ARX, RL, and LQR

Marco Conati

May 15, 2024

1 ARX

1.1 Introduction

In this task, we explore the identification of a dynamic system for a piezoelectric nozzle. Given time-domain step response data, we hope to model the system's behavior using ARX (AutoRegressive with eXogenous inputs) models. We are given external knowledge that the true model dynamics are between 5th and 10th order, we undertake an exhaustive search within this range to ascertain the best fitting model.

1.2 Methods

The ARX model is expressed as:

$$A(q)y(t) = B(q)u(t) + e(t)$$

where $y(t)$ is the system output, $u(t)$ is the system input, $e(t)$ is the noise term, and $A(q)$ and $B(q)$ are polynomials of the shift operator q^{-1}, representing the system's dynamics and how past inputs influence the current output. Specifically, $A(q) = 1 + a_1q^{-1} + \cdots + a_{na}q^{-na}$ and $B(q) = b_1q^{-1} + \cdots + b_{nb}q^{-nb}$, with na and nb indicating the orders of the respective polynomials.

The identification process involves estimating the coefficients of these polynomials such that the model output closely matches the observed output. This is achieved through the least squares problem, formulated as:

$$\hat{\theta} = \text{argmin}_\theta \|Y - A * \theta\|^2$$

where $\theta = [a_1, \ldots, a_{na}, b_1, \ldots, b_{nb}]^T$ is the parameter vector, Y is the vector of observed outputs, and A is the regression matrix composed of past output and input values corresponding to the orders na and nb. This problem has solution:

$$\hat{\theta} = (A^TA)^{-1}A^Tb$$

Where the first na elements of θ are the coefficients of a_{est} of $A(q)$, and the next nb elements are coefficients b_{est} of $B(q)$.

The matrices A and b are defined as:

$$A = \begin{bmatrix} -y(N_1-1) & \cdots & -y(N_1-n) & u(N_1) & \cdots & u(N_1-m) \\ \vdots & & \vdots & \vdots & & \vdots \\ -y(N_2-1) & \cdots & -y(N_2-n) & u(N_2) & \cdots & u(N_2-m) \\ \vdots & & \vdots & \vdots & & \vdots \end{bmatrix}$$

$$b = \begin{bmatrix} y(N_1) \\ \vdots \\ y(N_2) \\ \vdots \end{bmatrix}$$

with the amount of rows of A and b determined by the length of the data sample N. To identify the best-fitting model within the 5th to 10th order range, we first generate a random input sample of length N, and pass it through the unknown system to obtain the output y. Then, we iteratively construct ARX models for each combination of na and nb within the range [5, 10], ensuring $na \geq nb$, and evaluate their performance based on the Normalized Root Mean Squared Error (NRMSE) between the predicted and actual outputs. The procedure is summarized in the following pseudocode:

```
for na = 5 to 10
    for nb = 1 to na
        Construct ARX model using orders na and nb
        Estimate model parameters using least squares
        Calculate NRMSE for the estimated model
        If NRMSE < best_fit then
            Update best_fit, best_na, and best_nb
        End If
    end for
end for
Report best_fit, best_na, and best_nb
```

Listing 1: Pseudocode for Model Identification

This systematic search allows us to pinpoint the ARX model that best encapsulates the dynamics of the `piezoNozzle` system as per the provided step-response data, `Vexample.mat`. In addition to this least-squares driven approach, we also estimate the dynamics using Matlab's arx function for comparison.

1.3 Results

The search for the optimal ARX model using my Least Squares implementation concluded with a 10th order model exhibiting a fit of 97.28% across 100 samples of random input and corresponding output data. The best fitting model coefficients for $A(z)$ and $B(z)$ are detailed as follows:

$$A(z) = 1 - 0.4224z^{-1} + 0.04663z^{-2} + 0.04079z^{-3} - 0.137z^{-4} - 0.04906z^{-5} + 0.09359z^{-6} - 0.05533z^{-7} - 0.3532z^{-8} + 0.02261z^{-9} - 0.05159z^{-10}$$

$$B(z) = 0.01236z^{-1} - 0.09425z^{-2} - 0.1666z^{-3} + 0.2424z^{-4} + 0.1158z^{-5} - 0.1463z^{-6} - 0.01962z^{-7} + 0.1386z^{-8} + 0.001422z^{-9} - 0.07413z^{-10}$$

However, the presence of higher-order terms with very small coefficients suggests potential overfitting. We can lower the order of the model to 8th order without a significant impact to the fit. This new model has a fit of 96.85%:

$$A(z) = 1 - 0.9407z^{-1} + 0.6646z^{-2} + 0.03094z^{-3} - 0.39z^{-4} + 0.5882z^{-5} - 0.2344z^{-6} + 0.0227z^{-7} - 0.1296z^{-8}$$

$$B(z) = 0.01084z^{-1} - 0.1011z^{-2} - 0.1136z^{-3} + 0.2912z^{-4} - 0.09531z^{-5} - 0.1213z^{-6} + 0.133z^{-7}$$

Parallelly, employing MATLAB's System Identification Toolbox to construct an ARX model with the same order results in a ver similar model, albeit with a slightly better 97.35% fit:

$$A(z) = 1 - 0.9492z^{-1} + 0.6695z^{-2} + 0.0253z^{-3} - 0.3864z^{-4} + 0.5872z^{-5} - 0.2349z^{-6} + 0.02091z^{-7} - 0.1315z^{-8}$$

$$B(z) = 0.002821 + 0.01021z^{-1} - 0.1011z^{-2} - 0.1134z^{-3} + 0.2932z^{-4} - 0.09814z^{-5} - 0.121z^{-6} + 0.1327z^{-7}$$

The comparative performance of both models against the true system data is graphically represented in Figure 1, demonstrating their capability to closely track the actual data.

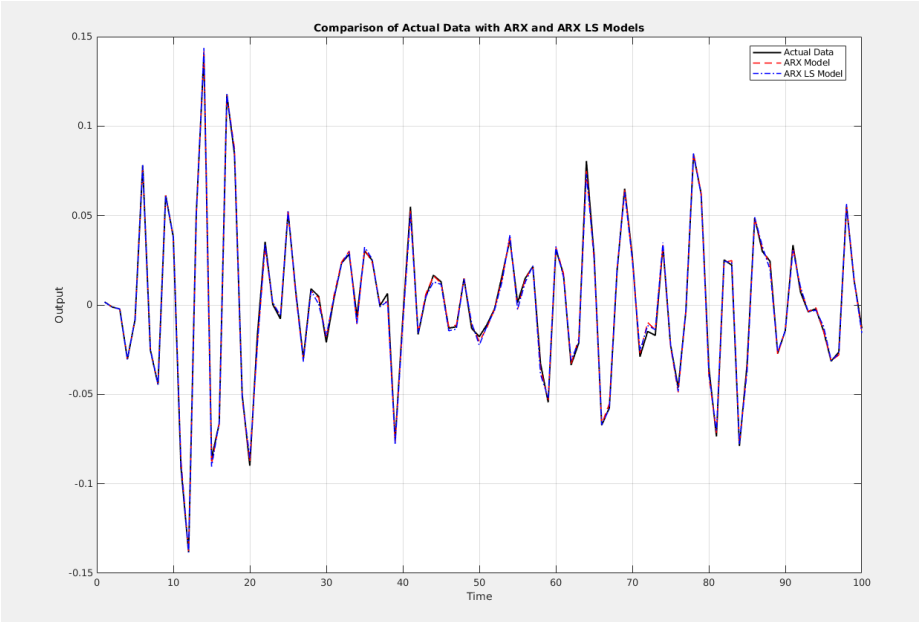

Figure 1: Performance of the Least Squares and MATLAB System Identification Toolbox ARX models against the true system data.

2 Problem 1b

2.1 Introduction and Assumptions

In this part of the study, the objective is to determine the appropriate voltage signal, $V^*(t)$, that should be input into a piezoelectric nozzle system to produce a specific desired pressure wave, P_{ref}, as dictated by data from the file $P_{\text{ref}.mat}$. Note that our ARX model from 1a) is linear, allowing the use of time reversal for approximating the adjoint operator G^* as $\tau G \tau e$. This is an important property for leveraging an Iterative Learning Control (ILC) approach. With this in mind, we can iteratively determine the signal $V^*(t)$ using our ARX model, and then compare the response of the true, hidden system $G(V^*(t))$ to the desired reference P_{ref}.

2.2 Methods

To derive the optimal voltage signal $V^*(t)$ for generating P_{ref}, I employed ILC with time reversal as follows:

1. Initialize V as a constant zero-voltage input and α as 0.1 (chosen heuristically). For iteration in maxIterations:

(a) Compute the current pressure wave P_{current} by inputting V into the arx model for the system.

(b) Calculate the error between P_{current} and P_{ref}.

(c) Reverse the error signal in time using a time-reversal matrix τ.

(d) Simulate the reversed error through the arx system to get P_{reversed}.

(e) Apply a second time reversal to P_{reversed} to estimate the gradient of the error with respect to the voltage signal (adjoint effect). $G^*e = \tau G \tau e$

(f) Update the voltage signal V according to $V_{i+1} = V_i + \alpha * G^*e$.

This process was repeated for 10,000 iterations, adjusting the voltage signal at each step to better match the desired pressure waveform. Note that G refers to the ARX model in this pseudocode.

2.3 Results

The iterative process resulted in a voltage signal $V^*(t)$ that successfully generated a pressure wave $G(V^*(t))$ closely matching P_{ref} within the 10,000 iterations. Importantly, the pressure wave $G(V^*(t))$ was generated by passing the signal $V^*(t)$ created using the ARX model through the hidden, real nozzle system. The reference pressure, achieved pressure, and corresponding input are shown in Figure 2.3.

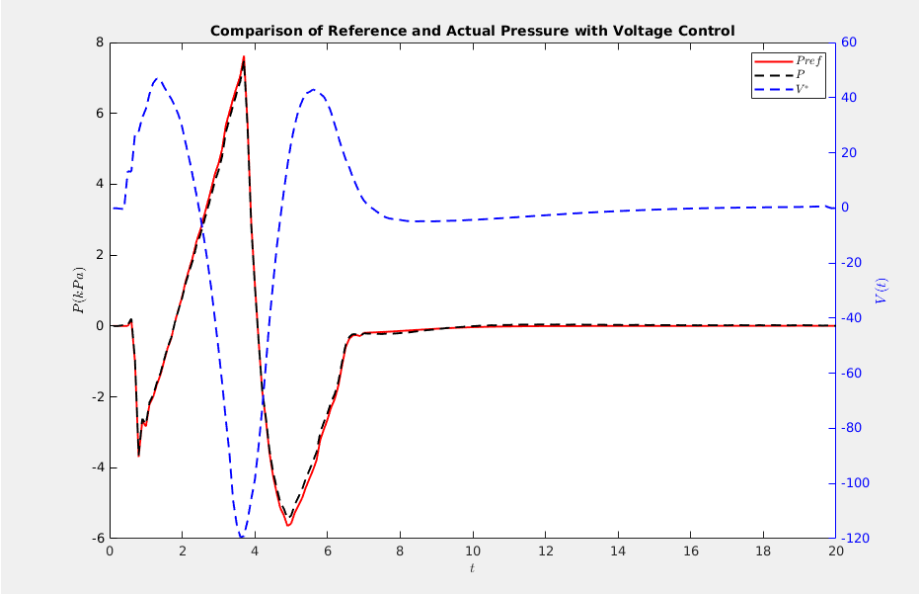

Figure 2: Reference pressure P_{ref}, achieved pressure $G(V^*(t))$, and corresponding input $V^*(t)$

3 Problem 2

3.1 Introduction

The wildlife park manager has tasked us with stabilizing the rabbit population at a constant level of 30 rabbits, despite the natural and predation-related fluctuations involving foxes. The populations are described by the Lotka-Volterra equations:

$$\begin{aligned} \frac{dR}{dt} &= \alpha R - \beta RF, \\ \frac{dF}{dt} &= \delta RF - \gamma F, \end{aligned} \tag{1}$$

where R and F denote the populations of rabbits and foxes, the challenge is to derive a control algorithm based on historical population data. Two methodologies are pursued: a Linear Quadratic Regulator (LQR) for managing the system within a linearized model, and Reinforcement Learning (RL) to iteratively discover an optimal control policy.

3.2 Methods - Parameter Identification

With both approaches, knowledge of the underlying population parameters is necessary. To determine population parameters for wildlife park's rabbit and fox populations, AutoRegressive with eXogenous inputs (ARX) models were fitted to historical data. The ARX model correlates the current output with past inputs and outputs, providing a framework suitable for systems influenced by prior states. In the context of the Lotka-Volterra predator-prey model, the interaction between rabbits and foxes suggests a specific structure for the ARX model's inputs.

Given the structure of the Lotka-Volterra equations:

$$\begin{aligned} \frac{dR}{dt} &= \alpha R - \beta RF, \\ \frac{dF}{dt} &= \delta RF - \gamma F, \end{aligned}$$

we can model each equation separately, treating the derivative terms as our state, and R,F, and RF terms as inputs. As an assumption, I approximated the population derivatives with a first-order difference. With this formulation, we get the following data for the rabbit model:

$$data_{rabbit} = iddata(dR\_dt', [U\_rabbit; U\_rabbit * U\_fox]', dt); \tag{2}$$

and for the fox population change model:

$$data_{fox} = iddata(dF\_dt', [U\_fox; U\_rabbit * U\_fox]', dt); \tag{3}$$

To fit the models, the following orders were set: $na = 0$ indicating no autoregressive component, $nb = [1, 1]$ reflecting the first-order dependence on both

input terms, and $nk = [0, 0]$ implying no delay. The ARX models are then computed using my Least Squares ARX implementation from task 1.

Model validation was performed by comparing the fitted models against the observed data, visually assessed by plotting the actual versus predicted population changes for both rabbits and foxes. The comparison plots are presented in Figures 1 and 2, demonstrating the models' fit to the historical data.

3.3 Results - Model Fitting and Validation

The ARX models fitted to the rabbit and fox populations yielded coefficients that align with the expected structure of the Lotka-Volterra equations. While the models do not perfectly capture the populations, they approximate the system's behavior generally. The initial parameters derived from the ARX models are $\alpha = 1.3$, $\beta = 0.089$, $\delta = 0.279$, and $\gamma = 0.009$, corresponding to the coefficients in the equations:

$$\frac{dR}{dt} = \alpha R - \beta RF,$$
$$\frac{dF}{dt} = \delta RF - \gamma F.$$

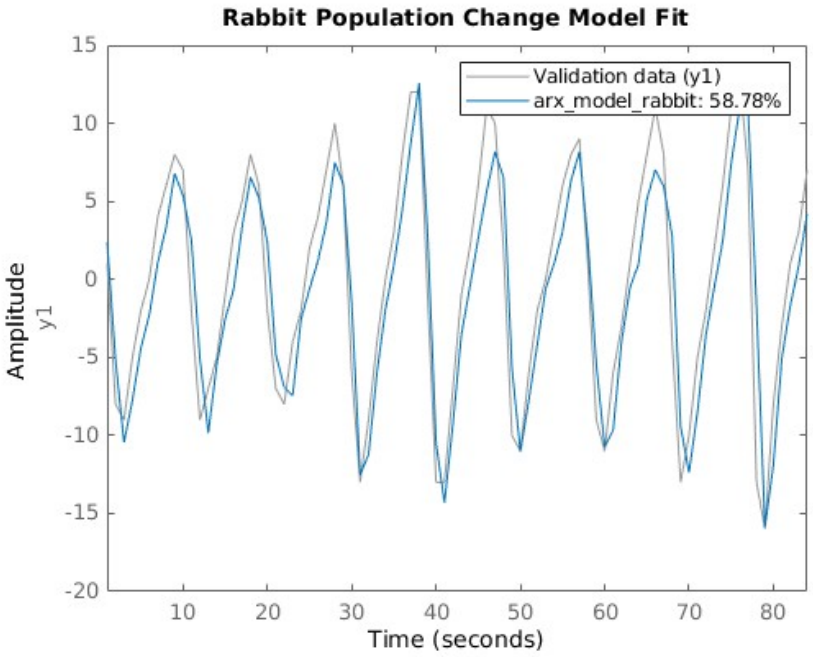

Figure 3: ARX model fit for rabbit population.

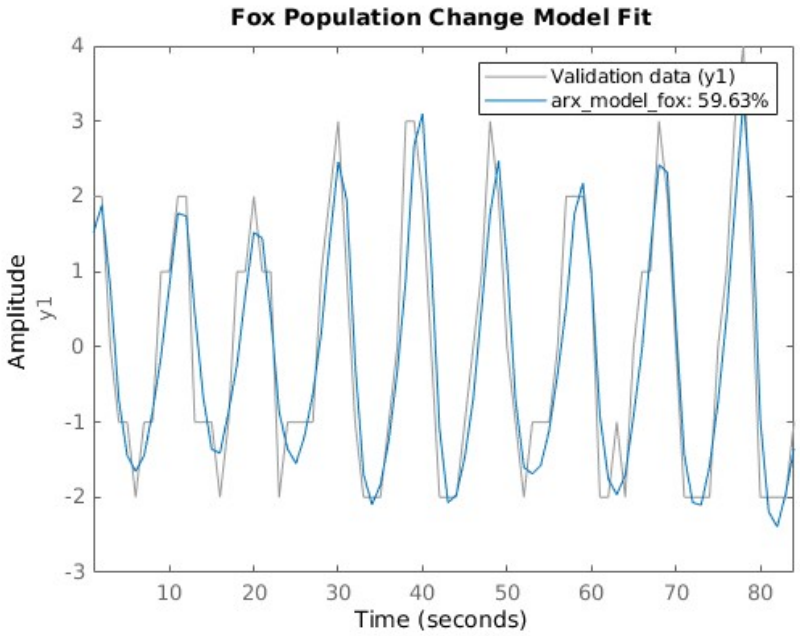

Figure 4: ARX model fit for fox population.

While the models are visually similar, the fits are not the most convincing. With this in mind, I tried refining these initial parameters using MATLAB's `fminsearch` function, which employs a derivative-free optimization algorithm based on the Nelder-Mead simplex method. This numerical optimization iteratively adjusts the parameters to minimize a cost function defined as the sum of squared errors between the observed populations and those predicted by integrating the Lotka-Volterra differential equations with the current parameter predictions:

$$\text{Cost Function} = \sum_{i=1}^{n} (R_{\text{observed},i} - R_{\text{predicted},i})^2 + (F_{\text{observed},i} - F_{\text{predicted},i})^2.$$

The refined model, with parameters $\alpha = 1.4787$, $\beta = 0.0990$, $\delta = 0.3025$, and $\gamma = 0.0103$, resulted in a lower cost function value, indicating a better fit to the data. The figures below showcase the improved fit of this refined model:

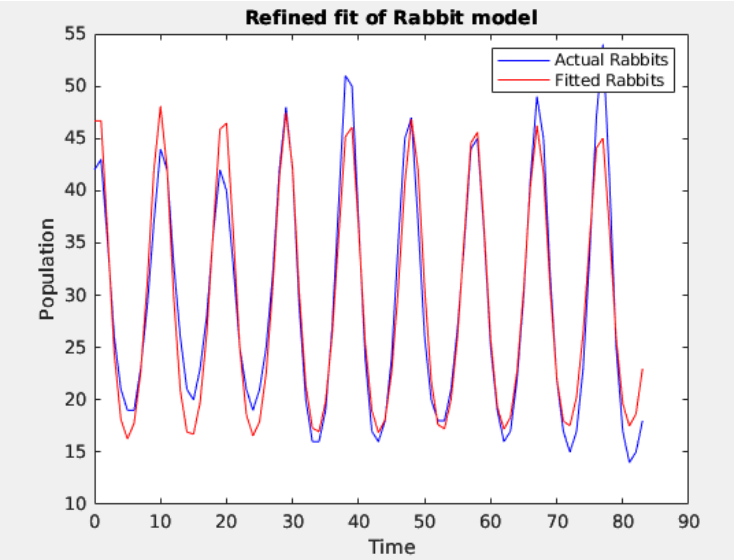

Figure 5: Refined model fit for rabbit population after numerical optimization.

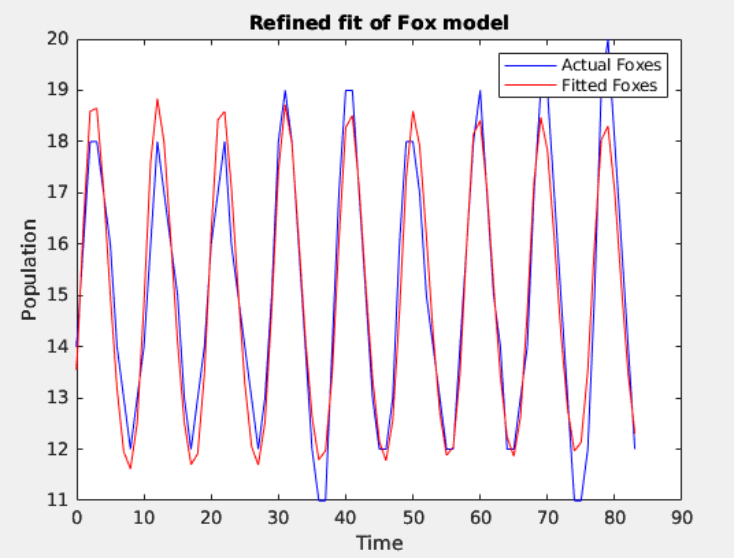

Figure 6: Refined model fit for fox population after numerical optimization.

Recall that the cost function, used by `fminsearch` for optimization, quantifies the aggregate discrepancy between the model's predictions and the actual data. The decrease in cost from 1011.53 to 711.785 confirms the better accuracy of the refined model, so it was used moving forward.

3.4 Controller 1 - Linearization and LQR

For the effective management of the rabbit and fox populations within the wildlife park, a control strategy employing Linear Quadratic Regulator (LQR) was developed. Two key assumptions were made with this algorithm:

1. All population values and changes were rounded up to whole numbers.
2. The nonlinear dynamics are closely approximated with a first-order linearization.

The linearization process involves calculating the Jacobian matrix, which represents a first-order approximation to the system's dynamics around a specified operating point. The Jacobian for the Lotka-Volterra system is given by:

$$J = \begin{bmatrix} \frac{\partial dR}{\partial R} & \frac{\partial dR}{\partial F} \\ \frac{\partial dF}{\partial R} & \frac{\partial dF}{\partial F} \end{bmatrix} \tag{4}$$

I set the desired state for the regulator at 30 rabbits and 15 foxes, as this is the target rabbit population with the closest whole number of foxes conducive to a balanced population.

Pseudocode for the LQR approach is provided below:

```
Set Lotka-Volterra parameters alpha, beta, delta, gamma
Define the equations dRdt and dFdt using Lotka-Volterra dynamics
Compute the Jacobian J of the system

Set the weight matrices Q for state and R_mat for control input
Q = [10 0; 0 1]; %Prioritize rabbit state
R_mat = 1;

for each simulation do
    Initialize state with random whole numbers for R0 and F0
    for each time step do
        Compute the state matrix A using JacobianFunc at current
    state
        Compute the LQR gain K using lqr(A, B, Q, R_mat)
        Calculate the control input using the gain K and the state
    error
        Apply control input, rounding up to whole numbers
        Update the state using the Lotka-Volterra dynamics and dt
        Round the updated state to whole numbers
    end for
end for
```

Listing 2: Pseudocode for LQR Control Strategy

This pseudocode captures the essence of applying LQR control to a linearized system, with a focus on the continuous adjustment of the operating point and application of control inputs derived from LQR gains. Note that Q and R were set to emphasize the importance of maintaining the rabbit state, as this term has the most weight. With this formulation, the control inputs are designed to drive the state towards the desired population levels, while accounting for the discrete nature of the wildlife populations.

3.5 Controller 2 - RL

My project for MECHENG599 involves Double Deep Q-Networks (DDQN) for a different task, so I leveraged my implementation as an alternative approach to managing the wildlife park populations. DDQN extends the traditional Q-learning in two ways.

1. A traditional Q-Table is replaced with a NN for approximating values
2. Two separate NNs are used: a primary network for selecting the action and a target network for evaluation (this is the distinction between DQN and DDQN). This maintains more stability in the target, resulting in more stable training overall.

Mathematically, this is expressed with a new Bellman equation:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q'(s_{t+1}, a) - Q(s_t, a_t) \right], \tag{5}$$

where Q and Q' denote the primary and target networks, respectively, and γ is the discount factor.

My agent was designed as a 3-layer Multi-Layer Perceptron (MLP) with hidden layer sizes [64,128,64], which has access to the system state at every time step, facilitating the understanding and prediction of the population dynamics.

For the reinforcement learning to converge to an optimal policy, an appropriate reward function was crucial. The final reward function implemented was:

$$\begin{aligned} \text{rabbit\_reward} &= -0.3 \times |\text{rabbits} - 30| \\ \text{fox\_reward} &= -0.1 \times |\text{foxes} - 15| \\ \text{reward} &= 0.75 \times (\text{rabbit\_reward} + \text{fox\_reward}) \\ \text{if } |\text{rabbits} - 30| &< 2 : \text{reward} = 10 \end{aligned}$$

Termination conditions were set such that the simulation ends if the population of either species hits zero, exceeds the maximum allowed(1000), or if the maximum number of steps is reached. Moreover, a significant penalty was imposed for extinction or overpopulation events:

Penalty for Extinction or Overpopulation:

- If the rabbits' population is either zero or reaches the maximum population, or if the foxes' population reaches the maximum population:

$$\text{reward} \mathrel{-}= 1000 - (\text{current\_step} \times 25)$$

This reward structure aims to avoid rapid extinction or overgrowth while densely encouraging the maintenance of the target populations. Coefficients were determined heuristically, with a few driving ideas:

1. The only positive reward should be achieved with a desirable outcome (In this case maintaining population within 2).

2. Maintaining rabbit population is more important than fox, in this case 3x as important

3. Surviving for one step without causing a termination condition should always be more beneficial than causing a termination condition (extinction or explosion).

The RL environment, along with the reward function, was set up using Python's `gym` and `torch` libraries, followed by the training of the agent to navigate the simulation towards the desired steady states of rabbit and fox populations.

3.6 Results - LQR and RL

The evaluation of the LQR controller involved initializing trajectories from randomized starting points within the ranges of [10-100] for both rabbits and foxes. The LQR methoid successfully guided the rabbit population towards the vicinity of the desired count of 30. While a small steady-state error persisted, the convergence of all populations remained within a margin of 2 rabbits, as depicted in Figures 7 and 8.

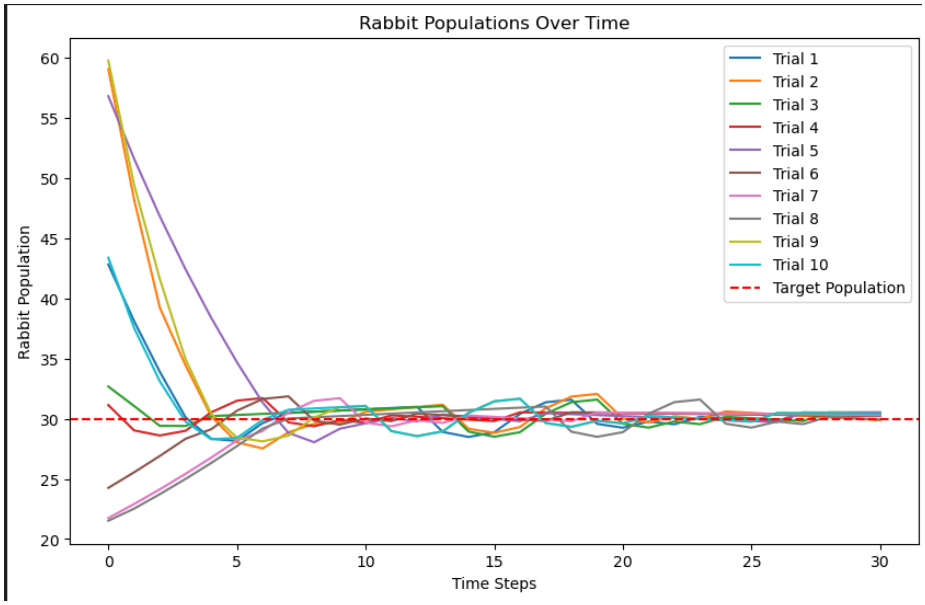

Figure 7: Rabbit population trajectories over time under LQR control.

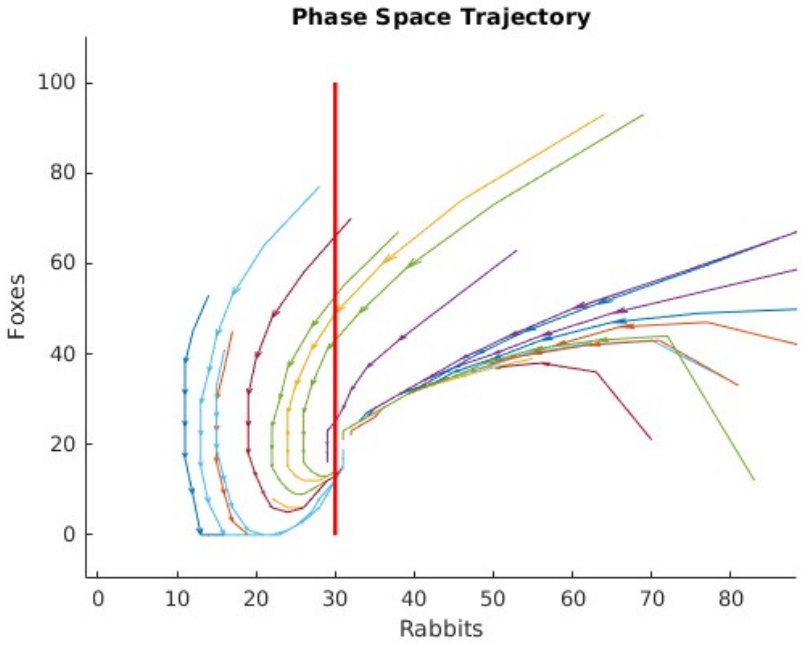

Figure 8: Population evolution in phase space with LQR control.

The RL-based controller also managed to achieve very similar convergence characteristics. Qualitatively, RL approach also appeared to be more stable at extreme starting locations, such as cases with 1000s of rabbits and few foxes, but numerically evaluating this goes beyond the scope of this task. The RL controller's performance with controlling the rabbit population is showcased in Figure 9, demonstrating its competent performance over a broad range of starting points.

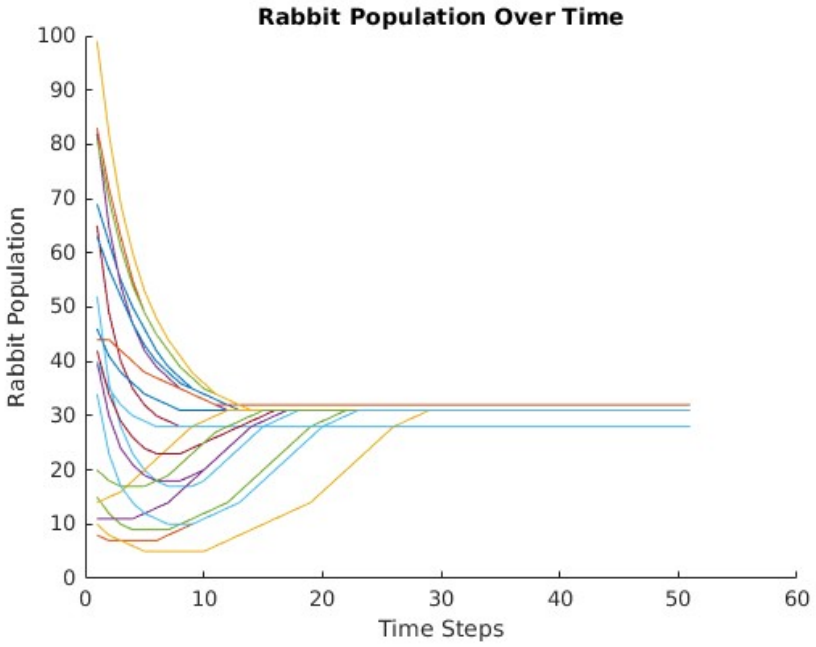

Figure 9: Rabbit population trajectories over time with RL control.

Overall, both control methodologies proved effective for randomized starting configurations within the [10-100] range for rabbits and foxes. With this in mind, both LQR and RL are viable solutions for addressing the wildlife park manager's challenge.

Appendix

Python Code

```
import torch
import torch.nn as nn
import torch.optim as optim
import os
import numpy as np
import random
from collections import deque
from torch.nn.utils import clip_grad_norm_

# Check if CUDA is available and set the device accordingly
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ")

class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_layers):
```

```
        super(QNetwork, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(len(hidden_layers)):
            self.layers.append(nn.Linear(input_dim if i == 0 else hidden_layers[i-1], hidden_layers[i]))
            self.layers.append(nn.ReLU())
        self.layers.append(nn.Linear(hidden_layers[-1], output_dim))

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

class RLAgent:
    def __init__(self, input_dim, action_dim, hidden_layers, lr=1e-4, gamma=0.99, buffer_size=10000, batch_size=32,
                 epsilon_start=1.0, epsilon_end=0.05, epsilon_decay=0.995, target_update_frequency=10,
                 learning_starts=1000, train_freq=4, max_grad_norm=10):
        #print(device)
        self.q_network = QNetwork(input_dim, action_dim, hidden_layers).to(device)
        self.target_q_network = QNetwork(input_dim, action_dim, hidden_layers).to(device)
        self.target_q_network.load_state_dict(self.q_network.state_dict())
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=lr)
        self.gamma = gamma
        self.replay_buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.action_dim = action_dim
        self.loss_fn = nn.MSELoss()
        self.epsilon = epsilon_start
        self.epsilon_end = epsilon_end
        self.epsilon_decay = epsilon_decay
        self.target_update_frequency = target_update_frequency
        self.learning_starts = learning_starts
        self.train_freq = train_freq
        self.max_grad_norm = max_grad_norm

    def update_replay_buffer(self, transition):
        self.replay_buffer.append(transition)

    def train(self):
        if len(self.replay_buffer) < self.batch_size:
            return
        transitions = random.sample(self.replay_buffer, self.batch_size)
        state, action, next_state, reward, done = zip(*transitions)
        state = torch.FloatTensor(np.array(state)).to(device)
        next_state = torch.FloatTensor(np.array(next_state)).to(device)
        action = torch.LongTensor(action).squeeze().to(device)
        reward = torch.FloatTensor(reward).to(device)
```

```
        done = torch.FloatTensor(done).to(device)

        current_q_values = self.q_network(state).gather(1, action.unsqueeze(1)).squeeze(1)
        ## Basic DQN
        next_q_values = self.target_q_network(next_state).max(1)[0]
        expected_q_values = reward + self.gamma * next_q_values * (1 - done)
        ##Double DQN
        selected_actions = self.q_network(next_state).max(1)[1].unsqueeze(1)
        # Evaluate the selected action Q-value using the target network
        next_q_values = self.target_q_network(next_state).gather(1, selected_actions).squeeze(1)
        expected_q_values = reward + self.gamma * next_q_values.detach() * (1 - done)

        loss = self.loss_fn(current_q_values, expected_q_values.detach())
        self.optimizer.zero_grad()
        loss.backward()

        # Gradient clipping
        clip_grad_norm_(self.q_network.parameters(), self.max_grad_norm, norm_type=2)

        self.optimizer.step()
        return loss

    def train_agent(self, env, num_episodes=1000, save_model_iters=None, save_model_folder='/models', progress_bar = None, verbose=False):
        learning_starts = self.learning_starts
        train_freq = self.train_freq
        epsilon = self.epsilon
        total_steps = 0
        rewards = []
        moving_averages = []  # Store moving averages

        os.makedirs(save_model_folder, exist_ok=True)

        for episode in range(num_episodes):
            state = env.reset()
            total_reward = 0
            steps = 0
            if progress_bar is not None:
                progress_bar.update(1)
            while True:
                total_steps += 1
                action = self.select_action(state, epsilon)
                next_state, reward, done, _ = env.step(action)
                total_reward += reward

                self.update_replay_buffer((state, action, next_state, reward, done))

```

```
                if total_steps > learning_starts and total_steps % train_freq == 0:
                    loss = self.train()

                state = next_state
                steps += 1

                if done:
                    break

            rewards.append(total_reward)

            # Calculate moving average of the last 100 rewards
            if episode >= 99:
                moving_avg = sum(rewards[-100:]) / 100
                moving_averages.append(moving_avg)
            else:
                # If less than 100 episodes, calculate the average of what we have
                moving_avg = sum(rewards) / len(rewards)
                moving_averages.append(moving_avg)

            epsilon = max(self.epsilon_end, epsilon * self.epsilon_decay)
            if episode % self.target_update_frequency == 0:
                self.update_target_network()

            if save_model_iters is not None and episode % save_model_iters == 0:
                self.save_model(episode, save_model_folder, verbose)
                print(f"Episode: {episode}, Total Reward: {total_reward}, Steps: {steps}, Epsilon: {epsilon}, Moving Avg Reward: {moving_avg}")
            if verbose:
                try:
                    print(f"Loss: {loss.item()}, Episode: {episode}, Total Reward: {total_reward}, Steps: {steps}, Epsilon: {epsilon}, Moving Avg Reward: {moving_avg}")
                except:
                    print(f"Episode: {episode}, Total Reward: {total_reward}, Steps: {steps}, Epsilon: {epsilon}, Moving Avg Reward: {moving_avg}")
        return moving_averages


    def update_target_network(self):
        self.target_q_network.load_state_dict(self.q_network.state_dict())

    def select_action(self, state, epsilon):
        if random.random() > epsilon:
            state = torch.FloatTensor(state).unsqueeze(0).to(device)  # Ensure state is on the correct device
            q_value = self.q_network(state)
            action = q_value.max(1)[1].item()
        else:
```

```
            action = random.randrange(self.action_dim)
        return action

    def save_model(self, episode, folder, verbose):
        """Save the current model to disk within a specified folder."""
        filename = os.path.join(folder, f'q_network_episode_{episode}.pkl')
        torch.save(self.q_network.state_dict(), filename)
        if verbose:
            print(f'Model saved to {filename}')

    def load_model(self, episode, folder):
        """Load a model from disk."""
        filename = os.path.join(folder, f'q_network_episode_{episode}.pkl')
        if os.path.exists(filename):
            self.q_network.load_state_dict(torch.load(filename, map_location=device))
            print(f'Model loaded from {filename}')
        else:
            print(f'No model file found at {filename}.')
```

Matlab Code

```
% FIT ARX MODEL TO 1A
N = 50;
dt = 1;
t = 0:dt:N-1;
rng(0);
u = 0.1 * randn(size(t));

% Obtain output from the piezo_nozzle model
P = piezo_nozzle(u, N); + 0.04

% Prepare the data
data = iddata(P, u', dt);

% Initialize variables to track the best models
best_fit_arx_ls = -Inf;
best_order_arx_ls = [];
best_a_est = [];
best_b_est = [];

best_fit_arx = -Inf;
best_order_arx = [];

% Test all 5th to 10th order models
for p = 8
    for z = 7  % Ensure model has more poles than zeros or equal
        % ARX Least Square Approach
        [a_est, b_est] = arx_LS(u, P, t, p, p);
        A = [1; a_est]; % Correct sign based on previous discussion
        B = b_est;
        C = 1;
        model_arx_ls = idpoly(A', B', C);
```

```
        y_pred_ls = predict(model_arx_ls, data);
        fit_arx_ls = goodnessOfFit(y_pred_ls.y, data.y, 'NRMSE');

        if fit_arx_ls > best_fit_arx_ls
            best_fit_arx_ls = fit_arx_ls;
            best_order_arx_ls_p = p;
            best_order_arx_ls_z = z;
            best_a_est = a_est;
            best_b_est = b_est;
            best_model_arx_ls = model_arx_ls;
        end

        % System Identification Toolbox Approach
        model_order = [p z 1];  % nk = 1 to handle MATLAB indexing
    (start from z^-1)
        arx_model = arx(data, model_order);
        y_pred_arx = predict(arx_model, data);
        fit_arx = goodnessOfFit(y_pred_arx.y, data.y, 'NRMSE');

        if fit_arx > best_fit_arx
            best_fit_arx = fit_arx;
            best_order_arx_p = p;
            best_order_arx_z = z;
            best_model_arx = arx_model;
        end
    end
end

% Report the best models
fprintf('Best ARX LS Model Order: %d, Fit: %.2f%%\n',
    best_order_arx_ls, best_fit_arx_ls*100);
fprintf('Best ARX Model Order: %d, Fit: %.2f%%\n', best_order_arx,
    best_fit_arx*100);

figure;

% Actual data
plot(data.y, 'k', 'LineWidth', 1.5);
hold on;

% Predictions from ARX model
y_pred_arx = predict(best_model_arx, data);
plot(y_pred_arx.y, 'r--', 'LineWidth', 1.2);

% Predictions from best ARX_LS model
y_pred_arx_ls = predict(best_model_arx_ls, data);
plot(y_pred_arx_ls.y, 'b-.', 'LineWidth', 1.2);

legend('Actual Data', 'ARX Model', 'ARX LS Model', 'Location', '
    Best');
xlabel('Time');
ylabel('Output');
title('Comparison of Actual Data with ARX and ARX LS Models');
grid on;

function [a_est, b_est] = arx_LS(u, y, t, n, m)
    N = length(y);
```

```
    A = zeros(N-n, n+m);
    b = zeros(N-n, 1);
    for i = (n+1):N
        y_segment = -y(i-1:-1:i-n)'; % Transpose to make it a row vector
        u_segment = u(i:-1:i-m+1);

        A(i-n, :) = [y_segment u_segment];
        b(i-n) = y(i);
    end
    x = (A'*A) \ (A'*b);
    a_est = x(1:n);
    b_est = x(n+1:end);
end

function fit = goodnessOfFit(y_pred, y_true, method)
    switch method
        case 'NRMSE'
            fit = 1 - norm(y_pred - y_true) / norm(y_true - mean(y_true));
        otherwise
            fit = NaN;
    end
end
%Identify parameters of LotVol equations
load('fox_rabbit.mat')

u = (rabbit.*fox);
U = u(1:end-1);
U_rabbit = rabbit(1:end-1);
U_fox = fox(1:end-1);

data1 = iddata(dx_dt', [U_rabbit, U], dt);
data2 = iddata(dy_dt', [U_fox, U], dt);

na = 0;
nb = [1 1];
nk = [0 0];

model_order_rabbit = [na nb nk];
arx_model_rabbit = arx(data1, model_order_rabbit);

model_order_fox = [na nb nk];
arx_model_fox = arx(data2, model_order_fox);

figure;
compare(data1, arx_model_rabbit);
title('Rabbit Population Change Model Fit');

figure;
compare(data2, arx_model_fox);
title('Fox Population Change Model Fit');

load('fox_rabbit.mat')
td = [0:83];
rabbit = rabbit(1:85)';
fox = fox(1:85)';
```

```
p = [rabbit(1) fox(1) 1.5343    0.1025    0.2981    0.0101];

[p,fval,exitflag] = fminsearch(@leastcomp,p,[],td,rabbit,fox);


[t,y] = ode23(@lotvol,td,[p(1),p(2)],[],p(3),p(4),p(5),p(6));

figure;
plot(td, rabbit(1:end-1), 'b', td, y(:,1), 'r-');
xlabel('Time');
ylabel('Population');
legend('Actual Rabbits', 'Fitted Rabbits');
title('Refined fit of Rabbit model')

figure;
plot(td, fox(1:end-1), 'b', td, y(:,2), 'r-');
xlabel('Time');
ylabel('Population');
legend('Actual Foxes', 'Fitted Foxes');
title('Refined fit of Fox model')


function J = leastcomp(p,tdata,xdata,ydata)
n1 = length(tdata);
[t,y] = ode23(@lotvol,tdata,[p(1),p(2)],[],p(3),p(4),p(5),p(6));
errx = y(:,1)-xdata(1:n1)';
erry = y(:,2)-ydata(1:n1)';
J = errx'*errx + erry'*erry;
end

function dydt = lotvol(t,y,a1,a2,b1,b2)
tmp1 = a1*y(1) - a2*y(1)*y(2);
tmp2 = -b1*y(2) + b2*y(1)*y(2);
dydt = [tmp1; tmp2];
end


%Simulate LQR 2
syms R F real
alpha = 1.5343;
beta = 0.1025;
delta = 0.2981;
gamma = 0.0101;

dRdt = alpha * R - beta * R * F;
dFdt = gamma * R * F - delta * F;

J = jacobian([dRdt, dFdt], [R, F]);

JacobianFunc = matlabFunction(J, 'Vars', [R, F]);

Q = [0.1 0; 0 0.1];
R_mat = 10;

dt = 1;

colors = lines(20);
```

```
197
198 R_values_all = cell(1, 20);
199 F_values_all = cell(1, 20);
200
201 figure;
202 hold on;
203
204 for j = 1:20
205     R0 = ceil(rand * 10);
206     F0 = ceil(rand * 10);
207
208     state = [R0; F0];
209
210     R_values = R0;
211     F_values = F0;
212
213     for i = 1:50
214         disp(state)
215         A = double(JacobianFunc(state(1), state(2)));
216         B = [0; 1];
217
218         K = lqr(A, B, Q, R_mat);
219         control_input = -K * (state - [30; 15]);
220
221         state = state + B * ceil(control_input * dt);
222         state = ceil(state);
223         state = max(state, [0; 0]);
224
225         dPop = double([alpha * state(1) - beta * state(1) * state
    (2); -delta * state(2) + gamma * state(1) * state(2)]);
226         state = state + dPop * dt;
227         state = ceil(state);
228         state = max(state, [0; 0]);
229
230         R_values = [R_values state(1)];
231         F_values = [F_values state(2)];
232     end
233
234     R_values_all{j} = R_values;
235     F_values_all{j} = F_values;
236
237     plot(R_values, F_values, 'Color', colors(j,:));
238     if j == 1 % Only label once to avoid duplicate labels in legend
239         quiver(R_values(1:end-1), F_values(1:end-1), diff(R_values)
    , diff(F_values), 0, 'MaxHeadSize', 0.5, 'Color', colors(j,:),
    'AutoScale', 'off');
240     end
241 end
242
243 xlabel('Rabbits');
244 ylabel('Foxes');
245 title('Phase Space Trajectory');
246 hold off;
247
248 % New figure for Rabbit population vs. Time
249 figure;
250 hold on;
```

```
251
252 for j = 1:20
253     plot(1:length(R_values_all{j}), R_values_all{j}, 'Color',
    colors(j,:));  % Plot each trial's rabbit population over time
254 end
255
256 xlabel('Time Steps');
257 ylabel('Rabbit Population');
258 title('Rabbit Population Over Time');
259 ylim([0, 100]);  % Fix the y-axis to be 0-100
260
261 hold off;
```