Trajectory Tracking in Unity Engine

BY

Abel Molinar

UNDERGRADUATE HONORS THESIS

Department of Mechanical Engineering
The University of New Mexico
Albuquerque, New Mexico

May 2025

Student

Supervising Instructor

Director of ME Undergraduate Programs

ABSTRACT

Modern autonomous systems need ways to track their trajectory in order to localize themselves in space. This task is accomplished by state estimation algorithms such as the Kalman Filter. In conjunction with sensor measurements and a controller, an autonomous system can operate with a reasonable self-estimation. Simulating autonomous systems is of utmost importance to predict behavior and gain an understanding before any kind of hardware implementation. Such a simulation necessitates realistic physics to properly asses an autonomous system in a semi-real-world environment. This research seeks to develop a software framework connecting the powerful game/physics engine of Unity with Python, utilizing Unity as the physics playground and Python as the computational hub. With this framework, control algorithms will be compared to one another to assess their ability to drive an object to a destination in the Unity Engine.

TABLE OF CONTENTS

1 Introduction

Autonomous systems are playing an increasingly important role in the development of humanity and in the ways humanity interacts with the world. Innovations such as robotic arms have made possible a kind of manufacturing that is unmatched in terms of speed and productivity. Transportation is increasingly becoming an autonomous process requiring less human input.

In a world that is quickly developing a greater role for autonomous systems, education and practical work is of utmost importance for students to learn the basics of autonomy or other topics [1] [2] [3]. It is also important to have solid simulation based tools to evaluate the behavior of autonomous systems and algorithms prior to any real world implementation. Taking these two problems side by side, a strong and realistic simulation framework ought to be combined with an easy to use and easy to understand programming language.

A popular engine used for video game development is the Unity Engine. Unity is free to use software that has a built in physics engine along with a vast array of assets that make it possible to create realistic environments. For the purposes of this research, a basic environment will be used along with the rigid body dynamics that Unity offers.

Unity, however, makes use of C Sharp (C#) as its native programming language. While C# is a very useful and powerful high level programming language in its own right, it is generally not viewed as favorably when thinking about its ease of use for beginners. Python, on the other hand, is famously known for its easy to understand syntax and friendliness to beginners. For this reason, this research seeks to combine the unique powers of Unity's extensive simulation potential with Python's friendly syntax. Unity will act as the physics playground while Python will act as the data processor/controller. This will add to

2 Problem Setup

2.1 Connecting Python and Unity

To successfully execute a simulation for an autonomous system that moves in Unity while being controlled by Python, we must first connect Python and Unity such that they can communicate together. This means that Unity can send relevant information to Python, Python will process this information and produce a control command that will then affect our Unity based autonomous system.
This will be done through sockets. Sockets establish networks based on addresses that make it possible to send data from one node in the network to another node in the network. For this research, the sockets are necessary to connect Python and Unity on the same computer. More specifically, a minimal amount of C# code will be used to send Unity data to Python using a socket. Luckily, both C# and Python have easy access to sockets using the System.Net.Sockets and Sockets packages, respectively. Once Python is able to process that data, Python will send the control command back to Unity which will be implemented via a basic C# command.

2.2 Autonomous System Design

To understand this more concretely, figure 1 shows the autonomous system diagram. The box shaded in blue represents a process that occurs in Unity/C#, while the red boxes represents processes that occur in Python. Essentially, the boxes are blocks of code that execute commands while the arrows between blocks represent signals or final answers that play a role in the next block. Next we will break down each block individually.

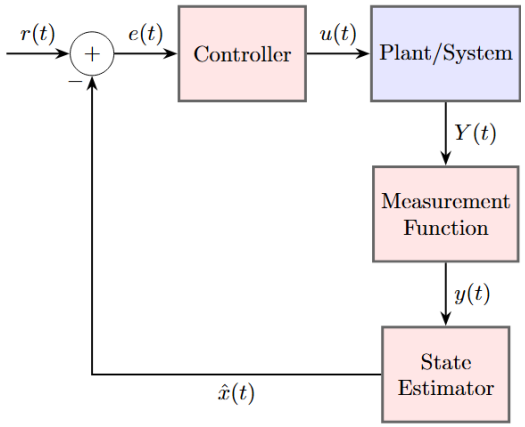


Figure 1: Autonomous System Diagram

Plant/System: This is the only block executed in Unity/C#. This block will capture the current state of our autonomous system, which will be its position in xyz space and the corresponding velocities. This results in a total of six states being captured which will then be sent off to Python via the socket connection. This is the $Y(t)$ signal. When the $u(t)$ signal is received, the commands will be applied via C# to a velocity change command. This means our controller will manipulate the velocity of our system.

Measurement Function: Given that we are using a simulation to conduct our analysis, we have access to the ground truth data. However, in real applications there is no ground truth data, and any sensors or equations are subject to uncertainty as a result. This is the impetus for state estimation algorithms. To reflect this fact, noise will be added artificially to our autonomous system such that measurements are uncertain. The measurement function will take ground truth data from Unity and add Gaussian noise to it, producing uncertain data $y(t)$.

State Estimator: The state estimation algorithm will take the uncertain data $y(t)$ and combine it with dynamic equations for our system to produce an optimal state estimate $\hat{x}(t)$. The state estimator used here will be the linear, multi-variable Kalman filter which is a widely used and practical algorithm when dealing with

Gaussian noise in linear systems.

Controller: The controller will take an error $e(t)$, which is made up of the difference between a reference and state estimate. The reference is a vector that gives a information about the end goal that we want for the position and velocity, this must be user defined (e.g. the target is to get to (15, 0, -30, 0, 0, 0), where the first 3 elements represent xyz coordinates, and the zeros represent the desired velocity at that point). The error will be processed by the controller which will then produce a control command $u(t)$ that will be sent for final execution in Unity.

2.3 Dynamic Equations

The dynamic equations for the autonomous system will contain information about the six relevant states, those being position in space and the corresponding velocities. It will follows the state space model that is widely used in control theory: $\vec{x}_{k+1} = A\vec{x}_k + B\vec{u}_k$. Where the subscript k represents the time index. The state space model is shown in equation (1).

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ z_{k+1} \\ \dot{x}_{k+1} \\ \dot{y}_{k+1} \\ \dot{z}_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & dt \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_k \\ y_k \\ z_k \\ \dot{x}_k \\ \dot{y}_k \\ \dot{z}_k \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_{x,k} \\ u_{y,k} \\ u_{z,k} \end{pmatrix} \tag{1}$$

Here dt represents the time step. This is a constant value that is set to 0.1. Notice that the B matrix has all zeros for the position values. Thus, as mentioned earlier, the commands that make up the $\vec{u}$ vector can only affect the velocities of the system. Equation (1) will be used in our state estimator to predict the future state of our autonomous system. The A and B matrices will also be used for some of the

controller algorithms.

2.4 State Estimator

The state estimator being used is a typical multivariate Kalman filter [4]. The equations for this state estimator are well known and shown below in their respective blocks.

$$\begin{aligned}
\text{Prior Block:} \quad \vec{x}_k^- &= A\vec{x}_{k-1} + B\vec{u}_k \\
P_k^- &= AP_{k-1}A^T + Q \\
\text{Posterior Block:} \quad K_k &= P_k^- H^T (HP_k^- H^T + R)^{-1} \\
\vec{x}_k &= \vec{x}_k^- + K_k(\vec{z}_k - H\vec{x}_k^-) \\
P_k &= (I - K_k H)P_k^-
\end{aligned}$$

Where the $^-$ superscript means the prior estimate, P is the error covariance matrix (6x6 diagonal with entries of 0.1), R is the measurement covariance matrix (6x6 diagonal with entries of 0.1), Q is the process noise matrix (6x6 diagonal with entries of 0.1), H is the observation matrix (6x6 identity), K is the Kalman gain matrix, and $\vec{z}_k$ is the measurement vector.

2.5 Objectives

The autonomous system will be tasked with moving from a point in space to another point in space. Figure 2 shows the autonomous system, a red sphere, on a large plane in the Unity Engine. Beginning at the point (20, 0, -30), the sphere will move to the point (-15, 0, 20).

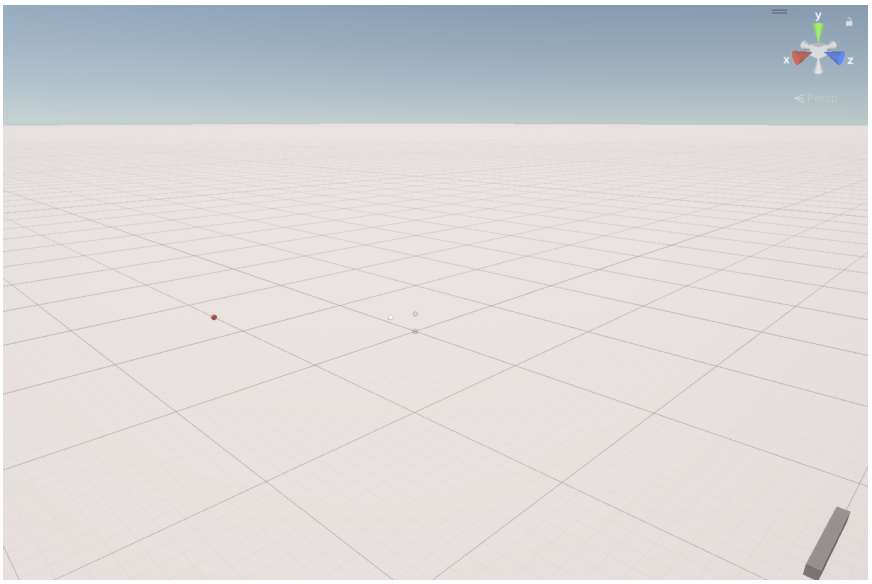


Figure 2: Simulation Environment

2.6 Control Algorithms

This will be accomplished with 3 different controllers, a Proportional Integral Derivative (PID), Linear Quadratic Regulator (LQR), and a Model Predictive Controller (MPC).

2.6.1 PID

PID controllers have proven to be incredibly useful in engineering applications due to the combination of simplicity and effectiveness. Equation (2) shows the equations for the PID controller.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \tag{2}$$

The coefficients for the PID K_p, K_i, and K_d are the key for PID effectiveness. For the purposes of this research, these values are 10, 0.1, 30. The PID will also have an input rate limiter capped at 2(i.e. the PID values cannot change by more than 2 for each time step) and an input limit of 0.5 (i.e. any element in the output vector

$\vec{u}_k$ cannot be greater than 0.5).

2.6.2 LQR

The LQR is an optimal controller that, conveniently, can be calculated entirely offline, i.e. there exists an analytic solution [5]. The LQR solves the cost function in equation (3). Of key importance for the LQR are the positive semi-definite weighing Q and R matrices. The Q matrix is the penalty for the state e, a higher penalty translates to a faster convergence towards the final target. The R matrix is the penalty for the control command u, a higher penalty translates to a greater restriction on the magnitude of the control command. For this research, the Q and R matrices are a 6x6 diagonal with elements of 10 and a 3x3 diagonal with elements of 1000, respectively.

$$J = \int_{t_0}^{\infty} e^\top Q e + u^\top R u \, dt \tag{3}$$

Equation (4) shows the control command produced by the LQR, with equation (5) and equation (6) representing the analytic solution to the K Matrix and the solution to the Algebraic Riccati equation respectively.

$$u(t) = -Ke(t) \tag{4}$$

$$K = R^{-1} B^\top \bar{P} \tag{5}$$

$$A^\top \bar{P} + \bar{P} A + Q - \bar{P} B R^{-1} B^\top \bar{P} = 0 \tag{6}$$

2.6.3 MPC

Model Predictive Control is an increasingly popular optimal controller known for its good performance in complex systems [6]. The idea with MPC is to construct a model of the system that allows you to predict the future states and, with that information, solve a cost function. A set of control commands will be produced from this process, only the first of those commands is actually implemented. After that, the process is rerun again. It is similar to an LQR that needs to be solved each time a control command is needed. Because of this, MPC is considerably more computationally expensive but it is much more flexible, especially when considering non-linear systems. MPC also allows for greater flexibility in constraints, allowing for soft or hard constraints as opposed to the LQR's soft constraints only approach with weighing matrices.

The equation that will be used to predict future states is equation (1). The cost function for this MPC is the sum of squares, shown in equation (7).

$$J = \int_{t_0}^{t_0+dt} e^\top e + u^\top u \, dt \tag{7}$$

Other than the constraints from the dynamics and starting conditions, the magnitude of the control command will be limited to less than 0.5 ($u^\top u \leq 0.5$). Moreover, this MPC will predict states 5 time steps into the future, this is because predicting more time steps becomes increasingly computationally expensive which results in lag between Unity and Python.

3 Results

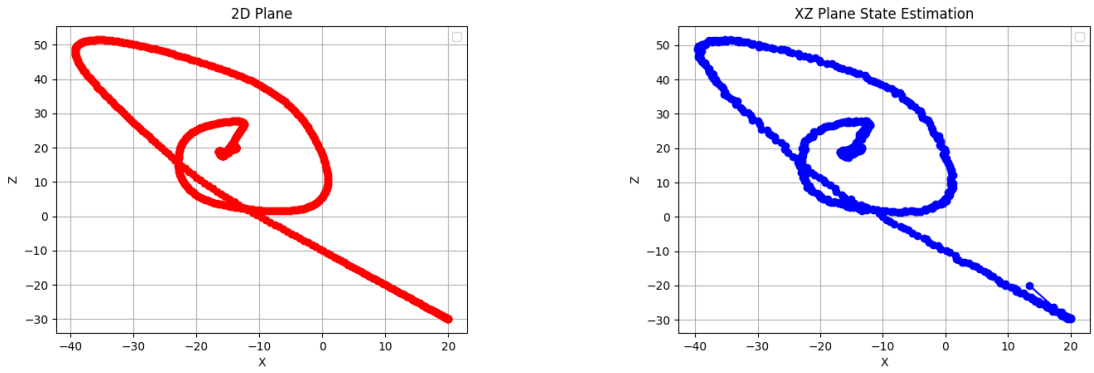
The results presented here will be:

- xz plane ground truth trajectory

- xz plane state estimation trajectory
- x time state estimate
- z time state estimate

3.1 PID Performance

Figure 3 shows the PID's trajectory. The PID struggles to reach the target without severe overshoot. However, the state estimator, which is what the PID is using as its input, does a good job at continually localizing and correcting the state estimate, as shown by the closeness of figure 3a and figure 3b. The shapes are very similar, with figure 3b understandably having more noise.

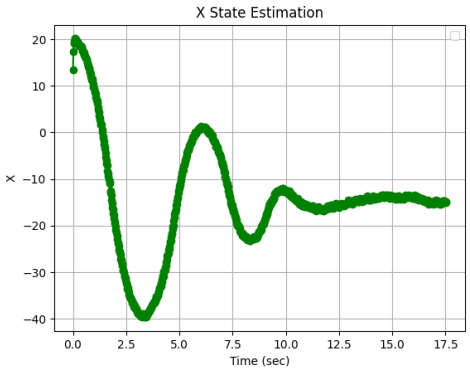


(a) PID Trajectory Ground Truth

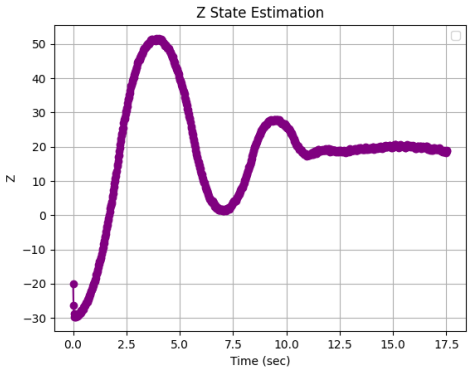
(b) PID Trajectory State Estimation

Figure 3: PID 2D Trajectory Plots

Figure 4 represent the time state estimation plots for both the x and z coordinates with the PID. These plots further explain the overshoot problems that the PID suffers from, overshooting the z coordinate by over 30. This demonstrates that the PID is considerably lacking with regard to driving the autonomous system to its stated goal.



(a) PID X State Estimate

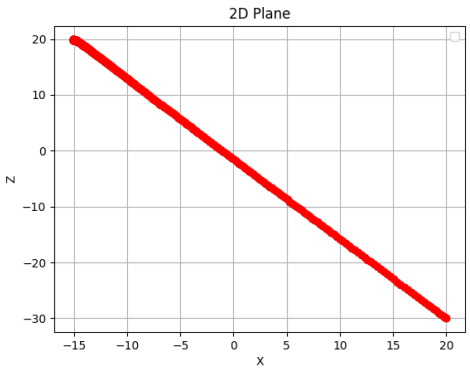


(b) PID Z State Estimate

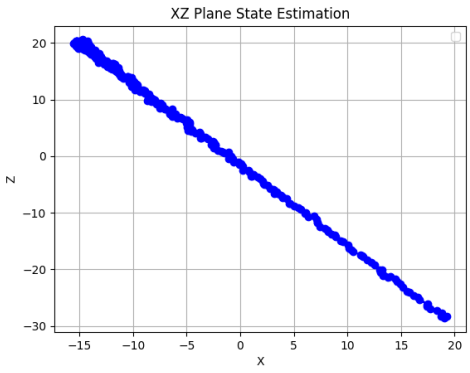
Figure 4: PID Time Plots

3.2 LQR Performance

Figure 5 shows the system's trajectory when using the LQR controller. With the LQR being an optimal controller, it is not surprise that it is considerably better at driving the autonomous system over to its end goal. The Kalman filter continues to accomplish its task of localizing the system from which the controller can produce optimal outputs.



(a) LQR Trajectory Ground Truth

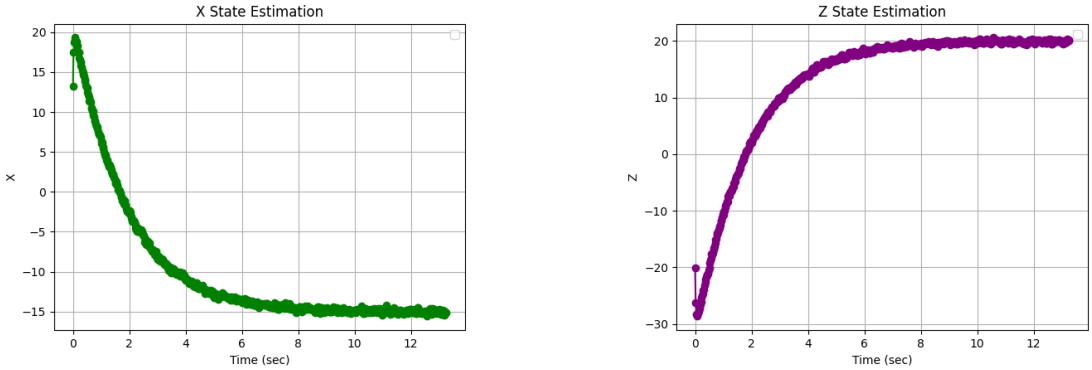


(b) LQR Trajectory State Estimation

Figure 5: LQR 2D Trajectory Plots

Figure 6 shows the time state estimation plots for the x and z using with the LQR. Both plots show a clear trend towards the final target that gets there without over-

shoot. Here the bigger problem is the noisiness of the plots which causes small perturbations for the system even when it has reached its target.

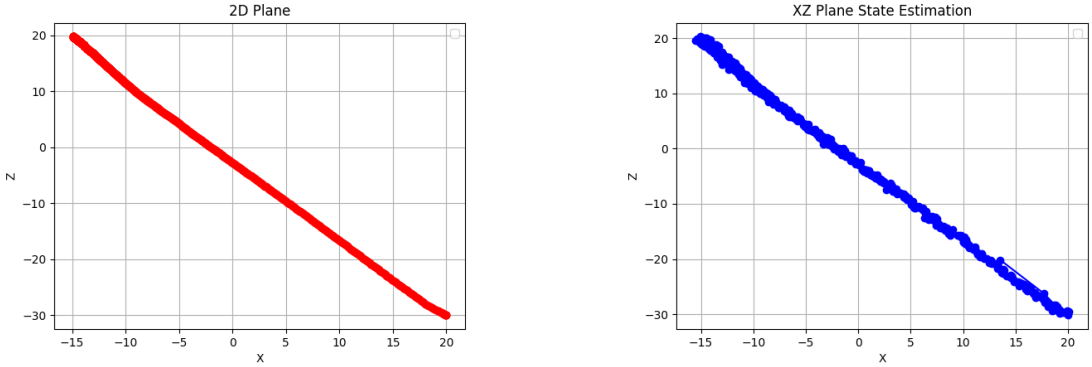


(a) LQR X State Estimate (b) LQR Z State Estimate

Figure 6: LQR Time Plots

3.3 MPC Performance

Figure 7 shows the xz trajectory when using the MPC. Being an optimal controller similar to the LQR, it is not surprise that both figure 7a and figure 7b look very similar to their LQR counterparts. Once more the state estimator is successful at harmonizing with the controller to produce an accurate localization and control command.

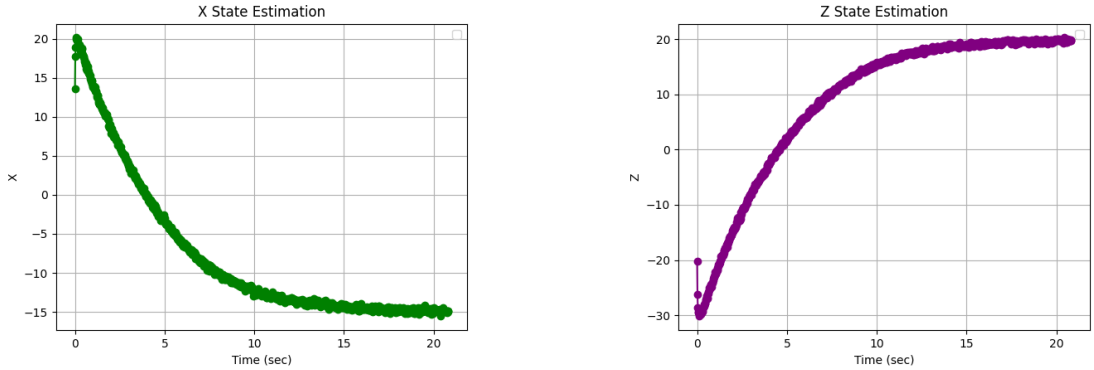


(a) MPC Trajectory Ground Truth (b) MPC Trajectory State Estimation

Figure 7: MPC 2D Trajectory Plots

Figure 8 represents the time state estimation for both x and z with the MPC. Both

plots are similar to their LQR counterparts once more, though the MPC takes longer to reach its target due to the hard constraints imposed on the controller. It takes roughly 7 more seconds for the MPC to reach its target compared to the LQR. Once more the noise causes small perturbations even when it has reached its end goal.



(a) MPC X State Estimate (b) MPC Z State Estimate

Figure 8: MPC Time Plots

4 Discussion

After analyzing the 2-dimensional trajectories and variable-time plots, some conclusions can be reached. Considering the 2-dimensional trajectory plots, the Kalman filter is able to localize the autonomous system no matter what control regime is in place and no matter its computational complexity. This give us confidence when analyzing any data coming out of our state estimator and makes it possible to rely on the state estimator and not just the ground truth. The most important conclusion here is that noise can be added to simulation models because the noise can be accounted for by the state estimator, as is the case in real life applications. Considering the variable-time plots, it is clear that both the LQR and MPC optimal controllers are superior to the PID. This is likely due to the asymmetry of the PID when moving in the xz plane (i.e. the goal is to move from 20 to -15 in the

x and from -30 to 20 in the z). Without a more in depth review of the PID coefficients for this specific situation or even for different PID coefficients in the x and z, it is difficult for the PID to not overshoot its target.

The optimal controllers show better performance in reaching their target. Both the MPC and LQR, because they use optimal solutions to cost functions, are more "holistic" control algorithms because they solve for a specific vector $\vec{u}_k$ rather than just the individual entries in that vector as the PID does. The LQR is able to reach the target faster than the MPC. This might seem like the LQR is better than the MPC, however in practice the LQR is only able to reach the target faster because the soft constraints on the LQR are unclear and cannot be tested in any way other than trial and error. The MPC, on the other hand, can incorporate hard constraints which limit the control output to some desired bounds explicitly, requiring no trial and error testing. The biggest flaw of the MPC is the computational expense, attempting to predict more than 5 time steps ahead leads to severe lag between Python and Unity.

5 Conclusion

In conclusion, a software framework connecting the physics playground of Unity with the easy to understand computational hub of Python for the purposes of testing autonomy algorithms has been established. PID, LQR, and MPC controllers were tested in order to asses their ability to drive a virtual autonomous system towards a reference point. In conjunction with artificial Gaussian noise and a Kalman filter, all of the controllers where able to reach their target in spite of added noise. This further proved the strength of this kind of simulation because algorithms can be tested and compared to the simulation's own ground truth, providing useful performance data. Moreover, the optimal controllers performed the best, with the LQR beating the MPC in terms of speed. However, the MPC still proves to be the

more flexible controller thanks to its optimal behavior and ability to incorporate hard constraints.

Future work can focus on creating more realistic environments in Unity and even implementing collision avoidance algorithms. This software framework can further utilize Unity's physics capabilities including its use of forces and torques as control variables instead of velocity. Using Python, it is possible to write or use an extensive controls/mathematical library that comes with being one of the most popular and intuitive programming languages in the world. There is potential to test a plethora of control or state estimation algorithms in Unity [7], only possibly limited by computational complexity. The software framework developed here has potential for engineering education due to the combination of the easy to understand programming language of Python and the 3D simulation capabilities of Unity.

A C# Code

The following code is used to control the autonomous system in Unity. This code applies rigid body dynamics to the autonomous system and allows for data transfer between Python and Unity.

```
using System;
using System.Net.Sockets;
using System.Text;
using Unity.Collections;
using Unity.Mathematics;
using UnityEngine;

public class PlayerMovements : MonoBehaviour
{
    private TcpClient client;
    private NetworkStream stream;
    private string receivedData;
    public Rigidbody rb;
    public Transform player;

    void Start()
    {
        // Ensure Unity runs at full speed even when not in focus
        Application.runInBackground = true;

        // Set a target frame rate
        Application.targetFrameRate = 60;
```

```
        // Connect to the Python server
        client = new TcpClient("127.0.0.1", 5000);
        stream = client.GetStream();
    }

    // Update is called once per frame
    void FixedUpdate()
    {
            if (true)
        {
        // Block that sends player coords from Unity to Python
        // Send the sphere's current position (x, y, z) to Python
        Vector3 currentPosition = transform.position;
        Vector3 currentVelocity = rb.linearVelocity;
        string Data = $"{currentPosition.x},{currentPosition.y},{currentPosition.z},
        {currentVelocity.x},{currentVelocity.y},{currentVelocity.z}";
        byte[] positionBytes = Encoding.UTF8.GetBytes(Data);
        stream.Write(positionBytes, 0, positionBytes.Length);
        Debug.Log("Data Sent: " + Data);

        // Block that receives player velocities to be applied
        byte[] buffer = new byte[1024];
        int bytesRead = stream.Read(buffer, 0, buffer.Length);
        string controlData = Encoding.UTF8.GetString(buffer, 0, bytesRead).Trim();
        Debug.Log("Received control inputs: " + controlData);

        // Parse the control inputs
        string[] controlInputs = controlData.Split(',');
        float velocityX = float.Parse(controlInputs[0]);
        float velocityY = float.Parse(controlInputs[1]);
        float velocityZ = float.Parse(controlInputs[2]);

        // Apply the control inputs to the sphere
        rb.AddForce(velocityX, velocityY, velocityZ, ForceMode.VelocityChange);
        }
    }
    void OnApplicationQuit()
    {
        // Close the connection when the application quits
        stream.Close();
        client.Close();
    }
}
```

B Python Code

The Python code for this research requires the following packages:

```
import socket
import numpy as np, scipy as scp, matplotlib.pyplot as plt, cvxpy as cvx
from pytictoc import TicToc
```

B.1 Measurement Function

```
def Measurement(data, R):
```

```
        n = data.size
        zk = np.random.multivariate_normal(data, R).reshape(n,)
        return zk
```

B.2 Kalman Filter

```
class KalmanFilter:
    def __init__(self, xk, A, B, H, P, Q, R):
        self.A, self.B = A, B
        self.Q, self.R = Q, R
        self.P = P
        self.H = H
        self.xk = xk

        self.x_list, self.y_list, self.z_list = [], [], []
        self.x_var, self.y_var, self.z_var = [], [], []

    def Prediction(self, uk):
        xk_pred = self.A @ self.xk + self.B @ uk
        P_pred = self.A @ self.P @ self.A.T + self.Q
        return xk_pred, P_pred

    def Posterior(self, xk_pred, zk, P_pred):
        yk = zk - self.H @ xk_pred
        S = self.H @ P_pred @ self.H.T + self.R
        K = P_pred @ self.H.T @ np.linalg.inv(S)
        self.xk = xk_pred + K @ yk
        self.P = (np.eye(len(self.P)) - K @ self.H) @ P_pred

        self.x_list += [self.xk[0]]
        self.y_list += [self.xk[1]]
        self.z_list += [self.xk[2]]
        self.x_var += [self.P[0,0]]
        self.y_var += [self.P[1,1]]
        self.z_var += [self.P[2,2]]


    def Execute(self, uk, zk):
        xk_pred, P_pred = self.Prediction(uk)
        self.Posterior(xk_pred, zk, P_pred)

    def xk_vec(self):
        return self.xk

    def graph(self, in_list, out_list, xlabel, ylabel, title, color, marker):
        plt.plot(in_list, out_list, color = color, marker = marker)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        plt.title(title)
        plt.grid(True)
        plt.legend()
        plt.show()

    def n_val(self):
        return len(self.x_list)

    def record_xk(self, Time, bin=False):
        self.graph(self.x_list, self.z_list, xlabel='X',ylabel='Z', title='XZ Plane State
        ↪  Estimation', color='blue', marker='o')
```

```
        if bin == True:
            self.graph(Time, self.x_list, xlabel='Time (sec)',ylabel='X', title='X State
            ↪  Estimation', color='green', marker='o')
            self.graph(Time, self.z_list, xlabel='Time (sec)',ylabel='Z', title='Z State
            ↪  Estimation', color='purple', marker='o')

    def record_var(self, Time):
        self.graph(Time, self.x_var, xlabel='Time (sec)',ylabel='x Variance', title='x
        ↪  Variance Over Time', color='orange', marker='o')
        self.graph(Time, self.z_var, xlabel='Time (sec)',ylabel='z Variance', title='z
        ↪  Variance Over Time', color='orange', marker='o')
```

B.3 Error Function

```
def error_func(state_estimate, reference):
        return reference - state_estimate
```

B.4 Reference Function

The reference function can be used to trace trajectories in 3-dimensional space that are not simply constants as was the case in this paper.

```
class reference_func:

    def __init__(self, ref_x, ref_y, ref_z, dt):
        self.ref_x = ref_x
        self.ref_y = ref_y
        self.ref_z = ref_z
        self.dt = dt
        self.x_vec = []
        self.y_vec = []
        self.z_vec = []
        self.x_in = 0
        self.y_in = 0
        self.z_in = 0

    def Execute(self):
        ref_x = self.ref_x(self.x_in)
        ref_y = self.ref_y(self.y_in)
        ref_z = self.ref_z(self.z_in)
        self.x_in += self.dt
        self.y_in += self.dt
        self.z_in += self.dt
        self.x_vec.append(ref_x)
        self.y_vec.append(ref_y)
        self.z_vec.append(ref_z)

        ref = np.array([ref_x, ref_y, ref_z, 0, 0, 0]).T
        return ref

    def plot(self):
        plt.plot(self.x_vec, self.z_vec, color = 'blue', marker = 'o')
        plt.title('Reference Trajectory')
        plt.xlabel('X-Axis')
        plt.ylabel('Z-Axis')
```

```
        plt.grid(visible = 'True')
        plt.show()
```

B.5 PID

```
class PIDController:
    def __init__(self, Kp, Ki, Kd, input_lim, rate_lim, dt):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.input_lim, self.rate_lim = input_lim, rate_lim
        self.previous_error = np.array([0., 0., 0., 0., 0., 0.]).T
        self.integral = np.array([0., 0., 0., 0., 0., 0.]).T
        self.dt = dt

    def input_ratelimiter(self, input_vector):
        change = input_vector - self.previous_error
        for i in range(len(change)):
            if abs(change[i]) > self.rate_lim:
                if change[i] < 0:
                    input_vector[i] = self.previous_error[i] - self.rate_lim
                else:
                    input_vector[i] = self.previous_error[i] + self.rate_lim

    def limit(self, input_vector):
        for i in range(len(input_vector)):
            if abs(input_vector[i]) >= self.input_lim:
                if input_vector[i] < 0:
                    input_vector[i] = -1.0 * self.input_lim
                else:
                    input_vector[i] = self.input_lim


    def u(self, error):
        # Proportional term
        P_out = self.Kp * error

        # Integral term
        self.integral += error * self.dt
        I_out = self.Ki * self.integral

        # Derivative term
        derivative = (error - self.previous_error) / self.dt
        D_out = self.Kd * derivative

        # Compute total output
        pid_output = P_out + I_out + D_out
        self.input_ratelimiter(pid_output)
        self.limit(pid_output)

        # Update previous error
        self.previous_error = error
        print(f"hey this is {pid_output}")
        return pid_output[0:3].T
```

B.6 LQR

```
class LQR:
```

```
    def __init__(self, A, B, Q, R):
        self.A, self.B = A, B
        self.Q, self.R = Q, R
        P = scp.linalg.solve_discrete_are(self.A, self.B, self.Q, self.R)
        self.K = np.linalg.inv(self.R + B.T @ P @ B) @ (B.T @ P @ A)

    def u(self, error):
        return self.K @ error
```

B.7 MPC

```
class MPC:
    def __init__(self, A, B):
        self.A, self.B = A, B

    def u(self, initial):
        T = 5
        u = cvx.Variable((3, T))
        s = cvx.Variable((6, T + 1))
        cost = 0
        constraints = []
        for t in range(T):
            cost += cvx.sum_squares(s[:,t + 1]) + cvx.sum_squares(u[:, t])
            constraints += [s[:,t + 1] == self.A @ s[:,t] + self.B @ u[:, t], cvx.norm(u[:,
            ↪  t], "inf") <= 0.5]
        constraints += [s[:, 0] == initial]
        prob = cvx.Problem(cvx.Minimize(cost), constraints)
        prob.solve()
        u = u.value[:,0]
        return -1 * u
```

B.8 Code Initialization

With the above code for specific portions of the system covered, the following code will initialize the simulation and produce the graphs seen in this paper after the simulation is terminated. Please keep in mind that, immediately after running this code, Unity must be engaged as well and, when the user deems it, Unity must be manually stopped for the code to terminate.

```
def ref_x(dt):
    return -15

def ref_y(dt):
    return 0.

def ref_z(dt):
    return 20

dt = 0.1
Reference = reference_func(ref_x, ref_y, ref_z, dt = 1/60)
control_vector = np.array([0, 0, 0]).T
```

```
process_var, meas_var = 0.1, 0.1

xk = np.array([0, 0, 0, 0, 0, 0]).T

A = np.array([[1, 0, 0, dt, 0, 0],
              [0, 1, 0, 0, dt, 0],
              [0, 0, 1, 0, 0, dt],
              [0, 0, 0, 1, 0, 0],
              [0, 0, 0, 0, 1, 0],
              [0, 0, 0, 0, 0, 1]])

B = np.array([[0, 0, 0],
              [0, 0, 0],
              [0, 0, 0],
              [1, 0, 0],
              [0, 1, 0],
              [0, 0, 1]])

H = np.eye(6)

P = np.eye(6) * meas_var

Q = np.eye(6) * process_var
R = np.eye(6) * meas_var

State_Est = KalmanFilter(xk, A, B, H, P, Q, R)


# Controller initialization

# PID
# Kp, Ki, Kd = 10., 0.01, 50.
# input_lim, rate_lim = 0.5, 2.
# Control = PIDController(Kp, Ki, Kd, input_lim, rate_lim, dt)

# LQR
# SM = 10 * np.eye(6)
# IM = 1000 * np.eye(3)
# Control = LQR(A, B, SM, IM)

# MPC
# Control = MPC(A, B)

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('127.0.0.1', 5000)
print(f"Starting server on {server_address[0]}:{server_address[1]}")
server_socket.bind(server_address)
server_socket.listen(1)

print("Waiting for connection...")
connection, client_address = server_socket.accept()
print(f"Connected to {client_address}")

xp, zp = [], []
t = TicToc()
t.tic()
try:
    while True:
         # Receive current position from Unity
        data = connection.recv(1024).decode().strip()
```

```
            current_position = list(map(float, data.split(',')))  # Parse x, y, z
            print(f"Received current position: {current_position}")
            current_position = np.array(current_position)

            # Measurement Function
            zk = Measurement(current_position, R)
            reference = Reference.Execute()

            # Develop a state estimation
            State_Est.Execute(control_vector, zk)
            xk_StateEst = State_Est.xk_vec()

            # Calculate the error
            error = error_func(xk_StateEst, reference)
            # error = error_func(current_position, reference)

            # Controller
            control_vector = Control.u(error)


            # x and z positions are saved
            xp.append(current_position[0])
            zp.append(current_position[2])



            # Send control inputs back to Unity
            control_data = ','.join(map(str, control_vector))
            connection.sendall(control_data.encode())
            print(f"Sent control inputs: {control_data}")

    finally:
        t.toc()
        T_final = t.tocvalue()
        connection.close()
        server_socket.close()
        time_vec = np.linspace(start=0, stop=T_final, num=State_Est.n_val())
        State_Est.record_xk(time_vec, bin=True)
        # State_Est.record_var(time_vec)
        # Reference.plot()
```

References

[1] M. Buszewicz and M. Plechawska-Wojcik, "Student research project on physical simulations in Unity3D game engine," in *INTED2017 Proceedings*, ser. 11th International Technology, Education and Development Conference, Valencia, Spain: IATED, Jun. 2017, pp. 7209–7214. DOI: 10.21125/inted.2017.1670

[2] S. Bin, W. Yanwu, Z. Xiyong, and C. Huabin, "Virtual reality design of industrial robot teaching based on Unity3D," in *2021 7th International Sympo-*

sium on Mechatronics and Industrial Informatics (ISMII), 2021, pp. 1–4. DOI: 10.1109/ISMII52409.2021.00072

[3] V. Andaluz et al., "Unity3d-matlab simulator in real time for robotics applications," Jun. 2016, pp. 246–263, ISBN: 978-3-319-40620-6. DOI: 10.1007/978-3-319-40621-3_19

[4] G. Welch and G. Bishop, "An introduction to the kalman filter," *Proc. Siggraph Course*, vol. 8, Jan. 2006.

[5] T. Basar, "Contributions to the theory of optimal control," in *Control Theory: Twenty-Five Seminal Papers (1932-1981)*. 2001, pp. 147–166. DOI: 10.1109/9780470544334.ch8

[6] J. Rawlings and D. Mayne, *Model Predictive Control: Theory and Design*. Jan. 2009.

[7] G. F. Welch, "History: The use of the kalman filter for human motion tracking in virtual reality," *Presence*, vol. 18, no. 1, pp. 72–91, 2009. DOI: 10.1162/pres.18.1.72