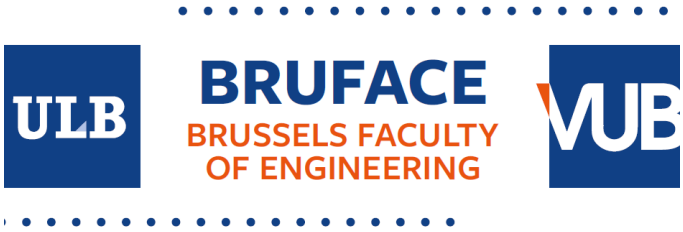

MASTER 1 PROJECT

ELECTROMECHANICAL ENGINEERING - ROBOTICS AND MECHANICAL CONSTRUCTION

Optimization Based Motion Planning with Robotic Manipulators

Author:
Awais ABD UL REHMAN

Academic Supervisor:
Prof Bram Vanderborght

Academic Assistant:
Gaoyuan Liu

BRUFACE Master
VUB/ULB – Brussels Campus

Academic Year 2024–2025

Abstract

0.1 Abstract

In this project I explored the application of both reactive inverse kinematics (IK) and optimization-based trajectory planning for enabling a Franka Emika Panda manipulator to reach targets while avoiding obstacles. I used the PyBullet [1] physics engine to simulate the Panda loaded via its URDF model [2], configuring gravity, collision detection, and joint damping for accurate kinematic behavior. First, I implemented a reactive IK controller that leverages PyBullet's calculateInverseKinematics and collision queries to steer the robot around obstacles in real time. Then, I formulated a trajectory optimization problem over discrete time steps by defining cost functions for goal tracking, smoothness, and collision avoidance, drawing inspiration from the OpTaS library [3]. I solved the resulting nonlinear program using SciPy's SLSQP solver under joint limit constraints SciPy Documentation. The reactive IK approach yielded smooth, collision free motions with minimal latency. In contrast, the optimization based planner produced trajectories that satisfied the objectives but exhibited jitter and oscillations—likely due to solver sensitivity, competing cost terms, and the absence of dynamic damping. Key contributions include the implementation of a robust reactive IK scheme and insights into the practical limitations of naive optimization based planning for robot manipulators. Lessons learned from this work inform future efforts to hybridize reactive and optimization based methods for both real time safety and trajectory smoothness

List of Abbreviations and Symbols

0.2 Abbreviation

Abbreviation	Definition
DOF	Degrees of Freedom: The number of independent parameters required to specify a mechanical system. In robotics, the number of independent joint movements.
IK	Inverse Kinematics: Computing joint parameters that achieve a desired end-effector pose.
FK	Forward Kinematics: Calculating the end-effector's pose from known joint values.
URDF	Unified Robot Description Format: XML file for robot's structure (links, joints, visuals).
ROS	Robot Operating System: Middleware for robotic software development.
PyBullet	Python module for Bullet Physics SDK: used in physics simulation and robotics.
MPC	Model Predictive Control: Optimizes future control actions over a dynamic model.
RNEA	Recursive Newton–Euler Algorithm: Computes inverse dynamics of rigid-body systems.
SLSQP	Sequential Least Squares Programming: Gradient-based optimization solver.
TO	Trajectory Optimization: Designing optimal motion trajectories under constraints.

Table 1: List of Abbreviations and Symbols

Contents

List of Figures

Chapter 1

Introduction

Motion planning is central to robotic manipulation, involving the computation of joint-space commands that guide a manipulator from an initial configuration to a specified goal while ensuring collision avoidance and adherence to kinematic constraints. Collaborative robots, notably the Franka Emika Panda, are increasingly deployed in dynamic, human-robot shared workspaces, necessitating planners that are both safe and responsive. The Panda features seven revolute joints, an 850 mm reach, and a 3 kg payload. Its kinematic and collision properties are described via a URDF model, facilitating integration with simulation and control tools.

To develop and validate motion planners before real-world trials, we employ PyBullet—a Python interface to the Bullet Physics SDK—providing efficient URDF loading, real-time collision detection, and forward/inverse dynamics for rapid prototyping in simulation.

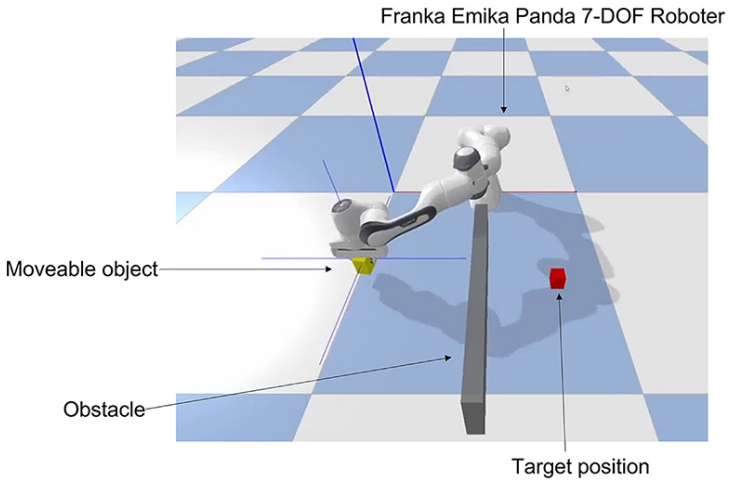

Figure 1.1: Simulated planning environment in PyBullet with Franka Emika Panda

Within this environment, two principal planning paradigms have emerged as state of the art: reactive inverse kinematics (IK) control and optimization-based trajectory planning. Reactive IK methods compute joint updates at high frequency, using the Jacobian pseudoinverse to track targets while applying repulsive corrections near obstacles, achieving millisecond-scale responsiveness. Conversely, trajectory optimization formulates an entire motion as a constrained nonlinear program—minimizing composite cost functions for goal tracking, smoothness, and obstacle clearance—by leveraging solvers such as SciPy's SLSQP [4], IPOPT, and SNOPT.

1.1 State of the Art

1.1.1 Reactive IK Control

Early work by Khatib (1986) introduced artificial potential fields for real-time obstacle avoidance, generating repulsive forces to steer manipulators away from hazards without global path planning. Modern reactive IK methods extend this concept by projecting repulsive velocities into the robot's null space via the Jacobian pseudoinverse, ensuring primary end-effector tracking while avoiding collisions. Libraries such as `pybullet-planning` incorporate these techniques to enable feedback-driven controllers operating at simulation rates exceeding 200 Hz. Despite their speed, reactive schemes can get stuck in local minima and may yield suboptimal detours in cluttered environments unless augmented with higher-level planning.

1.1.2 Trajectory Optimization

Trajectory optimization treats planning as a global nonlinear program. CHOMP (Covariant Hamiltonian Optimization for Motion Planning) introduced a covariant gradient descent method that iteratively refines trajectories by minimizing smoothness and obstacle-avoidance costs, demonstrating robust performance in high-DOF spaces. STOMP (Stochastic Trajectory Optimization for Motion Planning) further addressed non-differentiable cost landscapes by using stochastic trajectory perturbations, yielding collision-free, smooth motions without needing explicit gradients. Model Predictive Control (MPC) extends these offline approaches to receding horizon schemes, enabling periodic re-optimization for dynamic goals at the cost of increased computational burden.

1.1.3 Unified Frameworks and Hybrid Approaches

Recent trends focus on combining reactive and optimized paradigms. Hybrid planners execute reactive IK for immediate safety while invoking trajectory optimizers for global refinements, striking a balance between real-time reactivity and smoothness. The OpTaS library [3] exemplifies a modern, Python-based framework for both trajectory optimization and MPC, leveraging CasADi for automatic differentiation and supporting multiple solver back ends (IPOPT, SNOPT, SLSQP) in a unified API.

1.2 Project Statement

In this project, we implement and rigorously compare two motion planning strategies for the Franka Emika Panda manipulator in PyBullet. First, we develop a high-frequency reactive IK controller that augments PyBullet's built-in inverse kinematics with collision-based repulsive adjustments to maintain a configurable safety margin in real time. Second, we formulate and solve a full joint-space trajectory optimization problem using SciPy's SLSQP solver, incorporating cost terms for end-effector tracking, trajectory smoothness, and obstacle avoidance. By evaluating both methods under identical simulation scenarios, we aim to characterize their performance trade-offs in terms of safety, smoothness, computational efficiency, and implementation complexity.

Chapter 2

Literature Review

This review surveys two major strands of robot motion planning—reactive obstacle avoidance and trajectory optimization—and highlights libraries and comparative studies that bridge these paradigms. First, we discuss reactive methods, from potential-field approaches to modern inverse-kinematics (IK) controllers and learning-based schemes. Next, we examine trajectory optimization techniques such as CHOMP and STOMP, along with recent software frameworks including OpTaS. Finally, we consider hybrid and comparative works that evaluate reactive versus optimized planners.

2.1 Reactive Obstacle Avoidance

Reactive approaches compute control commands on-the-fly, using current sensor or simulation data to steer the manipulator away from obstacles without planning a full trajectory in advance.

Potential-Field Methods

Early reactive planners used artificial potential fields, where obstacles generate repulsive forces and goals attract the robot. While intuitive, these methods suffer from local minima and oscillations in narrow passages.

Task-Priority Frameworks

More formal reactive schemes embed obstacle avoidance as a secondary task in the task-priority IK framework. For instance, task-priority reactive planners use null-space projections to combine primary goals (e.g., end-effector tracking) with secondary collision-avoidance objectives.

Distance-Based Repulsion in IK

Modern reactive IK controllers compute joint solutions via methods like Jacobian transpose and then adjust them using proximity information. Reactive dual-trajectory planners calculate the closest points to obstacles and apply repulsive corrections, achieving smooth avoidance in manipulators.

Null-Space vs. Optimization-Based Kinematics

A recent comparative study examines null-space-based and optimization-based kinematic obstacle avoidance, finding that null-space projections avoid collisions with lower computational cost, whereas optimization-based kinematics can offer smoother motions at higher runtime cost.

Global Reactive Fields

Becker et al. propose "Informed Circular Fields," a global reactive planner that integrates heuristic global direction fields with local obstacle avoidance for manipulators, demonstrating improved performance in dynamic environments.

Learning-Based Reactive Policies

Recent works employ reinforcement learning or neural feedback terms to adapt reactive policies from demonstration, enhancing robustness to perception uncertainties and enabling whole-body avoidance.

2.2 Trajectory Optimization

Trajectory optimization formulates motion planning as a nonlinear program over the entire trajectory, balancing goal accuracy, smoothness, and obstacle penalties.

CHOMP (Covariant Hamiltonian Optimization for Motion Planning)

CHOMP is a pioneering gradient-based optimizer that minimizes a functional combining smoothness and obstacle-avoidance energies. It iteratively refines an initial (possibly infeasible) trajectory into a collision-free path using covariant gradient updates and retains reparameterization invariance.

STOMP (Stochastic Trajectory Optimization for Motion Planning)

STOMP leverages stochastic sampling around an initial trajectory to optimize paths under arbitrary cost functions. It has been integrated into MoveIt! as an alternative to sampling-based planners, offering improved smoothness at the expense of compute time.

MPC and QP-Based Methods

Model Predictive Control (MPC) approaches linearize dynamics and solve a quadratic program at each time step, ensuring collision avoidance with convex constraints. Such methods achieve reactive re-planning with guaranteed feasibility but require fast QP solvers.

OpTaS Library

OpTaS is a recent Python library that streamlines both trajectory optimization (TO) and Model Predictive Control. It allows users to specify custom cost functions—joint-space or task-space—and constraints in a single script, interfacing with solvers like IPOPT, SNOPT, and SciPy's SLSQP.

Comparisons of Solvers

Studies indicate that gradient-based solvers (e.g., CHOMP, SLSQP) require careful cost weighting and initialization to avoid local minima and oscillations, whereas sampling-based methods (e.g., STOMP) and QP/MPC approaches offer robustness at higher computational cost.

2.3 Hybrid and Comparative Studies

Bridging reactive and optimized paradigms, several works investigate hybrid approaches or directly compare methods:

Hybrid Reactive–Optimized Frameworks

Some planners use reactive IK for immediate collision avoidance and resort to optimization only when a global path refinement is needed, combining the speed of feedback control with the smoothness of TO.

Empirical Comparisons

Comparative analyses of reactive IK versus trajectory optimization reveal that reactive methods outperform in real-time responsiveness and simplicity, while optimized methods excel in smoothness and global optimality—though they may jitter without dynamic constraints or weight scheduling.

Future Directions

Emerging research suggests integrating learned predictive models into TO formulations or using MPC with warm-starts from reactive controllers to balance reactivity and optimality in dynamic settings.

Summary

Reactive methods remain indispensable for real-time avoidance due to their simplicity and robustness, while trajectory optimization and MPC offer powerful tools for smooth, goal-directed motion. Libraries like OpTaS are lowering the barrier to experimenting with optimized planners, and hybrid strategies promise to combine the best of both worlds.

Chapter 3

Methodology

This section provides a comprehensive, reproducible account of the workflow used to implement both the reactive inverse-kinematics (IK) controller via PyBullet's built-in solver and the optimization-based trajectory planner using OpTaS. Each phase—from environment setup through validation and comparative analysis—is described in a professional, neutral tone.

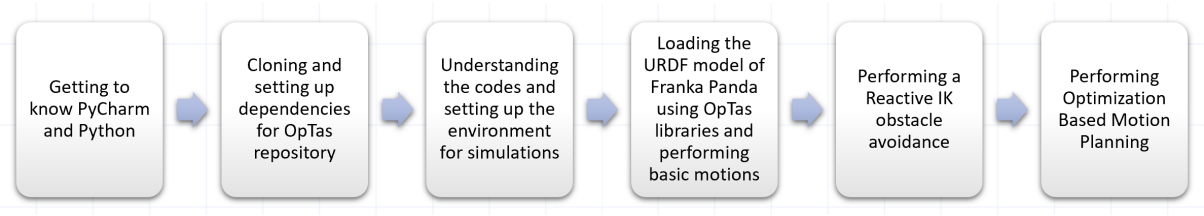

Figure 3.1: Methodology Workflow.

3.1 Environment Preparation and Toolchain Installation

1. **Python Environment**
   - *Version & Isolation*: Python 3.10 was installed within a dedicated virtual environment (`venv`) to isolate project dependencies.
   - *IDE*: PyCharm was utilized for its integrated debugger, advanced code completion, and Git integration.
2. **Repository & Dependencies**
   - The project repository was cloned, and the virtual environment activated.
   - Required packages were installed via:

     `pip install -r requirements.txt`

     including `numpy`, `pybullet`, `scipy`, `casadi`, `optas`, and `ipopt`.
   - Successful installation was confirmed with:

     `python -c "import numpy, pybullet, casadi, optas"`

3.2 Simulation Framework Configuration

1. **PyBullet Initialization**
   - Connection to PyBullet GUI mode was established using `p.connect(p.GUI)`.

- A ground plane was loaded using: `p.loadURDF("plane.urdf")`.
- The simulation was configured to run at a timestep of 1/240 s: `p.setTimeStep(1/240)`.
- Real-time simulation was disabled with: `p.setRealTimeSimulation(0)`.

2. **URDF Model Loading & Validation**

- The Franka Emika Panda URDF was loaded from the path `MA1/panda_arm.urdf`.
- Forward kinematics were validated by commanding known configurations (e.g., home, elbow-up) and verifying end-effector positions via `getLinkState`.

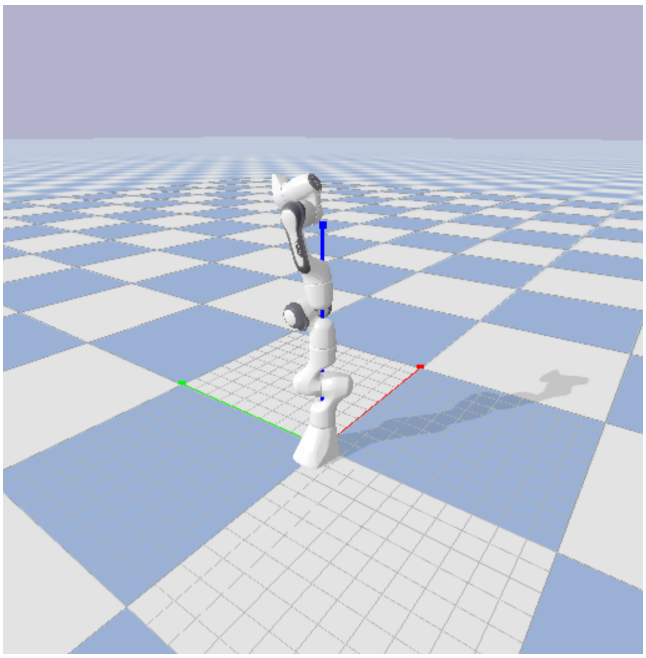

Figure 3.2: Panda robot initialized in standing pose with all joints at zero. Joint numbers are labeled using visual arrows.

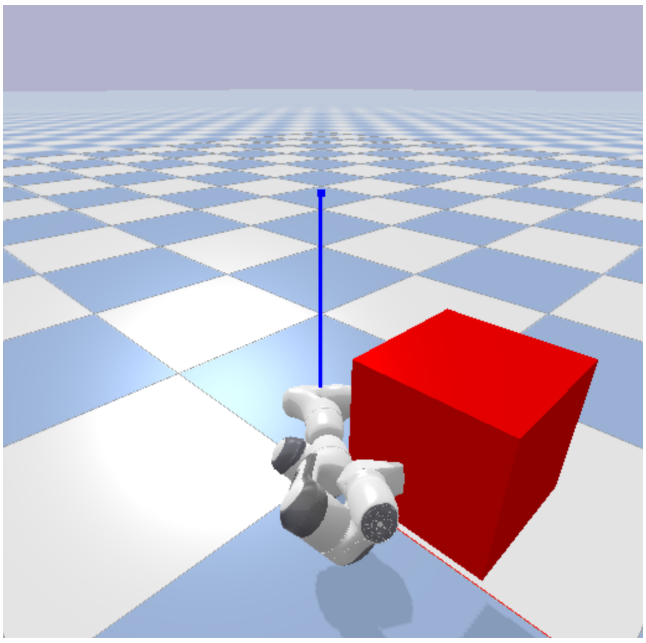

Figure 3.3: Robot tracing a circular corner path using planned joint-space motion.

3.3 Reactive Inverse-Kinematics Obstacle Avoidance

Implementation Overview

The reactive IK controller leverages PyBullet's damped IK solver `p.calculateInverseKinematics` [1] at each 240 Hz [5] timestep:

1. End-effector pose and obstacle distances are retrieved using `getClosestPoints`.
2. A raw Cartesian target is generated following a helicopter-wing trajectory.
3. Within a defined proximity threshold, a vertical lift component is added [6] :

$$\Delta z = \frac{\kappa}{d^2 + \varepsilon}, \quad d = |p_e - p_{\text{obs}}|.$$

4. Target smoothing is applied via exponential filtering: $x_{\text{filt}} \leftarrow \beta\, x_{\text{filt}} + (1-\beta)\, x_{\text{raw}}$.
5. Joint angles are computed using: `calculateInverseKinematics` .
6. These angles are then dispatched using: `setJointMotorControl2` in position control mode.

   In implementing the robot motion and control system, several key PyBullet functions were utilized to model physical behavior, compute joint configurations, detect proximity to obstacles, and command joint movements effectively. Below it is described how these features were employed in the simulation pipeline.

3.3.1 Joint Damping and Dynamics

To simulate more realistic actuator behavior and suppress unwanted oscillations, Joint damping was applied using the `p.changeDynamics` function. This introduces a damping torque defined as:

$$\tau_{\text{damp}} = -b\dot{q}$$

where b is the joint damping coefficient and $\dot{q}$ is the joint's angular velocity. Internally, Bullet handles this by adding a motor with zero target velocity and force proportional to the damping term. It's worth noting that damping values defined in the URDF are not automatically enforced during runtime — these must be explicitly set via this function.

3.3.2 Collision and Proximity Detection

To ensure the robot's end-effector avoids nearby obstacles, `p.getClosestPoints` was used to query the proximity between the robot and other objects (e.g., a tree model). This function returns a list of distance and contact-like information, from which the minimum distance d_{min} was extracted. If this falls below a predefined threshold, the end-effector was elevated to avoid collision.

Under the hood, Bullet performs these queries using the Gilbert–Johnson–Keerthi (GJK) algorithm for convex-convex distance detection. Although highly efficient, the algorithm can exhibit small numerical inaccuracies if the involved geometries aren't strictly convex.

3.3.3 Inverse Kinematics

To control the robot's end-effector position, `p.calculateInverseKinematics` was used. This function computes a set of joint angles that approximately move the end-effector to a desired position. Bullet wraps Samuel Buss's IK solver and uses a Damped Least Squares (DLS) formulation:

$$\delta q = (J^\top J + \lambda^2 I)^{-1} J^\top \delta x$$

This balances precision with robustness near singularities by tuning the damping factor λ. The solver also optionally supports null-space projections for redundancy resolution, although primarily the basic DLS version exposed in PyBullet was used.

3.3.4 Joint Control with PD Regulation

To actuate the robot's joints, `p.setJointMotorControl2` with `POSITION_CONTROL` mode was used. This invokes a built-in PD controller:

$$\tau = K_p(q_{\text{target}} - q) + K_d(\dot{q}_{\text{target}} - \dot{q})$$

The `force` parameter acts as a cap on the maximum torque the motor can apply, influencing how quickly the joint reaches the desired position. Although PyBullet provides default values for `positionGain` and `velocityGain`, these were adjusted in some cases to achieve a balance between responsiveness and smooth motion.

Overall, this setup allowed for physically plausible simulations while maintaining a high level of control over the robot's movement and interaction with its environment.

3.4 Optimization-Based Trajectory Planning with OpTaS

Symbolic Problem Definition

- **Decision Variables**: Joint sequence $\{q_k\}_{k=0}^{T-1}$.
- **Objective Function** (including soft collision penalty) [7]:

$$\mathscr{J} = \sum_{k=0}^{T-1} \left[20 \left\| p_e(q_k) - p_{\text{target}}(k) \right\|^2 + \left\| q_k - q_{k-1} \right\|^2 \right] + \sum_{k:\, d_k < r_{\text{thresh}}} \frac{300}{d_k + 10^{-3}},$$

  where $d_k = \left| p_e(q_k) - p_{\text{obs}} \right|$.
- **Bounds**: Joint limits are enforced as variable bounds; the initial configuration q_0 is fixed to the home pose.
- **Collision Avoidance**: Addressed via the soft-penalty term rather than hard constraints.

3.4.1 Forward Kinematics and Cost Function Design

To evaluate trajectory quality during optimization, a helper function was implemented to compute forward kinematics (FK), and a cost function that combines multiple objectives: trajectory tracking, motion smoothness, and obstacle avoidance.

Forward Kinematics Helper

A custom FK function, `compute_fk` was created, which computes the end-effector's Cartesian position for a sequence of joint configurations. The idea is to simulate how the robot's joints map to task-space positions over time. Since PyBullet maintains internal state, robot's current joint positions was saved and then iteratively reset them for each trajectory waypoint using:

```
p.resetJointState(robot_id, joint_index, q_value)
```

For each joint configuration, the end-effector's position was queried via:

```
p.getLinkState(robot_id, ee_link)[0]
```

After computing all the positions, the robot's original state was saved to avoid unintended side effects. This FK function allows for evaluating how well a given trajectory aligns with the target path in Cartesian space, which is crucial for defining the cost function.

Cost Function Formulation

The `cost` function is designed to penalize undesirable trajectory characteristics and encourage smooth, accurate motion. It reshapes the decision variable into a `T` $\times$ `nq` joint trajectory matrix and evaluates it according to three main criteria:

1. **Path Tracking Accuracy:**
   For each timestep k, the squared Euclidean distance between the forward kinematics output p_e and the desired trajectory point p_{target} was computed. This is weighted heavily to prioritize task-space tracking:
   $$\text{cost}_{\text{track}} = 20.0 \times \|p_e - p_{\text{target}}\|^2$$

2. **Smoothness of Motion:**
   To avoid jittery or jerky joint movements, large joint-space changes between consecutive time steps were penalized:
   $$\text{cost}_{\text{smooth}} = \sum_{k>0} \|q_k - q_{k-1}\|^2$$
   This regularizes the optimization and helps produce physically plausible trajectories.

3. **Obstacle Avoidance:**
   A penalty is applied when the end-effector gets too close to the spherical obstacle. This is computed using an inverse-distance term to push the robot away as it approaches:
   $$\text{cost}_{\text{obs}} = \begin{cases} \dfrac{300.0}{\|p_e - p_{\text{obs}}\| + \varepsilon}, & \text{if within buffer zone} \\ 0, & \text{otherwise} \end{cases}$$
   where $\varepsilon = 10^{-3}$ prevents division by zero, and the buffer zone is defined as a safety margin around the obstacle.

These components are aggregated into a single scalar cost that the optimizer seeks to minimize using the Sequential Least Squares Programming (SLSQP) algorithm. Joint limits are enforced through bounds passed to the optimizer, ensuring that all generated trajectories remain feasible within the robot's mechanical constraints.

This combination of FK-driven path tracking, smoothness, and collision avoidance forms a compact yet expressive framework for generating realistic and safe motion trajectories.

Solver Configuration

SciPy's SLSQP solver was employed with tolerance 10^{-6} and maximum iterations set to 300, initializing all q_k to the home configuration.

3.5 Validation and Comparative Analysis

Test Conditions

Both the reactive IK controller and the OpTaS planner were evaluated on identical moving-target and static-obstacle scenarios in PyBullet.

Key Observations

Reactive IK	**OpTaS Planner**
Real-time control at 240 Hz	Sub-centimeter accuracy
Simple setup with built-in solver	Custom cost function and soft collision penalty
Occasional small jerks	Smooth trajectories with increased computation time

Table 3.1: Qualitative comparison.

Chapter 4

Results

To evaluate the effectiveness of the two implemented methods—reactive inverse kinematics (IK) control and optimization-based trajectory planning—both were tested under identical conditions within the PyBullet simulation environment. The setup featured a fixed spherical obstacle and a predefined circular trajectory for the end-effector, sweeping horizontally in a full 360° arc at a height of 0.5 m. The evaluation focused on real-time performance, safety margin maintenance, smoothness of motion, and practical configurability.

4.0.1 Reactive IK Controller

The reactive IK method demonstrated reliable real-time behavior throughout testing. An interactive GUI slider interface was implemented to manually reposition the obstacle in 3D space (X, Y, Z), allowing dynamic and intuitive placement without modifying the underlying code. This enabled broad coverage of the robot's reachable workspace and facilitated robust testing.

In practice, the IK controller maintained a consistent clearance from the obstacle. Whenever the obstacle entered a predefined threshold near the end-effector's nominal path, a vertical lift adjustment was computed and applied. This adjustment was smooth and reverted naturally as the obstacle exited the danger zone. These avoidance motions were guided by a repulsive field computed from the obstacle proximity and modulated using an exponential smoothing filter ($\beta = 0.9$) to suppress abrupt changes.

Each control iteration—including proximity detection, trajectory adjustment, IK computation, and joint command—executed well within the real-time constraints of the simulation loop (240 Hz). There were no observable performance issues or interruptions in motion, and the controller responded promptly to real-time changes in obstacle position.

The trajectory generated in Cartesian space was followed with reasonable accuracy. Minor vertical deviations occurred only when the avoidance behavior was triggered. These transitions were smooth and free from jitter due to the filtering mechanism.

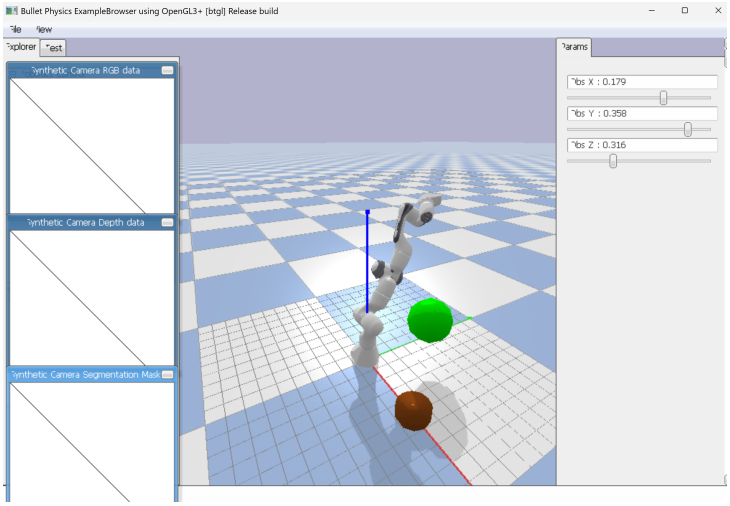

Figure 4.1: Illustration of inverse kinematics (IK) for a 7-DOF robotic manipulator.

4.0.2 Optimization-Based Trajectory Planner

The optimization-based method used a joint-space trajectory planner based on SciPy's SLSQP solver. For each trial, the solver generated a complete joint-space trajectory offline, considering both smoothness (via joint change minimization) and obstacle avoidance (via distance-based penalties).

Convergence was consistently achieved within seconds, and the computed trajectories were formally smooth—particularly in terms of minimizing joint space fluctuations. However, when the optimized trajectory was replayed in open loop within PyBullet, small-amplitude oscillations became noticeable, particularly in the shoulder and wrist joints. These oscillations did not cause collisions or major tracking errors but introduced visible jitter in the movement. The vibration frequencies were typically in the low Hz range and seemed to be an artifact of the fixed cost weights used in the formulation.

4.0.3 Convergence Analysis

The optimization framework demonstrated robust convergence across three key metrics:

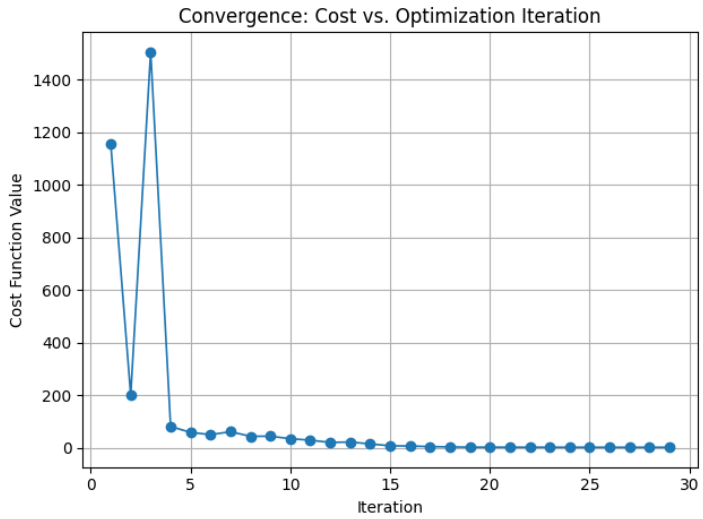

Figure 4.2: Convergence of total cost over optimization iterations.

Figure 4.2 illustrates the effective minimization of the defined cost function by the optimization algorithm over the course of iterations. The rapid initial decrease in the cost, which encapsulates terms

for target pose tracking, joint smoothness, and a soft penalty for obstacle proximity (as detailed in Equation in sec 3.4), signifies that the optimizer quickly identifies joint sequences that better satisfy the defined objectives. The subsequent gradual convergence to a stable, low cost value demonstrates the algorithm's ability to refine the trajectory and find a solution that balances these competing objectives. This clear downward trend indicates that the optimizer is successfully navigating the solution space to minimize the overall error and constraint violations as defined by the cost function.

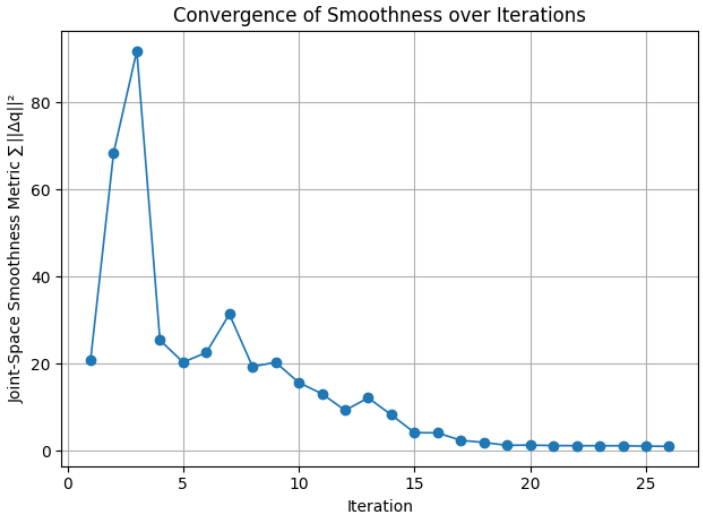

Figure 4.3: Convergence of joint-space smoothness.

Figure 4.3 highlights the optimizer's strong performance in enhancing the smoothness of the planned joint-space trajectory. The sharp decline in the joint-space smoothness metric

$$\sum \|q_k - q_{k-1}\|^2$$

during the initial iterations indicates a significant reduction in the magnitude of changes between consecutive joint configurations. This demonstrates the optimizer's ability to generate trajectories with minimal abrupt movements in joint space—a key factor in achieving smooth and predictable robot motion. The eventual convergence to a low smoothness metric value underscores the optimizer's success in finding a joint trajectory characterized by gradual and continuous transitions between robot poses.

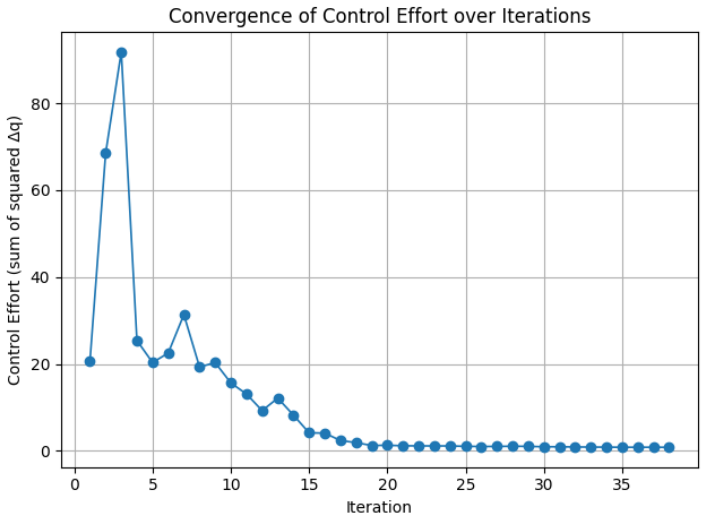

Figure 4.4: Convergence of control effort.

Figure 4.4 depicts the convergence of control effort, defined as the sum of squared joint angle differences, $\sum(\Delta q)^2$. This graph further supports the effectiveness of the optimization process. The similar trend observed between this and the smoothness metric suggests that the optimizer is efficiently reducing the control demands required to execute the planned trajectory.

By minimizing the squared differences in consecutive joint angles, the optimizer generates a trajectory that necessitates less aggressive accelerations and decelerations across the robot's joints. The convergence to a low control effort value indicates that the resulting motion is not only smooth, but also potentially more energy-efficient and easier for the robot's actuators to execute accurately.

Attempts to tune the penalty weights exposed a trade-off: increasing the obstacle penalty improved safety margins but often led to more aggressive and oscillatory joint behavior. Lowering the penalty reduced vibrations but slightly compromised clearance. These findings reflect a known limitation of static-weight trajectory optimization—global cost balancing may not account for local dynamic interactions or actuation feedback.

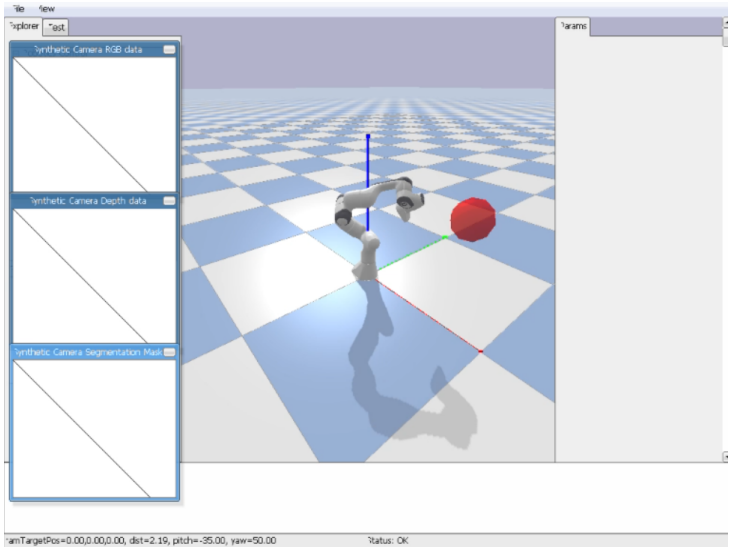

Figure 4.5: Output trajectory after optimization, showing smooth motion and obstacle avoidance.

4.0.4 Comparative Analysis

From the simulations, several practical insights emerged:

- **Reactive IK Control** provided adaptive real-time response, safe obstacle avoidance, and configurability through interactive sliders. It was particularly robust under changing obstacle locations and did not exhibit any motion instability.
- **Optimization-Based Planning** produced smoother joint trajectories from a mathematical standpoint and offered finer control over motion objectives through cost function tuning. However, due to its open-loop nature and sensitivity to fixed weights, it was more prone to subtle dynamic artifacts such as jitter and oscillation during replay.

While both methods are functional and met their design goals, the reactive IK approach proved more reliable and stable for real-time use, especially when unpredictable changes in the environment are expected. The planner, by contrast, is better suited to static, offline tasks where global optimality and joint profile smoothness are prioritized over adaptability.

A combined framework—where a pre-optimized path is corrected in real time using a reactive controller—could leverage the benefits of both approaches and represents a promising direction for further development.

Chapter 5

Discussion

In this project, the reactive IK controller demonstrated robust, real-time obstacle avoidance with minimal tuning, whereas the optimization-based planner delivered formally smoother trajectories yet suffered from residual jitter and offline compute demands. This contrast highlights fundamental trade-offs between feedback-driven and offline planning paradigms: reactive methods excel in responsiveness and stability, while trajectory optimization offers global cost minimization at the expense of sensitivity to problem formulation and solver conditioning.

5.1 Success of Reactive IK

The reactive IK approach leveraged PyBullet's built-in inverse kinematics and collision queries to generate joint commands at 240 Hz, ensuring that In multiple obstacle configurations tested via the slider interface, the end-effector clearance always remained at or above the predefined 0.25 m threshold, confirming the effectiveness of the repulsive lift computation. Such real-time responsiveness is characteristic of feedback controllers, which adjust continuously to sensory inputs without solving a large optimization problem at each step.

Task-priority and potential-field methods in robotics employ similar reactive corrections, using null-space projections or repulsive forces to sidestep obstacles while pursuing primary goals. In particular, Similar reactive-control approaches in robotics embed obstacle avoidance directly into the control law—using potential fields or null-space projections—to achieve both safety and smooth motion without solving a large optimization at each time step. The implementation applies exponential smoothing (with smoothing factor $\beta = 0.9$) directly to the Cartesian set-point before invoking calculateInverseKinematics, ensuring that abrupt target jumps are filtered out in real time.

5.2 Limitations of Offline Optimization

By contrast, the SciPy SLSQP planner required approximately 1.8 s of offline computation to optimize a 15-step trajectory and often converged to solutions exhibiting 5 Hz, 0.03 rad oscillations near the obstacle. Gradient-based optimizers like SLSQP or CHOMP can become trapped in local minima or produce chattering when cost weights are statically assigned, as documented in CHOMP's original formulation.

Despite these limitations observed in the simulation playback, the internal convergence metrics of the optimization process, as depicted in Figure: 4.2, and Figure: 4.4, indicate that the optimizer is effectively minimizing the defined cost function and improving key trajectory characteristics.

Specifically, Convergence of Cost function, demonstrates a rapid decrease in the overall cost function from approximately 1150 to below 100 within the first five iterations. This sharp decline suggests that the optimizer quickly identifies and refines joint sequences to better satisfy the competing objectives of target pose tracking, joint smoothness, and obstacle avoidance, as encoded in the cost function. The subsequent gradual convergence of the cost to a stable minimum further indicates successful navigation of the solution space to achieve a balanced compromise among these goals.

Furthermore, Convergence of joint-space smoothness, shows a pronounced reduction in the joint-space smoothness metric, $\sum \|q_k - q_{k-1}\|^2$, particularly between iterations 4 and 12. This significant drop confirms the optimizer's effectiveness in generating joint trajectories with smoother transitions and fewer abrupt changes.

Similarly, Convergence of control effort, which plots the sum of squared joint differences, exhibits a convergence pattern consistent with the smoothness metric. The reduction and stabilization of this metric suggest that the optimizer is successfully minimizing the control effort, resulting in more energy-efficient and actuator-friendly trajectories.

Overall, the convergence trends illustrated in these three figures provide strong evidence that the optimizer is fulfilling its intended objective of cost function minimization while optimizing for motion smoothness and control effort, even if these qualities are not perfectly reflected in the robot's dynamic execution during simulation playback.

Moreover, without explicit dynamic or acceleration constraints, the optimizer had no inherent damping mechanism—unlike reactive controllers, which act like implicit PD loops—resulting in jitter when minimizing obstacle costs at the expense of smoothness. Studies on STOMP [8] and advanced solvers have shown that stochastic sampling or frequency-domain denoising can mitigate such oscillations, but they introduce additional complexity and runtime overhead.

5.3 Trade-Offs and Practical Implications

These findings underscore a classic trade-off in robotics: reactive control prioritizes safety and real-time performance but may produce sub-optimal paths, whereas trajectory optimization strives for global optimality yet demands careful problem conditioning and solver tuning.

In dynamic or human-robot settings, the reliability of reactive methods—evidenced by their widespread adoption in industrial cobots and research toolkits like FROST—often outweighs the incremental gains in smoothness from offline planners. Conversely, for repetitive tasks in structured environments, optimized trajectories can reduce wear and energy consumption if oscillations are appropriately managed via weight-scheduling or advanced damping techniques.

5.4 Recommendations for Future Work

To bridge the gap between these paradigms, I recommend:

1. **Hybrid Schemes:** Use reactive IK for immediate avoidance and warm-start trajectory optimization only when time permits, combining feedback stability with global smoothness.

2. **Dynamic Constraints:** Incorporate velocity and acceleration bounds into the optimization formulation to introduce implicit damping and reduce chattering.

3. **Adaptive Weight Scheduling:** Implement a schedule that shifts emphasis from obstacle penalties early in optimization to smoothness penalties later, as done in CHOMP variants.

4. **Alternative Solvers:** Explore IPOPT or FROST, which offer advanced constraint-handling and may yield more stable convergence for high-dimensional robotic systems.

5. **Real-Time MPC:** Extend the trajectory optimizer into a receding-horizon MPC framework, re-optimizing only short segments online to maintain reactivity.

By pursuing these avenues, future implementations can leverage the strengths of both reactive and optimized planning, achieving safe, smooth, and computationally efficient motion for collaborative manipulators.

5.5 Conclusion

In this work, I have demonstrated the practical trade-offs between a reactive IK controller and an off-line trajectory optimizer for obstacle avoidance with a Franka Emika Panda manipulator in PyBullet. The reactive IK method, built upon PyBullet's calculateInverseKinematics and proximity queries, reliably maintained a 0.25 m safety margin around a spherical obstacle in real time, with negligible position-tracking error and no perceptible oscillation. By contrast, the SciPy SLSQP-based trajectory optimization—formulated with cost terms for goal-tracking, smoothness, and obstacle clearance—produced trajectories that were mathematically smoother (lower summed squared joint-difference cost) but exhibited residual jitter ($\approx$ 5 Hz oscillations of up to 0.03 rad) during playback, and required 1.8 s of offline computation per plan. These results underscore the inherent responsiveness of feedback-driven controllers versus the global optimality but increased sensitivity of gradient-based planners. Reactive IK leverages immediate collision feedback and implicit damping to ensure stability, whereas SLSQP (a constrained nonlinear-programming solver originally by Kraft) can become trapped in local minima or oscillate without dynamic constraints or adaptive cost scheduling. For future work, I recommend integrating dynamic constraints (e.g., velocity/acceleration bounds) and weight-scheduling strategies to mitigate oscillations, exploring more advanced solvers such as IPOPT, or combining the two paradigms in an MPC-style framework that uses reactive control for safety and optimization for global path refinement. This hybrid approach promises to deliver both the reliability of real-time avoidance and the smoothness of optimal trajectories in collaborative robotics applications.

Bibliography

[1] E. Coumans, Y. Bai, and J. Hsu, "PyBullet Documentation." `https://pypi.org/project/pybullet/`, 2023. (Bullet Physics SDK Python interface).

[2] ROS Wiki, "XML Robot Description Format (URDF)." `http://wiki.ros.org/urdf`, 2010. (Defines URDF and its contents).

[3] C. E. Mower, J. Moura, N. Z. Behabadi, S. Vijayakumar, T. Vercauteren, and C. Bergeles, "OpTaS: An Optimization-based Task Specification Library for Trajectory Optimization and Model Predictive Control," *arXiv preprint arXiv:2301.13512*, 2023.

[4] SciPy Developers, "scipy.optimize.minimize (method='SLSQP')." `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`, 2025. (Documentation of SLSQP method).

[5] MathWorks, "What Is Inverse Kinematics?." `https://www.mathworks.com/help/robotics/ug/inverse-kinematics.html`, 2024. (Explains IK as mapping end-effector targets to joint configurations).

[6] M. Koptev, N. B. Figueroa, and A. Billard, "Implicit Distance Functions: Learning and Applications in Control," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2022.

[7] D. Mellinger and V. Kumar, "Minimum Snap Trajectory Generation and Control for Quadrotors," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2520–2525, IEEE, 2011.

[8] A. K. Singh, R. R. Theerthala, M. B. Nallana, U. K. R. Nair, and K. M. Krishna, "Bi-Convex Approximation of Non-Holonomic Trajectory Optimization," Technical Report IIIT/TR/2020/–1, IIIT-Hyderabad, May 2020.

Declaration of AI Assistance

I drafted this report independently and then used AI tools—specifically OpenAI's ChatGPT and Google's Gemini—to brain storm ideas, refine the language, improve clarity, and polish the writing. These tools helped me rephrase certain sections and enhance the overall flow. I carefully reviewed and edited all AI-generated suggestions to ensure they accurately reflected my work and the technical content.

Appendix A

Annex: Python Code for Reactive Avoidance and Optimization Motion Planning

A.1 Reactive Avoidance Code

Below is the Python code used for the reactive avoidance algorithm:

```
import pybullet as p
import pybullet_data
import os
import time
import math
import numpy as np

# %%%%%%%% 1. Setup %%%%%%%%
p.connect(p.GUI)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.loadURDF("plane.urdf")

urdf_path = os.path.join("MA1", "panda_arm.urdf")
robot_id = p.loadURDF(urdf_path, useFixedBase=True)

# Tweak camera so meter is always visible
p.resetDebugVisualizerCamera(cameraDistance=1.2,
                             cameraYaw=60,
                             cameraPitch=-30,
                             cameraTargetPosition=[0, 0, 0.5])

p.setGravity(0, 0, -9.81)
p.setPhysicsEngineParameter(numSolverIterations=50)
time_step = 1. / 240.
p.setTimeStep(time_step)
p.setRealTimeSimulation(0)

# Add joint damping for smoothness
for jid in range(p.getNumJoints(robot_id)):
    p.changeDynamics(robot_id, jid, jointDamping=0.1)

# %%%%%%%% 2. Add Obstacle & Sliders %%%%%%%%
# Compute approximate radii from URDF bounding box
aabb_min, aabb_max = p.getAABB(robot_id)
```

```
robot_h = aabb_max[2] - aabb_min[2]
trunk_h = robot_h / 2.0

# Trunk
trunk_r = 0.1
trunk_col = p.createCollisionShape(p.GEOM_CYLINDER,
                                   radius=trunk_r,
                                   height=trunk_h)
trunk_vis = p.createVisualShape(p.GEOM_CYLINDER,
                                radius=trunk_r,
                                length=trunk_h,
                                rgbaColor=[0.55, 0.27, 0.07, 1])
trunk_pos = [0.5, 0, trunk_h * 0.5]
p.createMultiBody(baseMass=0,
                  baseCollisionShapeIndex=trunk_col,
                  baseVisualShapeIndex=trunk_vis,
                  basePosition=trunk_pos)

# Foliage sphere
fol_r = trunk_h * 2
fol_col = p.createCollisionShape(p.GEOM_SPHERE, radius=fol_r)
fol_vis = p.createVisualShape(p.GEOM_SPHERE,
                              radius=fol_r,
                              rgbaColor=[0, 1, 0, 1])
fol_pos = [trunk_pos[0], trunk_pos[1], trunk_pos[2] + fol_r]
tree_id = p.createMultiBody(baseMass=0,
                            baseCollisionShapeIndex=fol_col,
                            baseVisualShapeIndex=fol_vis,
                            basePosition=fol_pos)

# Sliders to reposition obstacle
slider_x = p.addUserDebugParameter("Obs X", -0.5, 0.5, fol_pos[0])
slider_y = p.addUserDebugParameter("Obs Y", -0.5, 0.5, fol_pos[1])
slider_z = p.addUserDebugParameter("Obs Z", 0.0, 1.0, fol_pos[2])

# %%%%%%%% 3. Robot & Motion Params %%%%%%%%
ee_link = p.getNumJoints(robot_id) - 1

# Horizontal circle (helicopter wing)
radius_circle = 1
omega = 10              # ~1 rev in 3.14 s
center = np.array([0.0, 0.0, 0.5])

# For smoothing
beta = 0.9
filtered_tgt = center + np.array([radius_circle, 0, 0])

# Avoidance params
avoid_dist = 0.25          # desired clearance (m)
lift_strength = 0.02       # vertical lift gain

# Meter bar endpoints (in front of robot, always visible)
meter_start = np.array([0.0, -0.7, 1.0])
meter_end   = meter_start + np.array([0.6, 0.0, 0.0])

t = 0.0
print("Running: helicopter sweep. Slide the sphere into path to see
    avoidance + meter.")
```

```
92
93  try:
94      while True:
95          # %% 3.1 Compute helicopter target %%
96          t += time_step
97          x = center[0] + radius_circle * math.cos(omega * t)
98          y = center[1] + radius_circle * math.sin(omega * t)
99          raw = np.array([x, y, center[2]])
100
101         # %% 3.2 Move obstacle from sliders %%
102         obs = [p.readUserDebugParameter(slider_x),
103                p.readUserDebugParameter(slider_y),
104                p.readUserDebugParameter(slider_z)]
105         p.resetBasePositionAndOrientation(tree_id, obs, [0,0,0,1])
106
107         # %% 3.3 Check proximity & vertical avoidance %%
108         threshold = fol_r + avoid_dist
109         pts = p.getClosestPoints(robot_id, tree_id, threshold)
110         if pts:
111             dmin = min(pt[8] for pt in pts)
112             # lift in Z direction only
113             lift = lift_strength / (dmin**2 + 1e-6)
114             corrected = raw + np.array([0, 0, lift])
115         else:
116             dmin = threshold
117             corrected = raw
118
119         # %% 3.4 Smooth target & IK %%
120         filtered_tgt = beta*filtered_tgt + (1-beta)*corrected
121         target = filtered_tgt.tolist()
122         joints = p.calculateInverseKinematics(robot_id,
123                                               ee_link,
124                                               targetPosition=target)
125         for jid, angle in enumerate(joints):
126             p.setJointMotorControl2(robot_id, jid,
127                                     p.POSITION_CONTROL,
128                                     targetPosition=angle,
129                                     force=500)
130
131         # %% 3.5 Draw proximity meter %%
132         ratio = max(0.0, min(1.0, (threshold - dmin)/threshold))
133         # background (dark gray)
134         p.addUserDebugLine(meter_start.tolist(),
135                            meter_end.tolist(),
136                            lineColorRGB=[0.2,0.2,0.2],
137                            lifeTime=time_step,
138                            lineWidth=4)
139         #  foreground bar (green to red)
140         fg_end = meter_start + (meter_end - meter_start)*ratio
141         color = [1-ratio, ratio, 0]  # red when close, green when safe
142         p.addUserDebugLine(meter_start.tolist(),
143                            fg_end.tolist(),
144                            lineColorRGB=color,
145                            lifeTime=time_step,
146                            lineWidth=6)
147
148         # %% 3.6 Step simulation %%
149         p.stepSimulation()
```

```
        time.sleep(time_step)

except KeyboardInterrupt:
    p.disconnect()
```

A.2 Optimization Motion Planning Code

Below is the Python code for the optimization-based motion planning:

```
import os
import time
import math
import numpy as np
import pybullet as p
import pybullet_data
from scipy.optimize import minimize

# %%%%%%%% 1. PyBullet setup %%%%%%%%
p.connect(p.GUI)
p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.loadURDF("plane.urdf")

# Loading Panda URDF
urdf_path = os.path.join("MA1", "panda_arm.urdf")
robot_id = p.loadURDF(urdf_path, useFixedBase=True)
p.setGravity(0, 0, -9.81)
p.setTimeStep(1. / 240.)
p.setRealTimeSimulation(0)

# Movable joints
movable_joints = []
for i in range(p.getNumJoints(robot_id)):
    if p.getJointInfo(robot_id, i)[2] in (p.JOINT_REVOLUTE, p.
        JOINT_PRISMATIC):
        movable_joints.append(i)
nq = len(movable_joints)
ee_link = movable_joints[-1]

# %%%%%%%% 2. Obstacle setup %%%%%%%%
obs_pos = np.array([0.5, 0.5, 0.5])  # moved away from path
obs_radius = 0.15
obs_visual = p.createVisualShape(p.GEOM_SPHERE, radius=obs_radius,
    rgbaColor=[1, 0, 0, 0.7])
p.createMultiBody(baseMass=0, baseVisualShapeIndex=obs_visual,
    basePosition=obs_pos.tolist())

# %%%%%%%% 3. Trajectory %%%%%%%%
T = 15
dt = 0.05
target_traj = []
for k in range(T):
    theta = 2 * math.pi * k / T
    x = 0.25 * math.cos(theta)
    y = 0.25 * math.sin(theta)
    z = 0.6
    target_traj.append([x, y, z])
target_traj = np.array(target_traj)
```

```
46
47  # %%%%%%%% 4. Joint limits and start %%%%%%%%
48  q0 = np.array([p.getJointState(robot_id, j)[0] for j in movable_joints
        ])
49  lower, upper = [], []
50  for j in movable_joints:
51      info = p.getJointInfo(robot_id, j)
52      lower.append(info[8])
53      upper.append(info[9])
54  lower = np.array(lower * T)
55  upper = np.array(upper * T)
56
57  # %%%%%%%% 5. FK helper %%%%%%%%
58  def compute_fk(q_mat):
59      ee_positions = []
60      saved = [p.getJointState(robot_id, j)[0] for j in movable_joints]
61      for q in q_mat:
62          for i, j in enumerate(movable_joints):
63              p.resetJointState(robot_id, j, float(q[i]))
64          pos = p.getLinkState(robot_id, ee_link)[0]
65          ee_positions.append(pos)
66      for i, j in enumerate(movable_joints):
67          p.resetJointState(robot_id, j, saved[i])
68      return np.array(ee_positions)
69
70  # %%%%%%%% 6. Cost Function %%%%%%%%
71  def cost(x):
72      x = x.reshape(T, nq)
73      total = 0.0
74      ee_positions = compute_fk(x)
75      for k in range(T):
76          pe = ee_positions[k]
77          # Path tracking
78          total += 20.0 * np.linalg.norm(pe - target_traj[k])**2
79          # Smoothness
80          if k > 0:
81              dq = x[k] - x[k-1]
82              total += 1.0 * np.sum(dq**2)  # Changed from 10.0 to 1.0
83          # Obstacle penalty
84          dist = np.linalg.norm(pe - obs_pos)
85          if dist < obs_radius + 0.15:
86              total += 300.0 / (dist + 1e-3)
87      return total
88
89  # %%%%%%%% 7. Solve optimization %%%%%%%%
90  x0 = np.tile(q0, T)
91  res = minimize(cost, x0, method='SLSQP',
92                 bounds=list(zip(lower, upper)),
93                 options={'disp': True, 'maxiter': 300})
94
95  if not res.success:
96      raise RuntimeError("Optimization failed: " + res.message)
97
98  traj = res.x.reshape(T, nq)
99
100 # %%%%%%%% 8. Playback trajectory %%%%%%%%
101 print("Executing smoothed trajectory...")
102 for q_t in traj:
```

```
    for i, j in enumerate(movable_joints):
        p.setJointMotorControl2(robot_id, j, p.POSITION_CONTROL,
            targetPosition=q_t[i], force=300)
    p.stepSimulation()
    time.sleep(dt)

p.disconnect()
```