CME 216, ME 343 - Winter 2021

Eric Darve, ICME

Stanford University

Let's apply the method of automatic differentiation to differentiate DNNs with respect to their input variable x.

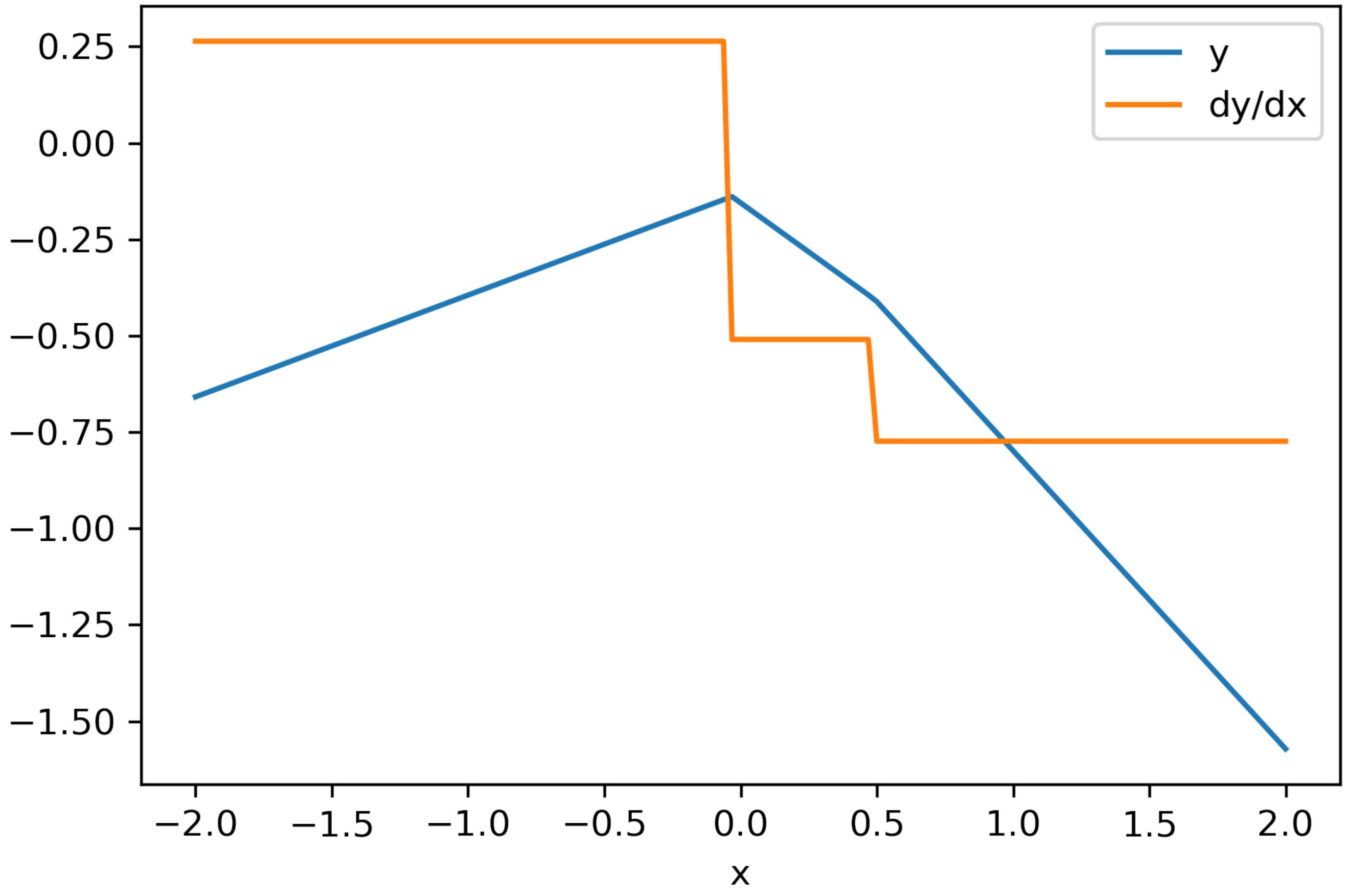
It's very simple now.

Build a model

```
class AD_Model(tf.keras.models.Model):
    def __init__(self):
        super(AD_Model, self).__init__()
        self.dense_1 = layer_1(2)
        self.dense_2 = layer_2(1)

    # Forward pass
    def call(self, inputs):
        x = self.dense_1(inputs)
        y = self.dense_2(x)
        return y

model = AD_Model()
model.build((1,1))
```

Differentiate

```
x = reshape_2d( tf.linspace(-2.0, 2.0, 129) )

with tf.GradientTape() as g:
    g.watch(x)
    y = model(x)

dy_dx = g.gradient(y, x)
```


0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x
y
dy/dx

Let's build a simple ODE solver using DNNs

$$y'' = -4\cos 2x, \quad y(0) = 1, \quad y'(0) = 0$$

Exact solution: $\cos 2x$

Build a model like before and add two functions:

`get_derivatives`

`loss`

```python
def get_derivatives(self, x_input):
    x = tf.constant(x_input)
    with tf.GradientTape() as g:
        g.watch(x)
        with tf.GradientTape() as gg:
            gg.watch(x)
            y = self(x)
        y_x = gg.gradient(y,x)
    y_xx = g.gradient(y_x,x)
    return y, y_x, y_xx
```

Second order derivatives are obtained by calling `gradient` twice.

Gradients can be nested as many times as needed to compute higher-order derivatives.

Our loss function is of the type:

$$L = \sum_{i=1}^{n_y} (y(x_i^y; \theta) - y_i)^2$$

$$+ \sum_{i=1}^{n_f} (y''(x_i^f; \theta) - f_i)^2$$

```python
def loss(self, X, Y):
    # data observation loss
    y = self(X[0]) # y(x)
    # Physics loss
    _, _, phys = self.get_derivatives(X[1]) # y''(x)
    return self.loss_fun(Y[0], y) + self.loss_fun(Y[1], phys)
```

```python
self.loss_fun = tf.keras.losses.MeanSquaredError()
```

`X[0]`: x_i^y, location of y_i data

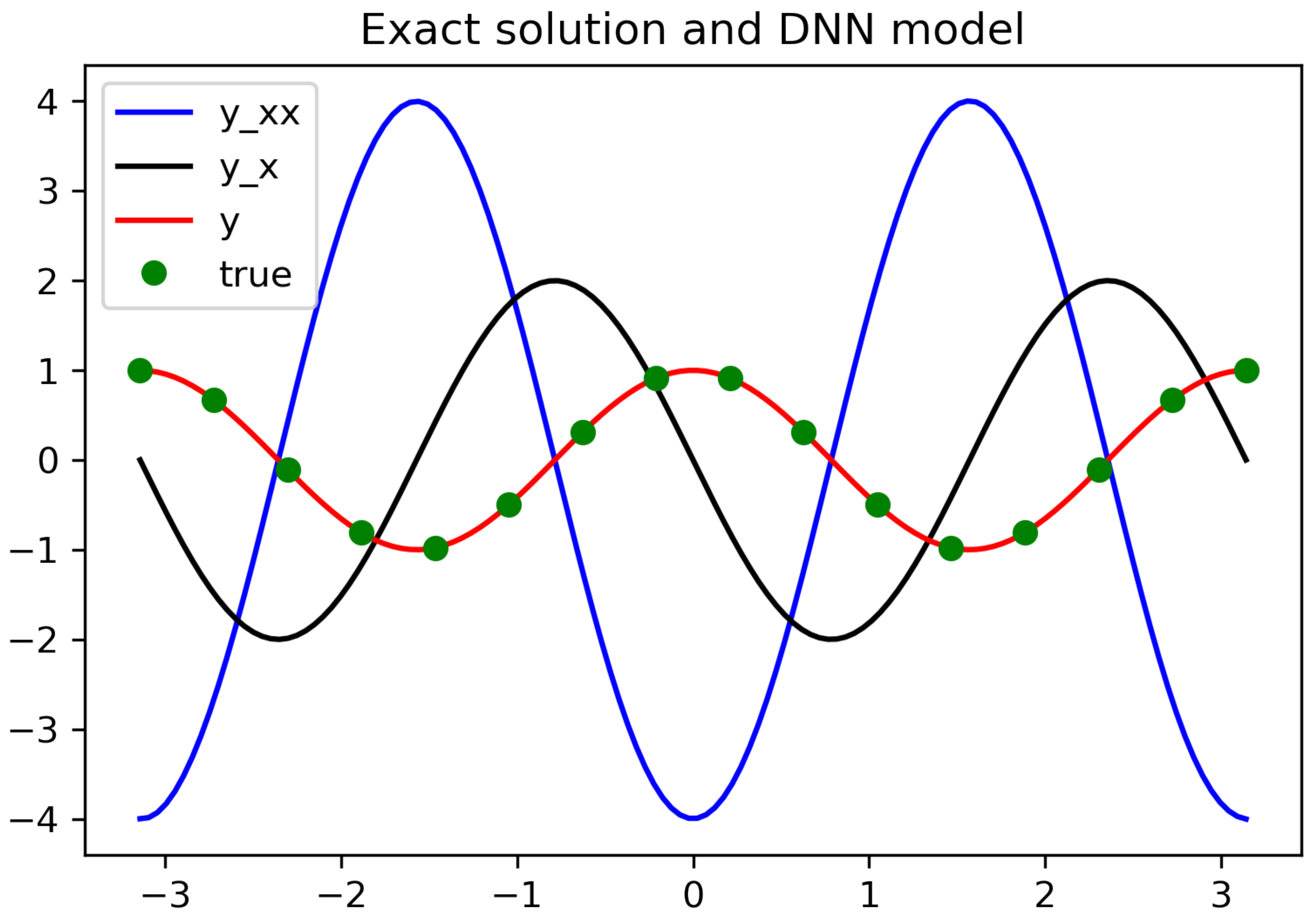
`Y[0]`: y_i data

`X[1]`: x_i^f, location of $f_i = y_i''$ data

`Y[1]`: $f_i = y_i''$ data

We train using the L-BFGS-B scipy optimizer.


Training and validation losses
loss
validation
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
100
200
300
400
500
600
700
epoch



Exact solution and DNN model
y_xx
y_x
y
true
4
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3

This method can be extended to any type of differential equations:

- PDEs with multiple input variables $(x_1, \ldots, x_d)$
- Non-linear PDEs
- Time-dependent PDEs

Example of a PDE in 2D

`get_derivatives` function

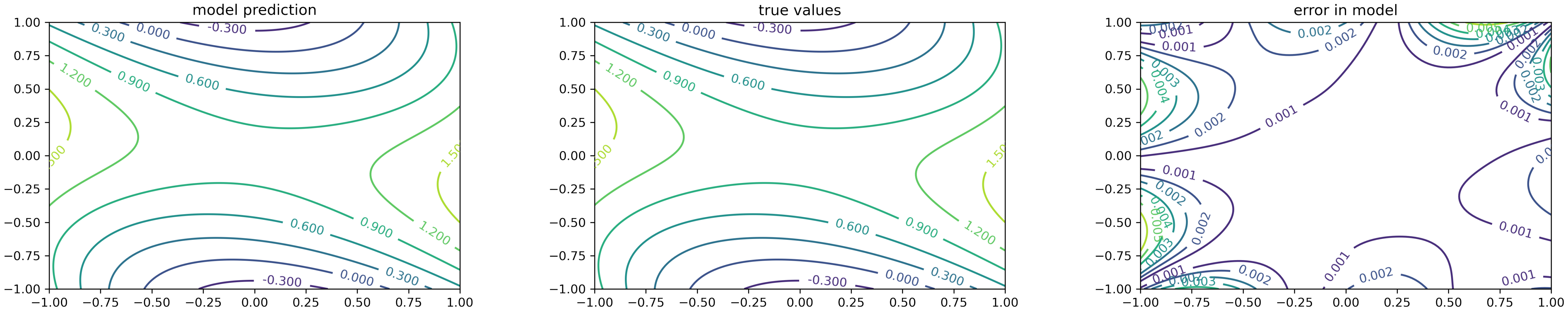
```python
def get_derivatives(self, x):
    x1 = tf.constant(x[:,0], dtype=tf.float64)
    x2 = tf.constant(x[:,1], dtype=tf.float64)
    with tf.GradientTape(persistent=True) as g:
        g.watch(x1)
        g.watch(x2)
        with tf.GradientTape() as gg:
            gg.watch(x1)
            gg.watch(x2)
            x = tf.stack([x1, x2], 1)
            u = self(x, training=True)
        [u_x, u_y] = gg.gradient(u,[x1,x2])
    u_xx = g.gradient(u_x,x1)
    u_yy = g.gradient(u_y,x2)
    del g
    return u, u_x, u_y, u_xx, u_yy
```

Example: solving

$$\triangle u = 2 - 5\cos(2 + y)$$

Solution is:

$$u(x, y) = x^2 + \cos(x + 2y) + xy$$


model prediction
true values
error in model
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0.300
0.000
-0.300
1.200
0.900
0.600
0.001
0.002
0.003
0.004

PDE: $\nabla \cdot (k\nabla u) = f$

Solving for $u(x, y)$ and $k(x, y)$.

Exact value for u: $u(x, y) = x^2 + y^2$

Exact value of k: $k(x, y) = 1 + x^2$

Exact value for $f(x, y) = \nabla \cdot (k\nabla u)$

$$f(x, y) = 8x^2 + 4$$

Python notebook for 2D examples

```python
def __init__(self):
    super(PI_u_k, self).__init__()
    # Define all layers
    self.dense_1 = tf.keras.layers.Dense(16,\
                        activation=tf.keras.activations.tanh)
    self.dense_2 = tf.keras.layers.Dense(16,\
                        activation=tf.keras.activations.tanh)
    self.dense_3 = tf.keras.layers.Dense(2,\
                        activation=tf.keras.activations.linear)
```

```python
with tf.GradientTape(persistent=True) as g:
    g.watch(x1)
    g.watch(x2)
    with tf.GradientTape() as gg:
        gg.watch(x1)
        gg.watch(x2)
        x = tf.stack([x1, x2], 1)
        z = self(x)
        u, k = tf.split(z, num_or_size_splits=2, axis=1)
    [u_x, u_y] = gg.gradient(u, [x1,x2])
    u_x = tf.reshape( u_x, (x1.shape[0], 1) )
    u_y = tf.reshape( u_y, (x2.shape[0], 1) )
    k_ux = k * u_x
    k_uy = k * u_y
pde = g.gradient(k_ux,x1) + g.gradient(k_uy,x2)
del g
return u, k, u_x, u_y, pde
```

See notebook for the complete code.

Loss contains 3 terms:

1. $n_{\mathrm{u}} \quad (u(x_i;\theta) - u_i)^2$
2. $n_{\mathrm{k}} \quad (k(x_i;\theta) - k_i)^2$
3. $n_{\mathrm{phys}} \quad (f - \nabla \cdot (k \nabla u))^2$

```python
def loss(self, X, Y):
    # u data observation loss
    u = self(X[0])[:,0]
    # k data observation loss
    k = self(X[1])[:,1]
    # PDE loss
    _, _, _, _, pde = self.get_derivatives(X[2])
    loss = self.loss_fun(Y[0], u)\
            + self.loss_fun(Y[1], k)\
            + 0.1 * self.loss_fun(Y[2], pde)
    return loss
```

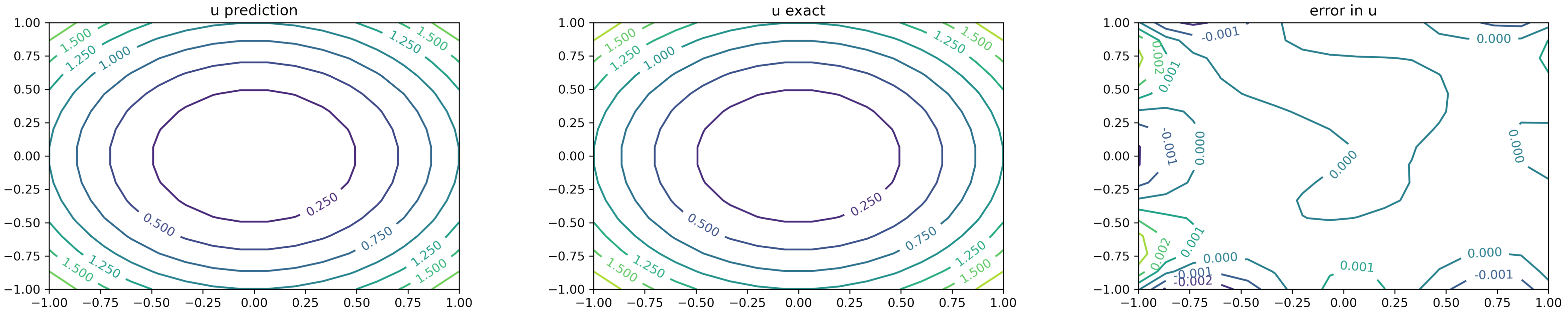
$$f = \nabla \cdot (k \nabla u)$$

If $n_{\mathrm{k}} \gg n_{\mathrm{u}}$: solving for u (forward problem).

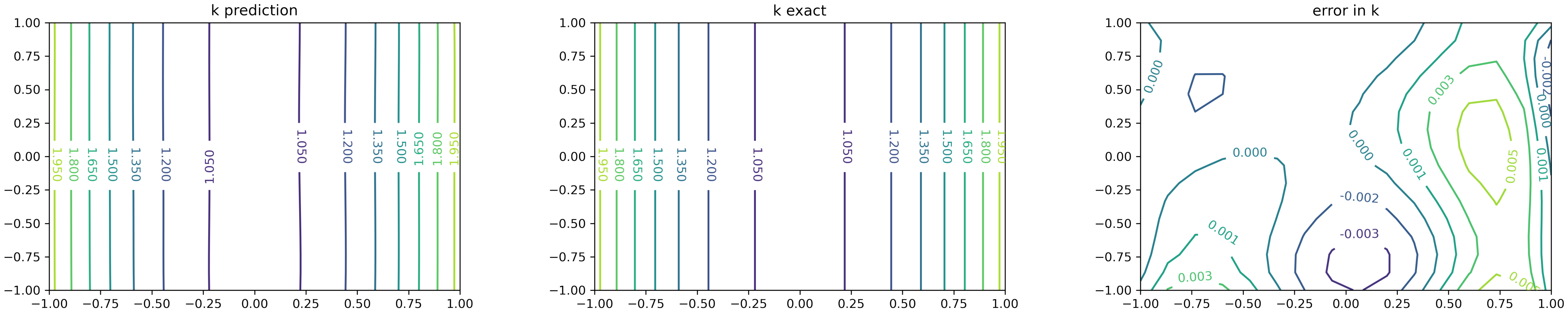
If $n_{\mathrm{u}} \gg n_{\mathrm{k}}$: solving for k (inverse problem).

If $n_{\mathrm{u}} \approx n_{\mathrm{k}}$: hybrid problem.

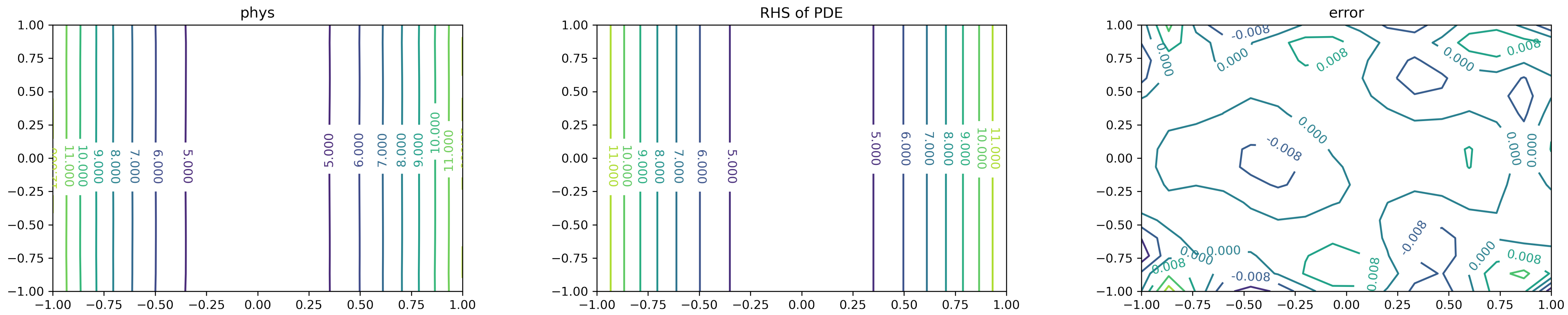
$n_{\mathrm{k}} \gg n_{\mathrm{u}}$: solving for u



$n_{\mathrm{u}} \gg n_{\mathrm{k}}$: solving for $\boldsymbol{k}$



Physics error: $(f - \nabla \cdot (k\nabla u))^2$



Final note

PIML can be very useful to solve PDEs in high-dimension.

Consider our 1D finite-difference model:

$$-u''(x) = f(x), \quad x \in [0,1]$$

$$u_i \approx u(x_i), \quad \frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i$$

If $h = 1/n$, we have n grid points x_i.

Poisson's equation in $\mathbb{R}^d$:

$$x = (x^1, \ldots, x^d)$$

$$-\triangle u(x) = f(x), \quad x \in [0,1]^d$$

In dimension d, we need n^d points for our finite-difference scheme.

This becomes quickly intractable even for moderate values of d.

However, PhysML does not require any grid.

It can evaluate

$$-\triangle u(x)$$

directly using automatic differentiation.

Assuming enough data for u and f are provided it is possible to solve high-dimensional PDEs.

A famous example is the Black-Scholes equation:

$$0 = \frac{\partial u}{\partial t} + \mu(x)\frac{\partial u}{\partial x} +$$

$$+ \frac{1}{2}\sum_{i,j=1}^{d} \rho_{ij}\sigma(x_i)\sigma(x_j)\frac{\partial^2 u}{\partial x_i \partial x_j} - ru(t,x)$$

$$x \in \mathbb{R}^d$$

Black-Scholes is a mathematical model for the dynamics of a financial market containing derivative investment instruments.

Derivative: a contract that derives its value from the performance of an underlying entity. Examples of underlying entities: asset, index, and interest rate.

Black-Scholes gives a theoretical estimate of the price of European-style options .

The key idea behind the model is to hedge the option by buying and selling the underlying asset in just the right way and, as a consequence, to eliminate risk.

Other examples of high-dimensional PDEs include:

- Hamilton–Jacobi–Bellman equation
- Allen–Cahn equation