MAT292 Final Report

Modelling Neural ODEs for Baxter Robot Dynamics

Oscar Low, Atharv Kudchadkar

September 15, 2026

Abstract

This report investigates the efficacy of Neural Ordinary Differential Equations (NODEs) in learning the dynamics of a 7-DOF Baxter robot from torque and state data. The NODE is evaluated against the ground-truth end-effector trajectories by comparing the performance of three numerical solvers (Euler's Method, Runge-Kutta 4, and Velocity Verlet) in terms of predictive accuracy and inference time. The results show that the Velocity Verlet solver offers the greatest accuracy and fastest inference time. The code and its implemention details are located at robotics-nodes [1].

1 Introduction

In the control theory of robotics, motion is modelled by non-linear differential equations derived from classical mechanics [2]. Most traditional analytical models fail to account for friction, backlash, hysteresis, and oscillations from series elastic actuators, among other factors. These unpredictable forces make it difficult to predict robot motion over time.

Modelling the dynamics of robot motion is of great interest in robotics because of its potential in assisting control and planning, especially in precision-heavy tasks such as simulation, automation, and human interaction [3]. For example, simulators are typically used to train robots to do complex tasks (policies), and using dynamics models trained on real data can help simulators accurately reflect real motion, thereby enhancing the transferability of tasks to the real world. Good dynamics models are also essential for robots to adapt to tasks, such as moving different payloads.

Machine learning methods have been used to bridge the gap between idealized and real behavior by extracting these complex behaviours [4]. In general, these methods *learn* the structure and parameters of ODEs governing movement through a hybrid of brute force and physics-informed approaches.

Recently, Neural Ordinary Differential Equations (NODEs) have emerged as a continuous-time, memory-efficient alternative for problems like prediction and classification [5]. In particular, NODEs specialize in modelling the nonlinear differential equations that govern robot locomotion and control. This classic robot dynamics modelling problem involves predicting the motion of robots while a sequence of input torques is applied. To put this problem into perspective, while torques depend on the electrical signals and forces applied to the motors and actuators, the actual joint movements also depend on the weight of other joints, the payload, friction, operating conditions, and more.

This report investigates the efficacy of NODEs in learning the dynamics of a 7-Degree-of-Freedom (DOF) Baxter robot [6] from torque and state data. This simulates on how the robot will react to control inputs over time. Furthermore, the model will be evaluated with 3 different solvers, namely Euler's method, RK4, and the Velocity Verlet, to determine their accuracy and inference time.

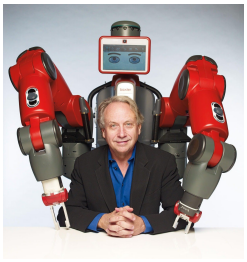


Figure 1: Baxter Robot [6]

2 Theoretical Foundation

2.1 Neural ODEs

Neural networks learn through data to produce an output from an input. The authors of the original NODE paper take inspiration from "iterative" type networks, such as RNNs and ResNets, in inputs are sequentially passed through discrete layers, and the state in a given layer depends on the state in the previous [5]. These networks can be modelled as

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f(\mathbf{z}_t, \theta_t) \tag{1}$$

where t is the hidden layer number, $\mathbf{z}_t$ is the hidden state, and θ are the model parameters.

In Neural ODEs, however, the input data continuously evolves. As the limit $\Delta t \to 0$, the number of layers increases and the step size between layers decreases, giving the following governing equation for updates:

$$\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}t} = f(\mathbf{z}(t), t, \theta) \tag{2}$$

The Neural ODE *learns* the evolution of the state, governed by the ODE in 2. The form and parameters of this ODE function are to be learned during training, as described later. In similar fashion to 1, the state at any time t_1 can then be found using integration:

$$\mathbf{z}(t_1) = \mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)\, \mathrm{d}t \tag{3}$$

2.2 Neural ODEs for Robotics and the Baxter Robot

The preceding equations are remarkably similar to the identities relating motion in physics: $\dot{x}(t) = v(t)$, $\dot{v}(t) = a(t)$, and so on. Thus, one distinct advantage of NODEs is their preservation of the dynamics identities. This shows how they may begin to be applied in the modelling of robot dynamics. Specifically, the 7-DOF Baxter dataset [7] gives 3 data for each of the 7 joints of the arm at each timestep: the *angle*, *angular velocity*, and *applied torque*. These are denoted, respectively, by $\mathbf{q}(t)$, $\dot{\mathbf{q}}(t)$, $\boldsymbol{\tau}(t) \in \mathbb{R}^7$.

To put this in the form of 3, the 2nd order ODE is reduced to a 1st order ODE by defining the state vector $\mathbf{x}(t) \in \mathbb{R}^{14}$. The evolution of the state is then given by $\mathrm{d}\mathbf{x}(t)/\mathrm{d}t$. Note that in 5, the angular acceleration $\ddot{\mathbf{q}}(t)$ is the *function to be learned*.

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{q}(t) \\ \dot{\mathbf{q}}(t) \end{bmatrix} \tag{4}$$

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \begin{bmatrix} \dot{\mathbf{q}}(t) \\ \ddot{\mathbf{q}}(t) \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}}(t) \\ f_\theta\big(\mathbf{q}(t), \dot{\mathbf{q}}(t), \boldsymbol{\tau}(t)\big) \end{bmatrix} \tag{5}$$

Then, this can be integrated as in 3 to predict the state at any future time.

Because these ODEs are so complex, NODEs use numerical integrators like Euler's method, Runge-Kutta 4, and Verlet integration [5]. An advantage with NODEs is being able to choose the type of integrator, which can allow different factors to be prioritized such as compute, memory, accuracy, and speed.

Another big advantage of using NODEs comes from the ability for numerical integrators to handle irregularly-sampled time series data (1). Sensor data from robots may not always be collected at uniform intervals due to network latency, malfunctions, or other disruptions, and solvers can naturally accommodate these irregular time steps by integrating over the actual time intervals between observations, as opposed to relying on fixed discrete time steps as in RNNs or Transformers.

Furthermore, the nature of the step update rules (1) allows for continuity and smoothness of the output paths, as opposed to jagged jumps in RNNs or Transformers. This is especially important in robotics, where smooth motion is essential for safety and precision.

2.3 ODE Solvers

In this paper, 3 solvers are compared for their precision, memory usage, and computational speed.

Table 1: Comparison of Numerical Integration Schemes.
Definitions: $\mathbf{x} = [\mathbf{q}, \dot{\mathbf{q}}]^\top$. Angular acceleration $\ddot{\mathbf{q}}_n = f_\theta(\mathbf{q}_n, \dot{\mathbf{q}}_n, \boldsymbol{\tau}_n)$.

Solver	**Type & Order**	**Update Equations**
Euler's Method	Explicit 1^{st} Order	$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot f_\theta(\mathbf{x}_n, \tau_n)$
Runge-Kutta 4 (RK4)	Explicit 4^{th} Order	$\mathbf{k}_1 = f_\theta(\mathbf{x}_n, \tau_n)$ $\mathbf{k}_2 = f_\theta(\mathbf{x}_n + \frac{h}{2}\mathbf{k}_1, \tau_{n+0.5})$ $\mathbf{k}_3 = f_\theta(\mathbf{x}_n + \frac{h}{2}\mathbf{k}_2, \tau_{n+0.5})$ $\mathbf{k}_4 = f_\theta(\mathbf{x}_n + h\mathbf{k}_3, \tau_{n+1})$ $\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$
Velocity Verlet	Symplectic 2^{nd} Order	$\mathbf{q}_{n+1} = \mathbf{q}_n + h\dot{\mathbf{q}}_n + \frac{h^2}{2}\mathbf{a}_n$ $\dot{\mathbf{q}}_{n+1} = \dot{\mathbf{q}}_n + \frac{h}{2}(\mathbf{a}_n + \mathbf{a}_{n+1})$

The velocity verlet (VV) method is a 2nd order symplectic solver. In general, *symplectic* methods are used to integrate Hamilton's equations of motion, and thus update position

while preserving energy. *Verlet* methods are used to integrate Newton's equations of motion, which also preserve energy. As in Table 1, this algorithm uses the current velocity to update position, but then uses the updated position to update velocity, which allows for a symmetric, time-reversible update. On the other hand, normal solvers like Euler's method and RK4 do not conserve energy, and thus can lead to divergence or instability (energy drift) in long-term predictions.

Therefore, it is predicted that the velocity verlet will be most accurate in long-term predictions for the robot arm.

Next, while the model learns dynamics in joint space $\mathbf{q}$ (i.e., the frame of reference of each joint), predictions must be transformed back to the robot's physical 3D workspace to be empirically validated. The Denavit-Hartenberg (DH) method was used to map joint angles to the end-effector position. A table of these parameters is given at [8]. For link i, the transformation matrix T_i^{i-1} is:

$$T_i^{i-1} = \begin{bmatrix} \cos q_i & -\sin q_i \cos \alpha_i & \sin q_i \sin \alpha_i & a_i \cos q_i \\ \sin q_i & \cos q_i \cos \alpha_i & -\cos q_i \sin \alpha_i & a_i \sin q_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

The global pose of the end-effector is computed by the product chain $T^{ee} = \prod_{i=1}^{7} T_i^{i-1}$, allowing the predicted trajectory to be visualized in Cartesian space (x, y, z).

3 Methodology

All training, testing, and visualization was done in `Python` using mainly `PyTorch` and `torchdiffeq`.

As shown in 7.4, class `ODEFunc` computes the derivative of the state from 5. It first computes the *function to be learned*, $\ddot{\mathbf{q}}(t) = f_\theta\big(\mathbf{q}(t), \dot{\mathbf{q}}(t), \boldsymbol{\tau}(t)\big)$, given by `self.net`. Then, it concatenates $\ddot{\mathbf{q}}(t)$ and $\dot{\mathbf{q}}(t)$ to form $\mathrm{d}\mathbf{x}(t)/\mathrm{d}t = [\dot{\mathbf{q}}, \ddot{\mathbf{q}}]^\top$, inside `ODEFunc.forward`. Note that the ODE function, $\ddot{\mathbf{q}}(t)$, is itself a multilayer perceptron, with thousands of parameters (weights and biases). In essence, this function can be anything, as long as it uses the correct dimension of inputs and outputs.

As shown in 7.5, class `NeuralODE` aggregates the ODE Function and the integrator, and predicts the next sequence of states given the input sequence of torques. The relatively simple algorithms of Euler's method and RK4 are included in `torchdiffeq`, while the Velocity Verlet integrator was hand-written (see 7.3).

Much of the ODE function training is handled by the `PyTorch` backend. However, it still leaves room for much optimization. These are discussed in the following sections.

3.1 Loss and Backpropagation

The predicted states were evaluated against the ground truth states using the mean squared error (MSE) of each of $\mathbf{q}$ and $\dot{\mathbf{q}}$. The formula is $\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$. The

total loss was calculated as $\text{Loss}_{\text{total}} = \text{MSE}(\mathbf{q}) + 0.5 \cdot \text{MSE}(\dot{\mathbf{q}})$. Note that these losses and thereby the gradients must be calculated at every single evaluation step of the ODE solver, an issue which will be addressed later.

3.2 Dataset Format

The datasets in [7] feature only four distinct types of motions of the end-effector: circle, random, squared, and spiral. This means that the results will be biased towards the evaluation of only these four motions. However, the training, validation, and testing portions of these datasets are still isolated to prevent overfitting.

3.3 Time and Torque Interpolation

The sampling rate of the dataset is defined in [9] to be a constant 500 Hz. Therefore, the ODE solvers were run with a constant timestep of $\Delta t = 0.002$ s.

However, higher-order solvers like RK4 require evaluation at half- or quarter-timesteps. In between data entries, there is no data on the input torques $\boldsymbol{\tau}(t)$. Therefore, torques must be interpolated, which can be done in different ways. Two techniques were used:

1. Constant Interpolator: assumes the control input to be constant between time steps.

$$\boldsymbol{\tau}(t) = \boldsymbol{\tau}_k, \text{where } k = \frac{t}{\Delta t} \tag{7}$$

2. Linear Interpolator: provides smooth continuous control input values.

$$\boldsymbol{\tau}(t) = \boldsymbol{\tau}_k + \frac{t - t_k}{\Delta t}(\boldsymbol{\tau}_{k+1} - \boldsymbol{\tau}_k) \tag{8}$$

3.4 Comparisons

In the results section later, different types of models and integrators will be evaluated against the ground truth end-effector trajectory, with the goal of finding a configuration of the model that offers the greatest accuracy and fastest inference.

Table 2: Trials conducted and objectives.

Models/Integrators	**Objective**
Torque Interpolators: Constant vs. Linear	Determine whether the choice of interpolator impacts accuracy.
Integrators: Euler vs. RK4 vs. Verlet	Determine the speed and accuracy of each ODE solver.
Different motions: Curved vs. Square vs. Random	Evaluate accuracy on different types of end-effector motions.

4 Results

4.1 Interpolation

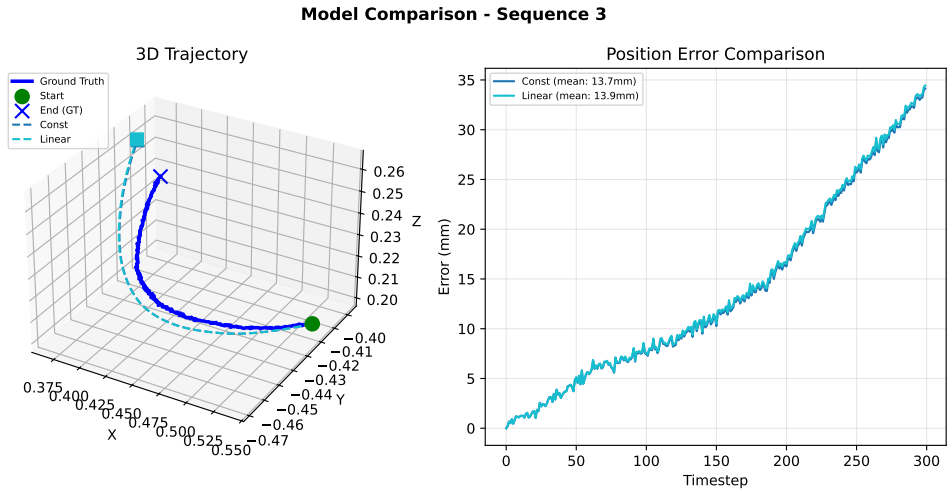
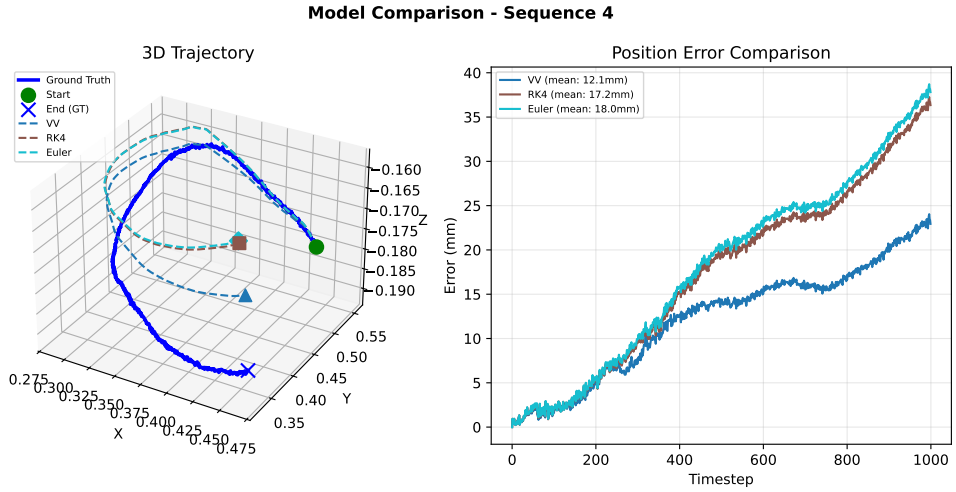


Figure 2: Predicted paths using constant (dark blue) and linear (light blue) interpolators.

For the first test, the constant and linear torque interpolators were compared. The mean errors were identical within a ± 0.02 mm margin. Therefore, from now on, only the constant interpolator will be used, since it requires less compute.

4.2 Integrators



Integrator	Euler	RK4	VV
Inference Time(s)	19.36	44.97	13.57

Figure 3: Top: Predicted paths using Euler (light blue), RK4 (brown), and Velocity Verlet (dark blue) solvers. Bottom: Inference times for each solver.

For many trial sequences (such as above), the Velocity Verlet solver consistently performed more accurately for a prediction window of 2 seconds, with RK4 and Euler's

falling far behind. This suggests how powerful the energy-conservation property of Verlet integration is for this robotics case. Furthermore, as a 2nd order solver, VV trumps 4th order solvers like RK4, showing that simply modelling the problem with more fidelity can achieve better results than using more computationally complex approaches.

In terms of inference time, RK4 took roughly 2.5x longer than Euler as expected, due to the added evaluation steps. However, more interestingly, VV was 30% faster than Euler despite being a 2nd order method. This is because both methods only need to call the ODE function once per step (see 7.3). In addition, there may be added overhead from calling the `torchdiffeq` library for Euler.

4.3 Different Motions

Next, the results varied depending on which type of motion the models were evaluated against. Because most of the data consisted of curved sequences, the three integrators had the lowest error for this type. VV was also the best for curved sequences because of its energy-conserving property.

The models accordingly struggled with square and random motions. More interestingly, VV actually performed worse for these types. This could be because the energy-conserving properties fail for sudden motions and direction changes, which dissipate energy quickly.

Table 3: Accuracies for different motions (two trials for each motion). See appendices.

Type	**Mean Error (mm)**		
	Euler	**RK4**	**VV**
Curved (4)	26.28	26.58	17.80
	19.55	19.68	16.85
Square (5)	53.76	53.61	72.63
	25.10	25.36	25.51
Random (6)	49.78	49.33	40.3
	21.64	20.61	32.21

5 Discussion

The results gave great insight on the viability of using neural ODEs to model robot dynamics. The Velocity Verlet was found to be the best solver for both accuracy and inference time. In particular, while the models excelled for curved motion, more work needs to be done for sudden motions such as squares.

In the next few sections, the limitations of this project and future steps are discussed.

5.1 Dataset Quality

As stated earlier, the Baxter dataset used only four distinct types of motions of the end-effector: circle, random, squared, and spiral. The results are also conducted on these four motions, and therefore the model cannot be seen as an accurate depiction of all real-world movements. Furthermore, the dataset is heavily weighted toward repetitive, curved motion (60% spiral, 30% random, 5% circle, 5% square) and the model accordingly struggles with sharp/square motions.

In addition, DH parameters are not universal; each individual Baxter model can have

slight differences in tolerances, friction, and wear. The parameters used [8] were from a different Baxter model than the dataset [7], and thus the ground-truth paths in the results may not be fully accurate to begin with. However, they are a good starting point.

The issues with dataset quality show how difficult it is to use small, online datasets for novel robotics research. Indeed, many researchers choose to generate their own data, to customize the range of movements and actions. In addition, no unified benchmark for robot dynamics currently exists, unlike MNIST or COCO for computer vision.

5.2 Memory Usage

In their raw form, Neural ODEs face the same problem as discrete NN's like ResNets, which is that prediction window lengths are limited by memory availability during training. As discussed earlier, traditional backpropagation works by storing the state (weights and biases) of each layer, which inevitably requires a lot of memory. This means that an integration with 500 time steps requires the same memory as a ResNet with 500 layers; furthermore, the required memory increases with the number of time steps.

The authors of the original NODE paper [5] proposed the adjoint sensitivity method to get around this problem, which uses another ODE to compute gradients backwards in time, thereby eliminating the need to store intermediate states and reducing memory usage to $\mathcal{O}(1)$ (see 7.1). However, the adjoint method is only available in `torchdiffeq` for Euler and RK4, and not for the VV method used in this paper. In the future, implementing the adjoint method for the Velocity Verlet could allow for much longer prediction windows.

5.3 Capturing Dynamics

The neural ODEs in this paper do a somewhat better job at capturing dynamics, at least compared to "brute-force" methods like RNNs and Transformers. First, the reduction in 5 preserves the identity $\mathrm{d}\mathbf{q}(t)/\mathrm{d}t = \dot{\mathbf{q}}(t)$. Secondly, the Velocity Verlet solver preserves the energy of the system.

However, there exist much more analytically-driven methods informed by fundamental principles, such as the Newton-Euler or the Euler-Lagrange equations of motion [4], which explicitly conserve inertia or energy respectively, but these were deemed too complex.

In a future project, there are two very different possible directions: one, a pure, "brute-force", black-box approach such as Transformers; or two, a more analytically-informed approach such as Physics-Informed Neural Networks.

6 Conclusion

In this report, a Neural Ordinary Differential Equation (NODE) framework for modelling the 7-DOF dynamics of a Baxter Robot was succesfully implemented and evaluated. By modelling the nonlinear differential equations that govern robot locomotion and control, the model displayed its ability to predict the end-effector trajectories from torque inputs.

The model demonstrates that while NODEs are a viable alternative to mainstream NNs for robotics, their success is highly dependent on the type of integrator used. The symplectic Velocity Verlet solver proved to be the most optimal choice, giving the best accuracy and computation time. It gave inference times 30% faster than Euler's Method, and 2.5x faster than RK4. However, VV struggled with sudden motions due to energy dissipation, with a 35% and 50% increase in mean error when compared to the two other integrators for square and random motions respectively.

Neural ODEs have great potential in robotics, where better and faster prediction can increase the proficiency of robots in assisting humanity.

References

[1] O. Low and A. Kudchadkar, "robotics-nodes," 2025. `https://github.com/goldenpuffle123/robotics-nodes`.

[2] J. Xu, E. Heiden, I. Akinola, D. Fox, M. Macklin, and Y. Narang, "Advancing robotics development with neural dynamics in newton," 2025. `https://developer.nvidia.com/blog/advancing-robotics-development-with-neural-dynamics-in-newton/`.

[3] B. Ai, S. Tian, H. Shi, Y. Wang, T. Pfaff, C. Tan, H. I. Christensen, H. Su, J. Wu, and Y. Li, "A review of learning-based dynamics models for robotic manipulation," *Science Robotics*, vol. 10, no. 106, p. eadt1497, 2025.

[4] A. R. Geist and S. Trimpe, "Structured learning of rigid-body dynamics: A survey and unified view from a robotics perspective," 2021.

[5] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," 2019.

[6] T. Steuer, *The Baxter Robot: Exploring the robot by Rapid Prototyping*. Phd thesis, University of Calgary, Calgary, Alberta, September 2014. `https://utouch.cpsc.ucalgary.ca/docs/thesisTim.pdf`.

[7] B. Valencia-Vidal, G. Jimenez-Perera, N. R. Luque Sola, E. Ros, and F. Barranco, "Baxter robot dataset for learning dynamic models," 2025. `https://doi.org/10.5281/zenodo.17035193`.

[8] A. Kumar, A. Sahasrabudhe, C. Perugu, S. Nirgude, and A. Murugan, "Kinematics & dynamics library for baxter arm," 2024.

[9] G. Jimenez-Perera, B. Valencia-Vidal, N. R. Luque, E. Ros, and F. Barranco, "Informed federated learning to train a robotic arm inverse dynamic model," *IEEE Robotics and Automation Letters*, vol. 10, no. 10, p. 11022, 2025.

7 Appendix

7.1 Adjoint Method

$$\mathbf{a}(t) = \frac{\partial L}{\partial \mathbf{x}(t)} \tag{9}$$

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^\top \frac{\partial f_\theta}{\partial \mathbf{x}} \tag{10}$$

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f_\theta}{\partial \theta} dt \tag{11}$$

7.2 Trajectories for Different Motions

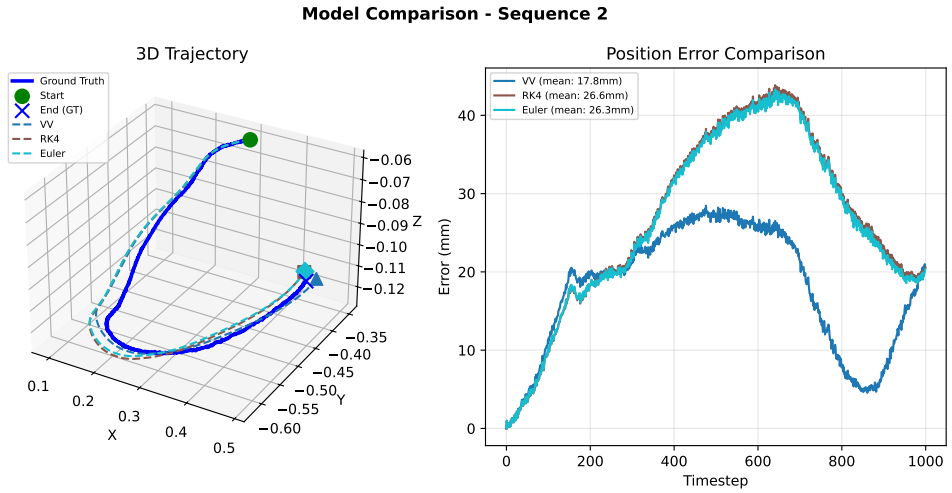


Figure 4: Curved motion. Predicted paths using Euler (light blue), RK4 (brown), and Velocity Verlet (dark blue) solvers.

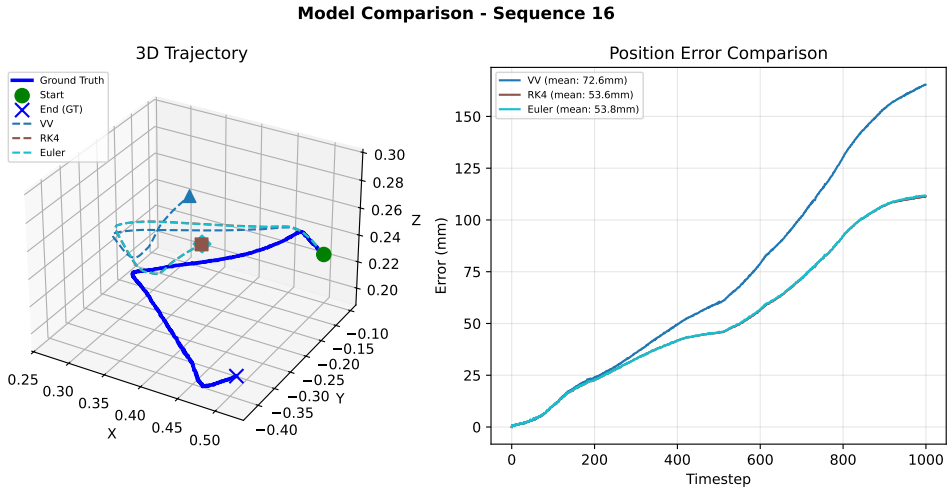


Figure 5: Squared motion. Predicted paths using Euler (light blue), RK4 (brown), and Velocity Verlet (dark blue) solvers.

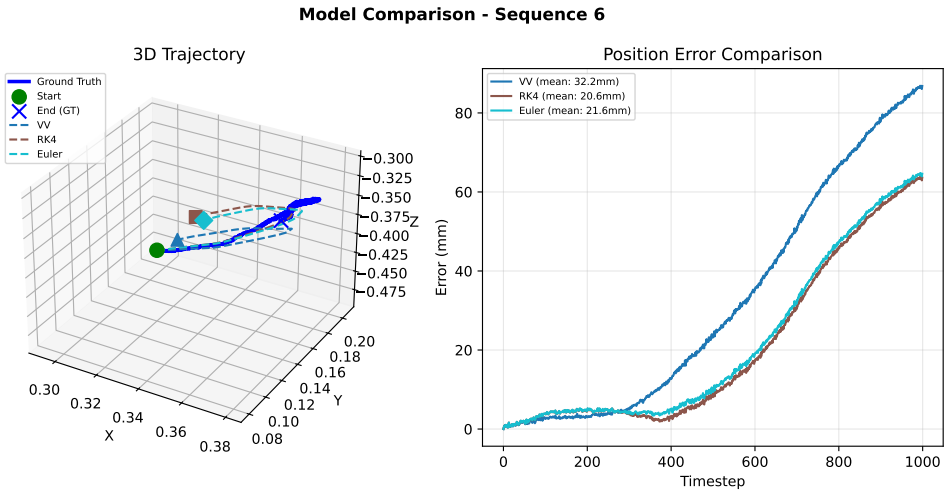


Figure 6: Random motion. Predicted paths using Euler (light blue), RK4 (brown), and Velocity Verlet (dark blue) solvers.

7.3 Velocity Verlet

```python
def velocity_verlet(func, y0, t, **kwargs):
    device = y0.device
    dtype = y0.dtype
    batch_size = y0.shape[0]
    state_dim = y0.shape[1]
    dof = state_dim // 2 # degrees of freedom of the arm

    num_steps = len(t)
    states = torch.zeros(num_steps, batch_size, state_dim, device=device, dtype=dtype) # initialize tensor of output states

    # Initial state
    y = y0.clone()
    states[0] = y

    # Extract initial position and velocity components
    q = y[:, :dof]  # positions (angles)
    v = y[:, dof:]  # velocities

    # Initial acceleration: predicted using ODE function
    dy = func(t[0], y)
    a = dy[:, dof:]  # acceleration part

    for i in range(1, num_steps):
        dt = t[i] - t[i-1]

        # Step 1: Update position using current velocity and acceleration
        # q_{n+1} = q_n + v_n * dt + 0.5 * a_n * dt^2
        q_next = q + v * dt + 0.5 * a * dt * dt

        # Step 2: Compute new acceleration at predicted state
```

```
        # Use predictor velocity: v_pred = v_n + a_n * dt
        v_pred = v + a * dt
        y_pred = torch.cat([q_next, v_pred], dim=-1)
        dy_next = func(t[i], y_pred)
        a_next = dy_next[:, dof:]

        # Step 3: Update velocity using average acceleration
        # v_{n+1} = v_n + 0.5 * (a_n + a_{n+1}) * dt
        v_next = v + 0.5 * (a + a_next) * dt

        # Store new predicted state
        q = q_next
        v = v_next
        a = a_next
        y = torch.cat([q, v], dim=-1)
        states[i] = y

    return states
```

7.4 ODE Function

```
class ODEFunc(nn.Module):
    def __init__(self, dof = 7, hidden_dim = 128):
        super().__init__()
        self.dof = dof
        self.state_dim = dof * 2  # angles + velocities
        self.interpolator = None
        # Input: state (14) + torque (7) = 21
        # Output: acceleration (7): learned
        self.net = nn.Sequential(
            nn.Linear(self.state_dim + dof, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, dof),  # Only 7-dim acceleration
        )
        # Reasonable init for dynamics learning
        nn.init.normal_(self.net[-1].weight, std=0.01)
        nn.init.zeros_(self.net[-1].bias)

    def set_interpolator(self, interpolator):
        self.interpolator = interpolator

    def forward(self, t, state):
        torque = self.interpolator(t)  # [batch, dof]
        q_dot = state[:, self.dof:]  # Extract velocities
        x = torch.cat([state, torque], dim=-1)
        q_ddot = self.net(x)  # Predicts acceleration
        return torch.cat([q_dot, q_ddot], dim=-1)
```

7.5 Neural ODE

```python
class NeuralODE(nn.Module):
    def __init__(self, dof = 7, hidden_dim = 128, dt = 0.002):
        super().__init__()
        self.dof = dof
        self.state_dim = dof * 2
        self.dt = dt
        self.ode_func = ODEFunc(dof=dof, hidden_dim=hidden_dim)

    def forward(self, initial_state, torques):
        batch_size, seq_len, _ = torques.shape
        device = initial_state.device
        # Time points
        t_span = torch.linspace(0, seq_len * self.dt, seq_len + 1, device=device)
        # Set batched interpolator
        self.ode_func.set_interpolator(TorqueInterpolatorConstant(torques, self.dt))
        # Batched integration: returns [seq_len+1, batch, state_dim]
        states = odeint(self.ode_func, initial_state, t_span)
        # Permute to [batch, seq_len+1, state_dim] and exclude t=0
        return states[1:].permute(1, 0, 2)
```