Indian Institute of Technology Gandhinagar

ME 639 : Introduction to Robotics

Miniproject : 2

2R Manipulator

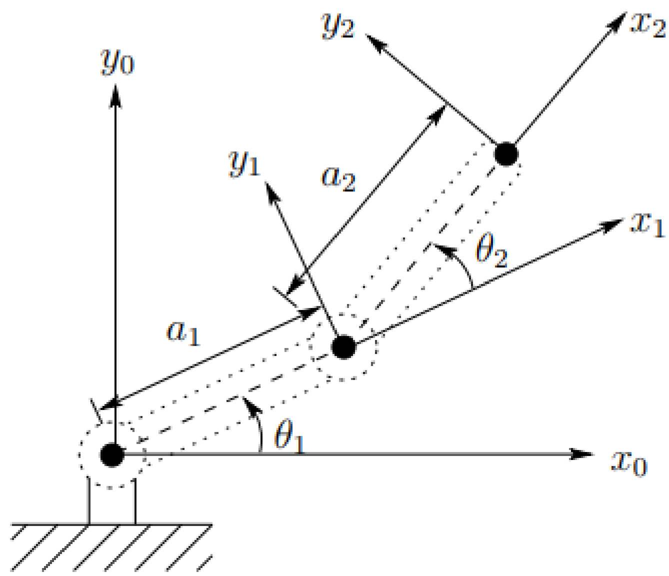
Team Members

Abhishek Mungekar - 20110005
Krish Raj - 20110160
Rohan Naika - 20110169

Under The Guidance Of

Prof. Harish P. M.

Task 0:



2R Manipulator

The length of link 1 = **105 mm**
The length of link 2 = **107 mm**
Height between links = 65 mm

The DH Table as shown below

Link	a_i	α_i	d_i	θ_i
1	105	0	0	θ_1
2	107	0	0	θ_2

We have attached the Python code for Task 0

$$H_0^1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & l_1 \cdot \cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & l_1 \cdot \sin\theta_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_1^2 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & l_2 \cdot \cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & l_2 \cdot \sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_0^2 = H_0^1 \cdot \ H_1^2$$

$$H_0^2 = \begin{bmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & l_1 \cdot \cos\theta_1 + l_2 \cdot \cos(\theta_1+\theta_2) \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & l_1 \cdot sin\theta_1 + l_2 \cdot \sin(\theta_1+\theta_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Problem Faced While Experimenting

- Pulleys were slipping during sudden motion
- Gear was slipping from the shaft of motor 2.

Python Code

```
import numpy as np

l1 = 105  # Link 1 length (mm)
l2 = 107  # Link 2 length (mm)
alpha1 = 0  # Link 1 twist (radians)
alpha2 = 0  # Link 2 twist (radians)
d1 = 0  # Link 1 offset (mm)
d2 = 0  # Link 2 offset (mm)

# Joint angles (theta_i) for a few representative configurations (in radians)
theta1 = np.pi / 4  # Configuration 1
theta2 = np.pi / 6
theta3 = np.pi / 2  # Configuration 2
theta4 = np.pi / 3

# Create DH parameter table
# Each row represents a link with parameters (a_i, alpha_i, d_i, theta_i)
```

```
dh_parameters = np.array([[l1, alpha1, d1, theta1],
                          [l2, alpha2, d2, theta2]])

# Calculate the Jacobian matrix
def calculate_jacobian(dh_params, joint_angles):
    num_links = dh_params.shape[0]
    jacobian = np.zeros((6, num_links))

    # Initialize transformation matrix
    T = np.identity(4)

    for i in range(num_links):
        # Extract DH parameters for the current link
        a, alpha, d, theta = dh_params[i]

        # Update the transformation matrix
        T_i = np.array([[np.cos(theta), -np.sin(theta) * np.cos(alpha), np.sin(theta) * np.sin(alpha), a
* np.cos(theta)],
                [np.sin(theta), np.cos(theta) * np.cos(alpha), -np.cos(theta) * np.sin(alpha), a *
np.sin(theta)],
                [0, np.sin(alpha), np.cos(alpha), d],
                [0, 0, 0, 1]])

        # Update the overall transformation matrix
        T = np.dot(T, T_i)

        # Calculate the linear and angular velocity components of the end-effector
        z_i = T[:3, 2]
        p_i = T[:3, 3]

        jacobian[:3, i] = np.cross(z_i, p_i)
        jacobian[3:, i] = z_i

    return jacobian

# Calculate the Jacobian matrix for the given joint angles
jacobian1 = calculate_jacobian(dh_parameters, [theta1, theta2])
jacobian2 = calculate_jacobian(dh_parameters, [theta3, theta4])

# Print DH parameter table and Jacobian matrices
```

```
print("DH Parameter Table:")
print("| Link |  a_i   |  alpha_i |  d_i | theta_i |")
for i, row in enumerate(dh_parameters):
    print( f"|  {i + 1}   | {row[0]:.2f} |   {row[1]:.2f}   | {row[2]:.2f} |  {row[3]:.2f}   |" )

print("\nJacobian Matrix for Configuration 1:")
print(jacobian1)

print("\nJacobian Matrix for Configuration 2:")
print(jacobian2)
```

Task 1:

```
// Motor control pins
const int motor1Pin1 = 14;  // Motor 1 control pin 1
const int motor1Pin2 = 27; // Motor 1 control pin 2
const int motor2Pin1 = 13; // Motor 2 control pin 1
const int motor2Pin2 = 12; // Motor 2 control pin 2

// Kinematic parameters
const float link1Length = 0.10800; // Length of link 1 (adjust as needed)
const float link2Length = 0.10765; // Length of link 2 (adjust as needed)

// Trajectory parameters
const float centerX = 0.0; // X-coordinate of the center of the circle
const float centerY = 0.0; // Y-coordinate of the center of the circle
const float radius = 5.0;  // Radius of the circular path (adjust as needed)
const float angularVelocity = 0.001; // Angular velocity (adjust as needed)

// Variables
float angle = 0.0; // Current angle

void setup() {
  // Initialize motor control pins as outputs
  pinMode(motor1Pin1, OUTPUT);
```

```
  pinMode(motor1Pin2, OUTPUT);
  pinMode(motor2Pin1, OUTPUT);
  pinMode(motor2Pin2, OUTPUT);

  // Set initial position
  float initialX = centerX + radius;
  float initialY = centerY;
  moveManipulator(initialX, initialY);

  // Initialize serial communication
  Serial.begin(9600);
}

void loop() {
  // Calculate the next position on the circular path
  float nextX = centerX + radius * cos(angle);
  float nextY = centerY + radius * sin(angle);

  // Move to the next position
  moveManipulator(nextX, nextY);

  // Increase the angle
  angle += angularVelocity;

  // Wrap angle within 0 to 2*PI radians (360 degrees)
  if (angle >= 2 * PI) {
    angle = 0;
  }

  // Delay to control the speed of motion
  delay(50);
}

// Move the manipulator to the specified end-tip position
void moveManipulator(float x, float y) {
  // Implement your kinematics and motor control logic here
  // Calculate motor angles and control the motors to move to the desired position

  // For simplicity, you can use forward kinematics to approximate motor angles
```

```
  float theta1 = atan2(y, x);
  float theta2 = acos((x * x + y * y - link1Length * link1Length - link2Length * link2Length) / (2 *
link1Length * link2Length));

  // Convert angles to degrees
  float angle1Degrees = degrees(theta1);
  float angle2Degrees = degrees(theta2);

  // Implement motor control logic here (e.g., using PWM for speed control)
  // Example:
  analogWrite(motor1Pin1, map(angle1Degrees, 0, 50, 0, 40));
  analogWrite(motor1Pin2, LOW);
  analogWrite(motor2Pin1, map(angle2Degrees, 0, 50, 0, 40));
  analogWrite(motor2Pin2, LOW);
}
```

Task 2:

Trail codes:

```
// Motor and encoder pins
const int motor1Pin1 = 14;
const int motor1Pin2 = 27;
const int motor2Pin1 = 12;
const int motor2Pin2 = 13;
const int encoderPinA = 25;
const int encoderPinB = 26;

// Motor PWM values
const int motor1PWM = 40;
const int motor2PWM = 40;

// Encoder variables
volatile long encoderCount = 0;
volatile int encoderState = 0;
int lastEncoderState = 0;
```

```
// Constants for encoder resolution and gear ratio
const float encoderResolution = 360.0 / 4096.0; // Degrees per encoder count
//const float gearRatio = 10.0; // Gear ratio of the motor

// Desired position on the rigid surface
const long desiredPosition = 5000; // Adjust as needed

// Force control parameters (adjust as needed)
const float forceThreshold = 100.0; // Force threshold for applying force
const int forceDirection = 1; // 1 for applying force in one direction, -1 for the opposite

void setup() {
  // Initialize motor control pins
  pinMode(motor1Pin1, OUTPUT);
  pinMode(motor1Pin2, OUTPUT);
  pinMode(motor2Pin1, OUTPUT);
  pinMode(motor2Pin2, OUTPUT);

  // Initialize encoder pins
  pinMode(encoderPinA, INPUT_PULLUP);
  pinMode(encoderPinB, INPUT_PULLUP);

  // Attach interrupts for encoder
  attachInterrupt(digitalPinToInterrupt(encoderPinA), updateEncoder, CHANGE);
  attachInterrupt(digitalPinToInterrupt(encoderPinB), updateEncoder, CHANGE);

  // Set initial PWM values for the motors
  analogWrite(motor1Pin1, motor1PWM);
  analogWrite(motor2Pin1, motor2PWM);
}

void loop() {
  // Read current encoder counts
  int encoderCounts = encoderCount;

  // Calculate error in position
  long positionError = desiredPosition - encoderCounts;

  // Implement force control logic
```

```
  if (abs(positionError) > forceThreshold) {
    // Apply force in the specified direction
    int motorSpeed = forceDirection * motor1PWM;

    // Set the motor PWM values
    analogWrite(motor1Pin1, motorSpeed);
    analogWrite(motor2Pin1, motorSpeed);
  } else {
    // Stop applying force when the desired position is reached
    analogWrite(motor1Pin1, 0);
    analogWrite(motor2Pin1, 0);
  }
}

void updateEncoder() {
  int aState = digitalRead(encoderPinA);
  int bState = digitalRead(encoderPinB);

  if (aState != lastEncoderState) {
    if (bState != aState) {
      encoderCount++;
    } else {
      encoderCount--;
    }
  } else {
    if (bState == aState) {
      encoderCount++;
    } else {
      encoderCount--;
    }
  }
  lastEncoderState = aState;
}
```

Task 3

Trail codes:

```
// Motor and encoder pins
const int motor1Pin1 = 14;
const int motor1Pin2 = 27;
const int motor2Pin1 = 12;
const int motor2Pin2 = 13;
const int encoderPinA = 25;
const int encoderPinB = 26;

// Motor PWM values
const int motor1PWM = 40;
const int motor2PWM = 40;

// Encoder variables
volatile long encoderCount = 0;
volatile int encoderState = 0;
int lastEncoderState = 0;

// Constants for encoder resolution and gear ratio
const float encoderResolution = 360.0 / 4096.0; // Degrees per encoder count
// const float gearRatio = 10.0; // Gear ratio of the motor

// Desired center position (xo, yo) in encoder counts
const long xo = 5000; // Adjust as needed
const long yo = 5000; // Adjust as needed

// PID control gains
const float kp = 0.1; // Proportional gain
const float ki = 0.01; // Integral gain
const float kd = 0.001; // Derivative gain

// PID control variables
float integralErrorX = 0;
float integralErrorY = 0;
float lastErrorX = 0;
float lastErrorY = 0;

void setup() {
```

```
  // Initialize motor control pins
  pinMode(motor1Pin1, OUTPUT);
  pinMode(motor1Pin2, OUTPUT);
  pinMode(motor2Pin1, OUTPUT);
  pinMode(motor2Pin2, OUTPUT);

  // Initialize encoder pins
  pinMode(encoderPinA, INPUT_PULLUP);
  pinMode(encoderPinB, INPUT_PULLUP);

  // Attach interrupts for encoder
  attachInterrupt(digitalPinToInterrupt(encoderPinA), updateEncoder, CHANGE);
  attachInterrupt(digitalPinToInterrupt(encoderPinB), updateEncoder, CHANGE);

  // Set initial PWM values for the motors
  analogWrite(motor1Pin1, motor1PWM);
  analogWrite(motor2Pin1, motor2PWM);
}

void loop() {
  // Read current encoder counts
  int encoderCounts = encoderCount;

  // Calculate errors in both x and y axes
  long xError = xo - encoderCounts;
  long yError = yo - encoderCounts;

  // Calculate PID control outputs
  float pidOutputX = kp * xError + ki * integralErrorX + kd * (xError - lastErrorX);
  float pidOutputY = kp * yError + ki * integralErrorY + kd * (yError - lastErrorY);

  // Update integral errors and last errors
  integralErrorX += xError;
  integralErrorY += yError;
  lastErrorX = xError;
  lastErrorY = yError;

  // Calculate motor speeds based on PID outputs
  int motor1Speed = motor1PWM + int(pidOutputX);
```

```
  int motor2Speed = motor2PWM + int(pidOutputY);

  // Set the motor PWM values
  analogWrite(motor1Pin1, motor1Speed);
  analogWrite(motor2Pin1, motor2Speed);
}

void updateEncoder() {
  int aState = digitalRead(encoderPinA);
  int bState = digitalRead(encoderPinB);

  if (aState != lastEncoderState) {
    if (bState != aState) {
      encoderCount++;
    } else {
      encoderCount--;
    }
  } else {
    if (bState == aState) {
      encoderCount++;
    } else {
      encoderCount--;
    }
  }
  lastEncoderState = aState;
}
```