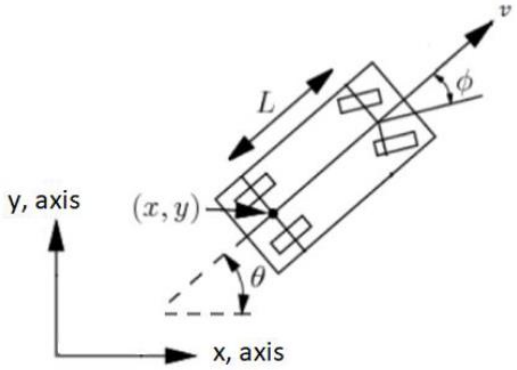
Project 1: Gradient-based Algorithms and Differentiable Program

MAE 598/494 Design Optimization

Emilio Montoya

Problem Formulation:

For this project I explored the dynamical systems of autonomous vehicles. I started out by making a sketch for my type of system and then defined the state variables such as position, velocity, steering angle, among others. Then I thought of what the variables should be the controller should operate to make the path of the vehicle the most optimum. For this reason, I went ahead to make acceleration and steering angle the input variables for the controller. The following figure shows how I derived the state variables of the system.



After performing dynamical analysis, we get the following equations of motion:

$$\dot{x} = v\cos\theta \qquad \dot{y} = v\sin\theta \qquad \dot{\theta} = \frac{v}{L}\tan\Phi$$

For the state space equations, we get:

$$x_{(t+1)} = x_t + \dot{x}\Delta t \qquad y_{(t+1)} = y_t + \dot{y}\Delta t$$
$$\dot{x}_{(t+1)} = \dot{x}_t + a_t\Delta t \qquad \dot{y}_{(t+1)} = \dot{y}_t + a_t\Delta t$$
$$\theta_{(t+1)} = \theta_t + \dot{\Phi}_t\Delta t$$

Where Δt is a time interval.

For the controller, we let the closed-loop controller be the following function:

$$u(t) = \left[a(t), \dot{\Phi}(t)\right] = \pi_w\big(x(t)\big)$$

Where $\pi_w(\,)$ is a neural network with parameters w, which are determined through optimization.

For each time step, we assign a loss as a function of the control input and the state: $l\big(x(T), u(T)\big)$ In this example, we simply set $l\big(x(T), u(T)\big) = 0$ for all $t = 1, \dots, T-1$, where T is the final step, and $l\big(x(T), u(T)\big) = \|x(T)\|^2$

The optimization problem can be formulated as:

$$\underbrace{min}_{w} \ \|x(T)\|^2$$
$$x_{(t+1)} = x_t + \dot{x}\Delta t$$
$$y_{(t+1)} = y_t + \dot{y}\Delta t$$
$$\dot{x}_{(t+1)} = \dot{x}_t + a_t\Delta t$$
$$\dot{y}_{(t+1)} = \dot{y}_t + a_t\Delta t$$
$$\theta_{(t+1)} = \theta_t + \dot{\Phi}_t\Delta t$$
$$u(t) = \pi_w(x(T), \forall t = 1, \dots, T-1$$

Programing:

To begin programing we first need to define what are the environmental conditions as well as setting up our state space equations for the system. This will make our equations be in a differentiable form so that we can input it into a pytorch script, and it should be able to run it.

We chose a time interval, Δt, of 0.1, an acceleration constant of 0.1 and a maximum rotation rate of 0.5.

```
# environment parameters
FRAME_TIME = 0.1  # time interval
BOOST_ACCEL = 0.1  # Acceleration constant
OMEGA_RATE = .5  # max rotation rate
```

We will be setting up out state tensor as [0, 0, cos(θ), sin(θ), 0]

And the delta state of theta as omega_rate * Δt * [0,0,0,0 ω(t)]

Then we set up our state space equations in matrix form in a state of t+1 since we are using the current position of the system as our feedback state therefore the neural network can adjust the controller accordingly.

$$x(t+1) = \begin{bmatrix} 1 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \end{bmatrix} + \begin{bmatrix} 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} * (delta\ state * \Delta t)$$

Where delta state is acceleration constant * Δt * [0, 0, a(t)sin(θ), a(t)cos(θ), 0]

Velocity and orientation are updated by state = state + delta_state + delta_state_theta

```
class Dynamics(nn.Module):

    def __init__(self):
        super(Dynamics, self).__init__()

    def forward(self, state, action):
        """
        action[0] = acceleration
        action[1] = phi_dot

        state[0] = x
        state[1] = y
        state[2] = vx
        state[3] = vy
        state[4] = theta

        """


        state_tensor = torch.zeros((1, 5))
        state_tensor[0, 3] = torch.sin(state[0, 4])
        state_tensor[0, 2] = torch.cos(state[0, 4])


        delta_state = BOOST_ACCEL * FRAME_TIME * torch.mul(state_tensor, action[0, 0].reshape(-1, 1))
```

```
        # Theta
        delta_state_phi = FRAME_TIME * OMEGA_RATE * torch.mul(torch.tensor([0., 0., 0., 0., 1.]),action[0, 1].reshape(-1, 1))

        # Update state
        step_mat = torch.tensor([[1., 0.,FRAME_TIME, 0., 0.],
                                 [0., 1., 0., FRAME_TIME, 0.],
                                 [0., 0., 1., 0., 0.],
                                 [0., 0., 0., 1., 0.],
                                 [0., 0., 0., 0., 1.]])

        shift_mat = torch.tensor([[0., 0.,FRAME_TIME, 0., 0.],
                                 [0., 0., 0., FRAME_TIME, 0.],
                                 [0., 0., 0., 0., 0.],
                                 [0., 0., 0., 0., 0.],
                                 [0., 0., 1., 1., 0]])

        state = torch.matmul(step_mat, state.T) + torch.matmul(shift_mat, delta_state.T)
        state = state.T

        state = state + delta_state   + delta_state_phi

        return state
```

Then, we define a controller that will control the acceleration and angular velocity of the system. We are going to do this by implementing another neural network. This neural network takes the 5 state variables of the system and outputs the 2 action variables mentioned above. We constrained the range of the angular velocity to [-1,1] that way it can rotate in either direction. Then the acceleration is also constrained but from the range of [0, 2] this way we can stay at a constant velocity or accelerate as needed.

```
class Controller(nn.Module):
    def __init__(self, dim_input, dim_hidden, dim_output):
        """
        dim_input: # of system states
        dim_output: # of actions
        dim_hidden:
        """
        super(Controller, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(dim_input, dim_hidden),
            nn.Tanh(),
            nn.Linear(dim_hidden, dim_hidden),
            nn.Tanh(),
            nn.Linear(dim_hidden, dim_output),
            nn.Sigmoid())

    def forward(self, state):
        action = self.network(state)
        action = (action - torch.tensor([0., 0.5]))*2  # bound theta_dot range -1 to 1
        return action
```

Now that the controller and dynamic system is defined, we can move onto the simulation. To do that we first start by defining initial conditions for our system. We are starting at the coordinate (-3,-3) with an initial v_y=1 m/s and θ=0.5 rad. We can write this as $\vec{x}_o = [-3, -3, 0, 1, 0.5]$ $\vec{x}$

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

    def __init__(self, controller, dynamics, T):
        super(Simulation, self).__init__()
        self.state = self.initialize_state()
        self.controller = controller
        self.dynamics = dynamics
        self.T = T
        self.theta_trajectory = torch.empty((1, 0))
        self.u_trajectory = torch.empty((1, 0))

    def forward(self, state):
        self.action_trajectory = []
        self.state_trajectory = []
        for _ in range(T):
            action = self.controller(state)
            state = self.dynamics(state, action)
            self.action_trajectory.append(action)
            self.state_trajectory.append(state)
        return self.error(state)

    @staticmethod
    def initialize_state():
        state = [[-3.,-3.,0.,1.,0.]]
        return torch.tensor(state, requires_grad=False).float()

    def error(self, state):
        return torch.mean(state ** 2)
```

Now that we have developed the computational graph for the error to be minimized, a gradient of the error with respect to the controller can be calculated based on the controller's weights. To minimize the error, we can implement a gradient descend algorithm. For this type of optimization technique, we shall use the quasi-Newton type algorithm that uses limited memory of past gradients to approximate the current Hessian. This type of algorithm guarantees the Hessian to be positive definite making it feasible to be used on non-convex scenarios.

```python
class Optimize:

    # create properties of the class (simulation, parameters, optimizer, lost_list). Where to receive input of objects

    def __init__(self, simulation):
        self.simulation = simulation # define the objective function
        self.parameters = simulation.controller.parameters()
        self.optimizer = optim.LBFGS(self.parameters, lr=0.01) # define the opmization algorithm
        self.loss_list = []

    # Define loss calculation method for objective function

    def step(self):
        def closure():
            loss = self.simulation(self.simulation.state)  # calculate the loss of objective function
            self.optimizer.zero_grad()
            loss.backward() # calculate the gradient
            return loss

        self.optimizer.step(closure)
        return closure()

    # Define training method for the model
```

```python
    def train(self, epochs):
        # self.optimizer = epoch
        l = np.zeros(epochs)
        for epoch in range(epochs):
            self.epoch = epoch
            loss = self.step() # use step function to train the model
            self.loss_list.append(loss) # add loss to the loss_list
            print('[%d] loss: %.3f' % (epoch + 1, loss))

            l[epoch]=loss
            self.visualize()

        plt.plot(list(range(epochs)), l)

        plt.title('Objective Function Convergence Curve')
        plt.xlabel('Training Iteration')
        plt.ylabel('Error')
        plt.show()
        self.animation(epochs)

    # Define result visualization method
```

```python
def visualize(self):
    data = np.array([self.simulation.state_trajectory[i][0].detach().numpy() for i in range(self.simulation.T)])
    x = data[:, 0]
    y = data[:, 1]
    vx = data[:, 2]
    vy = data[:, 3]
    theta = data[:, 4]
    action_data = np.array([self.simulation.action_trajectory[i][0].detach().numpy() for i in range(self.simulation.T)])
    acceleration = action_data[:,0]
    frame = range(self.simulation.T)

    fig, ax = plt.subplots(1, 4, tight_layout = 1, figsize = (15, 5))

    ax[0].plot(x, y, c = 'b')
    ax[0].set_xlabel("X")
    ax[0].set_ylabel("Y")
    ax[0].set(title=f'Displacement plot(x-y) at frame {self.epoch}')

    ax[1].plot(frame, vx, c = 'c', label = "Velocity in x")
    ax[1].plot(frame, vy, c = 'r', label = "Velocity in y")
    ax[1].set_xlabel("Time")
    ax[1].set_ylabel("Velocity (m/s)")
    ax[1].legend(frameon=0)
    ax[1].set(title =f'Velocity plot at frame {self.epoch}')
```

```python
        ax[2].plot(frame, theta, c = 'g', label = "theta")
        ax[2].set_xlabel("Time interval")
        ax[2].set_ylabel("Theta")
        ax[2].legend(frameon=0)
        ax[2].set(title=f'Theta plot at {self.epoch}')

        ax[3].plot(frame, acceleration, c = 'y', label = "Acceleration")
        ax[3].set_xlabel("Time interval")
        ax[3].set_ylabel("Acceleration")
        ax[3].legend(frameon=0)
        ax[3].set(title=f'Acceleration plot at {self.epoch}')
        plt.show()

    def animation(self, epochs):
            # Size
        length = 0.10          # m
        width = 0.02          # m

        #
        v_exhaust = 1
        print("Generating Animation")
        steps = self.simulation.T + 1
        final_time_step = round(1/steps,2)
        f = IntProgress(min = 0, max = steps)
        display(f)

        data = np.array([self.simulation.state_trajectory[i][0].detach().numpy() for i in range(self.simulation.T)])
        action_data = np.array([self.simulation.action_trajectory[i][0].detach().numpy() for i in range(self.simulation.T)])
```

```
    x_t = data
    u_t = action_data
    print(x_t.shape, u_t.shape)

    fig = plt.figure(figsize = (5,8), constrained_layout=False)
    ax1 = fig.add_subplot(111)
    plt.axhline(y=0., color='b', linestyle='--', lw=0.8)

    ln1, = ax1.plot([], [], linewidth = 10, color = 'lightblue') # kart body
    ln6, = ax1.plot([], [], '--', linewidth = 2, color = 'orange') # trajectory line
    ln2, = ax1.plot([], [], linewidth = 4, color = 'tomato') # acceleration line

    plt.tight_layout()

    ax1.set_xlim(-10, 10)
    ax1.set_ylim(-10, 10)
    ax1.set_aspect(1)  # aspect of the axis scaling, i.e. the ratio of y-unit to x-unit

    def update(i):
        kart_theta = x_t[i, 4]

        kart_x = x_t[i, 0]
        # length/1 is just to make kart bigger in animation
        kart_x_points = [kart_x + length/1 * np.sin(kart_theta), kart_x - length/1 * np.sin(kart_theta)]
```

```
        kart_y = x_t[i, 1]
        kart_y_points = [kart_y + length/1 * np.cos(kart_theta), kart_y - length/1 * np.cos(kart_theta)]

        ln1.set_data(kart_x_points, kart_y_points)

        acceleration_mag = u_t[i, 0]
        acceleration_angle = -u_t[i, 1]

        flame_length = (acceleration_mag) * (0.4/v_exhaust)
        flame_x_points = [kart_x_points[1], kart_x_points[1] - flame_length * np.sin(kart_theta)]
        flame_y_points = [kart_y_points[1], kart_y_points[1] - flame_length * np.cos(kart_theta)]

        ln2.set_data(flame_x_points, flame_y_points)
        ln6.set_data(x_t[:i, 0], x_t[:i, 1])
        f.value += 1

    playback_speed = 5000 # the higher the slower
    anim = FuncAnimation(fig, update, np.arange(0, steps-1, 1), interval= final_time_step * playback_speed)

    # Save as GIF
    writer = PillowWriter(fps=20)
    anim.save("kart_parking.gif", writer=writer)
```

Then we just need to specify some hyperparameters to see how the control system behaves and works.

```
T = 100  # number of time steps of the simulation
dim_input = 5  # state space dimensions
dim_hidden = 6  # latent dimensions
dim_output = 2  # action space dimensions
d = Dynamics()
c = Controller(dim_input, dim_hidden, dim_output)
s = Simulation(c, d, T)
o = Optimize(s)
o.train(50)  # training with number of epochs (gradient descent steps)
```

Plots of the system can be found in Appendix 1, Section 1 of the report

Running the next command will allow python to output a gif of the simulation

```
from IPython.display import display, Image

display(Image(filename='kart_parking.gif'))
```

Analysis of the Results:

First, from the simulation we got the expected path to the place of destination as well as getting to the specified destination pointing in the right direction. During my time tweaking the

code I changed the acceleration constant, and the max rate of rotation multiple times and I noticed that the displacement plots as well as the acceleration plots changed quite significantly. Moreover, the velocity and theta plots also changed but not as much as the position and acceleration plots. There were instances in which the kart would not reach the final position and times in which it would overshoot it and would not be able to change direction.

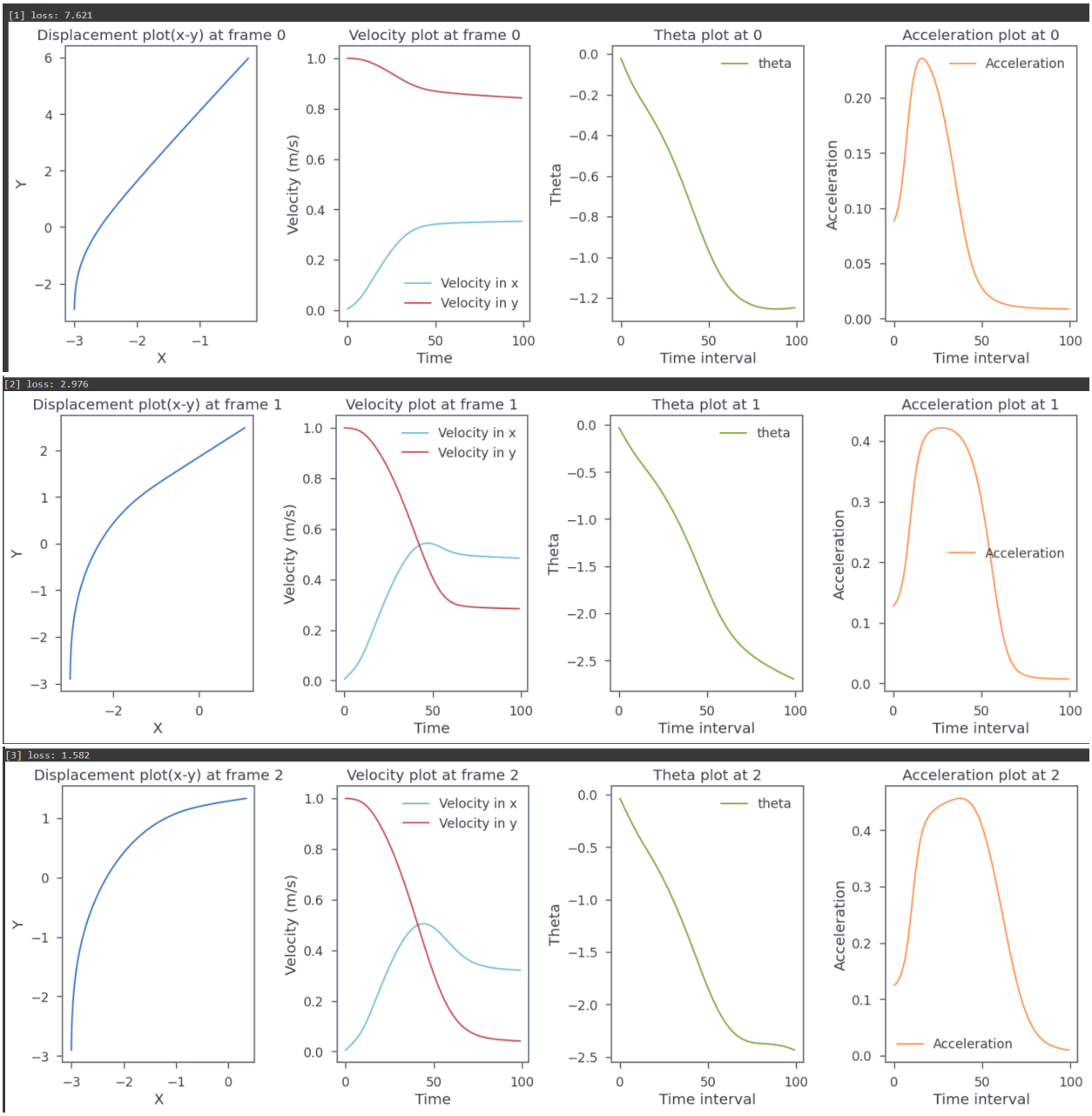
Next, when I was coding, I noticed that I had 6 state equations, but the tensor only wanted 5 input variables. I did some research on the matter, and it was pretty much a constrain on the tensor function that was supposed to be fixable by setting a NxN matrix as input, so I went ahead to change the state tensor to a 6x6 matrix and reference all the variables to be a 6x6 input, but the function still would not like it. I discussed this issue with an engineering mentor, and we concluded that that sixth variable was one of the controller variables. So, I just rearranged the state matrix as a 5x5. However, when analyzing the system's GIF, I noticed that even though the kart's motion was accurate the kart would change direction to show how the system was decelerating. For instance, if the kart was facing up and it started to decelerate the animation would turn the kart 180 degrees and display the acceleration motion to show this change in motion.

All in all, this project helped me a lot in when it comes to problem formulation because I knew all the different optimization techniques but was struggling in how to apply them. Honestly at the beginning of the project I did not know where to start or what system to model. So, I opted for a system whose dynamics I am familiar with me. Luckly for me it had some similar features to the example that was presented for this project. But defining the tensors for my problem was still challenging enough. I believe I changed the tensors one too many times for my liking because of the problem I was having with the input for the controller.

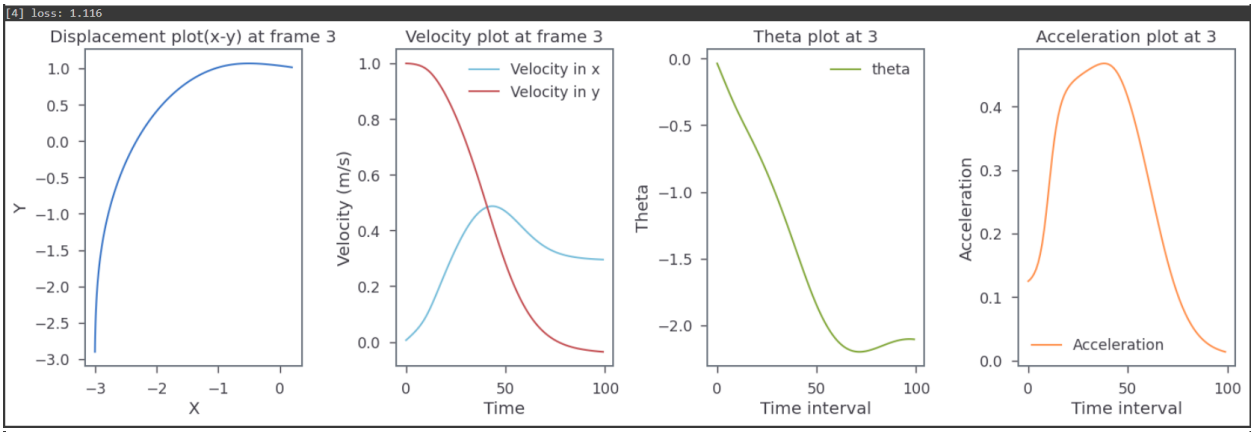
Appendix 1

Section 1

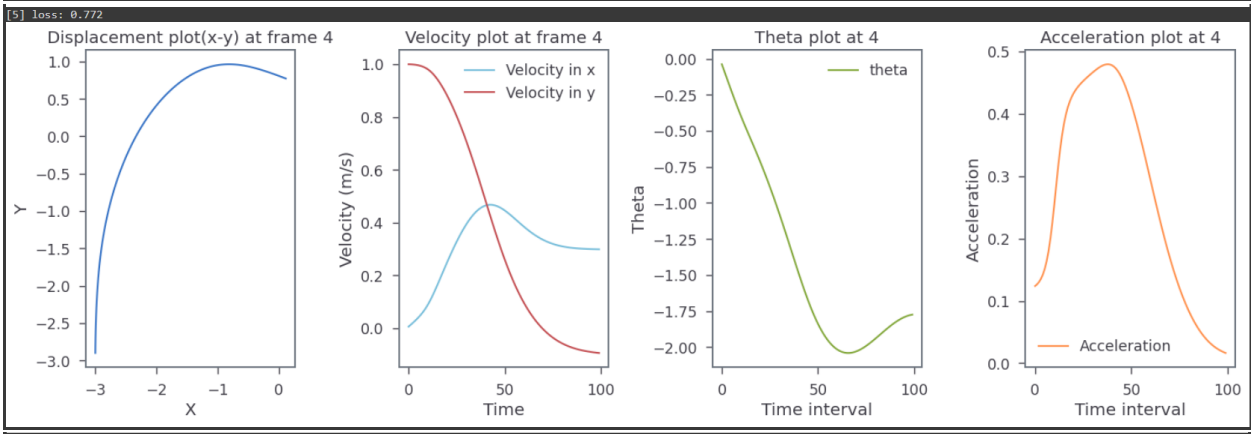
1.1 State Plots



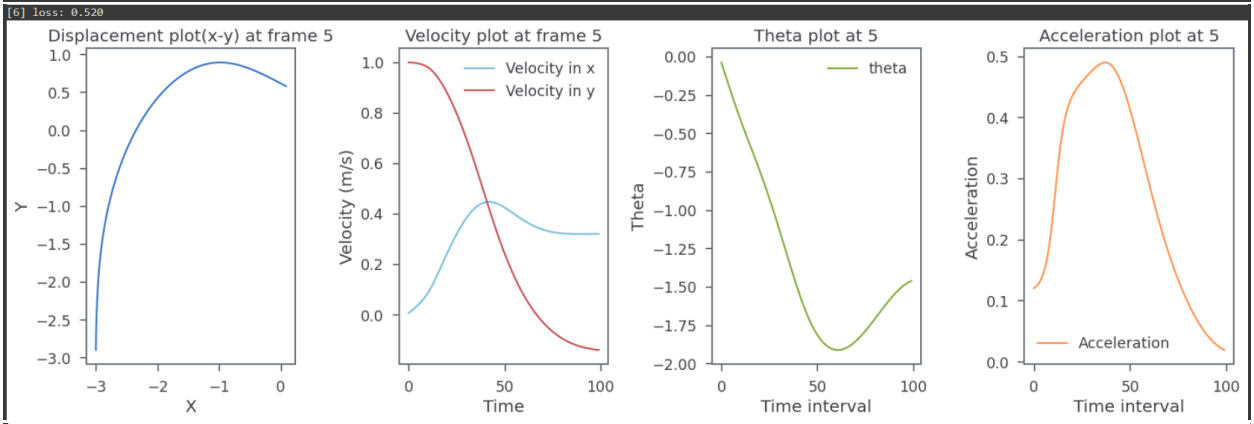
[4] loss: 1.116



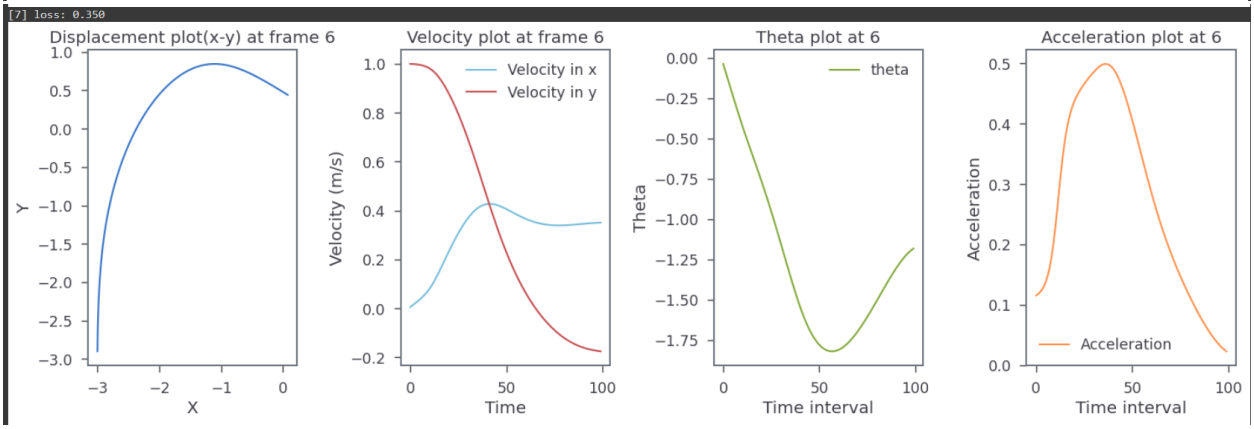
[5] loss: 0.772



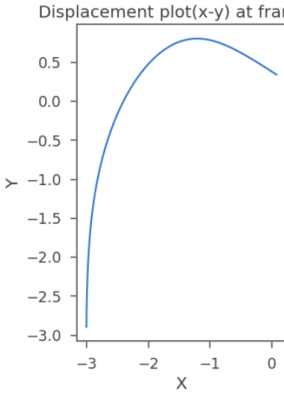
[6] loss: 0.520

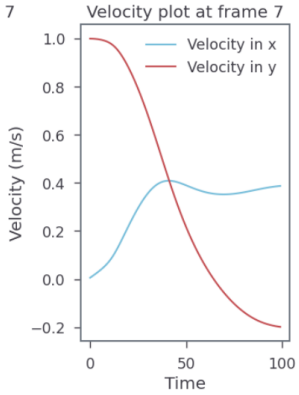


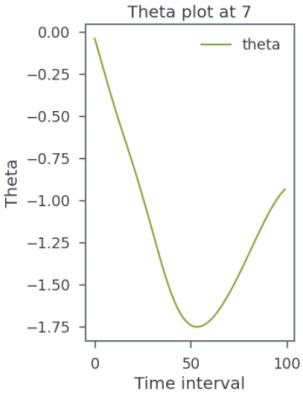
[7] loss: 0.350

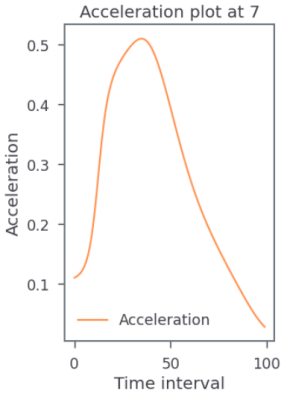


[8] loss: 0.237

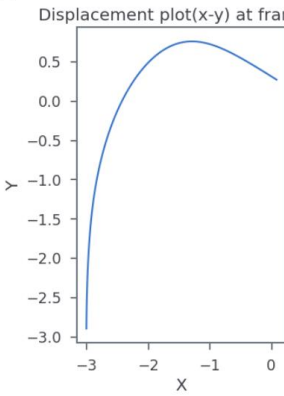


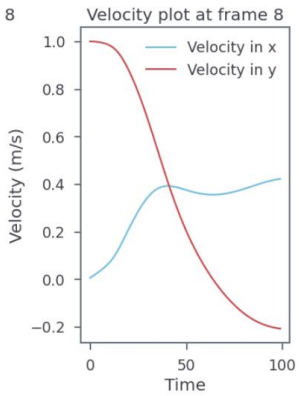


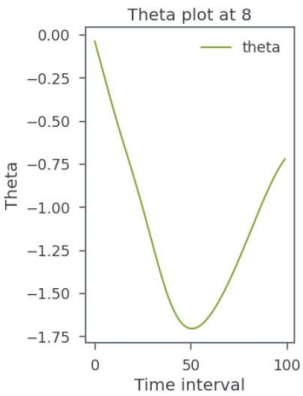


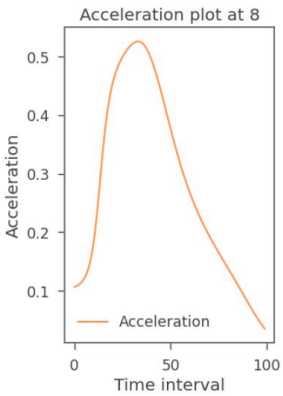


[9] loss: 0.165

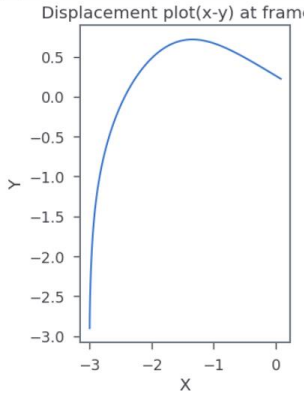


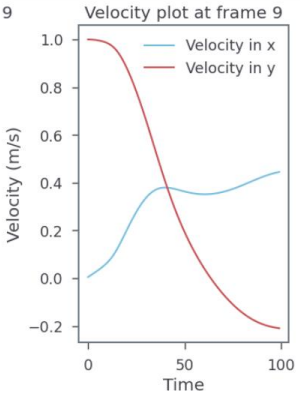


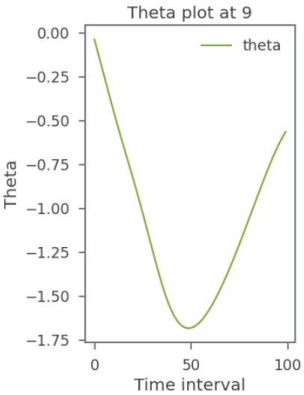


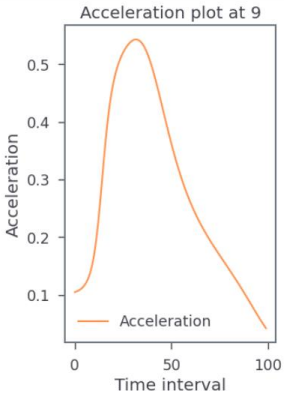


[10] loss: 0.124

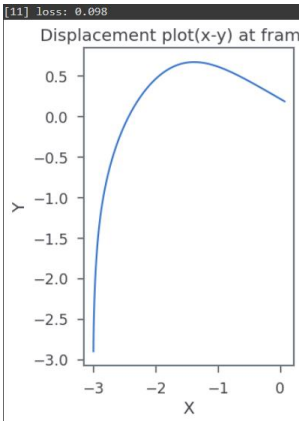


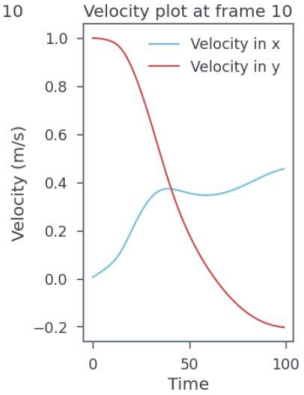


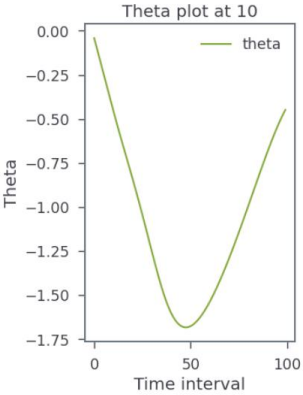


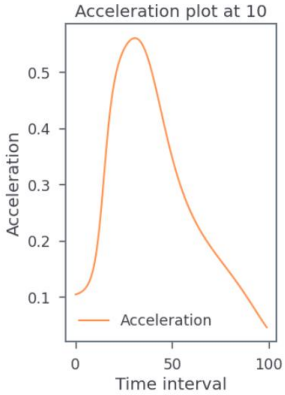


[11] loss: 0.098

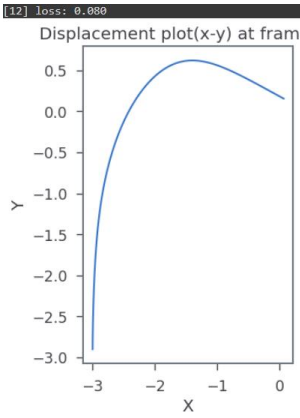


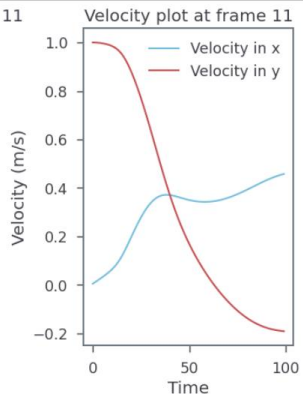


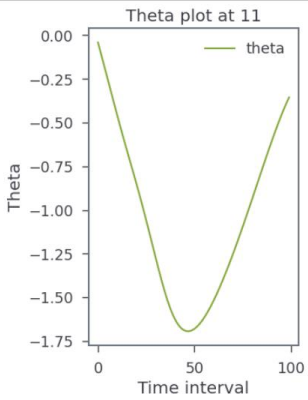


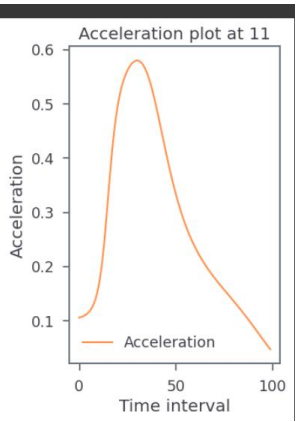


[12] loss: 0.080

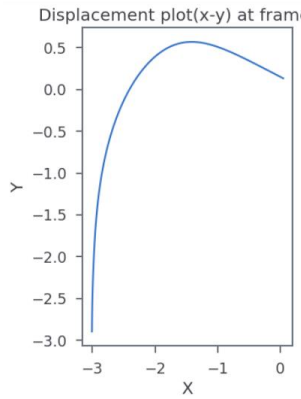


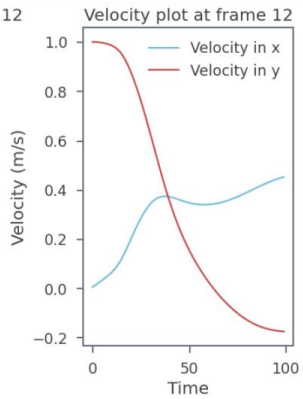


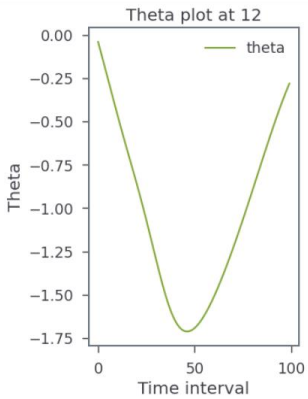


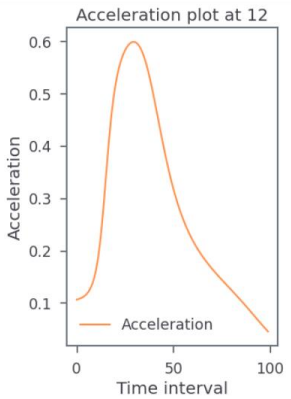


[13] loss: 0.067

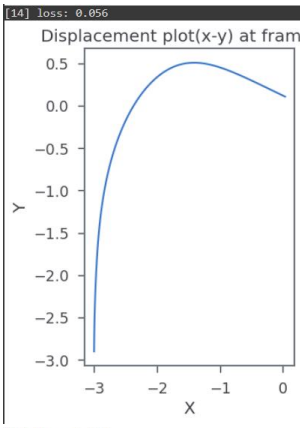


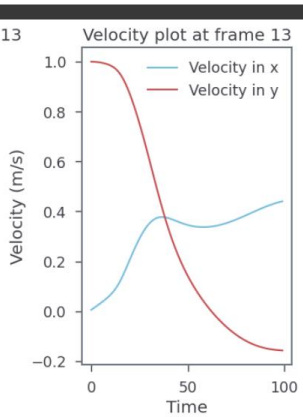


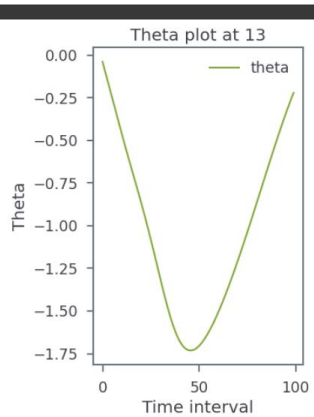


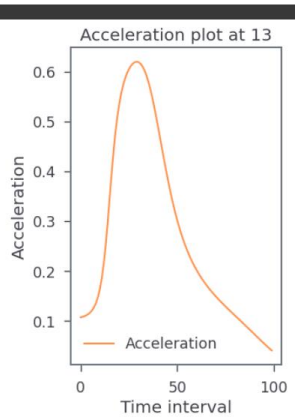


[14] loss: 0.056

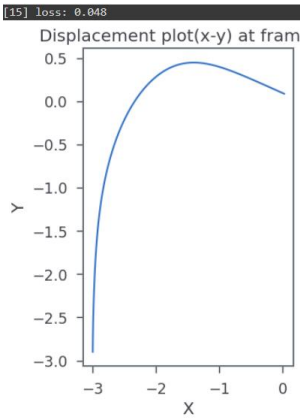


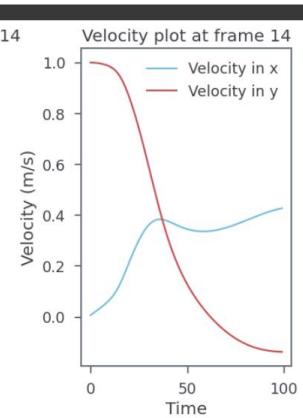


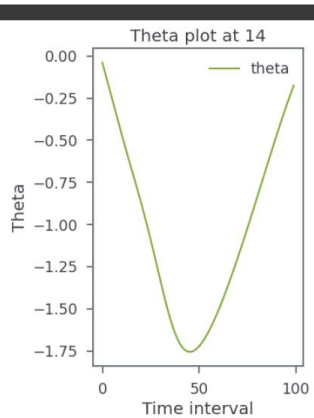


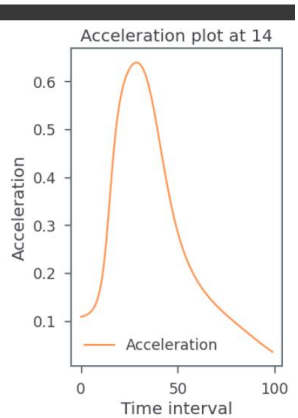


[15] loss: 0.048

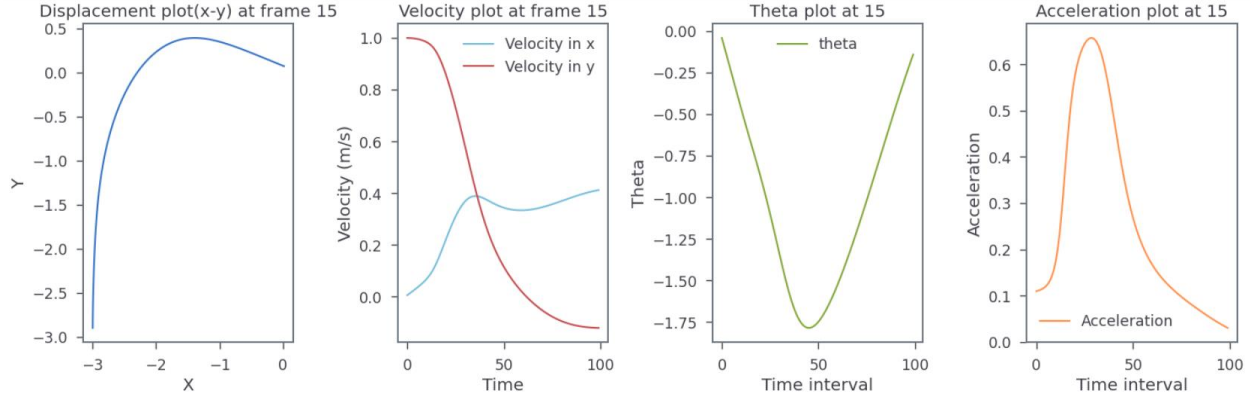




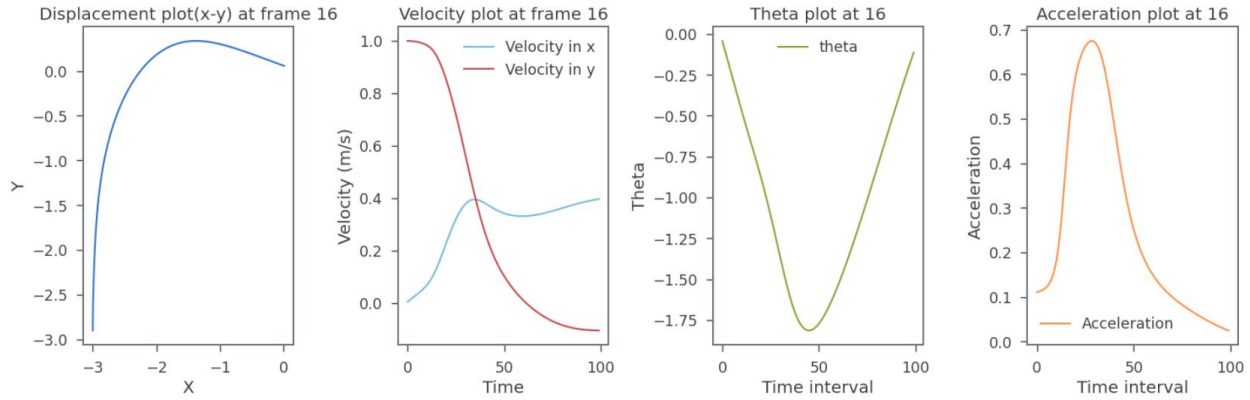




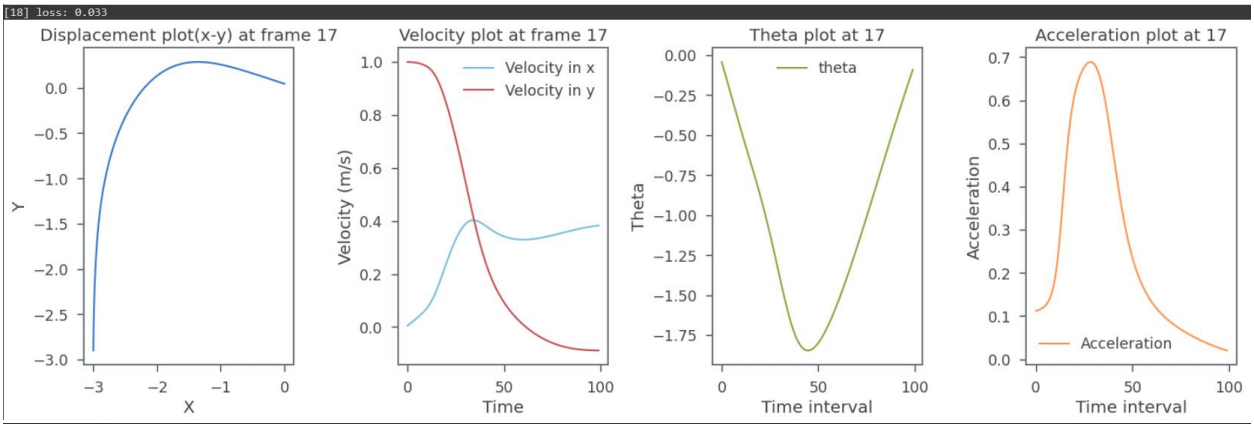
[16] loss: 0.042



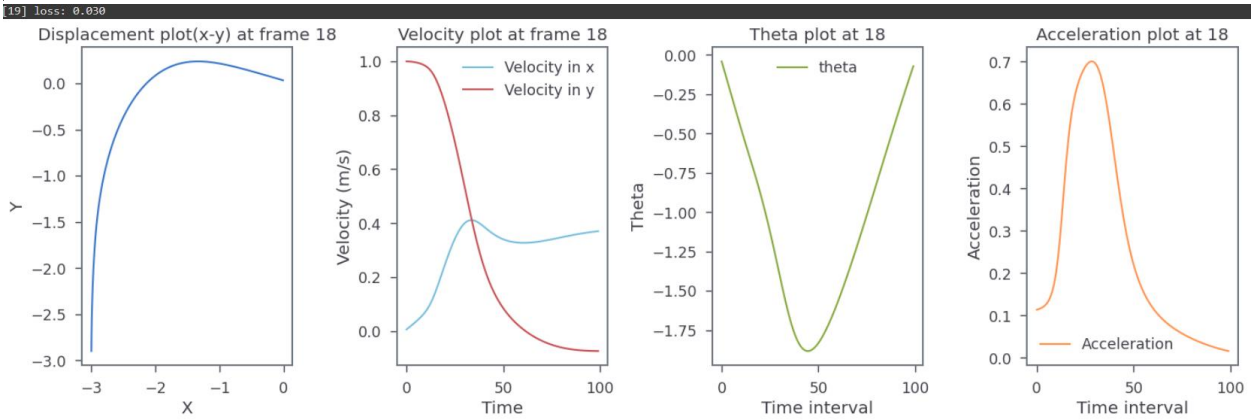
[17] loss: 0.037



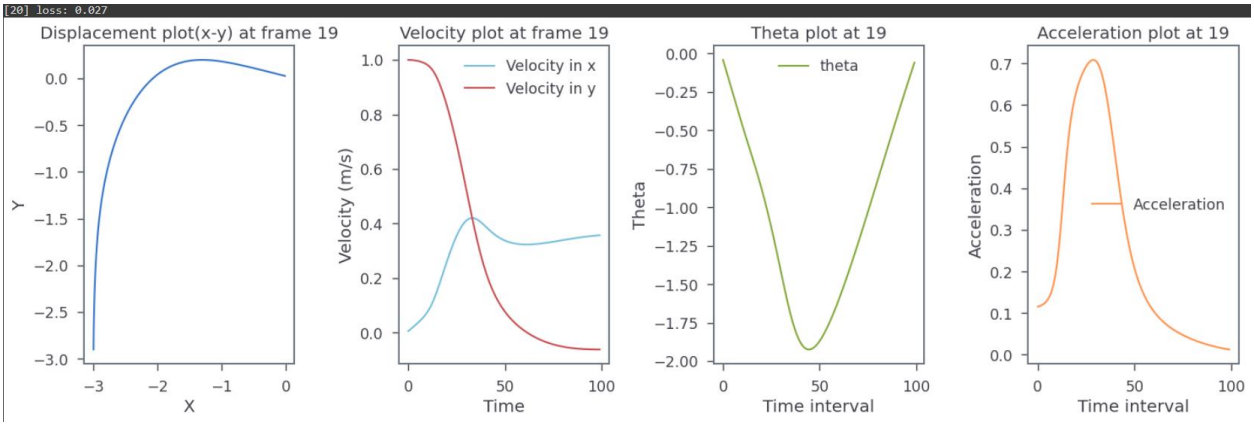
[18] loss: 0.033



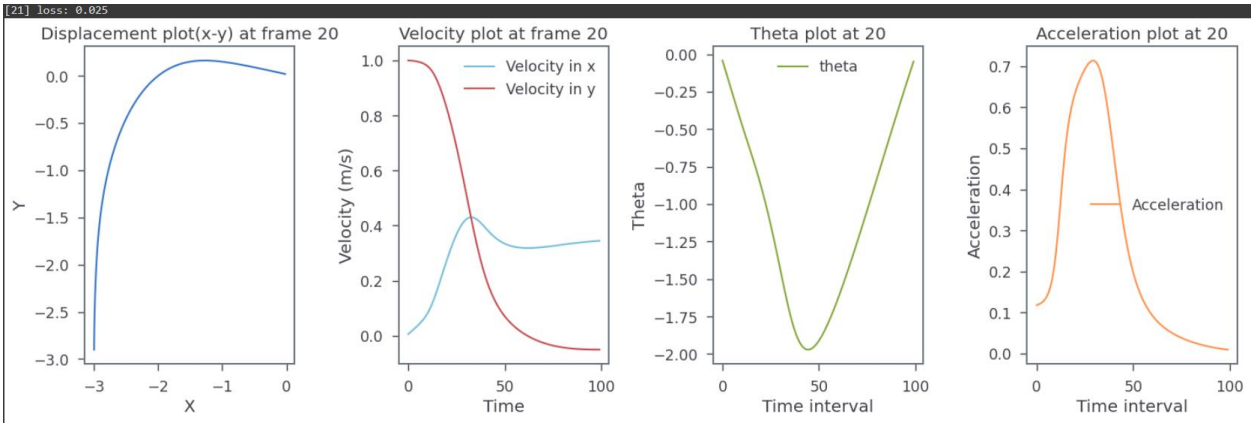
[19] loss: 0.030



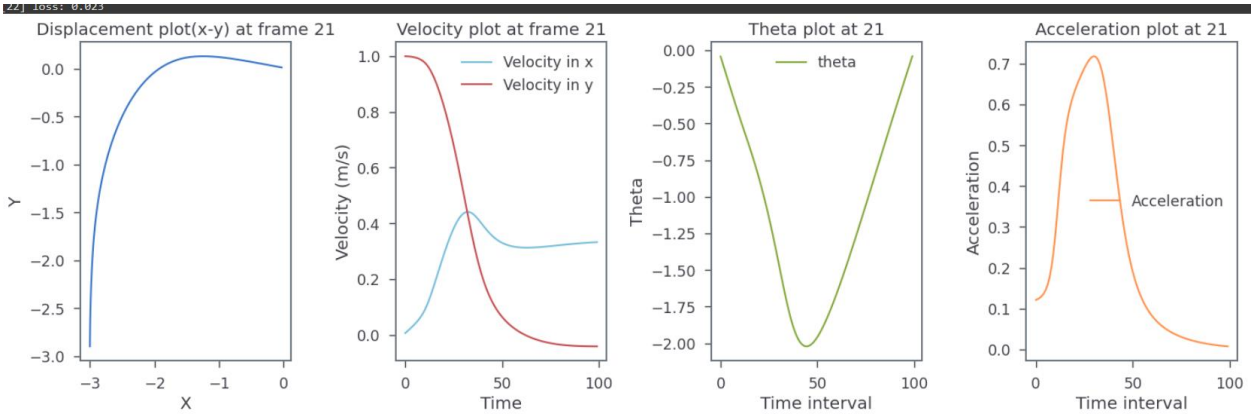
[20] loss: 0.027



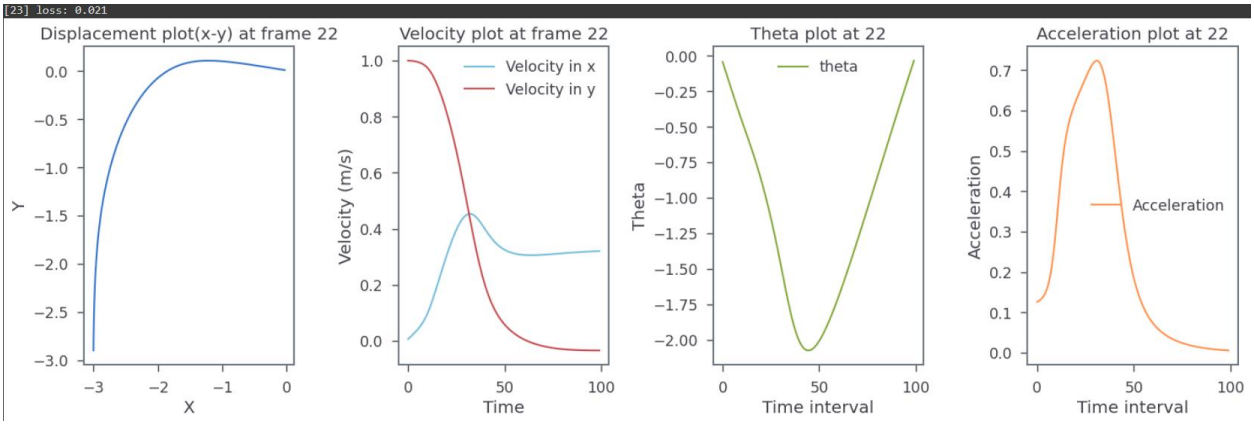
[21] loss: 0.025



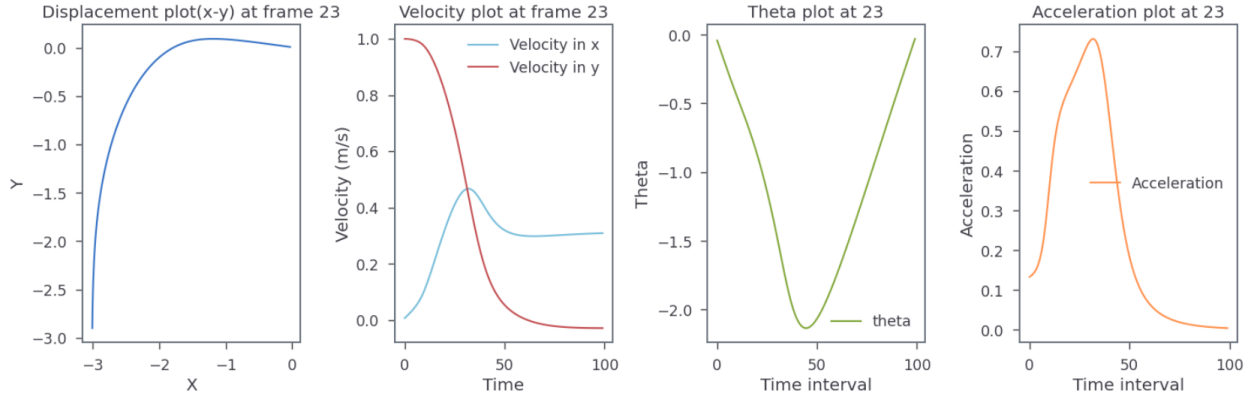
[22] loss: 0.023



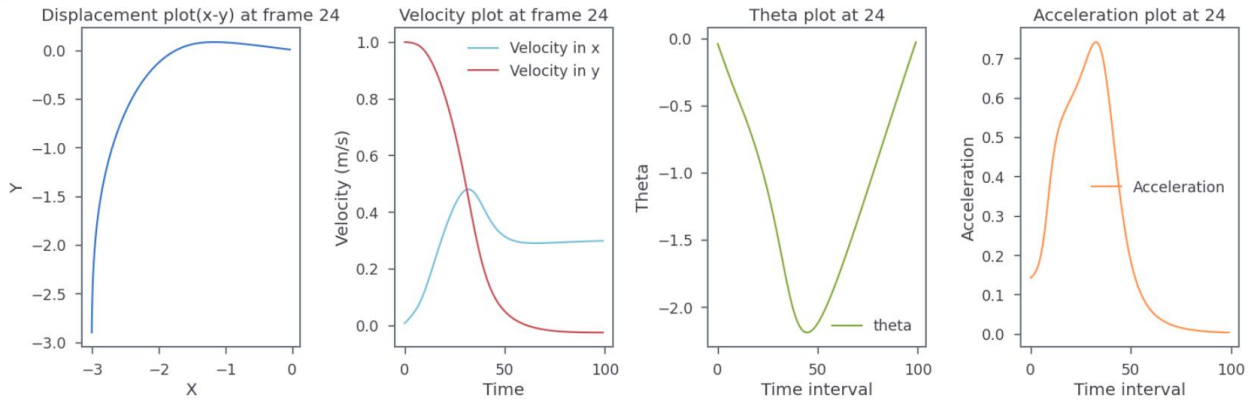
[23] loss: 0.021



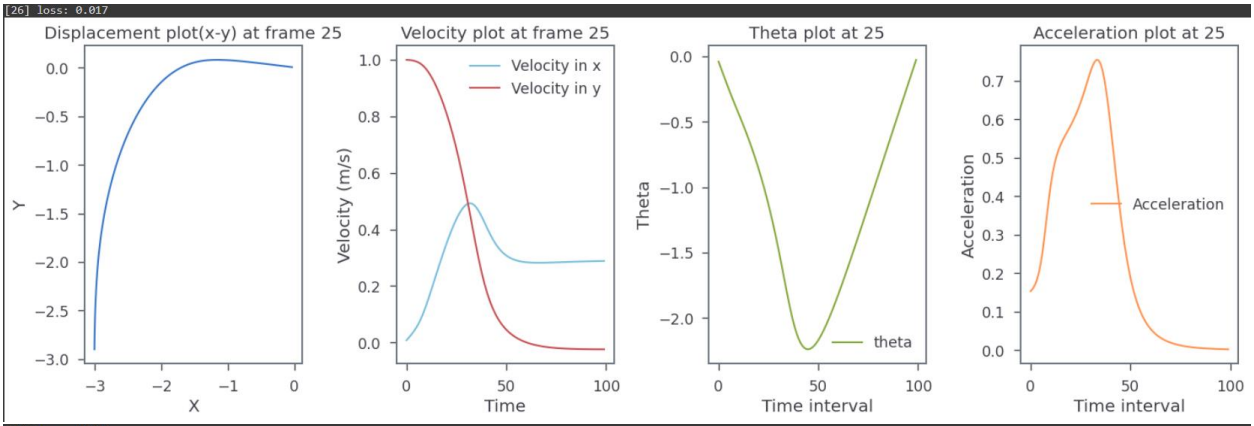
[24] loss: 0.020



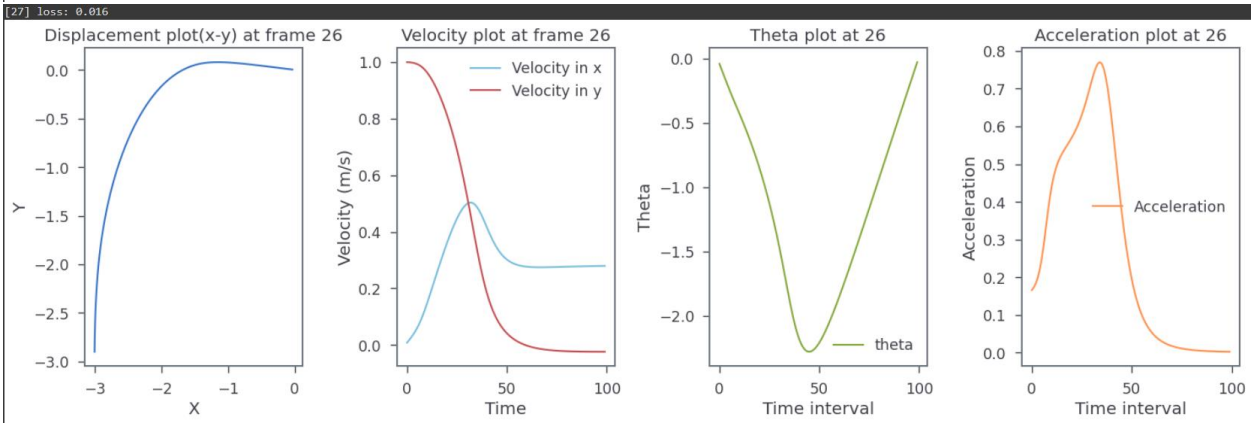
[25] loss: 0.018



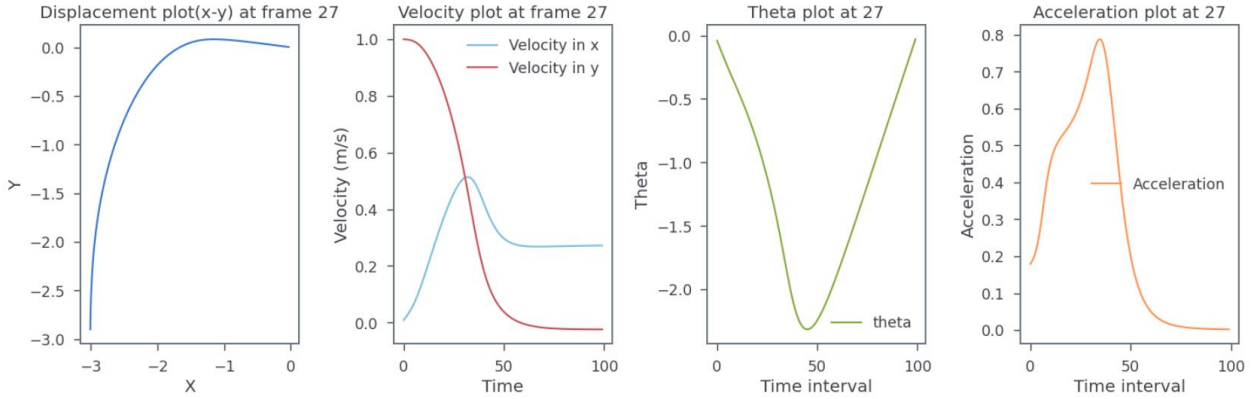
[26] loss: 0.017



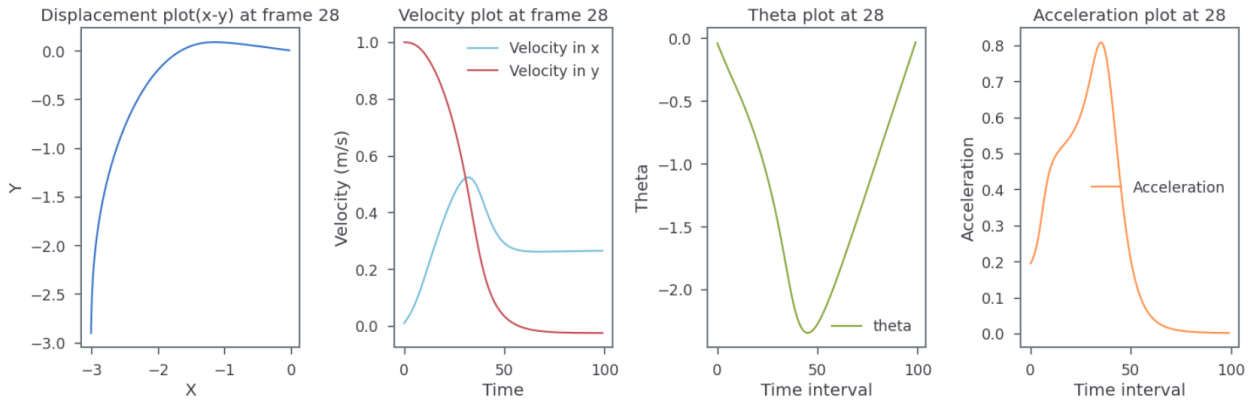
[27] loss: 0.016



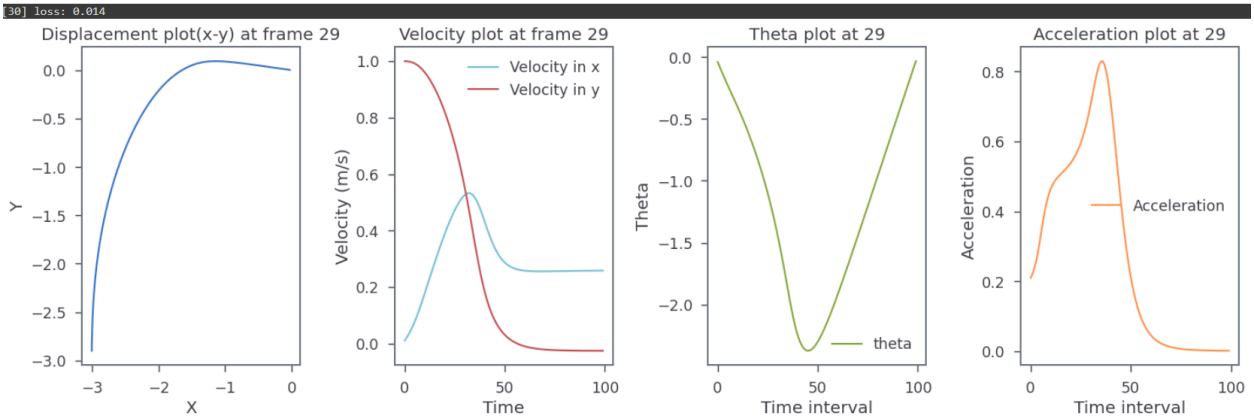
[28] loss: 0.015



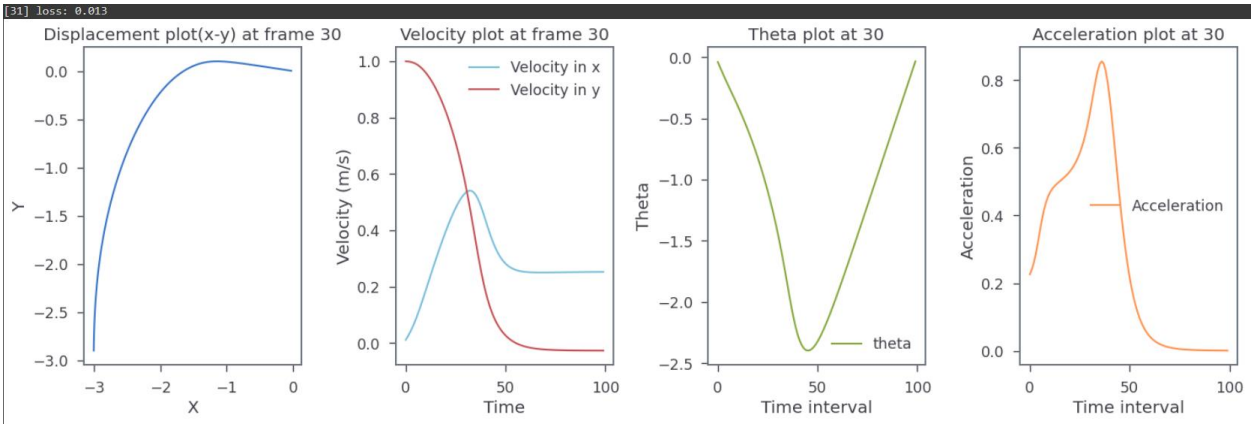
[29] loss: 0.015



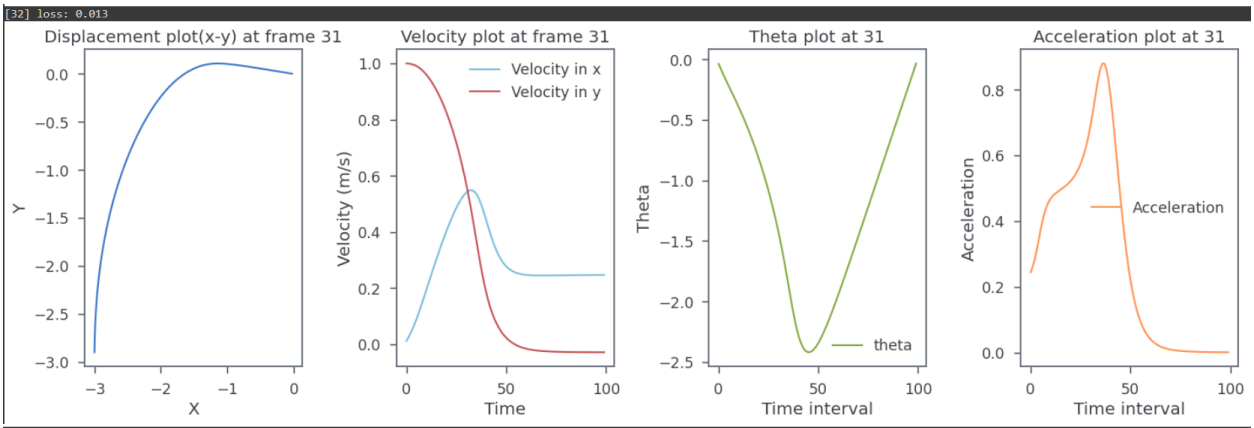
[30] loss: 0.014



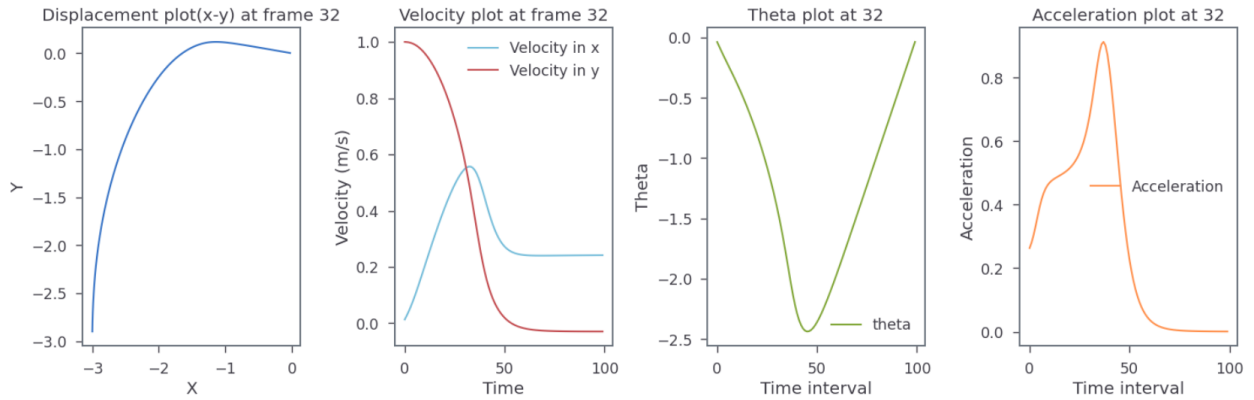
[31] loss: 0.013



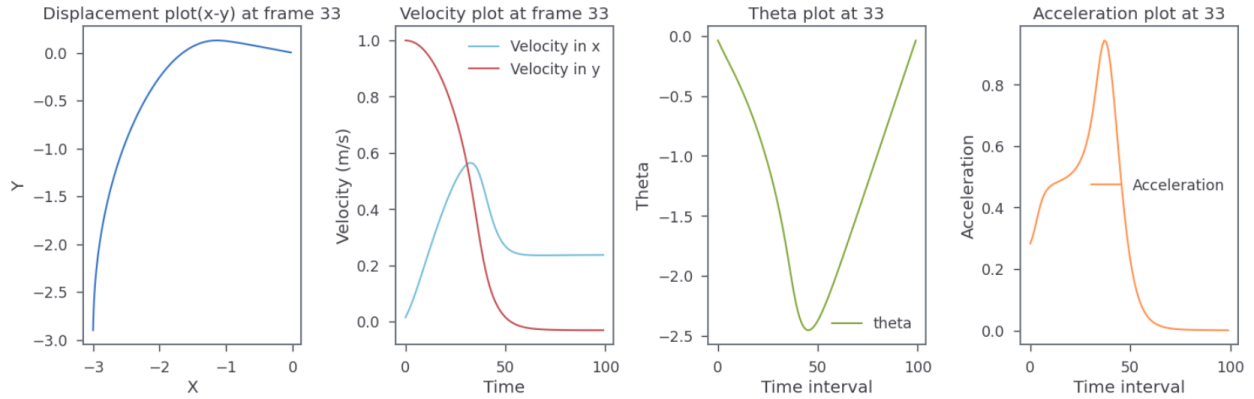
[32] loss: 0.013



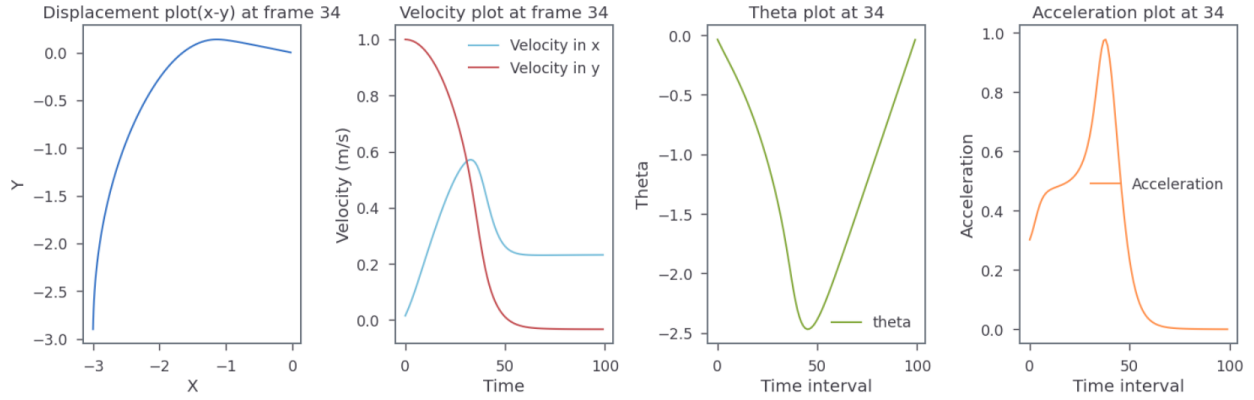
[33] loss: 0.012



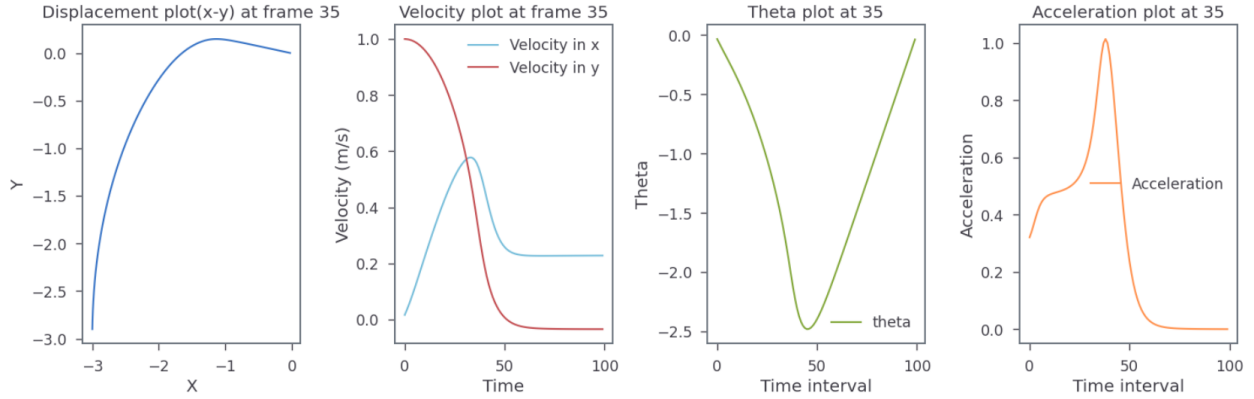
[34] loss: 0.012



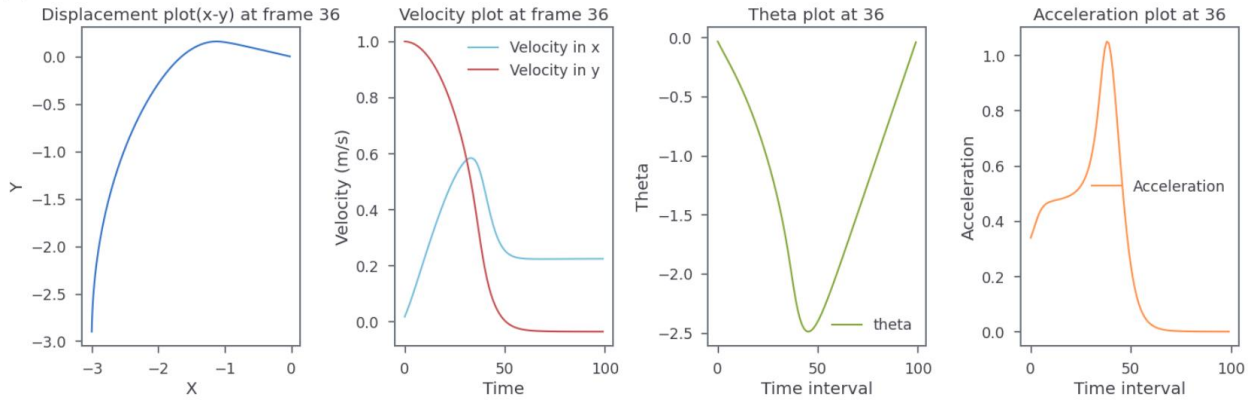
[35] loss: 0.011



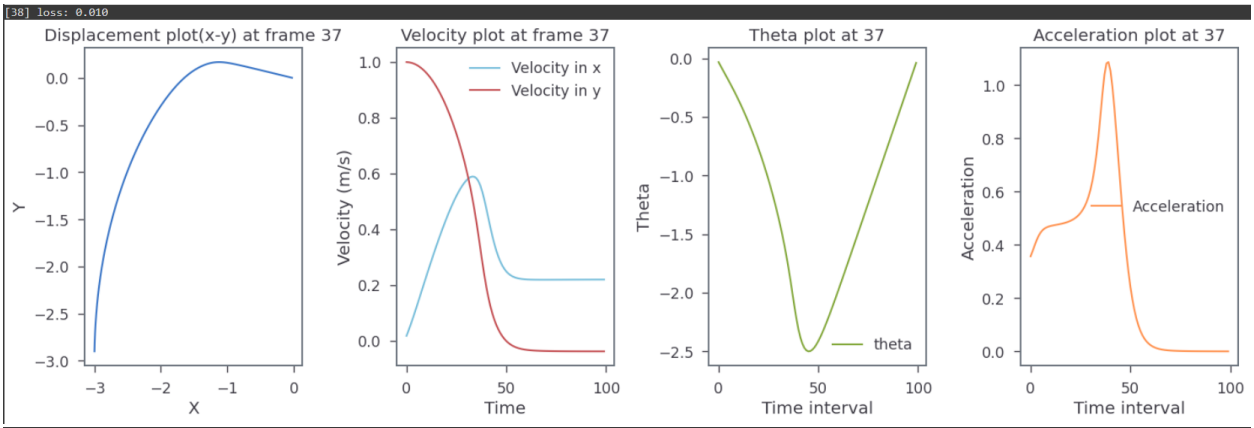
[36] loss: 0.011



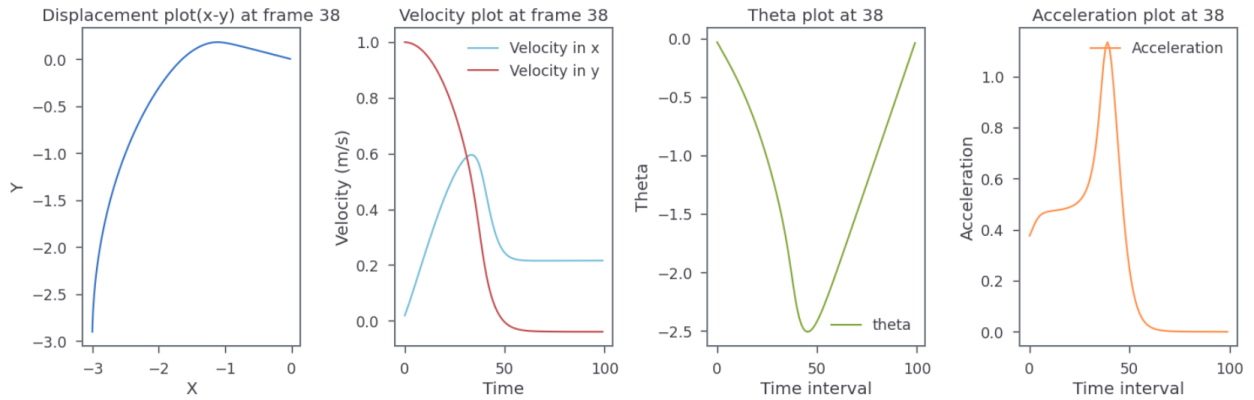
[37] loss: 0.011



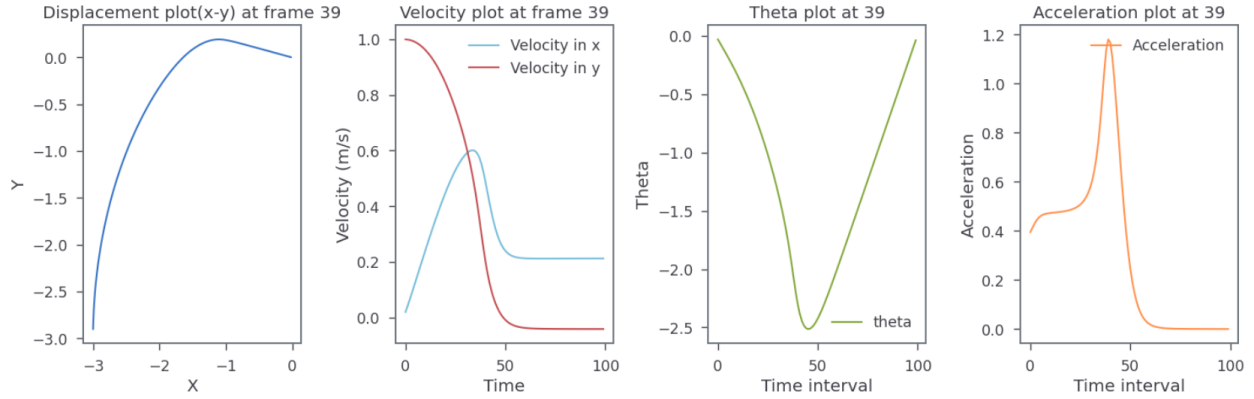
[38] loss: 0.010



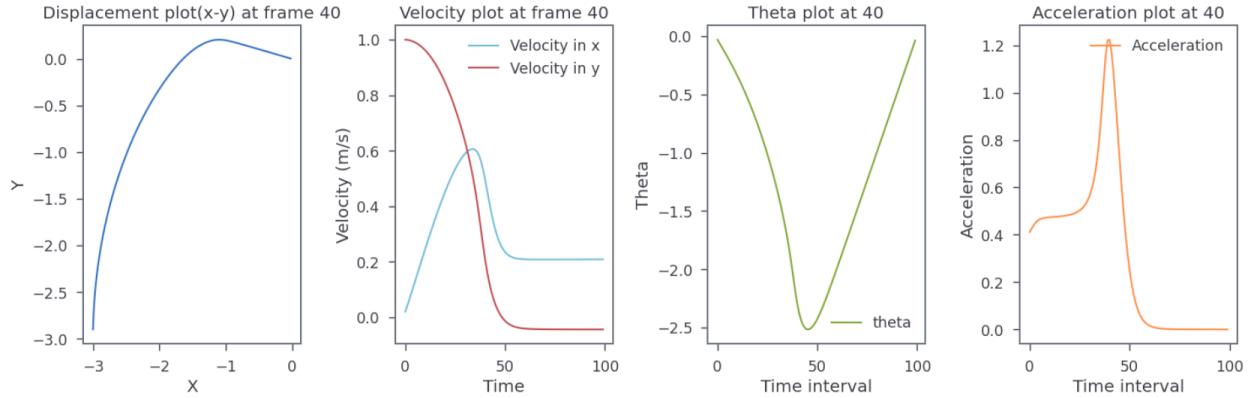
[39] loss: 0.010



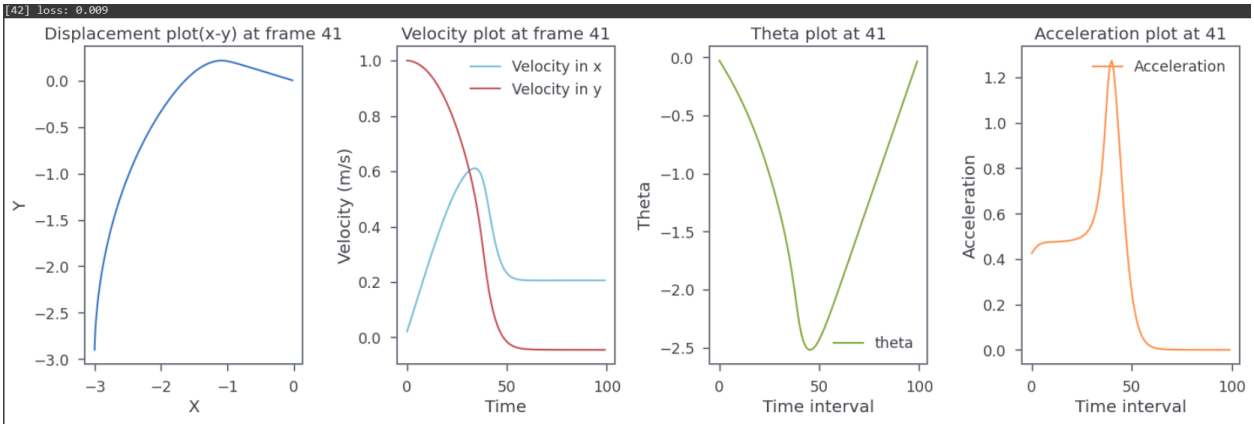
[40] loss: 0.010



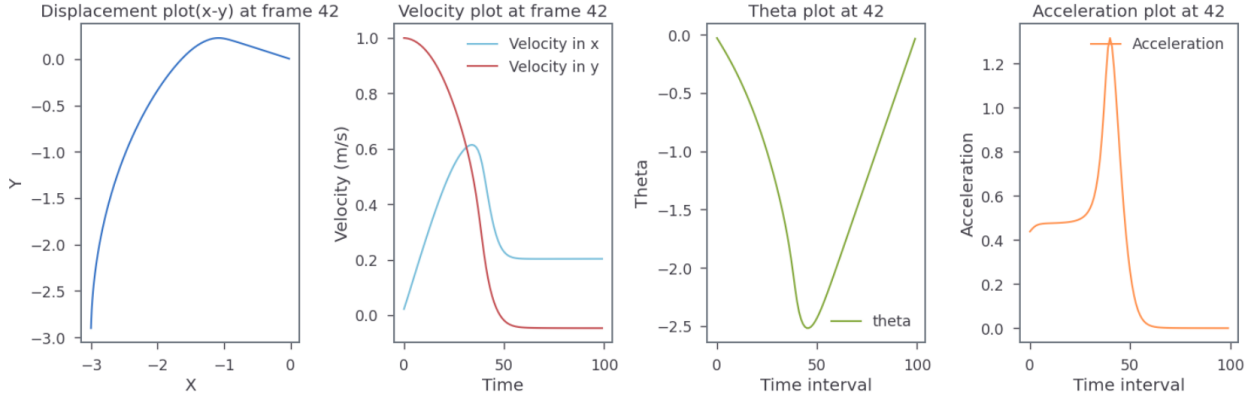
[41] loss: 0.009



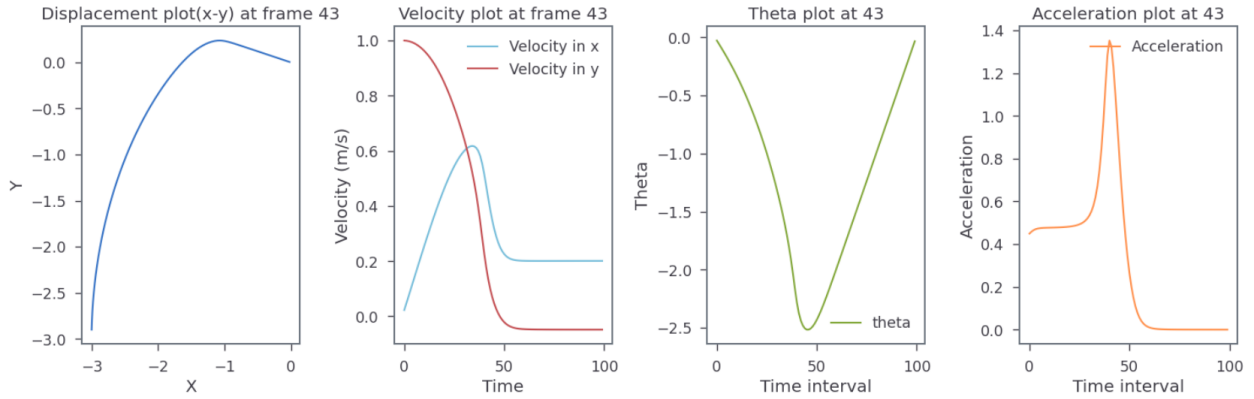
[42] loss: 0.009



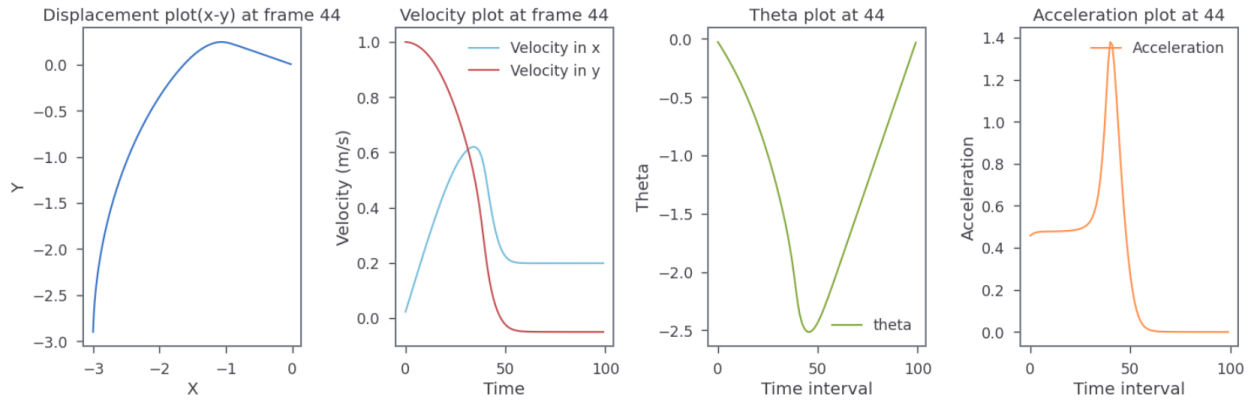
[43] loss: 0.009



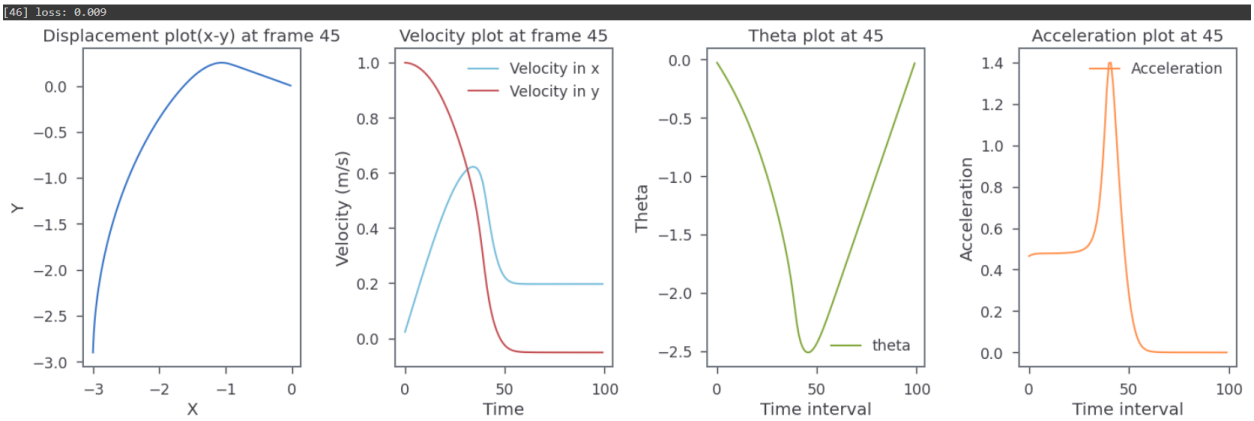
[44] loss: 0.009



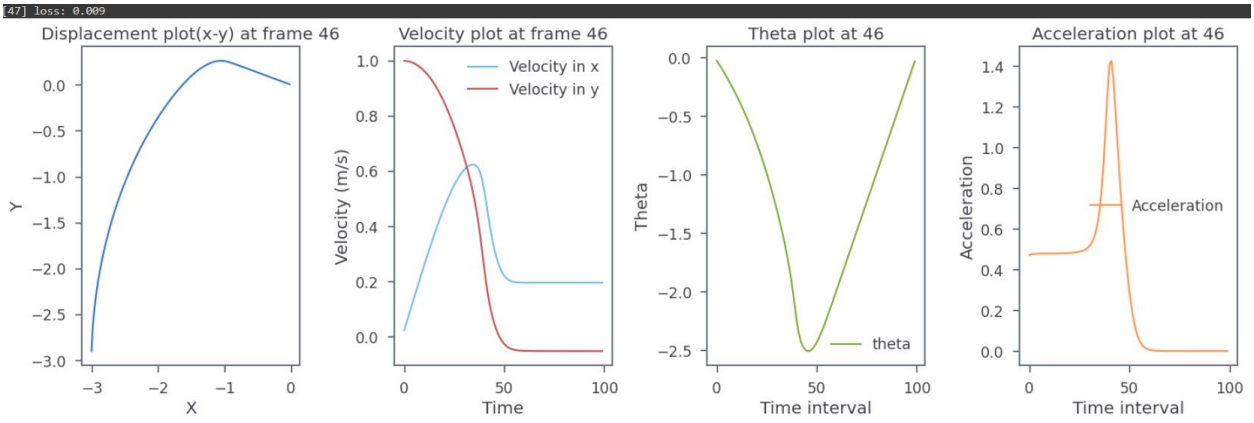
[45] loss: 0.009



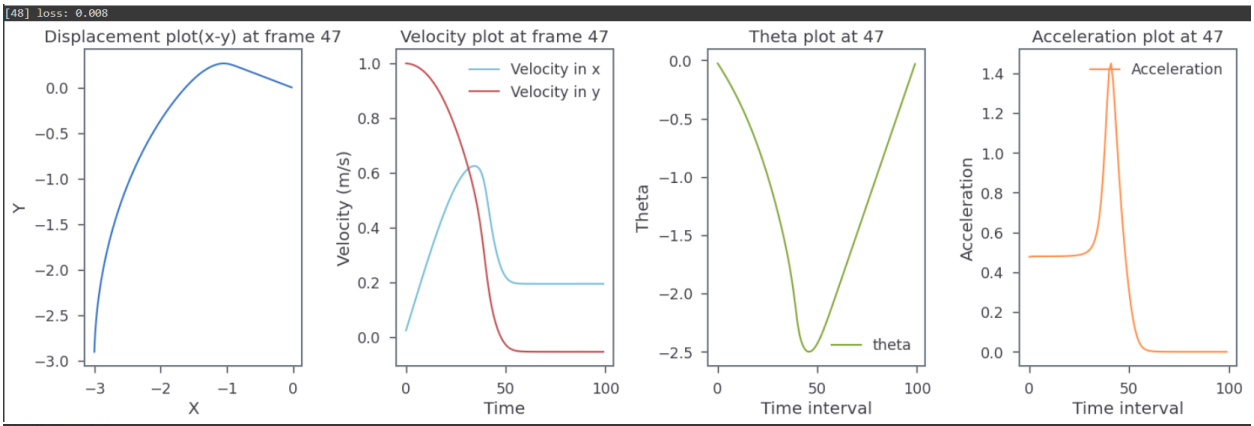
[46] loss: 0.009



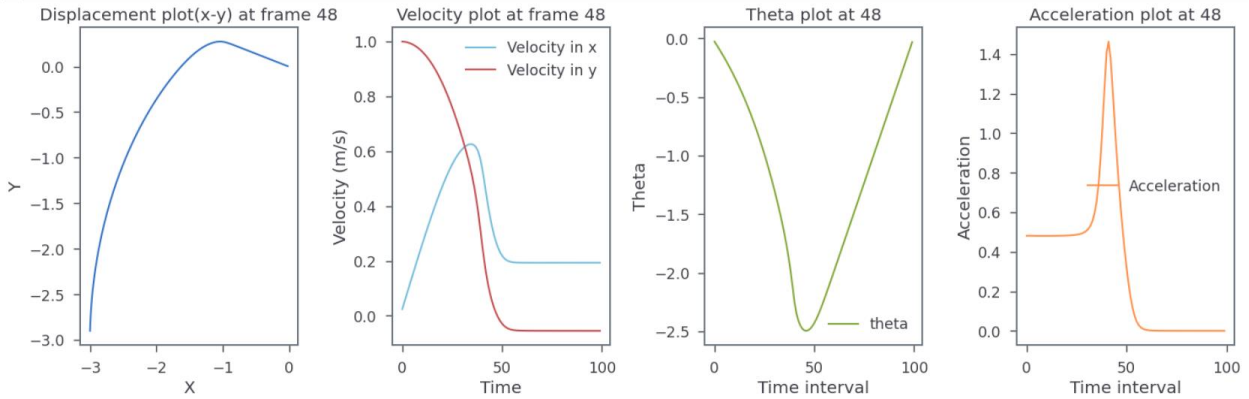
[47] loss: 0.009



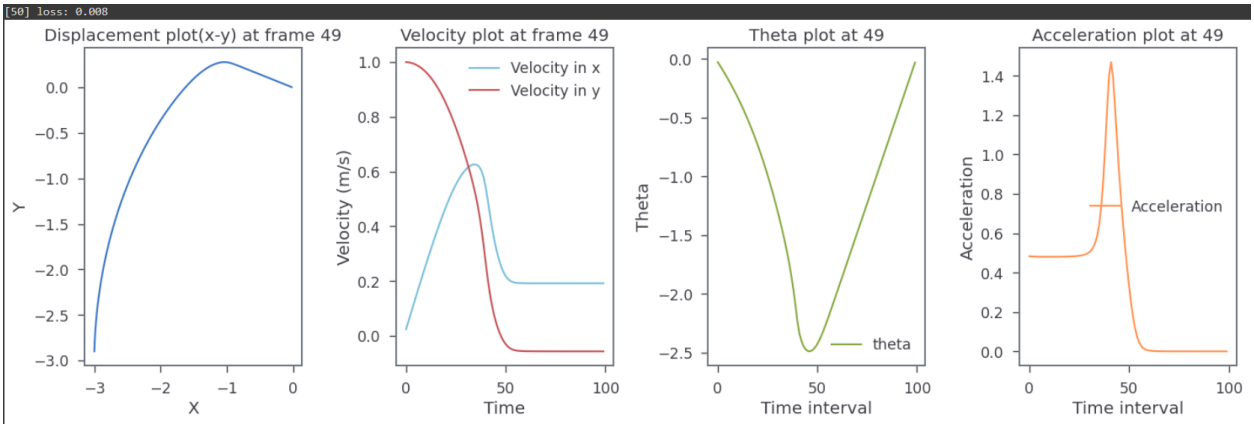
[48] loss: 0.008



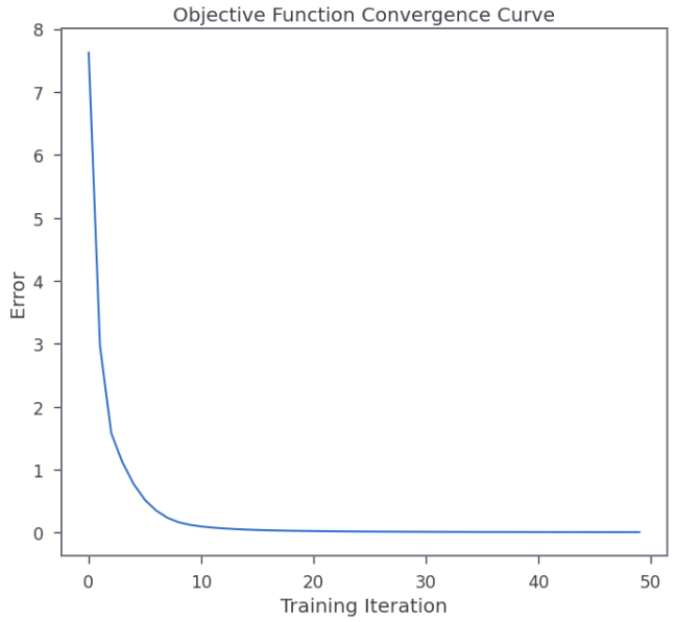
[49] loss: 0.008



[50] loss: 0.008



1.2 Convergence plot



Section 2

2.1 Simulation Path