Robot Arm

Joshoua Bigler

January 8, 2025

Abstract

The goal of this project is to develop a comprehensive simulation of a robotic arm using a physics based machine learning (PBML) approach. The simulation aims to effectively model the physical behavior of the robotic arm, incorporating a variety of disturbing factors to create a realistic environment for evaluating control algorithms and performance. To accomplish this, the analytical robotic arm is modeled using a Lagrangian formulation. Trajectories derived from this formulation are then used to train an ordinary differential equation network (ODE-Net) to replicate the dynamics of the robotic arm.

Declaration of Independence

"I hereby certify that I have written this thesis independently and have not used any auxiliary materials other than those indicated. The passages of the work, which are taken in the wording or the sense after other works (to it also Internet sources count), were marked under indication of the source".

Joshoua Bigler	January 8, 2025
	Date

Contents

1 Task **3**

2 The Lagragian System **4**

3 Euler-Lagrange Equation **5**
3.1 First-Order Transformation 5

4 Numeric Robot Arm **7**

5 ODE Network **9**
5.1 Mathematical Formulation 9
5.2 Network Architecture 9
5.3 Training the ODE Network 9
5.4 ODE-Net Architectures 11
5.5 Training and Evaluation Losses 12
5.6 Trajectories 13
5.7 Energy Conservation 13

6 Conclusion and Outlook **14**
6.1 Conclusion 14
6.2 Outlook 14

Glossary **16**

A Euler Lagrange Equation **18**
A.1 Euler Lagrange Equations to Theta 1 18
A.2 Euler Lagrange Equations to Theta 2 18
A.3 Second Order Derivative 19

B ODE Network **20**
B.1 ODE Net 2 20
B.2 ODE Net 3 20

C Robot Arm **21**
C.1 Pertubation 21

1 Task

The goal of this project is to model the equations of motion for a two-link robotic arm using a Lagrangian formulation. Additionally, an ODE network is trained to approximate the equations of motion, whereby disturbances from outside are taken into account. To model the robotic arm (Figure 1), the following assumptions are made:

- 2 degrees of freedom with joint angles: θ_1, θ_2
- Massless link lengths: l_1, l_2
- Point masses: m_1, m_2
- Gravitational acceleration: g

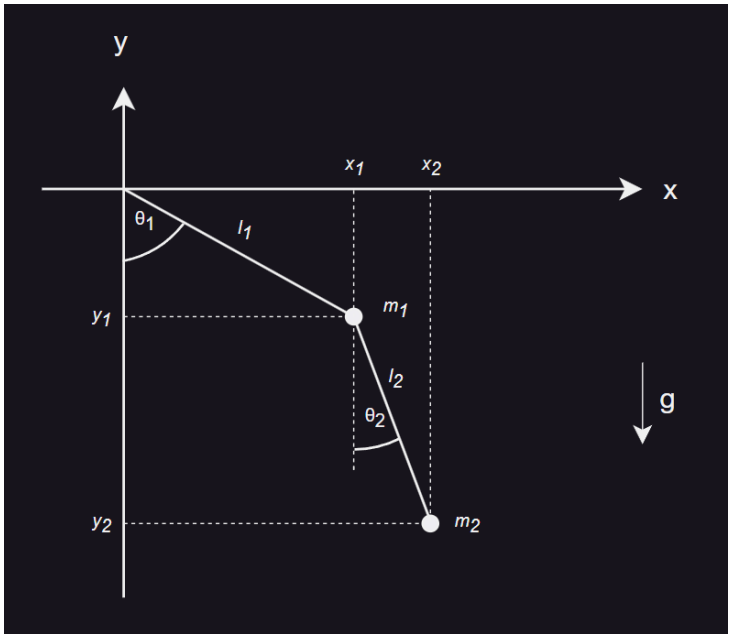

Figure 1: Robot arm sketch

Coordiantes

$$x_1 = l_1 \sin\theta_1 \tag{1}$$
$$x_2 = l_1 \sin\theta_1 + l_2 \sin\theta_2 \tag{2}$$
$$y_1 = -l_1 \cos\theta_1 \tag{3}$$
$$y_2 = -l_1 \cos\theta_1 - l_2 \cos\theta_2 \tag{4}$$

$$\dot{x}_1 = l_1\dot{\theta}_1 \cos\theta_1 \tag{5}$$
$$\dot{x}_2 = l_1\dot{\theta}_1 \cos\theta_1 + l_2\dot{\theta}_2 \cos\theta_2 \tag{6}$$
$$\dot{y}_1 = l_1\dot{\theta}_1 \sin\theta_1 \tag{7}$$
$$\dot{y}_2 = l_1\dot{\theta}_1 \sin\theta_1 + l_2\dot{\theta}_2 \sin\theta_2 \tag{8}$$

2 The Lagragian System

To define the Lagrangian system, the work of Würsch (2024a) was taken as a reference. The Lagrangian (L) of the system is defined as the difference between the kinetic energy (T) and the potential energy (V):

$$L = T - V$$

Potential Energy (V)
The total potential energy V is:

$$V = \sum_{i=1}^{2} m_i g y_i$$

$$V = m_1 g y_1 + m_2 g y_2$$

$$V = g\left(l_1 m_1 \cos\left(\theta_1(t)\right) + l_1 m_2 \cos\left(\theta_1(t)\right) + l_2 m_2 \cos\left(\theta_2(t)\right)\right)$$

Kinetic Energy (T)
The total kinetic energy T is:

$$T = \sum_{i=1}^{2} \frac{1}{2} m_i v_i^2 = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 (\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2} m_2 (\dot{x}_2^2 + \dot{y}_2^2)$$

$$T = 0.5 l_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 0.5 m_2 \left(l_1^2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 2 l_1 l_2 \cos\left(\theta_1(t) - \theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + l_2^2 \left(\frac{d}{dt}\theta_2(t)\right)^2\right)$$

Lagrange Equation (L)

$$L = T - V = T_1 + T_2 - (V_1 + V_2)$$

$$\begin{aligned} L = {} & g\Big(l_1 m_1 \cos\left(\theta_1(t)\right) + l_1 m_2 \cos\left(\theta_1(t)\right) + l_2 m_2 \cos\left(\theta_2(t)\right)\Big) + 0.5 l_1^2 m_1 \left(\frac{d}{dt}\theta_1(t)\right)^2 \\ & + 0.5 m_2 \Big(l_1^2 \left(\frac{d}{dt}\theta_1(t)\right)^2 + 2 l_1 l_2 \cos\left(\theta_1(t) - \theta_2(t)\right) \frac{d}{dt}\theta_1(t) \frac{d}{dt}\theta_2(t) + l_2^2 \left(\frac{d}{dt}\theta_2(t)\right)^2\Big) \end{aligned}$$

3 Euler-Lagrange Equation

The **Euler-Lagrange equation** is used to derive the equations of motion (Appendix A.1 and Appendix A.2):

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_i}\right) - \frac{\partial L}{\partial \theta_i} = Q_i = -\frac{\partial \mathcal{R}}{\partial \dot{\theta}_i} = -\mu_i \dot{\theta}_i, \quad i = 1, 2$$

with the dissipation function $\mathcal{R}_i$ defined as:

$$\mathcal{R}_i = \frac{1}{2}\mu_i \dot{\theta}_i^2$$

Here, μ_i ($\mathrm{kg} \cdot \mathrm{m}^2/\mathrm{s}$) denotes the damping coefficient associated with the i-th degree of freedom.

The euler lagrange equation of motion can be written as the following system of differential equations of second order. For more details, refer to Appendix A.3.

$$\frac{d^2}{dt^2}\theta_1(t) = ...$$
$$\frac{d^2}{dt^2}\theta_2(t) = ...$$

3.1 First-Order Transformation

To solve the differential equation, the system will be transformed into a first order system.

$$\ddot{\theta}_1(t) = f_1(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2),$$
$$\ddot{\theta}_2(t) = f_2(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2).$$

$$z_1 = \theta_1$$
$$z_2 = \theta_2$$
$$z_3 = \dot{z}_1$$
$$z_4 = \dot{z}_2$$

$$\vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} \theta_1 \\ \theta_2 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{pmatrix}, \quad \frac{d}{dt}\vec{z}(t) = \vec{f}\left(\vec{z}, t\right).$$

Now the system can be written as a first order system of differential equations:

$$\ddot{\theta}_1(t) = \frac{d}{dt}\vec{z}(t) = \frac{d}{dt}\left(\frac{-gl_1l_2m_1\sin\left(z_1\right) - \frac{gl_1l_2m_2\sin\left(z_1\right)}{2} - \frac{gl_1l_2m_2\sin\left(z_1-2z_2\right)}{2}}{l_1^2l_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}\right.$$
$$- \frac{\frac{l_1^2l_2m_2z_3^2\sin\left(2z_1-2z_2\right)}{2} + l_1l_2^2m_2z_4^2\sin\left(z_1 - z_2\right)}{l_1^2l_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}$$
$$\left. - \frac{l_1\mu_2z_4\cos\left(z_1 - z_2\right) - l_2\mu_1z_3}{l_1^2l_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}\right)$$

$$\ddot{\theta}_2(t) = \frac{d}{dt}\vec{z}(t) = \frac{d}{dt}\left(\frac{-\frac{gl_1l_2m_1m_2\sin\left(z_2\right)}{2} + \frac{gl_1l_2m_1m_2\sin\left(2z_1-z_2\right)}{2} - \frac{gl_1l_2m_2^2\sin\left(z_2\right)}{2} + \frac{gl_1l_2m_2^2\sin\left(2z_1-z_2\right)}{2}}{l_1l_2^2m_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}\right.$$
$$+ \frac{l_1^2l_2m_1m_2z_3^2\sin\left(z_1 - z_2\right) + l_1^2l_2m_2^2z_3^2\sin\left(z_1 - z_2\right) + \frac{l_1l_2^2m_2^2z_4^2\sin\left(2z_1-2z_2\right)}{2}}{l_1l_2^2m_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}$$
$$\left. + \frac{l_1m_1\mu_2z_4 + l_1m_2\mu_2z_4 - l_2m_2\mu_1z_3\cos\left(z_1 - z_2\right)}{l_1l_2^2m_2\left(m_1 - m_2\cos^2\left(z_1 - z_2\right) + m_2\right)}\right)$$

The first-order system can be solved numerically using integrators such as the Euler method, Runge-Kutta (RK) methods or adaptive techniques like `odeint` (*SciPy* 2024).

4 Numeric Robot Arm

To obtain the numerical representation of the robot arm, it's parameters need to be initialized. These parameters are defined as follows:

$$
\begin{aligned}
l_1 &= 1.0\,\mathrm{m} && \text{(Length of arm 1)} \\
l_2 &= 1.0\,\mathrm{m} && \text{(Length of arm 2)} \\
m_1 &= 1.0\,\mathrm{kg} && \text{(Mass of arm 1)} \\
m_2 &= 1.0\,\mathrm{kg} && \text{(Mass of arm 2)} \\
g &= 9.81\,\mathrm{m/s^2} && \text{(Acceleration due to gravity)} \\
\mu_1 &= 1\,\mathrm{kg \cdot m^2/s} && \text{(Damping coefficient 1)} \\
\mu_2 &= 1\,\mathrm{kg \cdot m^2/s} && \text{(Damping coefficient 2)}
\end{aligned}
$$

Like in Listing 1, the `solve_odeint` method takes the initial state

$$
z_0 = \begin{bmatrix} \theta_1(0) \\ \theta_2(0) \\ \omega_1(0) \\ \omega_2(0) \end{bmatrix}
$$

the equations of motion

$$
\begin{aligned}
&\texttt{eq\_theta1}(z_1, z_2, z_3, z_4) \\
&\texttt{eq\_theta2}(z_1, z_2, z_3, z_4)
\end{aligned}
$$

the time points t, and the perturbation variance if defined as input.

$$
\mathbf{z}(t) = \begin{bmatrix} \theta_1(t) \\ \theta_2(t) \\ \omega_1(t) \\ \omega_2(t) \end{bmatrix}
$$

The method returned trajectories defines the system at each time point. In order to simulate a pertubation, the pertubation variance can be set.

```
def solve_odeint(self,
                 z0: np.array,
                 eq_theta1: Function,
                 eq_theta2: Function,
                 time_points: np.array,
                 pertubation_variance: float = None) -> np.array:
  z1, z2, z3, z4 = symbols('z1 z2 z3 z4')
  func_theta1 = lambdify([z1, z2, z3, z4], eq_theta1, modules='numpy')
  func_theta2 = lambdify([z1, z2, z3, z4], eq_theta2, modules='numpy')

  def derivatives(state, t):
    z1, z2, z3, z4 = state
    dz1 = z3  # z1' = z3
    dz2 = z4  # z2' = z4
    dz3 = func_theta1(z1, z2, z3, z4)  # z3' from eq_theta1
    dz4 = func_theta2(z1, z2, z3, z4)  # z4' from eq_theta2
    return [dz1, dz2, dz3, dz4]

  trajectory = integrate.odeint(derivatives, z0, time_points)
  if pertubation_variance:
    noise = np.random.normal(0, pertubation_variance, trajectory.shape)
    trajectory += noise
  return trajectory
```

Listing 1: ODE solver using `odeint`

The robot arm trajectory over a timespan of 5 seconds, sampled at 30 time points with $\Delta t = \frac{5}{30}$ seconds, is depicted in Figure 2.

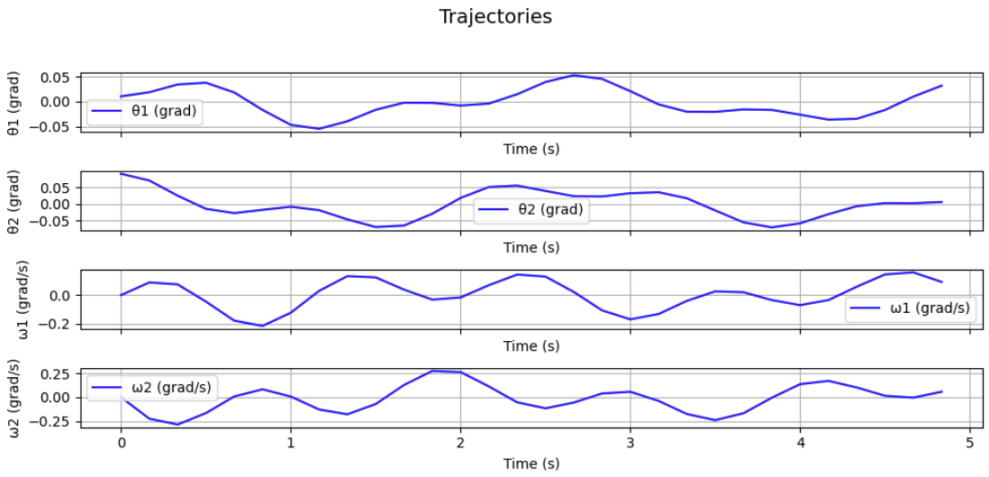

Figure 2: Analytical trajectory of the robot arm

The initial state of the robot arm without pertubation was defined as:

$$z_0 = \begin{bmatrix} \theta_1(0) = 0.01\,\text{grad} \\ \theta_2(0) = 0.09\,\text{grad} \\ \omega_1(0) = 0.0\,\text{grad/s} \\ \omega_2(0) = 0.0\,\text{grad/s} \end{bmatrix}$$

As one can see, the angles and the angular velocities of the robot arm are oscillating around the equilibrium position.

5 ODE Network

In order to model the dynamics of the robot arm, a ODE network is used. As described in Chen et al. (2019, p. 1), 'Instead of specifying a discrete sequence of hidden layers, we parameterize the derivative of the hidden state using a neural network'. This approach allows the model to learn the system's dynamics by approximating the derivative of the hidden state using a neural network.

5.1 Mathematical Formulation

The dynamics of the robot arm can be represented as a system of first-order ordinary differential equations describing the evolution of the system's state over time. Let the state of the robot arm at time t be denoted as $\mathbf{h}(t)$, which includes variables angles and angle velocities. The evolution of this state can be expressed as:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta)$$

where:

- f: A neural network parameterizing the derivative of $\mathbf{h}(t)$.
- t: Continuous time variable.
- θ: Learnable parameters of the neural network.

To solve the ODE, the adjoint sensitivity method (Pontryagin et al. 1962) can be employed. The adjoint method is a memory-efficient technique for computing gradients when training Neural ODEs. Instead of storing all intermediate states during the forward pass, the adjoint method computes gradients by solving an auxiliary ODE backward in time. This auxiliary ODE tracks how the loss function changes with respect to the systems states and model parameters, significantly reducing memory usage. For a detailed mathematical formulation and implementation of the adjoint method, refer to Chen et al. (2019, pp. 2–3).

5.2 Network Architecture

The input to the ODE Network represents the initial state of the robot arm, which includes the angles and angular velocities of the joints. The ODE network is implemented using PyTorch (*PyTorch* n.d.) as shown in Listing 2.

```
import torch.nn as nn

class OdeNet2(nn.Module):

  def __init__(self, features: int = 4, latent_dim: int = 128):
    super().__init__()
    self.net = nn.Sequential(
      nn.Linear(features, latent_dim),
      nn.Tanh(),
      nn.Linear(latent_dim, latent_dim),
      nn.Tanh(),
      nn.Linear(latent_dim, latent_dim),
      nn.Tanh(),
      nn.Linear(latent_dim, latent_dim),
      nn.Tanh(),
      nn.Linear(latent_dim, features)
    )

  def forward(self, t, z):
    return self.net(z)
```

Listing 2: Implementation of OdeNet2

5.3 Training the ODE Network

In order to train the ODE-Net, the following optimization techniques have been utilized to enhance the efficiency and accuracy of the training process:

- **Normalization:** All inputs are normalized to radians and scaled within the range $[-1, +1]$ to ensure consistent magnitude across features, which helps in stabilizing the training process.
- **Small Angle Deflection Initialization:** The initial values of joint positions and velocities are set to small angle deflections, such as $[0, 0.1]$, to improve convergence during training.
- **Reduced Time Range and Sampling:** A small time range of 5 to 15 seconds with only 30 to 100 sample points is used to make model training computationally feasible while retaining sufficient temporal resolution to learn the system's dynamics.
- **Diverse Analytical Trajectories:** Multiple analytical trajectories are generated to simulate a variety of motion patterns. This approach prevents overfitting and helps the model generalize across different scenarios and dynamic behaviors.
- **Batch Size:** A batch size of approximately 256 is chosen to prevent the model from converging to the average of the data. This ensures the model captures fine-grained dynamics and does not oversimplify the learned relationships.
- **Spectral Loss:** To enhance the model's ability to capture spectral characteristics of the system and not conerging to the average of the data, a spectral loss term is incorporated.

As shown in Listing 3, the training function computes the loss by comparing the integrated model's output data with the ground-truth data in both the time and frequency domains. This combined loss is backpropagated through the network to update the model's parameters.

```
def train_network(model: OdeNet, data_train: torch.utils.data.DataLoader, ...):
  ...
  for _, (z0, data) in enumerate(data_train):
    outputs = integrator(model, z0.float(), time_points.float())
    outputs = outputs.squeeze()
    data = data.squeeze()
    time_loss = criterion(outputs.squeeze(), data)
    fft_outputs = torch.fft.rfft(outputs, dim=1)  # FFT along the time dimension
    fft_data = torch.fft.rfft(data, dim=1)
    magnitude_outputs = torch.abs(fft_outputs)
    magnitude_data = torch.abs(fft_data)
    spectral_loss = criterion(magnitude_outputs, magnitude_data)
    train_loss = spectral_loss + time_loss
    train_loss.backward()
    optimizer.step()
    ...
```

Listing 3: Implementation of the training function

5.4 ODE-Net Architectures

In this study, various architectures of ODE networks were explored to optimize the trade-off between model complexity, accuracy and training efficiency. Different versions of the init states (Listing 4) were defined, to train the ODE network with different trajectories.

```python
def create_init_states(version: str | int) -> list[np.array]:
  version = int(version)
  np.random.seed(42)
  if version == 12:
    return [np.array([np.random.uniform(-5, 5), np.random.uniform(-5, 5), 0.0, 0.0])
  for _ in range(1008)]
  elif version == 20:
    return [np.array([np.random.uniform(0, 0.1), np.random.uniform(0, 0.1), 0.0, 0.0])
   for _ in range(1008)]
  elif version == 30:
    return [
        np.array(
            [np.random.uniform(-5, 5),
            np.random.uniform(-5, 5),
            np.random.uniform(-5, 5),
            np.random.uniform(-5, 5)]) for _ in range(1008)
    ]
```

Listing 4: Init state versions

As shown in Table 2, different versions of trajectories with different init states, time range and sample rates were defined.

t_{stop}	Δt	Init Version	Trajectory Version
5	$\frac{5}{30}$	20	20
5	$\frac{5}{30}$	30	30
5	$\frac{5}{100}$	20	40
15	$\frac{15}{100}$	20	41

Table 1: Trajectory Versions

Each model is characterized by its number of parameters, batch size and init states of the angular and angluar velocities (Listing 4). The different models (Appendix B) with their configuration are shown in Table 2.

Network Layers	**Parameters**	**Model V.**	**Traj. V.**	**Notes**
`OdeNet2(features=4, latent_dim=128)`	50,692	v2.5	v2.0	batch size = 256 epochs = 400 no perturbation
`OdeNet3(features=4, latent_dim=128)`	67,204	v2.6	v2.0	batch size = 256 epochs = 400 no perturbation
`OdeNet2(features=4, latent_dim=256)`	199,684	v2.7	v2.0	batch size = 256 epochs = 400 no perturbation
`OdeNet2(features=4, latent_dim=128)`	50,692	v2.8	v2.0	batch size = 256 epochs = 400 with perturbation
`OdeNet3(features=4, latent_dim=128)`	67,204	v33	v40	batch size = 256 epochs = 200 no perturbation
`OdeNet3(features=4, latent_dim=128)`	67,204	v35	v41	batch size = 256 epochs = 400 no perturbation

Table 2: ODE Network Architectures

5.5 Training and Evaluation Losses

In Figure 3, the training and evaluation losses of the ODE networks v2.5 - v2.8 are depicted.

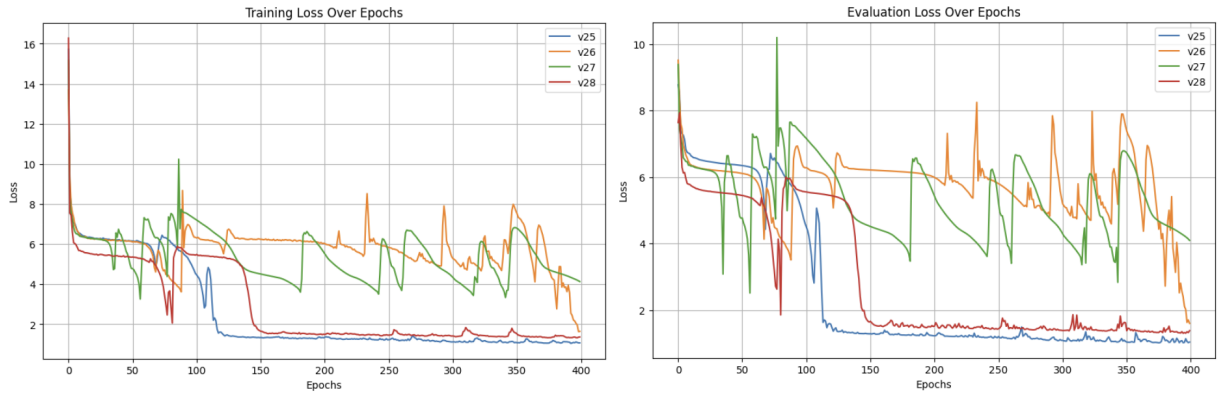

Figure 3: Training and evaluation losses of the ODE network

As observed, the training and evaluation losses for each version differ significantly. Model v2.5 demonstrates the fastest convergence and achieves the lowest loss, potentially due to an optimal balance of parameters and batch size. In contrast, model v2.7, despite having the most parameters, struggles to converge, possibly due to overparameterization. Model v2.6 (Appendix B.2), which includes an additional layer compared to the other models, converges toward the end of the training process. This suggests that with further training, model v2.6 could achieve an even lower loss, leveraging its more complex architecture to model intricate dynamics.

Model v2.8 was trained on trajectories with a perturbation variance of 0.01 (Appendix C.1). These trajectories simulate a more realistic real-world scenario where the robot experiences some noise in the system. As expected, this model converges later than model v2.5 and exhibits a slightly higher loss. This behavior aligns with expectations, as the model must account for and learn the noise present in the system.

The evaluation and training losses for all models visually converge similarly, indicating that none of the models are overfitting. This alignment suggests they generalize well to unseen data. However, the sampling strategy $\Delta t = \frac{5}{30}$seconds might also influence the models ability to capture the underlying dynamics. Exploring different resolutions could reveal further insights into their performance.

In order to observe the behavior of different time ranges and sample intervals the ODE networks v3.3, v3.5 were trained with the losses shown in Figure 4.

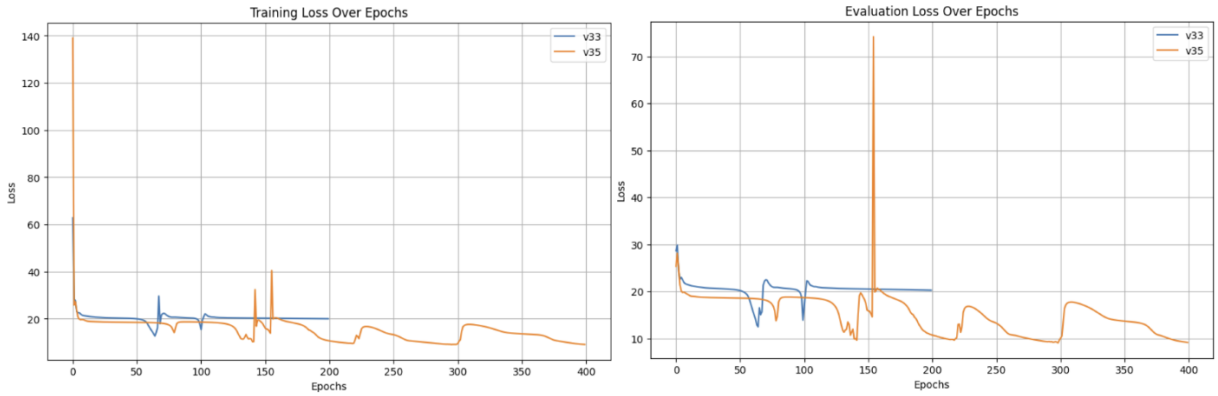

Figure 4: Training and evaluation losses of the ODE network

The model v3.3 had a higher sampling interval of $\Delta t = \frac{5}{100}$ seconds and the same time range of $t_{\text{stop}} = 5$ seconds as the other models. As observed the models have a higher training and evaluation loss compared

to the other models (v2.5 - v2.8). Model v3.5 has a higher time range of $t_{\text{stop}} = 15$ seconds and a higher sampling interval of $\Delta t = \frac{15}{100}$ seconds. The model also has a higher training and evaluation loss compared to the other models (v2.5 - v2.8).

5.6 Trajectories

In Figure 5 the trajectories of the ODE network models v2.5 and v3.5 are depicted.

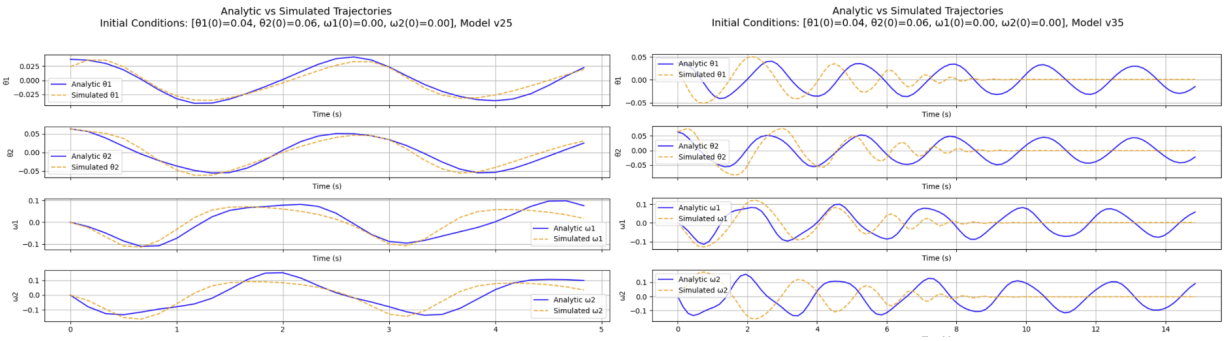

Figure 5: Trajectories of the ODE network models v2.5 and v3.5

As observed, the ODE network model v2.5 does a good job in approximating the ground truth data. In contrast, the ODE network model v3.5 does quite a good job at the first four seconds, but then starts to converge torwards the mean of the data.

5.7 Energy Conservation

To simplify the learning process and reduce the complexity of the solution space, the ODE networks were trained under the assumption of energy conservation, eliminating dissipation ($Q = 0$). This assumption allows the network to operate within a constrained solution space governed by conservation laws. Figure 6 illustrates different trajectories, corresponding to varying initial conditions specified in Listing 4, as modeled by ODE network version v3.5.

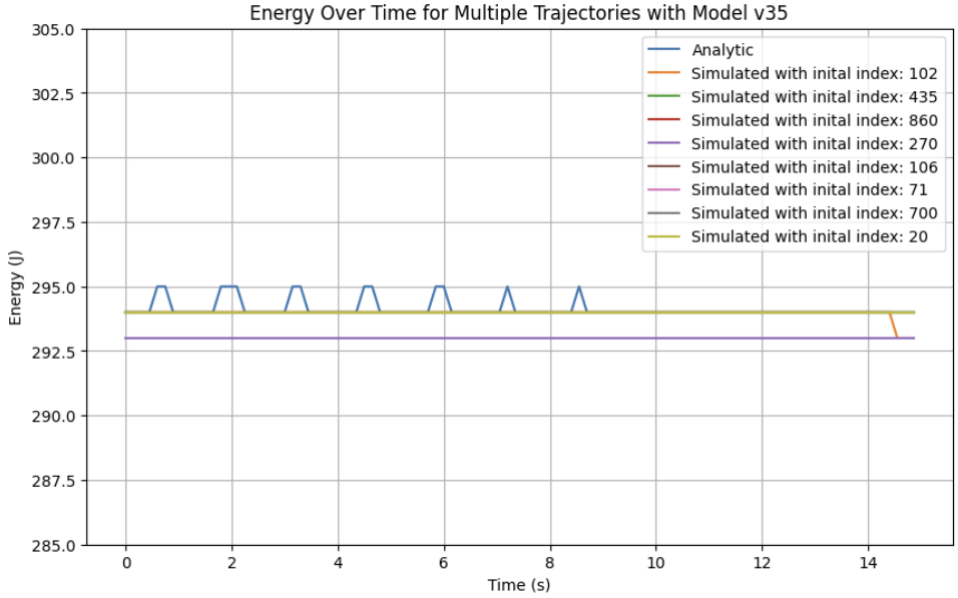

Figure 6: Trajectories of the ODE network models v2.5 and v3.5

As observed, model v3.5 performs exceptionally well in preserving the energy of the system. Even as the model converges toward the mean of the data, the total energy remains conserved, demonstrating its adherence to the system's physical constraints.

6 Conclusion and Outlook

6.1 Conclusion

As a general concept, the Lagrangian provides a systematic and physics-grounded framework for deriving equations of motion, making it a powerful approach for understanding the dynamics of complex systems. This method allows for an elegant and principled way to capture the relationships governing a system's behavior.

Training a neural ODE network presents several challenges that make it computationally expensive and technically demanding. The inherent difficulty in learning higher-frequency dynamics often results in the model converging toward lower-frequency solutions, limiting its ability to capture complex behavior accurately. To address these issues, a variety of optimization techniques must be employed. Strategies such as normalization, limiting the time range, adjusting batch sizes, including diverse trajectories, and leveraging spectral loss are essential for guiding the network to learn the intricate equations of motion. Despite these efforts, the process remains resource-intensive and sensitive to hyperparameter tuning. Even though these challenges exist, neural ODE networks offer some key advantages like the ability for continous time modeling which can incorporate regularly-sampled data (Chen et al. 2019, p. 6) with constant memory cost.

6.2 Outlook

As described in (Würsch 2024b, pp. 33–34) neural networks tend to learn low-frequency components first due to several factors. Small initial weights lead to smoother functions that capture broader patterns. Gradient descent also prioritizes low frequencies, as their loss surface is simpler and easier to optimize. Additionally, early network layers naturally represent smooth, low-frequency features, while deeper layers progressively handle more complex, high-frequency details. To address this limitation, different approaches can be considered.

Fourier Features
As stated in Tancik et al. (2020, p. 1), 'We show that passing input points through a simple Fourier feature mapping enables a multilayer perceptron (MLP) to learn high-frequency functions in low-dimensional problem domains'. By applying Fourier features, the network's ability to approximate high-frequency details can be enhanced, reducing the bias toward low frequencies.

Finite Basis Physics Informed Neural Networks (FBPINNs)
FBPINNs (Moseley, Markham, and Nissen-Meyer 2023) employ a 'divide and conquer' approach by splitting the problem domain into smaller, overlapping subdomains (Figure 7), a process known as domain decomposition. Each subdomain is assigned its own neural network, which is trained independently, allowing for parallel training across subdomains. This structure enables the networks to focus on localized solutions, making it easier to capture complex or high-frequency behaviors. Additionally, separate subdomain normalization ensures that each network operates within a well-scaled input space, improving convergence and stability during training. These features make FBPINNs highly scalable and efficient for solving physics-informed problems.

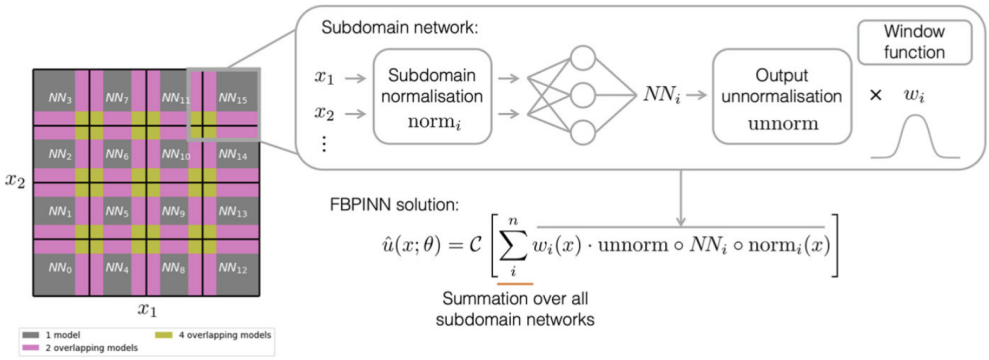

Figure 7: FBPINN workflow overview (Moseley, Markham, and Nissen-Meyer 2023, p. 12)

Lagrangian Neural Networks
Lagrangian Neural Networks (Cranmer et al. 2020) are a class of models that explicitly encode physical laws by parameterizing the system's Lagrangian. Unlike ODE networks, which rely on implicit learning of dynamics, LNNs enforce these principles directly within the network architecture. For a systems like explored in this work, LNNs could provide a significant advantage by:

- Improving long-term stability by ensuring that energy is conserved even over extended trajectories.
- Reducing the amount of training data required, as the network leverages known physical constraints.

List of Figures

1 Robot arm sketch . . . 3
2 Analytical trajectory of the robot arm . . . 8
3 Training and evaluation losses of the ODE network . . . 12
4 Training and evaluation losses of the ODE network . . . 12
5 Trajectories of the ODE network models v2.5 and v3.5 . . . 13
6 Trajectories of the ODE network models v2.5 and v3.5 . . . 13
7 FBPINN workflow overview (Moseley, Markham, and Nissen-Meyer 2023, p. 12) . . . 14

List of Tables

1 Trajectory Versions . . . 11
2 ODE Network Architectures . . . 11

Listings

1 ODE solver using `odeint` . . . 7
2 Implementation of OdeNet2 . . . 9
3 Implementation of the training function . . . 10
4 Init state versions . . . 11
5 Implementation of OdeNet2 . . . 20
6 Implementation of OdeNet3 . . . 20
7 Implementation of Pertubation . . . 21

References

Chen, Ricky T. Q. et al. (2019). "Neural Ordinary Differential Equations". In: URL: https://arxiv.org/pdf/1806.07366.

Cranmer, Miles et al. (2020). "LAGRANGIAN NEURAL NETWORKS". In: URL: https://arxiv.org/pdf/2003.04630.

Moseley, Ben, Andrew Markham, and Tarje Nissen-Meyer (2023). "Finite basis physics-informed neural networks (FBPINNs): a scalable domain decomposition approach for solving differential equations". In: DOI: 10.1007/s10444-023-10065-9. URL: https://link.springer.com/article/10.1007/s10444-023-10065-9.

Pontryagin, Lev Semenovich et al. (1962). *The Mathematical Theory of Optimal Processes*.

PyTorch (n.d.). Accessed: 2025-03-01. URL: https://pytorch.org.

SciPy (2024). Accessed: 2024-12-23. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html.

Tancik, Matthew et al. (2020). "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains". In: URL: https://arxiv.org/pdf/2006.10739.

Würsch, Christoph (2024a). *Computational Physics I: Deterministisches Chaos bei gekoppelten nichlinearen DGL*. OST Ostschweizer Fachhochschule.

— (2024b). *Physics InformedNeuralNetworks (PINNs)*. OST Ostschweizer Fachhochschule: Institute for Computational Engineering ICE.

A Euler Lagrange Equation

A.1 Euler Lagrange Equations to Theta 1

$$
\begin{aligned}
g\Bigg(& -l_1 m_1 \sin\left(\theta_1\left(t\right)\right) - l_1 m_2 \sin\left(\theta_1\left(t\right)\right) \\
& - l_1 l_2 m_2 \sin\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d}{dt}\theta_1\left(t\right) \frac{d}{dt}\theta_2\left(t\right) \\
& - l_1 \Big(l_1 m_1 \frac{d^2}{dt^2}\theta_1\left(t\right) \\
& \quad + m_2 \Big(l_1 \frac{d^2}{dt^2}\theta_1\left(t\right) - l_2 \Big(\frac{d}{dt}\theta_1\left(t\right) - \frac{d}{dt}\theta_2\left(t\right) \Big) \sin\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d}{dt}\theta_2\left(t\right) \\
& \quad + l_2 \cos\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d^2}{dt^2}\theta_2\left(t\right) \Big) \Big) \Bigg) = -\mu_1 \frac{d}{dt}\theta_1\left(t\right)
\end{aligned}
$$

A.2 Euler Lagrange Equations to Theta 2

$$
\begin{aligned}
-g l_2 m_2 \sin\left(\theta_2\left(t\right)\right) + l_1 l_2 m_2 \sin\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d}{dt}\theta_1\left(t\right) \frac{d}{dt}\theta_2\left(t\right) & \\
- l_2 m_2 \Bigg(- l_1 \Big(\frac{d}{dt}\theta_1\left(t\right) - \frac{d}{dt}\theta_2\left(t\right) \Big) \sin\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d}{dt}\theta_1\left(t\right) & \\
\quad + l_1 \cos\left(\theta_1\left(t\right) - \theta_2\left(t\right)\right) \frac{d^2}{dt^2}\theta_1\left(t\right) + l_2 \frac{d^2}{dt^2}\theta_2\left(t\right) \Bigg) & \\
= -\mu_2 \frac{d}{dt}\theta_2\left(t\right) &
\end{aligned}
$$

A.3 Second Order Derivative

$$
\begin{cases}
\frac{d^2}{dt^2}\theta_1(t) = -\frac{g l_1 l_2 m_1 \sin(\theta_1(t))}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad -\frac{g l_1 l_2 m_2 \sin(\theta_1(t))}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad +\frac{g l_1 l_2 m_2 \sin(\theta_2(t)) \cos(\theta_1(t)-\theta_2(t))}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad -\frac{l_1^2 l_2 m_2 \sin(\theta_1(t)-\theta_2(t)) \cos(\theta_1(t)-\theta_2(t)) \left(\frac{d}{dt}\theta_1(t)\right)^2}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad -\frac{l_1 l_2^2 m_2 \sin(\theta_1(t)-\theta_2(t)) \left(\frac{d}{dt}\theta_2(t)\right)^2}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad +\frac{l_1 \mu_2 \cos(\theta_1(t)-\theta_2(t)) \frac{d}{dt}\theta_2(t)}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2} \\
\quad -\frac{l_2 \mu_1 \frac{d}{dt}\theta_1(t)}{l_1^2 l_2 m_1 - l_1^2 l_2 m_2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1^2 l_2 m_2}, \\
\frac{d^2}{dt^2}\theta_2(t) = \frac{g l_1 l_2 m_1 m_2 \sin(\theta_1(t)) \cos(\theta_1(t)-\theta_2(t))}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad -\frac{g l_1 l_2 m_1 m_2 \sin(\theta_2(t))}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad +\frac{g l_1 l_2 m_2^2 \sin(\theta_1(t)) \cos(\theta_1(t)-\theta_2(t))}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad -\frac{g l_1 l_2 m_2^2 \sin(\theta_2(t))}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad +\frac{l_1^2 l_2 m_1 m_2 \sin(\theta_1(t)-\theta_2(t)) \left(\frac{d}{dt}\theta_1(t)\right)^2}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad +\frac{l_1^2 l_2 m_2^2 \sin(\theta_1(t)-\theta_2(t)) \left(\frac{d}{dt}\theta_1(t)\right)^2}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad +\frac{l_1 l_2^2 m_2^2 \sin(\theta_1(t)-\theta_2(t)) \cos(\theta_1(t)-\theta_2(t)) \left(\frac{d}{dt}\theta_2(t)\right)^2}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad -\frac{l_1 m_1 \mu_2 \frac{d}{dt}\theta_2(t)}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad -\frac{l_1 m_2 \mu_2 \frac{d}{dt}\theta_2(t)}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2} \\
\quad +\frac{l_2 m_2 \mu_1 \cos(\theta_1(t)-\theta_2(t)) \frac{d}{dt}\theta_1(t)}{l_1 l_2^2 m_1 m_2 - l_1 l_2^2 m_2^2 \cos^2(\theta_1(t)-\theta_2(t)) + l_1 l_2^2 m_2^2}
\end{cases}
$$

B ODE Network

B.1 ODE Net 2

```
class OdeNet2(nn.Module):

  def __init__(self, features: int = 4, latent_dim: int = 128):
    super().__init__()
    self.net = nn.Sequential(nn.Linear(features, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, features))

  def forward(self, t, z):
    return self.net(z)
```

Listing 5: Implementation of OdeNet2

B.2 ODE Net 3

```
class OdeNet3(nn.Module):

  def __init__(self, features: int = 4, latent_dim: int = 128):
    super().__init__()
    self.net = nn.Sequential(nn.Linear(features, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, latent_dim),
                             nn.Tanh(),
                             nn.Linear(latent_dim, features))

  def forward(self, t, z):
    return self.net(z)
```

Listing 6: Implementation of OdeNet3

C Robot Arm

C.1 Pertubation

```
def solve_odeint(self,
                 z0: np.array,
                 eq_theta1: Function,
                 eq_theta2: Function,
                 time_points: np.array,
                 pertubation_variance: float = None) -> np.array:
  z1, z2, z3, z4 = symbols('z1 z2 z3 z4')
  func_theta1 = lambdify([z1, z2, z3, z4], eq_theta1, modules='numpy')
  func_theta2 = lambdify([z1, z2, z3, z4], eq_theta2, modules='numpy')

  def derivatives(state, t):
    z1, z2, z3, z4 = state
    dz1 = z3  # z1' = z3
    dz2 = z4  # z2' = z4
    dz3 = func_theta1(z1, z2, z3, z4)  # z3' from equation_theta1
    dz4 = func_theta2(z1, z2, z3, z4)  # z4' from equation_theta2
    return [dz1, dz2, dz3, dz4]

  trajectory = integrate.odeint(derivatives, z0, time_points)
  if pertubation_variance:
    noise = np.random.normal(0, pertubation_variance, trajectory.shape)
    trajectory += noise
  return trajectory
```

Listing 7: Implementation of Pertubation