Iterative Inverse Kinematics and Torque Control for Precision Insertion with a 6-DoF Robot Arm

Yu Wati Nyi
ynyi@umass.edu

Jinghao Gao
jinghaogao@umass.edu

Po-Hsiang Wang
pohsiangwang@umass.edu

***Abstract*—We present a position-based control strategy for a 6-DoF robotic arm to insert a rectangular end-effector into a dynamically vibrating hole using the MuJoCo simulation environment. Our system combines a Newton–Raphson inverse kinematics (IK) solver with a custom position error function derived from real-time sensor data, enabling accurate pose tracking without relying on trajectory planning or force feedback.**

The control loop integrates pose estimation, IK solving, and torque-level execution via a PD controller. We evaluate performance across nine difficulty settings defined by increasing vibration amplitude and frequency. Results show that our custom error function improves insertion success rates from under 60% to over 90%, and reduces average insertion time by 3.5 seconds compared to the baseline.

This work demonstrates that analytical, geometry-driven methods can achieve robust and low-latency insertion in dynamic environments, offering a lightweight alternative to learning-based approaches.

I. Introduction

Precise insertion remains a core challenge in robotic manipulation, particularly when the target is not static but instead exhibits continuous motion and rotation. This scenario frequently arises in industrial automation settings, such as vibrating assembly lines or moving platforms, where end-effectors must adapt in real time to dynamic target conditions.

In this project, we tackle the problem of controlling a 6-degree-of-freedom (6-DoF) robotic arm to insert a rectangular end-effector into a moving and rotating square hole in MuJoCo simulation environment.

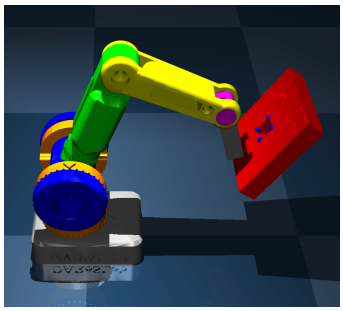

Fig. 1. 6-DoF robotic arm

Our solution integrates an iterative inverse kinematics (IK) solver based on the Newton-Raphson method with a custom position error estimation strategy. The position error is computed in real time from the sensed hole geometry, allowing the end-effector to gradually approach and align with the hole. Unlike methods that rely on predefined trajectories or force feedback, our approach is fully position-based and responsive to target motion.

The final control loop combines pose estimation, IK solving, and low-level joint control in a unified, continuously updated routine. Though not explicitly state-based, this pipeline implicitly achieves approach, alignment, and insertion behavior without relying on separate phases or high-level planning. The system's performance is evaluated based on insertion accuracy and its adaptability to varying hole dynamics.

II. Related Works

Accurate and responsive inverse kinematics (IK) is critical for enabling robotic manipulators to perform precise tasks, particularly when interacting with dynamic environments. A variety of numerical IK methods have been developed over the years, including Jacobian pseudo-inverse, damped least-squares, and Newton-Raphson approaches.

The Newton-Raphson method, although traditionally used for scalar root finding, can be extended to solve IK problems by treating the task-space error as a nonlinear function of joint angles. Its fast local convergence makes it well-suited for real-time applications where responsiveness is essential. Our implementation draws direct inspiration from the practical guide by Ale Fram [1], which demonstrates how to use MuJoCo's built-in Jacobian computation and forward simulation capabilities to iteratively minimize position and orientation error. We build upon this approach by extending it to a 6-DOF robot and incorporating dynamic trajectory interpolation and PD control for task execution.

Trajectory shaping using interpolation methods—such as cosine or polynomial splines—has been widely used to ensure smooth transitions and acceleration-limited motion profiles [2]. In this work, cosine interpolation is employed to gradually move the end-effector from a mid-approach point to the target insertion pose, thereby reducing overshoot and instability.

MuJoCo, a physics engine designed for efficient and accurate simulation of articulated bodies and contact dynamics, has seen increasing use in research on robotic control and learning. It provides low-level access to quantities such as joint positions, velocities, and Jacobians, which are essential for implementing low-latency controllers like the one proposed in this study [3]. Recent studies have also explored learning-based approaches for tasks like insertion and grasping in uncertain environments [4][5]. While these methods offer

generalizability, they often require large datasets and computational resources. In contrast, our method demonstrates that reliable and adaptive control can be achieved using analytical solutions and classical control strategies.

III. Methods

A. Objective

The goal of the method is to be able to insert a 6-DOF robot arm manipulator into a dynamically vibrating box in a simulation and maintain the ability to do so even as the 'difficulty' of doing so increases. The difficulty is measured by how fast and randomly the box moves.

To do so, we have implemented the optimization algorithm to iteratively reduce the error between the target position and the current end effector position, the Newton-Raphson method. While we computed the Newton-Raphson method, we also used a few methods that were given to us to help us get the proper alignment and positions of both the manipulator and the box.

B. The simulation and setup

The simulation is entirely run on MuJoCo. MuJoCo physics engine, which provides accurate modeling of multi-joint articulated bodies with contact dynamics. The simulation includes: A 6-DOF robot manipulator, a box with a hole target mounted on a vibrating base, real-time feedback from position sensors embedded in the box.

The simulation model is defined through a custom XML file that specifies geometry, sensors, and joint limits. Control logic is implemented externally in Python using MuJoCo's API.

C. Control Pipeline

Our system architecture is composed of three integrated modules that together facilitate real-time perception, motion planning, and control within the MuJoCo simulation environment: `BoxControlHandler`, `RunMiniArmBox`, and `YourControlCode`.

Environment Interface: `BoxControlHandler`.

This module serves as the interface between the simulation and the robot controller. It handles environmental perception, target pose estimation, and error feedback. Its core functionalities include:

- **Sensor Processing & Pose Estimation:** Computes the orientation of the box using four embedded position sensors. The resulting pose is rotated 90° about the y-axis to match the insertion direction.
- **Trajectory Shaping:** Performs cosine interpolation between an intermediate and final target pose to create smooth, progressive motion for insertion.
- **Error Computation:** Outputs the 3D position error between the desired insertion pose and the robot's end-effector. This error vector serves as input to the IK solver.
- **Difficulty Scaling:** Dynamically adjusts vibration parameters (amplitude, frequency) according to a difficulty level, simulating more challenging insertion tasks at higher levels.
- **Goal Detection:** Monitors contact between the end-effector and a detection plate. Upon successful insertion, it halts the robot and signals task completion via a visual cue.
- **Supporting Utilities:** Includes a suite of quaternion math operations, SO(3) conversions, collision detection, and box center estimation.

a) Simulation Management: `RunMiniArmBox`*:* This script coordinates the overall simulation, including setup, physics integration, and real-time rendering.

- **VibratingBox Class:** Simulates external disturbances by applying sinusoidal motion to the box using a proportional-derivative (PD) controller.
- **Simulation Loader:** Loads the MuJoCo model and data and initializes the simulation loop.
- **Main Physics Loop:** Applies control inputs, checks task success, advances physics simulation steps, and synchronizes visualization.

b) Robot Controller: `YourControlCode`*:* This module defines the control logic for computing and executing joint-space trajectories.

- **Initialization:** Connects to the MuJoCo model and the environment handler. Also configures the difficulty level.
- **Inverse Kinematics Solver:** Uses a Newton-Raphson method to iteratively solve for joint configurations that reduce both positional and orientational error between the robot's end-effector and the target pose.
- **PD Control:** Applies a joint-space PD controller to smoothly track the IK solution under dynamic and time-varying conditions.

D. Inverse Kinematics and Newton-Raphson

The Newton-Raphson method is a classical root-finding algorithm that iteratively generates improved approximations to the roots of a real-valued function. The core idea is to begin with an initial guess and approximate the function locally using its tangent line. The root of this linear approximation is then taken as the next guess, which is typically closer to the actual root.

In our context, we apply the Newton-Raphson method to compute joint configurations that bring the robot's end-effector closer to a desired target pose. At each iteration, we evaluate the 6-dimensional task-space error vector:

$$\mathbf{e} = \begin{bmatrix} \mathbf{p}_{\text{target}} - \mathbf{p}_{\text{ee}} \\ \log\left(\mathbf{R}_{\text{target}}\mathbf{R}_{\text{ee}}^{\top}\right) \end{bmatrix}$$

where $\mathbf{p}_{\text{ee}}$ and $\mathbf{p}_{\text{target}}$ denote the current and desired end-effector positions, respectively.

a) Rotational Error via Quaternions.: We compute the rotational component of the error using quaternion operations:

$$\mathbf{q}_{\text{err}} = \mathbf{q}_{\text{target}} \otimes \mathbf{q}_{\text{ee}}^{-1}$$

where $\otimes$ denotes quaternion multiplication. The quaternion error is then converted to a minimal 3D rotation vector using the logarithmic map:

$$\log(\cdot) \quad \text{or equivalently, } \texttt{quat\_to\_so3}.$$

b) Jacobian Construction.: The full spatial Jacobian $J \in \mathbb{R}^{6\times n}$ is assembled using MuJoCo's built-in `mj_jac` function. It includes:

$$J_{\text{pos}} \in \mathbb{R}^{3\times n} \quad \text{(linear velocity Jacobian)} \tag{1}$$
$$J_{\text{rot}} \in \mathbb{R}^{3\times n} \quad \text{(angular velocity Jacobian)} \tag{2}$$

These are vertically stacked as:

$$J = \begin{bmatrix} J_{\text{pos}} \\ J_{\text{rot}} \end{bmatrix}$$

c) Joint Update Rule: We solve for the joint update $\Delta\mathbf{q}$ using the pseudo-inverse:

$$\Delta\mathbf{q} = J^{\dagger}\mathbf{e}$$

If J is full-rank, we solve directly using:

$$\Delta\mathbf{q} = J^{-1}\mathbf{e}$$

d) Update Step and Termination.: The next joint position is updated using a learning rate $\alpha \in (0, 1]$:

$$\mathbf{q}_{\text{next}} = \mathbf{q}_{\text{current}} + \alpha\Delta\mathbf{q}$$

Empirically, higher difficulty settings benefit from a larger α (e.g., 0.8) to produce more aggressive steps, while lower difficulty settings use smaller values for finer accuracy.

After each update, we call `mj_forward` to refresh the forward kinematics and recompute the Jacobian. This process continues for a fixed number of steps or until the norm $\|\mathbf{e}\|$ falls below a predefined threshold.

This method enables adaptive and locally optimal joint-space motion planning, even under rapidly shifting target conditions in the simulated box environment.

E. Perception and Target Estimation

The `update()` method in `YourControlCode.py` implements a single control cycle for the robot's motion toward the dynamically estimated box hole target. It is invoked at every simulation timestep and performs both perception-based pose estimation and joint-level control.

At the beginning of each cycle, the method records the current simulation time, initializing it if unset. It then collects positional data from four box-mounted sensors to estimate the orientation of the box. This orientation is computed using geometric relationships between the sensor vectors and converted into a quaternion representation. To align with the desired end-effector insertion direction, the orientation is rotated by 90° about the y-axis.

Next, the method calls `get_EE_pos_err()`, which computes a smooth target trajectory by interpolating between an intermediate pose and the final insertion pose, based on the simulation clock. The interpolated position error is then added to the current end-effector position, yielding the desired target pose.

To move the robot toward this target, the method invokes the `Newton_Raphson()` function. This function performs iterative inverse kinematics to compute a joint configuration that minimizes both position and orientation error between the end-effector and the target pose. The final joint configuration from the iteration is selected as the motion goal.

Finally, a torque-level proportional-derivative (PD) controller is applied to drive the joints toward the computed targets. The controller uses high proportional and derivative gains to ensure rapid convergence and stable execution. This control cycle repeats until the robot successfully completes the insertion task or exceeds the specified time threshold.

F. Optional Custom End-Effector Position Error Function

We implemented a method to estimate the positional error between the robot's end-effector and a target insertion point within the box. This function `compute_custom_pos_err()` serves as a simplified and efficient alternative to the default `get_EE_pos_err()` method defined in the `BoxControlHandle` class.

Unlike the original method—which involves quaternion-based orientation estimation, time-dependent interpolation, and dynamic pose updates—the custom function leverages raw sensor data to construct a local box frame and identify a static insertion target.

a) Workflow of `compute_custom_pos_err()`:

- **Sensor Reading:** Extracts 3D positions from the four box-mounted sensors.
- **Validation:** Checks the numerical stability and validity of sensor readings.
- **Box Frame Construction:** Constructs an orthonormal local frame using:
  - $\mathbf{x}$: A vector between two sensors indicating lateral orientation.
  - $\mathbf{z}$: A surface normal computed via the cross product of edge vectors.
  - $\mathbf{y}$: Derived as $\mathbf{y} = \mathbf{z} \times \mathbf{x}$ to ensure orthogonality.
- **Target Estimation:** Defines a static target located 2 cm along the local X-axis of the box.
- **Error Computation:** Computes the difference vector between the target and the current end-effector position.

This approach bypasses the need for quaternion rotations, interpolation functions, and time-based control logic, resulting in lower computational overhead during real-time execution. It is particularly advantageous during early motion phases, when precise orientation alignment is less critical, or when a quick approximation of the insertion direction suffices.

When substituted for `get_EE_pos_err()`, this method enabled faster target convergence and more rapid Newton-Raphson iterations. It proved especially effective in low-difficulty settings or when benchmarking the performance of the IK solver.

IV. Result

We evaluated the controller under nine difficulty settings (0.1–0.9), where higher values correspond to faster and more erratic box motion. For each setting, we executed 10 independent trials and recorded both the success rate and average

insertion time using the original position error function and our custom position error function.

The aggregate results are visualized in Figure 1 (success rate) and Figure 2 (insertion time). The plots show a clear performance advantage for our custom position error function, which achieves consistently higher success rates and shorter insertion times across all difficulty levels.

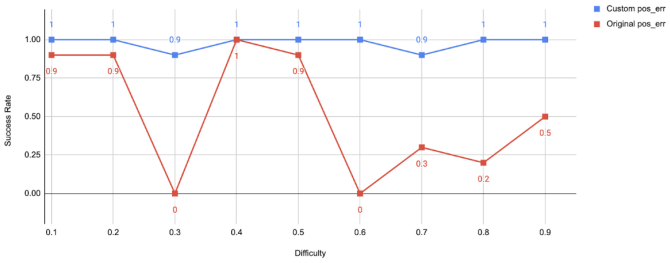

Fig. 2. Success rate across different difficulty levels

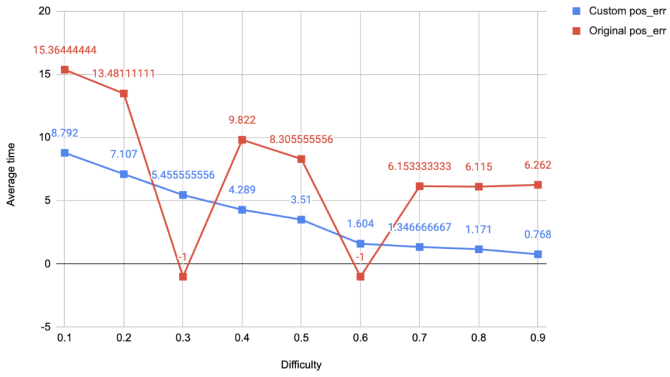

Fig. 3. Average insertion time under different difficulty levels

V. Analysis

Figure 2 shows that the original position error function performs poorly at difficulty levels 0.3 and 0.6, while our custom position error function achieves significant improvement—reaching at least a 90%, and in most cases a 100%, success rate across all difficulty levels.

Figure 3 demonstrates that our custom function also reduces the average insertion time by approximately 3.5 seconds (from 7.28 s to 3.78 s). Interestingly, both position error functions exhibit shorter insertion times as the difficulty increases, suggesting that the box's motion can sometimes assist with alignment under faster dynamics.

Why does our custom position error function improve so much?

A single, consistent target point: By defining the insertion target as a fixed 2 cm offset along the box's instantaneous X-axis, our custom position error function eliminates the multi-stage, time-based mid-target used in the baseline. The solver always "sees" the true hole center in the box's own frame, so each IK update is directly driving toward the same physical point, rather than chasing a moving interpolation that can lag behind reality.

Harnessing Newton-Raphson's strengths: With a clean, noise-free error vector, the Newton-Raphson routine can focus entirely on minimizing pose error. No extra quaternion-to-rotation or cosine-spline computation is muddying the Jacobian's input. As a result, each linearization step is more accurate, and the controller can safely increase its step size to achieve better results.

Simpler logic, fewer edge cases: Removing interpolation thresholds, alignment checks, quaternion operations, and dynamic time-based adjustments results in a leaner control logic: sense four points, compute a 3×3 frame, form a 6-D error, solve, and apply. Fewer lines of special-case logic translate directly into fewer opportunities for numerical or logical glitches.

Why can't the end-effector plug into the box every time?

Near-miss insertions due to slight misalignment: The box's sinusoidal motion creates moments where the end-effector approaches the hole but with small lateral or angular offsets. These "almost aligned" cases lead to frequent failures, as the controller lacks logic to verify or correct insertion alignment.

Fixed-step IK limits correction: The Newton-Raphson solver runs a fixed number of iterations per frame. When the initial error is large or the Jacobian is poorly conditioned, it may converge to a nearby but invalid configuration, stopping short or overshooting without actually inserting.

No recovery after overshoot: Once the insertion fails, the controller continues following the shifting target without recognizing the missed opportunity. Without rollback or re-alignment strategies, minor errors remain uncorrected, causing repeated failure under otherwise solvable conditions.

Why does it take more time to insert the hole under lower difficulty?

The box finds the end-effector itself: Our end-effector moves slowly when near the box. Under high difficulty, the box moves quickly, which increases the chance that the box moves toward the end-effector and "finds" it, reducing insertion time.

Slower box motion causes smaller position updates: When the box barely moves, the raw position error computed from the sensors is very small. Since the Newton-Raphson update scales directly with the error, smaller changes between frames lead to slower progression. As a result, the controller must take many micro-steps to close the gap, which lengthens the overall insertion time under low difficulty settings.

Acknowledgment

The authors would like to thank the course instructors and teaching assistants for guidance throughout the project.

REFERENCES

[1] A. Fram, "Basic inverse kinematics in MuJoCo," *Alefram.github.io*, 2022. [Online]. Available: https://alefram.github.io/posts/Basic-inverse-kinematics-in-Mujoco

[2] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

[3] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, 2012, pp. 5026–5033.

[4] D. Kalashnikov *et al.*, "QT-Opt: Scalable deep reinforcement learning for vision-based robotic manipulation," in *Proc. Conf. Robot Learn. (CoRL)*, 2018.

[5] S. James, A. J. Davison, and E. Johns, "Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 12627–12637.