Helmholtz2DInverse_2

December 20, 2023

[158]:
```
# Install the needed libraries
%pip install --upgrade pip
%pip install --upgrade "jax[cuda]" -f https://storage.googleapis.com/
  ↪jax-releases/jax_cuda_releases.html optax flax
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pip in /home/felip777/.local/lib/python3.10/site-
packages (23.3.1)
Note: you may need to restart the kernel to use updated packages.
Defaulting to user installation because normal site-packages is not writeable
Looking in links: https://storage.googleapis.com/jax-
releases/jax_cuda_releases.html
Requirement already satisfied: optax in
/home/felip777/.local/lib/python3.10/site-packages (0.1.7)
Requirement already satisfied: flax in
/home/felip777/.local/lib/python3.10/site-packages (0.7.5)
Requirement already satisfied: jax[cuda] in
/home/felip777/.local/lib/python3.10/site-packages (0.4.21)
Requirement already satisfied: ml-dtypes>=0.2.0 in
/home/felip777/.local/lib/python3.10/site-packages (from jax[cuda]) (0.3.1)
Requirement already satisfied: numpy>=1.22 in
/home/felip777/.local/lib/python3.10/site-packages (from jax[cuda]) (1.26.2)
Requirement already satisfied: opt-einsum in
/home/felip777/.local/lib/python3.10/site-packages (from jax[cuda]) (3.3.0)
Requirement already satisfied: scipy>=1.9 in
/home/felip777/.local/lib/python3.10/site-packages (from jax[cuda]) (1.11.4)
Requirement already satisfied: jaxlib==0.4.21+cuda11.cudnn86 in
/home/felip777/.local/lib/python3.10/site-packages (from jax[cuda])
(0.4.21+cuda11.cudnn86)
Requirement already satisfied: absl-py>=0.7.1 in
/home/felip777/.local/lib/python3.10/site-packages (from optax) (2.0.0)
Requirement already satisfied: chex>=0.1.5 in
/home/felip777/.local/lib/python3.10/site-packages (from optax) (0.1.84)
Requirement already satisfied: msgpack in /usr/lib/python3/dist-packages (from
flax) (1.0.3)
Requirement already satisfied: orbax-checkpoint in
/home/felip777/.local/lib/python3.10/site-packages (from flax) (0.4.3)
Requirement already satisfied: tensorstore in
```

```
/home/felip777/.local/lib/python3.10/site-packages (from flax) (0.1.50)
Requirement already satisfied: rich>=11.1 in
/home/felip777/.local/lib/python3.10/site-packages (from flax) (13.7.0)
Requirement already satisfied: typing-extensions>=4.2 in
/home/felip777/.local/lib/python3.10/site-packages (from flax) (4.7.1)
Requirement already satisfied: PyYAML>=5.4.1 in /usr/lib/python3/dist-packages
(from flax) (5.4.1)
Requirement already satisfied: toolz>=0.9.0 in
/home/felip777/.local/lib/python3.10/site-packages (from chex>=0.1.5->optax)
(0.12.0)
Requirement already satisfied: markdown-it-py>=2.2.0 in
/home/felip777/.local/lib/python3.10/site-packages (from rich>=11.1->flax)
(3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/home/felip777/.local/lib/python3.10/site-packages (from rich>=11.1->flax)
(2.17.2)
Requirement already satisfied: etils[epath,epy] in
/home/felip777/.local/lib/python3.10/site-packages (from orbax-checkpoint->flax)
(1.5.2)
Requirement already satisfied: nest_asyncio in
/home/felip777/.local/lib/python3.10/site-packages (from orbax-checkpoint->flax)
(1.5.7)
Requirement already satisfied: protobuf in
/home/felip777/.local/lib/python3.10/site-packages (from orbax-checkpoint->flax)
(4.23.4)
Requirement already satisfied: mdurl~=0.1 in
/home/felip777/.local/lib/python3.10/site-packages (from markdown-it-
py>=2.2.0->rich>=11.1->flax) (0.1.2)
Requirement already satisfied: fsspec in
/home/felip777/.local/lib/python3.10/site-packages (from
etils[epath,epy]->orbax-checkpoint->flax) (2023.10.0)
Requirement already satisfied: importlib_resources in
/home/felip777/.local/lib/python3.10/site-packages (from
etils[epath,epy]->orbax-checkpoint->flax) (6.1.1)
Requirement already satisfied: zipp in /usr/lib/python3/dist-packages (from
etils[epath,epy]->orbax-checkpoint->flax) (1.0.0)
Note: you may need to restart the kernel to use updated packages.
```

[159]:
```python
from typing import Callable, Tuple, Optional, Union, List, Any
from dataclasses import dataclass, field
import jax
from jax import random, grad, vmap, jit, value_and_grad
from functools import partial
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
import flax
```

```
from flax import linen as nn
from flax.training.train_state import TrainState
import optax
```

0.1 The problem at hand

We will now solve the 2D Helmholtz equation given by:

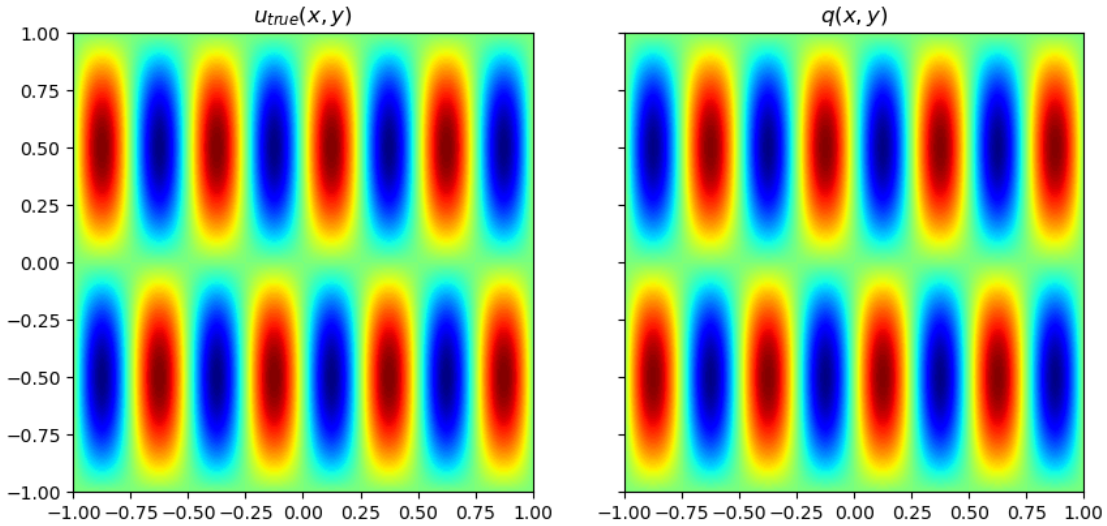
$$\Delta u + k^2 u = q(x, y), \text{with } (x, y) \in [-1, 1]^2$$

with homogeneous Dirichlet boundary conditions.

When $k = 2$

[160]:
```
a1=1
a2=4
k=2
u_true_fn = lambda x, y: jnp.sin(a1*jnp.pi * x) * jnp.sin(a2*jnp.pi * y)

q_fn = lambda x, y: \
    (-(a1 * jnp.pi)**2) * jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi * y) -␣
 ↪\
    ((a2 * jnp.pi)**2) * jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi * y) + \
    (k**2) * jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi * y)


xmin, xmax = -1.0, 1.0
ymin, ymax = -1.0, 1.0

x = jnp.linspace(xmin, xmax, 201)
y = jnp.linspace(ymin, ymax, 201)

u_true = vmap(vmap(u_true_fn, (None, 0)), (0, None))(x, y)
q = vmap(vmap(q_fn, (None, 0)), (0, None))(x, y)

fig, axs = plt.subplots(1, 2, figsize=(10, 10 / 1.618), sharex=True,␣
 ↪sharey=True)

axs[0].contourf(x, y, u_true, levels=128, cmap="jet")
axs[0].set_title(r"$u_{true}(x,y)$")
axs[1].contourf(x, y, q, levels=128, cmap="jet")
axs[1].set_title(r"$q(x,y)$")

for ax in axs:
    ax.set_aspect("equal")
# ax.plot(x, u_true, label=r'$u_{true}$')
# ax.scatter(x_meas, u_meas, label=r'$u_{meas}$', color='red')
```

```
# ax.legend()
```



0.2 Approximating the system's behavior using a Neural Network

[161]:
```
def _get_activation(activation_name):
    activations = {**jax.nn.__dict__, **jnp.__dict__}
    if activation_name in activations:
        return activations[activation_name]
    else:
        raise NotImplementedError("This activation function is not implemented␣
 ↪yet!")


class DenseLayer(nn.Module):
    features: int
    kernel_init: Callable = nn.initializers.glorot_normal()
    bias_init: Callable = nn.initializers.zeros

    @nn.compact
    def __call__(self, x):
        kernel = self.param(
            "kernel",
            self.kernel_init,  # Initialization function
            (x.shape[-1], self.features),
        )

        bias = self.param("bias", self.bias_init, (self.features,))
```

```
        return jnp.dot(x, kernel) + bias

class Mlp(nn.Module):
    arch_name: Optional[str] = "Mlp"
    num_layers: int = 4
    layer_size: int = 64
    out_dim: int = 1
    activation: str = "tanh"
    lb: List = field(default_factory=List)
    ub: List = field(default_factory=List)
    extra_params: Any = None

    def setup(self):
        self.activation_fn = _get_activation(self.activation)
        self.lb_array = jnp.asarray(self.lb)
        self.ub_array = jnp.asarray(self.ub)

    @nn.compact
    def __call__(self, *inputs):
        # Normalize the inputs
        x = (
            2.0 * (jnp.stack(inputs) - self.lb_array) / (self.ub_array - self.
 ↪lb_array)
            - 1.0
        )

        # Forward pass
        for _ in range(self.num_layers):
            x = DenseLayer(features=self.layer_size)(x)
            x = self.activation_fn(x)

        # Output layer
        x = DenseLayer(features=self.out_dim)(x)

        return x.flatten()
```

[162]:
```
# Initialize the random key
rng_key = random.PRNGKey(42)


class Helmholtz2DInv(Mlp):
    def setup(self):
        super().setup()
        self.ksq = self.param("ksq", jax.nn.initializers.uniform(scale=3.0), ())
        # sq == squared
```

```
# Define the Neural Network
model = Helmholtz2DInv(
    num_layers=4, layer_size=50, out_dim=1, lb=[xmin, ymin], ub=[xmax, ymax]
)

# Initialize the model parameters
rng_key, init_key = random.split(rng_key, 2)
dummy_input = jnp.asarray(0.0)
params = model.init(init_key, dummy_input, dummy_input)

## Define the optimizer using Optax
# The learning rate scheduler
lr = optax.exponential_decay(init_value=1e-3, transition_steps=5000,␣
 ↪decay_rate=0.98)

tx = optax.adam(learning_rate=lr)

# Create the training state
state = TrainState.create(
    apply_fn=lambda params_, x_, y_: model.apply(params_, x_, y_)[0],
    params=params,
    tx=tx,
)
```

[163]:
```
# Print the model structure in a fancy way
print(model.tabulate(rng_key, dummy_input, dummy_input))
```

Helmholtz2DInv Summary

path	module	inputs	outputs	params
	Helmholtz2DInv	- float32[]	float32[1]	ksq: float32[]
		- float32[]		
				1 (4 B)
DenseLayer_0	DenseLayer	float32[2]	float32[50]	bias: float32[50]

kernel:

float32[2,50]

150

(600 B)

DenseLayer_1 DenseLayer float32[50] float32[50]
bias:

float32[50]

kernel:

float32[50,50]

2,550

(10.2 KB)

DenseLayer_2 DenseLayer float32[50] float32[50]
bias:

float32[50]

kernel:

float32[50,50]

2,550

(10.2 KB)

DenseLayer_3 DenseLayer float32[50] float32[50]
bias:

float32[50]

kernel:

float32[50,50]

2,550

(10.2 KB)

DenseLayer_4 DenseLayer float32[50] float32[1]
bias: float32[1]

kernel:

float32[50,1]

```
                                                                    51

(204 B)


                        Total

 7,852 (31.4 KB)


                          Total Parameters: 7,852 (31.4 KB)
```

[164]:
```
state.params['params']['ksq']
```

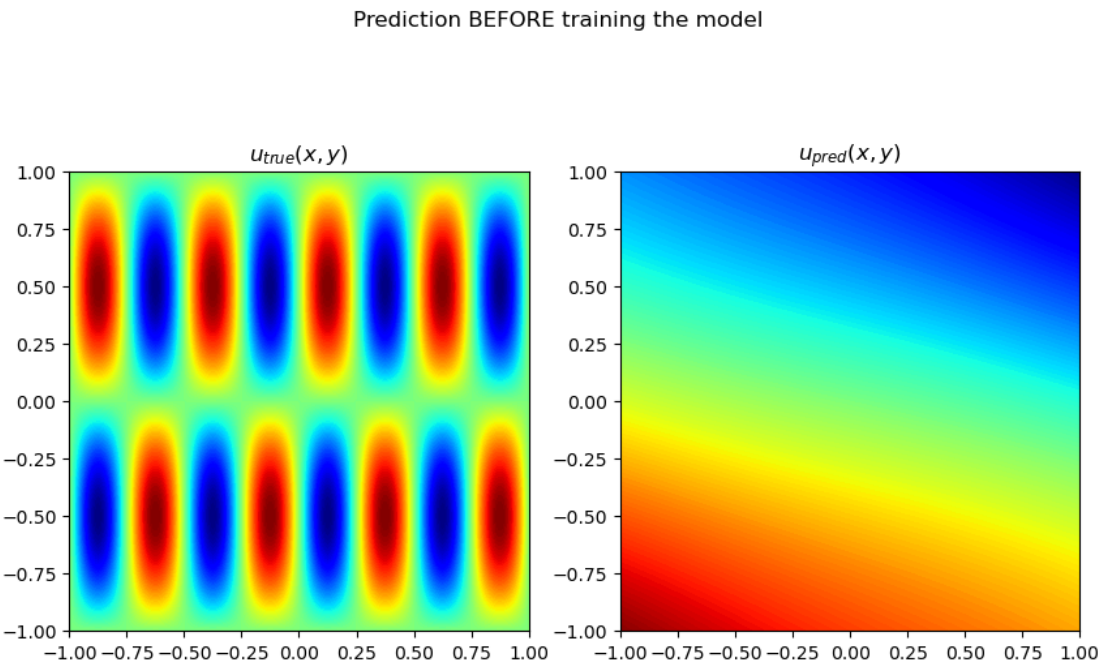
[164]:
```
Array(2.441193, dtype=float32)
```

0.3 How do we evaluate the model?

[165]:
```
# To evaluate the model for a single example, i.e., for a single x input:
x_single = 0.1
y_single = 0.1
u = state.apply_fn(state.params, x_single, y_single)
print(f"The solution for u(x={x_single}, y={y_single}) = {u}")

# The solution evaluate at the entire domain:
u_pred = vmap(vmap(state.apply_fn, (None, None, 0)), (None, 0, None))(
    state.params, x, y
)

fig, axs = plt.subplots(1, 2, figsize=(10, 10 / 1.618))

axs[0].contourf(x, y, u_true, levels=128, cmap="jet")
axs[0].set_title(r"$u_{true}(x,y)$")
axs[1].contourf(x, y, u_pred, levels=128, cmap="jet")
axs[1].set_title(r"$u_{pred}(x,y)$")

for ax in axs:
    ax.set_aspect("equal")


fig.suptitle("Prediction BEFORE training the model")
```

```
The solution for u(x=0.1, y=0.1) = -0.018735093995928764
```

[165]:
```
Text(0.5, 0.98, 'Prediction BEFORE training the model')
```



0.4 The batch sampler

This time, let's implement a sampler so we can obtain the PDE residual evaluation points randomly

[166]:
```
class Helmholtz2DSampler(object):
    def __init__(self, batch_size, dom_bds, rng_key):
        self.batch_size = batch_size
        self.key = rng_key
        self.dom_bds = dom_bds

    def __getitem__(self, index):
        self.key, subkey = random.split(self.key)
        batch = self.data_generation(subkey)
        return batch

    @partial(jit, static_argnums=(0,))
    def data_generation(self, key):
        key1, key2 = random.split(key)
        # Points for evaluating the PDE residual
        x = random.uniform(
            key1,
            shape=(self.batch_size,),
```

```
            minval=self.dom_bds["xmin"],
            maxval=self.dom_bds["xmax"],
        )
        y = random.uniform(
            key2,
            shape=(self.batch_size,),
            minval=self.dom_bds["ymin"],
            maxval=self.dom_bds["ymax"],
        )

        return {
            "x": x,
            "y": y,
            "xmin": self.dom_bds["xmin"],
            "xmax": self.dom_bds["xmax"],
            "ymin": self.dom_bds["ymin"],
            "ymax": self.dom_bds["ymax"],
        }
```

[167]:
```
rng_key, subkey = random.split(rng_key)
sampler = Helmholtz2DSampler(
    batch_size=600,
    dom_bds={"xmin": xmin, "xmax": xmax, "ymin": ymin, "ymax": ymax},
    rng_key=subkey,
)
batch_iterator = iter(sampler)
```

[168]:
```
# Define the train step function
@jit
def train_step(state, batch):
    def pde_residual_fn(params, x, y):
        u = state.apply_fn(params, x, y)
        u_xx = grad(grad(state.apply_fn, 1), 1)(params, x, y)
        u_yy = grad(grad(state.apply_fn, 2), 2)(params, x, y)
        ksq = params["params"]["ksq"]
        q=(-(a1 * jnp.pi)**2) * jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi␣
 ↪* y) - \
            ((a2 * jnp.pi)**2) * jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi␣
 ↪* y) + \
                (ksq)* jnp.sin(a1 * jnp.pi * x) * jnp.sin(a2 * jnp.pi * y)
        res = (u_xx + u_yy) + (ksq**2) * u - q
        return res, u

    # Define the loss function
    def loss_fn(params, batch):
        res_pred, u_pred = vmap(pde_residual_fn, (None, 0, 0))(
            params, batch["x"], batch["y"]
```

```
        )
        pde_loss = jnp.square(res_pred).mean()

        u_true = vmap(
            lambda x, y:  jnp.sin(jnp.pi *x*a1) * jnp.sin(jnp.pi *y*a2), (0, 0)
        )(batch["x"], batch["y"])

        u_west = vmap(state.apply_fn, (None, None, 0))(
            params, batch["xmin"], batch["y"]
        )
        u_east = vmap(state.apply_fn, (None, None, 0))(
            params, batch["xmax"], batch["y"]
        )
        u_north = vmap(state.apply_fn, (None, 0, None))(
            params, batch["x"], batch["ymax"]
        )
        u_south = vmap(state.apply_fn, (None, 0, None))(
            params, batch["x"], batch["ymin"]
        )

        data_loss = (
            jnp.square(u_pred - u_true).mean()
            + jnp.square(u_west).mean()
            + jnp.square(u_east).mean()
            + jnp.square(u_north).mean()
            + jnp.square(u_south).mean()
        )

        total_loss = 20.0*data_loss + pde_loss

        return total_loss, {
            "total_loss": total_loss,
            "data_loss": data_loss,
            "pde_loss": pde_loss,
        }

    # Compute the loss and its grads w.r.t. the model parameters
    (_, loss_components), grads = value_and_grad(loss_fn, has_aux=True)(
        state.params, batch
    )
    state = state.apply_gradients(grads=grads)
    return loss_components, state


@jit
def eval_step(state, batch):
    def l2_error(params, batch):
```

```
        u_pred = vmap(vmap(state.apply_fn, (None, None, 0)), (None, 0, None))(
            params, x, y
        )
        return jnp.linalg.norm(
            x=(u_pred.flatten() - batch["u"].flatten()), ord=2
        ) / jnp.linalg.norm(x=batch["u"].flatten(), ord=2)

    return l2_error(state.params, batch)
```

[169]:
```
total_loss_log = []
data_loss_log = []
pde_loss_log = []
epoch_log = []
error_log = []
k_log = []

# Since this is a very small example, we don't even need to create a fancy␣
 ↪sampler for batching our dataset.
# train_batch = {"x": x_meas, "u": u_meas, "x_pde": jnp.linspace(0, 6, 101)}
eval_batch = {"x": x, "y": y, "u": u_true}

for epoch in range(40000):
    train_batch = next(batch_iterator)
    loss, state = train_step(state, train_batch)
    if epoch % 1000 == 0:
        l2_error = eval_step(state, eval_batch)
        k = jnp.sqrt(state.params["params"]["ksq"])
        epoch_log.append(epoch)
        total_loss_log.append(loss["total_loss"])
        data_loss_log.append(loss["data_loss"])
        pde_loss_log.append(loss["pde_loss"])
        error_log.append(l2_error)
        k_log.append(k)

        print(
            f"Epoch: {epoch} -- Total Loss: {loss['total_loss']}  -- Data Loss:␣
 ↪{loss['data_loss']} -- PDE Loss: {loss['pde_loss']} -- Error: {l2_error} --␣
 ↪k: {k}"
        )
```

```
Epoch: 0 -- Total Loss: 6797.87890625  -- Data Loss: 0.3096241354942322 -- PDE
Loss: 6791.6865234375 -- Error: 1.0195094347000122 -- k: 1.5627517700195312
Epoch: 1000 -- Total Loss: 528.268310546875  -- Data Loss: 1.6454623937606812 --
PDE Loss: 495.35906982421875 -- Error: 1.3685932159423828 -- k:
1.8331280946731567
Epoch: 2000 -- Total Loss: 8.448742866516113  -- Data Loss: 0.11265216767787933
-- PDE Loss: 6.195699691772461 -- Error: 0.18494737148284912 -- k:
```

```
1.8360739946365356
Epoch: 3000 -- Total Loss: 3.6254732608795166  -- Data Loss: 0.03937627747654915
-- PDE Loss: 2.8379476070404053 -- Error: 0.11768965423107147 -- k:
1.8287278413772583
Epoch: 4000 -- Total Loss: 1.588897705078125  -- Data Loss: 0.01604348048567772
-- PDE Loss: 1.2680281400680542 -- Error: 0.07369387149810791 -- k:
1.8236396312713623
Epoch: 5000 -- Total Loss: 2.8286354541778564  -- Data Loss:
0.010648627765476704 -- PDE Loss: 2.6156628131866455 -- Error:
0.06355784833431244 -- k: 1.8193347454071045
Epoch: 6000 -- Total Loss: 0.9756120443344116  -- Data Loss:
0.008164326660335064 -- PDE Loss: 0.8123255372047424 -- Error:
0.06075014919042587 -- k: 1.8151599168777466
Epoch: 7000 -- Total Loss: 2.7438862323760986  -- Data Loss: 0.00672823004424572
-- PDE Loss: 2.6093215942382812 -- Error: 0.06522220373153687 -- k:
1.8104075193405151
Epoch: 8000 -- Total Loss: 3.1936724185943604  -- Data Loss:
0.004064336884766817 -- PDE Loss: 3.1123857498168945 -- Error:
0.06467755883932114 -- k: 1.8052175045013428
Epoch: 9000 -- Total Loss: 0.4617612659931183  -- Data Loss:
0.0035769694950431585 -- PDE Loss: 0.39022186398506165 -- Error:
0.05369272083044052 -- k: 1.7982027530670166
Epoch: 10000 -- Total Loss: 0.40263622999191284  -- Data Loss:
0.002866726368665695 -- PDE Loss: 0.34530168771743774 -- Error:
0.051452331244945526 -- k: 1.7900927066802979
Epoch: 11000 -- Total Loss: 1.0510622262954712  -- Data Loss:
0.0030203380156308413 -- PDE Loss: 0.9906554222106934 -- Error:
0.05595741048455238 -- k: 1.78103768825531
Epoch: 12000 -- Total Loss: 0.3075300455093384  -- Data Loss:
0.0023518733214586973 -- PDE Loss: 0.26049259305000305 -- Error:
0.04597628116607666 -- k: 1.7711957693099976
Epoch: 13000 -- Total Loss: 1.0034782886505127  -- Data Loss:
0.0035116036888211966 -- PDE Loss: 0.9332462549209595 -- Error:
0.0546741709113121 -- k: 1.7604317665100098
Epoch: 14000 -- Total Loss: 0.8230326771736145  -- Data Loss:
0.00277152843773365 -- PDE Loss: 0.7676020860671997 -- Error:
0.06257787346839905 -- k: 1.7508360147476196
Epoch: 15000 -- Total Loss: 0.24443425238132477  -- Data Loss:
0.00186527194455266 -- PDE Loss: 0.20712880790233612 -- Error:
0.04440108314156532 -- k: 1.7439758777618408
Epoch: 16000 -- Total Loss: 0.29474952816963196  -- Data Loss:
0.002059028949588537 -- PDE Loss: 0.25356894731521606 -- Error:
0.044251035898923874 -- k: 1.7440963983535767
Epoch: 17000 -- Total Loss: 1.486698865890503  -- Data Loss:
0.002369748428463936 -- PDE Loss: 1.4393038749694824 -- Error:
0.05395762622356415 -- k: 1.7321624755859375
Epoch: 18000 -- Total Loss: 0.16598328948020935  -- Data Loss:
0.0015706116100773215 -- PDE Loss: 0.13457106053829193 -- Error:
```

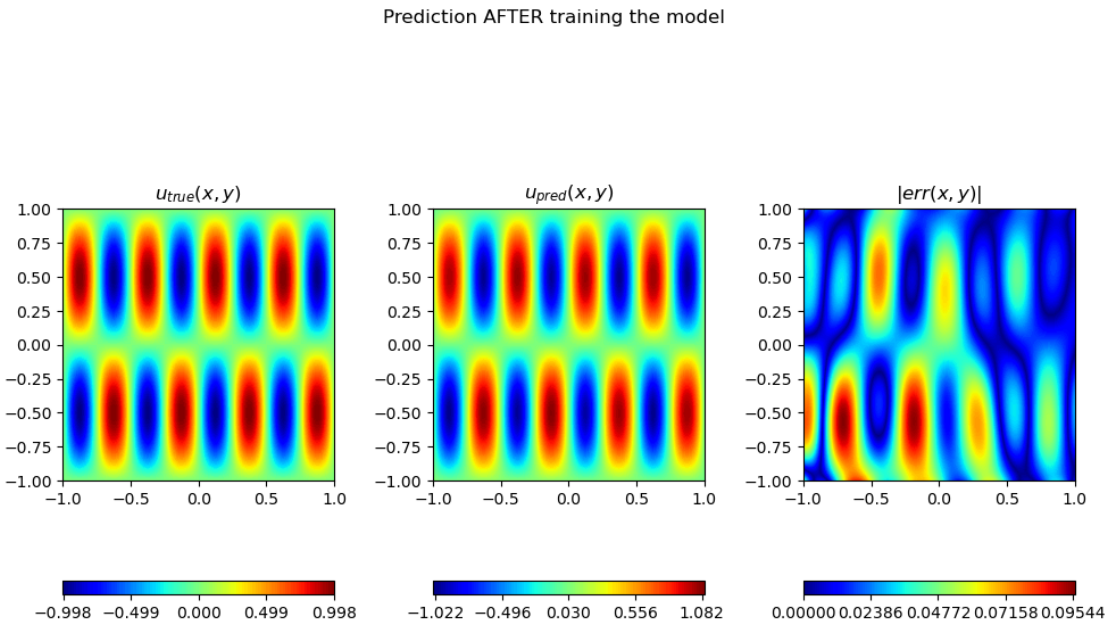
```
0.03955838456749916 -- k: 1.7260757684707642
Epoch: 19000 -- Total Loss: 0.5193566679954529  -- Data Loss:
0.001403404981829226 -- PDE Loss: 0.49128857254981995 -- Error:
0.041637714952230453 -- k: 1.7184627056121826
Epoch: 20000 -- Total Loss: 0.3783329725265503  -- Data Loss:
0.0014378865016624331 -- PDE Loss: 0.3495752513408661 -- Error:
0.04157109186053276 -- k: 1.7121038436889648
Epoch: 21000 -- Total Loss: 0.5493951439857483  -- Data Loss:
0.0013701880816370249 -- PDE Loss: 0.5219913721084595 -- Error:
0.04420984163880348 -- k: 1.707667350769043
Epoch: 22000 -- Total Loss: 0.5762460231781006  -- Data Loss:
0.00178213429171592 -- PDE Loss: 0.5406033396720886 -- Error:
0.057171594351530075 -- k: 1.7010517120361328
Epoch: 23000 -- Total Loss: 0.92247074842453  -- Data Loss:
0.0013931136345490813 -- PDE Loss: 0.8946084976196289 -- Error:
0.044254470616579056 -- k: 1.6973204612731934
Epoch: 24000 -- Total Loss: 0.44805780053138733  -- Data Loss:
0.0020161019638180733 -- PDE Loss: 0.40773576498031616 -- Error:
0.04223453253507614 -- k: 1.6924622058868408
Epoch: 25000 -- Total Loss: 0.645944356918335  -- Data Loss:
0.0015133386477828026 -- PDE Loss: 0.6156775951385498 -- Error:
0.038437578827142715 -- k: 1.6882951259613037
Epoch: 26000 -- Total Loss: 0.26819172501564026  -- Data Loss:
0.0012308545410633087 -- PDE Loss: 0.24357463419437408 -- Error:
0.038351885974407196 -- k: 1.6901381015777588
Epoch: 27000 -- Total Loss: 0.14608587324619293  -- Data Loss:
0.0009565852233208716 -- PDE Loss: 0.12695416808128357 -- Error:
0.03542248532176018 -- k: 1.6827951669692993
Epoch: 28000 -- Total Loss: 0.10398967564105988  -- Data Loss:
0.0009136743610724807 -- PDE Loss: 0.08571618795394897 -- Error:
0.0360189750790596 -- k: 1.6799602508544922
Epoch: 29000 -- Total Loss: 0.1465269774198532  -- Data Loss:
0.001488072914071381 -- PDE Loss: 0.11676552146673203 -- Error:
0.03433474525809288 -- k: 1.674474835395813
Epoch: 30000 -- Total Loss: 0.44463756680488586  -- Data Loss:
0.0011518171522766352 -- PDE Loss: 0.42160123586654663 -- Error:
0.035843830555677414 -- k: 1.6699198484420776
Epoch: 31000 -- Total Loss: 0.06689677387475967  -- Data Loss:
0.0008353455341421068 -- PDE Loss: 0.05018986016511917 -- Error:
0.03329624608159065 -- k: 1.6689828634262085
Epoch: 32000 -- Total Loss: 0.4563542306423187  -- Data Loss:
0.0011923554120585322 -- PDE Loss: 0.43250712752342224 -- Error:
0.034108202904462814 -- k: 1.665097713470459
Epoch: 33000 -- Total Loss: 0.10683517903089523  -- Data Loss:
0.0009234505705535412 -- PDE Loss: 0.08836616575717926 -- Error:
0.03344238921999931 -- k: 1.66269850730896
Epoch: 34000 -- Total Loss: 0.2980842888355255  -- Data Loss:
0.001084063434973359 -- PDE Loss: 0.27640300989151 -- Error:
```

```
0.034765176475048065 -- k: 1.660097599029541
Epoch: 35000 -- Total Loss: 0.13849858939647675  -- Data Loss:
0.0012124419445171952 -- PDE Loss: 0.11424974352121353 -- Error:
0.034803248941898346 -- k: 1.6561559438705444
Epoch: 36000 -- Total Loss: 0.6075961589813232  -- Data Loss:
0.0014676592545583844 -- PDE Loss: 0.5782429575920105 -- Error:
0.04207798093557358 -- k: 1.6551806926727295
Epoch: 37000 -- Total Loss: 0.11245809495449066  -- Data Loss:
0.0008806429104879498 -- PDE Loss: 0.0948452353477478 -- Error:
0.031729284673929214 -- k: 1.6520397663116455
Epoch: 38000 -- Total Loss: 0.7128930687904358  -- Data Loss:
0.0014846334233880043 -- PDE Loss: 0.6832004189491272 -- Error:
0.033788129687309265 -- k: 1.649876594543457
Epoch: 39000 -- Total Loss: 0.1632639616727829  -- Data Loss:
0.0006996163865551353 -- PDE Loss: 0.1492716372013092 -- Error:
0.030443917959928513 -- k: 1.6460901498794556
```

[170]:
```
u_pred = vmap(vmap(state.apply_fn, (None, None, 0)), (None, 0, None))(
    state.params, x, y
)

fig, axs = plt.subplots(1, 3, figsize=(10, 10 / 1.618))

im = axs[0].contourf(x, y, u_true, levels=128, cmap="jet")

fig.colorbar(
    im, ax=axs[0], location="bottom", ticks=np.linspace(u_true.min(), u_true.
 ↪max(), 5)
)

axs[0].set_title(r"$u_{true}(x,y)$")

im = axs[1].contourf(x, y, u_pred, levels=128, cmap="jet")
fig.colorbar(
    im, ax=axs[1], location="bottom", ticks=np.linspace(u_pred.min(), u_pred.
 ↪max(), 5)
)
axs[1].set_title(r"$u_{pred}(x,y)$")

error = jnp.abs(u_pred - u_true)
im = axs[2].contourf(x, y, error, levels=128, cmap="jet")
fig.colorbar(
    im, ax=axs[2], location="bottom", ticks=np.linspace(error.min(), error.
 ↪max(), 5)
)
axs[2].set_title(r"$|err(x,y)|$")
```

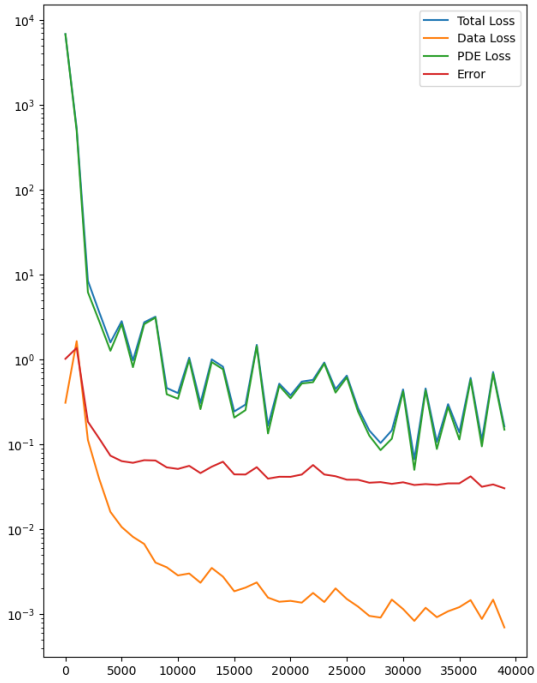
```
for ax in axs:
    ax.set_aspect("equal")

fig.tight_layout()
fig.suptitle("Prediction AFTER training the model")
```

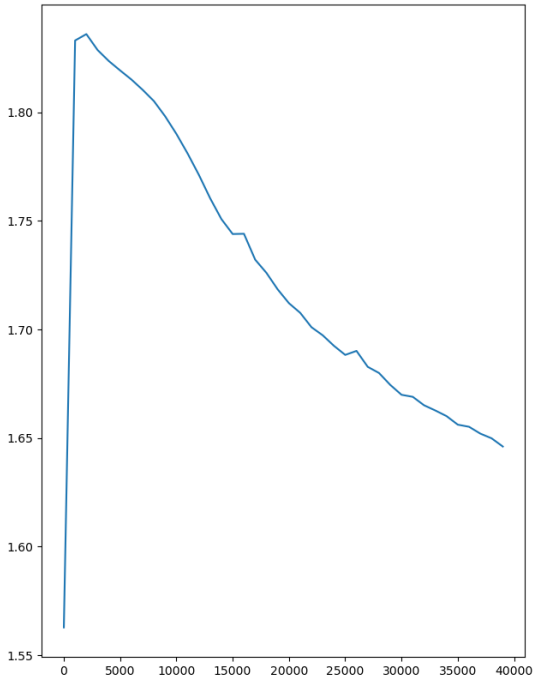
[170]: Text(0.5, 0.98, 'Prediction AFTER training the model')



[171]:
```
fig, axs = plt.subplots(1, 2, figsize=(16, 16 / 1.618))

axs[0].plot(epoch_log, total_loss_log, label="Total Loss")
axs[0].plot(epoch_log, data_loss_log, label="Data Loss")
axs[0].plot(epoch_log, pde_loss_log, label="PDE Loss")
axs[0].plot(epoch_log, error_log, label="Error")

axs[1].plot(epoch_log, k_log)

axs[0].set_yscale("log")
axs[0].legend()
```

[171]: <matplotlib.legend.Legend at 0x7f892d58e890>


Total Loss
Data Loss
PDE Loss
Error
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
0
5000
10000
15000
20000
25000
30000
35000
40000


1.80
1.75
1.70
1.65
1.60
1.55
0
5000
10000
15000
20000
25000
30000
35000
40000