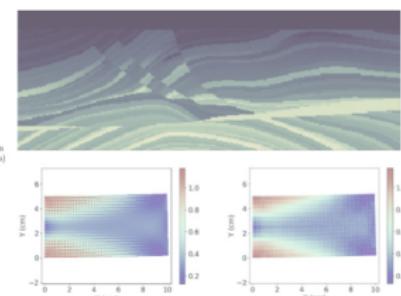
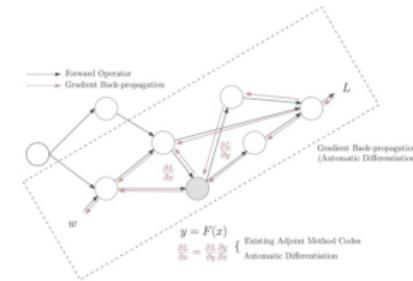
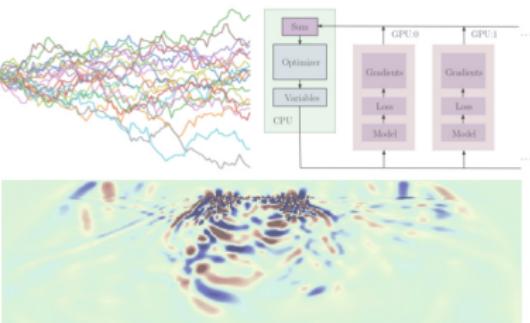


# Machine Learning for Computational Engineering

Kailai Xu  
Stanford University

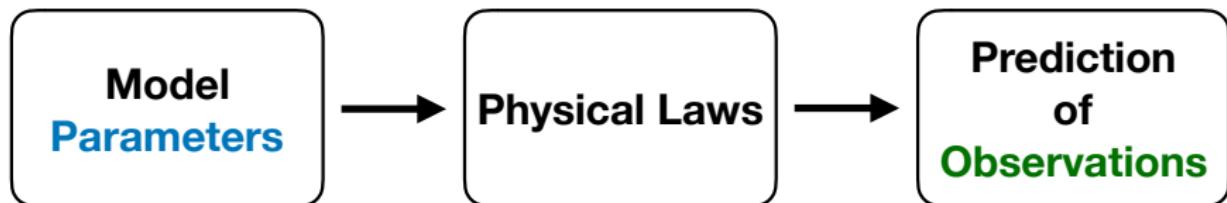


# Outline

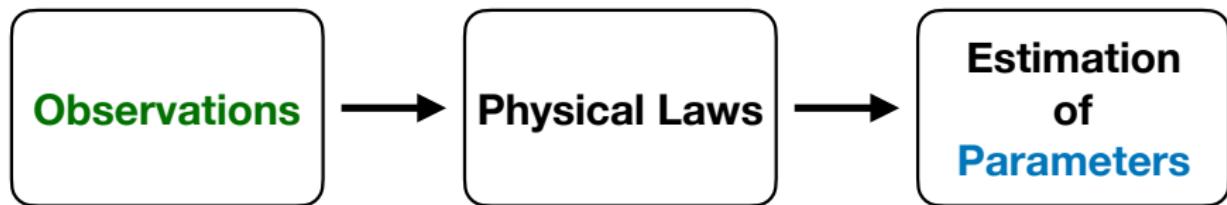
- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

# Inverse Modeling

## Forward Problem



## Inverse Problem



# Inverse Modeling

We can formulate inverse modeling as a PDE-constrained optimization problem

$$\min_{\theta} L_h(u_h) \quad \text{s.t. } F_h(\theta, u_h) = 0$$

- The **loss function**  $L_h$  measures the discrepancy between the prediction  $u_h$  and the observation  $u_{\text{obs}}$ , e.g.,  $L_h(u_h) = \|u_h - u_{\text{obs}}\|_2^2$ .
- $\theta$  is the **model parameter** to be calibrated.
- The **physics constraints**  $F_h(\theta, u_h) = 0$  are described by a system of partial differential equations or differential algebraic equations (DAEs); e.g.,

$$F_h(\theta, u_h) = A(\theta)u_h - f_h = 0$$

# Function Inverse Problem

$$\min_{\mathbf{f}} L_h(u_h) \quad \text{s.t. } F_h(\mathbf{f}, u_h) = 0$$

What if the unknown is a **function** instead of a set of parameters?

- Koopman operator in dynamical systems.
- Constitutive relations in solid mechanics.
- Turbulent closure relations in fluid mechanics.
- ...

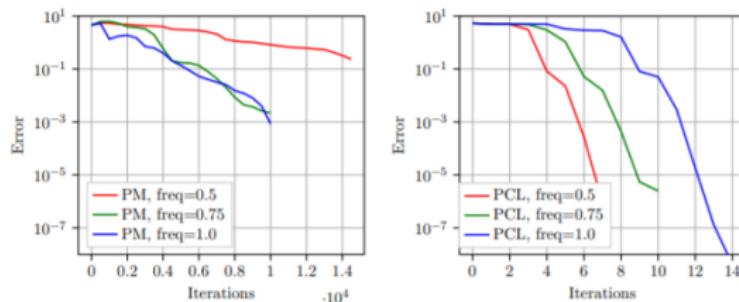
The candidate solution space is **infinite dimensional**.

# Penalty Methods

- Parametrize  $f$  with  $f_\theta$  and incorporate the physical constraint as a **penalty term** (regularization, prior, ...) in the loss function.

$$\min_{\theta, u_h} L_h(u_h) + \lambda \|F_h(f_\theta, u_h)\|_2^2$$

- May not satisfy physical constraint  $F_h(f_\theta, u_h) = 0$  accurately;
- Slow convergence for **stiff** problems;



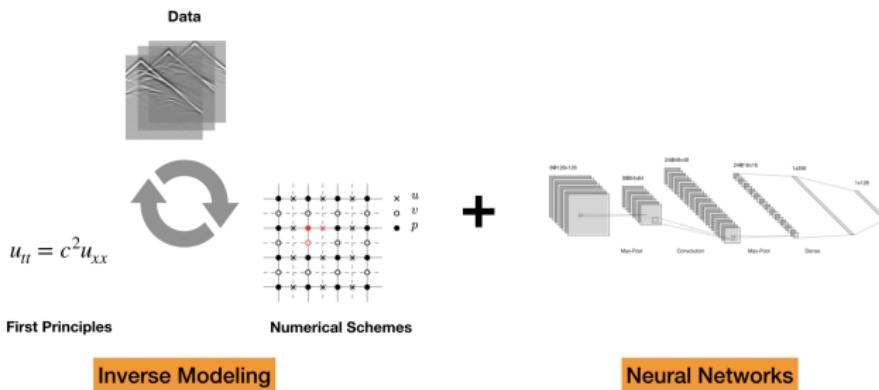
- High dimensional optimization problem; both  $\theta$  and  $u_h$  are variables.

# Machine Learning for Computational Engineering

$$\min_{\theta} L_h(u_h) \quad \text{s.t. } F_h(\mathbf{NN}_{\theta}, u_h) = 0 \leftarrow \text{Solved numerically}$$

- ① Use a deep neural network to approximate the (high dimensional) unknown function;
- ② Solve  $u_h$  from the physical constraint using a numerical PDE solver;
- ③ Apply an unconstrained optimizer to the reduced problem

$$\min_{\theta} L_h(u_h(\theta))$$

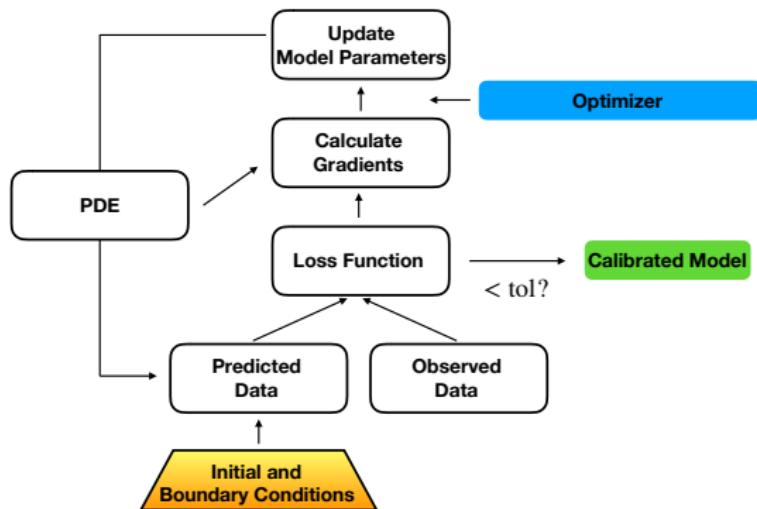


# Gradient Based Optimization

$$\min_{\theta} L_h(u_h(\theta))$$

- Steepest descent method:

$$\theta_{k+1} \leftarrow \theta_k - \alpha_k \nabla_{\theta} L_h(u_h(\theta_k))$$



# Research Question

## Overall

*Develop algorithms and tools for solving inverse problems by combining DNNs and numerical PDE solvers.*

Specifically, I discuss my contributions on:

- ① how to reconcile gradient calculations in both numerical PDE solvers and deep neural networks using **automatic differentiation**;
- ② how to calculate gradients for **implicit and/or iterative operators** in sophisticated numerical solvers;
- ③ how to **accelerate convergence and improve accuracy** with **Hessian information**;
- ④ how to solve **stochastic** inverse problems.

# Outline

- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

# Motivation

## Question

*How deep neural networks communicate gradients with numerical PDE solvers?*

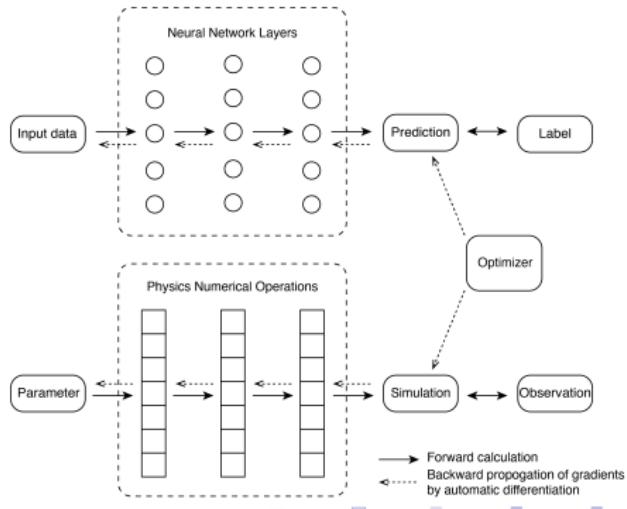
# Automatic Differentiation

The fact that bridges the **technical** gap between machine learning and inverse modeling:

- Deep learning (and many other machine learning techniques) and numerical schemes share the same computational model: composition of individual operators.

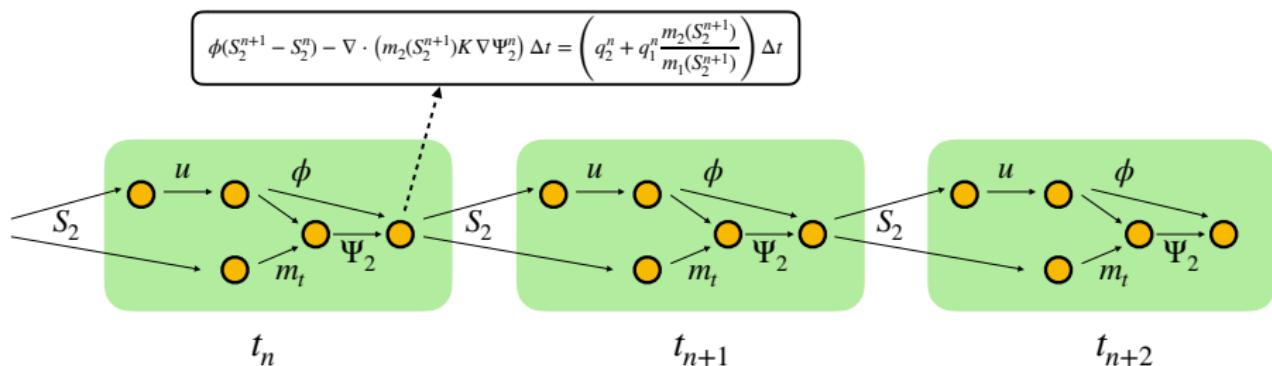
## Mathematical Fact

Back-propagation  
||  
Reverse-mode  
Automatic Differentiation  
||  
Discrete  
Adjoint-State Method

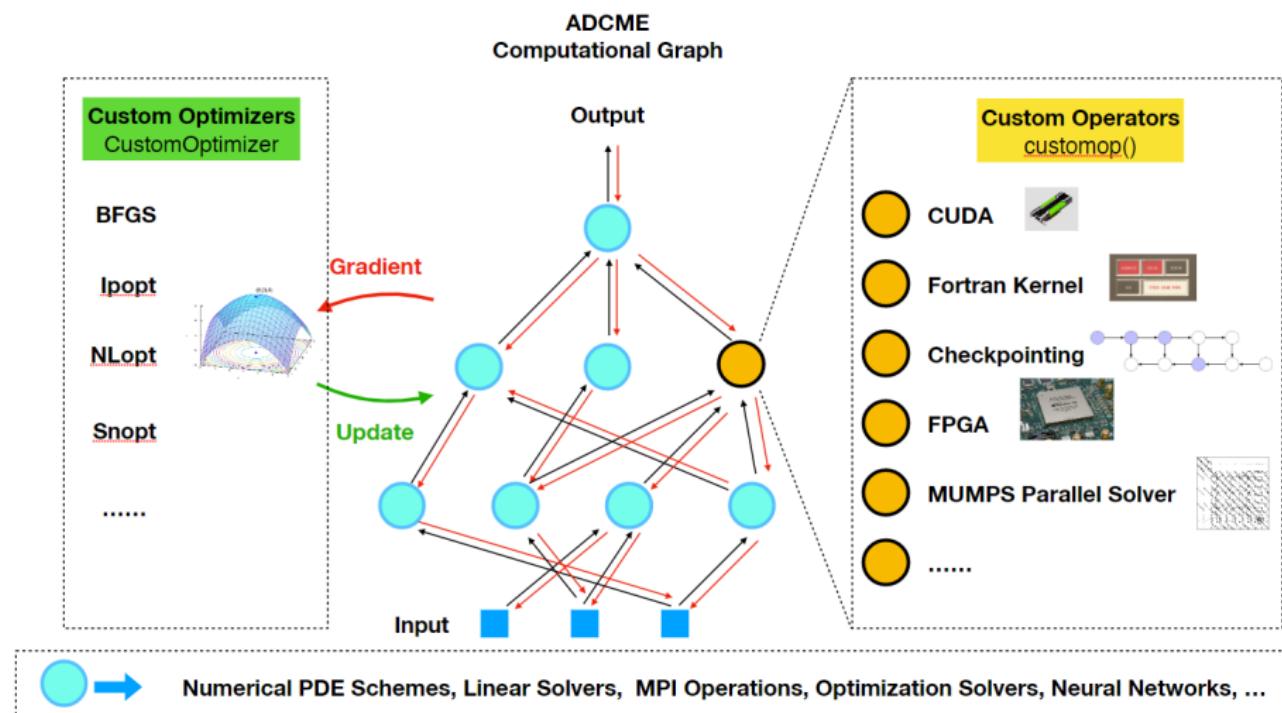


# Computational Graph for Numerical Schemes

- To leverage automatic differentiation for inverse modeling, we need to express the numerical schemes in the “AD language”: computational graph.
- No matter how complicated a numerical scheme is, it can be decomposed into a collection of operators that are interlinked via state variable dependencies.



# ADCME: Computational-Graph-based Numerical Simulation



# How ADCME works

- ADCME translates your numerical simulation codes to computational graph and then the computations are delegated to a heterogeneous task-based parallel computing environment through TensorFlow runtime.

```
div  $\sigma(u) = f(x)$        $x \in \Omega$ 
 $\sigma(u) = C\varepsilon(u)$ 
 $u(x) = u_0(x)$        $x \in \Gamma_u$ 
 $\sigma(x)n(x) = t(x)$        $x \in \Gamma_n$ 

mesh = Mesh(50, 50, 1/50, degree=2)

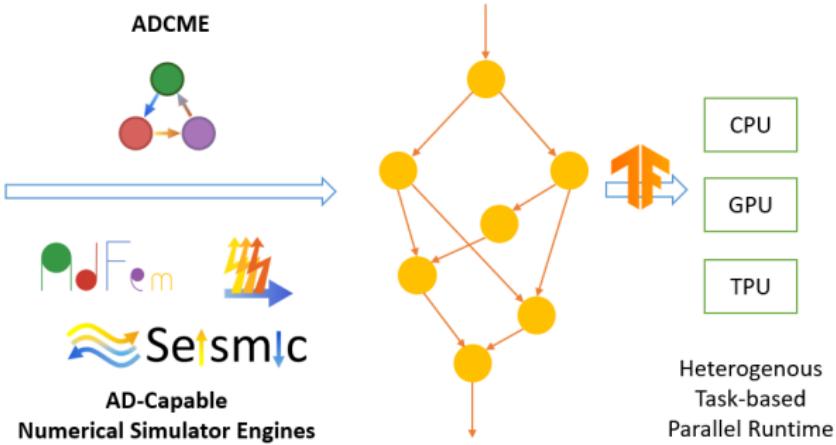
left = bcond((x,y)->x<1e-5, mesh)
right = bcond((x,y1,x2,y2)->(x2>0.049-1e-5) && (x2<0.050-1e-5), mesh)

t1 = eval_f_on_boundary_edge((x,y)->1.0e-4, right, mesh)
t2 = eval_f_on_boundary_edge((x,y)->0.0, right, mesh)
rhs = compute_fem_traction_term(t1, t2, right, mesh)

mu = 0.3
X = gauss_nodes(mesh)
E = abs(f(x, [20, 20, 20, 1]))>squeeze
# E = constant(eval_f_on_gauss_pts(f, mesh))

D = compute_plane_stress_matrix(X, mu*ones(gauss(mesh)))
K = compute_fem_stiffness_matrix(D, mesh)

bdval = [eval_f_on_boundary_node((x,y)->0.0, left, mesh);
         eval_f_on_boundary_node((x,y)->0.0, left, mesh)]
DOF = [left;left+mesh.ndof]
K, rhs = import_Dirichlet_boundary_conditions(K, rhs, DOF, bdval)
u = K\rhs
```



# Summary

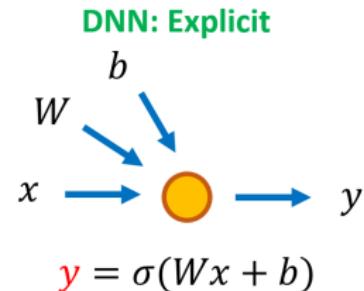
- Mathematically equivalent techniques for calculating gradients:
  - gradient back-propagation (DNN)
  - discrete adjoint-state methods (PDE)
  - reverse-mode automatic differentiation
- Computational graphs bridge the gap between gradient calculations in numerical PDE solvers and DNNs.
- ADCME extends the capability of TensorFlow to PDE solvers, providing users a single piece of software for numerical simulations, deep learning, and optimization.

# Outline

- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

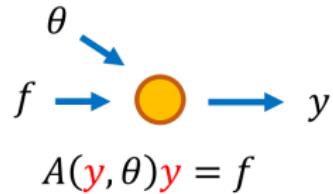
# Motivation

- Most AD frameworks only deal with **explicit operators**, i.e., the functions that has analytical derivatives, or composition of these functions.
- Many scientific computing algorithms are **iterative** or **implicit** in nature.



Linear/Nonlinear	Explicit/Implicit	Expression
Linear	Explicit	$y = Ax$
Nonlinear	Explicit	$y = F(x)$
<b>Linear</b>	<b>Implicit</b>	$Ay = x$
<b>Nonlinear</b>	<b>Implicit</b>	$F(x, y) = 0$

Numerical Schemes:  
**Implicit, Iterative**



## Example

- Consider a function  $f : x \rightarrow y$ , which is implicitly defined by

$$F(x, y) = x^3 - (y^3 + y) = 0$$

If not using the cubic formula for finding the roots, the forward computation consists of iterative algorithms, such as the Newton's method and bisection method

```
y0 ← 0  
k ← 0  
while |F(x, yk)| > ε do  
    δk ← F(x, yk)/Fy'(x, yk)  
    yk+1 ← yk - δk  
    k ← k + 1  
end while  
Return yk
```

```
I ← -M, r ← M, m ← 0  
while |F(x, m)| > ε do  
    c ←  $\frac{a+b}{2}$   
    if F(x, m) > 0 then  
        a ← m  
    else  
        b ← m  
    end if  
end while  
Return c
```

## Example

- An efficient way to do automatic differentiation is to apply the **implicit function theorem**. For our example,  $F(x, y) = x^3 - (y^3 + y) = 0$ ; treat  $y$  as a function of  $x$  and take the derivative on both sides

$$3x^2 - 3y(x)^2y'(x) - y'(x) = 0 \Rightarrow y'(x) = \frac{3x^2}{3y^2 + 1}$$

The above gradient is **exact**.

**Can we apply the same idea to inverse modeling?**

# Physics Constrained Learning (PCL)

$$\min_{\theta} L_h(u_h) \quad \text{s.t. } F_h(\theta, u_h) = 0$$

- Assume that we solve for  $u_h = G_h(\theta)$  with  $F_h(\theta, u_h) = 0$ , and then

$$\tilde{L}_h(\theta) = L_h(G_h(\theta))$$

- Applying the **implicit function theorem**

$$\frac{\partial F_h(\theta, u_h)}{\partial \theta} + \frac{\partial F_h(\theta, u_h)}{\partial u_h} \frac{\partial G_h(\theta)}{\partial \theta} = 0 \Rightarrow \frac{\partial G_h(\theta)}{\partial \theta} = - \left( \frac{\partial F_h(\theta, u_h)}{\partial u_h} \right)^{-1} \frac{\partial F_h(\theta, u_h)}{\partial \theta}$$

- Finally we have

$$\boxed{\frac{\partial \tilde{L}_h(\theta)}{\partial \theta} = \frac{\partial L_h(u_h)}{\partial u_h} \frac{\partial G_h(\theta)}{\partial \theta} = - \frac{\partial L_h(u_h)}{\partial u_h} \left( \frac{\partial F_h(\theta, u_h)}{\partial u_h} \Big|_{u_h=G_h(\theta)} \right)^{-1} \frac{\partial F_h(\theta, u_h)}{\partial \theta} \Big|_{u_h=G_h(\theta)}}$$

# Physics Constrained Learning for Stiff Problems

- For stiff problems, better to resolve physics using PCL.
- Consider a model problem

$$\min_{\theta} \|u - u_0\|_2^2 \quad \text{s.t. } Au = \theta y$$

$$\text{PCL : } \min_{\theta} \tilde{L}_h(\theta) = \|\theta A^{-1}y - u_0\|_2^2 = (\theta - 1)^2 \|u_0\|_2^2$$

$$\text{Penalty Method : } \min_{\theta, u_h} \tilde{L}_h(\theta, u_h) = \|u_h - u_0\|_2^2 + \lambda \|Au_h - \theta y\|_2^2$$

## Theorem

The condition number of  $A_\lambda$  is

$$\liminf_{\lambda \rightarrow \infty} \kappa(A_\lambda) = \kappa(A)^2, \quad A_\lambda = \begin{bmatrix} I & 0 \\ \sqrt{\lambda}A & -\sqrt{\lambda}y \end{bmatrix}, \quad y = \begin{bmatrix} u_0 \\ 0 \end{bmatrix}$$

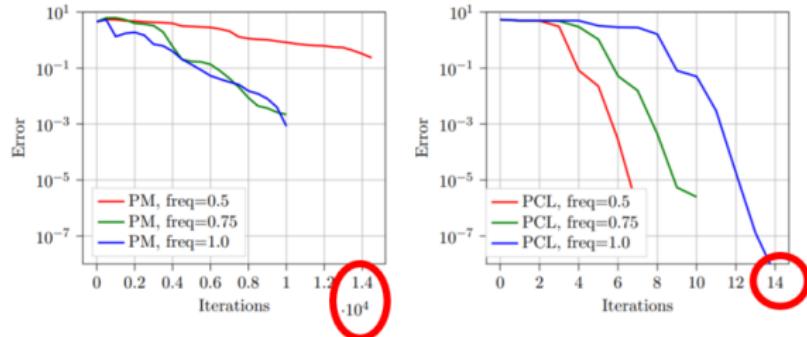
and therefore, the condition number of the unconstrained optimization problem from the penalty method is equal to the square of the condition number of the PCL asymptotically.

# Physics Constrained Learning for Stiff Problems

## Parameter Inverse Problem

$$\Delta u + k^2 g(x)u = 0$$
$$g(x) = 5x^2 + 2y^2$$

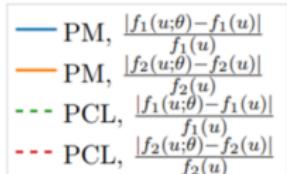
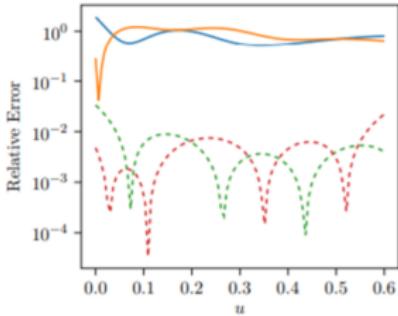
$$g_{\theta}(x) = \theta_1 x^2 + \theta_2 y^2 + \theta_3 xy \\ + \theta_4 x + \theta_5 y + \theta_6$$



## Approximate Unknown Functions using DNNs

$$-\nabla \cdot (\mathbf{f}(\mathbf{u}) \nabla \mathbf{u}) = h(\mathbf{x})$$

$$\mathbf{f}(\mathbf{u}) = \begin{bmatrix} NN(u; \theta_1) & 0 \\ 0 & NN(u; \theta_2) \end{bmatrix}$$



# Summary

- Implicit and iterative operators are ubiquitous in numerical PDE solvers. These operators are insufficiently treated in deep learning software and frameworks.
- Penalty methods suffer from slow convergence for stiff problems.
- First order physics constrained learning helps you calculate gradients of implicit/iterative operators efficiently.
- PCL leads to faster convergence and better accuracy compared to penalty methods.

# Outline

- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning**
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

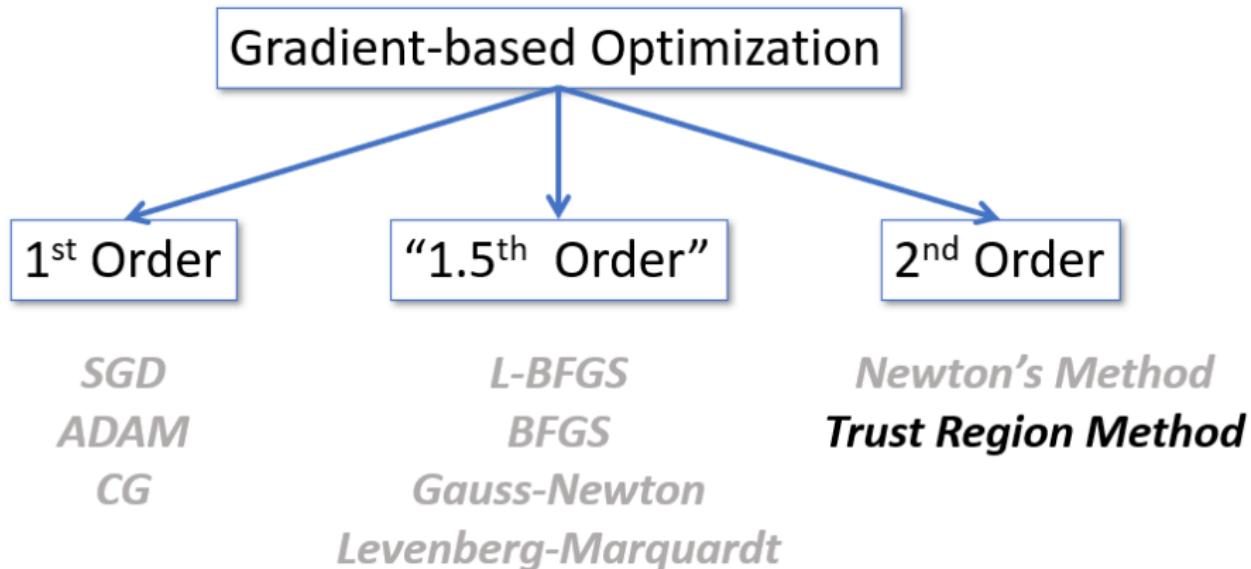
# Motivation

- First order methods and quasi-Newton methods are workhorse of physics informed machine learning
  - First order methods: stochastic gradient descent (SGD) and ADAM
  - Quasi-Newton methods: BFGS and L-BFGS
- Second order methods, such as trust region methods, are desirable for accelerating convergence and improving accuracy.
  - It requires us to calculate the curvature information (Hessians).

## Goal

*Accelerate convergence and improve accuracy with Hessian information*

# Optimization Algorithms



# Trust Region vs. Line Search

## Trust Region

- Approximate  $f(x_k + p)$  by a model quadratic function

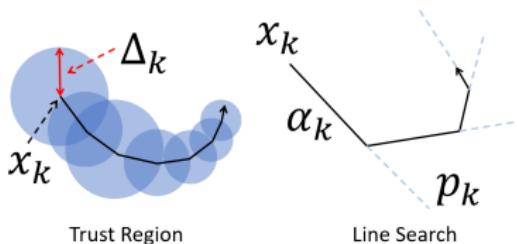
$$m_k(p) = f_k + g_k^T p + \frac{1}{2} p^T B_k p$$

$$f_k = f(x_k), g_k = \nabla f(x_k), B_k = \nabla^2 f(x_k)$$

- Solve the optimization problem within a trust region  $\|p\| \leq \Delta_k$

$$p_k = \arg \min_p m_k(p) \quad \text{s.t. } \|p\| \leq \Delta_k$$

- If decrease in  $f(x_k + p_k)$  is sufficient, then update the state  $x_{k+1} = x_k + p_k$ ; otherwise,  $x_{k+1} = x_k$  and improve  $\Delta_k$ .



## Line Search

- Determine a descent direction  $p_k$
- Determine a step size  $\alpha_k$  that sufficiently reduces  $f(x_k + \alpha_k p_k)$
- Update the state  
$$x_{k+1} = x_k + \alpha_k p_k$$

# Second-order Physics Constrained Learning

- Consider a composite function with a vector input  $x$  and scalar output

$$v = f(G(x)) \quad (1)$$

- Define

$$\begin{aligned}f_{,k}(y) &= \frac{\partial f(y)}{\partial y_k}, & f_{,kl}(y) &= \frac{\partial^2 f(y)}{\partial y_k \partial y_l} \\G_{k,I}(x) &= \frac{\partial G_k(x)}{\partial x_I}, & G_{k,Ir}(x) &= \frac{\partial^2 G_k(x)}{\partial x_I \partial x_r}\end{aligned}$$

- Differentiate Equation (1) with respect to  $x_i$

$$\frac{\partial v}{\partial x_i} = f_{,k} G_{k,i} \quad (2)$$

- Differentiate Equation (2) with respect to  $x_j$

$$\frac{\partial^2 v}{\partial x_i \partial x_j} = f_{,kr} G_{k,i} G_{r,j} + f_{,k} G_{k,ij}$$

# Second-order Physics Constrained Learning

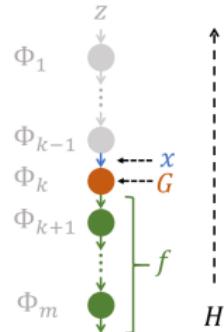
In the vector form,

$$\nabla^2 v = (\nabla G)^T \nabla^2 f(\nabla G) + \nabla^2(\bar{G}^T G) \quad \bar{G} = \nabla f$$

- Consider a function composed of a sequence of computations

$$v = \Phi_m(\Phi_{m-1}(\cdots(\Phi_1(z))))$$

- 1: Initialize  $H \leftarrow 0$
- 2: **for**  $k = m - 1, m - 2, \dots, 1$  **do**
- 3:   Define  $f := \Phi_m(\Phi_{m-1}(\cdots(\Phi_{k+1}(\cdot))))$ ,  $G := \Phi_k$
- 4:   Calculate the gradient (Jacobian)  $J \leftarrow \nabla G$
- 5:   Extract  $\bar{G}$  from the saved gradient back-propagation data.
- 6:   Calculate  $Z = \nabla^2(\bar{G}^T G)$
- 7:   Update  $H \leftarrow J^T H J + Z$
- 8: **end for**



## Numerical Benchmark

- We consider the heat equation in  $\Omega = [0, 1]^2$

$$\frac{\partial u}{\partial t} = \nabla \cdot (\kappa(x, y) \nabla u) + f(x, y) \quad x \in \Omega$$

$$u(x, y, 0) = x(1 - x)y^2(1 - y)^2 \quad (x, y) \in \Omega$$

$$u(x, y, t) = 0 \quad (x, y) \in \partial\Omega$$

- The diffusivity coefficient  $\kappa$  and exact solution  $u$  are given by

$$\kappa(x, y) = 2x^2 - 1.05x^4 + x^6 + xy + y^2$$

$$u(x, y, t) = x(1 - x)y^2(1 - y)^2 e^{-t}$$

- We learn a DNN approximation to  $\kappa$  using full-field observations of  $u$

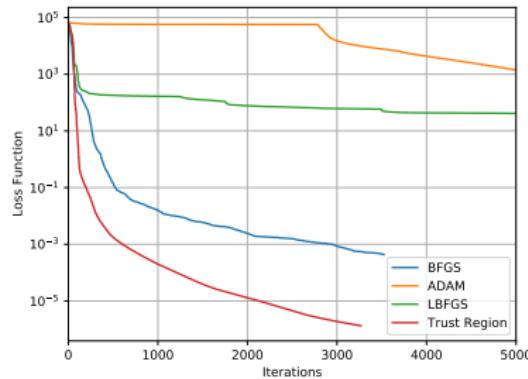
$$\kappa(x, y) \approx \text{NN}_\theta(x, y)$$

# Convergence

- The optimization problem is given by

$$\min_{\theta} L(\theta) = \sum_n \sum_{i,j} \left( \frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} - F_{i,j}(u^{n+1}; \theta) - f_{i,j}^{n+1} \right)^2 \quad (3)$$

Here  $F_{i,j}(u^{n+1}; \theta)$  is the 4-point finite difference approximation to the Laplacian  $\nabla \cdot (\text{NN}_{\theta} \nabla u)$ .

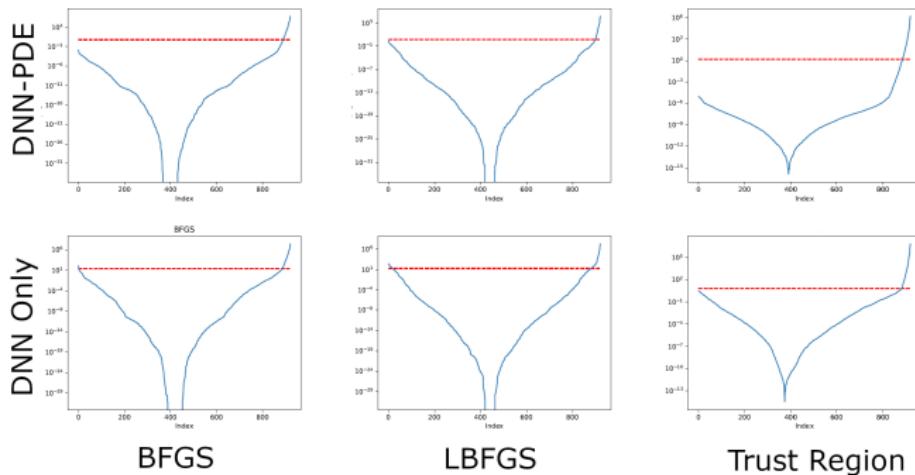


# Effect of PDEs

- Consider the loss function excluding the effects of PDEs

$$I(\theta) = \sum_{i,j} (\text{NN}_\theta(x_{i,j}, y_{i,j}) - \kappa(x_{i,j}, y_{i,j}))^2$$

- Eigenvalue magnitudes of  $\nabla^2 L(\theta)$  and  $\nabla^2 I(\theta)$



# Effect of PDEs

- Most of the eigenvalue directions at the local landscape of loss functions are “flat”  $\Rightarrow$  “effective degrees of freedom (DOFs)”.
- Physical constraints (PDEs) further cannibalize effective DOFs:

	BFGS	LBFGS	Trust Region
DNN-PDE	<b>31</b>	<b>22</b>	<b>35</b>
DNN Only	34	41	38

## Effect of Widths and Depths

- The ratio of zero eigenvalues **increases** as
  - the number of hidden layers increase for a fixed number (20) of neurons per layer

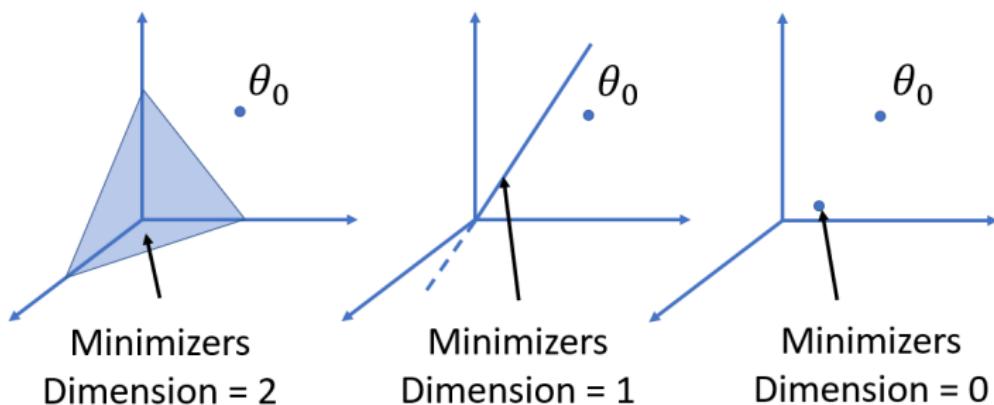
# Hidden Layers	LBFGS	BFGS	Trust Region
1	76.54	72.84	77.78
2	98.2	94.41	93.21
3	98.7	98.15	96.09

- the number of neurons per layer increases for a fixed number (3) of hidden layers

# Neurons per Layer	LBFGS	BFGS	Trust Region
5	93.83	85.19	69.14
10	97.7	83.52	89.66
20	96.2	97.39	96.42

# Effect of Widths and Depths

- Implications for overparametrization: **the minimizer lies on a relatively higher dimensional manifold of the parameter space.**



# Summary

- Trust region methods converge significantly faster compared to first order/quasi second order methods by leveraging Hessian information.
- Second order physics constrained learning helps you calculate Hessian matrices efficiently.
- The local minimum of DNNs have small effective degrees of freedom compared to DNN sizes.

# Outline

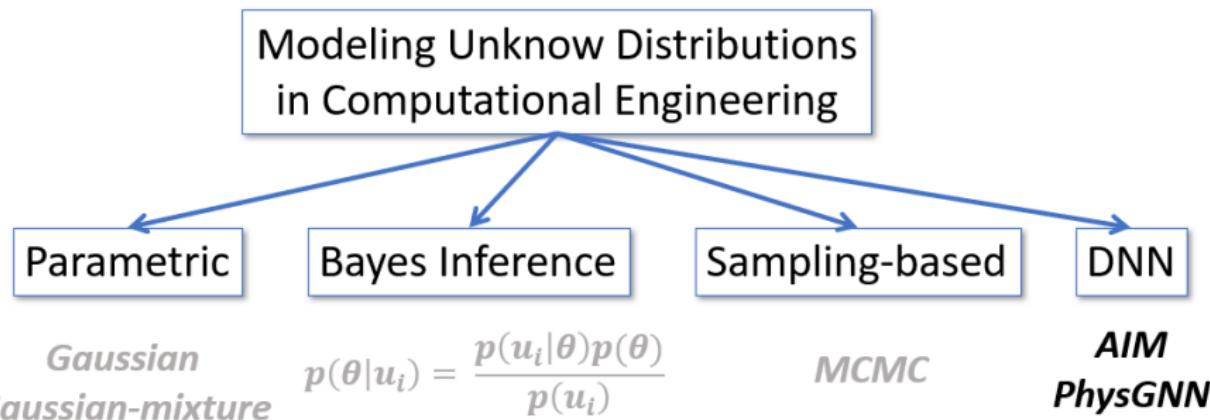
- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

# Motivation

$$\min_{\mathbf{f}} L_h(u_h) \quad \text{s.t. } F_h(\mathbf{f}, u_h) = 0$$

- What if  $f$  is a random variable?
  - uncertainty estimation
- What if  $F_h$  is a stochastic model?
  - stochastic differential equations (SDE)

# Methodologies



# Modeling Stochasticity using Deep Neural Networks

- We consider a map  $F$

$$F : (w, \theta) \mapsto u(\cdot, \cdot)$$

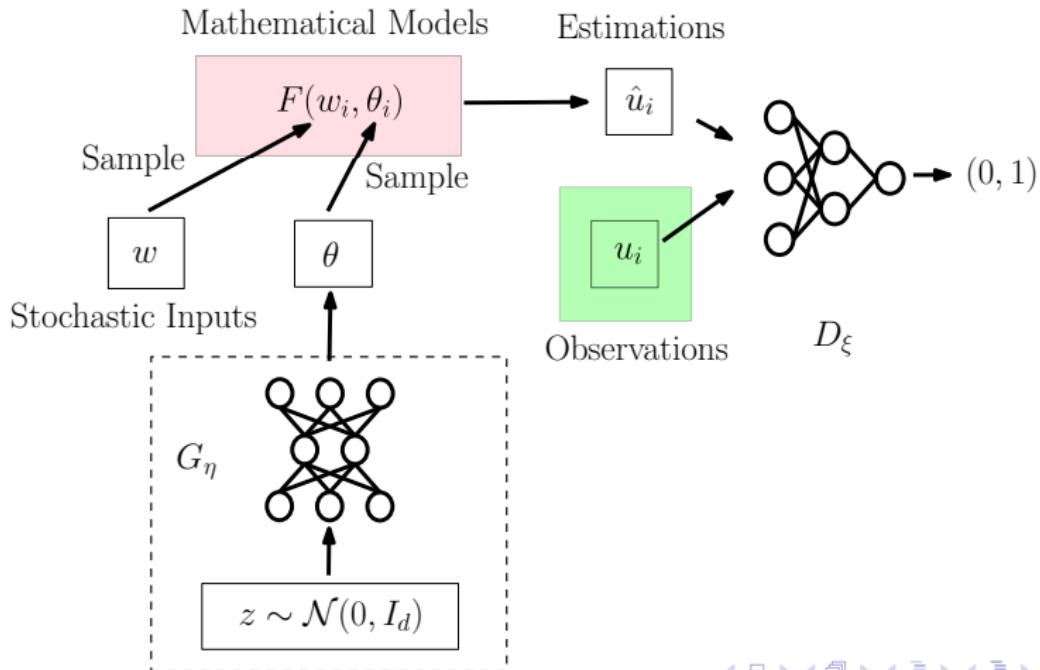
- $w$  is sampled from a **know** stochastic process (stochasticity intrinsic to the model).
- $\theta$  is a random variable of interest.
- $u(x, t)$  is the output of the mapping, which depends on the location  $x$  and time  $t$ .  $u$  is also a random variable.
- Examples:
  - CIR process (a short-term interest rate model)

$$dr_t = \kappa(\tau - r_t)dt + \sqrt{r_t}\sigma dW_t$$

Here  $\theta = (\kappa, \tau, \sigma)$ ,  $w = W_t$ , and  $u = r_t$ .

# Adversarial Inverse Modeling (AIM)

- Approximate the unknown distribution  $\theta$  with a DNN  $G_\eta$  (generator);
- A discriminator  $D_\xi$  tries to distinguish between generated and observed samples.



# Adversarial Inverse Modeling

- Find the **Nash equilibrium** by solving a min-max problem

$$\min_{\eta} \max_{\xi} V(D_\xi, G_\eta)$$

Here  $V$  is the reward to the discriminator  $D$  for distinguishing generated and observed samples.

- Inspired by generative adversarial nets (GAN): iteratively minimizing discriminator and generator loss functions
  - Example: Kullback-Leibler (KL) GAN

$$L^G(\{\hat{u}_i\}; \eta) = \frac{1}{n} \sum_{i=1}^n \log \frac{1 - D_\xi(\hat{u}_i(\eta))}{D_\xi(\hat{u}_i(\eta))}$$

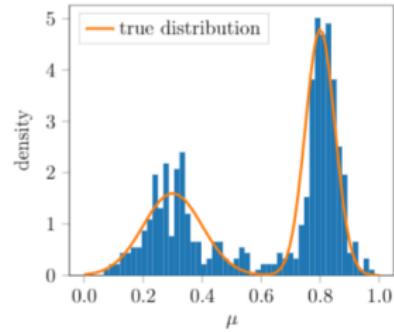
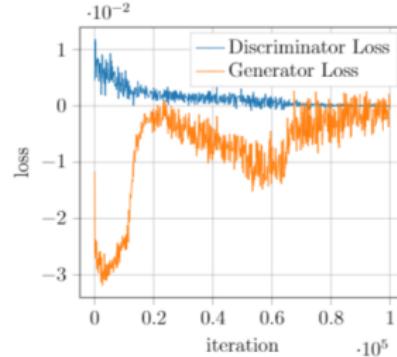
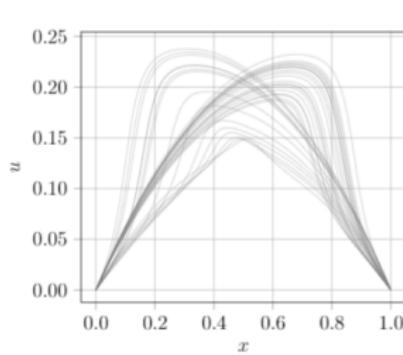
$$L^D(\{u_i\}, \{\hat{u}_i\}; \xi) = - \frac{1}{n} \sum_{i=1}^n (\log D_\xi(\hat{u}_i) + \log(1 - D_\xi(u_i)))$$

# Numerical Benchmarks

- We consider a Poisson equation

$$\begin{cases} -\nabla \cdot (a(x)\nabla u(x)) = 1 & x \in (0, 1) \\ u(0) = u(1) = 0 & \text{otherwise} \end{cases}$$

$$a(x) = 1 - 0.9 \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)$$

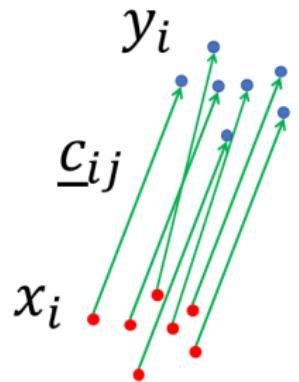


# Optimal Transport

- Solve the transport matrix from a linear programming problem

$$\min_{p_{ij}} W(p; x, y) = \sum_{i=1}^n \sum_{j=1}^m p_{ij} c_{ij}$$

$$\text{s.t. } \sum_{j=1}^m p_{ij} = \frac{1}{n}, \quad \sum_{i=1}^n p_{ij} = \frac{1}{m}, \quad 0 \leq p_{ij} \leq 1$$



- Discrete Wasserstein distance between two sets of samples:

$$D(x, y) = W(p^*; x, y) = \sum_{1 \leq i \leq n, 1 \leq j \leq m} p_{ij}^* \|x_i - y_j\|_1$$

# AD for Discrete Wasserstein Distance

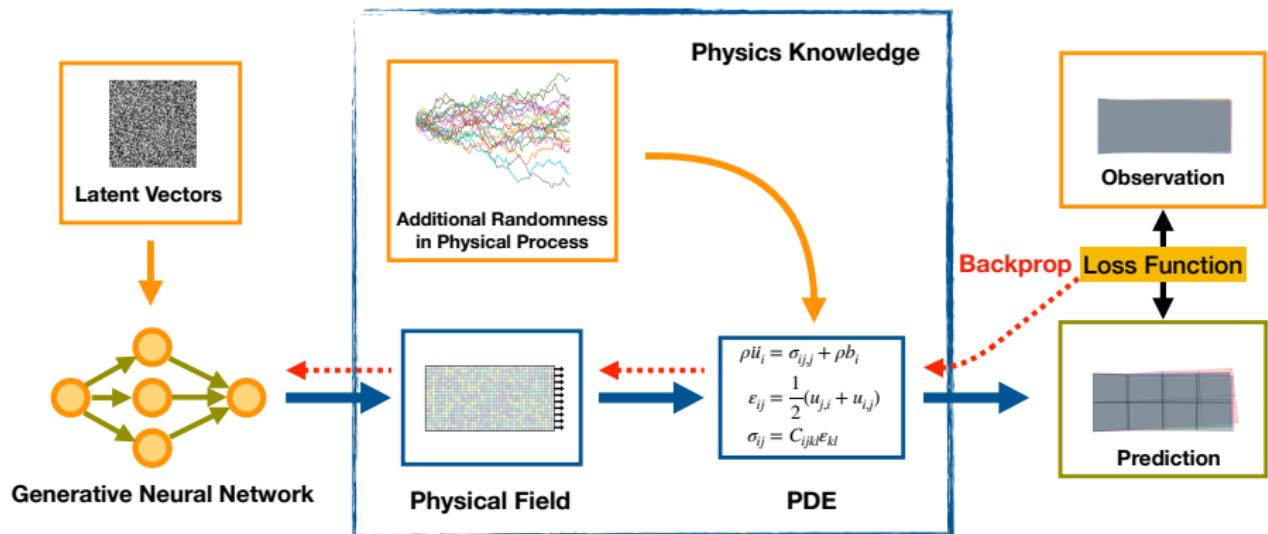
## Theorem

Assume that  $\mathbf{p}^*([\underline{c}_{ij}])$  is the optimal solution to the linear programming problem given the cost matrix  $[\underline{c}_{ij}]$ , then we have

$$\frac{\partial W(\mathbf{p}^*([\underline{c}_{ij}]))}{\partial \underline{c}_{ij}} = p_{ij}^*$$

- We have a “**handpicked discriminator**”.
- Numerical implementation is very similar to OptNet (Amos, Brandon, and J. Zico Kolter, PMLR 2017):
  - The forward computation constitutes solving an optimization problem.

# PhysGNN: Training Generative Neural Networks with Discrete Wasserstein Distance

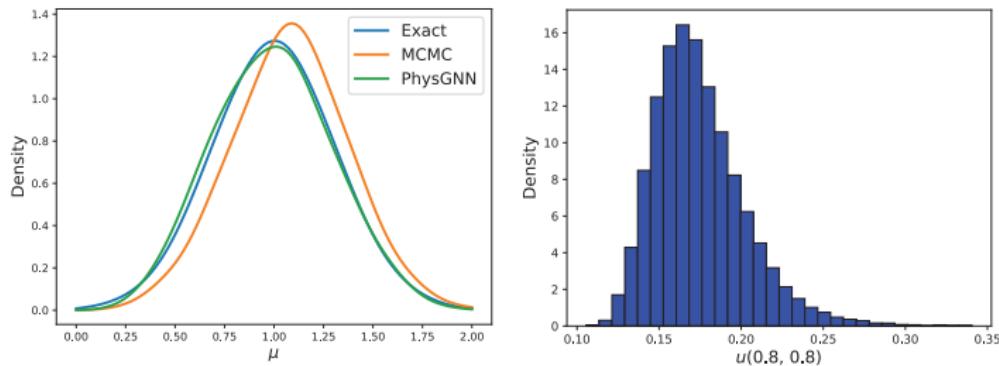


# Numerical Benchmarks

- We consider a 2D Poisson's equation

$$\begin{aligned}\nabla \cdot (\kappa \nabla u) &= 2\pi^2 \sin(\pi x) \sin(\pi y) & (x, y) \in (0, 1)^2 \\ u(x, y) &= 0 & (x, y) \in \partial(0, 1)^2 \\ \kappa &= |v|, \quad v \sim \mathcal{N}(1.0, 0.3^2)\end{aligned}\tag{4}$$

- Observation:  $u(0.8, 0.8)$
- $\sim 20,000$  PDE solves for both MCMC and PhysGNN:



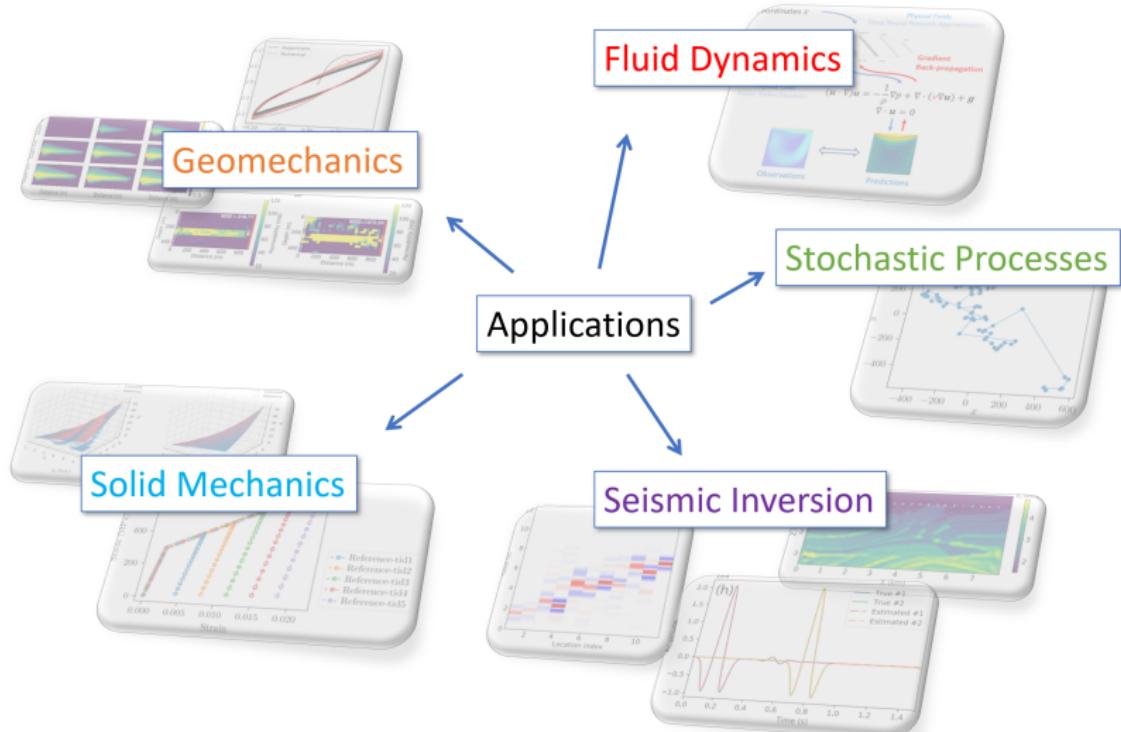
# Summary

- Generative neural networks are used to approximate unknown distributions in a stochastic inverse problem.
- Training generative neural networks
  - Adversarial inverse modeling (AIM): solving a min-max optimization problem involving generator and discriminator nets
  - Physics Generative Neural Network (PhysGNN): minimizing a discrete Wasserstein distance

# Outline

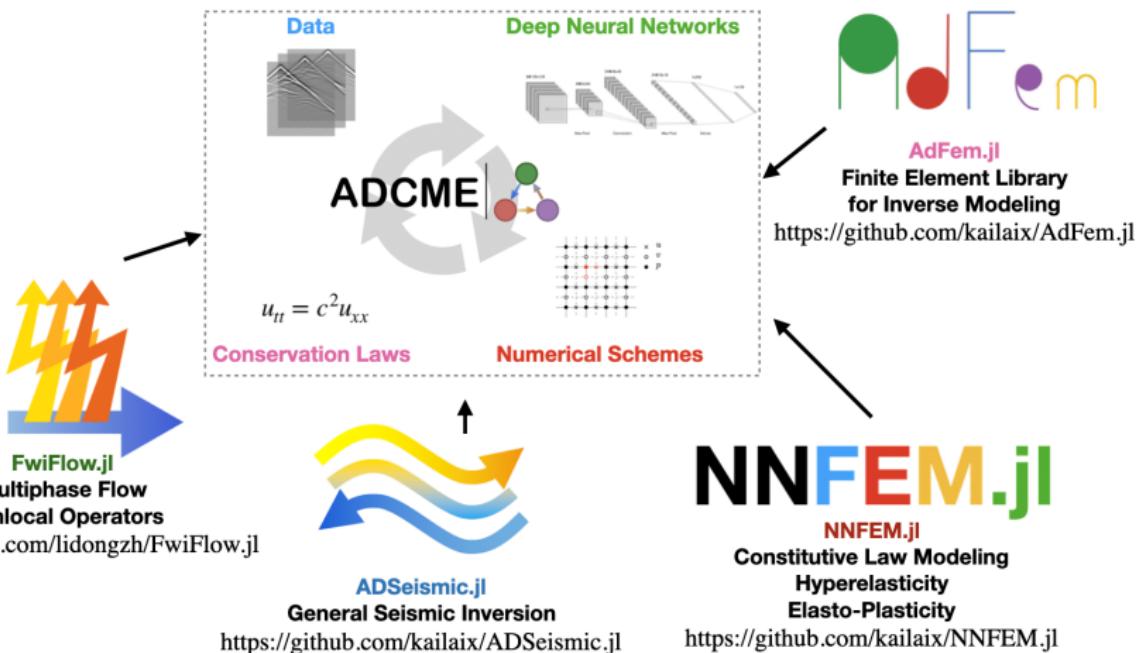
- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 First Order Physics Constrained Learning
- 4 Second Order Physics Constrained Learning
- 5 Generative Neural Networks for Stochastic Inverse Problems
- 6 Conclusion

# Applications



<https://github.com/kailaiX/ADCME.jl>

# A General Approach to Inverse Modeling



PLACEHOLDER