PINN_harmonic

September 14, 2023

1 PINN for harmonic oscillator

Credit: tutorial of Ben Moseley.

1.1 Formulation

We are going to use a PINN to solve problems related to the **damped harmonic oscillator**:

We are interested in modelling the displacement of the mass on a spring (green box) over time.

This is a canonical physics problem, where the displacement, $u(t)$, of the oscillator as a function of time can be described by the following differential equation,

$$m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku = 0~,$$

where m is the mass of the oscillator, μ is the coefficient of friction and k is the spring constant.

We will focus on solving the problem in the **under-damped state**, i.e. where the oscillation is slowly damped by friction (as displayed in the animation above).

Mathematically, this occurs when

$$\delta < \omega_0~,~~~~~\mathrm{where}~~\delta = \dfrac{\mu}{2m}~,~\omega_0 = \sqrt{\dfrac{k}{m}}~.$$

Furthermore, we consider the following initial conditions of the system,

$$u(t=0) = 1~~,~~\frac{du}{dt}(t=0) = 0~.$$

For this particular case, the exact solution is known and given by

$$u(t) = e^{-\delta t}(2A\cos(\phi + \omega t))~,~~~~~\mathrm{with}~~\omega=\sqrt{\omega_0^2 - \delta^2}~,$$

with

$$A = \frac{1}{2\cos(\phi)}, \quad \phi = \tan^{-1}(-\delta/\omega).$$

[1]:
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

1.2 Initial setup

Functions for

- ode solution
- neural net

[2]:
```
def exact_solution(d, w0, t):
    "Defines the analytical solution to the under-damped harmonic oscillator␣
    ↪problem above."
    assert d < w0
    w = np.sqrt(w0**2-d**2)
    phi = np.arctan(-d/w)
    A = 1/(2*np.cos(phi))
    cos = torch.cos(phi+w*t)
    exp = torch.exp(-d*t)
    u = exp*2*A*cos
    return u

class FCN(nn.Module):
    "Defines a standard fully-connected network in PyTorch"

    def __init__(self, N_INPUT, N_OUTPUT, N_HIDDEN, N_LAYERS):
        super().__init__()
        activation = nn.Tanh
        self.fcs = nn.Sequential(*[
                        nn.Linear(N_INPUT, N_HIDDEN),
                        activation()])
        self.fch = nn.Sequential(*[
                        nn.Sequential(*[
                            nn.Linear(N_HIDDEN, N_HIDDEN),
                            activation()]) for _ in range(N_LAYERS-1)])
        self.fce = nn.Linear(N_HIDDEN, N_OUTPUT)

    def forward(self, x):
        x = self.fcs(x)
        x = self.fch(x)
        x = self.fce(x)
        return x
```

1.3 Train PINN

Train the PINN to solve the ODE.

Approach The PINN is trained to directly approximate the solution to the differential equation, i.e.

$$u_{\text{PINN}}(t;\theta) \approx u(t)\ ,$$

where θ are the weights of the PINN.

Loss function To simulate the system, the PINN is trained with the following loss function

$$\mathcal{L}(\theta) = \mathcal{L}_0 + \lambda_1 \mathcal{L}_1 + \lambda_2 \mathcal{L}_2,$$

where

$$\mathcal{L}_0 = (u_{\text{PINN}}(t=0;\theta) - 1)^2, \quad \mathcal{L}_1 = \left(\frac{d\,u_{\text{PINN}}}{dt}(t=0;\theta) - 0\right)^2$$

are the **boundary loss** terms that ensure the PINN matches the given initial conditions , and

$$\mathcal{L}_2 = \frac{1}{N}\sum_i^N \left(\left[m\frac{d^2}{dt^2} + \mu\frac{d}{dt} + k \right] u_{\text{PINN}}(t_i;\theta) \right)^2$$

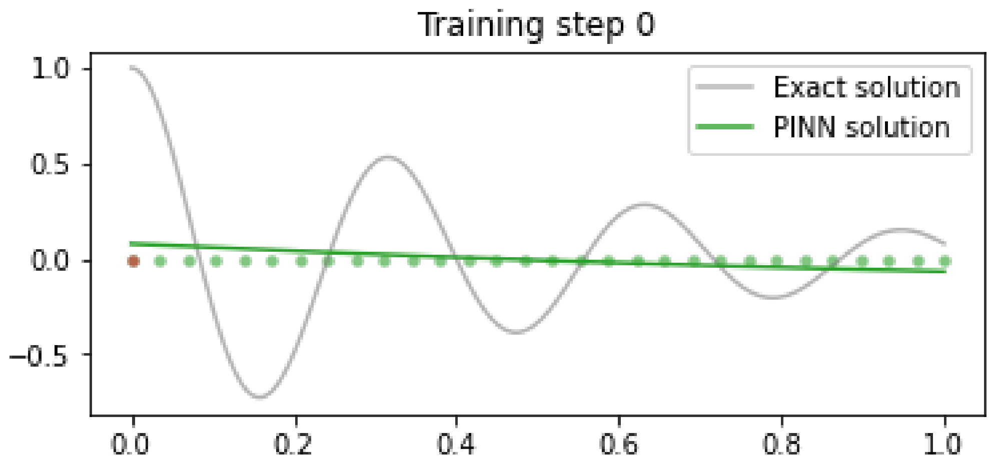
is the **physics loss**, and and tries to ensure that the PINN solution satisfies the underlying differential equation at a set of training points $\{t_i\}$ sampled over the entire domain.

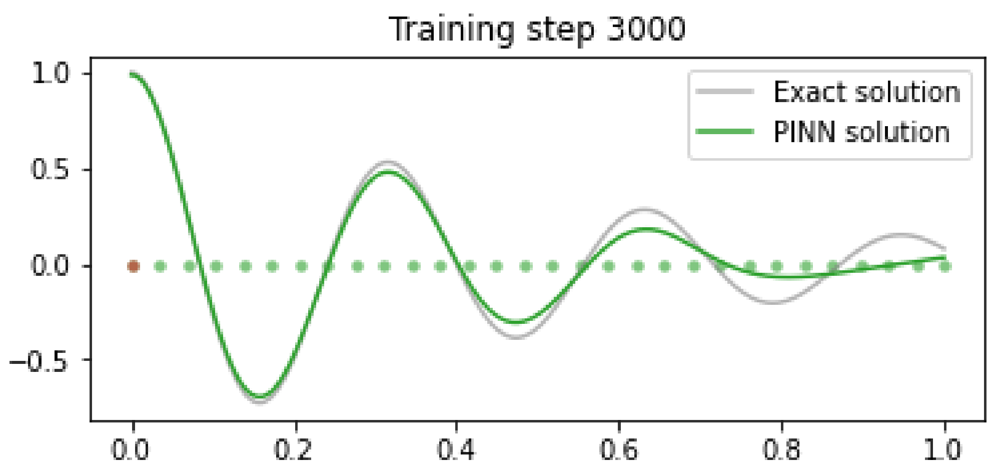
Notes

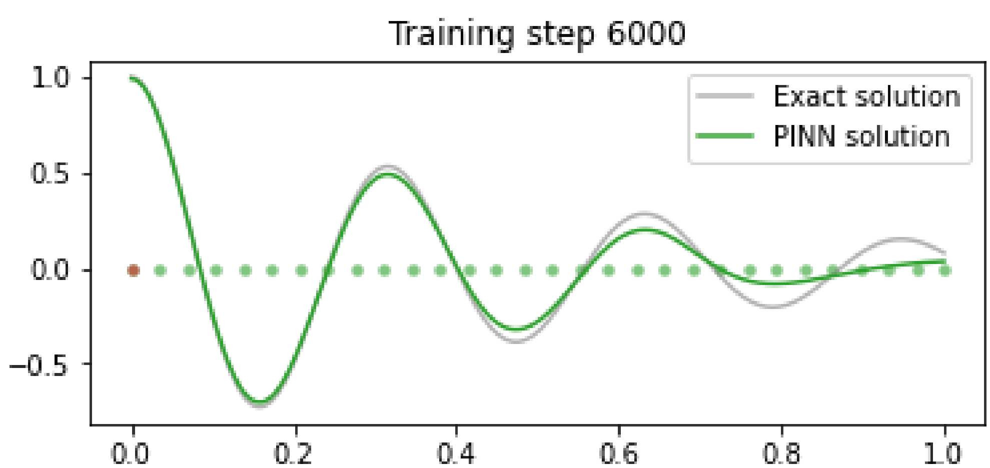
- The hyperparameters, λ_1 and λ_2, are used to balance the terms in the loss function, to ensure stability during training. These need to be found by trial and error.
- We set $\delta = 2$, $\omega_0 = 20$, and try to learn the solution over the domain $t \in [0, 1]$.

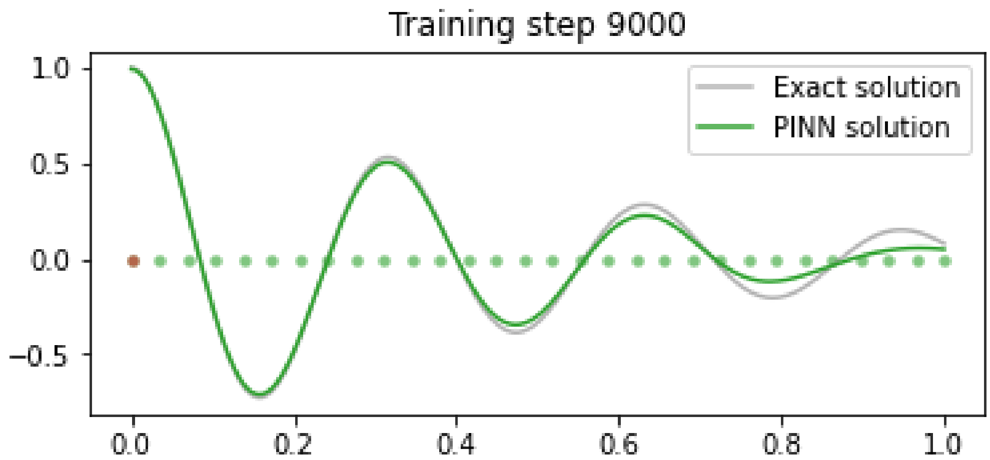
[5]:
```
torch.manual_seed(123)

# define a neural network to train
pinn = FCN(1,1,32,3)

# define boundary points, for the boundary loss
t_boundary = torch.tensor(0.).view(-1,1).requires_grad_(True)

# define training points over the entire domain, for the physics loss
t_physics = torch.linspace(0,1,30).view(-1,1).requires_grad_(True)

# train the PINN
d, w0 = 2, 20
mu, k = 2*d, w0**2
t_test = torch.linspace(0,1,300).view(-1,1)
u_exact = exact_solution(d, w0, t_test)
optimiser = torch.optim.Adam(pinn.parameters(),lr=1e-3)
for i in range(15001):
```

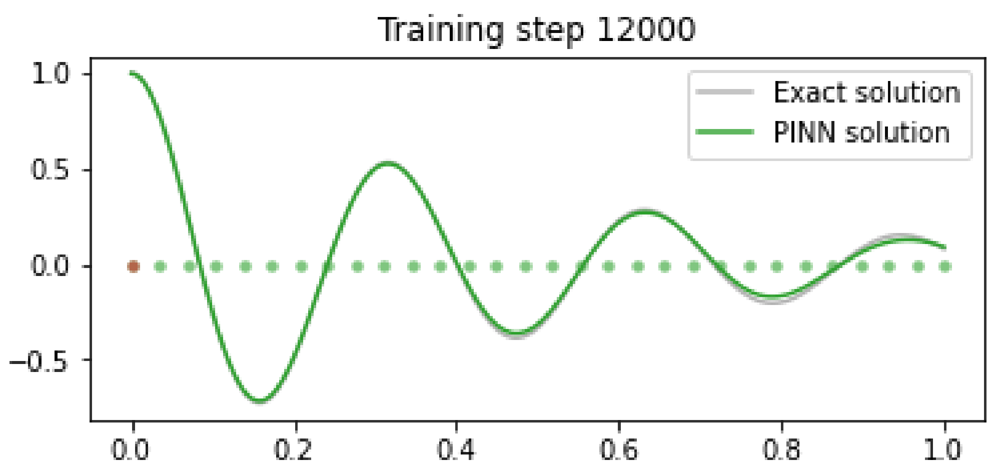
```
    optimiser.zero_grad()

    # compute each term of the PINN loss function above
    # using the following hyperparameters:
    lambda1, lambda2 = 1e-1, 1e-4

    # compute boundary loss terms
    u = pinn(t_boundary)
    loss1 = (torch.squeeze(u) - 1)**2
    dudt = torch.autograd.grad(u, t_boundary, torch.ones_like(u), create_graph=True)[0]
    loss2 = (torch.squeeze(dudt) - 0)**2

    # compute physics loss using autograd
    u = pinn(t_physics)
    dudt = torch.autograd.grad(u, t_physics, torch.ones_like(u), create_graph=True)[0]
    d2udt2 = torch.autograd.grad(dudt, t_physics, torch.ones_like(dudt), create_graph=True)[0]
    loss3 = torch.mean((d2udt2 + mu*dudt + k*u)**2)

    # backpropagate joint loss, take optimiser step
    loss = loss1 + lambda1*loss2 + lambda2*loss3
    loss.backward()
    optimiser.step()

    # plot the result as training progresses
    if i % 3000 == 0:
        #print(u.abs().mean().item(), dudt.abs().mean().item(), d2udt2.abs().mean().item())
        u = pinn(t_test).detach()
        plt.figure(figsize=(6,2.5))
        plt.scatter(t_physics.detach()[:,0],
                    torch.zeros_like(t_physics)[:,0], s=20, lw=0, color="tab:green", alpha=0.6)
        plt.scatter(t_boundary.detach()[:,0],
                    torch.zeros_like(t_boundary)[:,0], s=20, lw=0, color="tab:red", alpha=0.6)
        plt.plot(t_test[:,0], u_exact[:,0], label="Exact solution", color="tab:grey", alpha=0.6)
        plt.plot(t_test[:,0], u[:,0], label="PINN solution", color="tab:green")
        plt.title(f"Training step {i}")
        plt.legend()
        plt.show()
```

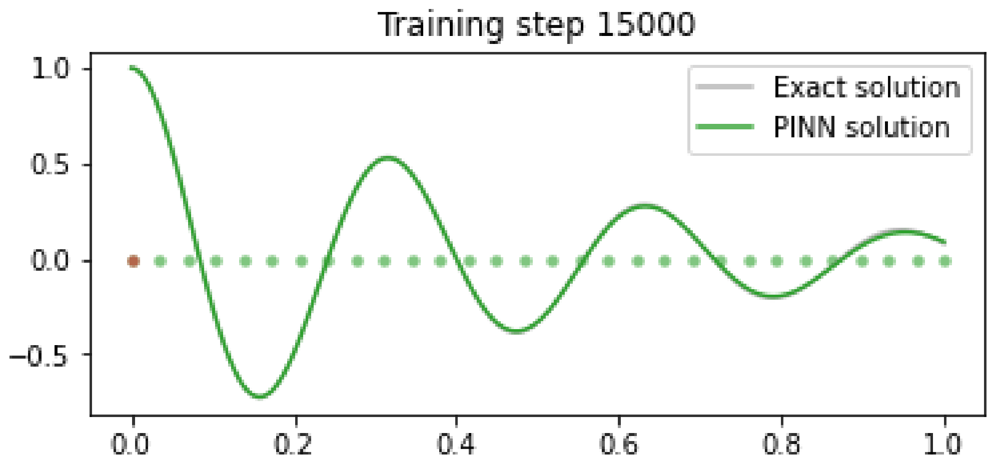

Training step 0
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 3000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 6000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 9000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0


Training step 12000
Exact solution
PINN solution
1.0
0.5
0.0
−0.5
0.0
0.2
0.4
0.6
0.8
1.0



1.4 Conclusion

Starting from a very rough initial solution, the PINN model has effectively solved the direct problem, albeit with a relatively large number of iterations.

1.5 Inverse Problem

Now, we want to find an unknown μ from noisy observations of the oscillators position over time.

The key idea here is to also treat μ as a *learnable parameter* when training the PINN - so that we both simulate the solution and invert for this parameter.

Loss function The PINN is trained with a slightly different loss function,

$$\mathcal{L}(\theta, \mu) = \mathcal{L}_0 + \lambda_1 \mathcal{L}_1$$

where

$$\mathcal{L}_0 = \frac{1}{N} \sum_{i}^{N} \left(\left[m \frac{d^2}{dt^2} + \mu \frac{d}{dt} + k \right] u_{\text{PINN}}(t_i; \theta) \right)^2$$

is the *physics loss*, and

$$\mathcal{L}_1 = \frac{1}{M} \sum_{j}^{M} \left(u_{\text{PINN}}(t_j; \theta) - u_{\text{obs}}(t_j) \right)^2$$

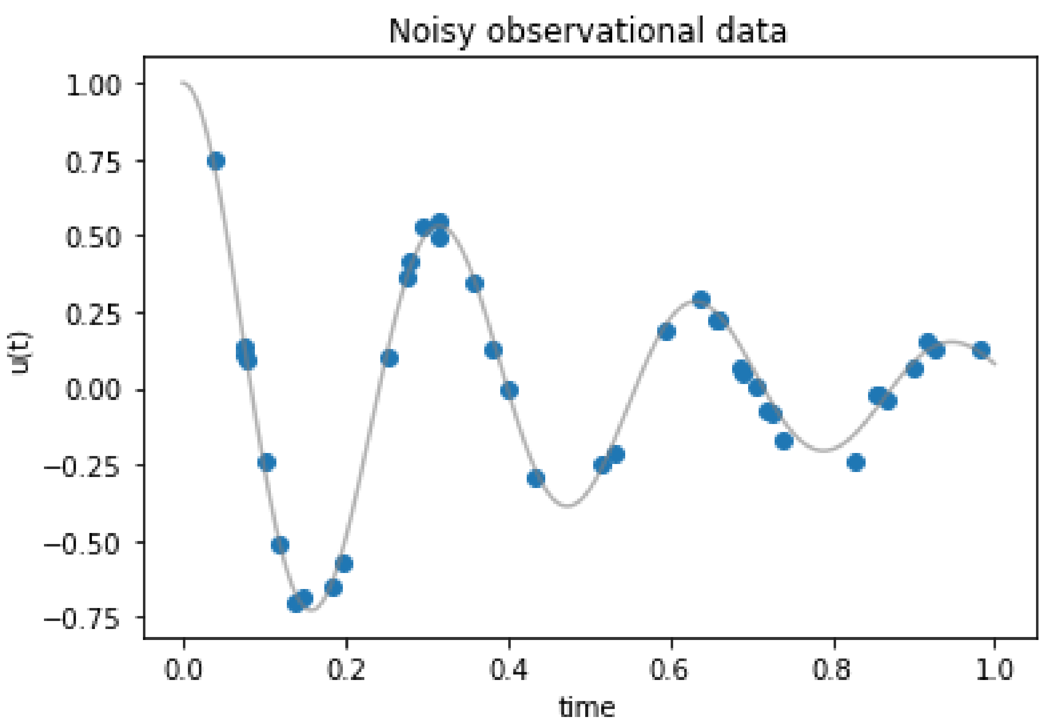
is the *data loss*.

Notes

- we have removed the boundary loss terms, as we do not know these (i.e., we are only given the observed measurements of the system).
- the PINN parameters θ and μ are *jointly* learned during optimisation.

[7]:
```
# first, create some noisy observational data
torch.manual_seed(123)
d, w0 = 2, 20
print(f"True value of mu: {2*d}")
t_obs = torch.rand(40).view(-1,1)
u_obs = exact_solution(d, w0, t_obs) + 0.04*torch.randn_like(t_obs)

plt.figure()
plt.title("Noisy observational data")
plt.scatter(t_obs[:,0], u_obs[:,0])
t_test, u_exact = torch.linspace(0,1,300).view(-1,1), exact_solution(d, w0,␣
↪t_test)
plt.plot(t_test[:,0], u_exact[:,0], label="Exact solution", color="tab:grey",␣
↪alpha=0.6)
plt.xlabel("time")
plt.ylabel("u(t)")
plt.show()
```

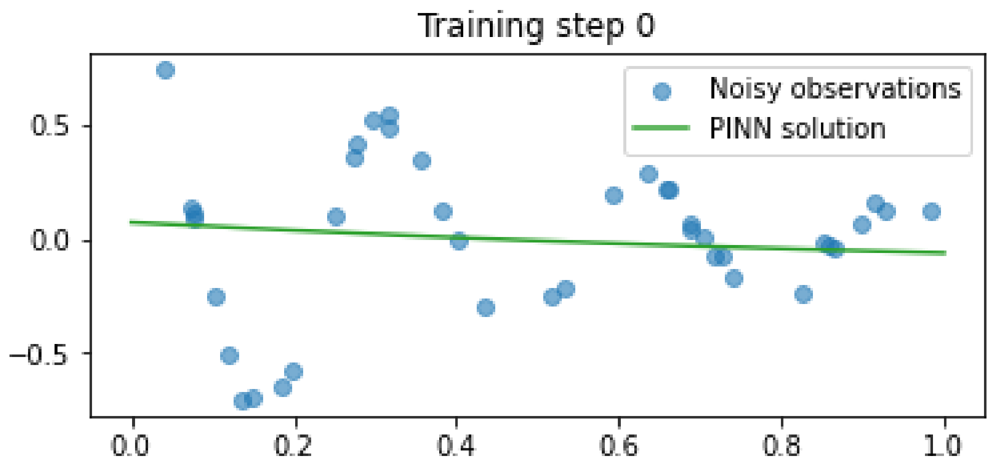
```
True value of mu: 4
```

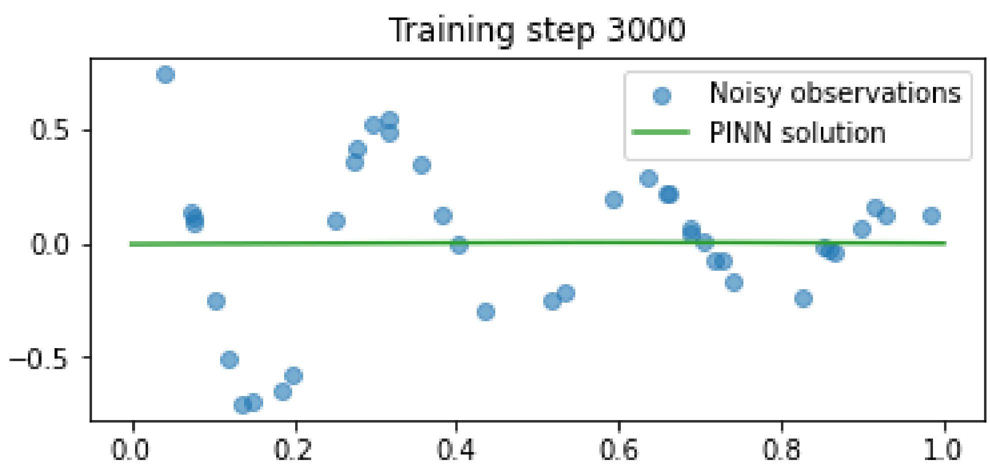


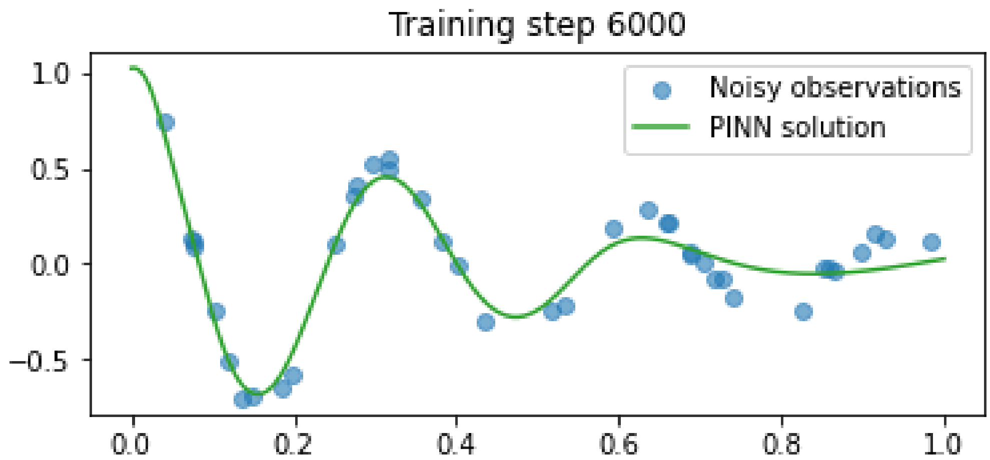
[10]:
```
torch.manual_seed(123)

# define a neural network to train
```

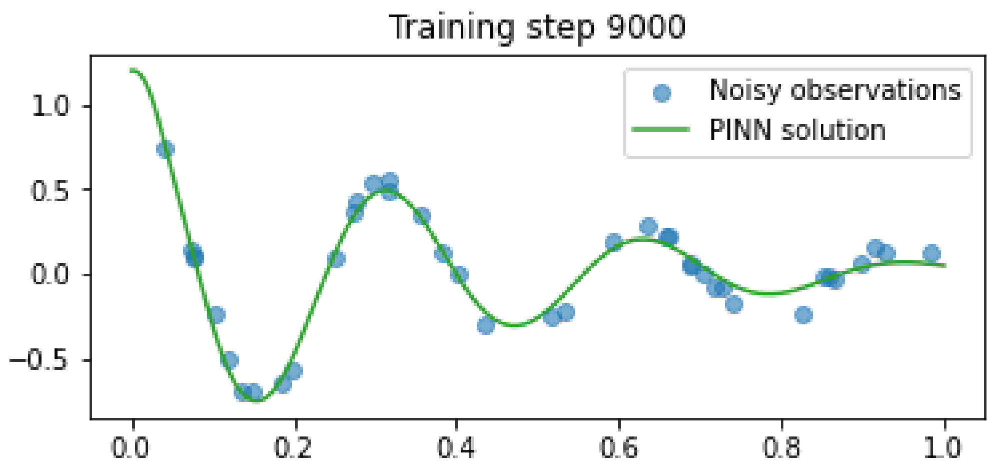
```
pinn = FCN(1,1,32,3) # originally 3 layers

# define training points over the entire domain, for the physics loss
t_physics = torch.linspace(0,1,30).view(-1,1).requires_grad_(True)

# train the PINN
d, w0 = 2, 20
_, k = 2*d, w0**2

# treat mu as a learnable parameter
mu = torch.nn.Parameter(torch.zeros(1, requires_grad=True))
mus = []

# add mu to the optimiser
optimiser = torch.optim.Adam(list(pinn.parameters())+[mu],lr=1e-3)
for i in range(15001):
    optimiser.zero_grad()

    # compute each term of the PINN loss function above
    # using the following hyperparameters:
    lambda1 = 1e4

    # compute physics loss
    u = pinn(t_physics)
    dudt = torch.autograd.grad(u, t_physics, torch.ones_like(u),␣
 ↪create_graph=True)[0]
    d2udt2 = torch.autograd.grad(dudt, t_physics, torch.ones_like(dudt),␣
 ↪create_graph=True)[0]
    loss1 = torch.mean((d2udt2 + mu*dudt + k*u)**2)

    # compute data loss
    u = pinn(t_obs)
    loss2 = torch.mean((u - u_obs)**2)

    # backpropagate joint loss, take optimiser step
    loss = loss1 + lambda1*loss2
    loss.backward()
    optimiser.step()

    # record mu value
    mus.append(mu.item())

    # plot the result as training progresses
    if i % 3000 == 0:
        u = pinn(t_test).detach()
        plt.figure(figsize=(6,2.5))
```

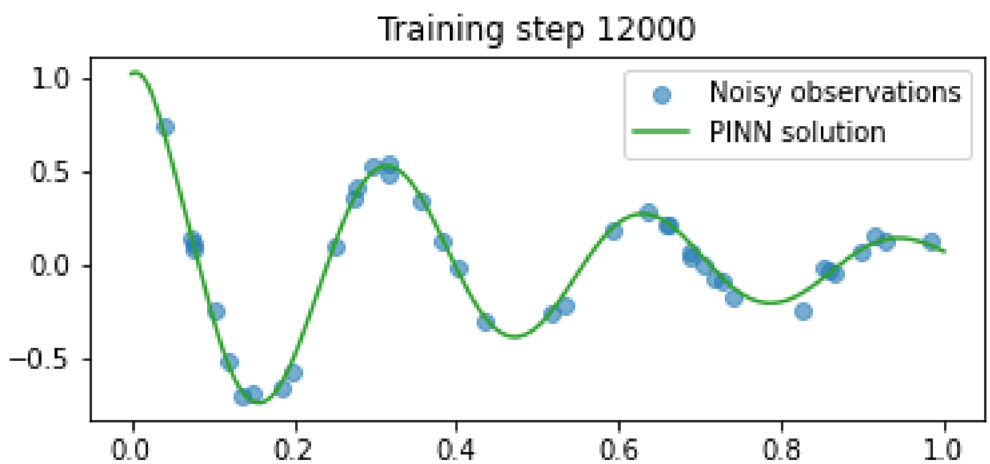
```
        plt.scatter(t_obs[:,0], u_obs[:,0], label="Noisy observations", alpha=0.
↪6)
        plt.plot(t_test[:,0], u[:,0], label="PINN solution", color="tab:green")
        plt.title(f"Training step {i}")
        plt.legend()
        plt.show()

plt.figure()
plt.title("Convergence of $\mu$")
plt.plot(mus, label="PINN estimate")
plt.hlines(2*d, 0, len(mus), label="True value", color="tab:green")
plt.legend()
plt.xlabel("Training step")
plt.show()
```

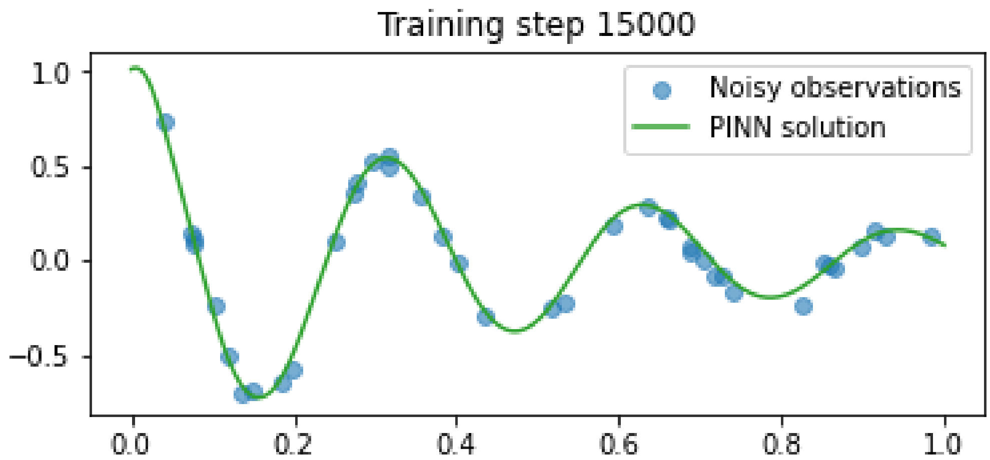


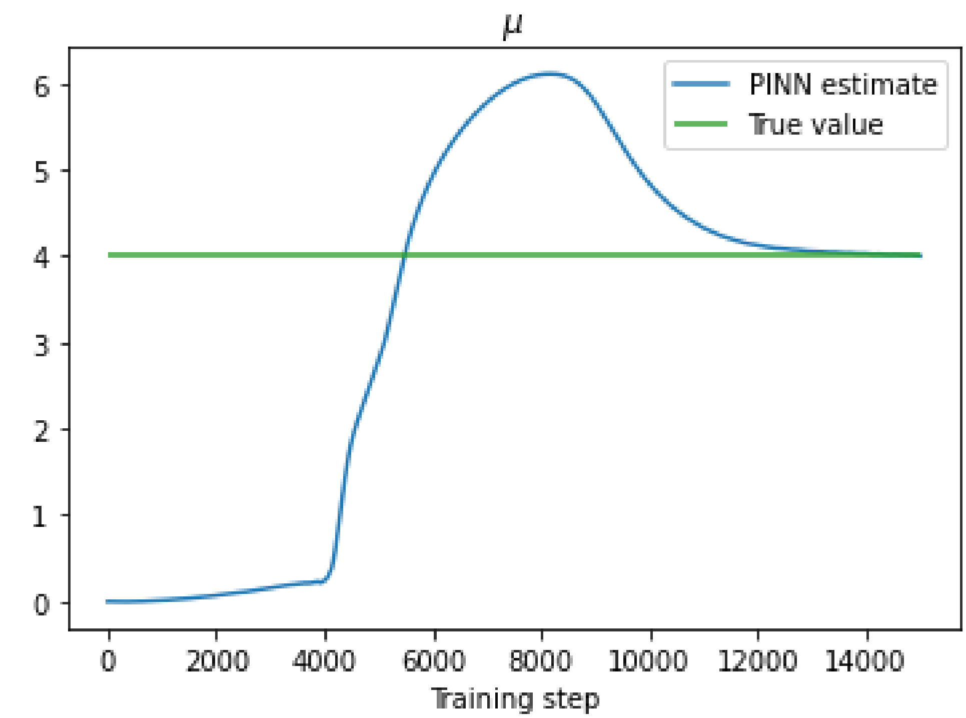



Training step 6000
Noisy observations
PINN solution


Training step 9000
Noisy observations
PINN solution


Training step 12000
Noisy observations
PINN solution





[15]:
```
print("error = ", mus[15000]-4)
```

```
error =  0.0028738975524902344
```

1.6 Conclusions

We have succeeded in solving the inverse problem, using PINN, for the identification of an unknown damping coefficient, μ, in a linear, harmonic oscillator from noisy observations.

```
[ ]:
```