Make Everything Differential

Physically based Deep Learning

Hanming Hou

2024-08-12

Physics Group

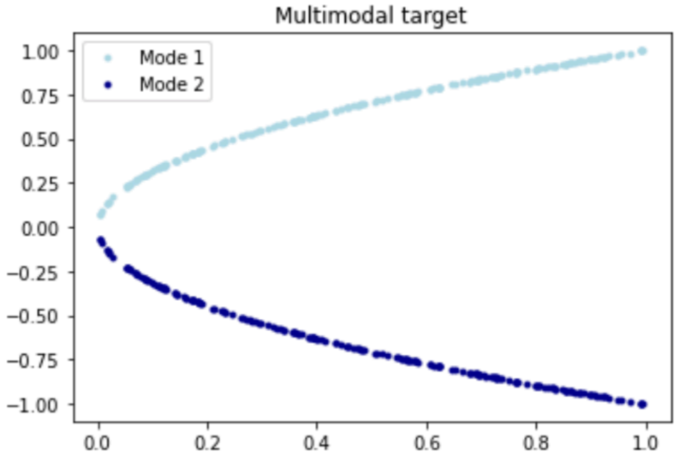
Outline

Outline

1.1 Start with an example

Painful $x = y^2$

If we have the parabola...

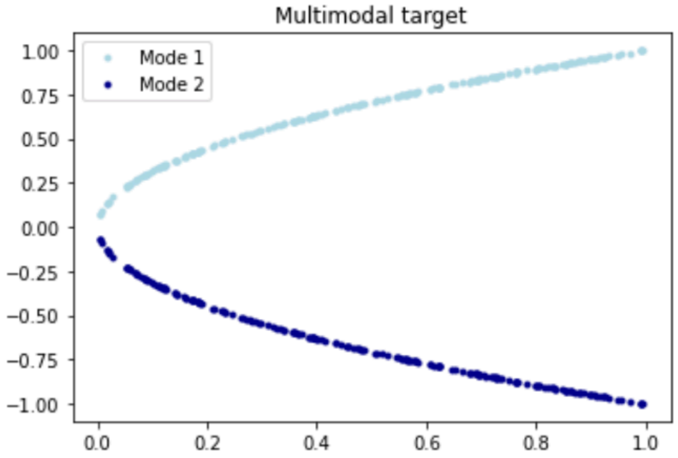


How to train the model for this function?

1.1 Start with an example

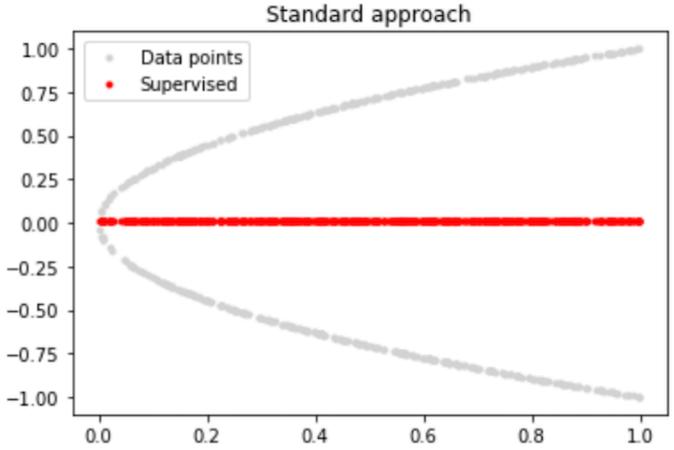
Painful $x = y^2$

If we have the parabola...



How to train the model for this function?

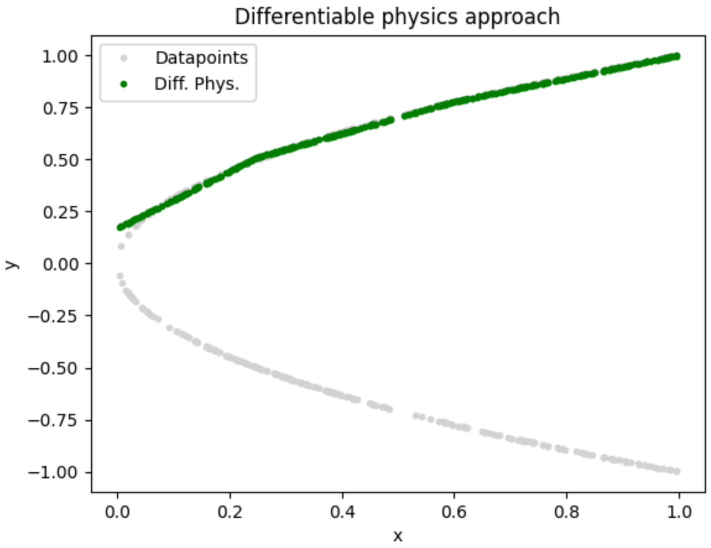
Random Select $x \in [0, 1]$ and $y = \pm\sqrt{x}$ to build the dataset, and train a MLP. For 200 points,



Obviously completely wrong!

Prediction?

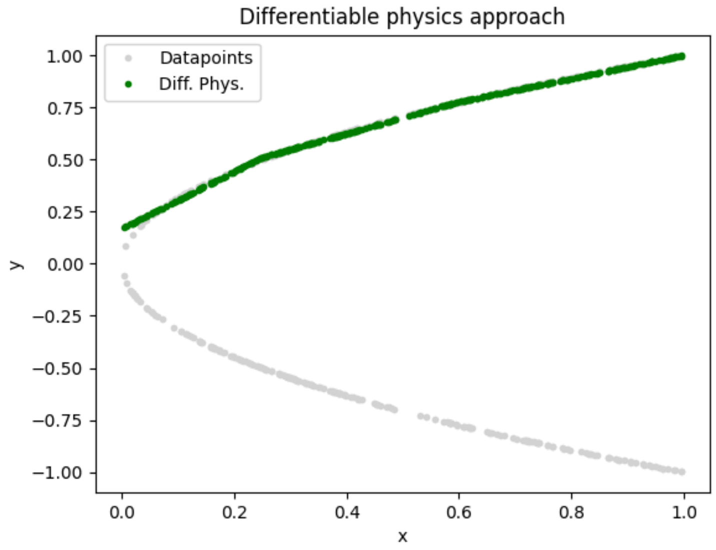
If we use the loss function $\|x^2_{\{\text{pred}\}} - x_{\{\text{real}\}}\|$ to train the network?



Looks much better…

Prediction?

If we use the loss function $\|x^2_{\{\text{pred}\}} - x_{\{\text{real}\}}\|$ to train the network?



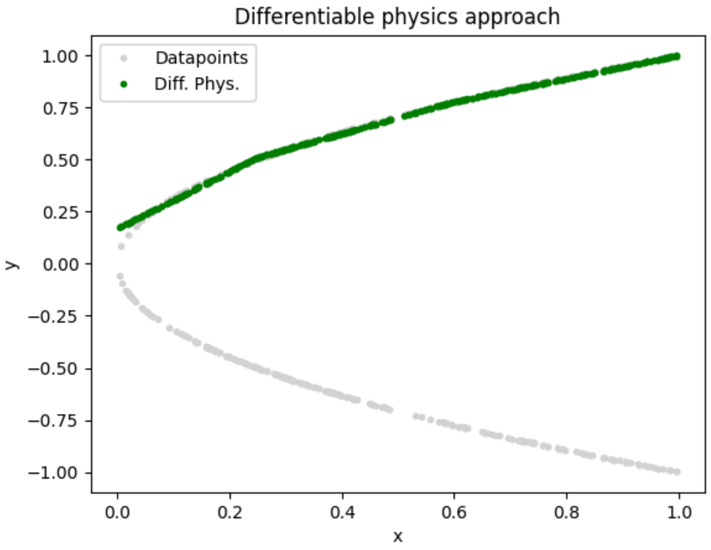
Looks much better…

Discussion: Real function: $f^* : X \mapsto Y$, hard to implement

However, if we know info about $P : Y \mapsto X$, we can use the form to guide the traing of f^*, instead of surpvised training!

Prediction?

If we use the loss function $\|x^2_{\{\text{pred}\}} - x_{\{\text{real}\}}\|$ to train the network?



Looks much better…

Discussion: Real function: $f^* : X \mapsto Y$, hard to implement

However, if we know info about $P : Y \mapsto X$, we can use the form to guide the traing of f^*, instead of surpvised training!

Question

Can we predict another half of the parabola? Why the region with near zero is typically still off?

code: teaser.py

1.2 Model Equation

Burgers Equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial t^2}$$

An advection term and a diffusion term! Easy to forward simulation.

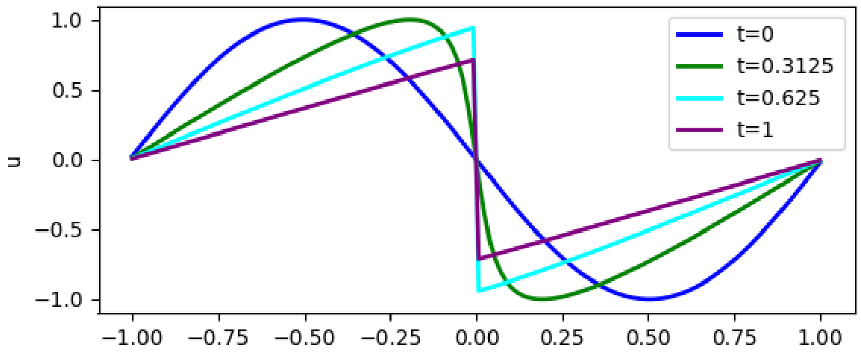
$$u(x - u_i^t \Delta t) = \nu \frac{u_{i-1}^t - 2u_i^t + u_{i+1}^t}{\Delta t^2}$$

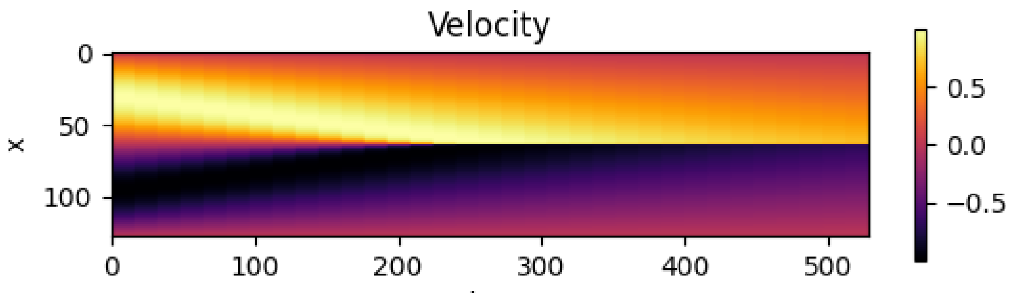
Simply implemented:

```
# based on PhiFlow package, 1 order
for i in range(STEPS):
    v1 = diffuse.explicit(v2, NU, DT)        # Diffuse term
    v2 = advect.semi_lagrangian(v1, v1, DT) # Advection term
```

Burgers Equation (Cont.)

Result of visualization:





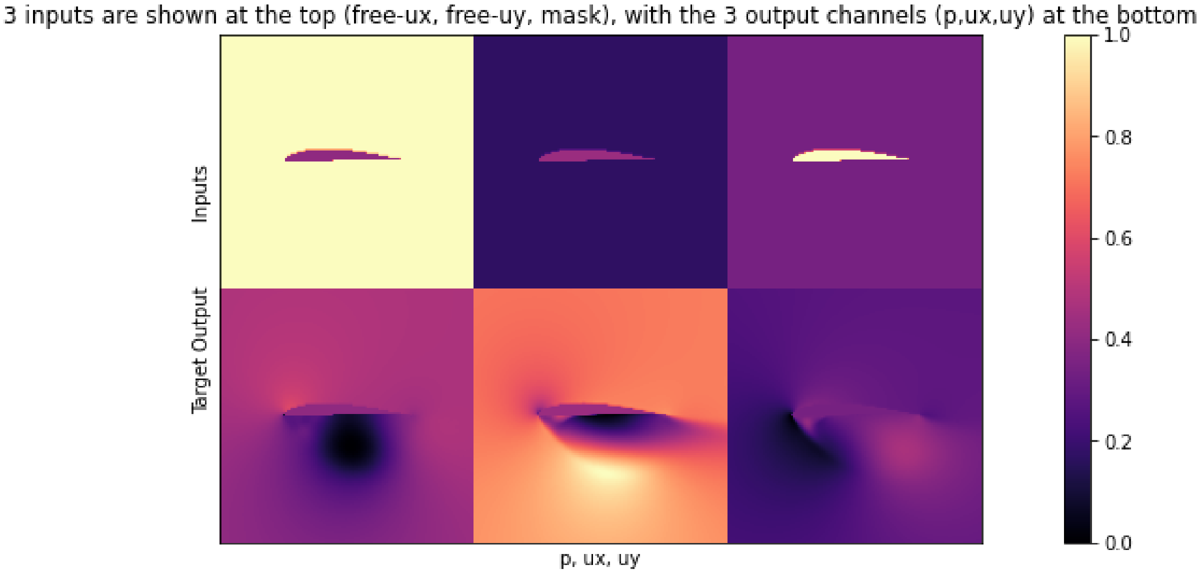
The shock developing in the center of our domain, which forms from the collision of the two initial velocity "bumps", the positive one on left (moving right) and the negative one right of the center (moving left).

code: burgers1d_fwd.py

1.3 Surpvised Training

Build a big dataset!

Used in some 2D problem, such as RANS turbulence



[Chen et al. 2021]

Surpvised Training (Cont.)

Pros and Cons:

Pros:	**Cons**:
• Fast training • Stable • Simple	• Large quantity of data • Hard to generalization • Low accuracy • Hard to interact with solver

Another challenge: Handle the unstructured mesh?

Surpvised Training (Cont.)

Pros and Cons:

Pros:	**Cons**:
• Fast training • Stable • Simple	• Large quantity of data • Hard to generalization • Low accuracy • Hard to interact with solver

Another challenge: Handle the unstructured mesh?

Use a GNN to maintain the topology of mesh.

1.4 Physical Loss

For a PDE of $\boldsymbol{u}(\boldsymbol{x}, t)$, we express it in term of

$$\boldsymbol{u}_t = \mathcal{F}(\boldsymbol{u}_x, \boldsymbol{u}_{xx}, \boldsymbol{u}_{xxx})$$

The residual R should be equal to zero for the accurate solution:

$$R = \boldsymbol{u}_t - \mathcal{F}(\boldsymbol{u}_x, \boldsymbol{u}_{xx}, \boldsymbol{u}_{xxx})$$

The training objective becomes

$$\arg\min_{\theta} \sum_i \alpha_0 (f(x_i; \theta) - y_i^*)^2 + \alpha_1 R(x_i)$$

α_0, α_1 are hyperparameters.

- conventional L2 loss term: approximate the traning sample
- physical residual term: satisfy the PDE (note that nullspace $\neq$ target solution)

Outline

2.1 Pysical Inform

For a PDE defined in region Ω, the Residual may includes 4 terms:

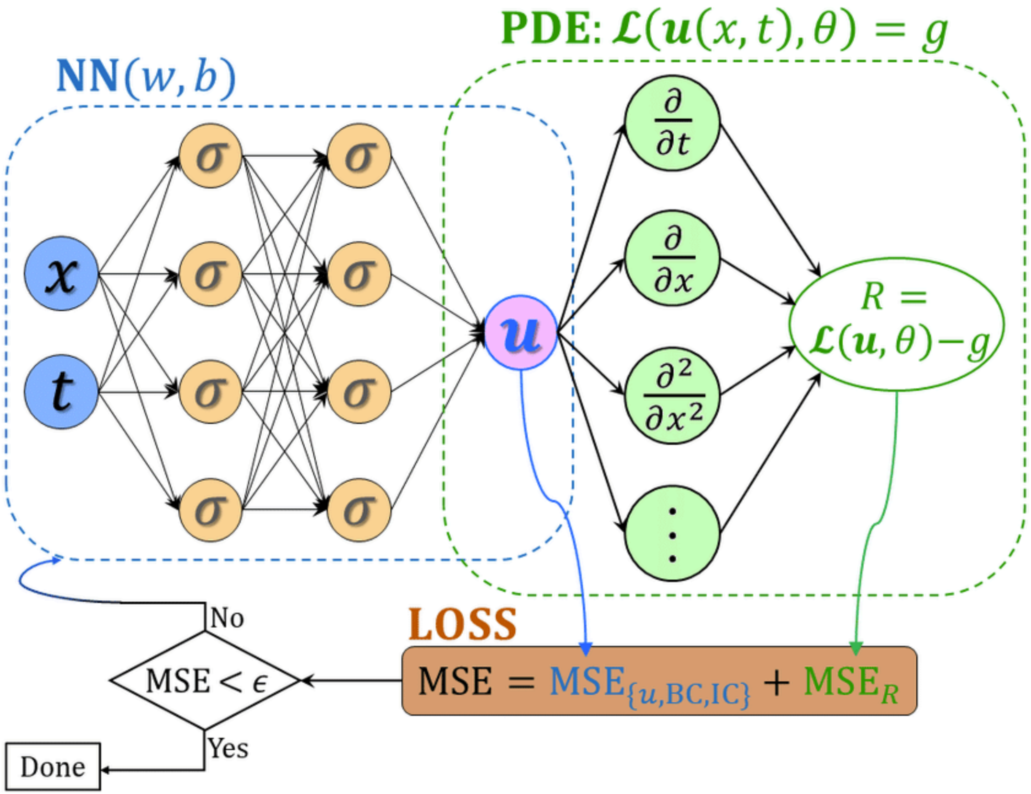
- $\mathcal{L}_{\text{PDE}}$: loss in $\Omega / \partial\Omega$
- $\mathcal{L}_{\text{BC}}$: loss in $\partial\Omega$
- $\mathcal{L}_{\text{IC}}$: loss in initial condition
- $\mathcal{L}_{\text{DATA}}$: loss upon real data

Mix these terms:

$$\mathcal{L}(\theta) = w_f \mathcal{L}_{\text{PDE}}(\theta; \mathcal{T}_f) + w_b \mathcal{L}_{\text{BC}}(\theta; \mathcal{T}_b) + w_i \mathcal{L}_{\text{IC}}(\theta; \mathcal{T}_i) + w_d \mathcal{L}_{\text{DATA}}(\theta; \mathcal{T}_d)$$

Which $\mathcal{T}_f, \mathcal{T}_b, \mathcal{T}_i, \mathcal{T}_d$ means sampled data.

Physics inform (cont.)



[Meng et al. 2019]

Structure of PINN:

- connected layers
- different category of LOSS terms
- gradient is same to GD process

2.2 PI in Burgers 1-D

For burgers equation, we can use information from

- initial condition: $u = \sin(\pi x)$
- boundary condition: open boundary, $u_{+1} = u_{-1} = 0$
- physics term: $u_t + u u_x - \nu u_{xx} = 0$

Network definition: 8 hidden layers

```
class PINN(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(2, 16)); self.layers.append(nn.Tanh())
        for _ in range(8):
            self.layers.append(nn.Linear(16, 16)) ;self.layers.append(nn.Tanh())
        self.layers.append(nn.Linear(16, 1))
```

PI in Burgers 1-D (Cont.)

IC : generate data at time 0

```
def initial_cond(num):
    x = np.random.uniform(-1, 1, (num,))
    t = np.zeros_like(x)
    u = -np.sin(np.pi * x)
    return x, t, u
```

PI in Burgers 1-D (Cont.)

IC : generate data at time 0

```
def initial_cond(num):
    x = np.random.uniform(-1, 1, (num,))
    t = np.zeros_like(x)
    u = -np.sin(np.pi * x)
    return x, t, u
```

BC : generate data at boundary

```
def open_bound(num):
    t = np.random.uniform(0, 1, (num,))
    x = np.concatenate([np.zeros((num // 2,)) + 1, np.zeros((num // 2,)) - 1])
    u = np.zeros(num)
    return x, t, u
```

PI in Burgers 1-D (Cont.)

Residual : calculate the residual of PDE, use torch.autograd

```
def residual(u: torch.Tensor, x: torch.Tensor, t: torch.Tensor):
    u_x = torch.autograd.grad(
        inputs=x,  outputs=u,
        grad_outputs=torch.ones_like(u),
        retain_graph=True,  create_graph=True)[0]
    u_t = torch.autograd.grad(
        inputs=t,  outputs=u,
        grad_outputs=torch.ones_like(u),
        retain_graph=True, create_graph=True)[0]
    u_xx = torch.autograd.grad(
        inputs=x, outputs=u_x,
        grad_outputs=torch.ones_like(u_x),
        retain_graph=True, create_graph=True)[0]
    return u_t, u_x, u_xx
```

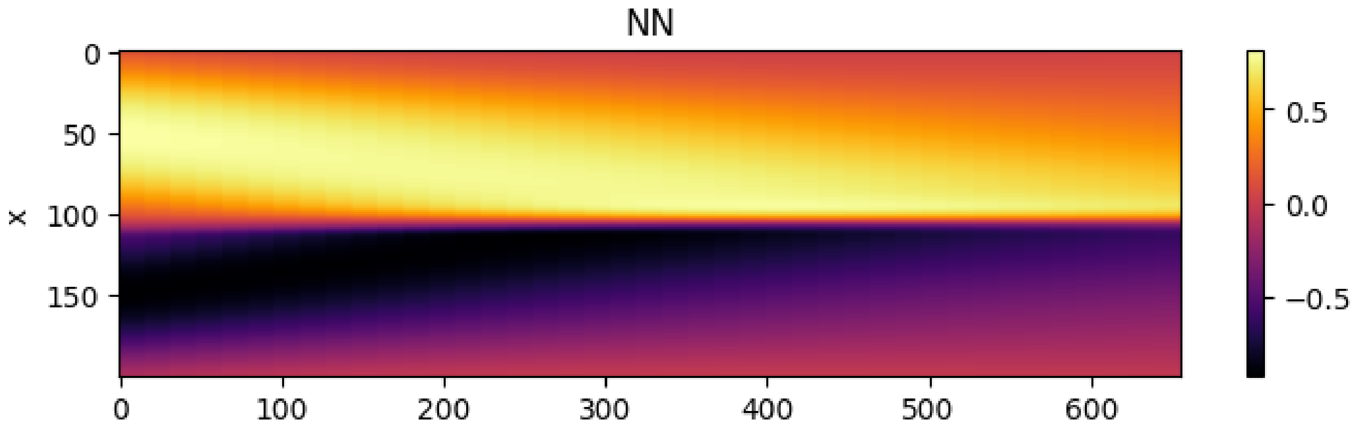
PI in Burgers 1-D (Cont.)

Loss Calc : calculate the blended loss

```
def loss():
    # ... generate IC and BC, store in x_def, t_def, u_def
    u_imp = model(x_def, t_def)
    loss_def = l2loss(u_def, u_imp[:, 0])
    # ...
    # generate grid points inner, store in g_x, g_t
    u_res = model(g_x, g_t)[:, 0]
    u_t, u_x, u_xx = residual(u_res, g_x, g_t) # residual term
    loss_inner = l2loss(u_t + u_res * u_x, NU * u_xx)
    loss_sum = loss_def + ALPHA * loss_inner
    loss_sum.backward()
    return loss_sum
```

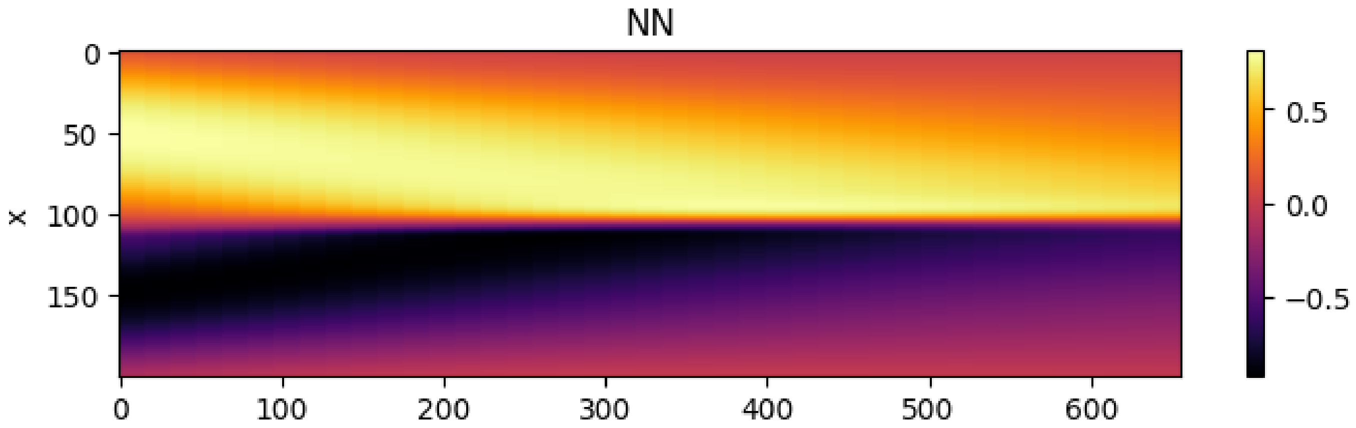
Code: *burgers1d_pinn.py*

PI in Burgers 1-D (Cont.)



Result of PINN

PI in Burgers 1-D (Cont.)



Result of PINN

If we know the state at time t, we can use it to predict IC…

Backpropagation process!

Besides, we can use PINN to estimate parameters.

2.3 Discussion

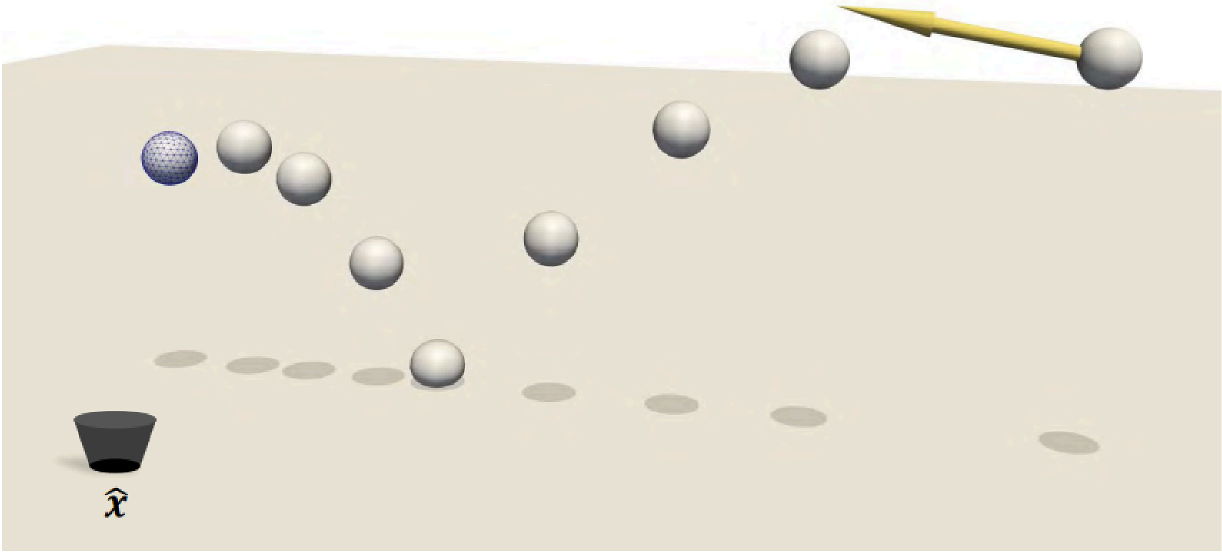
Pros and Cons:

Pros:	**Cons**:
• Physical model	• Really, really slow
• Easy to handle derivatives	• Physical constraints are enforced softly
• Easy to implement	• Incompatible with tradictional methods
• Meshless	• A network for a specific condition

Outline

3.1 Basics

A Control Problem



An optimization problem

- Target function: O
- Arguments: $\boldsymbol{p}$
- Forward process: $\boldsymbol{x}(\boldsymbol{p})$

Then, the optimization problem is

$$\arg\min_{\boldsymbol{p}} O(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p})$$

Calculate the gradient:

$$\frac{dO}{d\boldsymbol{p}} = \frac{\partial O}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial O}{\partial \boldsymbol{p}}$$

Calculate the Jacobian

Find the conserved equation in dynamics perspective:

$$\boldsymbol{M}\boldsymbol{x}'' = F(\boldsymbol{x}, \boldsymbol{p})$$

Use the dynamics theory, build the implicit form:

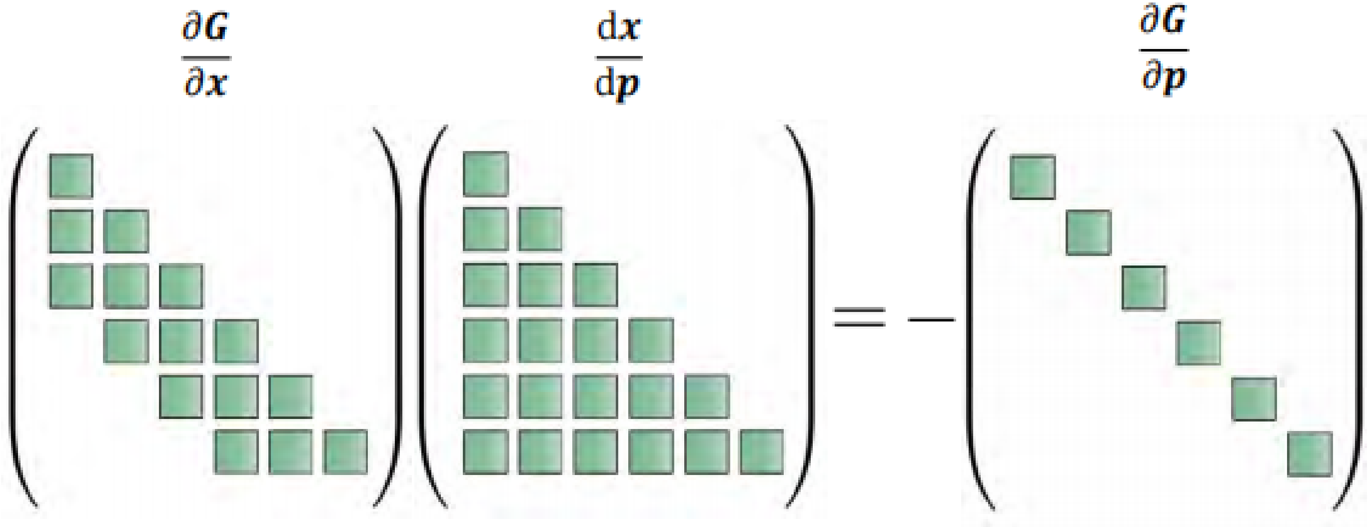
$$\boldsymbol{M}\boldsymbol{x}''_k = F(\boldsymbol{x}_k, \boldsymbol{p}), \boldsymbol{x}''_k \approx \frac{\boldsymbol{x}_k - 2\boldsymbol{x}_{k-1} + \boldsymbol{x}_{k-2}}{h^2}$$

Construct $\boldsymbol{G}(\boldsymbol{x}(\boldsymbol{p}), \boldsymbol{p}) = \boldsymbol{M}\boldsymbol{x}'' - \boldsymbol{F} = \boldsymbol{0}$, then $\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = 0$.

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}} = 0$$

$$\boxed{\frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}}$$

Calculate the Jacobian (Cont.)



- $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}$: Residual upon at time step i change wrt system configuration at time step j
- $\frac{d\boldsymbol{x}}{d\boldsymbol{p}}$: System's configuration at time step i change with respect to the jth input parameter
- $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$: The residual at time step i change wrt the jth input parameter p

Optimization step

Note that $\frac{\partial O}{\partial \boldsymbol{x}} \frac{d\boldsymbol{x}}{d\boldsymbol{p}} = -\frac{\partial O}{\partial \boldsymbol{x}} \left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}}\right)^{-1} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}$, we can solve the system $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}} \boldsymbol{g} = \frac{\partial O}{\partial \boldsymbol{x}}$

Gradient Descent

Until Convergence:

Solve $\boldsymbol{g}$ for $\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}} \boldsymbol{g} = \frac{\partial O}{\partial \boldsymbol{x}}$

$\Delta p = \boldsymbol{g} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}} - \frac{\partial O}{\partial \boldsymbol{p}}$

$\alpha = \text{line\_search}(\Delta \boldsymbol{p})$

$\boldsymbol{p} = \boldsymbol{p} + \alpha \Delta \boldsymbol{p}$

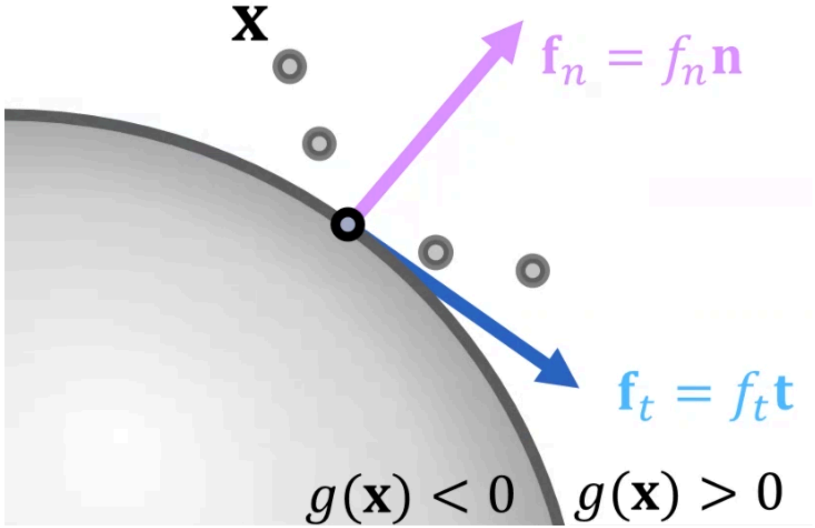
$\boldsymbol{x} = \text{simulate}(\boldsymbol{x}, \boldsymbol{p})$

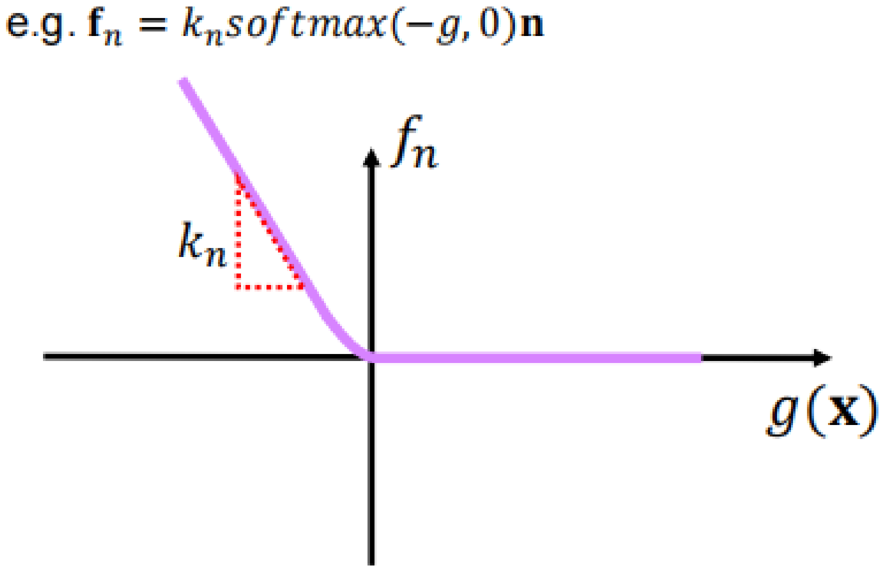
End

We can use L-BFGS or generalized Gauss-Newton method for acceleration, see [Zehnder et al. 2021] for details.

Solve Contact

If the SDF function of surface is $g(\boldsymbol{x})$, then normal $\boldsymbol{n} = \nabla_x g$, normal force $\boldsymbol{f}_n = f_n \boldsymbol{n}$.

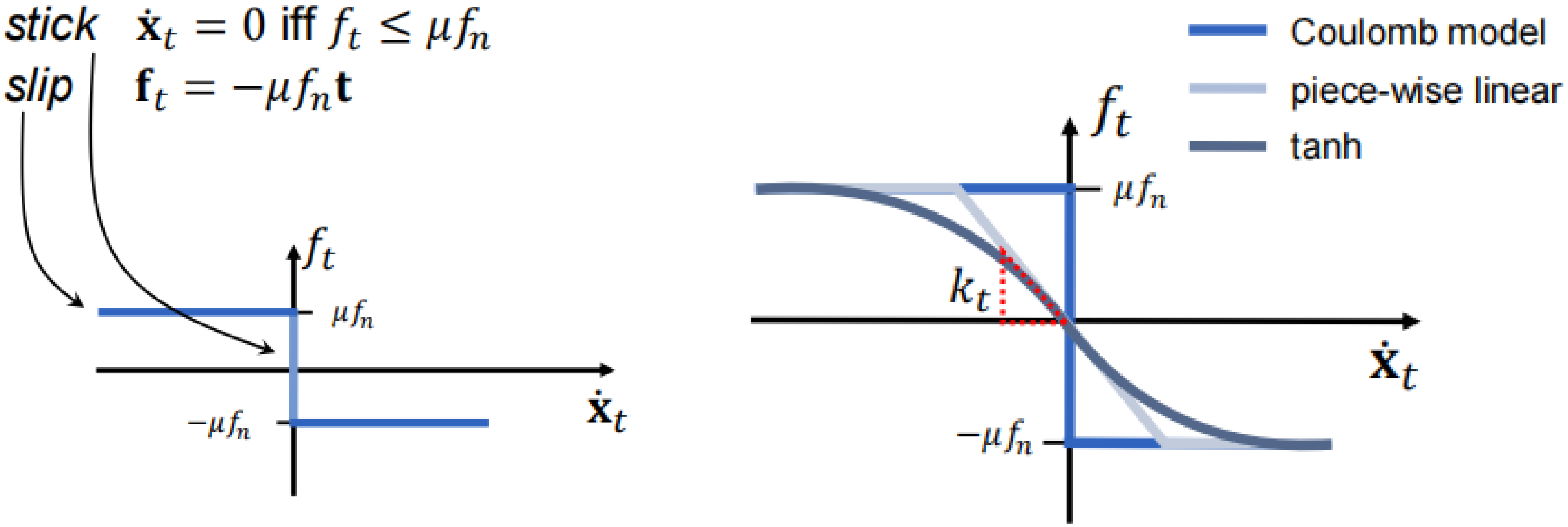




Make it C^2 continuity!

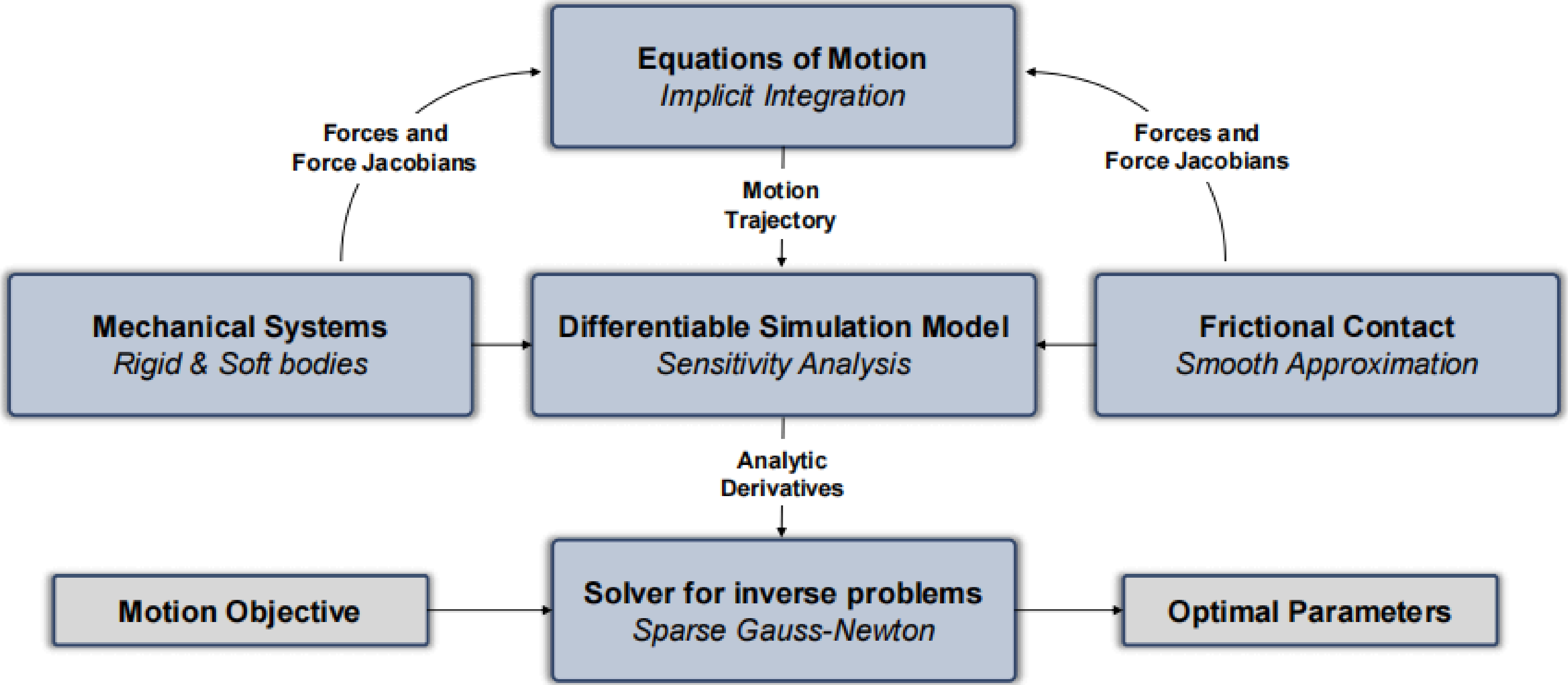
Solve Contact (Cont.)

The situation is more complex for tangent force...



Make it C^2 continuity!

DP in a nutshell



3.2 Autograd

Single variable

For a discreted system $\boldsymbol{x}^{t+1} = f(\boldsymbol{x}^t)$, we have

$$s(\boldsymbol{x}^{t+3}) = s(f(f(f(\boldsymbol{x}^t))))$$

Where s is loss function. Apply chain rule,

$$\frac{\partial s}{\partial \boldsymbol{x}^t} = \frac{\partial f}{\partial \boldsymbol{x}^t} \cdot \frac{\partial f}{\partial \boldsymbol{x}^{t+1}} \cdot \frac{\partial f}{\partial \boldsymbol{x}^{t+2}} \cdot \frac{\partial s}{\partial \boldsymbol{x}^{t+3}}$$

Let $x^*(t) = \left(\frac{\partial s}{\partial \boldsymbol{x}}\right)^T$ (**adjoint vector**), we find

$$\boxed{x^*(t) = \left(\frac{\partial f}{\partial \boldsymbol{x}^t}\right)^T x^*(t+1)}$$

Function

Take account of one step $z = f(x, y)$

$$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial z}\frac{\partial f}{\partial x}, \frac{\partial s}{\partial z}\frac{\partial s}{\partial f}\frac{\partial f}{\partial y}$$

Therefore, we find that

$$(x^*, y^*) = \left(z^* \frac{\partial f}{\partial x}, z^* \frac{\partial f}{\partial y}\right)$$

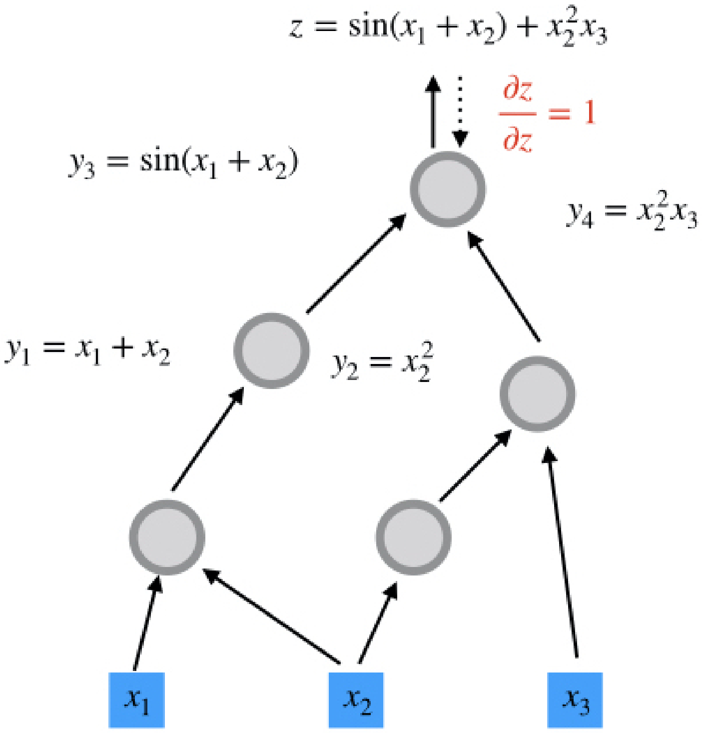
Naturally, we define **adjoint function** f:

$$\boxed{f^*(x, y, z^*) = \left(\left(\frac{\partial f}{\partial x}\right)^T z^*, \left(\frac{\partial f}{\partial y}\right)^T z^*\right)}$$

which propagate the result z^* back to x^* and y^*

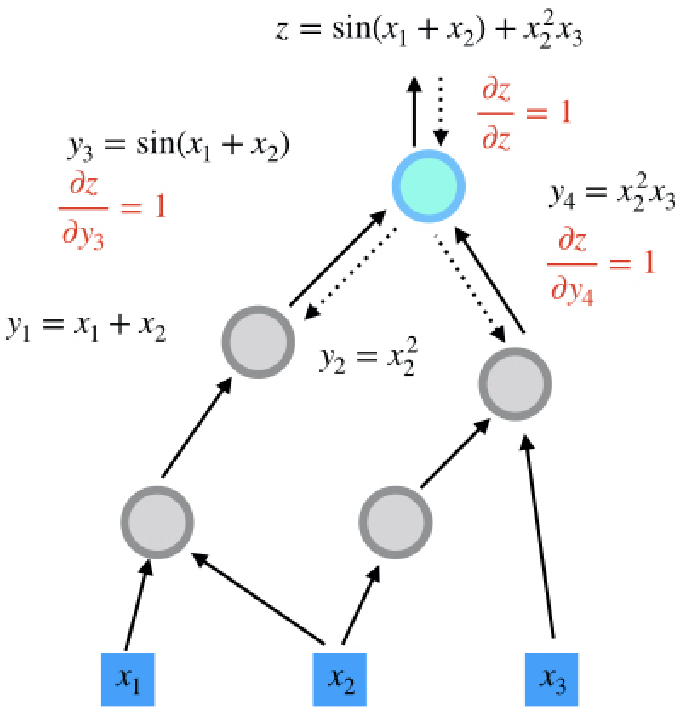
Build a graph

For $z = \sin(x_1 + x_2) + x_2^2 x_3$, we have such a graph:



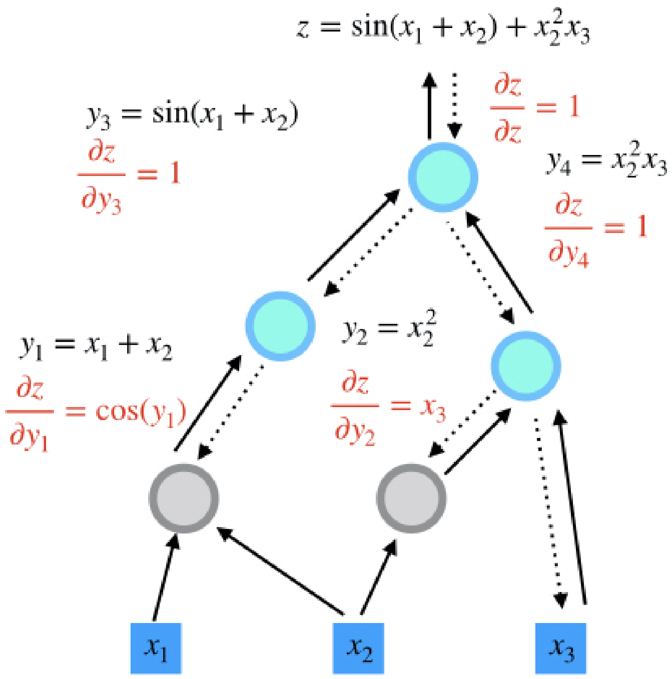
Build a graph

For $z = \sin(x_1 + x_2) + x_2^2 x_3$, we have such a graph:



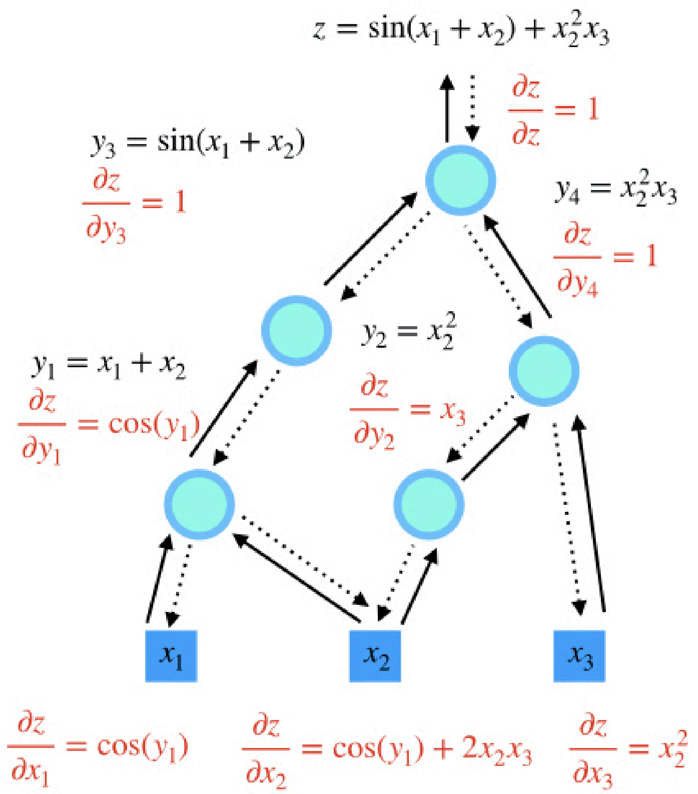
Build a graph

For $z = \sin(x_1 + x_2) + x_2^2 x_3$, we have such a graph:



Build a graph

For $z = \sin(x_1 + x_2) + x_2^2 x_3$, we have such a graph:



3.3 DP in Burgers 1-D

We directly implement the Burgers 1-D in PhiFlow.

```
# loss function
def loss_function(velocity):
  velocities = [velocity]
  # forward step
  for time_step in range(STEPS):
      v1 = diffuse.explicit(1.0*velocities[-1], NU, DT, substeps=1)
      v2 = advect.semi_lagrangian(v1, v1, DT)
      velocities.append(v2)
  # l2 loss
  loss = field.l2_loss(velocities[16] - SOLUTION_T16)*2./N # MSE
  return loss, velocities
```

DP in Burgers 1-D (Cont.)

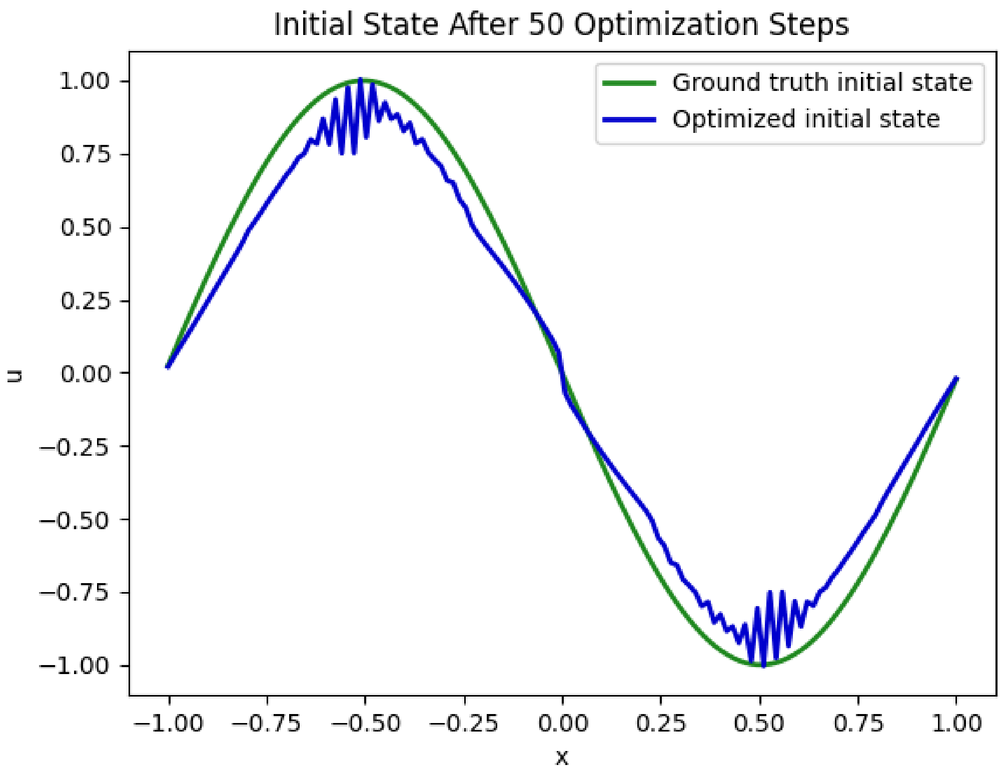
```
# call the autograd model in pytorch
gradient_function = math.gradient(loss_function)
grads=[]
for optim_step in range(0,50):
  # calculate gradient
  (loss,velocities), grad = gradient_function(velocity)
  # gradient descent
  velocity = velocity - LR * grad[0]
```

Very easy!

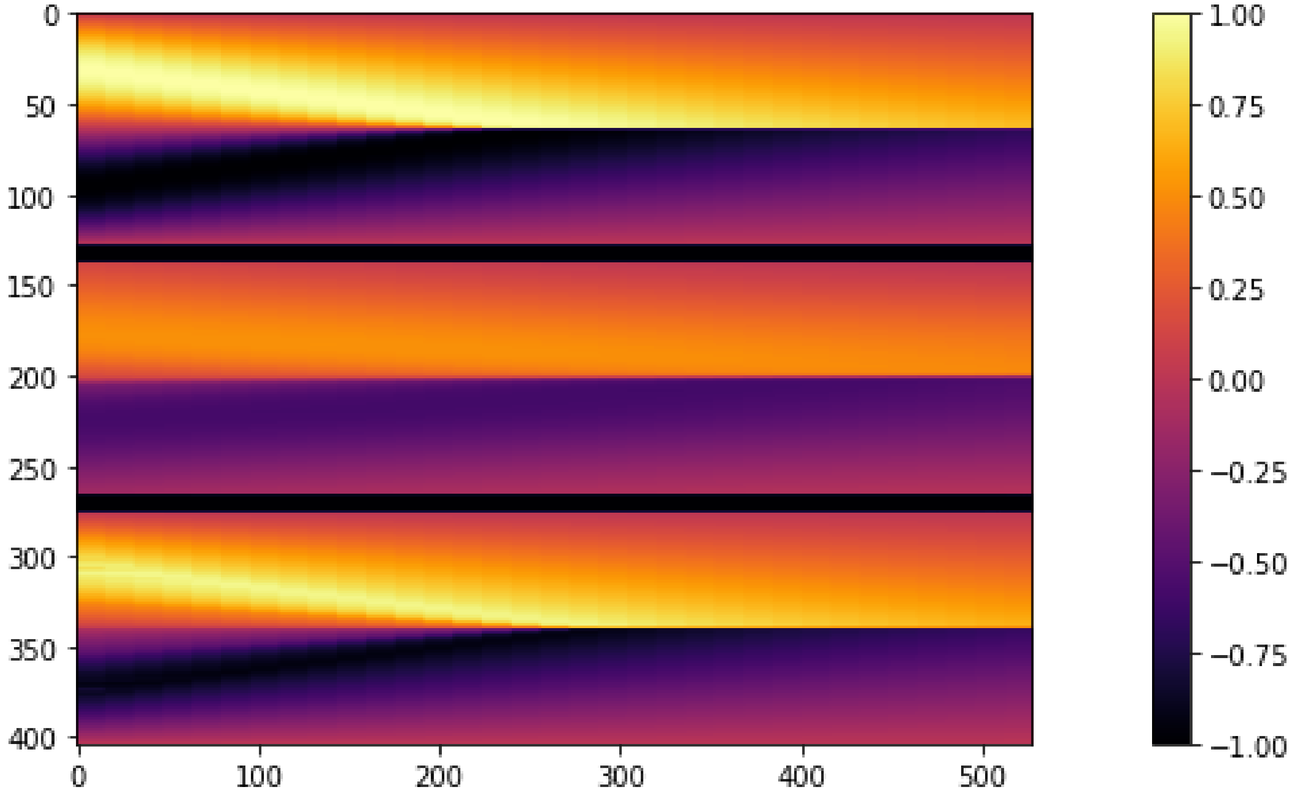
code: *burgers1d_dp.py*

DP in Burgers 1-D (Cont.)

Result over ground truth:



3.4 Discussion



Ground truth(top), PINN (middle) and DP (bottom)

Pros and Cons

- Select a suitable discretization
- Efficiency depends on time steps

Pros and Cons:

Pros:	**Cons**:
• Base on physical model	• Require mesh
• Use existing numerical methods	• Implement complicatedly
• Evaluate efficiently	• Need to choose a suitable discretization

Outline

4.1 Integration Guide

Overview

Recall the $x = y^2$ sample...

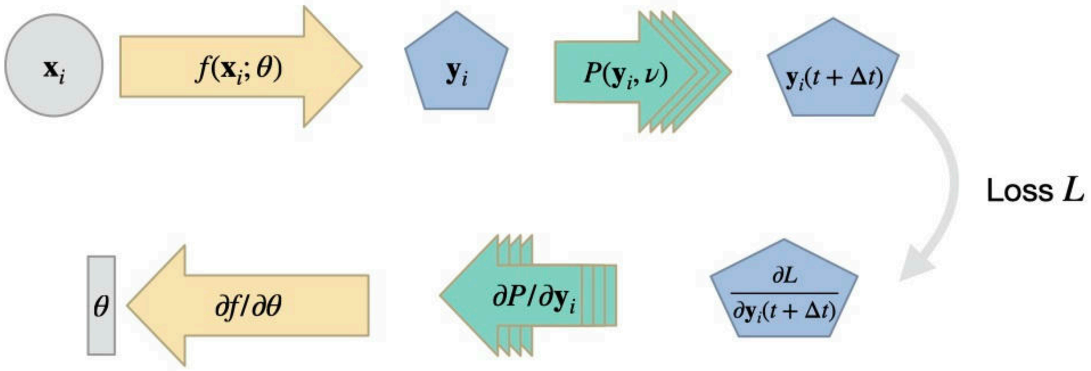
How to add DP process into NN?

4.1 Integration Guide

Overview

Recall the $x = y^2$ sample...

How to add DP process into NN?



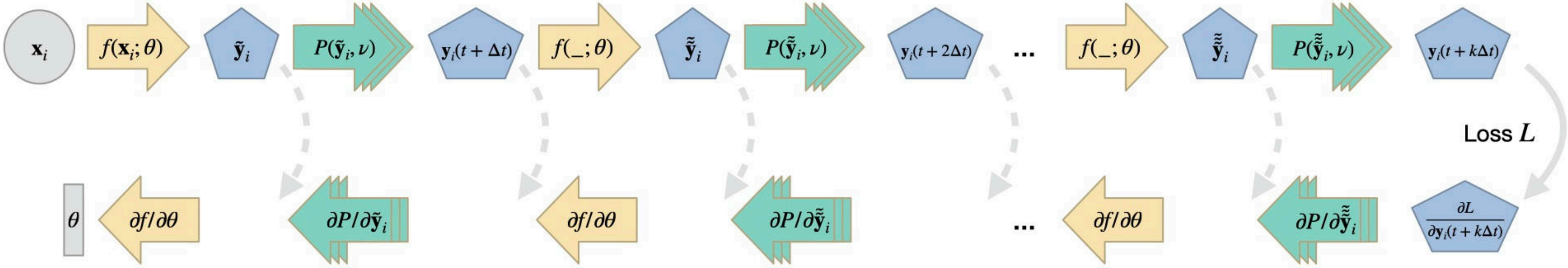
Forward pass: $y_i \xrightarrow{\text{NN}} \tilde{y}_i \xrightarrow{\text{solver}} y_{i+n}$

Backward pass: $L_i \xleftarrow{\text{NN}} \tilde{L}_i \xleftarrow{\text{solver}} L_{i+n}$

Overview

Recall the $x = y^2$ sample...

How to add DP process into NN?



Forward pass: $y_i \xrightarrow{\text{NN}} \tilde{y}_i \xrightarrow{\text{solver}} y_{i+1}$

Backward pass: $L_i \xleftarrow{\text{NN}} \tilde{L}_i \xleftarrow{\text{solver}} L_{i+1}$

NN as **correction**

Loss Term

$$\frac{\partial L}{\partial \theta} = \sum_{m=1}^{N} \left(\frac{\partial L}{\partial x_N} \frac{\partial x_N}{\partial x_{N-1}} \frac{\partial x_{N-1}}{\partial x_{N-2}} \dots \frac{\partial x_{m+1}}{\partial x_m} \frac{\partial x_m}{\partial \tilde{x}_{m-1}} \frac{\partial \tilde{x}_{m-1}}{\partial \theta} \right)$$

- iteration: from step 1 to N
- final step: final loss in time step N
- chain prop: iteration from N to m
- current step: current step of NN and DP

Loss Term

$$\frac{\partial L}{\partial \theta} = \sum_{m=1}^{N} \left(\frac{\partial L}{\partial x_N} \frac{\partial x_N}{\partial x_{N-1}} \frac{\partial x_{N-1}}{\partial x_{N-2}} \dots \frac{\partial x_{m+1}}{\partial x_m} \frac{\partial x_m}{\partial \tilde{x}_{m-1}} \frac{\partial \tilde{x}_{m-1}}{\partial \theta} \right)$$

- iteration: from step 1 to N
- final step: final loss in time step N
- chain prop: iteration from N to m
- current step: current step of NN and DP

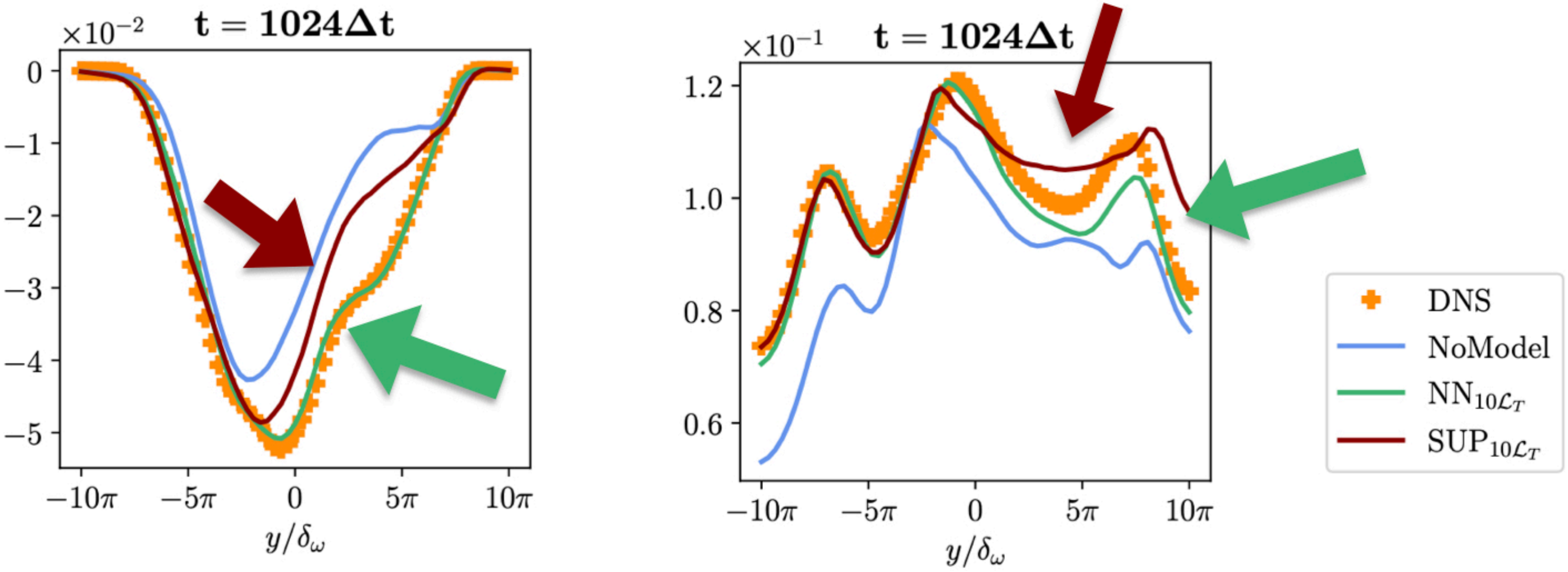
Risk:

- error increase exponentially over iterations
- later time steps have stronger influence

Cause **vanishing** and **exploding** gradient problem

Evaluation

Turbulence example: Reynolds stresses (L) and Turbulence kinetic energy (R)

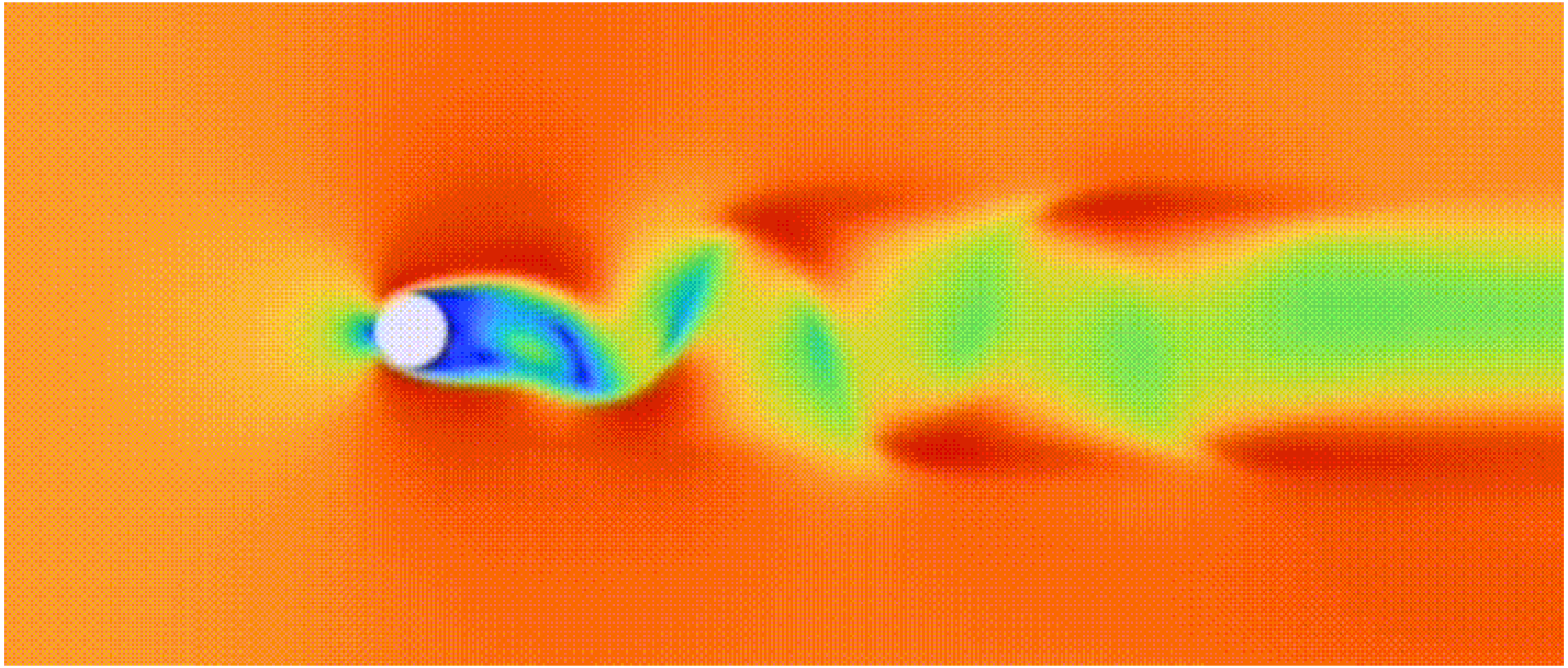


[List et al. 2022]

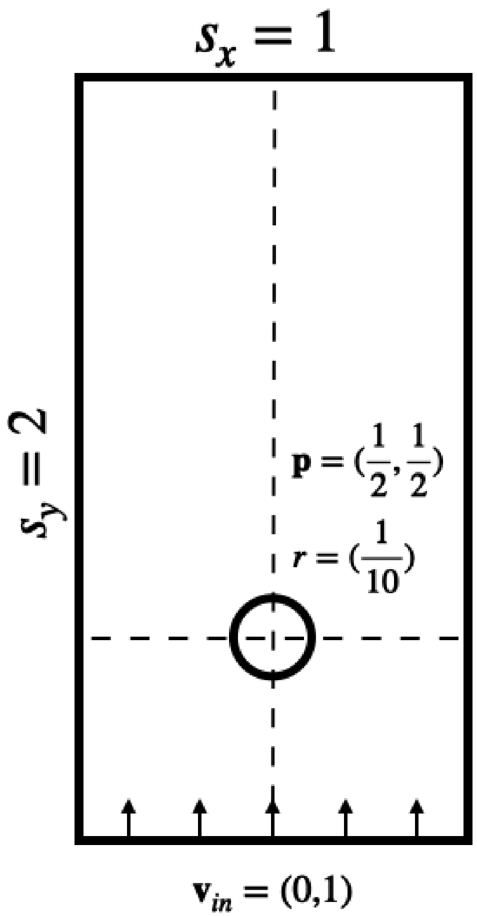
Ground truth Solver NN without DP NN with DP

4.2 Error Analysis

Kármán Vortex Street



Kármán Vortex Street (cont.)



Governing equation: 2D incompressible N-S equation

$$\begin{cases} \boldsymbol{u}_t + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\rho} \nabla p + \frac{1}{Re} \nabla^2 u \\ \nabla \cdot \boldsymbol{u} = 0 \end{cases}$$

Compute region:

- Box (1×2)
- Barrier : cylinder, radius = 0.1

Boundary condition:

- top: outflow
- bottom: inflow
- left / right: stick

Forward simulation

code: *numerror.py* . The code work in tensorflow 2 and phiflow 2.2

```
def step(density, velocity, Re, res, dt=1.0):
  # viscosity, res is the resolution
  velocity=phi.flow.diffuse.explicit(field=velocity, diffusivity=1.0 / Re * dt
* res * res, dt = dt)
  # inflow boundary conditions, set the velocity boundary
  velocity = velocity*(1.0 - self.vel_BcMask) + self.vel_BcMask * (1,0)
  # advection for density and velocity
  density = advect.semi_lagrangian(density+self.inflow, velocity, dt=dt)
  velocity = advect.semi_lagrangian(velocity, velocity, dt=dt)
  # mass conservation (pressure solve)
  velocity, pressure = fluid.make_incompressible(velocity, self.obstacles)
  return [density, velocity]
```

Reference [Anderson 2012] for more details.

Network

a simplified structure

```
def network_small(inputs_dict):
  l_input = keras.layers.Input(dict(shape = (32,64,3)))
   block_0 = keras.layers.Conv2D(filters=32, kernel_size=5, padding='same')
(l_input)
  block_0 = keras.layers.LeakyReLU()(block_0)
   l_conv1 = keras.layers.Conv2D(filters=32, kernel_size=5, padding='same')
(block_0)
  l_conv1 = keras.layers.LeakyReLU()(l_conv1)
   l_conv2 = keras.layers.Conv2D(filters=32, kernel_size=5, padding='same')
(l_conv1)
  block_1 = keras.layers.LeakyReLU()(l_conv2)
  l_output = keras.layers.Conv2D(filters=2,  kernel_size=5, padding='same')
(block_1) # u, v
  return keras.models.Model(inputs=l_input, outputs=l_output)
```

Training step

```
def training_step(dens_gt, vel_gt, Re):
  with tf.GradientTape() as tape:
    prediction, correction = [ [dens_gt[0],vel_gt[0]] ], [0]
    # pred: solver result, correction: NN result
    for i in range(msteps):
      prediction += [
          simulator.step( density_in=prediction[-1][0],
              velocity_in=prediction[-1][1], re=Re, res=source_res[1])]
      # prediction: [[density1, velocity1], [density2, velocity2], ...]

      model_input = to_keras(prediction[-1], Re)
      model_input = normalize(model_input) # [u, v, Re], normalized
      model_out = network(model_input, training=True) # correction
      model_out = denormalize(model_out) # [u, v], unstretched
      correction += [ to_phiflow(model_out, domain) ]
      prediction[-1][1] = prediction[-1][1] + correction[-1]
```

Forward simulation for prediction
Precition[−1] stores last step

Training step

```
def training_step(dens_gt, vel_gt, Re):
  with tf.GradientTape() as tape:
    prediction, correction = [ [dens_gt[0],vel_gt[0]] ], [0]
    # pred: solver result, correction: NN result
    for i in range(msteps):
      prediction += [
```

Use the NN to correct the result for low resolution
Normalize : scalar tensor with STD

```
                                                       urce_res[1]))]
      # p                                                ocity2], ...]

      model_input = to_keras(prediction[-1], Re) # transfer data to keras
      model_input = normalize(model_input) # [u, v, Re], normalized
      model_out = network(model_input, training=True) # correction
      model_out = denormalize(model_out) # [u, v], unstretched
      correction += [ to_phiflow(model_out, domain) ]
      prediction[-1][1] = prediction[-1][1] + correction[-1]
```

Training step

```
def training_step(dens_gt, vel_gt, Re):
  with tf.GradientTape() as tape:
    prediction, correction = [ [dens_gt[0],vel_gt[0]] ], [0]
    # pred: solver result, correction: NN result
    for i in range(msteps):
      prediction += [
          simulator.step( density_in=prediction[-1][0],
              velocity_in=prediction[-1][1], re=Re, res=source_res[1])]
      # prediction: [[density1, velocity1], [density2, velocity2], ...]

      mod                                                  r data to keras
      mod                                                  ormalized
      mod                                                  rrection
      model_out = denormalize(model_out) # [u, v], unstretched
      correction += [ to_phiflow(model_out, domain) ]
      prediction[-1][1] = prediction[-1][1] + correction[-1]
```

Add the correction to prediction
note that predition[−1] means velocity at last step

Training step

```python
# evaluate loss
# prediction[i] comes up with [dens_gt, vel_gt]
loss_steps_x = [
  tf.nn.l2_loss(normalize(
    vel_gt[i].vector['x'] - prediction[i][1].vector['x']))
  for i in range(1,msteps+1)  ] # velocity x-axis
loss_steps_x_sum = tf.math.reduce_sum(loss_steps_x)
loss_steps_y = [
  tf.nn.l2_loss(normalize(
    vel_gt[i].vector['y'] - prediction[i][1].vector['y']))
  for i in range(1,msteps+1)  ] # velocity y-axis
loss_steps_y_sum = tf.math.reduce_sum(loss_steps_y)
loss = (loss_steps_x_sum + loss_steps_y_sum)/msteps
gradients = tape.gradient(loss, network.trainable_variables)
opt.apply_gradients(zip(gradients, network.trainable_variables))
return math.tensor(loss)
```

Loss of velocity x component

Training step

```
# evaluate loss
# prediction[i] comes up with [dens_gt, vel_gt]
loss_steps_x = [
  t
                              ction[i][1].vector['x']))
  f                              # velocity x-axis
loss_steps_x_sum = tf.math.reduce_sum(loss_steps_x)
loss_steps_y = [
  tf.nn.l2_loss(normalize(
    vel_gt[i].vector['y'] - prediction[i][1].vector['y']))
  for i in range(1,msteps+1)  ] # velocity y-axis
loss_steps_y_sum = tf.math.reduce_sum(loss_steps_y)
loss = (loss_steps_x_sum + loss_steps_y_sum)/msteps
gradients = tape.gradient(loss, network.trainable_variables)
opt.apply_gradients(zip(gradients, network.trainable_variables))
return math.tensor(loss)
```

Loss of velocity y component

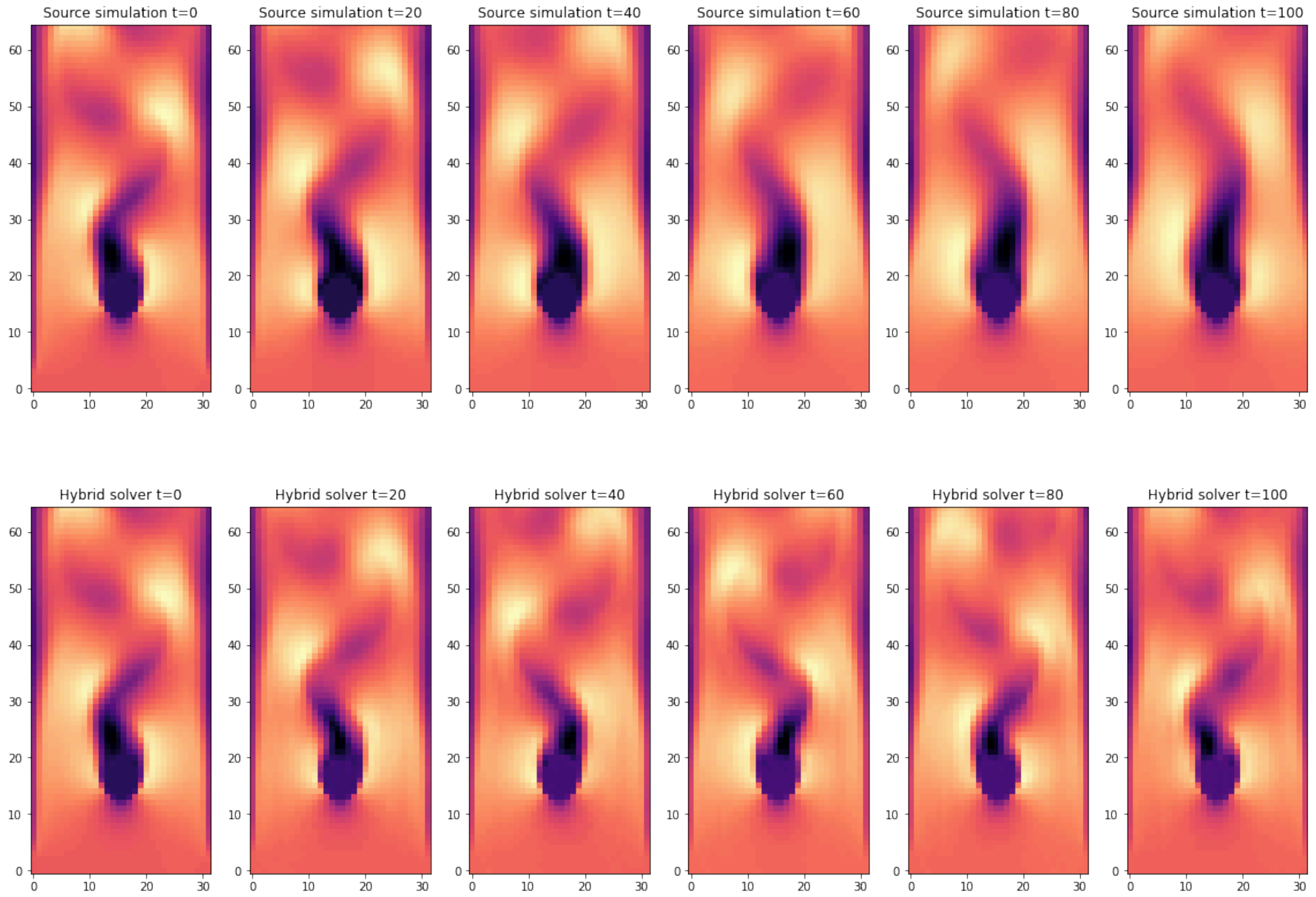
Then plus loss x and y

Training step

```
# evaluate loss
# prediction[i] comes up with [dens_gt, vel_gt]
loss_steps_x = [
  tf.nn.l2_loss(normalize(
    vel_gt[i].vector['x'] - prediction[i][1].vector['x']))
  for i in range(1,msteps+1)  ] # velocity x-axis
loss_steps_x_sum = tf.math.reduce_sum(loss_steps_x)
loss_steps_y = [
  tf.nn.l2_loss(normalize(
                   rediction[i][1].vector['y']))
  for i in range(1,msteps+1)  ] # velocity y-axis
loss_steps_y_sum = tf.math.reduce_sum(loss_steps_y)
loss = (loss_steps_x_sum + loss_steps_y_sum)/msteps
gradients = tape.gradient(loss, network.trainable_variables)
opt.apply_gradients(zip(gradients, network.trainable_variables))
return math.tensor(loss)
```

Delieve the gradient back

Result

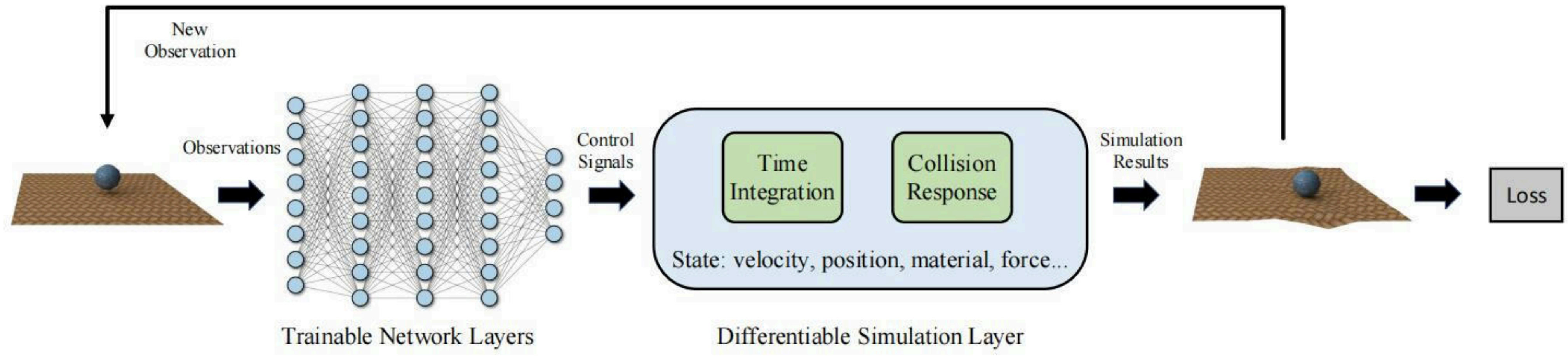


Direct solver

Hybrid solver

4.3 Applications

Pipeline



- Control
- Inverse problem
- Parameter estimation

Pipeline

We need to optimize simultaneously arguments of DP and NN! i.e.

$$\arg\min_{\theta} \sum_m \sum_i \left(f(x_{m,i};\theta) - y^*_{m,i}\right)^2$$

where i denotes spatial locations, m denotes states, f is our model, y^* is target states

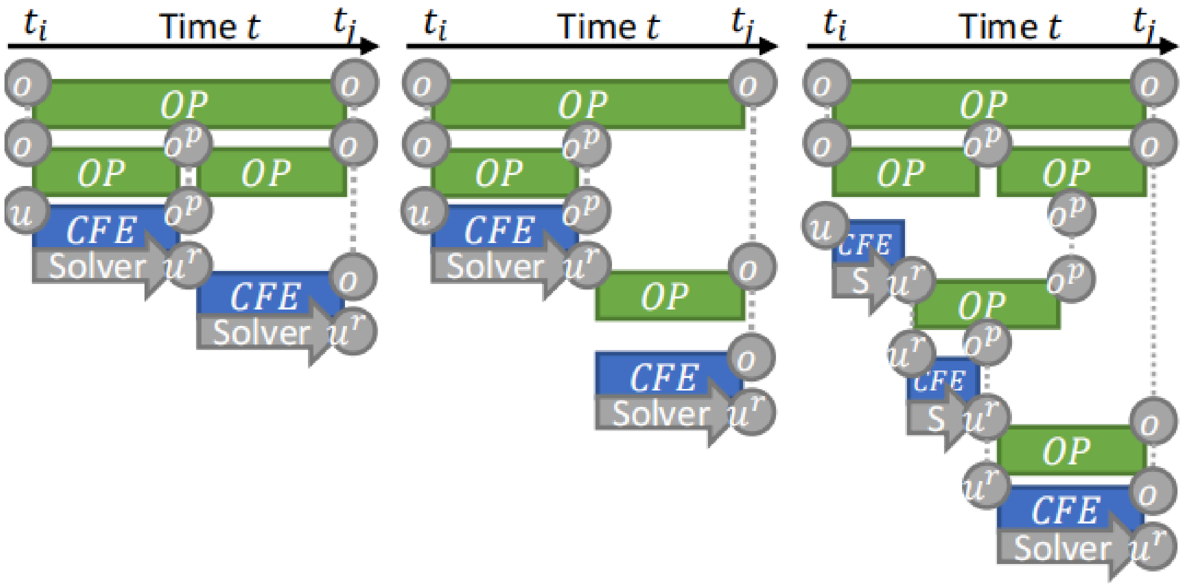
Therefore, we can build 2 networks:

- OP: predictor the target, $\mathrm{OP}(d, d^*)$
- CFE: act additively on field, $\boldsymbol{u} + \mathrm{CFE}(\boldsymbol{u}, d, \mathrm{OP}(d, d^*))$

$$\boldsymbol{u}_n, d_n = \mathcal{P}(\mathrm{CFE}(\mathcal{P}(\mathrm{CFE}(...\mathcal{P}(\mathrm{CFE}(\boldsymbol{u}_0, \mathrm{OP}(d_0, d^*)))...)))) = (\mathcal{P}\ \mathrm{CFE})^n(\boldsymbol{u}_0, d_0, \mathrm{OP}(d_0, d^*))$$

Optimization

Use the different execution sheme..



Control and operation recursively

4.4 Discussion

- seamlessly bring the two fields together
  - best numerical method
  - best traning model

Pros and Cons:

Pros:	**Cons**:
• Use physical model for discretization	• Not compatible with all simulators
• Control accuracy by selected methods	• Require heavier frame support
• Generalize easily via solver interactions	• Efficiency depends on implementation

Outline

Use in Inverse Problem

PPO Strategy: actor and critic network

- actor: policy function, return probability distribution for actions
  - $\pi(a; s, \theta)$: prob of choose action a from network param θ and state s
- critic: critic function, reward of action
  - $V(s; \varphi)$: expected reward to be received from state s

Apply to inverse problem?

Use in Inverse Problem

PPO Strategy: actor and critic network

- actor: policy function, return probability distribution for actions
  - $\pi(a; s, \theta)$: prob of choose action a from network param θ and state s
- critic: critic function, reward of action
  - $V(s; \varphi)$: expected reward to be received from state s

Apply to inverse problem?

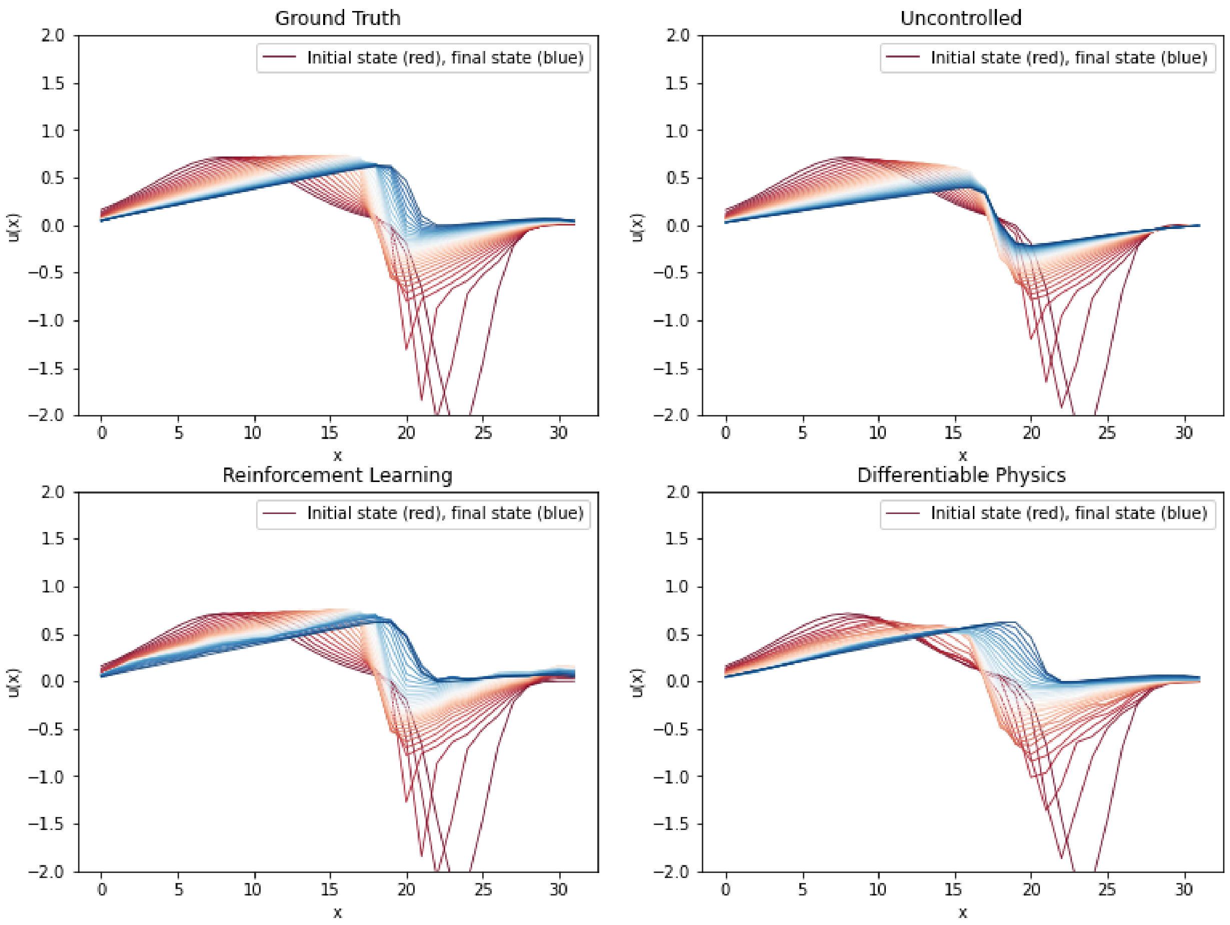
$$\boldsymbol{u}_{t+1} = \mathcal{P}(\boldsymbol{u}_t + \pi(\boldsymbol{u}_t; \boldsymbol{u}^*, t, \theta)\Delta t)$$

π directly computes an action in terms of a force

Reward: punishment of applied force and final trajectory

$$r_t = -\|\boldsymbol{f}_t\|_2^2 - \|\boldsymbol{u}^* - \boldsymbol{u}_t\|^2 I_{\{t=n-1\}}$$

Result



Ground Truth
Uncontrolled
Reinforcement Learning
Differentiable Physics
Initial state (red), final state (blue)
u(x)
x

Outline

6.1 Details of Adjoint Method

To be continued

6.2 Gradient Descent

To be continued

6.3 Unstructured mesh, particles and GNN

To be continued

Outline

Lectures, Slides, Posts

1. **GAMES 103**, 基于物理的计算机动画入门
2. **Differentiable simulation**, SigAsia 2021 Course Note, Tokyo
3. **Adjoint method**, SIGGRAPH 2019
4. **Physics-based Deep Learning**, Book of PBDL
5. **Fluid Simulation for Computer Graphics**, Best of fluid simulation
6. Physics Based Machine Learning for Inverse Problems

Papers

1. Holl, P. et al. 2020. Learning to Control PDES with Differentiable Physics. (2020).
2. Kochkov, D. et al. 2021. Machine learning accelerated computational fluid dynamics.
3. Liang, J. et al. Differentiable Cloth Simulation for Inverse Problems.
4. Meng, X. et al. 2020. PPINN: Parareal physics-informed neural network for time-dependent PDEs. Computer Methods in Applied Mechanics and Engineering. 370, (Oct. 2020), 113250. DOI:https://doi.org/10.1016/j.cma.2020.113250.
5. Qiao, Y.-L. et al. 2020. Scalable Differentiable Physics for Learning and Control. arXiv.
6. Raissi, M. et al. 2019. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics. 378, (Feb. 2019), 686–707. DOI:https://doi.org/10.1016/j.jcp.2018.10.045.

Thank’s for Listening!