XArm6 Robot Simulation

Manan Anjaria (mha9531)
MS Mechatronics and Robotics
New York University
Submitted to: Professor William Peng

12/19/2024

Abstract

This report provides an in-depth analysis of a Python implementation for controlling and simulating an XArm6 robot. The implementation encompasses the complete robotics pipeline, from basic matrix operations to advanced trajectory optimization, analyzes each component's mathematical foundations, implementation details, and practical considerations, with special attention to extra credit features including workspace computation, inverse dynamics, and trajectory planning. The code also has extensive comments as it is big and diffcult to navigate

Contents

1 Mathematical Foundations

1.1 Linear Algebra Fundamentals

As per instruction no Libary has been used to assist in the Maths has been utilized , including Numpy

1.1.1 Matrix Determinant

For a square matrix A, the determinant is computed recursively:

$$\det(A) = \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \det(M_{1j}) \tag{1}$$

where M_{ij} is the minor matrix obtained by removing row i and column j.

Implementation:

```
def determinant(matrix):
    if len(matrix) == 1:
        return matrix[0][0]
    if len(matrix) == 2:
        return matrix[0][0]*matrix[1][1] - matrix[0][1]*matrix[1][0]

    det = 0
    for c in range(len(matrix)):
        minor = [row[:c] + row[c+1:] for row in matrix[1:]]
        det += ((-1)**c) * matrix[0][c] * determinant(minor)
    return det
```

1.1.2 Matrix Inverse

For a non-singular matrix A, the inverse A^{-1} is computed using Gaussian elimination with the following properties:

$$AA^{-1} = A^{-1}A = I \tag{2}$$

The implementation uses the Gaussian elimination method:

Algorithm 1 Matrix Inverse via Gaussian Elimination

```
**Input:** Square matrix $A$
**Output:** Inverse matrix $A^{-1}$
Create augmented matrix $[A|I]$
for $i = 1$ to $n$ do
  Find pivot
  Normalize row $i$
  for $j = 1$ to $n$, $j \neq i$ do
    Eliminate entry $(j, i)$
  end for
end for
return Right half of augmented matrix
```

Implementation details:

```
def matrix_inverse(matrix):
    n = len(matrix)
    augmented = [row[:] + [1 if i == j else 0
                 for j in range(n)] for i, row in enumerate(matrix)]

    for i in range(n):
        pivot = augmented[i][i]
        if pivot == 0:
            raise ValueError("Matrix is singular")

        # Normalize row
        augmented[i] = [elem / pivot for elem in augmented[i]]
        # Eliminate column
        for j in range(n):
            if j != i:
                factor = augmented[j][i]
                augmented[j] = [augmented[j][k] -
                               factor * augmented[i][k]
                               for k in range(2*n)]

    return [row[n:] for row in augmented]
```

1.1.3 Pseudo-Inverse

For a non-square matrix A, the Moore-Penrose pseudo-inverse A^+ is defined as:

$$A^+ = (A^T A)^{-1} A^T \tag{3}$$

Properties of the pseudo-inverse:

$$\begin{aligned} AA^+A &= A \\ A^+AA^+ &= A^+ \\ (AA^+)^T &= AA^+ \\ (A^+A)^T &= A^+A \end{aligned}$$

Implementation:

```
def pseudo_inverse(matrix):
    A_T = transpose(matrix)
    A_TA = matrix_multiply(A_T, matrix)
    A_TA_inv = matrix_inverse(A_TA)
    A_pseudo_inv = matrix_multiply(A_TA_inv, A_T)
    return A_pseudo_inv
```

2 Matrix Operations Implementation

2.1 Matrix Transpose

The fundamental matrix transpose operation is implemented as:

Algorithm 2 Matrix Transpose

```
Input: Matrix $A \in \mathbb{R}^{m \times n}$
Output: Matrix $A^T \in \mathbb{R}^{n \times m}$
Initialize empty matrix $B$
for $i = 1$ to $n$ do
    Initialize empty row
    for $j = 1$ to $m$ do
        $B_{ij} = A_{ji}$
    end for
end for
return $B$
```

Implementation analysis:

```python
def transpose(matrix):
    transposed = []
    for i in range(len(matrix[0])):
        transposed_row = []
        for row in matrix:
            transposed_row.append(row[i])
        transposed.append(transposed_row)
    return transposed
```

2.2 Matrix Multiplication

Matrix multiplication follows the standard mathematical definition:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{4}$$

Implementation:

```python
def matrix_multiply(a, b):
    result = []
    for i in range(len(a)):
        result_row = []
        for j in range(len(b[0])):
            sum_elements = 0
            for k in range(len(b)):
                sum_elements += a[i][k] * b[k][j]
            result_row.append(sum_elements)
        result.append(result_row)
    return result
```

2.3 Cross Product

The cross product implementation for 3D vectors follows:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} \tag{5}$$

3 Robot Kinematics

3.1 Denavit-Hartenberg Parameters

The DH parameters for the XArm6 robot define the following transformations:

$$T_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

XArm6 specific parameters:

Joint	θ	a (mm)	d (mm)	α (rad)
1	θ_1	0	267	$-\pi/2$
2	$\theta_2 + \text{offset}$	290	0	0
3	$\theta_3 + \text{offset}$	77.5	0	$-\pi/2$
4	θ_4	0	343	$\pi/2$
5	θ_5	76	0	$-\pi/2$
6	θ_6	0	97	0

Table 1: DH Parameters for XArm6

3.2 Forward Kinematics

The forward kinematics implementation combines DH transformations:

$$T_0^6 = T_0^1 T_1^2 T_2^3 T_3^4 T_4^5 T_5^6 \tag{7}$$

Implementation details:

```
def compute_forward_kinematics(self):
    dh_parameters = [
        [self.joint_angles_rad[0], 0, 267, -math.pi/2],
        [self.joint_angles_rad[1] + self.T2_offset, 290, 0, 0],
        [self.joint_angles_rad[2] + self.T3_offset, 77.5, 0, -math.pi/2
        [self.joint_angles_rad[3], 0, 343, math.pi/2],
        [self.joint_angles_rad[4], 76, 0, -math.pi/2],
        [self.joint_angles_rad[5], 0, 97, 0]
    ]
```

3.3 Jacobian Matrix

The Jacobian matrix combines linear and angular velocity components:

$$J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} = \begin{bmatrix} z_i \times (p_e - p_i) \\ z_i \end{bmatrix} \tag{8}$$

Implementation analysis:

```
def compute_jacobian(self):
    J = [[0 for _ in range(6)] for _ in range(6)]
    end_pos = self.get_end_effector_position()

    for i in range(6):
        zi = self.joint_axes[i]
        pi = self.joint_positions[i]
        end_minus_pi = [end_pos[j] - pi[j] for j in range(3)]
        Jp = cross_product(zi, end_minus_pi)
        Jo = zi
```

4 Inverse Kinematics

4.1 Inverse Kinematics Methods

4.1.1 Newton-Raphson Method (Classical)

The classical Newton-Raphson method for IK would use:

$$\Delta\theta = J^{-1}e \tag{9}$$

Where:

- J is the Jacobian matrix
- e is the position error
- $\Delta\theta$ is the joint angle update

However, this method has limitations:

- Requires square, invertible Jacobian
- Unstable near singularities
- Can have convergence issues

4.1.2 Implemented Method: Damped Least Squares

Instead, the implementation uses the Damped Least Squares method (also known as Levenberg-Marquardt):

$$\Delta\theta = (J^T J + \lambda^2 I)^{-1} J^T e \tag{10}$$

Where:

- λ is the damping factor (set to 0.1 in the code)
- e is the position error
- I is the identity matrix

Implementation details:

```
def inverse_kinematics(self, target_pos, max_iterations=1000,
                       threshold=0.01, damping=0.1):
    for iteration in range(max_iterations):
        # Compute current position error
        current_pos = self.get_end_effector_position()
        position_error = [target_pos[i] - current_pos[i]
                          for i in range(3)]
        error_norm = math.sqrt(sum([error**2
                                    for error in position_error]))

        if error_norm < threshold:
            return self.joint_angles_deg, error_history

        # Get Jacobian and extract velocity part
        J = self.get_jacobian()
        J_velocity = [row[:6] for row in J[:3]]

        # Compute J^T * J +  ^2 * I
        J_T = transpose(J_velocity)
        J_TJ = matrix_multiply(J_T, J_velocity)
        for i in range(len(J_TJ)):
            J_TJ[i][i] += damping ** 2

        # Compute update step
        try:
            J_TJ_inv = matrix_inverse(J_TJ)
        except ValueError:
            break

        J_pseudo_inverse = matrix_multiply(J_TJ_inv, J_T)
        delta_theta_rad = [0 for _ in range(6)]
        for i in range(6):
            for j in range(3):
                delta_theta_rad[i] += J_pseudo_inverse[i][j] * position
```

Advantages of Damped Least Squares:

- More stable near singularities

- Works with non-square Jacobians
- Better numerical conditioning
- Smoother motion

The damping factor λ provides a trade-off between:

- Accuracy (small λ)
- Stability (large λ)

Error convergence is monitored using:

$$\|e\| = \sqrt{\sum_{i=1}^{3}(target_i - current_i)^2} \tag{11}$$

The algorithm terminates when either:

- $\|e\| < threshold$ (success)
- Maximum iterations reached
- Singular configuration encountered

Implementation:

```
def inverse_kinematics(self, target_pos, max_iterations=1000,
                        threshold=0.01, damping=0.1):
    for iteration in range(max_iterations):
        current_pos = self.get_end_effector_position()
        position_error = [target_pos[i] - current_pos[i]
                         for i in range(3)]
        error_norm = math.sqrt(sum([error**2
                                   for error in position_error]))
```

5 Workspace Analysis(Extra Credit)

5.1 Monte Carlo Sampling

The workspace is analyzed using random sampling:

Algorithm 3 Workspace Analysis

```
Input: Number of samples $N$, Joint limits $[\theta_{min}, \theta_{max}]$
Output: Set of reachable points $P$
for $i = 1$ to $N$ do
    Generate random joint angles within limits
    Compute forward kinematics
    Store end-effector position
end for
Plot points in 3D space
```

6 Advanced Joint Control

6.1 Joint Limits Management

Implementation of joint limits enforcement:

$$\theta_i = \begin{cases} \theta_{min,i} & \text{if } \theta_i < \theta_{min,i} \\ \theta_{max,i} & \text{if } \theta_i > \theta_{max,i} \\ \theta_i & \text{otherwise} \end{cases} \tag{12}$$

Implementation:

```
def enforce_joint_limits(self):
    for i in range(len(self.joint_angles_deg)):
        min_angle, max_angle = self.joint_limits_deg[i]
        if self.joint_angles_deg[i] < min_angle:
            self.joint_angles_deg[i] = min_angle
        elif self.joint_angles_deg[i] > max_angle:
            self.joint_angles_deg[i] = max_angle
```

6.2 Adjoint Transformation(Extra Credit)

The adjoint transformation matrix is defined as:

$$Ad_T = \begin{bmatrix} R & 0 \\ [p]_\times R & R \end{bmatrix} \tag{13}$$

where $[p]_\times$ is the skew-symmetric matrix:

$$[p]_\times = \begin{bmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{bmatrix} \tag{14}$$

Implementation:

```
@staticmethod
def adjoint(transformation_matrix):
    R = [row[:3] for row in transformation_matrix[:3]]
    p = [transformation_matrix[i][3] for i in range(3)]
    p_skew = [
        [0, -p[2], p[1]],
        [p[2], 0, -p[0]],
        [-p[1], p[0], 0]
    ]
```

7 Trajectory Planning(Extra Credit)

7.1 Quintic Polynomial Trajectory

The quintic polynomial trajectory is defined as:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{15}$$

With boundary conditions:

$$\begin{aligned} q(0) = q_0, \quad \dot{q}(0) = 0, \quad \ddot{q}(0) = 0 \\ q(t_f) = q_f, \quad \dot{q}(t_f) = 0, \quad \ddot{q}(t_f) = 0 \end{aligned}$$

7.2 B-Spline Trajectory

The quadratic B-spline implementation uses:

$$B_{i,2}(t) = \sum_{j=0}^{2} \frac{N_{j,2}(t)}{w_j} P_j \tag{16}$$

Where $N_{j,2}(t)$ are the basis functions.

7.3 Trapezoidal Velocity Profile

The trapezoidal velocity profile consists of three phases:

1. Acceleration: $v(t) = at$
2. Constant velocity: $v(t) = v_{max}$
3. Deceleration: $v(t) = v_{max} - a(t - t_1)$

8 Inverse Dynamics(Extra Credit)

8.1 Newton-Euler Formulation

The inverse dynamics follows the Newton-Euler formulation:

$$\tau = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) \tag{17}$$

Implementation:

```
def inverse_dynamics(self, position, force):
    torque = cross_product(position, force)
    return torque
```

9 Visualization and Simulation

9.1 PyBullet Integration

The simulation environment is implemented using PyBullet:

```
def visualize_simulation(initial_joint_angles_deg,
                         final_joint_angles_deg,
                         joint_limits_deg):
    physics_client = p.connect(p.GUI)
    p.setAdditionalSearchPath(pybullet_data.getDataPath())
    robot_id = p.loadURDF(robot_urdf_path, useFixedBase=True)
```

9.2 Trajectory Visualization

Multiple plotting functions are implemented for:

- Joint trajectories over time
- Workspace visualization
- Error convergence plots
- Jacobian matrix heatmap

10 Kineostatic Duality(Extra Credit)

The implementation explores the fundamental relationship between forces and velocities in the robotic system. This duality is expressed through the Jacobian matrix and its transpose.

10.0.1 Mathematical Foundation

The kineostatic duality principle states that:

$$\begin{aligned} \text{Velocity mapping: } & V = J\dot{q} \\ \text{Force mapping: } & \tau = J^T F \end{aligned} \tag{18}$$

Where:

- $V \in \mathbb{R}^6$ is the end-effector twist (linear and angular velocity)
- $\dot{q} \in \mathbb{R}^n$ is the joint velocity vector
- $\tau \in \mathbb{R}^n$ is the joint torque vector
- $F \in \mathbb{R}^6$ is the end-effector wrench (force and moment)

10.0.2 Code Implementation

Here's the implementation of kineostatic duality calculations:

```
# Forward Kinematics Duality: Mapping forces to joint torques
applied_force = [7, 33, 5]  # Example force vector
# Create a 6-element wrench (force and zero torque)
wrench = applied_force + [0, 0, 0]

# Get Jacobian
J = robot.get_jacobian()
# Transpose J to get 6x6
J_T = transpose(J)

# Multiply J_T (6x6) with wrench (6x1) to get joint torques (6x1)
tau_duality_matrix = matrix_multiply(J_T,
                        [[wrench[i]] for i in range(6)])
# Flatten the result to a list
tau_duality = [row[0] for row in tau_duality_matrix]
print("Joint Torques from Kineostatic Duality:", tau_duality)
```

10.0.3 Implementation Details

The implementation handles several key aspects:

1. **Wrench Transformation:**

$$\begin{bmatrix} f \\ n \end{bmatrix} = \begin{bmatrix} R & 0 \\ [p]_\times R & R \end{bmatrix} \begin{bmatrix} f' \\ n' \end{bmatrix} \tag{19}$$

2. **Power Conservation:**

$$\dot{q}^T \tau = V^T F \tag{20}$$

10.0.4 Practical Applications

The implementation enables:

- Force control strategies
- Impedance control implementation

- Contact force estimation
- Joint torque computation from external forces

10.0.5 Code Example: Force Control

Note: the following code is not implemented in the main code, this is a suggestion about what can be done for force control Implementation of force control using duality:

```
def compute_force_control(self, desired_force, current_pos):
    # Get current Jacobian
    J = self.get_jacobian()
    J_T = transpose(J)

    # Compute required joint torques
    force_vector = desired_force + [0, 0, 0]  # Add moments
    torques = matrix_multiply(J_T,
                  [[f] for f in force_vector])

    return [t[0] for t in torques]
```

10.0.6 Limitations and Considerations

The implementation must handle:

- Singular configurations
- Numerical stability near singularities
- Joint limits during force control
- Friction effects

Code example for handling singularities: Note: the following code is not implemented in the main code, this is a suggestion about what can be done for force control

```
def check_singularity(self, jacobian):
    # Compute determinant of J*J^T
    JJT = matrix_multiply(jacobian, transpose(jacobian))
    det = determinant(JJT)

    # Check if near singular
    return abs(det) < 1e-6
```

10.1 Advanced Trajectory Optimization

Comparison of different trajectory types:

- Linear interpolation
- Cubic polynomial
- Quintic polynomial
- B-spline
- Bézier curves
- Trapezoidal velocity profile

11 Performance Analysis

11.1 Inverse Kinematics Convergence

Analysis of convergence properties:

- Convergence rate
- Error norm evolution
- Singularity handling

11.2 Workspace Coverage

Quantitative analysis of:

- Reachable workspace volume
- Dexterity measures
- Singularity regions

12 Future Improvements

12.1 Suggested Enhancements

Potential areas for improvement:

1. Implementation of collision avoidance
2. Advanced path planning algorithms
3. Real-time control implementation
4. Dynamic parameter identification
5. Integration with ROS framework

13 Conclusion

The implementation successfully demonstrates robotics concepts from fundamental matrix operations to advanced trajectory planning.

References

[1] Craig, J. J. (2009). Introduction to Robotics: Mechanics and Control.

[2] PyBullet Physics Simulation Documentation.

14 Fun Fact : Personal Experience with XArm6

During my hands-on experience with the XArm6 robot, I encountered an interesting incident . One day while working with the robot, the room suddenly filled with a burning scent. I immediately initiated an emergency shutdown of the robot and investigated the issue. The diagnosis revealed that Joint 4 had burned out.

This presented an interesting technical challenge: theoretically, it should have been possible to convert the XArm6 into a 5-link robot by bypassing the damaged joint. However, due to the specific implementation of Modbus communication protocol and the control feedback loop in the XArm6's architecture, this modification proved impossible. Murphy's Law demonstrates how the theoretical flexibility of robotic systems can sometimes be constrained by their practical implementation, it also set me back 10,000 USD.

Thank You, Manan Anjaria