problem

April 29, 2024

1 Numeric Solution of an Inverse Kinematic Problem using the Newton Method

Inverse kinematics is a subfield of robotics and computer graphics that focuses on computing the joint parameters necessary to move a robotic arm or character to a particular position. It's called "inverse" because it reverses the direction of the kinematic equations, which usually go from joint angles to end-effector position.

The Newton method, also known as Newton-Raphson method, is a powerful technique for solving equations numerically. It's an iterative method that starts with an initial guess and then refines that guess using the derivative of the function.

In this notebook, we will apply the Newton method to solve an inverse kinematic problem. We will start by defining the problem and the method, and then we will implement the method in Python and use it to solve a specific problem.

Please note that a good understanding of calculus, linear algebra, and robotics kinematics is required to fully understand the content of this notebook.

2 Jacobian for a 2-link (RR) Robot Arm

The Jacobian matrix is a crucial concept in the field of robotics, particularly for manipulator arms. It provides a relationship between the joint velocities and the end-effector velocity in the Cartesian space.

For a 2-link (RR) robot arm, the Jacobian matrix can be derived from the forward kinematics equations. The forward kinematics equations for the end-effector position (x, y) in terms of the joint angles (θ_1, θ_2) and link lengths (l_1, l_2) are:

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

The Jacobian matrix J is the matrix of all first-order partial derivatives of the forward kinematics function. For our 2-link robot arm, it is a 2x2 matrix:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix}$$

By differentiating the forward kinematics equations with respect to the joint angles, we get the elements of the Jacobian matrix:

$$J = \begin{bmatrix} -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2) & -l_2 \sin(\theta_1 + \theta_2) \\ l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2) & l_2 \cos(\theta_1 + \theta_2) \end{bmatrix}$$

The Jacobian matrix is used in the Newton-Raphson method to relate the change in joint angles to the error in the end-effector position.

[1]:
```
import numpy as np

def forward_kinematics(theta1, theta2, l1, l2):
    x = l1 * np.cos(theta1) + l2 * np.cos(theta1 + theta2)
    y = l1 * np.sin(theta1) + l2 * np.sin(theta1 + theta2)
    return x, y

def jacobian(theta1, theta2, l1, l2):
    j11 = -l1 * np.sin(theta1) - l2 * np.sin(theta1 + theta2)
    j12 = -l2 * np.sin(theta1 + theta2)
    j21 = l1 * np.cos(theta1) + l2 * np.cos(theta1 + theta2)
    j22 = l2 * np.cos(theta1 + theta2)
    return np.array([[j11, j12], [j21, j22]])

def inverse_kinematics(target, theta, l1, l2, epsilon=1e-10,␣
 ↪max_iterations=1000):
    for i in range(max_iterations):
        current = forward_kinematics(theta[0], theta[1], l1, l2)
        error = target - current
        if np.linalg.norm(error) < epsilon:
            break
        delta_theta = np.linalg.solve(jacobian(theta[0], theta[1], l1, l2),␣
 ↪error)
        theta += delta_theta
    return theta

# Test the function
target_position = np.array([1.5, 1.5])
initial_angles = np.array([0.1, 0.1])
link_lengths = [2, 1]

angles = inverse_kinematics(target_position, initial_angles, *link_lengths)
print(f"Computed angles: {angles}")
```

```
Computed angles: [-5.9844821  20.54568008]
```

3 Numeric Solution of an Inverse Kinematic Problem using the Gradient Descent Method

Inverse kinematics is a crucial aspect of robotics and computer graphics that focuses on computing the joint parameters necessary to move a robotic arm or character to a specific position. It's called "inverse" because it reverses the direction of the kinematic equations, which usually go from joint angles to end-effector position.

The Gradient Descent method is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent.

In this notebook, we will apply the Gradient Descent method to solve an inverse kinematic problem. We will start by defining the problem and the method, and then we will implement the method in Python and use it to solve a specific problem.

Please note that a good understanding of calculus, linear algebra, and robotics kinematics is required to fully understand the content of this notebook.

3.1 Gradient Descent Method

The general equation for the Gradient Descent method is:

$$\theta_{new} = \theta_{old} - \alpha \nabla F(\theta_{old})$$

where: - θ_{new} is the updated value of θ - θ_{old} is the current value of θ - α is the learning rate - $\nabla F(\theta_{old})$ is the gradient of the function F at θ_{old}

The gradient of a function at a point is a vector that points in the direction of the steepest increase of the function. The negative gradient points in the direction of the steepest decrease. Therefore, subtracting the gradient from the current point moves us towards the local minimum of the function.

[2]:
```
import numpy as np

def forward_kinematics(theta1, theta2, l1, l2):
    x = l1 * np.cos(theta1) + l2 * np.cos(theta1 + theta2)
    y = l1 * np.sin(theta1) + l2 * np.sin(theta1 + theta2)
    return np.array([x, y])

def gradient(theta1, theta2, l1, l2, target):
    current = forward_kinematics(theta1, theta2, l1, l2)
    error = current - target
    grad1 = 2 * error[0] * (-l1 * np.sin(theta1) - l2 * np.sin(theta1 +␣
 ↪theta2)) + 2 * error[1] * (l1 * np.cos(theta1) + l2 * np.cos(theta1 +␣
 ↪theta2))
    grad2 = 2 * error[0] * (-l2 * np.sin(theta1 + theta2)) + 2 * error[1] * (l2␣
 ↪* np.cos(theta1 + theta2))
    return np.array([grad1, grad2])
```

```
def inverse_kinematics(target, theta, l1, l2, alpha=0.01, epsilon=1e-10,␣
 ↪max_iterations=1000):
    for i in range(max_iterations):
        grad = gradient(theta[0], theta[1], l1, l2, target)
        if np.linalg.norm(grad) < epsilon:
            break
        theta = theta - alpha * grad
    return theta

# Test the function
target_position = np.array([1.0, 1.5])
initial_angles = np.array([0.1, 0.1])
link_lengths = [2, 1]

angles = inverse_kinematics(target_position, initial_angles, *link_lengths)
print(f"Computed angles: {angles}")
```

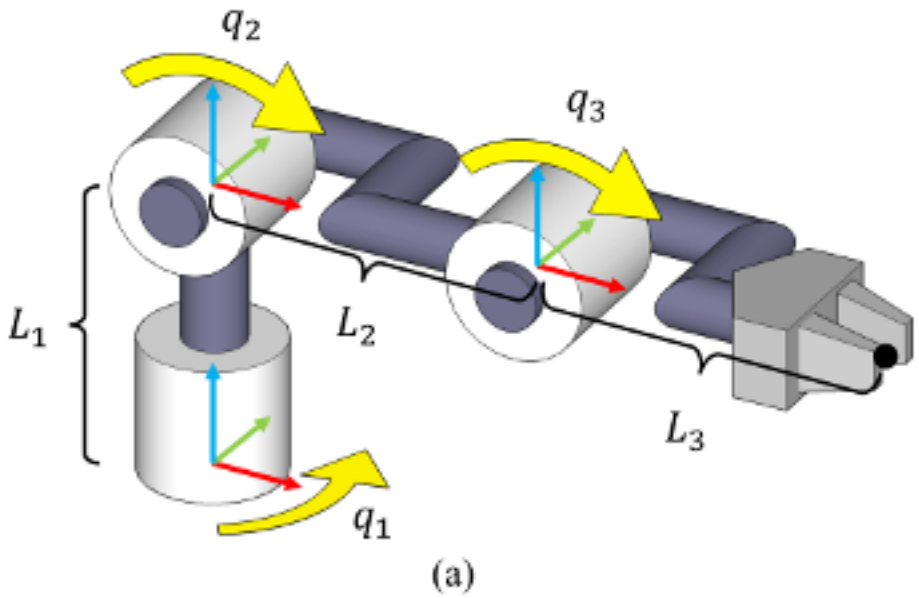
```
Computed angles: [0.46058393 2.0236129 ]
```

[3]:
```
forward_kinematics(0.4605, 2.02, 2, 1)
```

[3]:
```
array([1.00233899, 1.50277183])
```

3.2 Homework

Develop the inverse kinematics problem with the numeric methods given in this notebook (Newton and Gradient Descent) for the RRR robot shown below:



(a)

Give some numeric values for the lengths and some desired positions.

[ ]: