GE Research: GE:AI

Dom, Ansh, Mo, Sam, Adi, Gabe

Open AI - Gymnasium

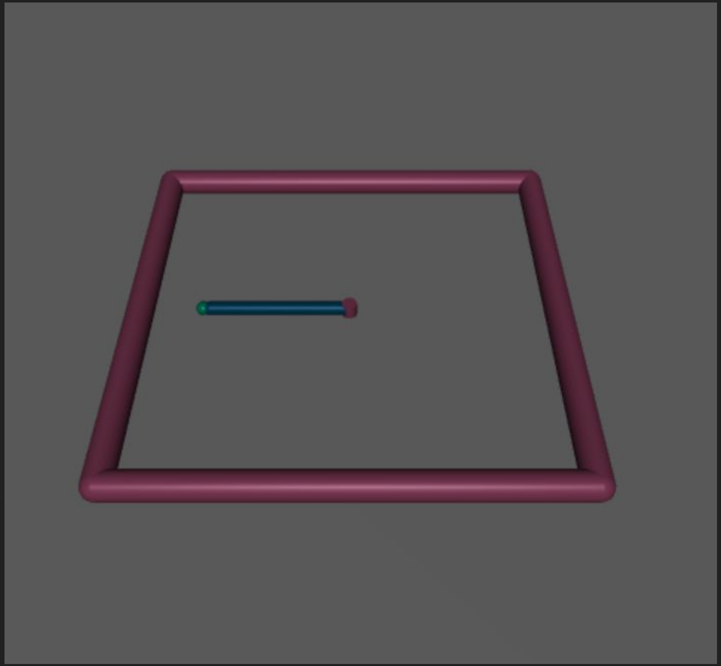
- Using OpenAI's Gymnasium, we were able to simulate movements of basic robotics and test our code.

MuJoCo - Robotic Simulator

MuJoCo is a multi-joint robotic arm physics/dynamics simulator for Gymnasium that we used to test and run our simulations.

Reacher Environment Introduction

Reacher Environment is an arm with two joints that can travel in a circle



Control of Robot

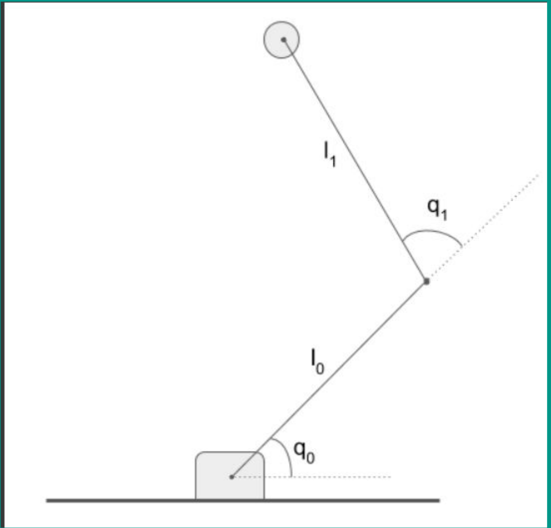
Computing Forward Kinematics

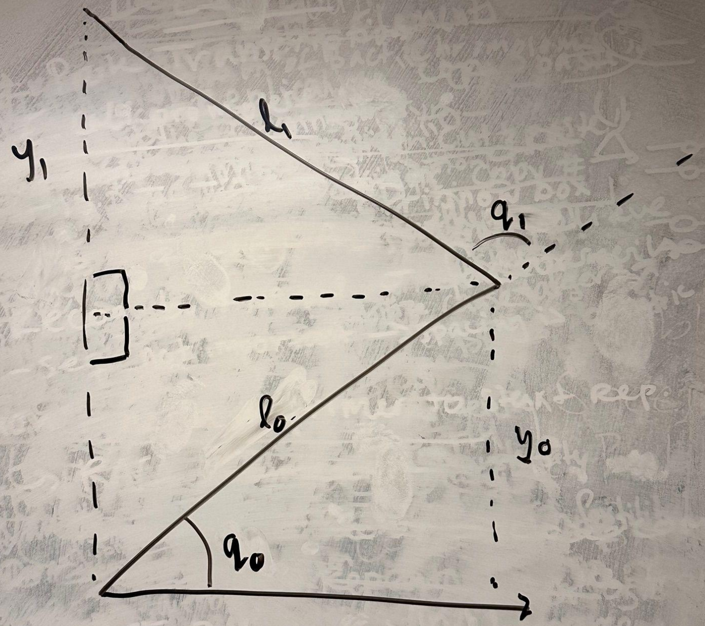
Given the joint angles and the length of each part of the arm, we derived a formula using trigonometry for the end effector position.

```python
def getForwardModel(q0, q1):
    # WRITE CODE HERE
    x = (l0 * np.cos(q0)) + (l1 * np.cos(q0+q1))
    y = (l0 * np.sin(q0)) + (l1 * np.sin(q0+q1))

    return np.array([x,y])
```

$$x = l0 * cos(q0) + l1 * cos(q0 + q1)$$

$$y = l0 * sin(q0) + l1 * sin(q0 + q1)$$




y_1
l_1
q_1
l_0
y_0
q_0

Jacobian Calculation

- The change in position due to the input joint angles
- Calculated from the partial derivatives of Forward Kinematics

$$\frac{dx}{dq_0} = -1 * l0 * sin(q0) - l1 * sin(q0 + q1)$$

$$\frac{dx}{dq_1} = -l1 * sin(q0 + q1)$$

$$\frac{dy}{dq_0} = l0 * cos(q0) + l1 * cos(q0 + q1)$$

$$\frac{dy}{dq_1} = l1 * cos(q0 + q1)$$

Python Development for Jacobian Calculation

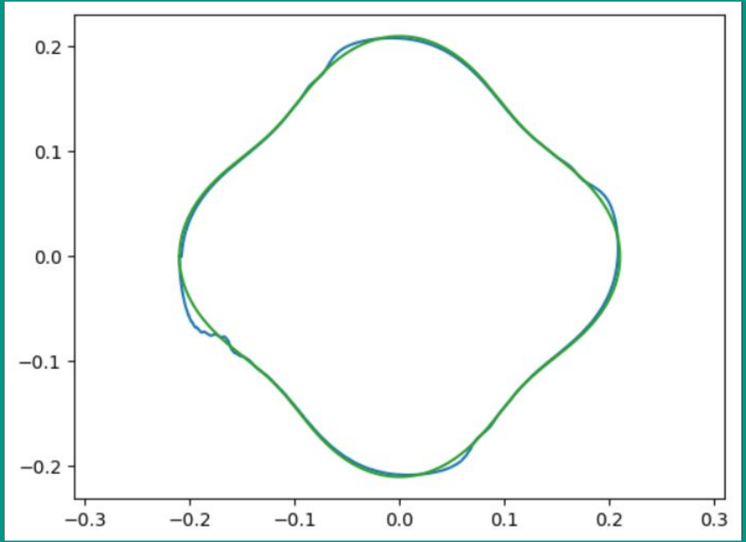
```python
def getJacobian(q0, q1):
    def x(q0, q1):
        return (l0 * np.cos(q0)) + (l1 * np.cos(q0 + q1))

    def y(q0, q1):
        return (l0 * np.sin(q0)) + (l1 * np.sin(q0 + q1))

    # Calculate partial derivatives manually
    dx_dq0 = -l0 * np.sin(q0) - l1 * np.sin(q0 + q1)
    dx_dq1 = -l1 * np.sin(q0 + q1)
    dy_dq0 = l0 * np.cos(q0) + l1 * np.cos(q0 + q1)
    dy_dq1 = l1 * np.cos(q0 + q1)

    # Create the Jacobian matrix
    jacX = np.array([[dx_dq0, dx_dq1]])
    jacY = np.array([[dy_dq0, dy_dq1]])

    jacobian = np.vstack((jacX, jacY))

    return jacobian
```

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Basic Controller (PD Control)

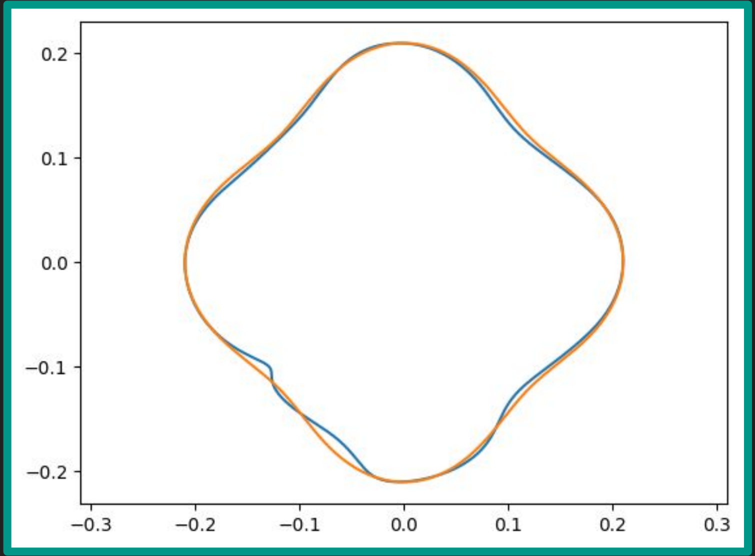
- torque-to-force equation to create a feedback loop
- Constants Kp and Kd

$$u(t) = K_p \cdot e(t) + K_d \cdot \frac{d}{dt}e(t)$$



Inverse Modular Kinematics (IK)

- Working Backwards - Output to Input
- Iterative refinement and backwards derivation based on error minimization
- Testing and tuning of predicted inputs

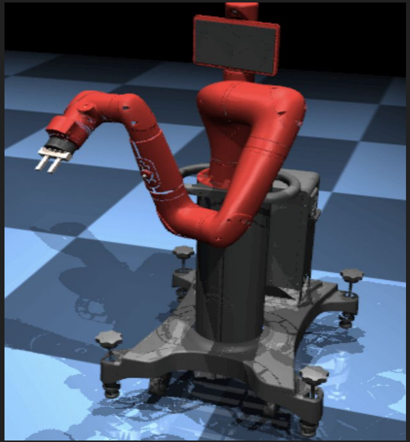


Python Code for Inverse Kinematics (IK) Controller

```
for t in range(len(traj)):
    q0, q1 = env.data.qpos[:2] # current joint angles
    desired_xy = traj[t]
    # WRITE CODE HERE
    IKq = getIK(desired_xy, ([q0, q1]))
    q0err = IKq[0] - q0
    q1err = IKq[1] - q1

    if t!= 0:
      d_q0err = q0err - last_q0err
      d_q1err = q1err - last_q1err
    else:
      d_q0err = 0
      d_q1err = 0
    tau0 = (kp*q0err + kd*d_q0err)
    tau1 = (kp*q1err + kd*d_q1err)
```

How is this related to GE:AI?

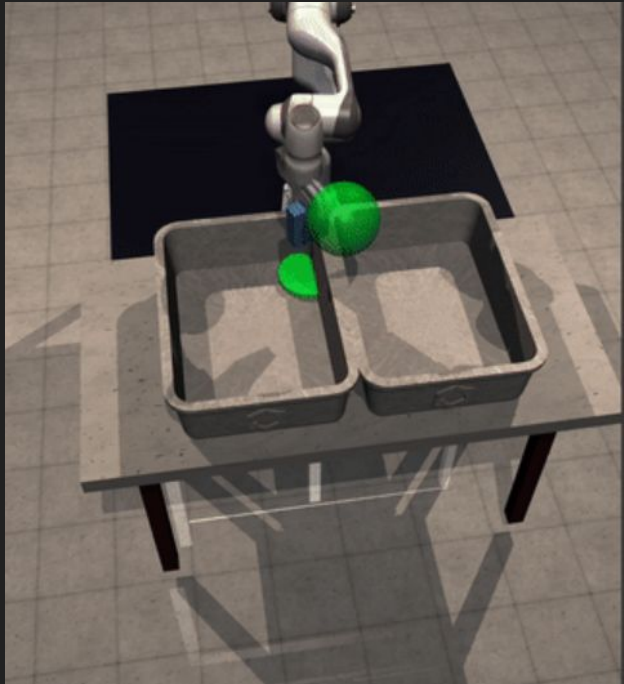
- The expectation of machine understanding has increased
- Enhancing their robotic simulations through python will speed up training
- We have learned the basics of controlling robots
- Now we will work with Reinforcement Learning tools in PyTorch.



Next Steps For Project

- Using these techniques we’ve learned like forward and inverse kinematics to move an object from point A to point B

- Will use a pick and place scenario to do this with RL.



Thank you!