Industrial_Manipulator_Kinematics_Simulation

RRPR Robot Arm Simulation (single-file)

This repository contains a pick-and-place RRPR robot simulation implemented in a single Python file: `CW_RRPR.py`.

Approach (Jacobian & IK):

- We compute the geometric Jacobian from the joint-to-joint transforms produced by `RRPRRobot.forward_kinematics_all`. For revolute joints the column contributions are given by linear and angular components computed from joint axes and their origins. For a prismatic joint the linear component aligns with the joint axis and the angular component is zero.
- For inverse kinematics we use a numeric pseudo-inverse method. The pseudo-inverse method computes a least-squares solution to the linearized relation between joint increments and end-effector position errors. Given the positional Jacobian `J_pos`, the pseudo-inverse `J_pos^+` provides the minimum-norm delta-q that best fits the position error in a least-squares sense. Applied iteratively (with a small step size), this update moves the end-effector toward the target and can be regularized to improve numerical stability when the Jacobian is ill-conditioned.
- Why this approach?
  - Compact and robust for mixed joint types (R, R, P, R) without lengthy symbolic derivations.
  - Easy to extend (add damping, Tikhonov regularization, null-space control, secondary objectives).
  - Handles redundancy numerically and lets us clamp or regularize joint variables (we clamp the prismatic joint and normalize revolute angles).
- Advantages of this approach:
  - **Simple & implementable:** Works directly from FK transforms and requires no symbolic derivation.
  - **Flexible & extensible:** Easy to add damping, regularization, null-space control, and secondary objectives.
  - **Numerically robust (with regularization):** Regularized pseudo-inverse handles near-singularities and redundancy stably.
  - **Maintainable:** Less brittle than hand-derived closed-form solutions and easier to adapt to new joint types or constraints.
  - **Practical:** Seeding IK with approach/contact poses and using regularized pseudo-inverse when necessary gives reliable behavior for position-only tasks.

Real-Life Applications of RRPR Configuration

1. Industrial Assembly & Manufacturing SCARA robots (Selective Compliance Assembly Robot Arm) often use RRPR-like configurations.

They are common in electronics assembly (placing chips, circuit boards) and automotive manufacturing (fastening, welding, or part transfer).

The prismatic joint allows vertical motion for pick-and-place tasks, while revolute joints handle horizontal positioning.

2. Warehousing & Logistics RRPR arms can be programmed to pick items from bins or shelves and place them elsewhere.

Example: Automated systems in libraries or storage facilities where books or packages must be retrieved and placed accurately.

3. Medical & Laboratory Robotics Used in lab automation for handling test tubes, reagents, or instruments.

The prismatic joint provides precise vertical control, reducing risk of spills or contamination.

In surgery, RRPR-style manipulators can position tools with both rotational flexibility and linear reach.

4. Drawing, Painting, and 3D Printing RRPR arms have been applied in drawing robots where the prismatic joint controls pen up/down motion while revolute joints move across the surface.

Similar logic applies to painting robots or additive manufacturing arms.

5. Inspection & Quality Control RRPR manipulators can extend into tight spaces for visual inspection or sensor placement.

Useful in aerospace and automotive industries where components are densely packed.

6. Education & Research RRPR robots are popular in robotics labs for teaching kinematics and control.

Their mixed joint types (R + P) make them ideal for demonstrating both rotational and translational motion.

🔎 Why RRPR is Useful Revolute joints (R): Provide rotational flexibility for positioning.

Prismatic joint (P): Adds linear extension/retraction, enabling reach into vertical or confined spaces.

Combination: Makes RRPR arms compact yet versatile — perfect for repetitive, high-precision tasks.

Requirements

- Python 3.8+
- NumPy
- Matplotlib

Install dependencies:

```
pip install numpy matplotlib
```

Run the Simulation

```
python CW_RRPR.py
```

Full Source: CW_RRPR.py

~~~
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# 1. Math & Kinematics Library (Custom Implementation)

## Source: key sections from CW_RRPR.py

Below are the major sections of `CW_RRPR.py` split into logical blocks with short
explanations and the reasoning behind the design decisions.

**Math & Kinematics:**

```python
import numpy as np

def dh_transform(theta, d, a, alpha):
    """Create a DH transform matrix for given DH parameters."""
    ct, st = np.cos(theta), np.sin(theta)
    ca, sa = np.cos(alpha), 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   ]
    ])

class RRPRRobot:
    def __init__(self, L1, L2, L3, d4_limits):
        self.L1, self.L2, self.L3 = L1, L2, L3
        self.d3_min, self.d3_max = d4_limits

    def forward_kinematics_all(self, q):
        # Returns list of transforms [T0, T1, T2, T3, T4]
        ...

    def jacobian(self, q):
        # Geometric Jacobian with R, R, P, R joint types
        ...

    def inverse_kinematics(self, target_pos, q0):
        # Simple damped/pseudo-inverse iterative IK for position
        ...
~~~

- **Logic:** This block implements basic spatial kinematics using DH parameters. The `RRPRRobot` encapsulates FK, the Jacobian and an iterative IK solver (pseudo-inverse on the positional Jacobian). The prismatic joint is handled specially in the Jacobian and is clamped within configured limits during IK updates.

Robot Definition:

```
L1, L2, L3 = 0.25, 0.25, 0.25
d_prismatic_min, d_prismatic_max = 0, 0.45
robot = RRPRRobot(L1, L2, L3, [d_prismatic_min, d_prismatic_max])
```

DH Parameters (used in `RRPRRobot.forward_kinematics_all`):

Joint	Type	θ (theta)	d	a	α (alpha)	Notes
1	Revolute (R)	q1	L1 = 0.25	0	π/2	Base revolute (first link offset d=L1)
2	Revolute (R)	q2	0	L2 = 0.25	0	Second revolute (a=L2)
3	Prismatic (P)	0	q3 (clamped 0–0.45)	0	π/2	Prismatic vertical joint (d = q3)
4	Revolute (R)	q4	0	L3 = 0.25	0	Final revolute, end-effector offset a=L3

- **Logic:** The table maps directly to the DH calls in `forward_kinematics_all` and records the numeric link constants used in the simulation. Keeping the DH table here makes it simpler to reason about FK, Jacobian columns, and joint limits.
- **Logic:** Link lengths and prismatic limits are chosen to give a safe reachable workspace; these constants live near the top to make parameter tuning straightforward.

Tasks & Trajectory Generation:

```
floor_targets = [(0.3, -0.4, 0.0), (-0.4, -0.3, 0.0), (-0.1, 0.4, 0.0)]
shelf_targets = [(0.4, 0.55, 0.30), (0.4, 0.55, 0.40), (0.4, 0.55, 0.50)]

def jtraj(q0, q1, steps):
    # Linear interpolation in joint-space; wrap revolute joints for shortest path
    ...

def build_task(q_current, start_xyz, end_xyz, obj_index):
    # Builds approach, contact, grip/place dwell, and transfer sub-trajectories
    ...

def build_program(tasks):
    # Concatenate all task trajectories into a program
    ...
```

- **Logic:** Trajectories are built in joint-space so that FK and visualization are simple. Each pick-and-place task uses approach → contact → grip → retreat → transfer → place sequences with short dwells for grip/place actions. Revolute joints are angle-wrapped to avoid long rotations.

Visualization Helpers:

```python
def hex_faces(center, radius=0.05, height=0.05):
    # Returns polygon faces for a hexagonal prism used as an object
    ...

def make_hex_collection(center, radius=0.05, height=0.05, color='gray'):
    return Poly3DCollection(hex_faces(center, radius, height), facecolors=color,
alpha=0.85, edgecolor='black')
```

- **Logic:** Small helper functions render simple hexagonal objects used for pick/place targets. Abstracting this keeps plotting code clean and reusable.

Plotting & Animation:

```python
# Setup Matplotlib 3D scene, floor, shelves, and object handles
... (visualization setup)

def reset_objects():
    # Reset position and flags for all objects
    ...

def update(frame_idx):
    # Called every animation frame to update robot and object visuals
    ...

def run_animation(frames, actions, indices, interval=20):
    # Create scene, instantiate FuncAnimation, and show
    ...

if __name__ == '__main__':
    frames, actions, indices = build_program(tasks)
    run_animation(frames, actions, indices)
```

- **Logic:** The animation loop updates link line data from the FK of the current joint configuration and applies grip/place state changes to the corresponding object polygons. The scene includes a right-side, color-coded table (created by `create_scene()`) that lists link/joint names and displays live joint values each frame (q1, q2, q4 shown in degrees; q3 in meters). The work envelope is shown as a wireframe hemisphere ($Z \geq 0$) indicating the approximate reachable workspace. The visualization code is isolated so unit tests or alternate front-ends could reuse the trajectory program without plotting.

🎬 Project Demo Video

Watch the full demonstration here: Demo Video

🎬 Project Git Link

Full Project Go Through here: Git Link

References

This project builds upon foundational programming concepts and code structures introduced in coursework and academic resources.
In particular, the initial framework for kinematics and simulation logic was adapted from:

- **Dr. Judhi Prasetyo** – *Foundation Code for Programming* (Course materials, Middlesex University Dubai, 2025).

Jacobian pseudo inverse method learned from NPTEL IIT DELHI YOUTUBE CHANNEL. The Readme formatting,inclusion of perfect comments have been added using co-pilot. Additional features like work envelope and the link diagram in animation coded using vs code co pilot.