pinn-2

April 23, 2024

[3]:
```
from typing import Callable
import argparse
import matplotlib.pyplot as plt
import torch
from torch import nn
import numpy as np
import torchopt
from collections import OrderedDict
from torch.func import functional_call, grad, vmap
```

[4]:
```
class LinearNN(nn.Module):
    def __init__(self,
                 num_inputs: int = 1,
                 num_layers: int = 1,
                 num_neurons: int = 5,
                 act: nn.Module = nn.Tanh())->None:
        super().__init__()
        self.num_inputs = num_inputs
        self.num_neurons = num_neurons
        self.num_layers = num_layers
        layers = []
        layers.append(nn.Linear(self.num_inputs, num_neurons))
        for _ in range(num_layers):
            layers.extend([nn.Linear(num_neurons, num_neurons), act])
        layers.append(nn.Linear(num_neurons, 1))#output
        self.network = nn.Sequential(*layers)
    def forward(self, x:torch.Tensor)->torch.Tensor:
        return self.network(x.reshape(-1,1)).squeeze()
```

[8]:
```
def make_forward_fn(model: nn.Module, derivative_order: int =␣
↪1)->list[Callable]:
    def f(x: torch.Tensor, params: dict[str, torch.nn.Parameter] | tuple[torch.
↪nn.Parameter, ...]) -> torch.Tensor:
        if isinstance(params, tuple):
            params_dict = tuple_to_dict_parameters(model, params)
        else:
            params_dict = params
```

```
        return functional_call(model, params_dict, (x, ))
    fns = []
    fns.append(f)
    dfunc = f
    for _ in range(derivative_order):
        dfunc = grad(dfunc)
        dfunc_vmap = vmap(dfunc, in_dims = (0,None))
        fns.append(dfunc_vmap)
    return fns
```

[5]:
```
def tuple_to_dict_parameters(
        model: nn.Module, params: tuple[torch.nn.Parameter, ...]
) -> OrderedDict[str, torch.nn.Parameter]:
    keys = list(dict(model.named_parameters()).keys())
    values = list(params)
    return OrderedDict(({k:v for k,v in zip(keys, values)}))
```

[9]:
```
if __name__ == "__main__":
    model = LinearNN(num_layers=2)
    fns = make_forward_fn(model, derivative_order=2)

    batch_size = 10
    x = torch.randn(batch_size)
    params = dict(model.named_parameters())

    fn_x = fns[0](x, params)
    assert fn_x.shape[0] == batch_size

    dfn_x = fns[1](x, params)
    assert dfn_x.shape[0] == batch_size

    ddfn_x = fns[2](x, params)
    assert ddfn_x.shape[0] == batch_size
```

[10]:
```
R = 1.0
X_B= 0.0
F_B= 0.5
```

[11]:
```
def make_loss_fn(f: Callable, dfdx: Callable) -> Callable:
    def loss_fn(params: torch.Tensor, x: torch.Tensor):

        # interior loss
        f_value = f(x, params)
        interior = dfdx(x, params) - R * f_value * (1 - f_value)

        # boundary loss
```

```
        x0 = X_B
        f0 = F_B
        x_boundary = torch.tensor([x0])
        f_boundary = torch.tensor([f0])
        boundary = f(x_boundary, params) - f_boundary

        loss = nn.MSELoss()
        loss_value = loss(interior, torch.zeros_like(interior)) + loss(
            boundary, torch.zeros_like(boundary)
        )

        return loss_value

    return loss_fn
```

[15]:
```
torch.manual_seed(42)
parser = argparse.ArgumentParser()
parser.add_argument("-n", "--num-hidden", type=int, default=5)
parser.add_argument("-d", "--dim-hidden", type=int, default=5)
parser.add_argument("-b", "--batch-size", type=int, default=30)
parser.add_argument("-lr", "--learning-rate", type=float, default=1e-1)
parser.add_argument("-e", "--num-epochs", type=int, default=100)
args, unknown = parser.parse_known_args()
num_hidden = args.num_hidden
dim_hidden = args.dim_hidden
batch_size = args.batch_size
num_iter = args.num_epochs
tolerance = 1e-8
learning_rate = args.learning_rate
domain = (-5.0, 5.0)
```

[17]:
```
model = LinearNN(num_layers=num_hidden, num_neurons=dim_hidden, num_inputs=1)
funcs = make_forward_fn(model, derivative_order=1)
f = funcs[0]
dfdx = funcs[1]
loss_fn = make_loss_fn(f, dfdx)
optimizer = torchopt.FuncOptimizer(torchopt.adam(lr=learning_rate))
params = tuple(model.parameters())
loss_evolution = []
for i in range(num_iter):
    x = torch.FloatTensor(batch_size).uniform_(domain[0], domain[1])
    loss = loss_fn(params, x)
    params = optimizer.step(loss, params)
    print(f"Iteration {i} with loss {float(loss)}")
    loss_evolution.append(float(loss))
x_eval = torch.linspace(domain[0], domain[1], steps=100).reshape(-1, 1)
f_eval = f(x_eval, params)
```

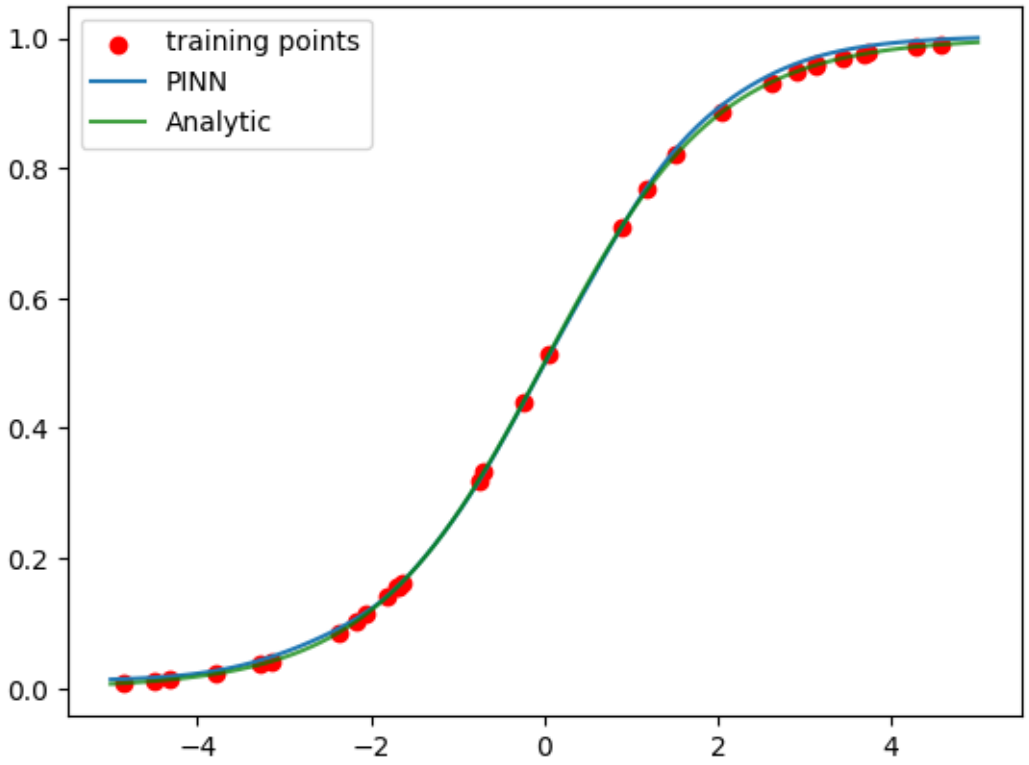
```
analytical_sol_fn = lambda x: 1.0 / (1.0 + (1.0/F_B - 1.0) * np.exp(-R * x))
x_eval_np = x_eval.detach().numpy()
x_sample_np = torch.FloatTensor(batch_size).uniform_(domain[0], domain[1]).
  ↪detach().numpy()
```

```
Iteration 0 with loss 0.6875214576721191
Iteration 1 with loss 0.07539822161197662
Iteration 2 with loss 0.0613323450088501
Iteration 3 with loss 0.06273967027664185
Iteration 4 with loss 0.060336850583553314
Iteration 5 with loss 0.06283891201019287
Iteration 6 with loss 0.06132698059082031
Iteration 7 with loss 0.055478230118751526
Iteration 8 with loss 0.056064020842313766
Iteration 9 with loss 0.04689629748463631
Iteration 10 with loss 0.034501511603593826
Iteration 11 with loss 0.023010820150375366
Iteration 12 with loss 0.01398435514420271
Iteration 13 with loss 0.040047064423561096
Iteration 14 with loss 0.018317606300115585
Iteration 15 with loss 0.011284511536359787
Iteration 16 with loss 0.022073861211538315
Iteration 17 with loss 0.02971693128347397
Iteration 18 with loss 0.020954838022589684
Iteration 19 with loss 0.01573588326573372
Iteration 20 with loss 0.013831223361194134
Iteration 21 with loss 0.006837497465312481
Iteration 22 with loss 0.013024188578128815
Iteration 23 with loss 0.015599016100168228
Iteration 24 with loss 0.0032003475353121758
Iteration 25 with loss 0.007893401198089123
Iteration 26 with loss 0.014230795204639435
Iteration 27 with loss 0.013726560398936272
Iteration 28 with loss 0.009232047945261002
Iteration 29 with loss 0.008101379498839378
Iteration 30 with loss 0.0031866386998444796
Iteration 31 with loss 0.003127886913716793
Iteration 32 with loss 0.004407278727740049
Iteration 33 with loss 0.006560429465025663
Iteration 34 with loss 0.008410363458096981
Iteration 35 with loss 0.005242552608251572
Iteration 36 with loss 0.0057198842987418175
Iteration 37 with loss 0.003596833674237132
Iteration 38 with loss 0.0022838586010038853
Iteration 39 with loss 0.00464941281825304
Iteration 40 with loss 0.0027761550154536963
Iteration 41 with loss 0.0010910824639722705
```

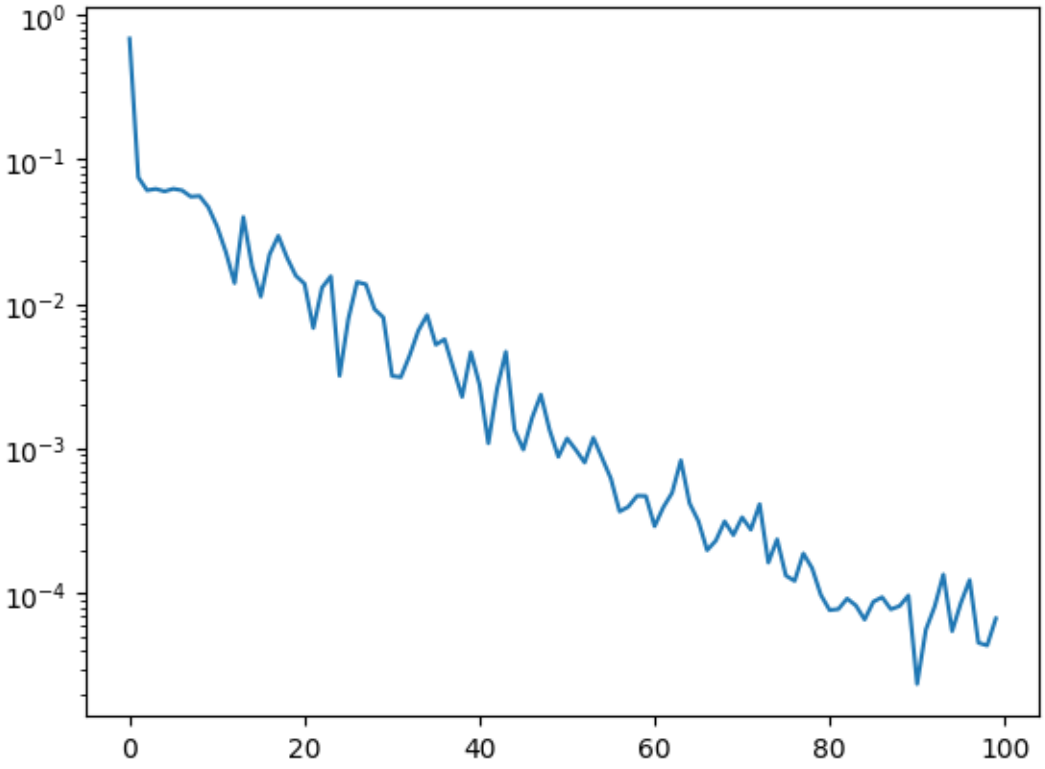
```
Iteration 42 with loss 0.002609358634799719
Iteration 43 with loss 0.0046862210147082806
Iteration 44 with loss 0.001347860088571906
Iteration 45 with loss 0.0009862588485702872
Iteration 46 with loss 0.0016334026586264372
Iteration 47 with loss 0.0023678955622017384
Iteration 48 with loss 0.0013480997877195477
Iteration 49 with loss 0.00088052375940606
Iteration 50 with loss 0.0011786948889493942
Iteration 51 with loss 0.0009816037490963936
Iteration 52 with loss 0.000802985392510891
Iteration 53 with loss 0.001190709532238543
Iteration 54 with loss 0.0008626162307336926
Iteration 55 with loss 0.0006226187106221914
Iteration 56 with loss 0.0003678762586787343
Iteration 57 with loss 0.0003967300581280142
Iteration 58 with loss 0.0004725415783468634
Iteration 59 with loss 0.0004704086168203503
Iteration 60 with loss 0.0002909587637986988
Iteration 61 with loss 0.00039411787292920053
Iteration 62 with loss 0.0004952283343300223
Iteration 63 with loss 0.0008313889265991747
Iteration 64 with loss 0.00041895825415849686
Iteration 65 with loss 0.00031620822846889496
Iteration 66 with loss 0.0001985271373996511
Iteration 67 with loss 0.0002311652060598135
Iteration 68 with loss 0.0003141782362945378
Iteration 69 with loss 0.00025299814296886325
Iteration 70 with loss 0.00033503465238027275
Iteration 71 with loss 0.000275611033430323
Iteration 72 with loss 0.00041377960587851703
Iteration 73 with loss 0.00016295319073833525
Iteration 74 with loss 0.0002361411607125774
Iteration 75 with loss 0.00013253075303509831
Iteration 76 with loss 0.00012168133980594575
Iteration 77 with loss 0.00018844756414182484
Iteration 78 with loss 0.0001490379509050399
Iteration 79 with loss 9.73524438450113e-05
Iteration 80 with loss 7.64491269364953e-05
Iteration 81 with loss 7.766728231217712e-05
Iteration 82 with loss 9.185601811623201e-05
Iteration 83 with loss 8.228338992921636e-05
Iteration 84 with loss 6.575234147021547e-05
Iteration 85 with loss 8.74139295774512e-05
Iteration 86 with loss 9.388646139996126e-05
Iteration 87 with loss 7.744751201244071e-05
Iteration 88 with loss 8.162077574525028e-05
Iteration 89 with loss 9.647205297369 51e-05
```

```
Iteration 90 with loss 2.35798488574801e-05
Iteration 91 with loss 5.603784302365966e-05
Iteration 92 with loss 8.072312630247325e-05
Iteration 93 with loss 0.00013489340199157596
Iteration 94 with loss 5.46908522665035e-05
Iteration 95 with loss 8.57450213516131e-05
Iteration 96 with loss 0.0001235880918102339
Iteration 97 with loss 4.546347554423846e-05
Iteration 98 with loss 4.3530530092539266e-05
Iteration 99 with loss 6.692014721920714e-05
```

[19]:
```
plt.scatter(x_sample_np, analytical_sol_fn(x_sample_np), color="r",␣
 ↪label="training points")
plt.plot(x_eval_np, f_eval.detach().numpy(), label="PINN")
plt.plot(
        x_eval_np,
        analytical_sol_fn(x_eval_np),
        label=f"Analytic",
        color="g",
        alpha=0.75,
    )
plt.legend()
plt.show()
```



[21]:
```
plt.semilogy(loss_evolution)
plt.show()
```



#PINNS IMPLEMENTATION ON FLUID FLOW (model fluid flow around a cylinder)

governing equation: / t + ½ * | |^2 = H (bernoulli equation) where:

1. (psi) is the velocity potential (a scalar field related to velocity) 2.t is time 3. represents the gradient of 4. H is a constant representing the total head (combination of pressure and kinetic energy)

[35]:
```
import torch
from torch import nn
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
```

[2]:
```
class CylinderFlowPINN(nn.Module):
  def __init__(self, input_dim, output_dim):
      super(CylinderFlowPINN, self).__init__()
      self.layers = nn.Sequential(
```

```
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32,32),
            nn.ReLU(),
            nn.Linear(32, output_dim)
        )
    def forward(self, x):
        psi = self.layers(x)
        return psi
```

[24]:
```
def bernoulli_residual(psi, H):
    psi.requires_grad_(True)
    psi_grad = torch.autograd.grad(psi.sum(), psi, create_graph=True)[0]
    kinetic_energy = 0.5 * torch.linalg.norm(psi_grad, dim=1)**2
    residual = kinetic_energy + psi - H
    return torch.mean(torch.abs(residual))
```

[25]:
```
def generate_data(num_points, radius):
    X = np.random.rand(num_points, 2)*2 -1
    y = np.zeros(num_points)
    for i in range(num_points):
        x, y_val = X[i]
        if np.linalg.norm([x,y_val])< radius:
            y[i] = 1#inside
    return torch.tensor(X, dtype = torch.float), torch.tensor(y, dtype = torch.
    ↪float)
```

[33]:
```
def train(num_epochs=num_epochs):
    losses = []
    progress_bar = tqdm(range(num_epochs), desc='Training', leave=True)

    for epoch in progress_bar:
        psi_pred = model(data_points)
        data_loss = torch.nn.functional.mse_loss(psi_pred, boundary_points.
    ↪unsqueeze(1))
        physics_loss = bernoulli_residual(psi_pred, H)
        net_loss = alpha * data_loss + beta * physics_loss

        optimizer.zero_grad()
        net_loss.backward()
        optimizer.step()

        current_loss = torch.mean(net_loss.detach())
        losses.append(current_loss.item())

        progress_bar.set_description(f'Epoch {epoch}: Loss = {current_loss:.
    ↪4f}')
```

```
    return losses
```

[97]:
```
input_dim = 2
output_dim = 1
num_epochs = 2000
learning_rate = 0.001
alpha = 0.5
beta = 0.5
H = 0.5
radius = 0.5
num_data_points = 1000
```

[98]:
```
data_points, boundary_points = generate_data(num_data_points, radius)
```

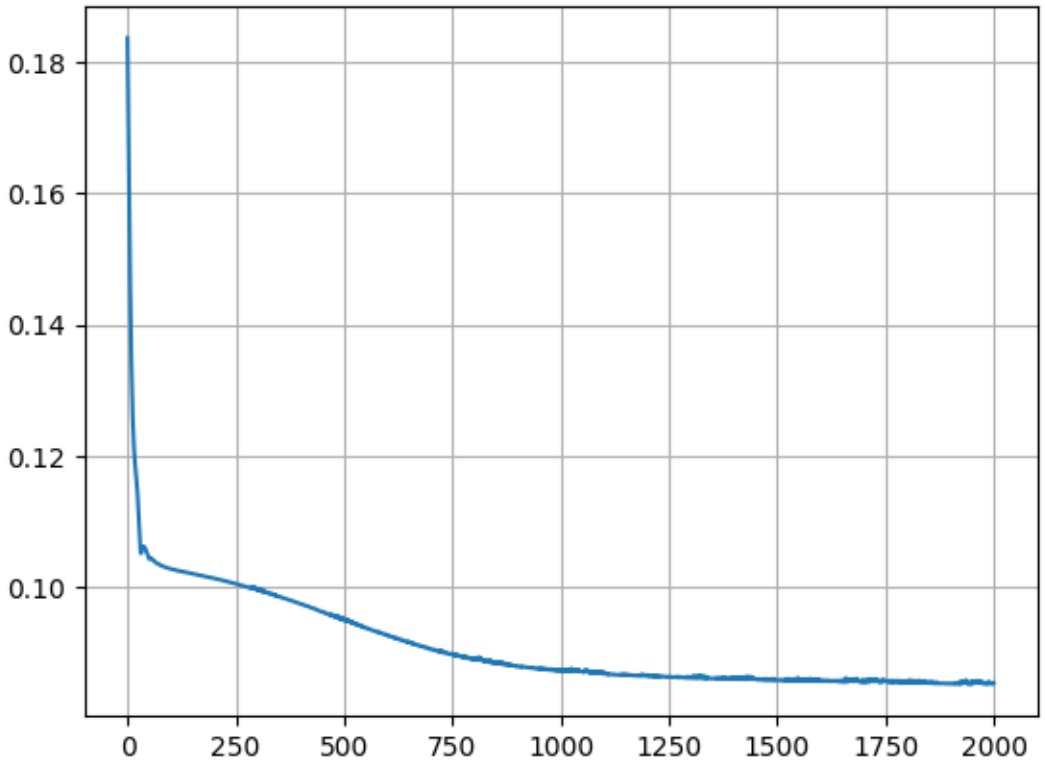
[99]:
```
model = CylinderFlowPINN(input_dim, output_dim)
```

[100]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[101]:
```
loss_history = train()
```

```
Epoch 1999: Loss = 0.0855: 100%|      | 2000/2000 [00:35<00:00, 56.98it/s]
```

[102]:
```
plt.plot(loss_history)
plt.grid()
plt.show()
```



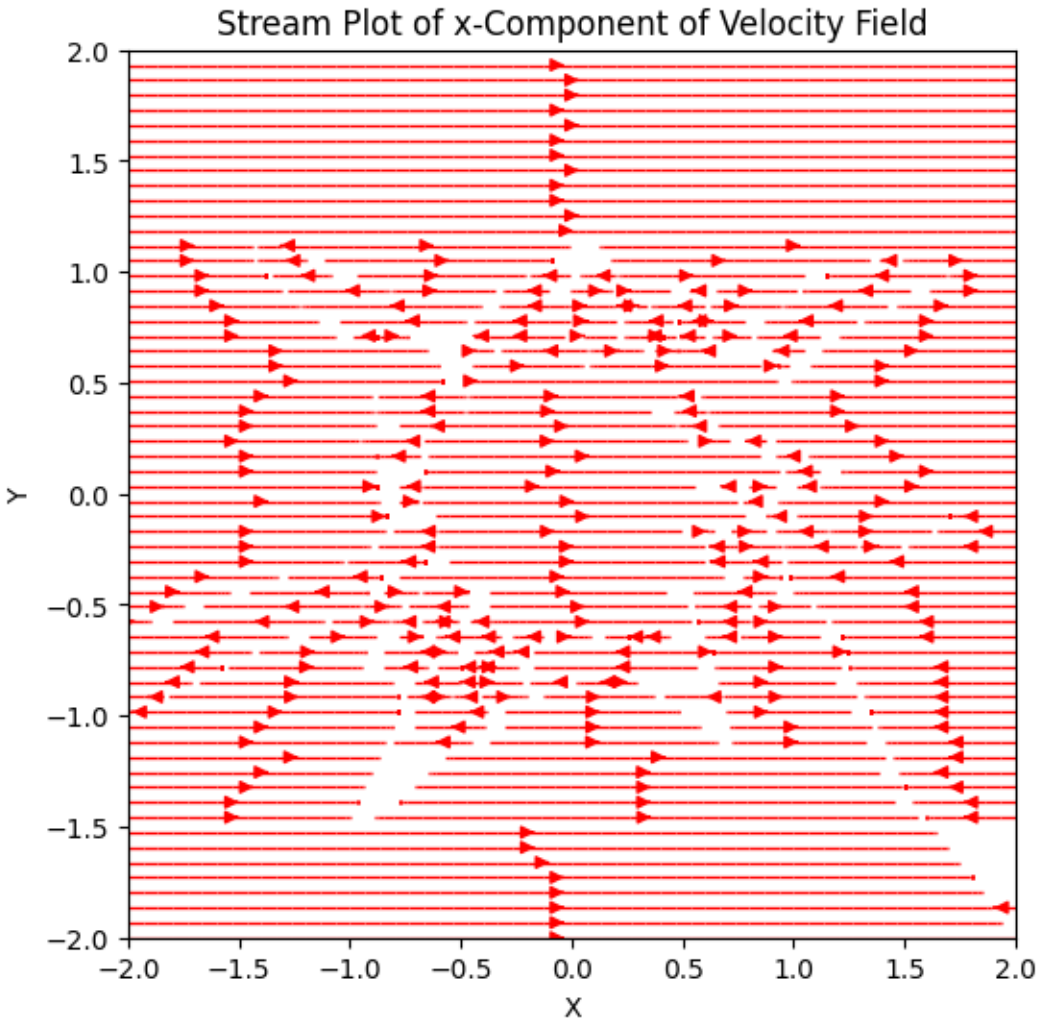
[104]:
```
x_min, x_max = -2.0, 2.0
y_min, y_max = -2.0, 2.0
num_points = 100
x = np.linspace(x_min, x_max, num_points)
y = np.linspace(y_min, y_max, num_points)
X, Y = np.meshgrid(x, y)
grid_points = torch.tensor(np.column_stack([X.flatten(), Y.flatten()]),␣
 ↪dtype=torch.float32)
```

[105]:
```
with torch.no_grad():
    velocity_field = model(grid_points).numpy()
```

[106]:
```
U = velocity_field[:, 0].reshape(num_points, num_points)
V = np.zeros_like(U)
```

[107]:
```
plt.figure(figsize=(8, 6))
plt.streamplot(X, Y, U, V, color='r', density=2.0, linewidth=1)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Stream Plot of x-Component of Velocity Field')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
```

```
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

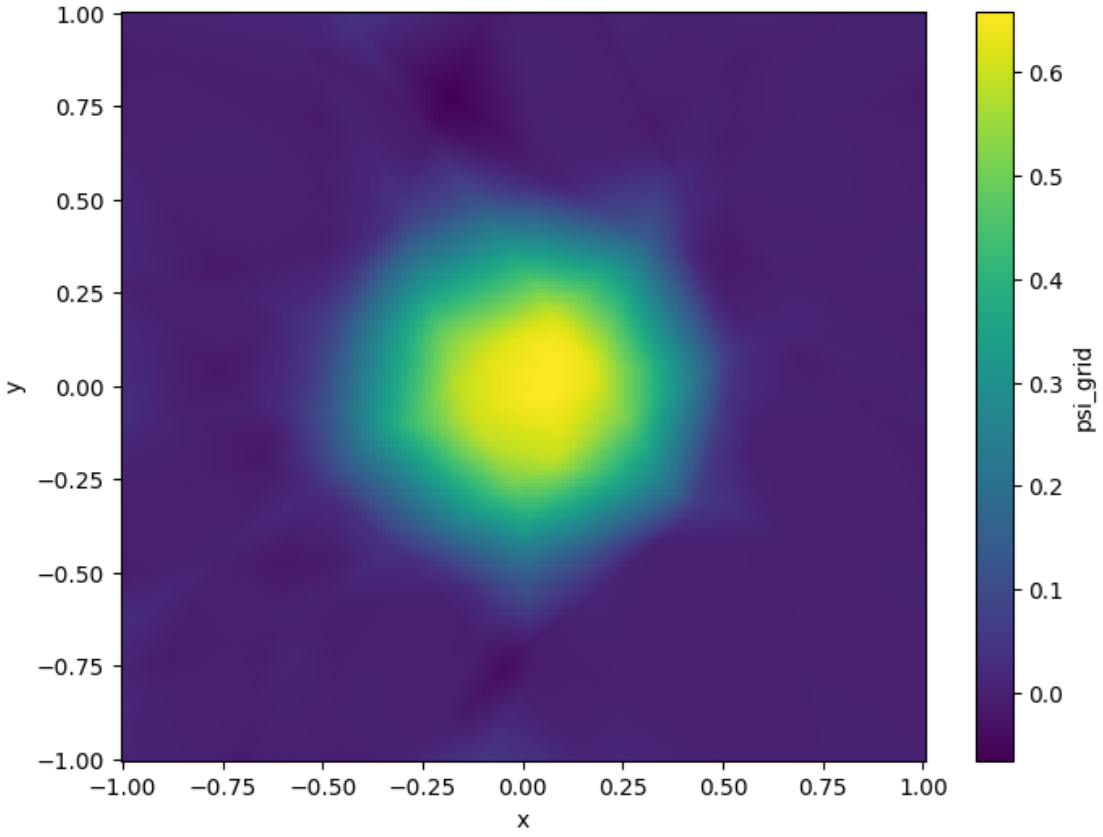


[108]:
```
x_grid = np.linspace(-1, 1, 200)
y_grid, x_grid = np.meshgrid(x_grid, x_grid)
grid_points = torch.tensor(np.vstack([x_grid.ravel(), y_grid.ravel()]).T,␣
  ↪dtype=torch.float)
psi_grid = model(grid_points)
```

[109]:
```
psi_grid = psi_grid.detach().numpy().reshape(x_grid.shape)
```

[111]:
```
plt.figure(figsize=(8, 6))
plt.pcolormesh(x_grid, y_grid, psi_grid, shading='auto')
plt.colorbar(label='psi_grid')
plt.xlabel('x')
```

```
plt.ylabel('y')
plt.show()
```



[ ]: