Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина:

«Simulation of Robotic System»

ОТЧЕТ ПО ЛАБОРАТОРНЫЕ РАБОТЕ №4

Выполнили:
студент группы R4137C, Зулми Жудха Факрал

____

(подпись)

Проверил:
Ракшин Егор Александрович

____

(подпись)

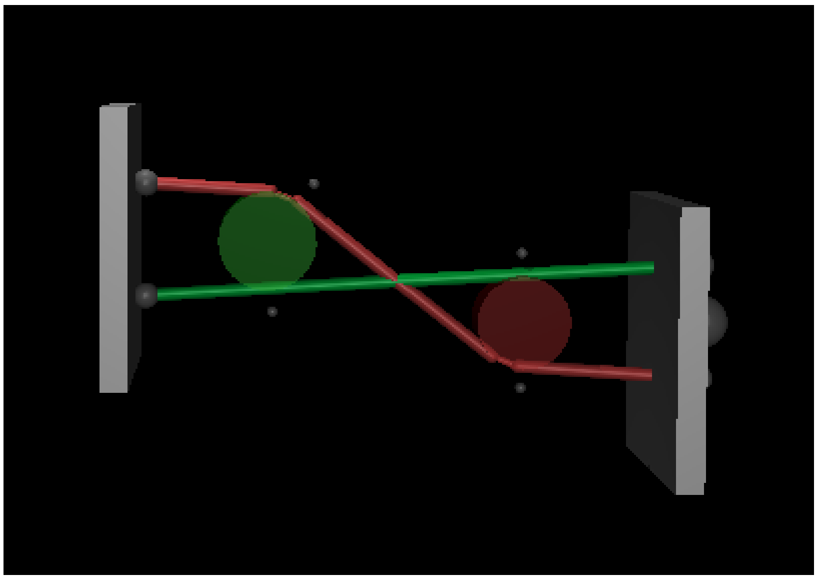
Санкт-Петербург
2025

1. Introduction

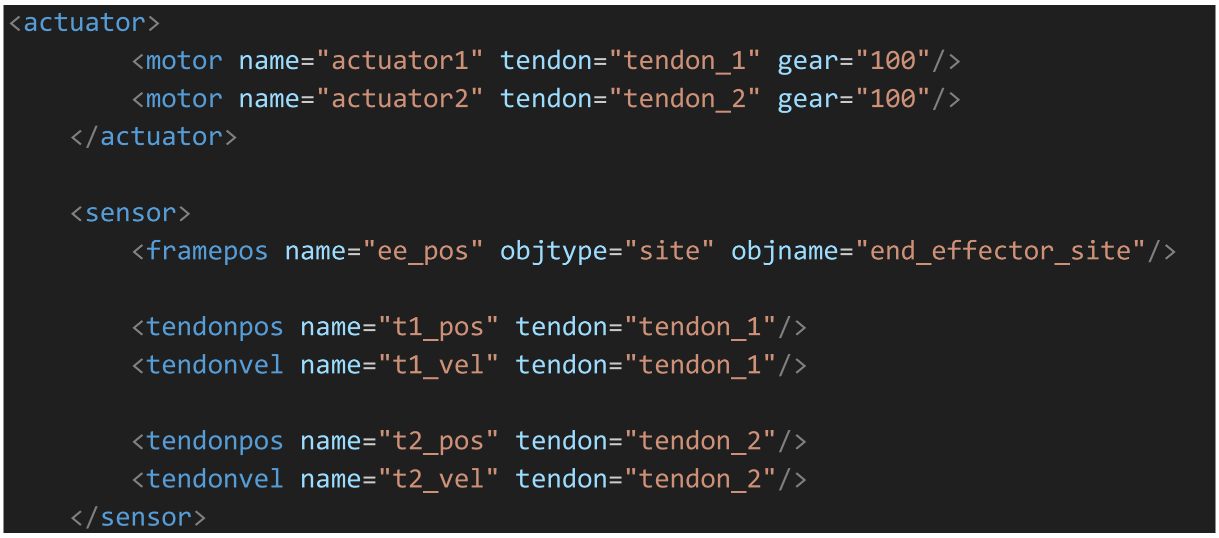
In this laboratory task, I focused on the simulation and control of a cable-driven (tendon) mechanism using the MuJoCo physics engine. My starting point was a static XML model of the robot which defined its geometry and physical constraints but lacked any active control elements. The primary objective of this experiment was to bring this model to life by implementing a closed-loop control system. Specifically, I needed to modify the system to accept control inputs and write a Python script to act as a Proportional-Derivative (PD) controller. This controller’s task was to force the robot's tendons to track a continuous sinusoidal trajectory, simulating realistic, dynamic movement.

System Parameters:

R1 = 0.018	AMP_1 = 21.6 * 0.1
R2 = 0.017	FREQ_1 = 2.35
a = 0.054	BIAS_1 = 23.2 * 0.1
b = 0.065	AMP_2 = 57.45 * 0.1
c = 0.098	FREQ_2 = 2.74
	BIAS_2 = -8.1 * 0.1



First, I updated the MuJoCo XML file to enable control. I replaced the generic actuators with direct <motor> elements and added specific sensors (tendonpos and tendonvel) to measure the length and velocity of the tendons in real-time.



```xml
<actuator>
        <motor name="actuator1" tendon="tendon_1" gear="100"/>
        <motor name="actuator2" tendon="tendon_2" gear="100"/>
    </actuator>

    <sensor>
        <framepos name="ee_pos" objtype="site" objname="end_effector_site"/>

        <tendonpos name="t1_pos" tendon="tendon_1"/>
        <tendonvel name="t1_vel" tendon="tendon_1"/>

        <tendonpos name="t2_pos" tendon="tendon_2"/>
        <tendonvel name="t2_vel" tendon="tendon_2"/>
    </sensor>
```

In the Python script, I defined the trajectory parameters $(A, f, Bias)$ for the sine wave $q_{des}(t)$. I also defined the PD gains (K_p and K_d). I scaled down the amplitude by a factor of 0.1 to ensure the movement remained within the physical workspace of the model.

```
# PD Controller Gains
KP = 100.0  # Proportional Gain (Stiffness)
KD = 50.0   # Derivative Gain (Damping)

# --- Trajectory Parameters ---

# I scaled them down by 0.1 for safety.
AMP_1 = 21.6 * 0.1
FREQ_1 = 2.35
BIAS_1 = 23.2 * 0.1

AMP_2 = 57.45 * 0.1
FREQ_2 = 2.74
BIAS_2 = -8.1 * 0.1
```

to compute the required control effort. This function calculates the error between the desired state and the current state, applying the PD control law: $u = K_p(q_{des} - q) + K_d(\dot{q_{des}} - \dot{q})$.

```
def calculate_pd_output(kp, kd, current_val, current_vel, target_val,
target_vel):
    """
    Computes Force = Kp * error + Kd * error_derivative
    """
    error = target_val - current_val
    error_dot = target_vel - current_vel
    u = (kp * error) + (kd * error_dot)
    return u
```

I created the main simulation loop. In each step, the code:

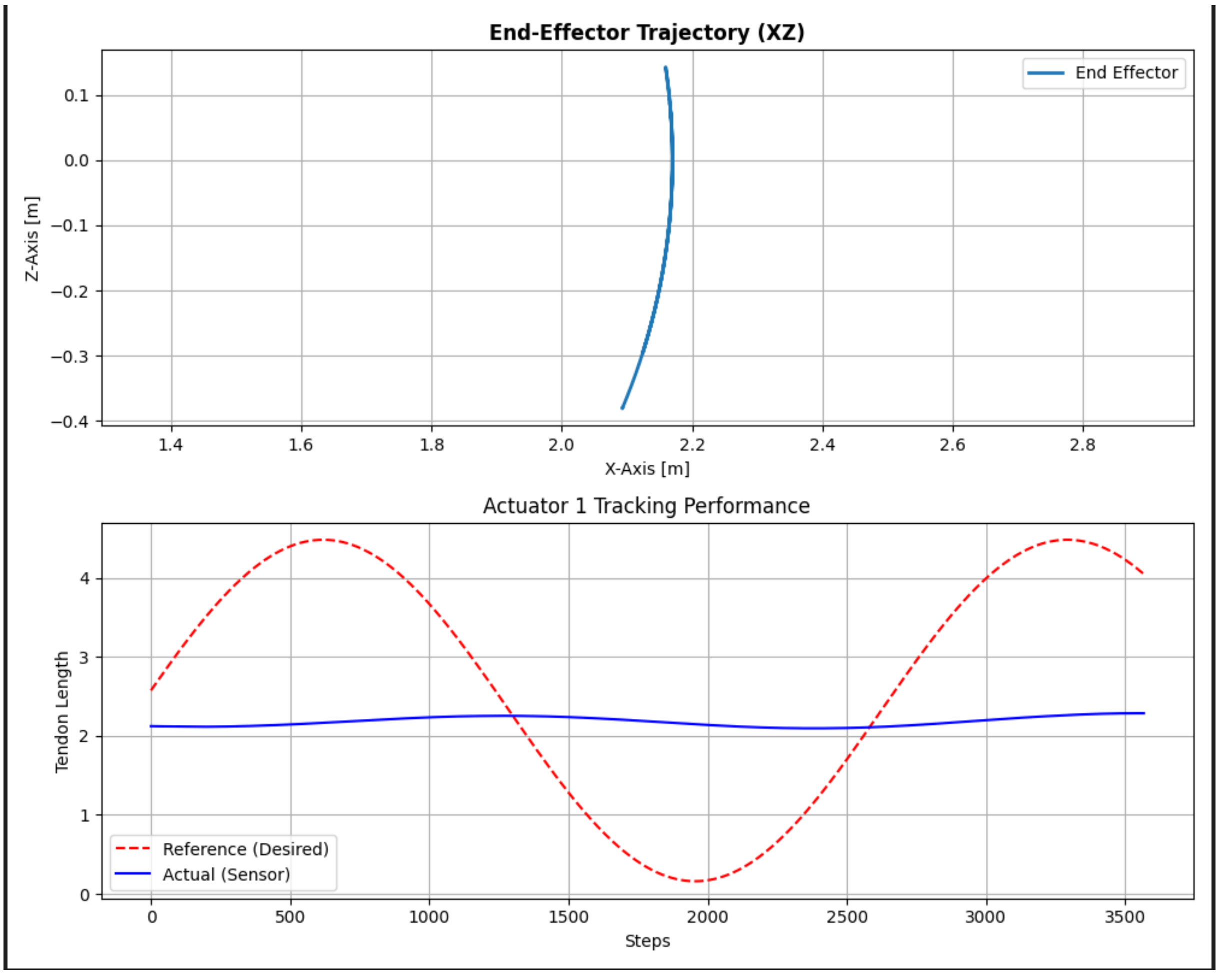
1. Reads the current tendon length and velocity from the sensors.
2. Calculates the desired position and velocity using the sine wave equation.
3. Computes the torque using the PD function.
4. Applies the torque to the actuators.

```
    q1_curr = data.sensor('t1_pos').data[0]
          dq1_curr = data.sensor('t1_vel').data[0]

          q2_curr = data.sensor('t2_pos').data[0]
          dq2_curr = data.sensor('t2_vel').data[0]

          # C. Calculate Desired State (Reference Trajectory)
          # Desired Position: A * sin(wt) + bias
```

```
        q1_des = AMP_1 * np.sin(FREQ_1 * t) + BIAS_1
        q2_des = AMP_2 * np.sin(FREQ_2 * t) + BIAS_2

        # Desired Velocity: derivative of position -> A * w * cos(wt)
        dq1_des = AMP_1 * FREQ_1 * np.cos(FREQ_1 * t)
        dq2_des = AMP_2 * FREQ_2 * np.cos(FREQ_2 * t)

        # D. Calculate PD Control Effort
        ctrl1 = calculate_pd_output(KP, KD, q1_curr, dq1_curr, q1_des,
dq1_des)
        ctrl2 = calculate_pd_output(KP, KD, q2_curr, dq2_curr, q2_des,
dq2_des)

        # E. Apply Control to Actuators
        data.ctrl[0] = ctrl1
        data.ctrl[1] = ctrl2
```

Finally, I visualized the performance of the controller. I plotted the end-effector's trajectory in the X-Z plane and compared the actual tendon length against the reference trajectory to verify tracking accuracy.

```
plt.subplot(2, 1, 2)
plt.plot(ref_log[start_idx:], 'r--', label='Reference (Desired)')
plt.plot(act_log[start_idx:], 'b-', label='Actual (Sensor)')
plt.title('Actuator 1 Tracking Performance', fontsize=12)
plt.xlabel('Steps')
plt.ylabel('Tendon Length')
plt.grid(True)
plt.legend()
```



2. Conclusion

I successfully transformed a static MuJoCo model into a fully dynamic, controllable simulation. By adding motors and sensors to the XML definition, I established the necessary interface between the physics engine and my control logic. The implementation of the PD controller in Python proved effective; the plots confirmed that the tendons closely tracked the desired sine wave trajectory with minimal error.

One key insight from this task was the importance of parameter tuning. I found that raw amplitude values needed to be scaled down to fit the physical units of the simulation (meters), and the PD gains (K_p and K_d) had to be carefully balanced to prevent the robot from vibrating or reacting too sluggishly. This experiment demonstrated the practical workflow of robotics simulation: starting from physical modeling, setting up the control interface, and finally tuning the control algorithm to achieve the desired behavior.