# PickleBot: An Autonomous Pickleball-Playing Mobile Manipulator

CS225A Final Project Report

Hemal Arora  Hannah Watkins  Adhi Daiv

Stanford University  June 9, 2026

## 1. Project Description and Objectives

PickleBot is an autonomous pickleball-playing robot built on a mobile manipulator platform. The system tracks an incoming foam ball using an OptiTrack motion-capture array, predicts its intercept point on a fixed strike plane, plans a joint-space arm trajectory with a specified impact velocity, and executes the swing — all in real time at 100 Hz.

Core objectives:

- Real-time ball tracking and physics-based trajectory prediction via OptiTrack at ∼120 Hz.
- Impact-velocity-controlled strikes using Hermite cubic joint-space trajectories.
- Mobile base lateral repositioning to extend the reachable strike envelope along the Y-axis.
- Safe operation within the Franka FR3's position-dependent joint velocity envelopes.

## 2. Robot Description

### 2.1 Arm

A Franka Emika Panda (FR3) 7-DOF torque-controlled manipulator is controlled via OpenSai using a joint-space PD controller with online trajectory generation (OTG) disabled. Joint gains are listed in Table 1.

Table 1: Joint PD gains (joints 1–7).

| | q1 | q2 | q3 | q4 | q5 | q6 | q7 |
|---|---|---|---|---|---|---|---|
| $k_p$ | 200 | 200 | 200 | 200 | 400 | 600 | 400 |
| $k_v$ | 20 | 20 | 20 | 20 | 28 | 28 | 28 |

### 2.2 Mobile Base

A TidyBot omnidirectional cart serves as the mobile base. During a rally the base continuously tracks the predicted ball intercept Y-coordinate (clamped to a configured range). A Python bridge (`base_bridge.py`) converts world-frame goal poses $[x, y, \theta]$ into the TidyBot odometry frame and forwards them to the base driver.

### 2.3 End-Effector

A custom 3D-printed pickleball paddle is mounted at the Panda flange. The paddle tool-center point (TCP) is located 0.368 m along link7 $+z$ (0.107 m flange offset + 0.261 m to the face center). Inertial properties were computed by approximating the paddle geometry as a solid cuboid:

| Parameter | Value |
|---|---|
| Mass | 0.5 kg |
| Center of mass | [0, 0, 0.2] m from attachment |
| $I_{xx}$ | $7.93 \times 10^{-3}$ kg m$^2$ |
| $I_{yy}$ | $8.55 \times 10^{-3}$ kg m$^2$ |
| $I_{zz}$ | $7.25 \times 10^{-4}$ kg m$^2$ |

### 2.4 OptiTrack

Ball and robot base pose are provided by an OptiTrack infrared motion-capture system (NatNet UDP, ∼120 Hz) mounted to the ceiling around the perimeter of the kitchen. The foam ball (Streaming ID 13) and the TidyBot cart (Streaming ID 11) are tracked as rigid bodies. Retroreflective markers are attached to the foam ball in asymmetric clusters for optimal tracking. The NatNet stream is bridged to Redis by `StreamDataSkeleton.py`. A fixed world frame calibration `world_calibration.json` is applied on the streamer to convert the OptiTrack frame into the world frame, which is used by our controller, ball tracker, and base goals. No wrist cameras are used.

## 3. Simulation World Environment

The simulation runs in the SAI framework (SaiSimulation + SaiGraphics), with the world defined in `world_mmp_panda.urdf`.

- **Coordinate frame:** World $+X$ points toward the net. The robot base is at the world origin; the ground surface is at $z = 0$.
- **Court:** Flat floor plane representing a pickleball court surface.
- **Net:** Visual-only static geometry at $x = 6.71$ m (USA Pickleball standard), with posts, tape, and mesh modeled.
- **Ball:** Dynamic rigid body (25 g, 74 mm diameter hollow plastic in simulation; foam ball on hardware). Parked above the opponent court when idle.
- **Ball launching:** `launch_ball.py` writes a desired start pose and velocity to Redis and increments a launch counter. `simviz.cpp` polls the counter and teleports the ball on the next physics tick. The default launch is a lob from position [5.0, 0.0, 1.6] m with velocity [−6.0, 0.0, 1.5] m/s.

A launcher script (`pickleball/run.sh`) starts the SAI simulator, C++ controller, and Python planner in parallel, then drops the operator into an interactive REPL. The operator types `ball` to launch a new shot, `state` to inspect the current phase, `watch` to monitor Redis keys live, and `quit` to shut everything down.

## 4. Human Interface and Interaction

The operator physically throws a foam ball covered with retroreflective markers into the robot's workspace. OptiTrack detects the ball; the J10 planner begins tracking and computing a strike candidate in the world frame; the base translates laterally ($Y$-axis) so the intercept remains within the arm workspace; the arm continuously re-solves IK using the current base pose; and the arm commits to a swing once the time-to-impact (TTI) enters the commit window.

The bring-up sequence is as follows:

1. `redis-server`, OpenSai controller (`cd OpenSai && sh scripts/launch.sh picklebot_j7.xml`), and Franka Redis driver (`cd OpenSai/drivers/FrankaPanda/redis_driver && sh launch.sh`)
2. Mobile base: `cd` into the `tidybot2` repo, activate the environment with `mamba activate tidybot2`, and run `python3 redis_driver.py` for autonomous operation

3. `StreamDataSkeleton.py`: publishes ball and cart pose to Redis, converting OptiTrack coordinates to the world frame via `world_calibration.json`
4. `base_bridge.py`: translates world-frame base goals to the odometry frame, which is the native command frame of the mobile base
5. `stepj10_impact_velocity_planner.py`: main strike planner

## 5. State Machine and Controllers

The demo is driven by `stepj10_impact_velocity_planner.py`. The system implements an *implicit* finite state machine (FSM) within a single 100 Hz control loop. Each tick evaluates ball visibility, TTI, and trajectory feasibility, and branches into one of six states. Figure 1 shows the state transition diagram.

### 5.1 State Diagram

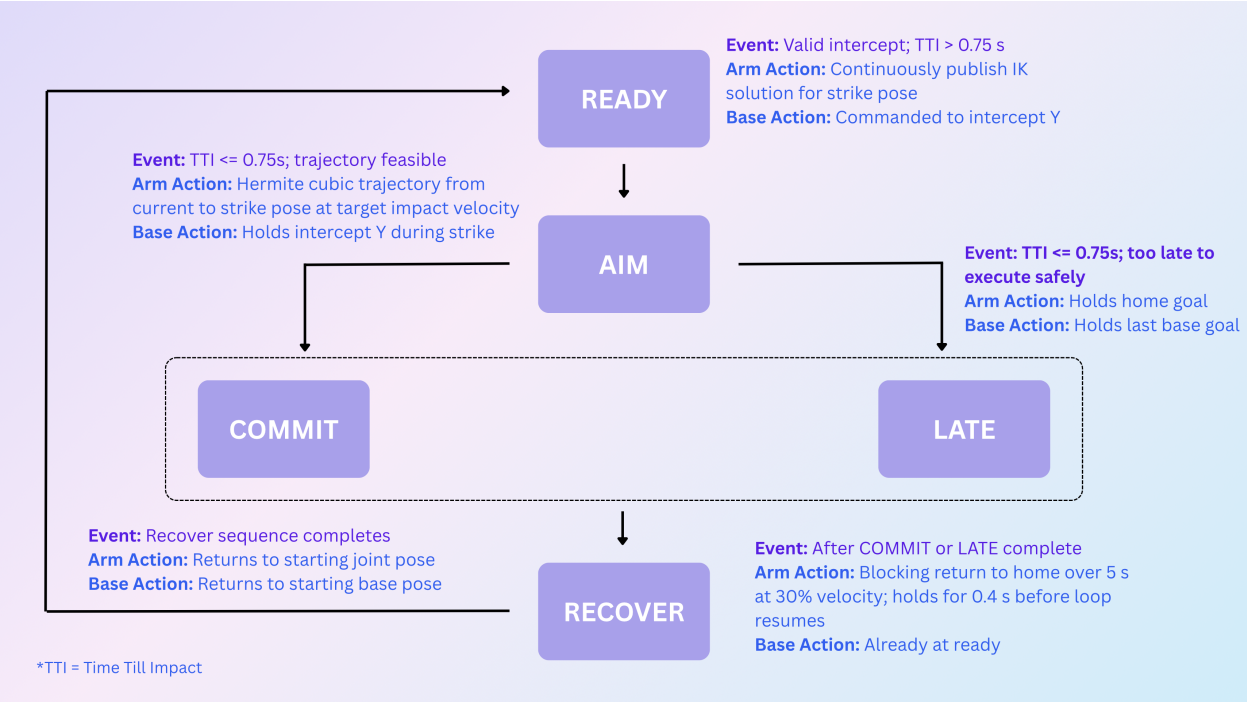


Figure 1: Implicit FSM in `stepj10_impact_velocity_planner.py`.

### 5.2 State Descriptions

Table 2: Implicit FSM states, entry conditions, and actions.

| State | Entry condition | Arm action | Base action |
|---|---|---|---|
| **READY** | No valid intercept exists, or post-impact cooldown timer hasn't elapsed yet (want to allow time for arm to settle) | Return to starting joint pose | Return to starting pose $>0.6$ s |
| **AIM** | Valid intercept; TTI $>$ 0.75 s | Continuously publishes IK solution for strike pose and solution published as joint goal | Commanded to intercept Y, clamped to $[y_{\min}, y_{\max}]$ |
| **COMMIT** | TTI $\leq$ 0.75 s; trajectory feasible. There is enough time remaining (margin $\geq$ 0) and no joint would exceed its velocity limit (peak vel $\leq$ cap) | Arm executes swing — Hermite cubic trajectory from current to strike pose at target impact velocity, followed by decel segment to a stop pose | Holds intercept Y during strike; commanded to ready pose after impact |
| **LATE** | TTI $\leq$ 0.75 s; too late to execute safely | Skips commit; holds home goal | Holds last base goal |
| **RECOVER** | After COMMIT or LATE completes | Blocking return to home over 5 s at 30% velocity; holds for 0.4 s before loop resumes normal tracking | Already at ready — the base was commanded there after COMMIT |

### 5.3 Strike Execution (COMMIT / LATE)

On entering COMMIT or LATE, an inner loop runs at 200 Hz:

1. **Approach** — Hermite cubic trajectory from the current joint state to the strike joint angles, with boundary velocity $\dot{q}_{\text{strike}}$ computed via damped Jacobian pseudoinverse of the paddle sweet-spot Jacobian:
$$\dot{q} = J^\top \left( JJ^\top + \lambda^2 I \right)^{-1} v_{\text{desired}},$$
scaled so no joint exceeds its velocity cap.
2. **Decel** — Hermite cubic from strike joints at $\dot{q}_{\text{strike}}$ to a stop configuration at zero velocity (default 0.18 s), bleeding off the swing after contact.
3. At the moment of contact, actual joint velocities are read back and the measured end-effector velocity in world frame is computed and logged.

### 5.4 Ball Tracker

`FoamBallTracker` maintains a rolling window of OptiTrack samples, applies a median filter for outlier rejection, and fits a physics model with gravity and aerodynamic drag:

$$m\ddot{\mathbf{r}} = m\mathbf{g} - k \left\| \dot{\mathbf{r}} \right\| \dot{\mathbf{r}},$$

with drag coefficient $k = 0.20$ m$^{-1}$ fitted empirically for the foam ball. An intercept on the fixed strike plane (world $x = -0.55$ m in arm frame) is predicted and refreshed every tick. A minimum sample count is required before a prediction is accepted.

### 5.5 Key Controller Parameters

| Parameter | Value |
|---|---|
| Velocity saturation | Enabled (85% of XML limits) |
| Impact velocity target | +0.75 m/s forward, +0.15 m/s upward (world frame) |
| Commit TTI threshold | $\leq 0.75$ s |
| Post-impact return | 5 s at 30% velocity |
| Main loop rate | 100 Hz |
| Impact segment rate | 200 Hz |

## 6. Challenges and Future Work

### 6.1 Cartesian vs. Joint-Space Control

Cartesian `MotionForceTask` control produced slow, unpredictable motion near singularities. Switching to a joint-space PD controller with pre-computed Hermite trajectories yielded faster, repeatable motion and tractable timing analysis.

### 6.2 Ball Tracking and Foam Ball Transition

A simple ballistic model sufficed for the hard pickleball, but switching to a foam ball (to stay within joint torque limits) required a drag-aware model. At short distances, aerodynamic drag dominates the foam ball's trajectory; fitting a drag coefficient ($k = 0.20$ m$^{-1}$) restored reliable intercept predictions.

### 6.3 FR3 Joint Velocity Limits

Disabling software velocity saturation caused Franka safety trips, as the FR3 enforces hardware position-dependent velocity envelopes that aggressive Hermite trajectories violated. Velocity saturation was retained, accepting a slightly slower swing for reliability.

### 6.4 Mobile Base Speed

The TidyBot base used conservative default motion parameters. A custom Python script raises these at runtime, allowing the base to track the ball's intercept Y-coordinate before the arm commits.

### 6.5 Future Work

- **Faster arm without safety trips:** Characterize which joints and configurations produce limit violations when saturation is disabled, and design per-joint trajectory re-timing to recover speed where it is safe.
- **Swing via base rotation:** Rather than relying solely on increasing the arm velocity, the mobile base could be commanded to rotate in synchrony with the swing. The swing would come from the base, and not the arm.
- **Aiming the return:** Incorporate a return target goal (e.g., corners, middle of court) into the impact velocity planner as a step toward genuine rally-capable play.

## 7. Demo Videos and GitHub Repository

| Resource | Link |
|---|---|
| Demo Video | VIDEO LINK |
| GitHub Repository | GITHUB LINK |