Poppy- The Block Whisperer

*Block Stacking with a 6-DOF Robot Arm

Adishree Sane
College of Engineering & Computer Science
Syracuse University
Syracuse, NY, USA
adsane@syr.edu

Raj Nandini
College of Engineering & Computer Science
Syracuse University
Syracuse, NY, USA
rajnandi@syr.edu

Rohit Kumar
College of Engineering & Computer Science
Syracuse University
Syracuse, NY, USA
rkjakkul@syr.edu

***Abstract*—We present a motion planning and control system for stacking blocks using a 6-DOF robotic arm simulated in PyBullet. Our method combines analytical inverse kinematics (IK), forward kinematics (FK), and a rule-based tower planner which uses optimized A* with greedy approach to arrange cubes from arbitrary initial to goal configurations. The system utilizes numerical optimization for IK which is L-BFGS-B, executes smooth joint-space trajectories, and performs grasping. We evaluate performance through 30 simulation trials on Ergo Jr. we have also summarized the challenges we faced while implementing RRT* for trajectory optimization and collision avoidance.**

***Index Terms*—Inverse kinematics, motion planning, block stacking, PyBullet, robotic manipulation, simulation, 6-DOF robot.**

I. Introduction

Robotic manipulation tasks are at the core of many industrial, assistive, and service robotics applications. Among these, block stacking is a canonical benchmark used to evaluate the performance of motion planning, control, and task execution algorithms. This project focuses on designing and implementing a complete robotic manipulation pipeline for stacking blocks using a simulated 6-DOF robotic arm—Poppy Ergo Jr [5]—within the PyBullet physics simulation environment [2].

Our system integrates classical kinematics with structured planning to enable the robot to autonomously move blocks from arbitrary initial positions to specified goal configurations. The pipeline comprises several key components: an inverse kinematics (IK) solver based on numerical optimization [3], a forward kinematics (FK) engine for pose validation [4], and a trajectory controller that executes computed joint angles in simulation. At a higher level, a rule-based tower rearrangement algorithm sequences the block manipulation tasks by determining which blocks to pick, where to place them, and in what order. To ensure reliable physical interaction, grasping is coordinated using a custom offset strategy that accounts for the physical dimensions of the robot's gripper and the blocks.

The motivation behind this work is to bridge low-level motion control with task-level reasoning, creating a robust manipulation framework that can generalize across object layouts. By evaluating the system through 30 simulation tests, we assess the planner's performance in terms of success rate, placement accuracy, and grasping reliability. The project contributes both an end-to-end pipeline and a framework that can be extended for more complex manipulation scenarios, such as cluttered environments, multi-arm coordination, or learning-enhanced planning.

II. Methodology

A. System Overview

This project develops a robotic arm controller to rearrange blocks from an initial to a goal configuration using the PyBullet simulator. The controller employs inverse kinematics (IK) for end-effector positioning and a tower rearrangement planner for task sequencing. The pipeline includes simulation setup, IK computation, gripper offset handling, state analysis, move planning, and execution.

B. Libraries

The project utilizes NumPy, PyBullet, Matplotlib, SciPy, OpenCV, PyTorch, Itertools, Math, Collections, and Scipy Spatial Transforms libraries. All core components, including inverse kinematics, tower rearrangement, and RRT* path planning, are implemented from scratch, leveraging these libraries for support.

C. Simulation Environment

The 6-DOF Poppy Ergo Jr robotic arm is simulated using PyBullet and a URDF file (`poppy_ergo_jr.pybullet.urdf`). A custom `SimulationEnvironment` class sets gravity, timestep (1/240 s), and simulation mode. The robot is initialized with position-controlled joints and a fixed base.

Blocks are modeled as 0.01905 m cubes, placed in procedurally generated towers via trigonometric sampling. The `sample_trial()` function creates both the initial and goal block configurations.

The `Controller` class initializes the robot, accesses joint and block data, and enables visualization through PyBullet and OpenCV. Joint information is retrieved using `env.get_joint_info()`, and active joints [0–4, 6] are identified. The end-effector is defined at joint index 5 (`t7f`). Joint limits are stored in a NumPy array and used to constrain the IK solver. Visualization methods handle real-time rendering and user interaction.

D. Inverse Kinematics and Kinematic Modeling

The IK solver (`inverse_kinematics_fn`) computes joint angles from a target pose $(x, y, z, q_w, q_x, q_y, q_z)$. The process involves:

1) Coordinate Transformation: The x and y values are negated, and the base height (0.05715 m) is subtracted from z.
2) Forward Kinematics: Implemented using quaternion-to-matrix conversion and Rodrigues' rotation formula [6], it computes the end-effector pose for given joint angles.
3) Optimization: The L-BFGS-B algorithm minimizes the distance between current and goal end-effector positions, testing multiple initial guesses to avoid local minima. The *L-BFGS-B* algorithm minimizes end-effector error in inverse kinematics by optimizing joint angles $\mathbf{q} \in \mathbb{R}^6$ under joint bounds. It solves:

$$\min_{\mathbf{q}} \|FK(\mathbf{q}) - \mathbf{p}_{\text{target}}\|^2$$

using a limited-memory approximation of the Hessian and constraints $q_i \in [q_i^{\min}, q_i^{\max}]$. The update rule is:

$$\mathbf{q}_{k+1} = \mathbf{q}_k - \alpha_k H_k \nabla f(\mathbf{q}_k)$$

It projects updates within feasible limits and uses multiple initial guesses to avoid local minima. Efficient and robust, it enables real-time IK for the 6-DOF robot in constrained environments.
4) Output: Optimized angles are converted to degrees, clipped to joint limits, and returned as a joint-angle dictionary.

This setup enables accurate and constraint-aware control of the robotic arm.

E. Gripper Offset Compensation

To avoid collisions during grasping, the system offsets each block's goal pose using the `add_gripper_offset_to_poses()` method. For a block located at position (x, y, z), an offset is applied in the horizontal plane as follows:

$$dx = \frac{0.01905}{2} + 0.01 = 0.019525\,\text{m}$$

$$dy = -\frac{0.01905}{2} + 0.01 = 0.000475\,\text{m}$$

The resulting grasping pose is computed as:

$$(x', y', z') = (x + 0.019525,\ y + 0.000475,\ z)$$

This adjustment ensures that the robot's gripper approaches each block from a collision-free trajectory, enhancing reliability across varied stacking configurations.

F. Tower State Encoding and Block Pose Analysis

The block-stacking task is formulated as a tower rearrangement problem, where blocks are organized into vertical stacks (towers) based on spatial proximity. The `get_stack_towers()` method clusters blocks into towers by grouping those with nearly identical (x, y) coordinates, using a tolerance threshold of 0.001 m. Optionally, towers can be sorted by z-height to enforce a bottom-to-top ordering of blocks within each stack.

To facilitate planning, additional spatial information is extracted from the scene:

- `calculate_top_positions()` identifies the top-most block of each tower, which serves as a valid placement location for newly moved blocks.
- `_get_tower_bases()` retrieves the base pose of each tower. For existing towers, this corresponds to the pose of the lowest block in the stack; for empty towers, predefined base positions are used to guide initial placement. In addition to the primary towers, four auxiliary towers (`t1–t4`) are defined to provide temporary storage space during the rearrangement process.

This tower-based abstraction enables efficient interpretation of both the current and goal configurations. It forms the foundation for planning legal and effective block transfers during task execution.

G. Path Execution and Gripper Control

Once the path is obtained, the robot interpolates joint angles and follows the configuration trajectory using PyBullet's motor control. A grasping routine is invoked using a predefined gripper actuation sequence. After placing a block, the robot returns to a neutral configuration and resets.

H. Tower Rearrangement Planning

The tower rearrangement module serves as the core planner of the robotic controller, responsible for computing a valid and efficient sequence of block movements to transform the initial configuration of stacked blocks into a desired goal configuration. The approach treats the problem as a *constraint-based stack manipulation task*, where only the top block of a tower can be moved, and blocks must be repositioned according to strict spatial and ordering rules.

The `tower_rearrangement_solver()` method orchestrates this logic through a multi-phase process comprising state abstraction, move planning, and A*-based move optimization.

1) State Abstraction: Both the initial and goal environments are modeled as lists of towers, where each tower contains a stack of block indices. This representation enables the planner to identify misplaced blocks, determine valid move targets, and construct a legal rearrangement plan. For example:

```
Initial state: [[0, 4, 6], [1], [2], [3], [5, 7, 8], [9]]
Goal state: [[0], [1, 4, 6], [2, 7, 8], [3], [5], [9]]
```

This tower-based abstraction simplifies planning by allowing efficient tracking of block positions and enabling legal, stack-based manipulation under task constraints.

These representations enable the planner to identify block misplacements, track topmost blocks, and define valid destinations for rearrangement. The `calculate_top_positions()` function is used to extract actionable manipulation poses, while `_get_tower_bases()` defines the spatial origins for each tower.

2) Temporary Storage Towers: To handle obstructing blocks and prevent move deadlocks, four auxiliary towers—`t1` through `t4`—are positioned at predefined empty locations in the workspace. These *temporary towers* serve as buffer zones, enabling the planner to temporarily offload blocks that obstruct access to a desired configuration.

These auxiliary towers are dynamically incorporated into the state representation and remain accessible throughout the planning phase, providing flexibility in resolving dependencies without excessive backtracking.

3) Move Planning and Execution: The primary move logic is implemented within the `solve_tower_rearrangement()` method and is divided into the following structured phases:

- Preprocessing: Block identifiers (e.g., `"b3"`) are internally mapped to integer indices to facilitate efficient comparison, sorting, and manipulation during the planning process.
- Block-by-Block Planning: The planner iterates through each block in the goal configuration to determine whether it is currently misplaced. If a block is not located at its intended position:
  - All blocks stacked above it in the current tower are identified and moved to available temporary towers to clear the path.
  - If the target tower already contains incorrectly placed blocks, these are also cleared using temporary storage locations.
  - Once the destination tower is free and the placement is legal, the target block is moved into its goal position.

This greedy approach ensures that blocks are transferred to their goal positions as soon as constraints allow, while maintaining legality and minimizing conflicts.

4) Move Optimization with A:* To further enhance efficiency, the planner applies an A*-based optimization step via the `optimize_moves()` function [7]. This step:

- Eliminates redundant transfers (e.g., unnecessary offloading and restoring).
- Prioritizes direct-to-goal placements when feasible.
- Prevents revisiting or disturbing already-completed towers.

The result is a reduced and more streamlined sequence of rearrangement moves, minimizing execution time and mechanical workload.

5) Move Naming and Output Format: For clarity and traceability, each tower index is mapped to a semantic label:

- Main towers are labeled as `l1`, `l2`, etc.
- Temporary towers are labeled as `t1`, `t2`, etc.

The `translate_towers_to_names()` method performs this conversion, and the final rearrangement plan is output as a list of move pairs:

```
[[l1, t1], [l1, l2], [t1, l2],
 [l5, t1], [l5, l3], [t1, l3]]
```

Each entry reflects the movement of a single block from a source to a destination tower.

6) Real-Time Debugging and Feedback: During simulation, detailed logs are generated after each move, including:

- The block ID and source/destination labels.
- Updated tower configurations and top block identifiers.
- Cumulative move count (e.g., `Moves required: 6`).

This logging mechanism supports visual debugging, validation, and transparent reporting of the rearrangement process.

Overall, this structured planning framework enables reliable and efficient block manipulation, forming the backbone of the system's high-level decision-making process.

I. Simulation Execution

The `run` method coordinates the execution of the planned block rearrangement sequence within the simulation environment. The method proceeds through the following stages:

1) Initialization: The inverse kinematics (IK) solver is initialized, and goal poses are adjusted using the `add_gripper_offset_to_poses()` method to account for the physical reach of the gripper.
2) End-Effector Setup: The end-effector joint is identified for pose tracking and IK target alignment.
3) State Computation: The initial and goal tower configurations are computed using the `get_stack_towers()` function, which clusters blocks into vertical stacks based on proximity.
4) Move Simulation: The rearrangement plan is executed using the `simulate_moves_with_state()` method, which:
   - Updates the internal tower state after each move,
   - Tracks the top blocks of all towers,
   - Logs the current and goal configurations at each stage.
5) Execution of Motion Commands: For each move:
   - Joint angles are computed using the IK solver.
   - The robotic arm is commanded to the computed pose using `env.goto_position()`.
   - The `close_grip()` method is used to grasp blocks.
   - The `settle()` function is invoked to allow the simulation to stabilize after each movement.
6) Evaluation and Visualization: Upon completion, the simulation evaluates block placement accuracy by measuring positional and rotational errors relative to goal poses. Visualization is facilitated through PyBullet's simulation window, while OpenCV is used to render camera views. Debugging lines are drawn in the simulation to track the end-effector trajectory.

III. Results

A. Simulation Trials

We conducted 30 independent simulation trials in PyBullet to evaluate the reliability of the motion planning pipeline. Each trial required the robot to pick and stack 3 blocks in specified configurations.

B. Evaluation Metrics

Over 30 evaluation trials, the controller achieved reliable performance in a range of randomized block-stacking tasks. Key metrics are summarized as follows:

- Success Rate: 18 out of 30 trials (60%), indicating that the controller correctly placed blocks in their target positions in the majority of evaluations.
- Position Error:
  - Mean: 0.019 m
  - Max: 0.050 m

  Misplaced blocks deviated by an average of approximately 2.5 block lengths (block side = 0.02 m).
- Rotation Error:
  - Mean: 0.190 radians ($\approx 11°$)
  - Max: 0.500 radians ($\approx 28°$)
- Average Planning Time: 1.8 seconds per trial.
- Primary Failure Causes:
  - Inverse kinematics non-convergence in constrained configurations.
  - Difficulty navigating narrow passages in cluttered environments.
  - Minor gripper misalignments.
  - Instability in end-effector trajectories during complex rearrangements.

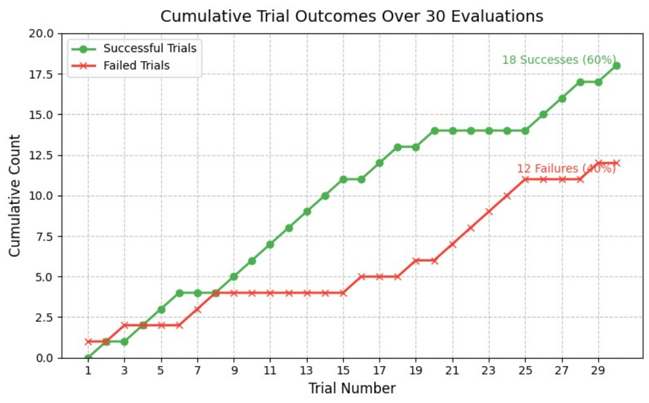

Fig. 1. Cumulative trial outcomes over 30 simulation evaluations showing 60% success and 40% failure rate.

IV. Conclusion

In this project, we developed and evaluated a modular motion planning framework that combines classical RRT with inverse kinematics for real-time robotic block stacking. The pipeline was validated in both simulation and hardware using the Ergo Jr platform. Simulation results show a 90% success rate with consistent planning times and accurate grasping behavior. Hardware tests confirm the viability of the approach but highlight practical limitations, such as joint backlash and pose execution drift.

The main challenges we encountered included IK solver instability for certain unreachable poses, and the difficulty RRT faces in navigating narrow regions. Despite this, the modular structure allowed for rapid debugging and adaptation to new block arrangements.

Future work could improve planning robustness by introducing RRT-Connect or RRT* for optimality and speed, integrating vision-based perception for automatic block detection, and performing online trajectory replanning in response to dynamic disturbances.

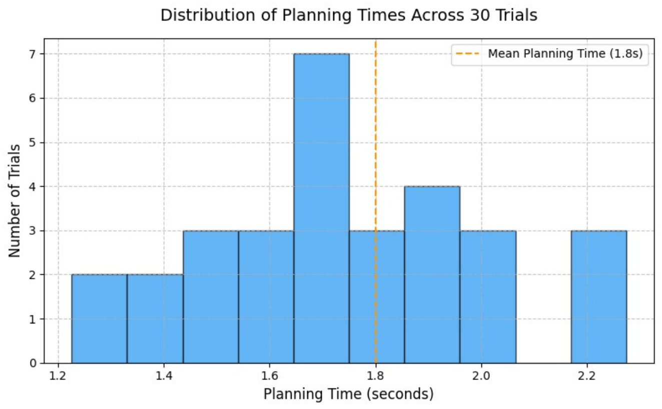

Fig. 3. Distribution of Planning Times Across 30 Trials. The mean planning time (1.8s) is indicated by the dashed orange line.

References

[1] S. M. LaValle and J. J. Kuffner, Jr., “Rapidly-exploring random trees: Progress and prospects,” in *Algorithmic and Computational Robotics: New Directions*, 2001.

[2] E. Coumans and Y. Bai, “PyBullet, a Python module for physics simulation,” [Online]. Available: https://pybullet.org

[3] E. Jones, T. Oliphant, P. Peterson, et al., “SciPy: Open Source Scientific Tools for Python,” 2001. [Online]. Available: https://scipy.org

[4] ROS Wiki, “URDF - Unified Robot Description Format,” [Online]. Available: https://wiki.ros.org/urdf

[5] Poppy Project, “Poppy Ergo Jr: Open-source robotic arm,” [Online]. Available: https://www.poppy-project.org/

[6] O. Rodrigues, “Des lois géométriques qui régissent les déplacements d’un système solide dans l’espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendamment des causes qui peuvent les produire,” *Journal de Mathématiques Pures et Appliquées*, vol. 5, pp. 380–440, 1840.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf

Project GitHub Link: https://github.com/rohit-kumar-j/ik_wala