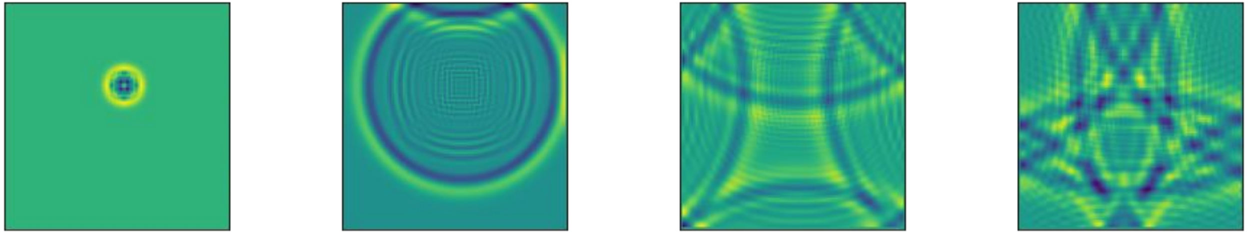
Learning Earth system model dynamics with implicit schemes



Marcel Nonnenmacher

Model-driven Machine Learning
Institute for Coastal Research
Helmholtz-Centre Hereon

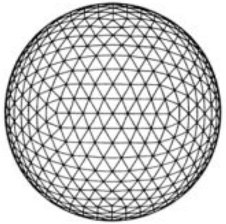
Helmholtz-Zentrum hereon

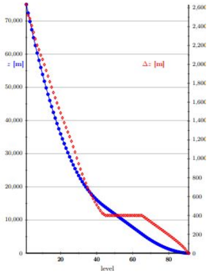
Model-driven Machine Learning

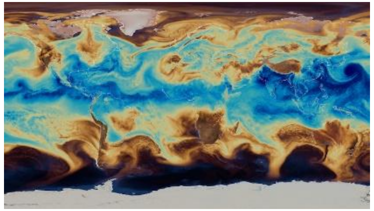
HELMHOLTZ AI | ARTIFICIAL INTELLIGENCE COOPERATION UNIT

Earth Science models and their applications

ICON model for numerical weather prediction (DWD, MPI-M)







2949120
grid cells

90
z-levels

10+
variables / location

Total values / time point: **2.6 billion**

Or with a time step of 120 seconds,
Total values / day: **1.9 trillion**

Images: Stevens et al., 2019, Reinert et al., 2021

Data assimilation application for weather forecasts (ECMWF)

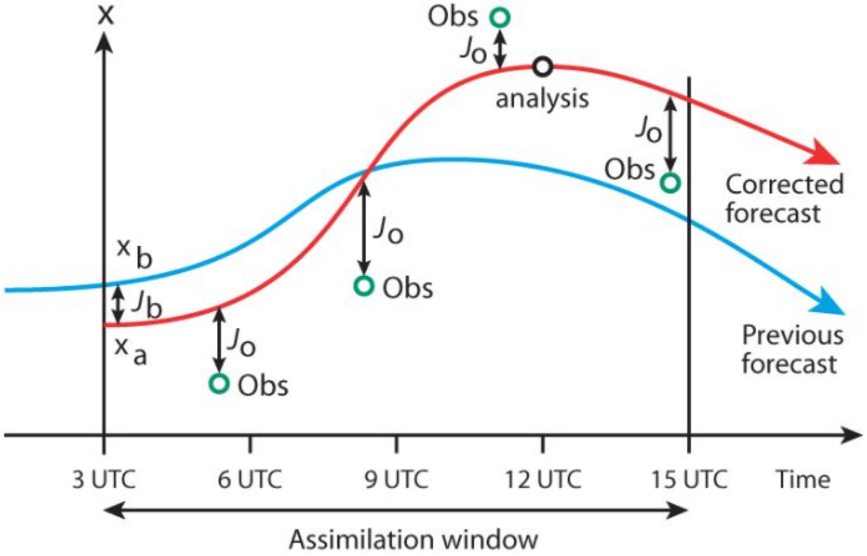
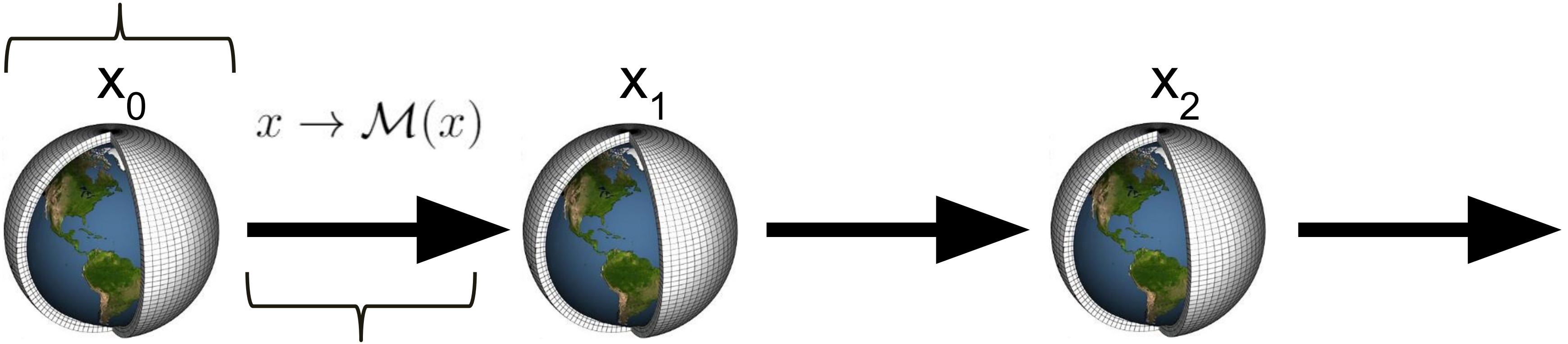


Image source: ecmwf.int
“20 years of 4D-Var: better forecasts through a better use of observations“

Physics-based Simulation in Earth Science

System state
- Physical variables (pressure, temperature, …)
- Snapshot for one time point



State update function
- Physical processes (radiation, turbulence, phase changes, …)
- Mass transfer due to gravity, wind, precipitation, …

Derivatives of state update function $\frac{dX_1}{dX_0}$
- Data assimilation, model tuning, ...

Emulation of numerical simulators

Simulation for Earth Science models is **expensive**.

Mistral, DKRZ (>10^5 processors, 3.6 PetaFLOPS)

Emulation: ML model imitates the state update function of a numerical simulator.

Possible advantages of ML models:

- parallelization on GPU clusters
- may find "shortcuts" from large datasets
- trained in "auto-diff" environments
  (no need to write and maintain derivative routines)

Simulation code: FORTRAN, often > 10^5 lines

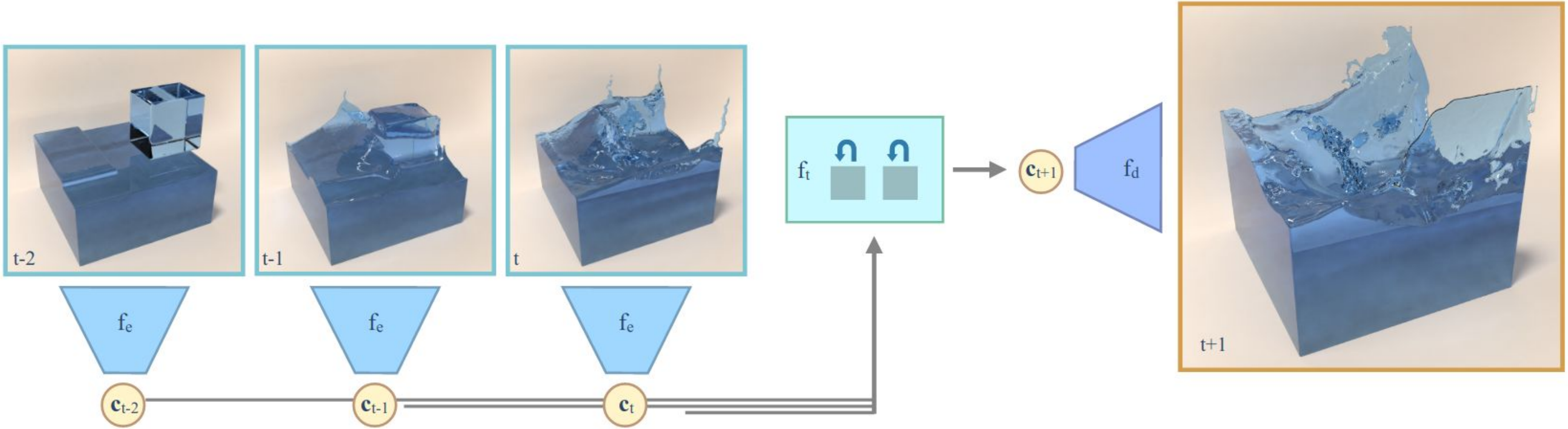
```
do i = 1,nx,1
  im = max(1,i-1)
  ip = min(nx,i+1)

  if((d(i).gt.0.1).and.(d(ip).gt.0.1)) then
    ce(i) = dt*dt*wimp*wimp*g*0.5*(h(i)+h(ip))/(dx*dx)
  endif

  if((d(i).gt.0.1).and.(d(im).gt.0.1)) then
    cw(i) = dt*dt*wimp*wimp*g*0.5*(h(i)+h(im))/(dx*dx)
  endif

enddo
```

Emulator examples



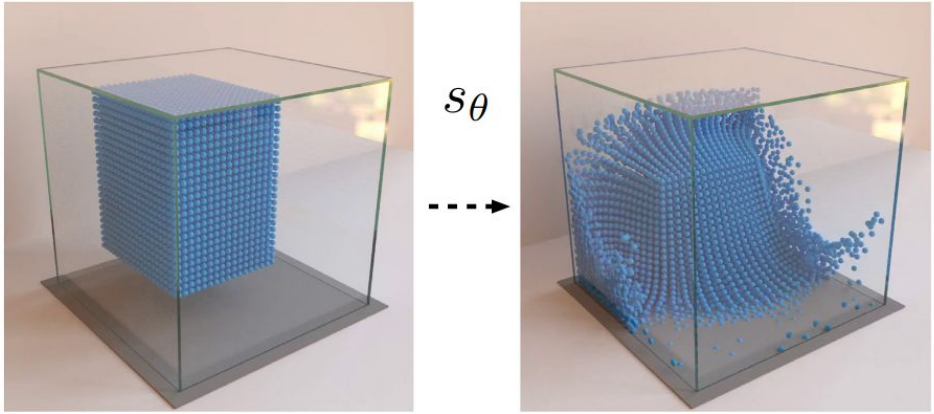
S. Wiewel[1], M. Becher[1], N. Thuerey[1 †]

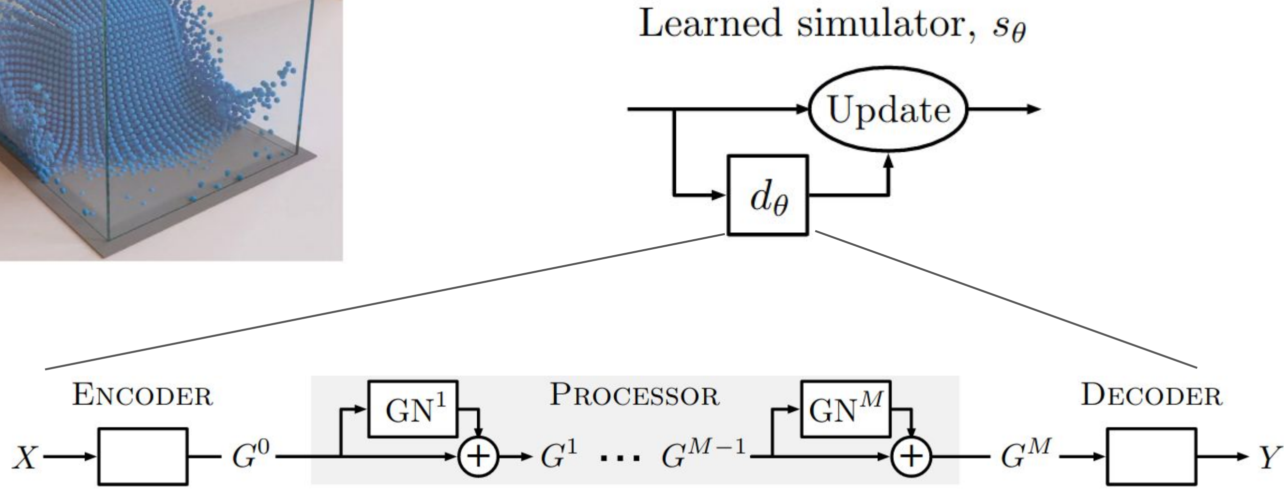
[1]Technical University of Munich

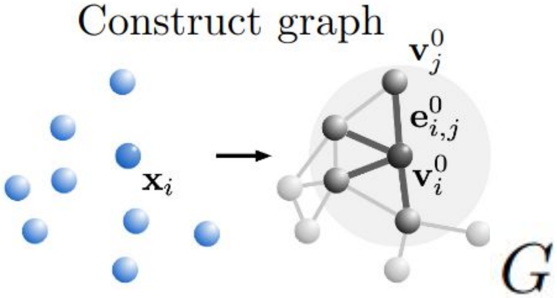
Computer Graphics Forum, Volume 38 (2019) – Issue 2

Presented at Eurographics Conference 2019, May 2019

Emulator examples

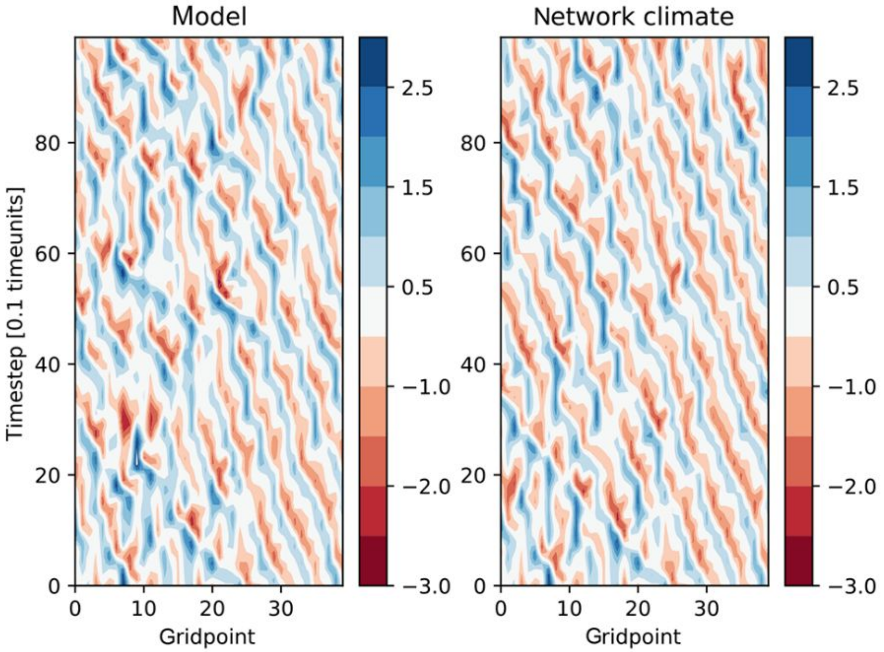






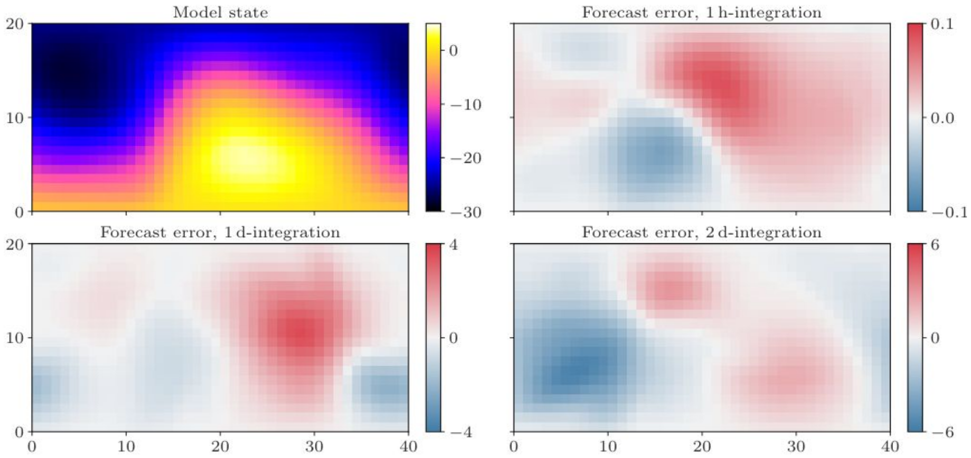
Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, Peter Battaglia *Proceedings of the 37th International Conference on Machine Learning*, PMLR 119:8459-8468, 2020.

Emulator examples



Nonlin. Processes Geophys., 26, 381–399, 2019
https://doi.org/10.5194/npg-26-381-2019

Sebastian Scher[1] and Gabriele Messori[1,2]
[1]Department of Meteorology and Bolin Centre for Climate Research, Stockholm University,
[2]Department of Earth Sciences, Uppsala University, Uppsala, Sweden



Alban Farchi
Marc Bocquet
CEREA
joint laboratory École des Ponts ParisTech and EDF R&D
Champs-sur-Marne, France

Patrick Laloyaux
Massimo Bonavita
ECMWF
Shinfield Park
Reading, United Kingdom

arXiv:2010.12605v2 [stat.ML] 10 May 2021

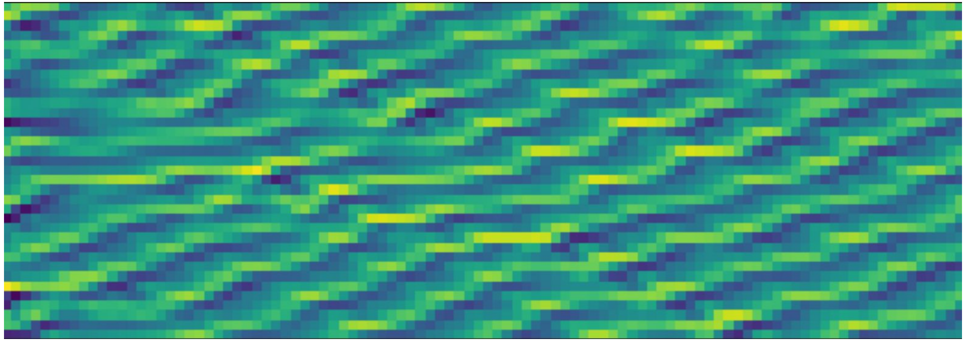
The core principles (L96)

Model System: Lorenz `96

$$\frac{dx_k}{dt} = -x_{k-1}\left(x_{k-2} - x_{k+1}\right) - x_k + F$$

Lorenz, 1996

- 40 coupled nonlinear differential equations
- Chaotic dynamics (Lyapunov time 1.67 for F = 8)



Time

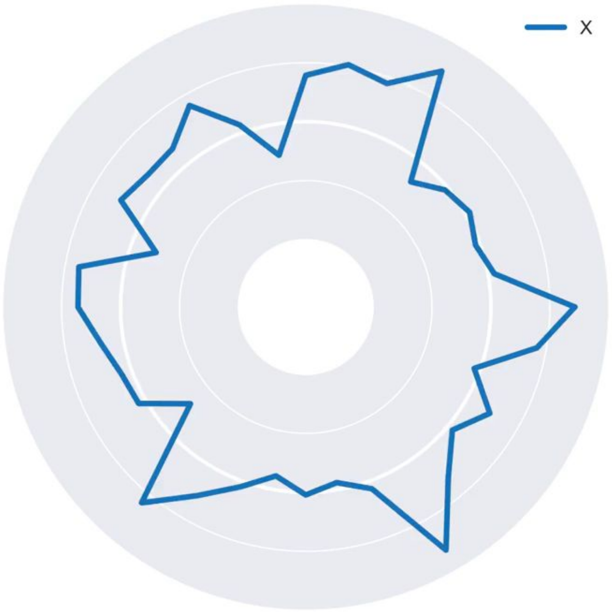
Model System: Lorenz `96

$$\frac{dx_k}{dt} = -x_{k-1}\left(x_{k-2} - x_{k+1}\right) - x_k + F$$

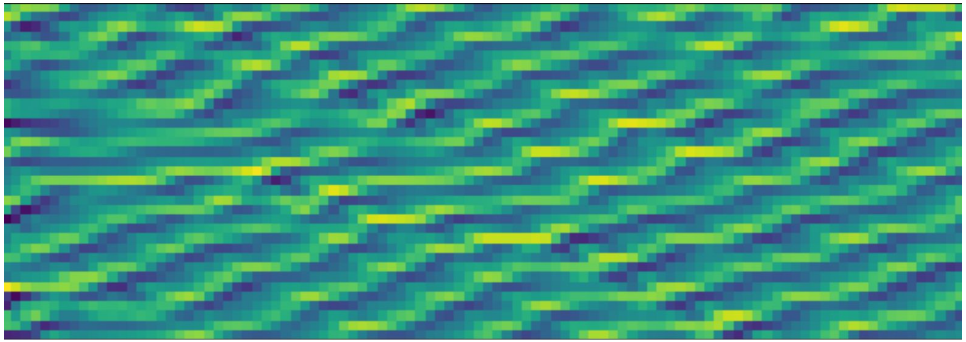
local computations !

Lorenz, 1996

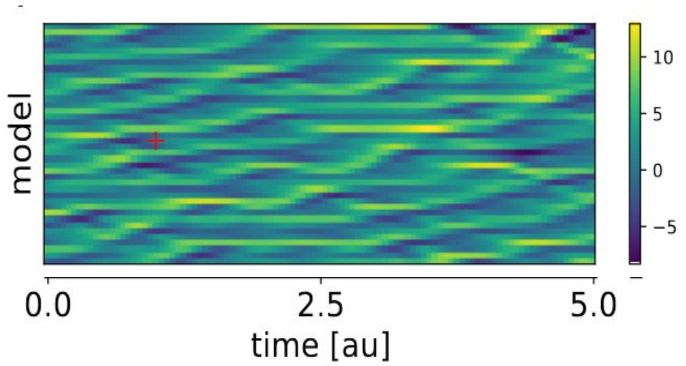
- 40 coupled nonlinear differential equations
- Chaotic dynamics (Lyapunov time 1.67 for F = 8)

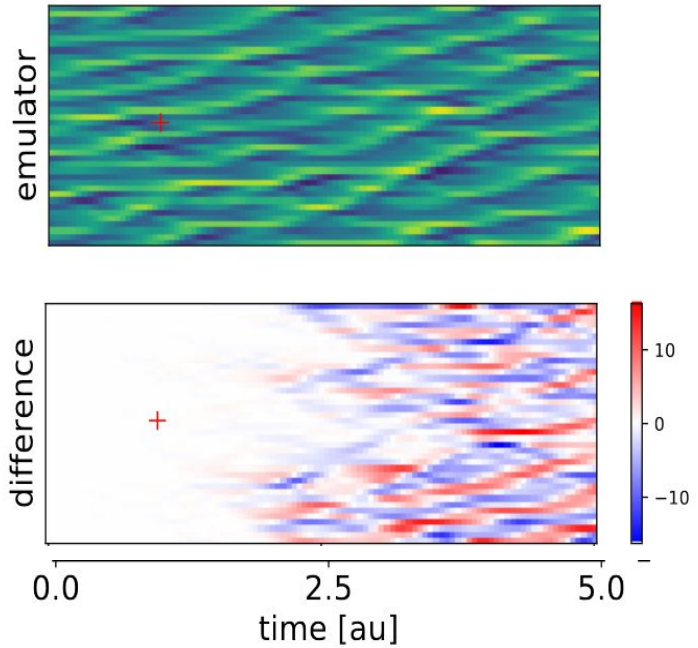


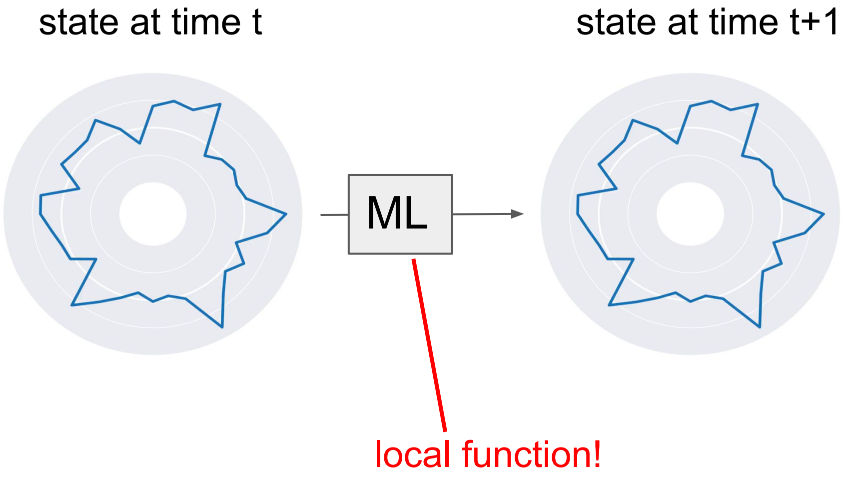
Time



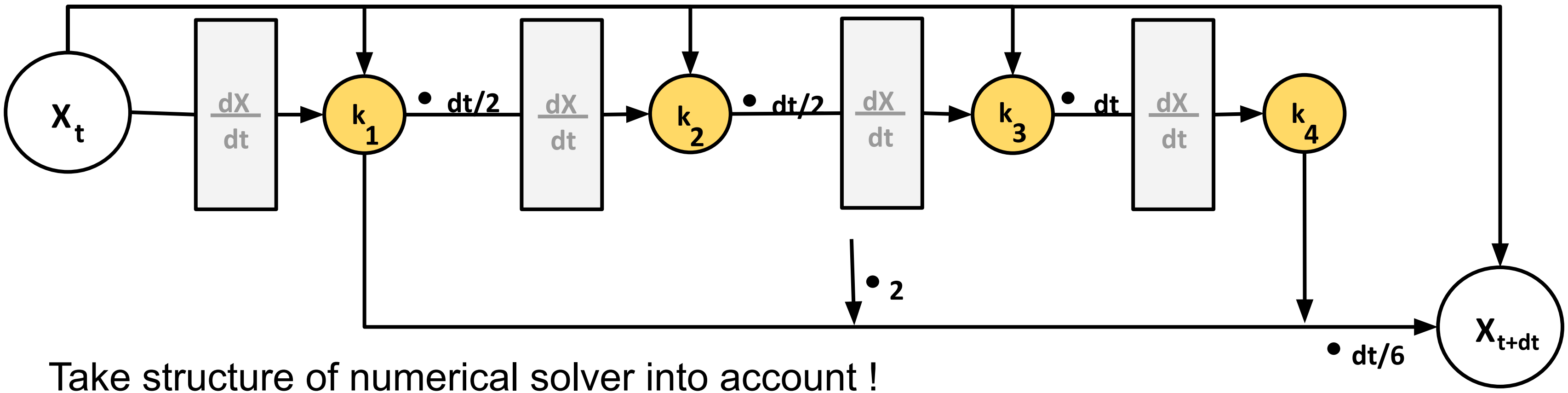
Basic emulator training







Structured ML models for learning the update step



Take structure of numerical solver into account !

RK4: $$X_{t+dt} = X_t + \frac{dt}{6}(k_1 + 2(k_2 + k_3) + k4)$$

$$k_1 = f(X_t)$$

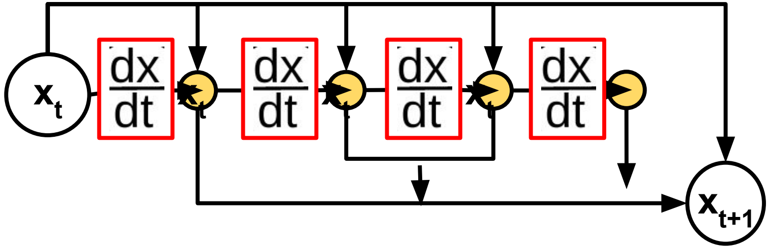
$$k_2 = f\left(X_t + k_1 \cdot \frac{dt}{2}\right)$$

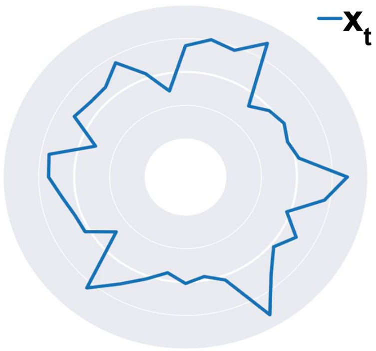
$$k_3 = f\left(X_t + k_2 \cdot \frac{dt}{2}\right)$$

$$k_4 = f\left(X_t + k_3 \cdot dt\right)$$

If f is differentiable (wrt. $\mathbf{x}_t$), so is $\mathbf{x}_{t+1}$!

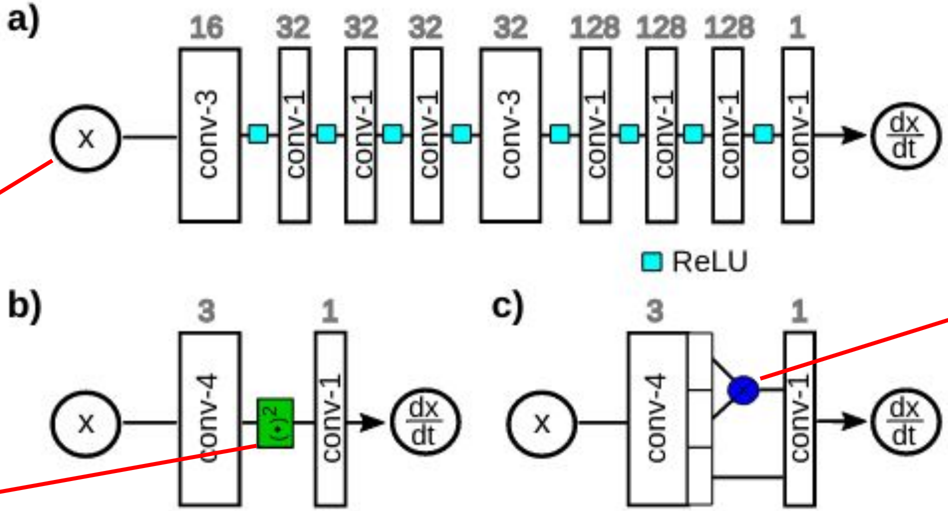
Network architectures





(circular) convs & ReLUs

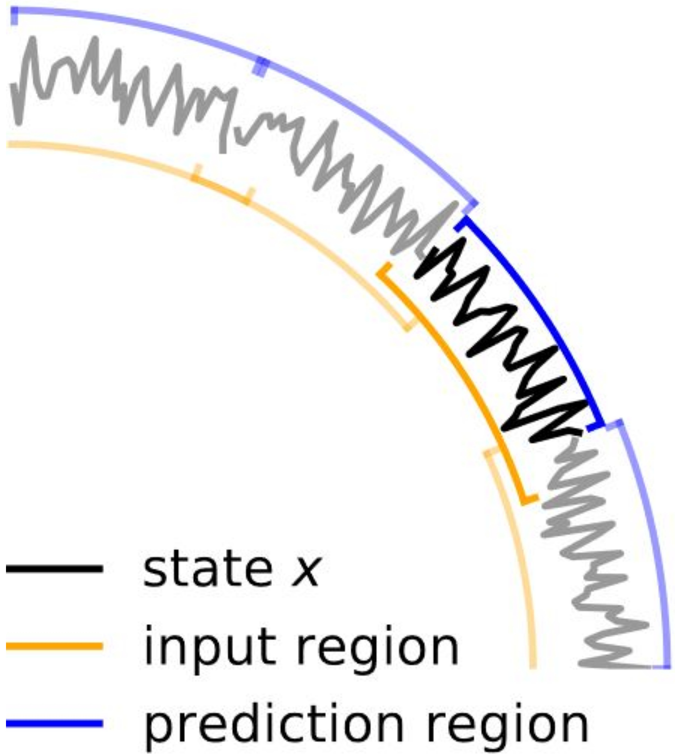
convolution = local function

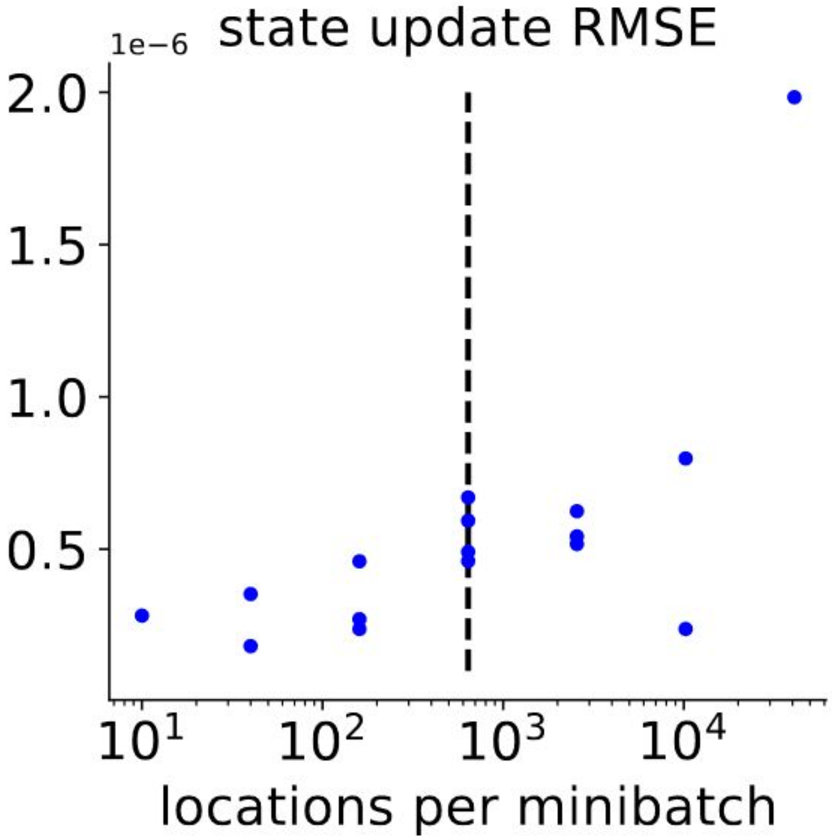


bilinear layer

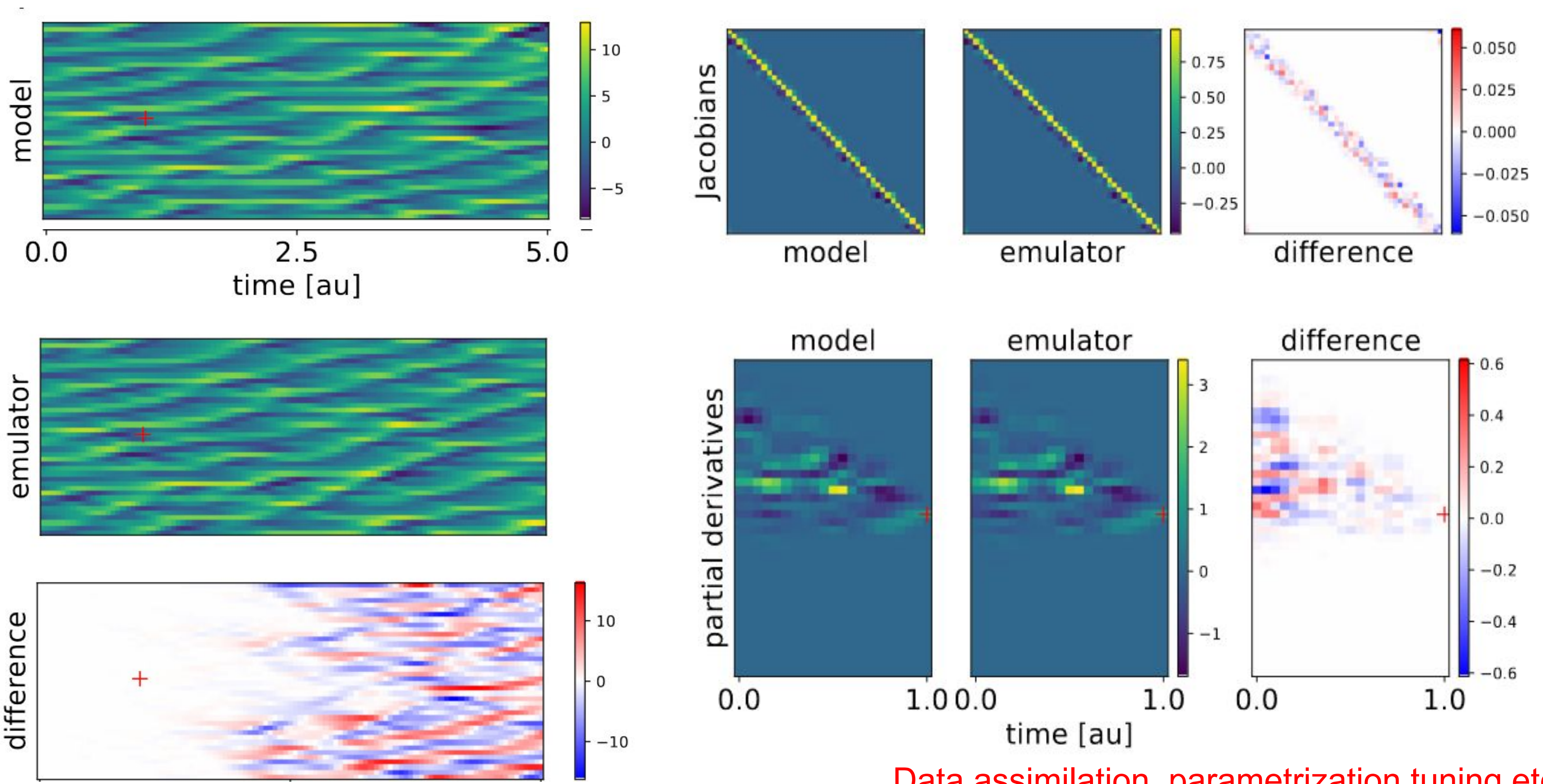
quadratic nonlinearity

Training on partial system states





Trained emulators reproduce system states and derivatives



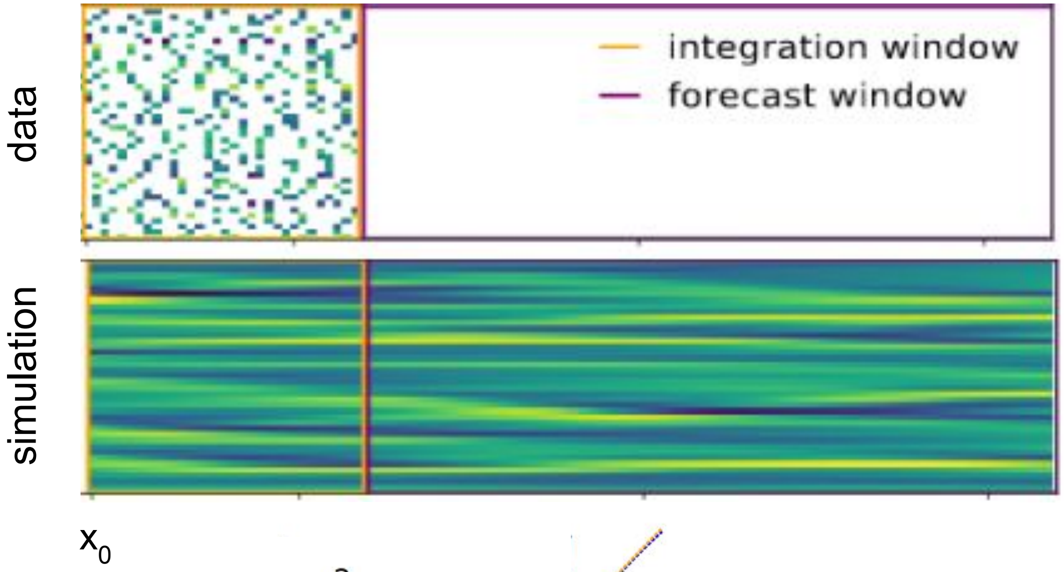
Data assimilation, parametrization tuning etc.

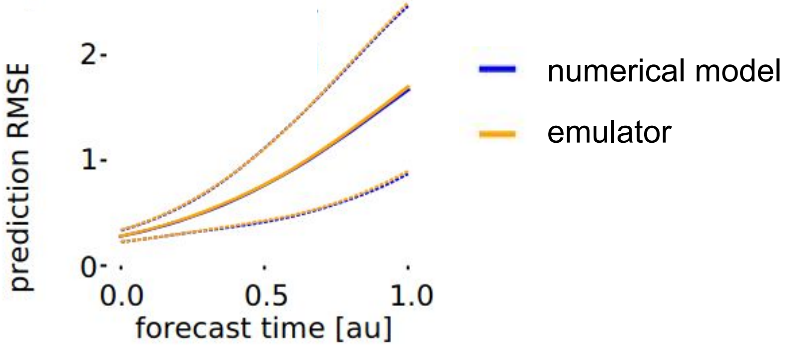
Emulators for 4D-Var data assimilation

1) with **differentiable** dynamical model, **estimate initial state x_0** from noisy & incomplete data.

2) starting from x_0, simulate future states beyond final data point.

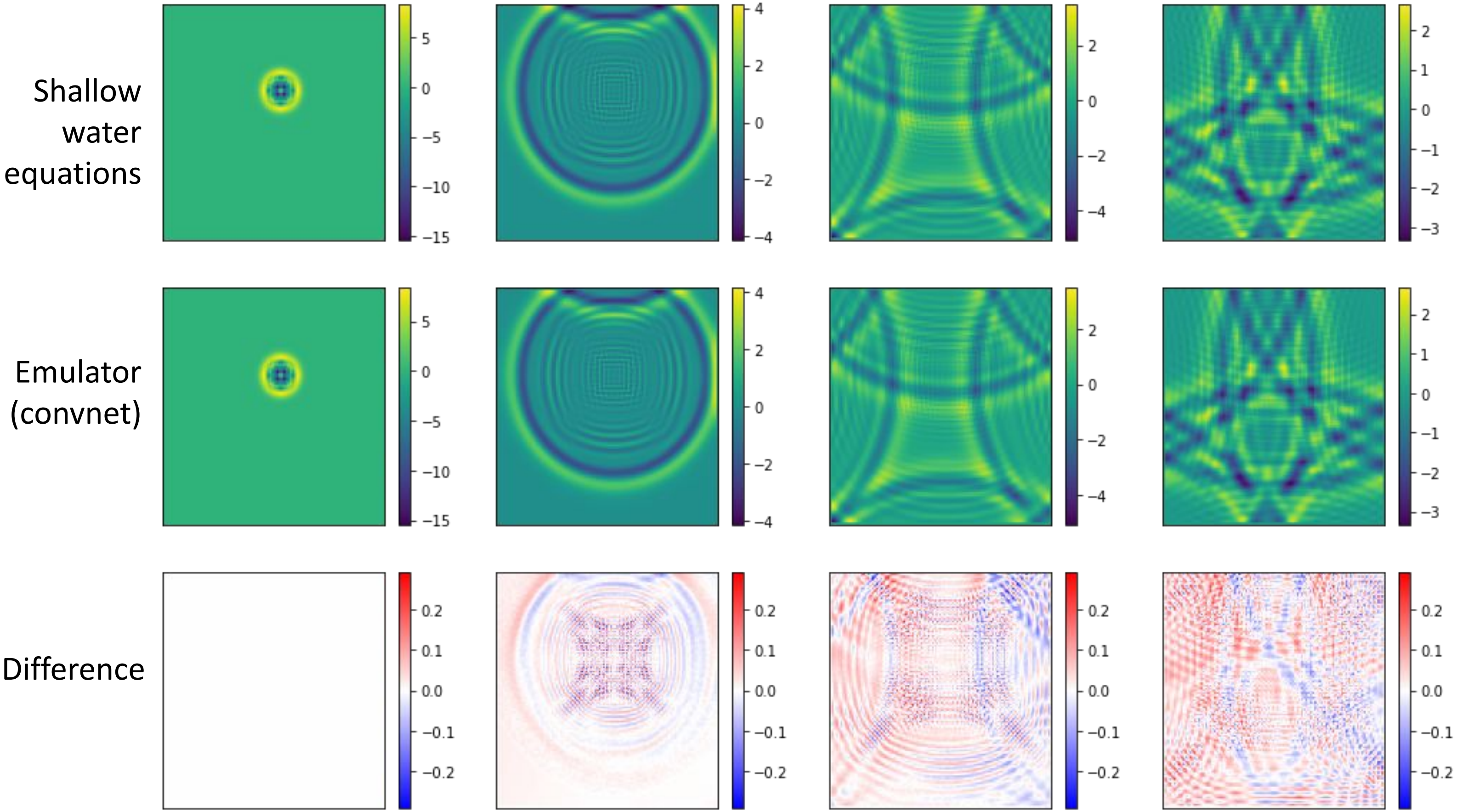
(optional) evaluate prediction error



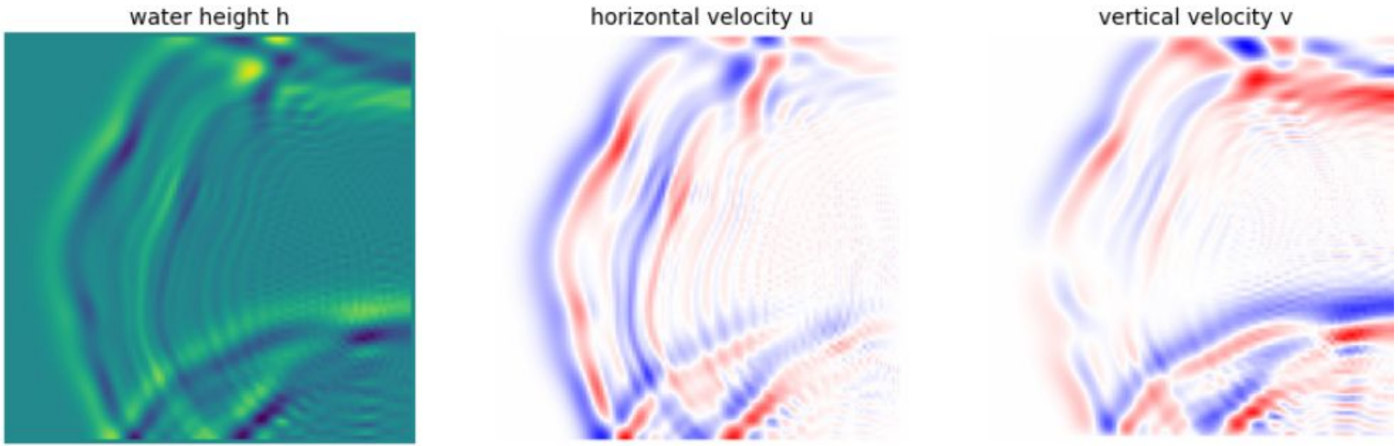


2D systems and beyond explicit solvers

Shallow Water Equations



Structure of SWE numerical solver



State update function of the numerical solver:

$$\left(h^{t+1},\ u^{t+1},\ v^{t+1} \right) = f\left(h^{t+1},\ h^{t},\ u^{t}, v^{t} \right)$$

(semi-)implicit !

(other fluid equations: similar for h = pressure)

More specifically here: $A\ h^{t+1} = b$

$$A = A(h^t, u^t, v^t),$$
$$b = b(h^t, u^t, v^t)$$

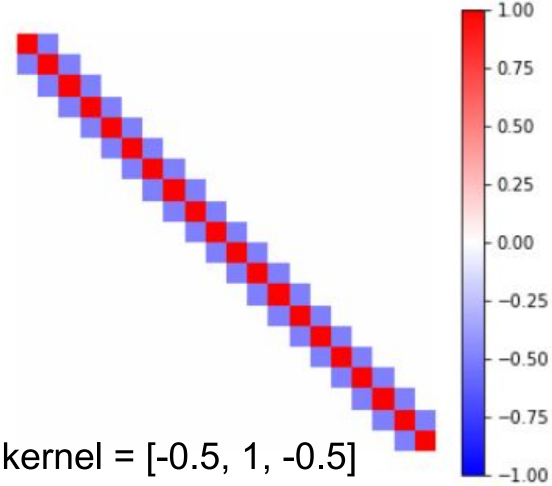
Implicit schemes = non-local state update functions

$$A\ h^{t+1} = b \qquad\qquad h^{t+1} = A^{-1}b$$

A and b may generally be local functions (in h^t, u^t, v^t),

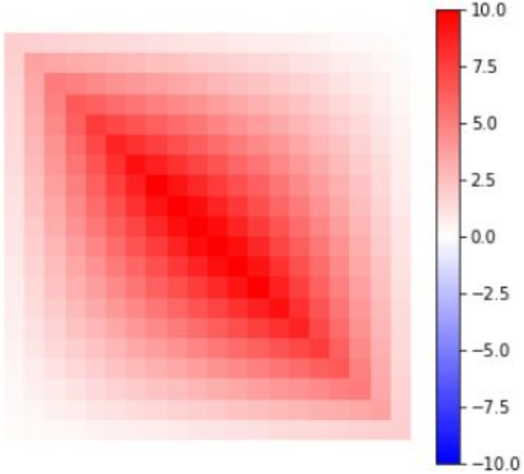
but A^{-1}generally is not!

A



kernel = [-0.5, 1, -0.5]

A^{-1}



Implicit schemes = non-local state update functions

$$A\ h^{t+1} = b \qquad\qquad h^{t+1} = A^{-1}b$$

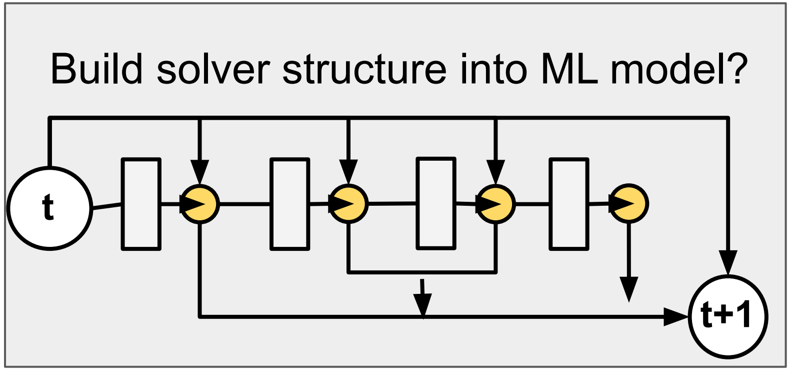
A and b may generally be local functions (in h^t, u^t, v^t),

but A^{-1}generally is not!

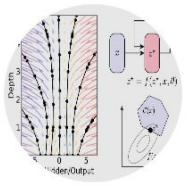
We could try:

- non-local feed-forward networks
  (lots of network parameters)

- U-Nets
  (scaling with e.g. doubled resolution?)



Linear implicit network layers



Deep Implicit Layers - Neural ODEs, Deep Equilibirum Models, and Beyond

http://implicit-layers-tutorial.org/

(linear) solve operations as part of a neural network model: for training need **backpropagation**!

1) forward pass: $y = \mathrm{layer}(x)$, where $A(x)\, y = b(x)$

2) backward pass: $\dfrac{dy}{dx} = \dfrac{\partial y}{\partial \mathrm{vec}(A)} \dfrac{d\mathrm{vec}(A)}{dx} + \dfrac{\partial y}{\partial b}\dfrac{db}{dx}$

Implicit function theorem (IFT) gives: $A\dfrac{\partial y}{\partial b} = I$ $\qquad A\dfrac{\partial y}{\partial a_j} = -y_j I$

$\dfrac{d\mathrm{vec}(A)}{dx}, \dfrac{db}{dx}$ depend on how we parametrize $A(x), b(x)$

Linear solvers

Any standard-library linear solver would do.

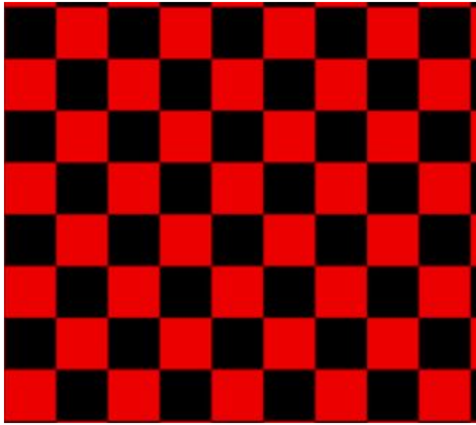
numpy.linalg.solve

```
linalg.solve(a, b)
```

For large system sizes (>>10^3 grid points), use linear solvers that **scale** accordingly.

One possible choice: implement **multi-color Gauss-Seidel** in pytorch

- valid for an important class of **banded matrices** A.
- **iterative method**, initialize with estimate of solution $\hat{y}$
- divide grid points into colors (e.g. red vs black)
- iterations only require dot products $A\ \hat{y}$ -> GPU !



Linear implicit layer in simple pytorch

```python
import torch
from utils import linear_solver

class LinearSolve(torch.autograd.Function):

    @staticmethod
    def forward(ctx, b, A):

        y = linear_solve(A, b)
```

solve #1

```python
        ctx.save_for_backward(A, y) # torch.autograd logistics

        return y

    @staticmethod
    def backward(ctx, grad_output):

        A, y = ctx.saved_tensors    # torch.autograd logistics
```

solve #2

```python
        z = linear_solve(
            A.T,            # tranpose of A
            grad_output     # backward gradient from reverse-mode differentiation : dL / dy
        )

        grad_input = (z,                                      # dL/dy * dy / db
                      -torch.bmm(z, y.transpose(-2, -1))      # dL/dy * dy / dA
                     )

        return grad_input

class LinearLayer(torch.nn.Module):

    def __init__(self, comp_bA):
        self.comp_bA = comp_bA # function that returns (b, A) as function of x

    def forward(self, x):
        b, A = self.comp_bA(x)
```

b, A = f(x) **and** (db/dx, dA/dx)

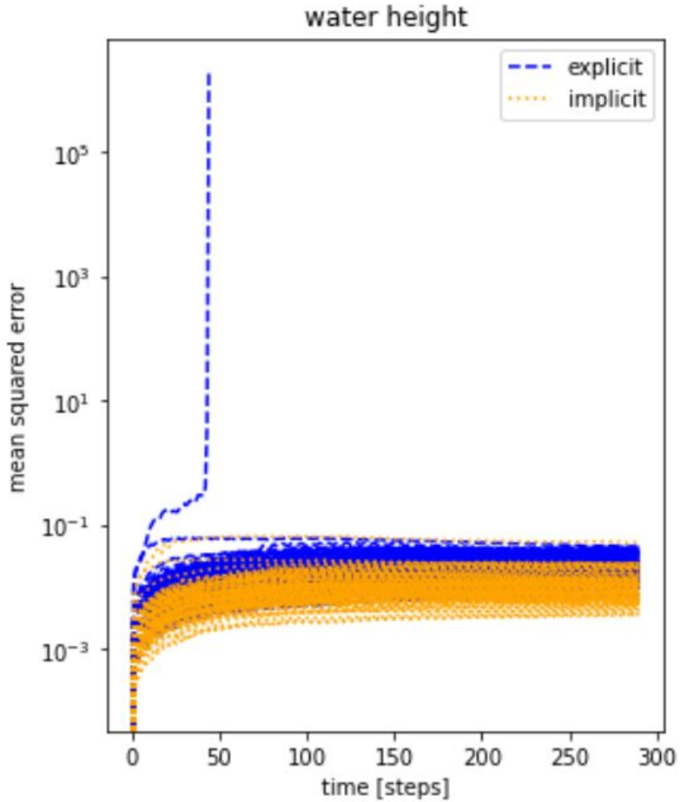
```python
        y = LinearSolve.apply(b, A)
        return y
```

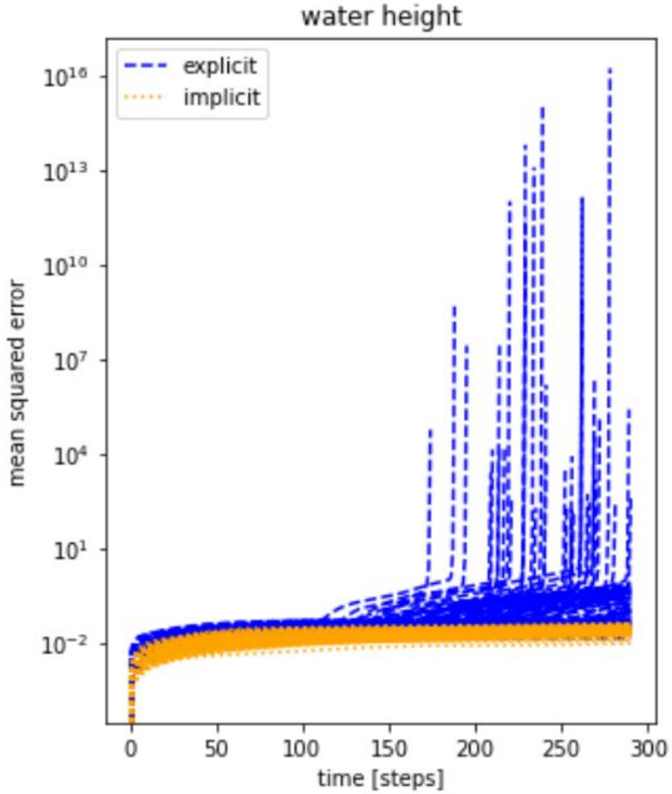
[this is essentially a re-write of torch.linalg.solve()
in pure pytorch (no c++) and for custom solvers]

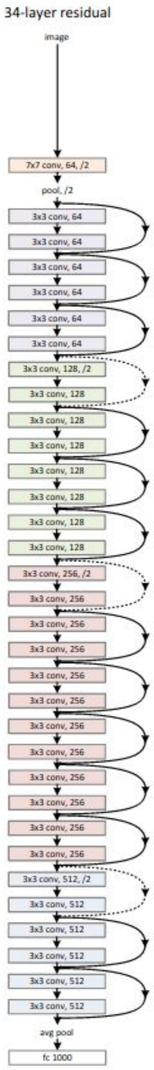
Results on SWE

Solver step length: dt = 300s

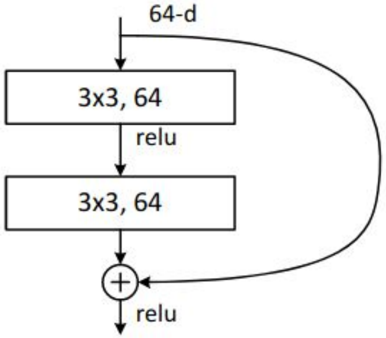
dt = 600s



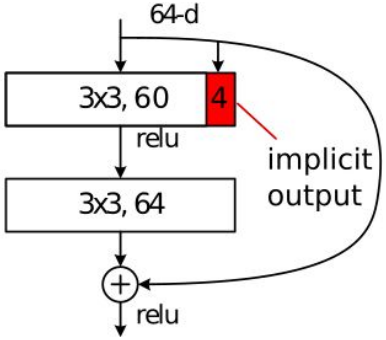




Composability: implicit output in neural architectures



Residual block: conv & relu



Residual block: stack(conv, implicit) & relu

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

arXiv:1512.03385v1 [cs.CV] 10 Dec 2015

Outlook

- (linear) implicit layers as flexible building blocks of neural networks
- scaling to large systems (Gauss-Seidel only a start: multigrid methods)
- expressivity of linear implicit layer & parametrization of $A(x)$

Thank you!

HELMHOLTZ AI | ARTIFICIAL INTELLIGENCE COOPERATION UNIT

Helmholtz-Zentrum hereon

Model-driven Machine Learning Group

http://m-dml.org

David Greenberg

Marcel Nonnenmacher

Tobias Machnitzki

Shivani Sharma

Vadim Zinchenko

Kubilay Demir