# Numerical Solvers of Neural Differential Equations

Yan Huang

Department of Mathematics
Southern University of Science and Technology

May 20, 2022

# Contents

# General principles

- Neural networks are unstructured vector fields, we must use general solvers. (white lie?)
- **Implicit solvers**: Computationally expensive.
- **Adaptive versus fixed step solvers**: Both are reasonable choice for NDE.
- **Baked-in discretizations**: Single solver may overfit, but for many applications this is not a problem.
- **Step size and error tolerance**: They are very large compared to that in usual numerical differential equation method since it can reduce computational cost.
- **Order of solver**: Low-order solvers are reasonable in the large step size regime. If aiming to fit the idealized continuous model, then higher-order solvers are better.

# Methods for ODE, CDE and SDE

- **ODE and CDE**: CDE can reduced to ODE.
  Standard low-order solvers: explicit Euler method (1 order), the midpoint method (2 order), Heun's method (2 order).
  Standard higher-order solvers: RK4 (4 order), Dormand-Prince (5 order), Tsit5 (5 order).
- **SDE**: Euler-Maruyama method or Heun's method for *Itô* or Stratonovich SDE respectively. If problem has commutative noise then Milstein's method can be applied for both *Itô* and Stratonovich SDE.

# Contents

# Backpropagation through a reversible solver

## Definition 1

A solver is reversible if $(y_j, \alpha_j)$ can be computed from $(y_{j+1}, \alpha_{j+1})$.

---

**Algorithm** Backpropagation through a reversible solver.

$t$ denotes time, $y$ denotes the numerical solution, $\alpha$ denotes any additional state the solver keeps around, and $\Delta t$ denotes a step size. $x$ denotes a possible control input, which may be unused if the equation is an ODE and may be a Brownian motion if the equation is an SDE.

---

**Input:** $t_{j+1}, y_{j+1}, \alpha_{j+1}, \Delta t, x, \frac{\partial L(y_n)}{\partial y_{j+1}}, \frac{\partial L(y_n)}{\partial \alpha_{j+1}}$

$y_j, \alpha_j = \mathit{Reverse}(t_{j+1}, y_{j+1}, \alpha_{j+1}, \Delta t, x)$

$y_{j+1}, \alpha_{j+1} = \mathit{Forward}(t_j, y_j, \alpha_j, \Delta t, x)$

$$\frac{\partial L(y_n)}{\partial (y_j, \alpha_j)} = \frac{\partial L(y_n)}{\partial (y_{j+1}, \alpha_{j+1})} \cdot \frac{\partial (y_{j+1}, \alpha_{j+1})}{\partial (y_j, \alpha_j)}$$

**Output:** $t_j, y_j, \alpha_j, \frac{\partial L(y_n)}{\partial y_j}, \frac{\partial L(y_n)}{\partial \alpha_j}$

---

# Backpropagation through a reversible solver

- **Computational cost**: Assuming that a forward and reverse step cost the same. Overall computational cost of evaluating and backpropagating through a step of a reversible solver is 3 forward operations and 1 backward operation. As a combined forward/backward operation costs at most 4 forward operations, then the overall computational cost is approximately 6 forward operations.
- **Precise gradients**: The computed gradients are precisely the discretize-then-optimise gradients of the numerical discretization of the forward pass since the same $y_j$ is recovered on both the forward and backward passes.

# Two types of reversibility

- **Analytic reversibility**: $(y_j, \alpha_j)$ can not be written as a closed-form expression of $(y_{j+1}, \alpha_{j+1})$, like backwards-in-time implicit Euler method, it is both approximate and computationally expensive.
- **Algebraic reversibility**: $(y_j, \alpha_j)$ can be written as a closed-form expression of $(y_{j+1}, \alpha_{j+1})$, so it is computationally reasonable.
- **Symmetric algebraically reversibility**:
  $(y_{j+1}, \alpha_{j+1}) = \mathit{step}(y_j, \alpha_j, \Delta t)$ implies
  $(y_j, \alpha_j) = \mathit{step}(y_{j+1}, \alpha_{j+1}, -\Delta t)$.

# Method 1: Reversible Heun method

- **Introduction**: Reversible Heun method is a symmetric algebraically reversible ODE, CDE, or SDE solver. Moreover, it is the only non-symplectic algebraically reversible SDE solver by now.
- Consider solving the Stratonovich SDE

$$y(0) = y_0, dy(t) = \mu(t, y(t))dt + \sigma(t, y(t)) \circ d\omega(t)$$

over [0,T]. If solving an ODE, set $\sigma = 0$. If solving a CDE then either reduce it to an ODE or replace $\omega$ with some control $x$.

- **Use cases**: If training via discretise-then-optimise is not an option, then the reversible Heun is an excellent choice of solver for any of neural ODE, CDE, or SDE. But it is not recommended when solving neural differential equations with built-in structure such as the physical neural networks, where the low order and lack of stability be concerned.

# Method 1: Reversible Heun method

**Algorithm** Forward pass for the reversible Heun method.

**Initialization:** $\hat{y}_0 = y_0$, $\mu_0 = \mu(0, y_0)$, $\sigma_0 = \sigma(0, y_0)$, $\omega$ be a single sample of Brownian motion

**Input:** $t_j, y_j, \hat{y}_j, \mu_j, \sigma_j, \Delta t, \omega$

$$t_{j+1} = t_j + \Delta t$$
$$\Delta\omega_j = \omega(t_{j+1}) - \omega(t_j)$$
$$\hat{y}_{j+1} = 2y_j - \hat{y}_j + \mu_j \Delta t + \sigma_j \Delta\omega_j$$
$$\mu_{j+1} = \mu(t_{j+1}, \hat{y}_{j+1})$$
$$\sigma_{j+1} = \sigma(t_{j+1}, \hat{y}_{j+1})$$
$$y_{j+1} = y_j + \tfrac{1}{2}(\mu_j + \mu_{j+1})\Delta t + \tfrac{1}{2}(\sigma_j + \sigma_{j+1})\Delta\omega_j$$

**Output:** $t_{j+1}, y_{j+1}, \hat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1}$

**Remark.** The solver tracks some extra pieces of state, $\hat{y}_j, \mu_j, \sigma_j$ which have solution-like, drift-like and diffusion-like interpretations respectively.

# Method 1: Reversible Heun method

**Algorithm** Reverse pass for the reversible Heun method.

**Input:** $t_{j+1}, y_{j+1}, \hat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1}, \Delta t, \omega, \frac{\partial L(y_n)}{\partial y_{j+1}}, \frac{\partial L(y_n)}{\partial \hat{y}_{j+1}}, \frac{\partial L(y_n)}{\partial \mu_{j+1}}, \frac{\partial L(y_n)}{\partial \sigma_{j+1}}$

**Reverse step:**

$t_j = t_{j+1} - \Delta t$

$\Delta\omega_j = \omega(t_{j+1}) - \omega(t_j)$

$\hat{y}_j = 2y_{j+1} - \hat{y}_{j+1} - \mu_{j+1}\Delta t - \sigma_{j+1}\Delta\omega_j$

$\mu_j = \mu(t_j, \hat{y}_j)$

$\sigma_j = \sigma(t_j, \hat{y}_j)$

$y_j = y_{j+1} - \frac{1}{2}(\mu_{j+1} + \mu_j)\Delta t - \frac{1}{2}(\sigma_{j+1} + \sigma_j)\Delta\omega_j$

**Local forward:** $y_{j+1}, \hat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1} = \mathit{Forward}(t_j, y_j, \hat{y}_j, \mu_j, \sigma_j, \Delta t, \omega)$

**Local backward:** $\frac{\partial L(y_n)}{\partial(y_j, \hat{y}_j, \mu_j, \sigma_j)} = \frac{\partial L(y_n)}{\partial(y_{j+1}, \hat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1})} \cdot \frac{\partial(y_{j+1}, \hat{y}_{j+1}, \mu_{j+1}, \sigma_{j+1})}{\partial(y_j, \hat{y}_j, \mu_j, \sigma_j)}$

**Output:** $t_j, y_j, \hat{y}_j, \mu_j, \sigma_j, \frac{\partial L(y_n)}{\partial y_j}, \frac{\partial L(y_n)}{\partial \hat{y}_j}, \frac{\partial L(y_n)}{\partial \mu_j}, \frac{\partial L(y_n)}{\partial \sigma_j}$

# Method 1: Reversible Heun method

## Theorem 2

*The reversible Heun method, when applied to ODE, is a second order method.*

## Theorem 3

*The reversible Heun method, when applied to SDE, exhibits strong convergence of order $\frac{1}{2}$. i.e. there exists a constant $C > 0$ so that $\mathbb{E}[\|y_n - y(T)\|_2] \leq C\sqrt{\Delta t}$. If the noise is additive then this improves to order 1.*

## Theorem 4

*The region of stability for the reversible Heun method (for ODE) is the complex interval $[-i, i]$.*

# Method 2: Asynchronous leapfrog method

- **Introduction**: Asynchronous leapfrog method is a symmetric algebraically reversible ODE or CDE solver.
- Consider solving the ODE:

$$y(0) = y_0, \frac{dy(t)}{dt} = f(t, y(t))$$

  over $[0, T]$.
- **Use cases**: The asynchronous leapfrog method is useful in essentially the same cases as the reversible Heun method, except that it applies only for ODE.

# Method 2: Asynchronous leapfrog method

**Algorithm** Forward pass through the asynchronous leapfrog method.

**Initialization:** $v_0 = f(y_0)$

**Input:** $t_j, y_j, v_j, \Delta t$

$$\hat{t}_j = t_j + \frac{\Delta t}{2}$$
$$\hat{y}_j = y_j + v_j \frac{\Delta t}{2}$$
$$\hat{v}_j = f(\hat{t}_j, \hat{y}_j)$$

$$t_{j+1} = t_j + \Delta t$$
$$y_{j+1} = y_j + \hat{v}_j \Delta t$$
$$v_{j+1} = 2\hat{v}_j - v_j$$

**Output:** $t_{j+1}, y_{j+1}, v_{j+1}$

**Remark.** The solver tracks a single state $v_j$, which has a velocity-like interpretation.

# Method 2: Asynchronous leapfrog method

**Algorithm** Efficient backward pass through the asynchronous leapfrog method (local forward can be elided).

**Input:** $t_{j+1}, y_{j+1}, v_{j+1}, \Delta t, \frac{\partial L(y_n)}{\partial y_{j+1}}, \frac{\partial L(y_n)}{\partial v_{j+1}}$

$$\hat{t}_j = t_{j+1} - \frac{\Delta t}{2}$$
$$\hat{y}_j = y_{j+1} - v_{j+1}\frac{\Delta t}{2}$$
$$\hat{v}_j = f(\hat{t}_j, \hat{y}_j)$$

$$t_j = t_{j+1} - \Delta t$$
$$y_j = y_{j+1} - \hat{v}_j \Delta t$$
$$v_j = 2\hat{v}_j - v_{j+1}$$
$$\frac{\partial L(y_n)}{\partial y_j} = (2\frac{\partial L(y_n)}{\partial v_{j+1}} + \frac{\partial L(y_n)}{\partial y_{j+1}}\Delta t) \cdot \frac{\partial \hat{v}_j}{\partial \hat{y}_j} + \frac{\partial L(y_n)}{\partial y_{j+1}}$$
$$\frac{\partial L(y_n)}{\partial v_j} = \frac{1}{2}\frac{\partial L(y_n)}{\partial y_{j+1}}\Delta t - \frac{\partial L(y_n)}{\partial v_{j+1}}$$

**Output:** $t_j, y_j, v_j, \frac{\partial L(y_n)}{\partial y_j}, \frac{\partial L(y_n)}{\partial v_j}$

# Method 2: Asynchronous leapfrog method

## Theorem 5

*The asynchronous leapfrog method is a second-order method. Specifically, the local truncation error in $y$ is $O(\Delta t^3)$, while the local truncation error in $v$ is $O(\Delta t^2)$*

## Theorem 6

*The region of stability for the asynchronous leapfrog method is the complex interval $[-i, i]$.*

# Method 3: Symplectic solvers

- **Introduction**: Symplectic integrator is a numerical scheme that intends to solve a Hamiltonian system approximately, while preserving its Hamiltonian and flow.

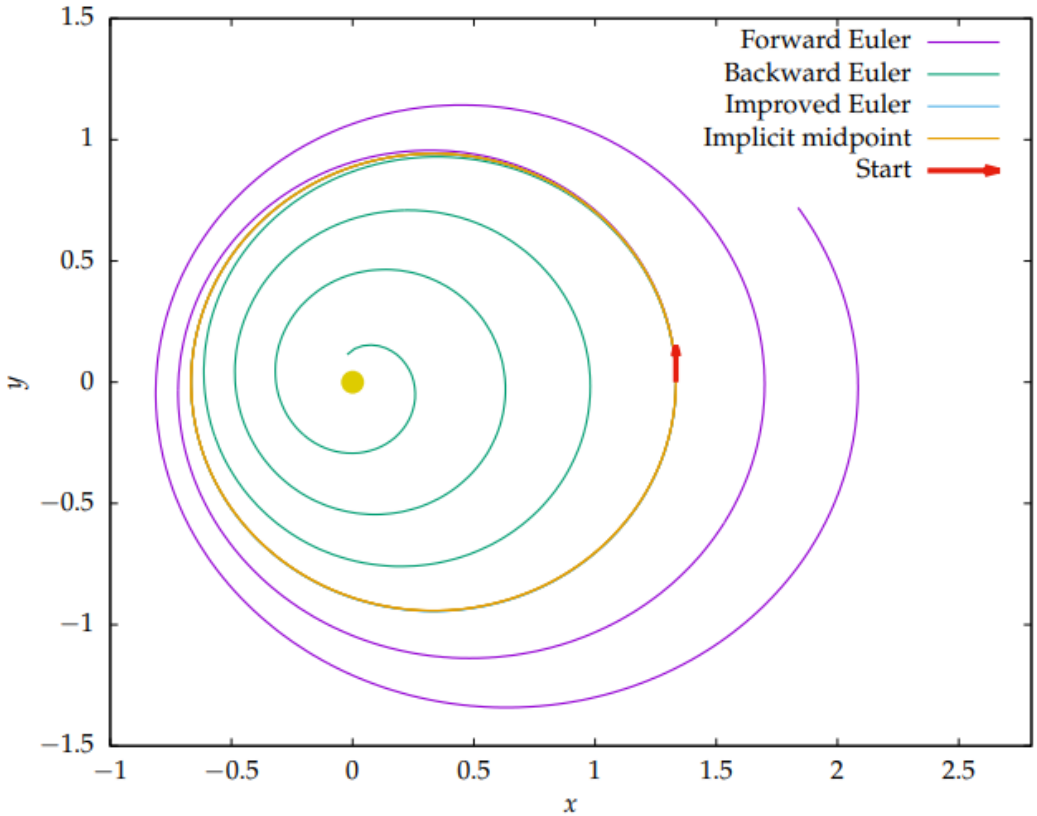



Figure 1: Computed trajectories using four different integration methods for the asteroid orbit test problem. An elliptical orbit with semi-major axis $a = 1$ and eccentricity $e = 1/3$ is used. The vector shows the initial position and direction of the asteroid. The yellow circle indicates the position of the star.

# Method 3: Symplectic solvers

- Many pre-existing symplectic solvers are already algebraically reversible.
- **Semi-implicit Euler method**: Given an ODE system,

$$y(0) = y_0, \quad \frac{dy(t)}{dt} = f(t, v(t)),$$
$$v(0) = v_0, \quad \frac{dv(t)}{dt} = g(t, y(t)).$$

the semi-implicit Euler method is defined by

$$y_{j+1} = y_j + f(t_j, v_{j+1})\Delta t,$$
$$v_{j+1} = v_j + g(t_j, y_j)\Delta t.$$

- **Use cases**:It is the solver used in the rotational vector fields and momentum residual networks.

# Method 3: Symplectic solvers

- **Leapfrog/midpoint method**: Given an ODE over $[0, T]$,

$$y(0) = y_0, \quad \frac{dy(t)}{dt} = f(t, y(t))$$

the leapfrog/midpoint method is defined by

$$y_{j+1} = y_{j-1} + f(t_j, y_j)\Delta t,$$
$$y_{j+2} = y_j + f(t_{j+1}, y_{j+1})\Delta t.$$

- **Use cases**: It can be applied to general first-order system.

# Contents

# Solving vector fields with jumps

- **Background**: The vector field of a neural ODE has a piecewise structure w.r.t time.

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

where

$$f_\theta(t, y(t)) = \begin{cases} f_{\theta,1}(t, y) & t \in [t_0, t_1] \\ \vdots & \\ f_{\theta,n}(t, y) & t \in [t_{n-1}, t_n] \end{cases}$$

This occurs when solving a neural CDE, reduced to a Neural ODE, using linear or rectilinear interpolation.

# Solving vector fields with jumps

- **Separate calls**: If making n separate calls to an ODE solver, and training the neural ODE either via optimise-then-discretize or via reversible ODE solvers, the memory cost will be n times larger than if we had made a single ODE solver.
- **Single call**: If making a single call to the ODE solver, and using an adaptive step size ODE solver, then the solver should be informed about the location and size of the jumps.

# Contents

# Hypersolvers

- **Motivations**: Higher-order solvers requires to compute and store many intermediate variables. To avoid this situation, we can introduce an additional neural network $g_\omega$ to approximate the higher–order terms of a given low-order solver.
- **General formulation**: Consider some $q$-th order ODE solver:

$$y_{j+1} = y_j + \phi(t_j, y_j)\Delta t$$

A hypersolver is defined by a learnt correction:

$$y_{j+1} = y_j + \phi(t_j, y_j)\Delta t + g_\omega(t_j, y_j, \Delta t)\Delta t^{q+1}$$

where $g_\omega$ is some neural network depending on $\omega$.

# Hypersolvers

- **Residual fitting**: Training is performed by assuming access to the true solution of the neural ODE. In practice it may be obtained by using a traditional numerical solver with high order, small step sizes, or tight error tolerances. The hypersolvers may trained by minimizing either

$$\frac{1}{n}\sum_{j=0}^{n-1} \|R(t_j, y(t_j), y(t_{j+1})) - g_\omega(t_j, y(t_j), \Delta t)\|^2$$

or

$$\frac{1}{n}\sum_{j=0}^{n-1} \|R(t_j, y_j, y(t_{j+1})) - g_\omega(t_j, y_j, \Delta t)\|^2$$

where

$$R(t, y_{prev}, y_{next}) = \frac{y_{next} - y_{prev} - \phi(t, y_{prev})\Delta t}{\Delta t^{q+1}}$$

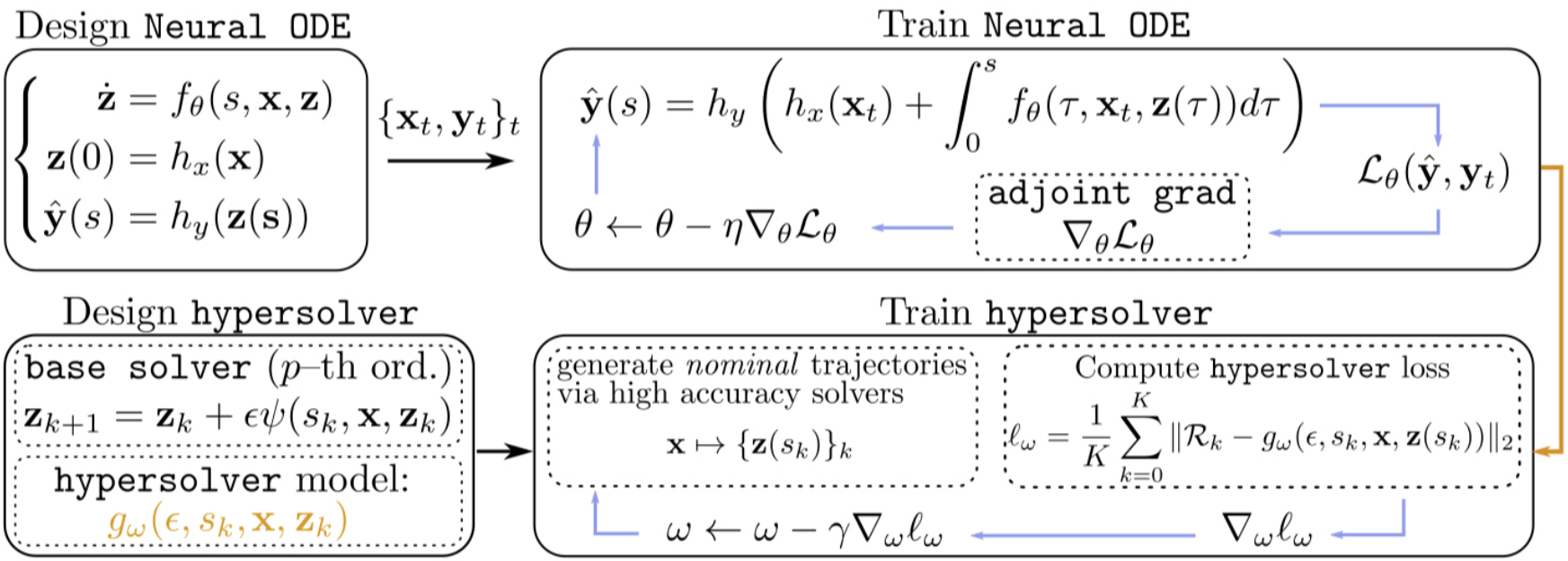
# Hypersolvers

- **Trajectory fitting**: The second type of hypersolvers training aims at containing the global truncation error by minimizing the difference between the exact and approximated solutions in the whole depth domain. i.e.

$$\sum_{j=1}^{n} \|y(t_j) - y_j\|^2$$

- **Applications**: In continuous normalising flow, 80 Dormand–Prince steps may be replaced by only 2 HyperHeun steps, without decreasing accuracy. However, hypersolvers are generally only useful for speeding up inference, not training, since the neural ODE changes during training.

# Hypersolvers




**A blueprint for hypersolver design**

1. design your continuous model & task
2. train your NeuralODE
3. choose a high accuracy solver
4. choose a base solver ($\psi$)
5. choose a model $g_\omega$ for the hypersolver
6. generate nominal trajectories of the Neural ODE via the high accuracy solver
7. train the hypersolver network