frac{\partial y}{\partial t}\Delta t$$

Order of the solver:

Taylor expansion

$$y^{n+1} = y^n + \frac{\partial y^n}{\partial t}\Delta t + \mathcal{O}(\Delta t^2)$$

→ first order accurate

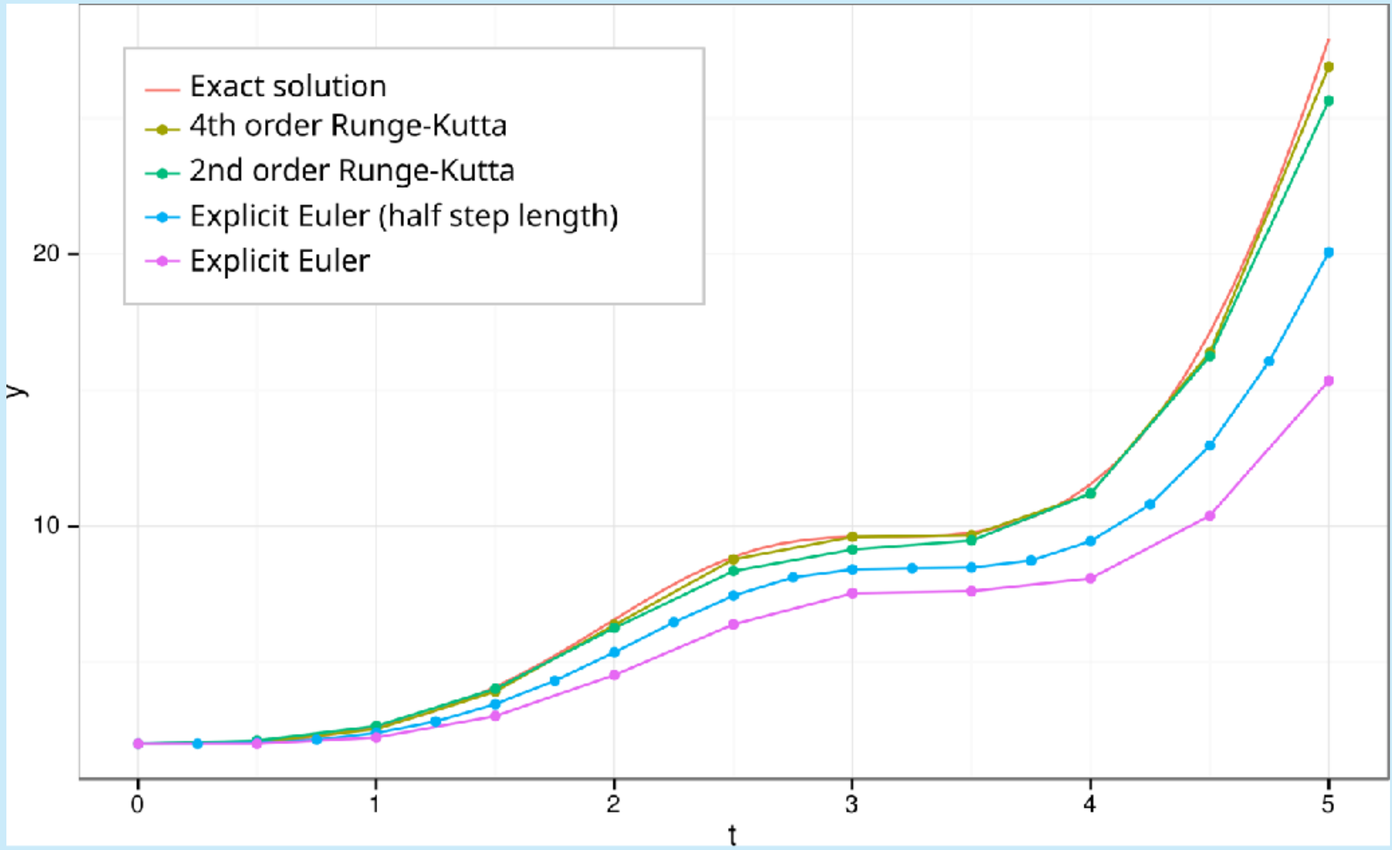


HIGHER ORDER SOLVERS

Runge-Kutta-Method:

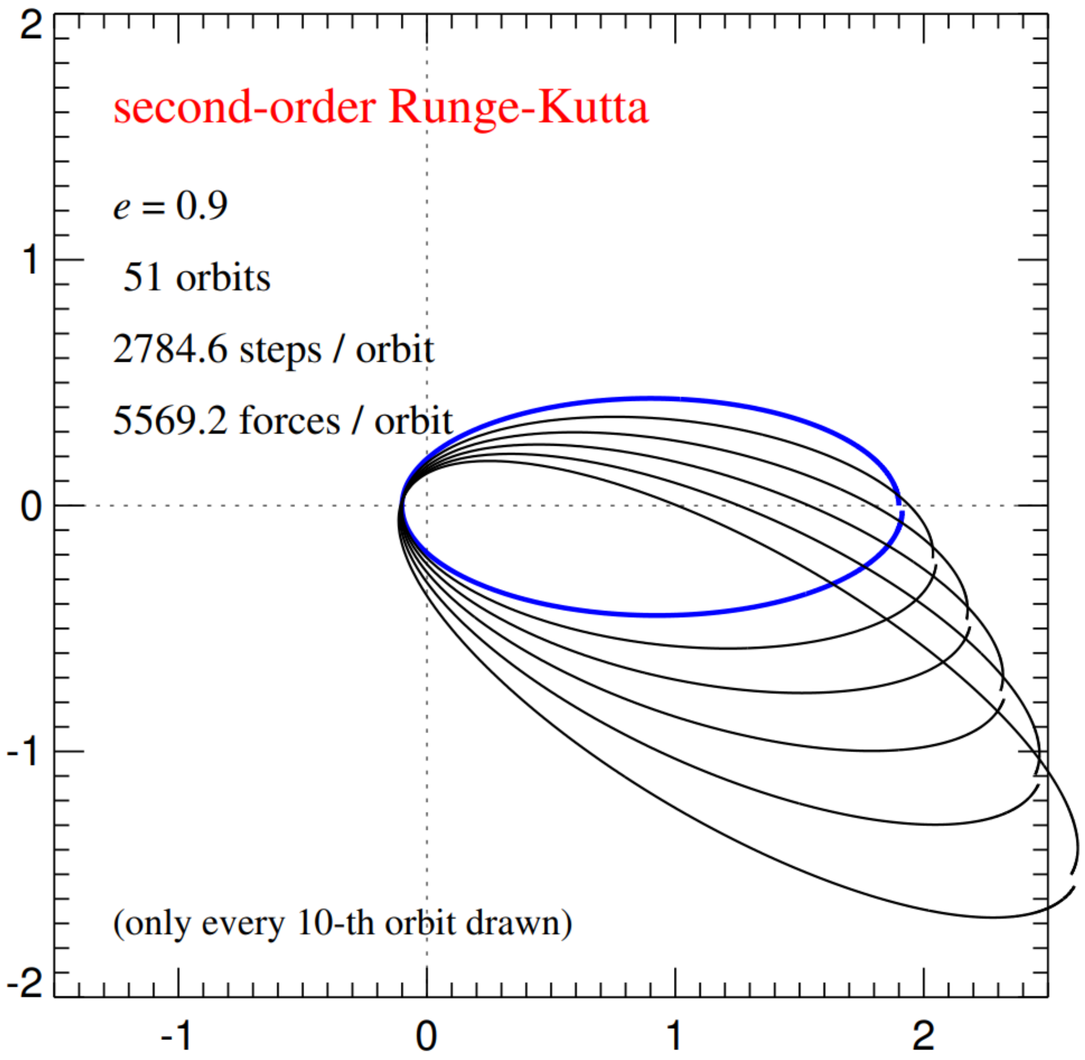
weighted combination of simple derivatives

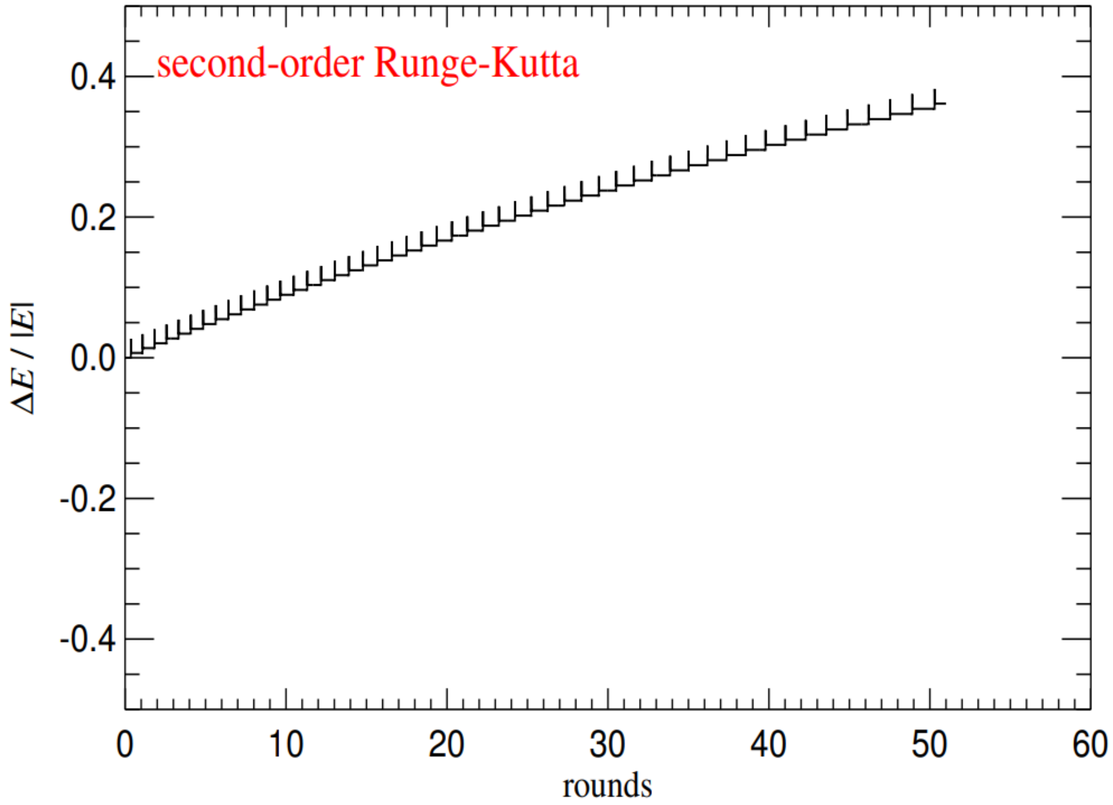
$$y^{n+1} = y^n + h \sum_{i=1}^{s} b_i k_i$$



$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f(t_n + c_2 h, y_n + (a_{21} k_1) h), \\
k_3 &= f(t_n + c_3 h, y_n + (a_{31} k_1 + a_{32} k_2) h), \\
&\vdots \\
k_s &= f(t_n + c_s h, y_n + (a_{s1} k_1 + a_{s2} k_2 + \cdots + a_{s,s-1} k_{s-1}) h).
\end{aligned}$$

SOLUTION OF THE TWO-BODY PROBLEM



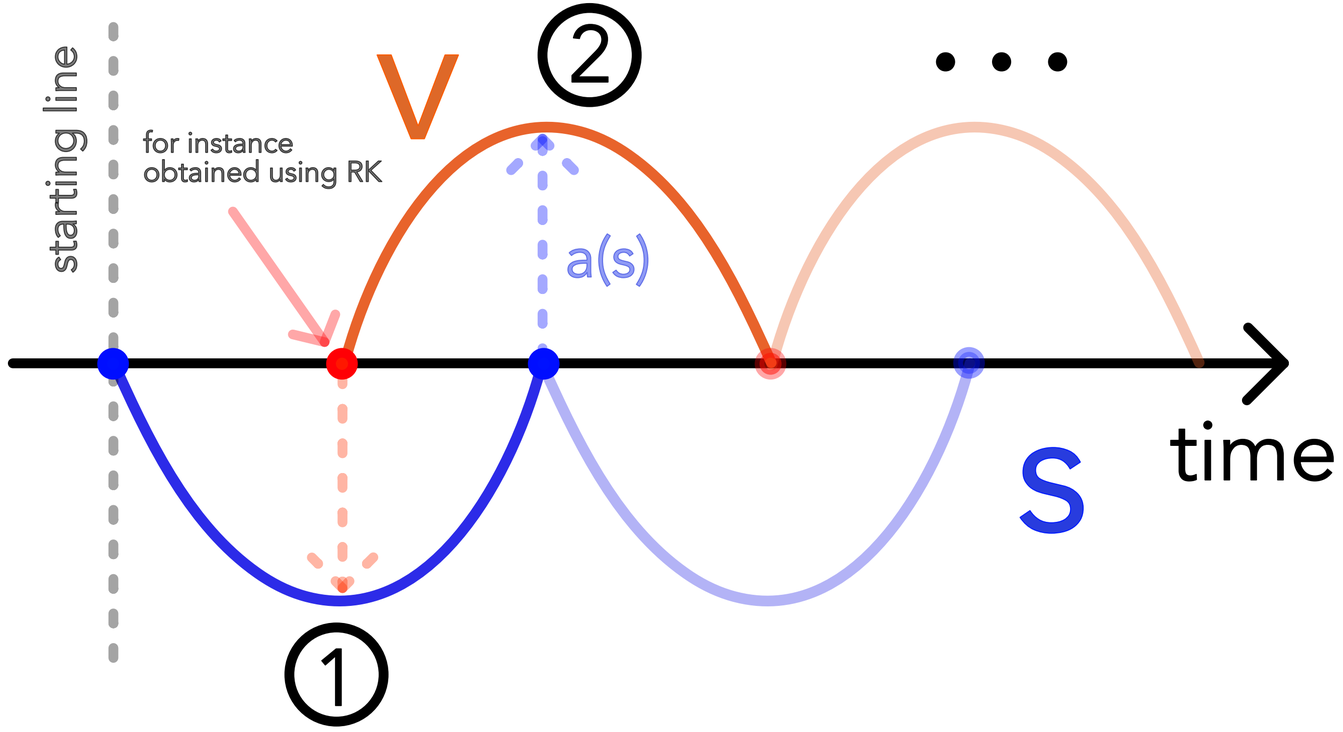


Energy error in each step!

[from the Fundamentals of Simulation Methods lecture notes by Springel et al., version 2022/2023]

SOLUTION OF THE TWO-BODY PROBLEM

Leapfrog



Symplectic integrator with good energy conservation properties; time reversibility; exact angular momentum conservation

1. update location:

$$\underline{s}\left(t+\frac{1}{2}\Delta t\right)=\underline{s}\left(t-\frac{1}{2}\Delta t\right)+\underline{v}(t)\Delta t+\mathcal{O}(\Delta t^3)$$

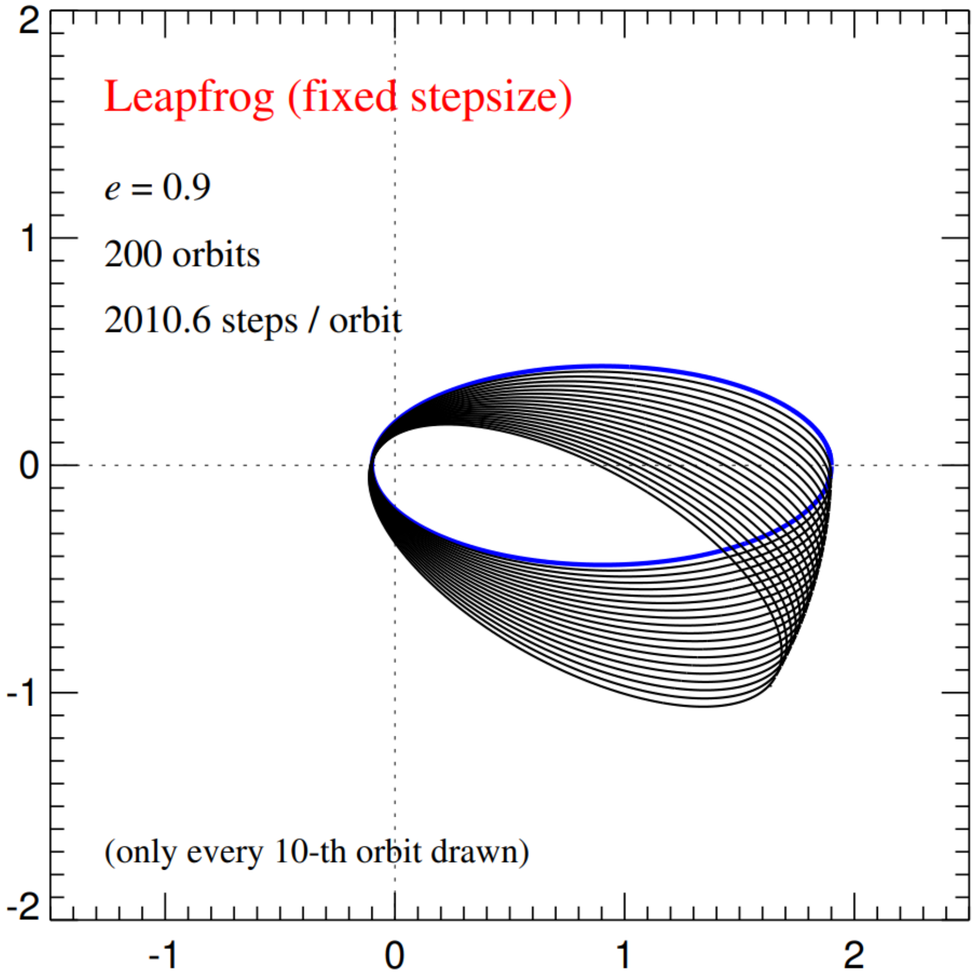
2. update velocity:

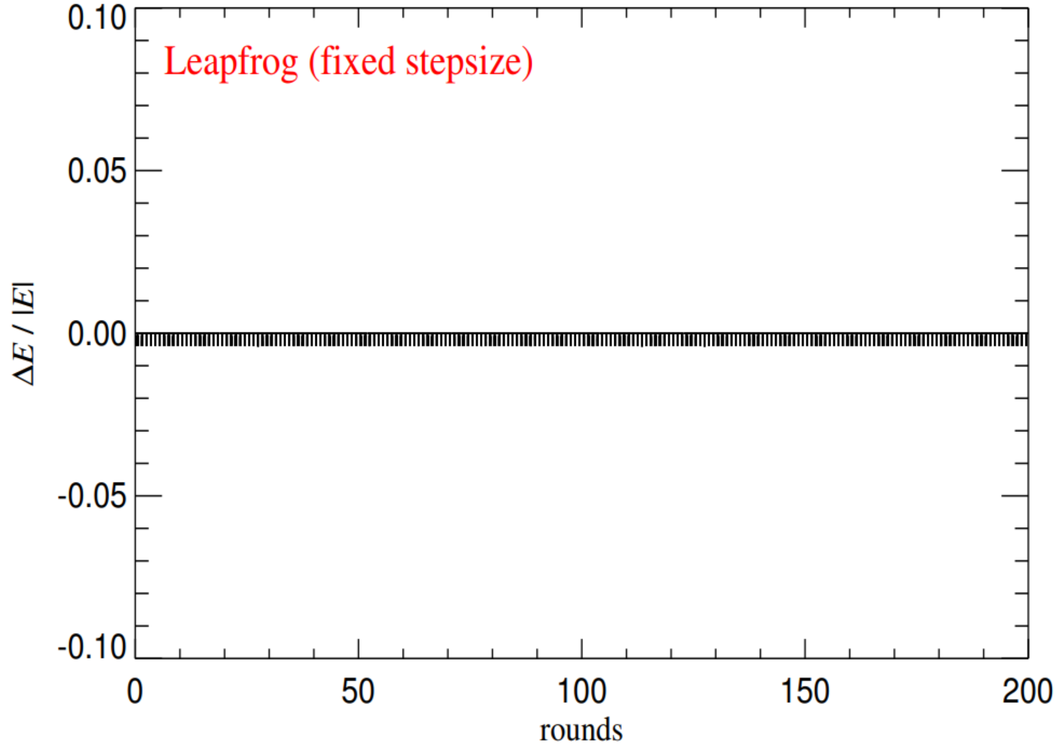
$$\underline{v}(t+\Delta t)=\underline{v}(t)+\underline{a}\left(t+\frac{1}{2}\Delta t\right)\Delta t+\mathcal{O}(\Delta t^3)$$

SOLUTION OF THE TWO-BODY PROBLEM

Exact solution to $H_{leap} = H + H_{err}$,

$$H_{err} \propto \frac{\Delta t^2}{12}\left\{\{H_{kin}, H_{pot}\}, H_{kin} + \frac{1}{2}H_{pot}\right\} + \mathcal{O}(\Delta t^3)$$





[from the "Fundamentals of Simulation Methods" lecture notes by Springel et al., version 2022/2023]

BREAK DOWN OF CLASSICAL METHODS

- Strong turbulence:
  - turbulent flow simulations are limited by Reynolds numbers. Unreasonable small grid required
  - →handle small scales by ML

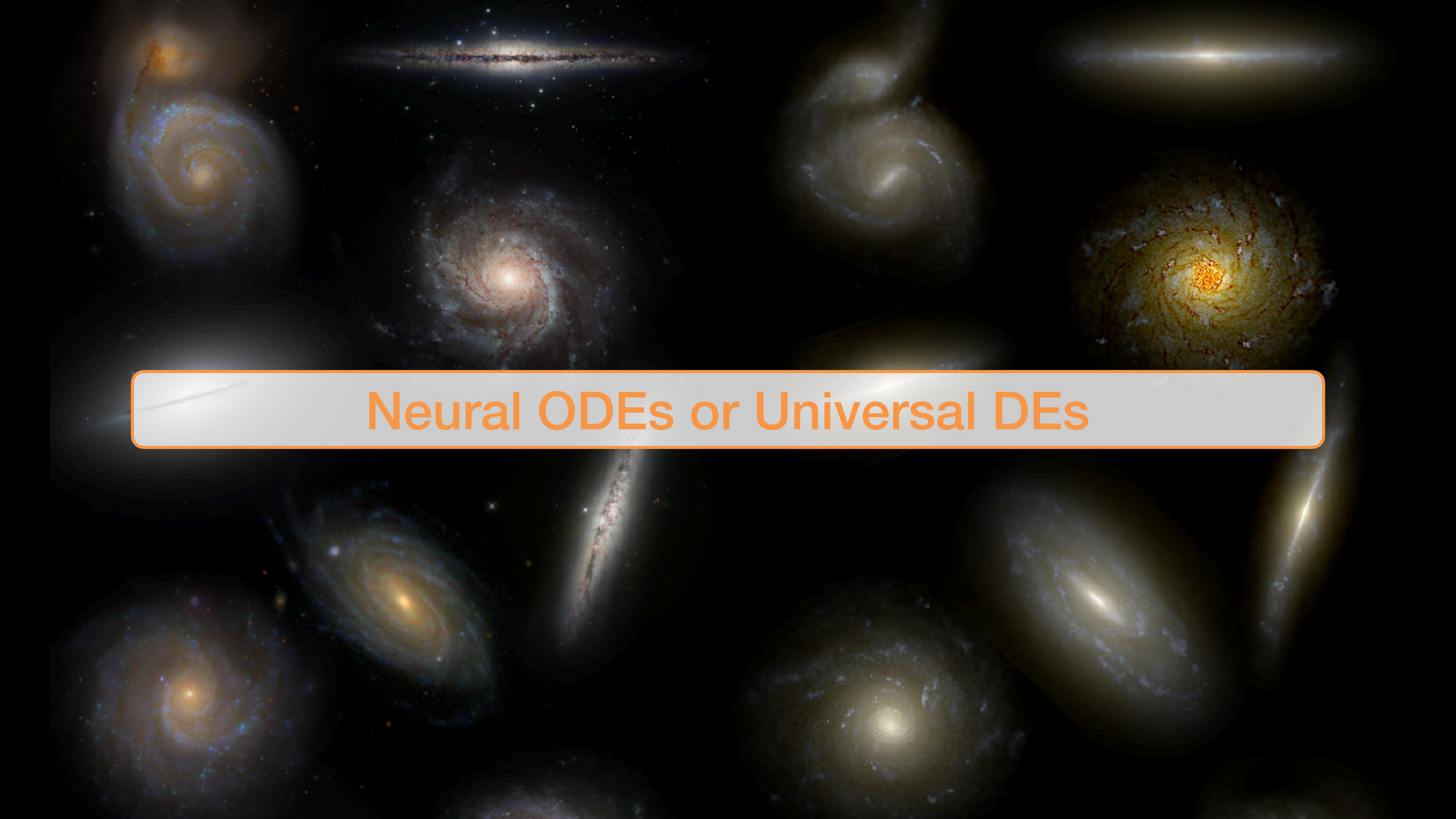
- Problem of different scales
  - Different scales can be very problematic (although there are adaptive methods, mesh free methods, tree algorithms etc.)

- curse of dimensionality for high-dimensional PDEs
  - (e. g. Schrödinger in Physics, Black-Scholes in finance): The computational cost for solving them goes up exponentially with the dimensionality. [Han et al, 2018]

- Need for inverse modelling

- how to blend DEs with vast data sets?

- …


Neural ODEs or Universal DEs

SCIENTIFIC ML — LEARNING THE SOLUTION OF ODES AND PDES

- XDEs are good for:
  - population models
  - motion of the planets
  - structural integrity of a bridge
  - fluid dynamics
  - …

ODEs are kind of easy — only derivatives with respect to one variable

PDEs are more complicated — derivatives with respect to many variables and differential equations are local while solutions exhibit non-local properties

- Traditional solution: discretisation (in time and space) and iterative solution

NEURAL ODES, OPERATOR LEARNING AND PINNS

Neural ODE: $\frac{df}{dt} = h_\theta(x_0, t, p)$

(neural net = right hand side of diff eq.

solution: integrate entire neural net.)

Neural Operator: $G_\theta : X \to Y \quad u \mapsto G_\theta(u)$ with X, Y function spaces (infinite dimensional)

(neural net approximates the operator
i.e. the map between function space)

PINN: $f(x,t) = h_\theta(x,t,p)$ with $\frac{df}{dt} = \frac{dh_\theta}{dt}, \quad \frac{df}{dx} = \frac{dh_\theta}{dx}$ need to fulfil the diff eq.

(solution is given by neural net,
autodiff and diff eq. are used in loss)

An introduction to neural ODEs in scientific machine learning

Patrick Kidger

Cradle.bio

Slides taken from Patrick Kidger

01

From traditional parameterised modelling to neural ODEs

Slides taken from Patrick Kidger

ODEs are really really (really) good models.

🐰(0) = 2 $\quad d🐰/_{dt} =$ 🐰 − 🐰🦊

🦊(0) = 1 $\quad d🦊/_{dt} =$ − 🦊 + 🐰🦊

Good for:

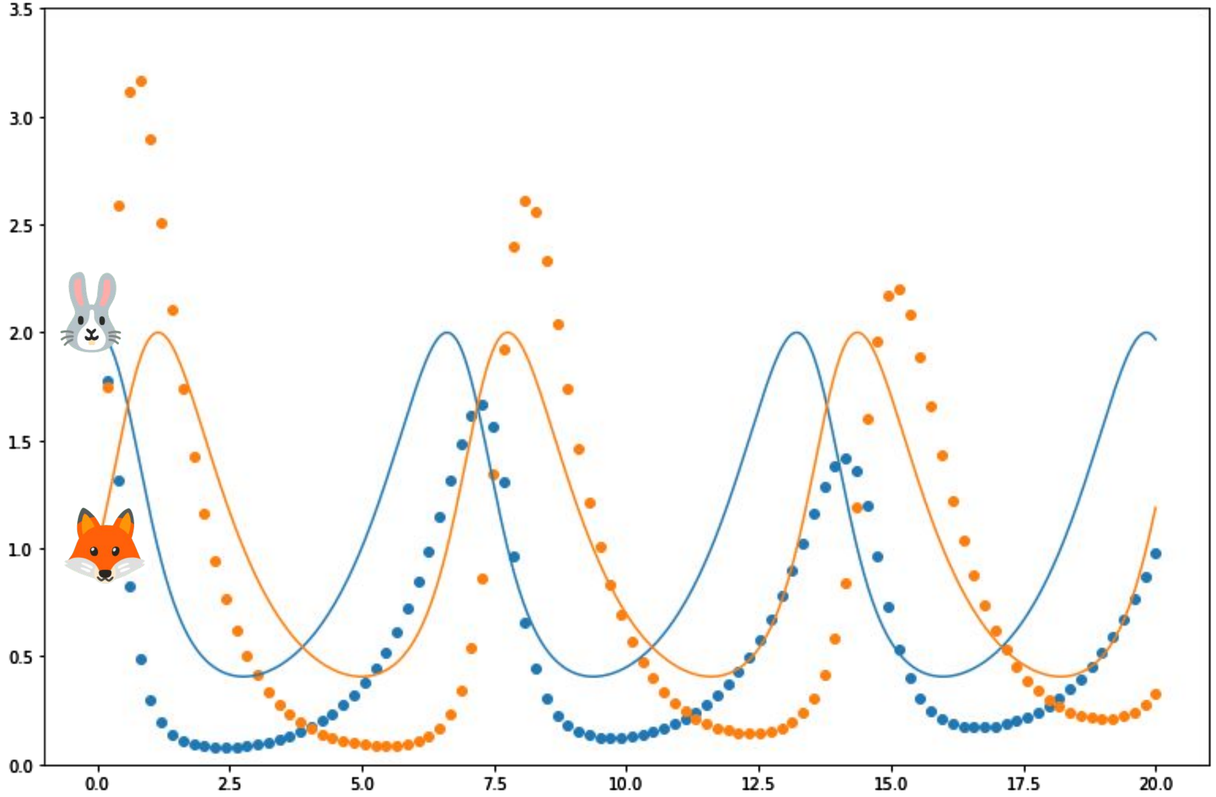
- Population modelling;
- Motion of the planets;
- Structural integrity of a bridge;
- Fluid dynamics;
- …

Simple idea: just write down how a value:

- Starts
- And how it changes over time.

Slides taken from Patrick Kidger

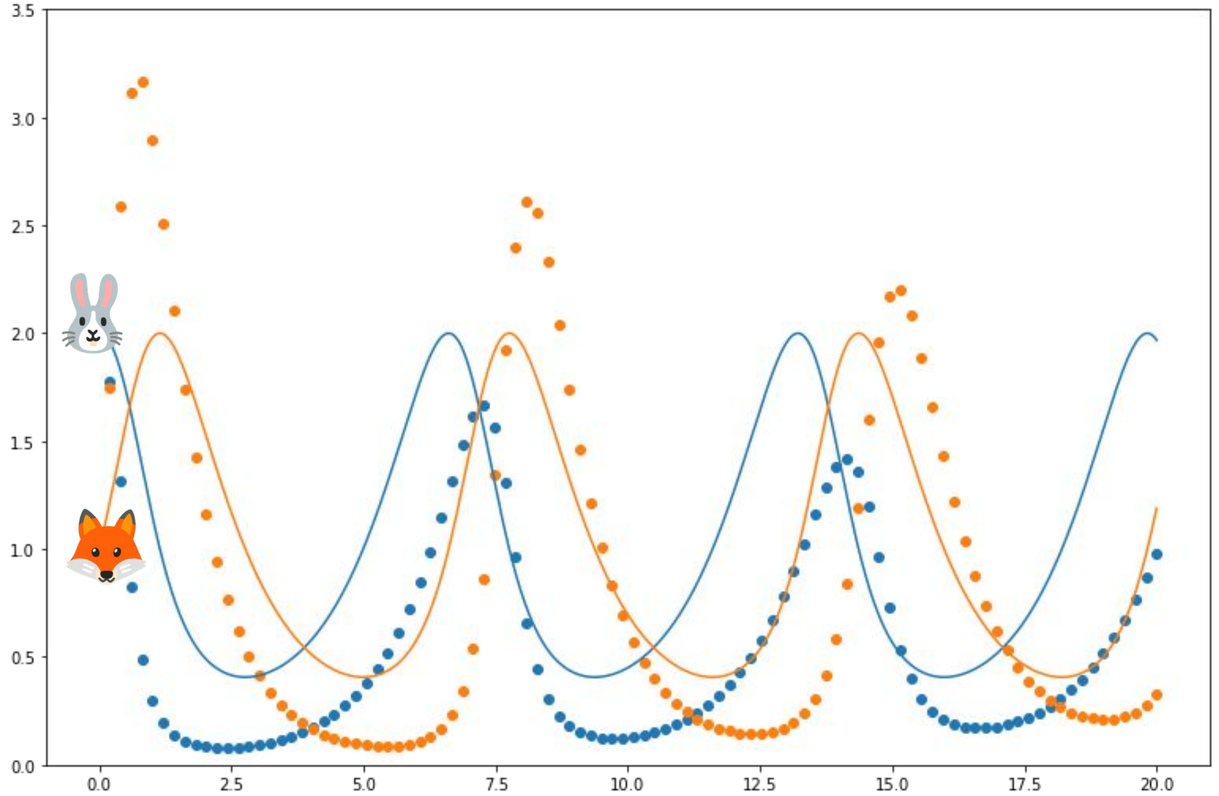
Lotka—Volterra (Predator–Prey)



Parameterised ODEs

🐰(0) = 2 $\quad \frac{d🐰}{dt} =$ 🐰 − 🐰🦊

🦊(0) = 1 $\quad \frac{d🦊}{dt} =$ − 🦊 + 🐰🦊



Slides taken from Patrick Kidger

Parameterised ODEs

$🐰(0) = 2 \qquad d🐰/dt = \alpha🐰 - \beta🐰🦊$

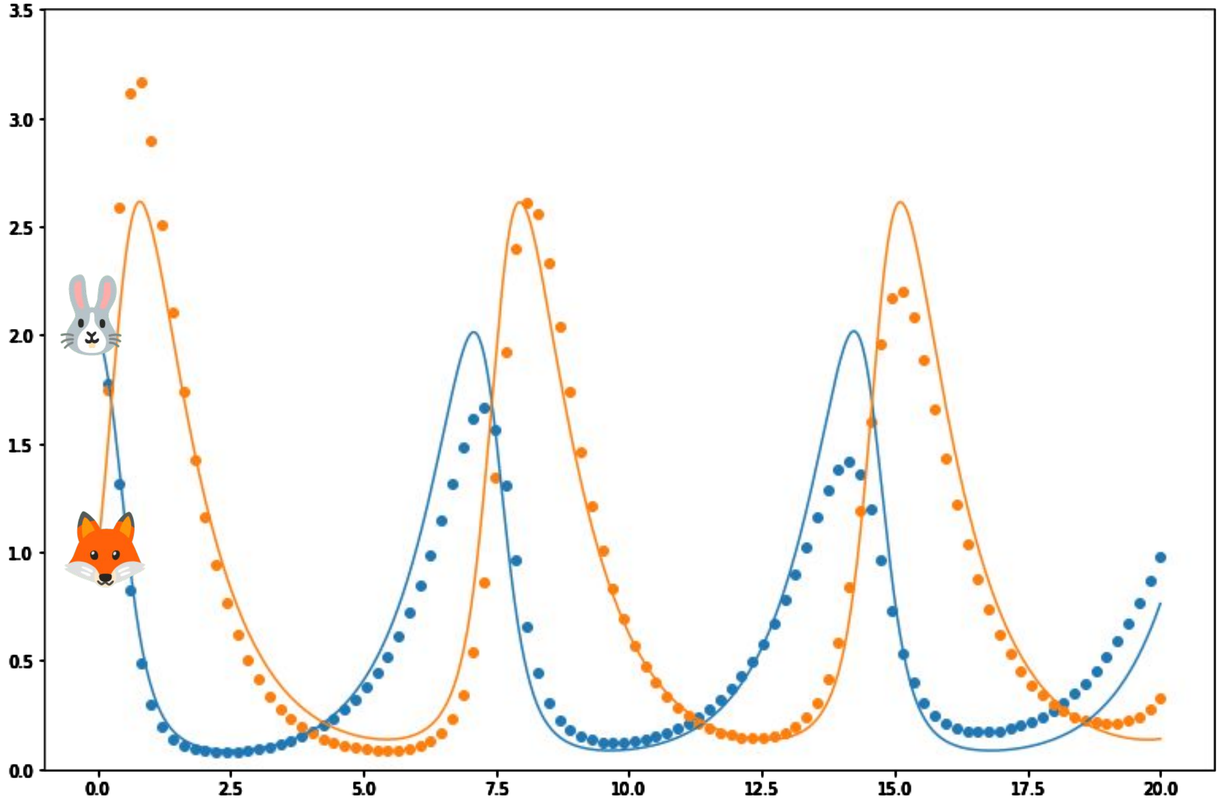
$🦊(0) = 1 \qquad d🦊/dt = -\gamma🦊 + \delta🐰🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs

$🐰(0) = 2 \qquad d🐰/dt = \alpha 🐰 - \beta 🐰 🦊 + NN_\theta(🐰, 🦊)$

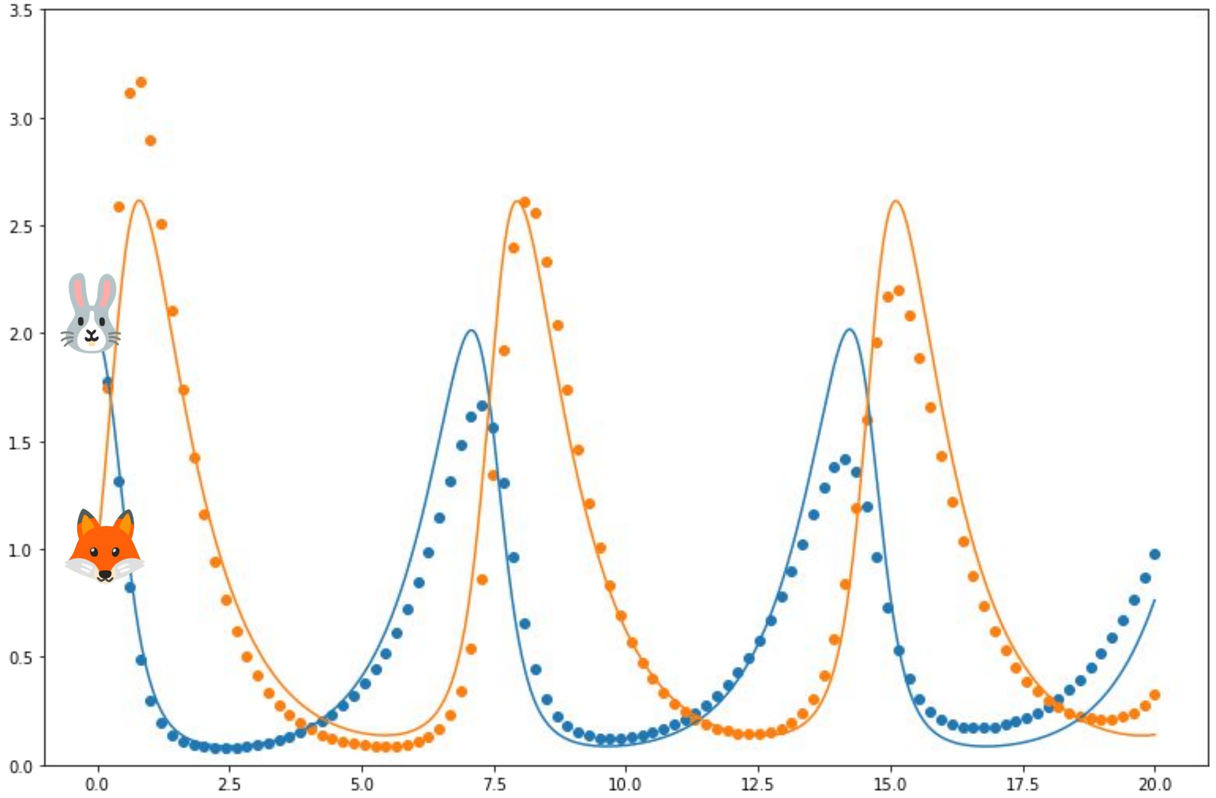
$🦊(0) = 1 \qquad d🦊/dt = -\gamma 🦊 + \delta 🐰 🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs

$🐰(0) = 2$ $\quad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🦊 + NN_\theta(🐰, 🦊)$

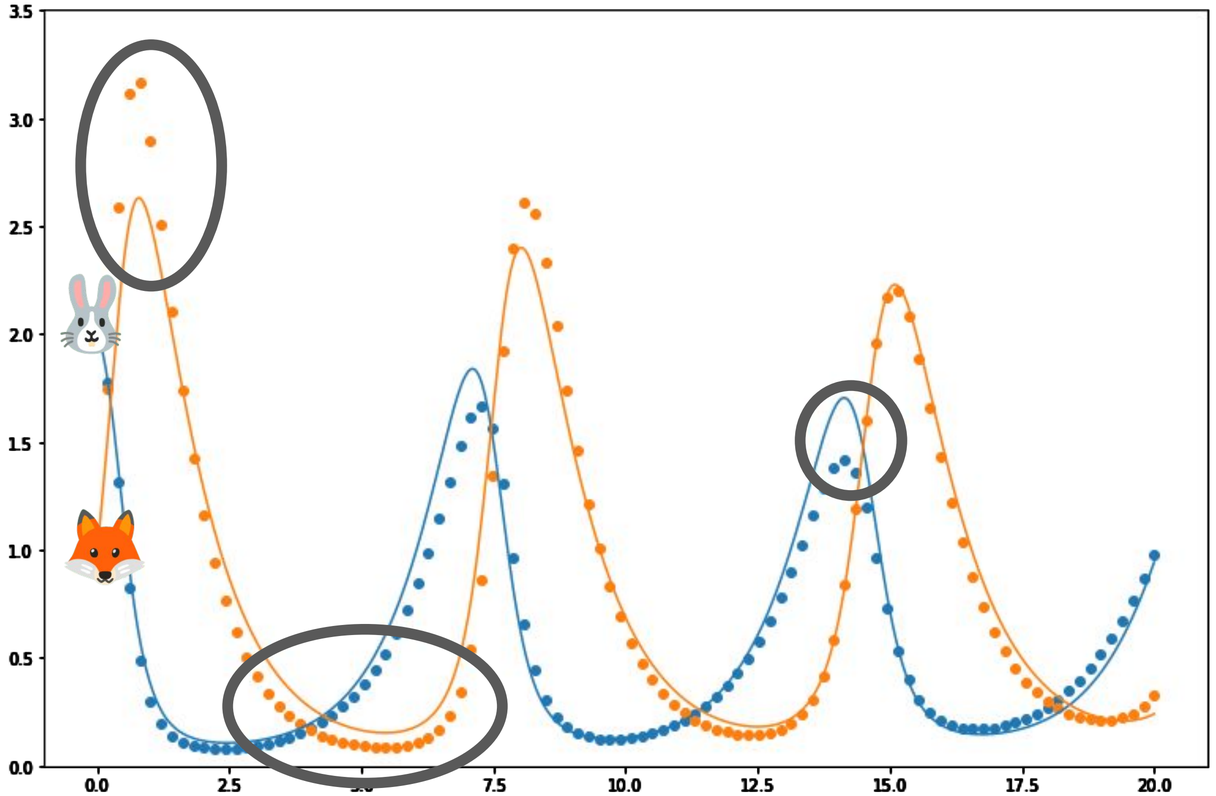
$🦊(0) = 1$ $\quad d🦊/_{dt} = -\gamma 🦊 + \delta 🐰 🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs: version 2

$🐰(0) = 2 \qquad d🐰/dt = \alpha 🐰 - \beta 🐰🦊$

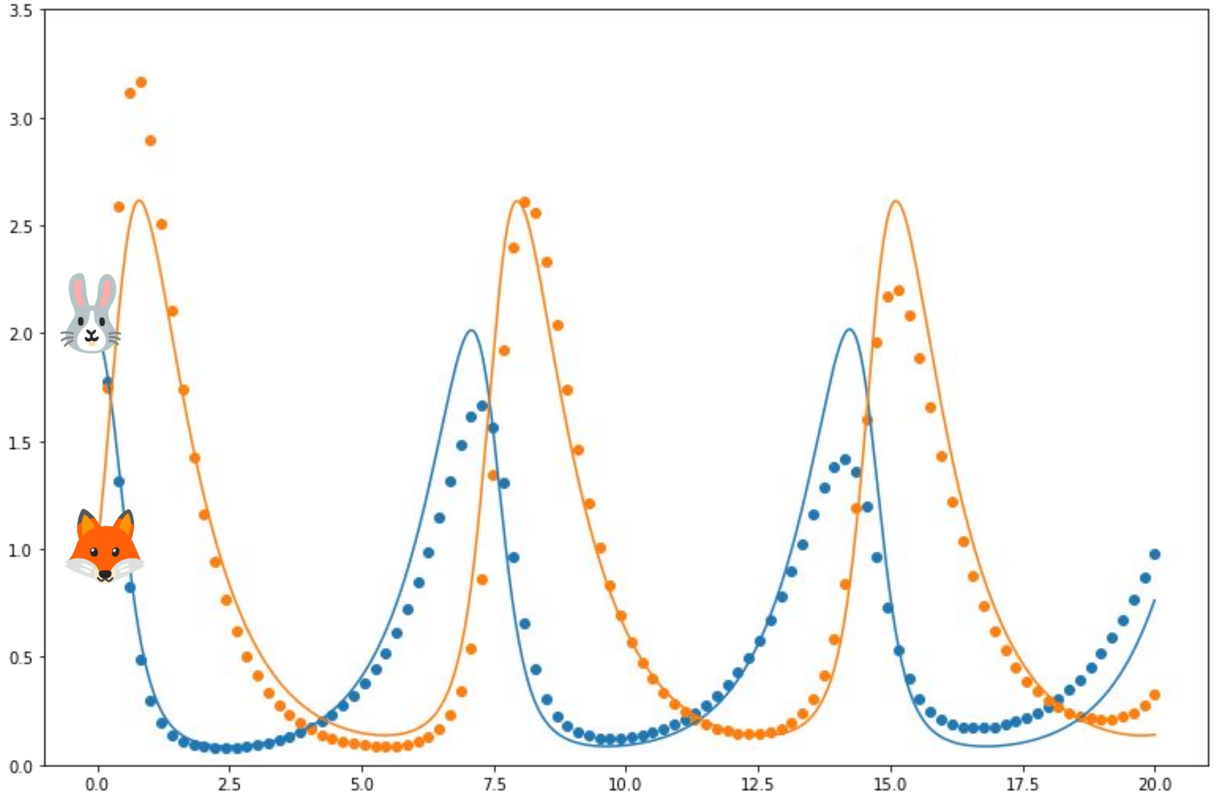
$🦊(0) = 1 \qquad d🦊/dt = -\gamma 🦊 + \delta 🐰🦊$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Neural ODEs: version 2

$$🐰(0) = 2 \qquad d🐰/_{dt} = \alpha 🐰 - \beta 🐰 🦊$$

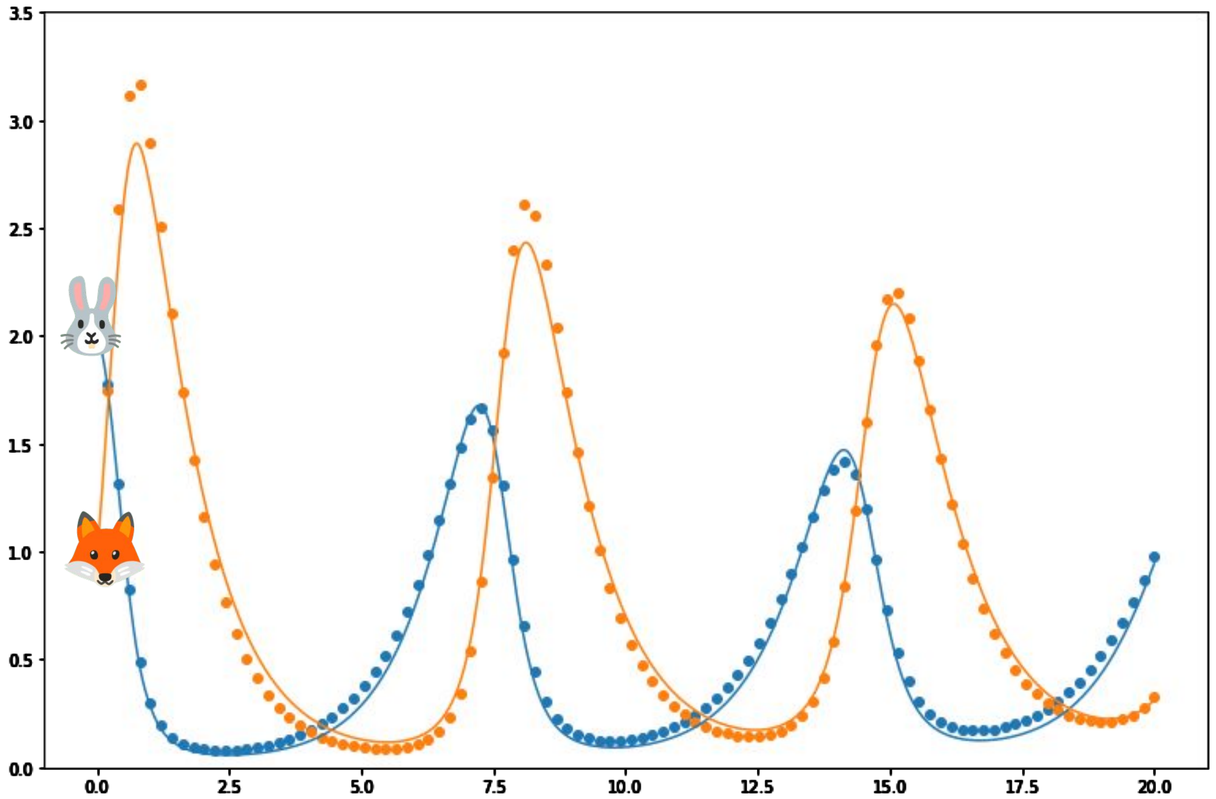
$$🦊(0) = 1 \qquad d🦊/_{dt} = -\gamma 🦊 + \delta 🐰 (🦊 + NN_\theta(🐰, 🦊))$$

- Initialise
  $\alpha=1$
  $\beta=1$
  $\gamma=1$
  $\delta=1$
  $\theta \sim N(0, \sigma^2)$

- Set up a loss function:

  $(\text{data} - \text{model})^2$

- Backpropagate (through the ODE solver)
- Train via gradient descent



Slides taken from Patrick Kidger

Summary of part 1:

- Neural ODEs are a great model for unknown physics / unknown biology /…
    - When you don’t fully understand the physics.
    - When the physics is too expensive to fully simulate. (E.g. in weather.)
    - When a traditional parameterised model is difficult to optimise. (E.g. in biology.)
    - *Differentiable simulation, surrogate models*.

- Keep the physical model if you think it’s good.
    - Put the NN on the bit you think isn’t well-modelled.
    - Here the NN is tiny: 4 neurons wide, 1 layer deep.
    - Physical parameters trained first; them kept fixed whilst training NN.

- Neural ODE trained in the same way as the mechanistic ODE.
    - In both cases: a parameterised vector field trained with backprop+SGD.
    - Easy to implement: software for NN, ODEs, SGD, …

- “Neural ODE” ≈ “hybrid ODE” ≈ “universal ODE” ≈ “parameterised ODE”

Slides taken from Patrick Kidger

Symbolic regression

Let's try to interpret NN_θ.

$$\mathrm{d}🐰/\mathrm{dt} = 1.1🐰 - 1.3🐰🦊$$
$$\mathrm{d}🦊/\mathrm{dt} = -0.98🦊 + 1.6🐰(🦊 + \mathrm{NN}_\theta(🐰, 🦊))$$

Generate a dataset of ~10k samples:

$$\{(🐰_1, 🦊_1, y_1), \ldots, (🐰_n, 🦊_n, y_n)\} \text{ with } y_i = \mathrm{NN}_\theta(🐰_i, 🦊_i).$$

(Sample $🐰_i$ and $🦊_i$ in your favourite way, e.g. points from typical ODE trajectories.)

Symbolically regress y_i against $(🐰_i, 🦊_i)$:

$$\{\text{PySR, Lasso, ...}\} \Rightarrow y = 🦊^{0.9} - 🦊 + 0.5🦊^2$$

Retrain all parameters differentiably:

$$\mathrm{d}🐰/\mathrm{dt} = 0.9🐰 - 1.1🐰🦊$$
$$\mathrm{d}🦊/\mathrm{dt} = -1.2🦊 + 2.1🐰🦊^{0.95}$$

vs SINDy?

SINDy performs Lasso of d(🐰,🦊)/dt against (🐰, 🦊). Thus it requires an estimate of the derivative. That's not needed here!

Also, we're not constrained to just Lasso: other symbolic regression (e.g. PySR i.e. regularised evolution) can be used.

Slides taken from Patrick Kidger

Conditions + Learnt Initialisations

Recall our general ODE formulation.

$$(🐰, 🦊)(0) = g \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊)$$

Both f and g may depend on additional data, e.g., the latitude L that the 🐰 and 🦊 live at.

$$(🐰, 🦊)(0) = g(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

Recall that an ODE is determined by:

- An initial condition;
- A vector field.

So far we've tried learning the vector field.

$$(🐰, 🦊)(0) = g_\theta(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

We can also learn the initial condition.

Slides taken from Patrick Kidger

Generalised observables

We might observe things that aren't part of the evolving state of our ODE, but which still somehow give us information.

For example, we might observe the amount of rabbit... output 💩.

Now fit this to the data as well.

Even if we don't care about 💩, this can help us fit the 🐰, 🦊 parts better.

Important extreme case: "hidden state".

Slides taken from Patrick Kidger

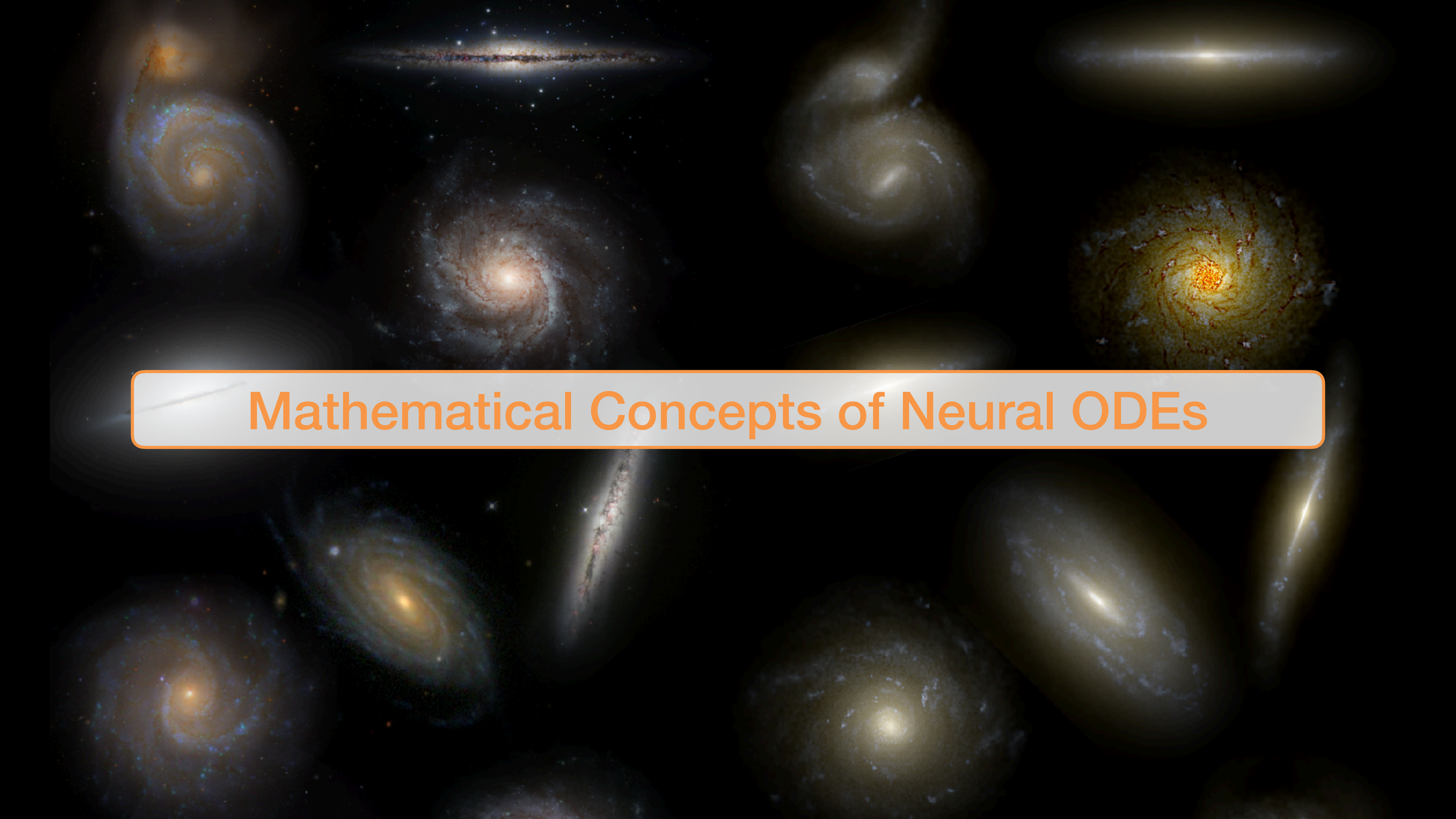
$$(🐰, 🦊)(0) = g_\theta(L) \qquad d(🐰, 🦊)/dt = f_\theta(🐰, 🦊, L)$$

$$💩 = h_\theta(🐰, L)$$

$$\text{loss} = \|🐰 - 🐰_{\text{data}}\|^2 + \|🦊 - 🦊_{\text{data}}\|^2 + \|💩 - 💩_{\text{data}}\|^2$$

$$y(0) = g_\theta(c) \qquad dy/dt = f_\theta(y, c) \qquad (🐰, 🦊, 💩) = h_\theta(y, c)$$

This is actually the the most typical formulation of a neural ODE!


Mathematical Concepts of Neural ODEs

NEURAL ODES

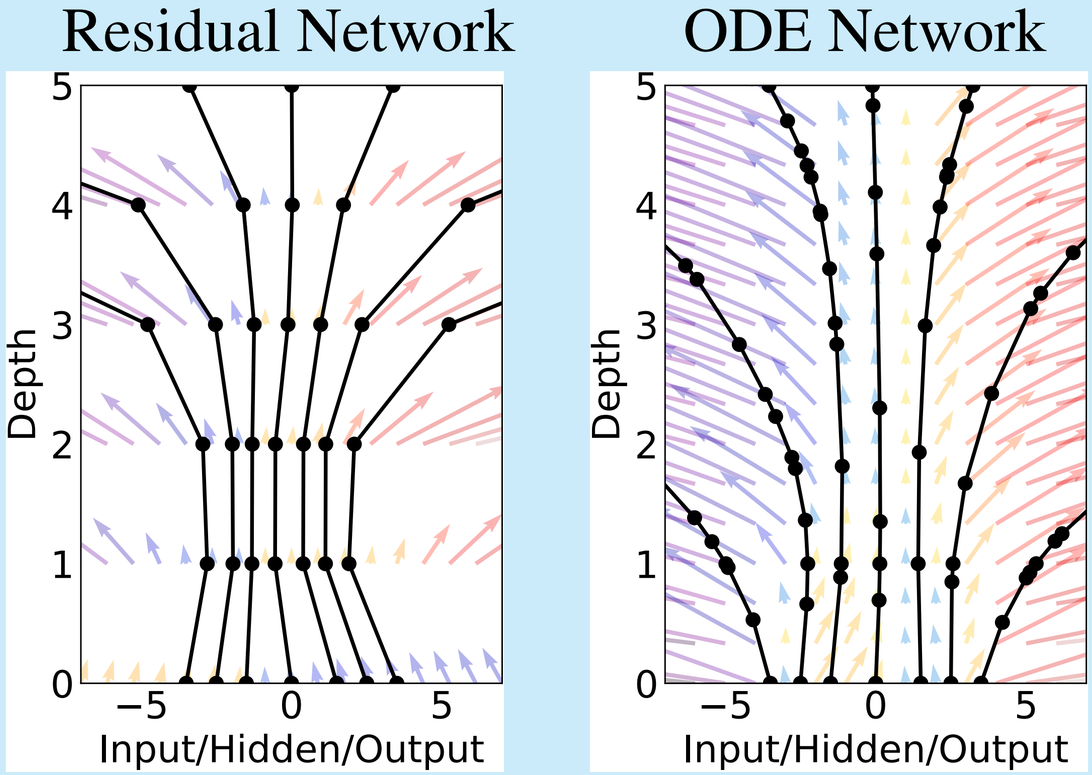
- Traditional solution: discretisation (in time and space) and iterative solution

Euler discretization

$$h_{t+1} = h_t + f_t(h_t, \theta_t)$$

$$h(t + \Delta t) = h(t) + \Delta t \cdot f(t, h(t), \theta)$$

$$\frac{h(t + \Delta t) - h(t)}{\Delta t} = f(t, h(t), \theta)$$



Chen+2019

NEURAL ODES

- Traditional solution: discretisation (in time and space) and iterative solution

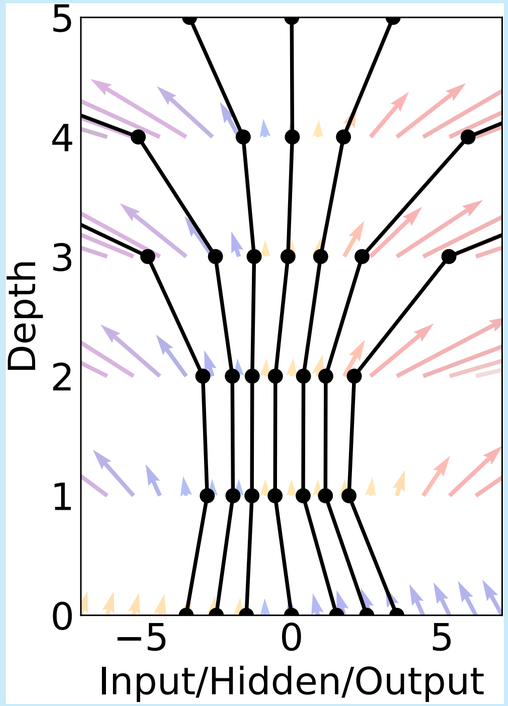
Euler discretization

$$h_{t+1} = h_t + f_t(h_t, \theta_t)$$

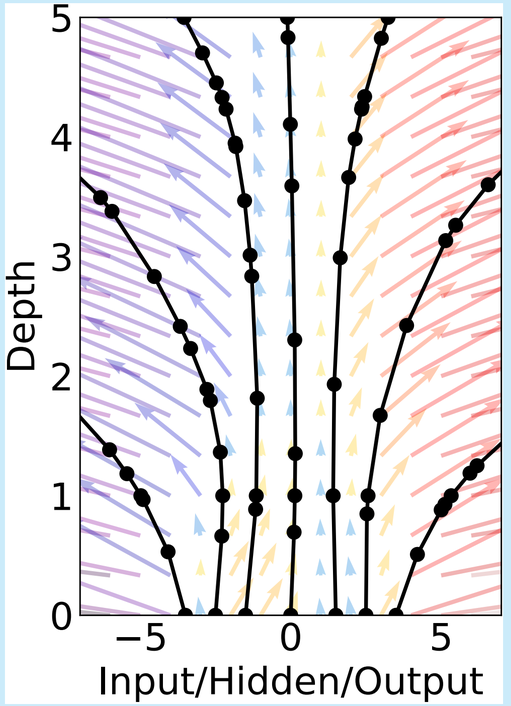
$$h(t + \Delta t) = h(t) + \Delta t \cdot f(t, h(t), \theta)$$

$$\frac{h(t + \Delta t) - h(t)}{\Delta t} = f(t, h(t), \theta)$$

Residual Network



ODE Network



Chen+2019

NEURAL ODES

What do we need to train a NeuralODE? → Reverse-mode automatic differentiation of ODE solutions

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L\left(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)\right)$$

Problem is to find $\frac{\mathrm{dL}(\mathrm{z}(\mathrm{t}_1))}{\mathrm{d}\theta}$ efficiently!

Don’t worry about the details… diffrax (Patrick Kidger), torchdiffeq (Ricky Chen), DiffEqFlux.jl got you convert!

NEURAL ODES TRAINING

In general two options for NODE training:

1. directly use autodiff for $\frac{dL(z(t_1))}{d\theta}$ and back propagate through the solver.
aka „discretize then optimise“

advantage: faster, more accurate gradient; disadvantage: more memory consumption

2. solve $\frac{dL(z(t_1))}{d\theta}$ analytically.

Doing so will result in a backwards-in-time ODE (adjoint equation) that must be solved numerically
aka „optimise then discretise“

advantage: less memory consumption; disadvantage: more computationally expensive, only approximate gradients

ADJOINT EQUATION: PROBLEM STATEMENT

$$\text{Find} \underset{\theta}{\operatorname{argmin}} \; L(z(t_1)) \tag{PM}$$

$$\text{subject to}$$

$$F(\dot{z}(t), z(t), \theta, t) = \dot{z}(t) - f(z(t), \theta, t) = 0 \tag{1}$$

$$z(t_0) = z_{t_0} \tag{2}$$

$$t_0 < t_1$$

- f is our neural network with parameters θ
- $z(t_0)$ is our input, $z(t_1)$ is the output, $z(t)$ is the state reached from $z(t_0)$ at time $t \in [t_0, t_1]$
- $\dot{z}(t) = \frac{\mathrm{d}z(t)}{\mathrm{d}t}$ is the time derivative of our state $z(t)$.
- L is our loss and its a function of the output $z(t_1)$.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION: PROBLEM STATEMENT

$$\text{Find} \underset{\theta}{\operatorname{argmin}} \; L(z(t_1)) \tag{PM}$$

$$\text{subject to}$$

$$F(\dot{z}(t), z(t), \theta, t) = \dot{z}(t) - f(z(t), \theta, t) = 0 \tag{1}$$

$$z(t_0) = z_{t_0} \tag{2}$$

$$t_0 < t_1$$

- f is our neural network with parameters θ
- $z(t_0)$ is our input, $z(t_1)$ is the output, $z(t)$ is the state reached from $z(t_0)$ at time $t \in [t_0, t_1]$
- $\dot{z}(t) = \frac{\mathrm{d}z(t)}{\mathrm{d}t}$ is the time derivative of our state $z(t)$.
- L is our loss and its a function of the output $z(t_1)$.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let’s rewrite our initial objective with a Lagrange multiplier $\lambda(t)$

$$\psi = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t) F(\dot{z(t)}, z(t), \theta, t) dt$$

since

$$F(\dot{z(t)}, z(t), \theta, t) = \dot{z(t)} - f(z(t), \theta, t) = 0$$

the integral vanishes and

$$\frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let's rewrite our initial objective with a Lagrange multiplier $\lambda(t)$

$$\psi = L(z(t_1)) - \int_{t_0}^{t_1} \lambda(t) F(\dot{z(t)}, z(t), \theta, t) dt$$

since

$$F(\dot{z(t)}, z(t), \theta, t) = \dot{z(t)} - f(z(t), \theta, t) = 0$$

the integral vanishes and

$$\frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$$

Let's choose $\lambda(t)$ to eliminate hard to compute derivatives, such as the Jacobian $\frac{\mathrm{dz}(t_1)}{\mathrm{d}\theta}$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1}\left(\dot{\lambda(t)} + \lambda(t)\frac{\partial f}{\partial z}\right)\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}dt + \int_{t_0}^{t_1}\lambda(t)\frac{\partial f}{\partial \theta}dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Derivative	Description	Ease
$\frac{\partial L}{\partial z(t_1)}$	Gradient of loss with respect to output.	Easy
$\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta}$	Jacobian of output with respect to params.	Not easy
$\dot{\lambda(t)}$	Derivative of lambda, a vector, with respect to time.	Easy
$\lambda(t)\frac{\partial f}{\partial z}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial z}$.	Easy
$\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}$	Jacobian of arbitrary layer with respect to params.	Not easy
$\lambda(t)\frac{\partial f}{\partial \theta}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial \theta}$.	Easy

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Derivative	Description	Ease
$\frac{\partial L}{\partial z(t_1)}$	Gradient of loss with respect to output.	Easy
$\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta}$	Jacobian of output with respect to params.	Not easy
$\dot{\lambda(t)}$	Derivative of lambda, a vector, with respect to time.	Easy
$\lambda(t)\frac{\partial f}{\partial z}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial z}$.	Easy
$\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}$	Jacobian of arbitrary layer with respect to params.	Not easy
$\lambda(t)\frac{\partial f}{\partial \theta}$	vector-Jacobian Product. Can compute with reverse mode autodiff without explicitly constructing Jacobian $\frac{\partial f}{\partial \theta}$.	Easy

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1}\left(\dot{\lambda(t)} + \lambda(t)\frac{\partial f}{\partial z}\right)\frac{\mathrm{d}z(t)}{\mathrm{d}\theta}dt + \int_{t_0}^{t_1}\lambda(t)\frac{\partial f}{\partial \theta}dt$$

Can we get rid of $\frac{\mathrm{dz(t)}}{\mathrm{d}\theta}$ by a smart choice of $\lambda(t)$?

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

Let the math magic happen…
(using integration by parts, chain rule and some recursive plugging in of equations…)
if you want the details, see the blog post.

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Let's define the backward ODE as:

$$\dot{\lambda}(t) = -\lambda(t)\frac{\partial f}{\partial z} \quad \text{s.t.} \quad \lambda(t_1) = \frac{\partial L}{\partial z(t_1)}$$

giving

$$\lambda(t_0) = \lambda(t_1) - \int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial z} dt$$

$\lambda(t)$ is called the adjoint.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Which simplifies our objective a lot!

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

ADJOINT EQUATION VIA LAGRANGE MULTIPLIER

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = \underbrace{\left[\frac{\partial L}{\partial z(t_1)} - \lambda(t_1)\right]}_{=0} \frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \underbrace{\left(\dot{\lambda}(t) + \lambda(t)\frac{\partial f}{\partial z}\right)}_{=0} \frac{\mathrm{d}z(t)}{\mathrm{d}\theta} dt + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

Which simplifies our objective a lot!

$$\frac{\mathrm{d}L}{\mathrm{d}\theta} = -\int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial \theta} dt$$

(note, we switch the integration limits)

With this equation we trade memory for computation! Memory consumption is O(1) with respect to the number of layers.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

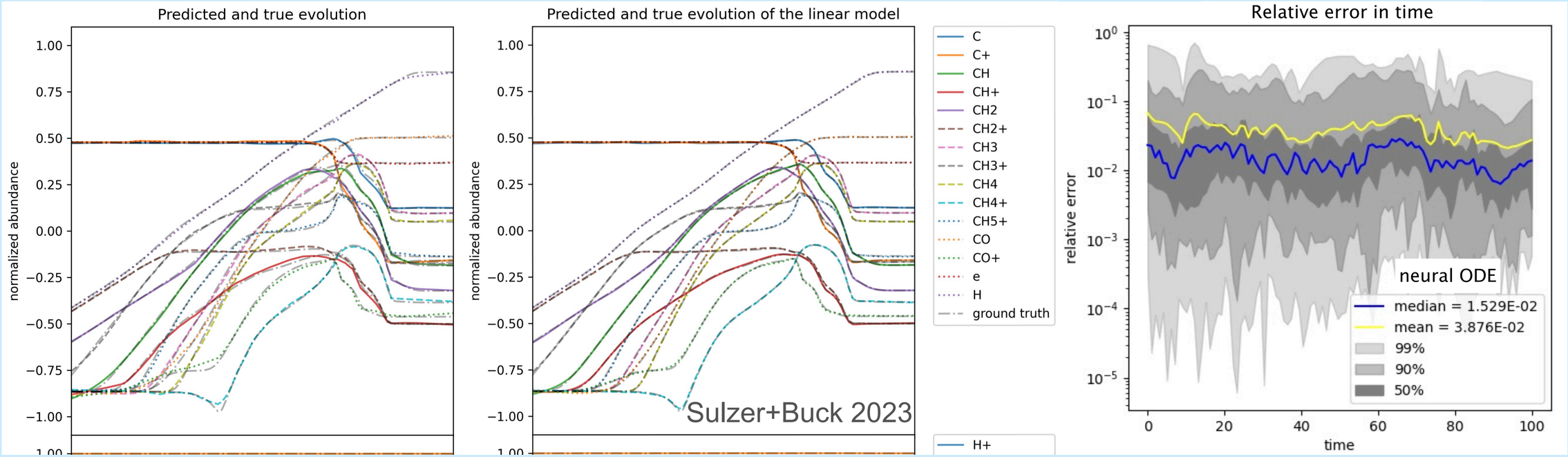
TRAINING NODE WITH ADJOINT METHOD STEP BY STEP RECIPE

1. Forward pass: Solve the ODE (1)-(2) from time t_0 to t_1 and get the output $z(t_1)$.
2. Loss calculation: Calculate $L(z(t_1))$.
3. Backward pass: Solve ODEs (11) and (12) from reverse time t_1 to t_0 to get the gradient of the loss $\frac{\mathrm{d}L(z(t_1))}{\mathrm{d}\theta}$.
4. Use the gradient to update the network parameters θ.

blog post by https://vaipatel.com/posts/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/

NEURAL ODES IN ASTROPHYSICS

- Neural Astrophysical Wind Models (Nguyen 2023)
- Neural ODEs as a discovery tool to characterize the structure of the hot galactic wind of M82 (Nguyen+2023)
- Speeding up astrochemical reaction networks with autoencoders and neural ODEs (Sulzer+Buck 2023)



SUMMARY & CONCLUSION

Main take away:

scientific motivated inductive bias helps to be more robust, more data efficient and better interpretable

My personal message:

Write better code!
Share more data!
Build more open-source software!

This will accelerate research cycles and lets you engage with peers early on!

WHERE TO FIND THE LECTURE MATERIAL

https://github.com/TobiBu/graddays


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS