Python Code Explanation: Forward Kinematics

Viktor

December 10, 2024

1 Introduction

This document explains a Python implementation of a `Robot` class. The class models a robot's position, orientation, and Denavit-Hartenberg (DH) parameters to calculate its forward kinematics (FK).

2 Code Breakdown

2.1 Class Definition

The `Robot` class is initialized with a base position, base orientation, and DH parameters.

Listing 1: Robot Class Definition

```python
import numpy as np

class Robot:
    """
    A class to represent a robot with base position,
        rotation, and Denavit-Hartenberg parameters.
    """
    def __init__(self, base_position=[0, 0, 0],
        base_orientation=[0, 0, 0, 1]):
        """
        Initialize the Robot.

        Args:
            base_position (list): The base position of the
                robot [x, y, z].
            base_orientation (list): The base rotation of
                the robot specified as a quaternion.
        """
        self.base_position = np.array(base_position)
        self.base_rotation = self.
            quaternion_to_rotation_matrix(base_orientation)
        self.dh_params = [
            {"a": 0, "d": 0.4, "alpha": np.pi/2, "theta": np.pi/2},
            {"a": 0.4, "d": 0, "alpha": -np.pi/2, "theta": 0},
            {"a": 0, "d": 0, "alpha": -np.pi/2, "theta": -np.pi/2},
            {"a": 0, "d": 0, "alpha": np.pi/2, "theta": 0}
        ]
```

Explanation: The constructor initializes:

- `base_position`: A 3D position vector $[x, y, z]$.
- `base_rotation`: A rotation matrix computed from a quaternion.
- `dh_params`: A list of Denavit-Hartenberg parameters for each link of the robot.

2.2 Quaternion to Rotation Matrix

The quaternion-to-rotation matrix conversion is implemented in the `quaternion_to_rotation_matrix` method.

Listing 2: Quaternion to Rotation Matrix

```python
def quaternion_to_rotation_matrix(self, q):
    """
    Convert a quaternion [x, y, z, w] to a 3x3 rotation matrix.

    Args:
        q (list or np.array): Quaternion [x, y, z, w].

    Returns:
        np.array: A 3x3 rotation matrix.
    """
    x, y, z, w = q
    return np.array([
        [1 - 2*(y**2 + z**2), 2*(x*y - z*w), 2*(x*z + y*w)],
        [2*(x*y + z*w), 1 - 2*(x**2 + z**2), 2*(y*z - x*w)],
        [2*(x*z - y*w), 2*(y*z + x*w), 1 - 2*(x**2 + y**2)]
    ])
```

Explanation: This method computes a rotation matrix from the quaternion representation $[x, y, z, w]$ using mathematical formulas.

2.3 Homogeneous Transformation

The `homogeneous_transform` method calculates the DH transformation matrix.

Listing 3: Homogeneous Transformation

```
@staticmethod
def homogeneous_transform(a, d, alpha, theta):
    """
    Compute the Denavit-Hartenberg transformation matrix.

    Args:
        a (float): Link length.
        d (float): Link offset.
        alpha (float): Link twist.
        theta (float): Joint angle.

    Returns:
        np.array: The 4x4 homogeneous transformation matrix.
    """
    ct = np.cos(theta)
    st = np.sin(theta)
    ca = np.cos(alpha)
    sa = np.sin(alpha)
    return np.array([
        [ct, -st*ca, st*sa, a*ct],
        [st, ct*ca, -ct*sa, a*st],
        [0, sa, ca, d],
        [0, 0, 0, 1]
    ])
```

Explanation: This method generates the 4x4 transformation matrix based on the DH parameters.

2.4 Forward Kinematics

The `calc_fk` method calculates the forward kinematics using the base transformation and DH parameters.

Listing 4: Forward Kinematics

```
def calc_fk(self, q):
    """
    Calculate the forward kinematics of the robot.

    Returns:
        np.array: The overall transformation matrix from
            world frame to the end effector.
    """
    T_base = np.eye(4)
    T_base[:3, 3] = self.base_position
    T_base[:3, :3] = self.base_rotation
    transform = T_base

    for i, params in enumerate(self.dh_params):
        if i <= 2:
            T_i = self.homogeneous_transform(params["a"],
                params["d"], params["alpha"], params["theta"]
                 + q[i])
        else:
            T_i = self.homogeneous_transform(params["a"],
                params["d"] + q[i], params["alpha"], params["
                theta"])
        transform = transform @ T_i
    return np.round(transform, 3)
```

Explanation:

- Combines base transformation and DH transformations iteratively.
- Supports revolute and prismatic joints.

3 Conclusion

The entire forward kinematics calculation is encapsulated within the `Robot` class to ensure it is specific to the robot it represents. The computation is modular, with different components of the calculation implemented in internal methods of the `Robot` class. These methods come together in the `calc_fk` function to produce the overall forward kinematics transformation matrix.
The modular design of the `Robot` class ensures flexibility, making it adaptable for various robotic applications. The actual application of the forward kinematics calculation is defined in the `__main__.py` file. This file serves as a terminal-based program that imports the `Robot` class from the `utils.py` module and provides a user interface for calculating forward kinematics.
Below is the implementation of the `__main__.py` program:

Listing 5: Forward Kinematics Application

```
import numpy as np
from .utils import Robot

def main():
    """
    Terminal program to calculate the forward kinematics of
        the robot.
    """

    # Create the Robot instance
    robot = Robot()

    print("\nWelcome to the Robot Forward Kinematics
        Calculator!")

    while True:
        print("\nMenu:")
        print("1. Input joint values (theta1 to theta4) and calculate FK")
        print("2. Exit")

        choice = input("\nEnter your choice: ").strip()

        if choice == "1":
            try:
                # Prompt user for joint values
                theta1 = float(input("Enter theta1 (in radians): "))
                theta2 = float(input("Enter theta2 (in radians): "))
                theta3 = float(input("Enter theta3 (in radians): "))
                theta4 = float(input("Enter theta4 (prismatic joint displacement): "))

                # Calculate forward kinematics
                q = [theta1, theta2, theta3, theta4]
                fk_transform = robot.calc_fk(q)

                # Display the result
                print("\nForward Kinematics Transformation Matrix:\n")
                print(fk_transform)

            except ValueError:
                print("\nInvalid input. Please enter numerical values for the joint angles.\n")
        elif choice == "2":
            print("\nExiting the program. Goodbye!\n")
            break
        else:
            print("\nInvalid choice. Please select 1 or 2.\n")

if __name__ == "__main__":
    main()
```

This implementation allows users to interactively input joint values, compute the forward kinematics, and view the resulting transformation matrix.