Neural State-Space models

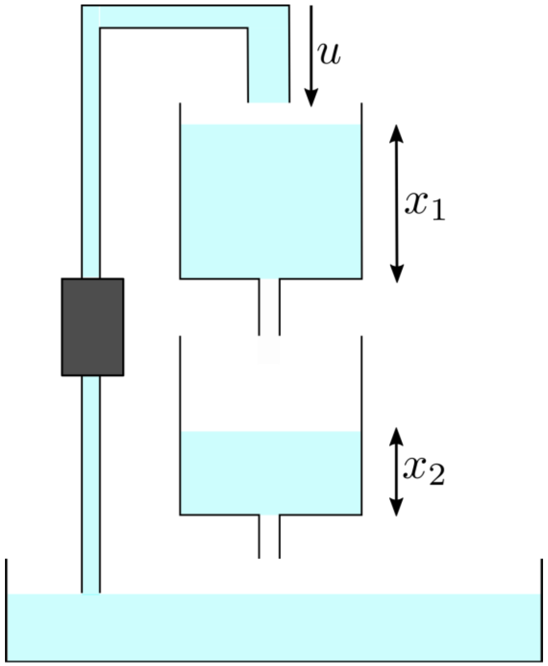
Exercises

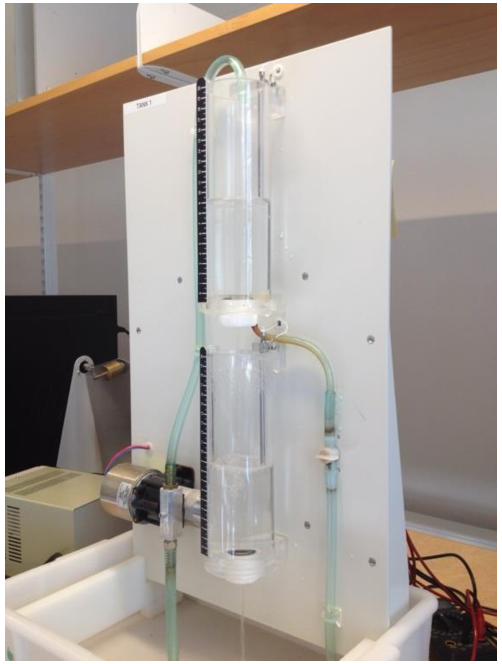
Marco Forgione, Dalle Molle Institute for Artificial Intelligence, Lugano, Switzerland

Lugano, April, 23rd, 2024

Physics-inspired neural state-space models

Cascaded tanks system. Input: upper tank inlet flow u. Output: lower tank level.





Schoukens, M. et al. "Cascaded tanks benchmark combining soft and hard nonlinearities." Workshop on nonlinear system identification benchmarks, 2016.

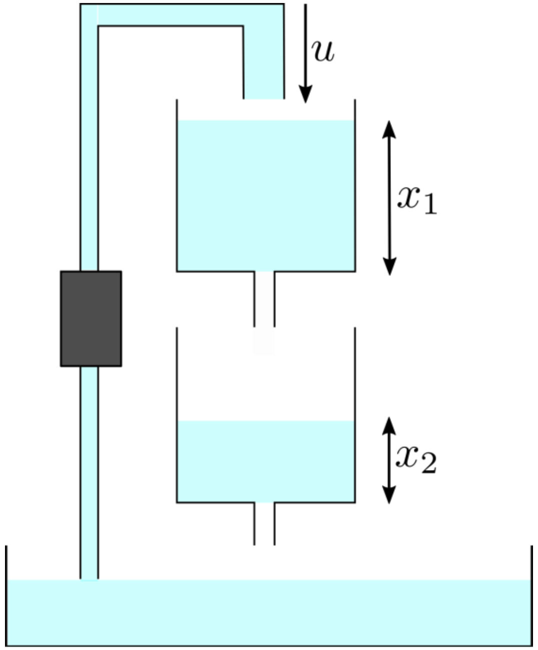
The dataset has just T=1024 time points. We implement **full simulation error minimization**.

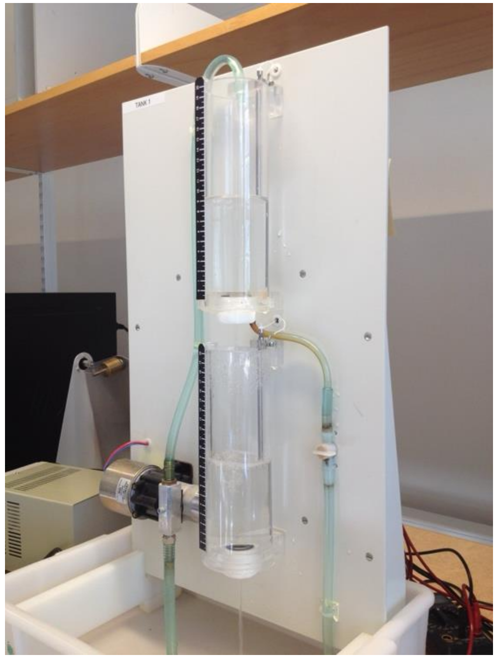
Physics-inspired neural state-space models – model #1

Cascaded tanks system. Input: upper tank inlet flow u. Output: lower tank level.

Intuitive physics:

- **The system has two states: levels x_1, x_2**
- The state x_2 is measured: $y = x_2$
- State x_1 does not depend on x_2
- State x_2 does not depend on u directly



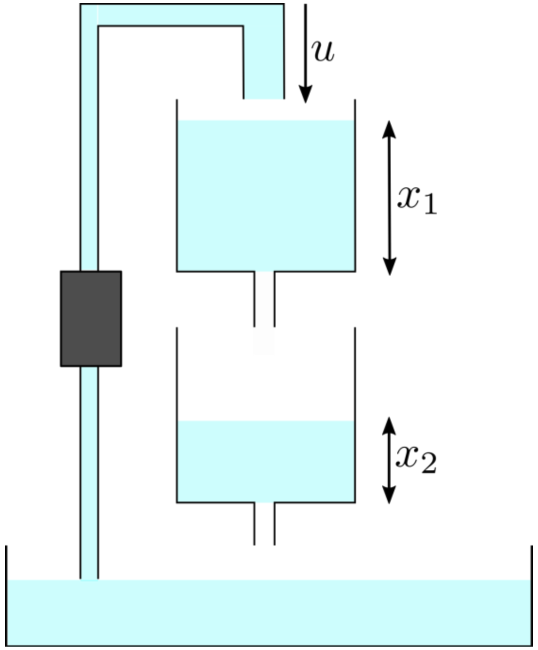


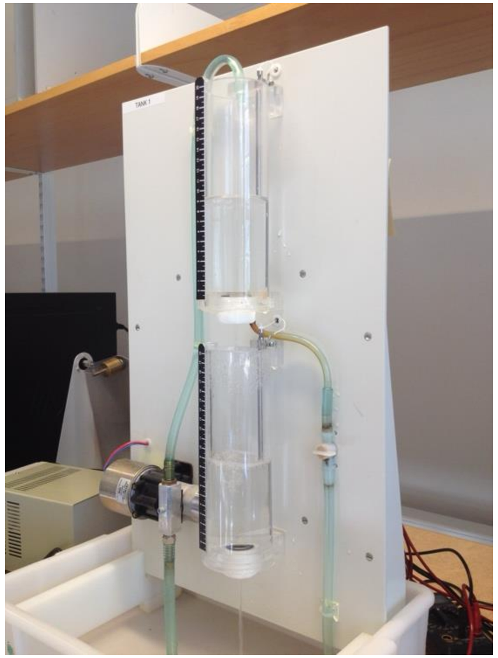
Physics-inspired neural state-space models – model #2

Cascaded tanks system. Input: upper tank inlet flow u. Output: lower tank level.

Intuitive physics:

- **The system has two states: levels x_1, x_2**
- **The state x_2 is measured: $y = x_2$**
- State x_1 does not depend on x_2
- State x_2 does not depend on u directly



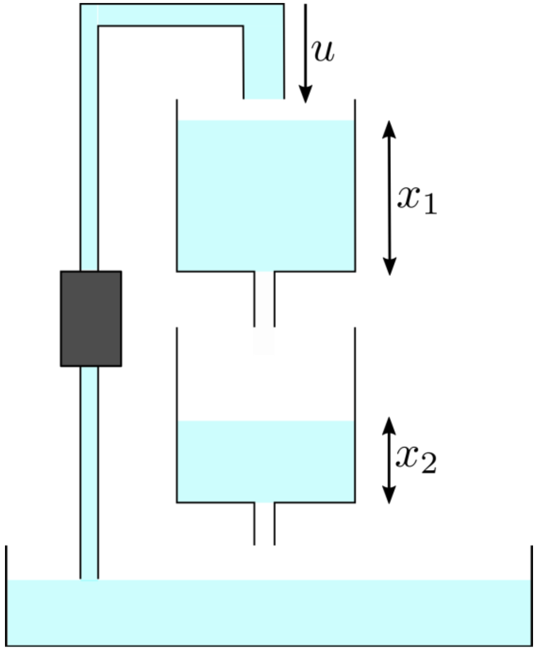


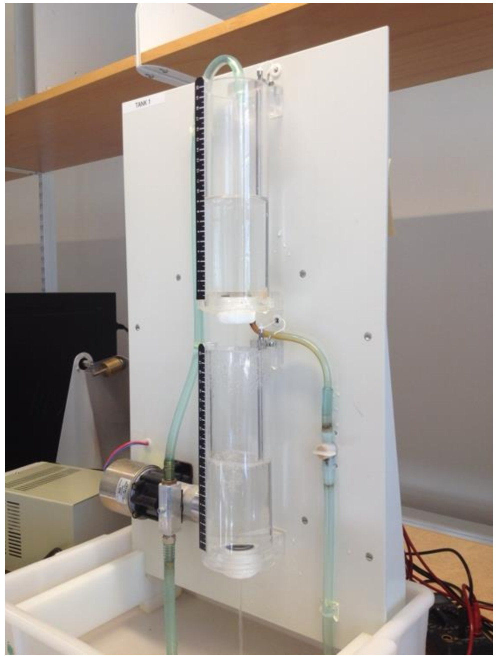
Physics-inspired neural state-space models – model #3

Cascaded tanks system. Input: upper tank inlet flow u. Output: lower tank level.

Intuitive physics:

- **The system has two states: levels x_1, x_2**
- **The state x_2 is measured: $y = x_2$**
- **State x_1 does not depend on x_2**
- ~~State x_2 does not depend on u directly~~
  (according to the benchmark description, water overflows from the upper to the lower tank. Thus, there is a direct term from u to x_2)





Simulation error minimization – model #1

Let us implement **simulation error minimization**

$$\hat{\theta}, \hat{x}(0) = \arg\min_{\theta, x(0)} \frac{1}{T} \sum_{k=0}^{T-1} \left\| \hat{y}^{\text{sim}}(k) - y(k) \right\|^2$$

$$x(k+1) = x(k) + f(x(k), u(k);\ \theta)$$
$$\hat{y}^{\text{sim}}(k) = g(x(k);\ \theta)$$

Where f and g are generic feed-forward neural networks with:

- $n_u = 1, n_y = 1$ (not a choice, dimensionality constraint)
- $n_x = 2$ (as known from the physics)
- One hidden layer
- Tanh non-linearity
- 64 hidden units

NOTE:

- You should train with respect to both θ and $x(0)$.
- In test, exploit the additional knowledge from the benchmark info that $x(0)$ **is the same in the two experiments**.

PyTorch Implementation (hints)

Custom state-update function $f(x, u)$

```python
class NeuralStateUpdate(nn.Module):

    def __init__(self, n_x=2, n_u=1, n_feat=32):
        super(NeuralStateUpdate, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(n_x+n_u, n_feat),
            nn.Tanh(),
            nn.Linear(n_feat, n_x),
        )

        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=1e-2)
                nn.init.constant_(m.bias, val=0)

    def forward(self, x, u):
        z = torch.cat((x, u), dim=-1)
        dx = self.net(z)
        return dx
```

Custom code to unroll $f(x, u)$ over time

```python
class StateSpaceSimulator(nn.Module):
    def __init__(self, f_xu):
        super().__init__()
        self.f_xu = f_xu

    def forward(self, x_0, u):
        B, n_x = x_0.shape
        _, T, _ = u.shape # B, T, n_u
        x = torch.empty((B, T, n_x))
        x_step = x_0

        # manually unroll f_xu over time
        for t in range(T):
            x[:, t, :] = x_step
            dx = self.f_xu(x_step, u[:, t, :])
            x_step = x_step + dx

        return x
```

Code adapted from https://github.com/forgi86/pytorch-ident

PyTorch Implementation (hints)

Initializations

```
n_y = 1; n_x = 2; n_u = 1;
B = 1
T = 1024
u = torch.randn((B, T, n_u)) # replace with actual training input
y = torch.randn((B, T, n_y)) # replace with actual training output
```

Define model and initial state

```
x0 = torch.zeros((B, n_x), requires_grad=True) # this is also a training variable
f_xu = NeuralStateUpdate(n_x, n_u, n_feat=32)
simulator = StateSpaceSimulator(f_xu) #
g_x = NeuralOutput(n_x, n_y, n_feat=32) # an MLP with n_x input and n_y outputs
```

Define optimizer

```
opt = torch.optim.AdamW([
        {'params': f_xu.parameters(),    'lr': 1e-3},
        {'params': g_x.parameters(),    'lr': 1e-3},
        {'params': x0 , 'lr': 1e-3},
    ], 1e-3
)
#opt = torch.optim.AdamW(list(f_xu.parameters()) + list(g_x.parameters()) + [x0], 1e-3)
```

Compute loss (in a trainig loop…)

```
x_sim = simulator(x0, u) # B, T, n_x
y_sim = g_x(x_sim) # # B, T, n_y
loss = torch.nn.functional.mse_loss(y, y_sim)
```

Simulation error minimization – model #2

Let us implement **simulation error minimization**

$$\hat{\theta}, \hat{x}(0) = \arg\min_{\theta, x(0)} \frac{1}{T} \sum_{k=0}^{T-1} \left\| \hat{y}^{\text{sim}}(k) - y(k) \right\|^2$$

Where f is a generic feed-forward neural networks and the second state is observed:

$$\dot{x} = f(x, u;\ \theta)$$
$$y = x_2$$

Simulation error minimization – model #3

Let us implement **simulation error minimization**

$$\hat{\theta}, \hat{x}(0) = \arg\min_{\theta, x(0)} \frac{1}{T} \sum_{k=0}^{T-1} \left\| y^{\text{sim}}(k) - y(k) \right\|^2$$

Implement and train a tailor-made architecture consistent with all the physics constraints assumed for model #3:

$$\begin{aligned}
\dot{x}_1 &= f_1(x_1, u; \theta) \\
\dot{x}_2 &= f_2(x_1, x_2, u; \theta) \\
y &= x_2
\end{aligned}$$

Thank you for your attention