Vision-Based Autonomous Quadrotor Landing on a Moving Platform: System Design and Simulation Validation

Leonardo Sandri (2137374), Lorenzo Barbieri (2144036), Alessandro Carotenuto (1847282)

I. Abstract

Autonomous landing on moving platforms addresses a fundamental limitation of quadrotors systems: battery capacity. By enabling battery swapping at mobile charging stations, aerial vehicles can extend mission duration significantly. This work presents a complete system implementation for vision-based autonomous landing on a moving platform, validated through simulation.

We did a full re-implementation of the approach described by Falanga *et al.* (2017) [1], consisting of modular components for quadrotor dynamics, cascaded PID control, vision-based platform detection, Kalman filtering with attitude gating, and adaptive trajectory planning. The system is implemented in Simulink with CoppeliaSim for visualization and camera simulation.

Our implementation follows a progressive two-phase approach: first, a simplified version with known platform trajectories tests the core control and planning components; second, the complete system with unknown trajectories incorporates vision-based detection and Kalman filtering for platform state estimation. We test across multiple platform motions (stationary, linear, circular, figure-8) at constant velocity.

The system demonstrates successful autonomous landing across all tested scenarios, with Kalman filtering providing robust tracking despite vision dropouts.

Results show that a simplified control architecture with careful parameter tuning is sufficient for stable hover-to-landing regimes, though extensions to aggressive maneuvers would require nonlinear control. We discuss design trade-offs relative to the reference work and identify opportunities for real-world validation.

II. Introduction

A. Motivation and Problem Statement

Quadrotor unmanned aerial vehicles have emerged as versatile platforms for inspection, surveillance, and autonomous missions. However, limited battery capacity—typically 15–30 minutes flight time—constrains operational utility. One solution is to land autonomously at mobile charging stations, enabling battery swaps or in-situ charging without human intervention. This extends mission duration and enables scenarios like long-duration surveillance over large or cooperative multi-agent operations.

Traditional fixed landing sites provide stable targets but limit vehicle flexibility. Moving platforms—whether ground vehicles, ships, or other platforms—offer significantly greater operational flexibility. Landing on a moving platform, however, introduces challenges across perception, estimation, planning, and control:

- **Perception**: The landing target must be detected reliably despite changing viewpoint and quadrotor's own motion.
- **Estimation**: Platform state (position, velocity, orientation) must be inferred from camera measurements and predicted despite intermittent vision.
- **Planning**: Feasible trajectories must account for both quadrotor and platform dynamics, with the ability to re-plan as new information arrives.
- **Control**: Precise attitude and thrust control must track desired trajectories while stabilizing against disturbances.

B. Related Work and Reference Implementation

Falanga *et al.* [1] demonstrated a complete system for vision-based autonomous landing on moving platforms using only onboard sensing and computation. Their work employed:

- Visual-inertial odometry for quadrotor pose estimation
- Tag-based platform detection
- Extended Kalman filtering with a constant-velocity platform model
- Minimum-jerk trajectory optimization with feasibility checking
- Cascaded nonlinear feedback control with explicit feedforward terms

This architecture successfully landed a quadrotor on moving platforms.

C. Contribution and Scope

This work presents a complete reimplementation of the moving-platform landing system in Simulink and CoppeliaSim. Rather than extending or validating a specific aspect, we reproduce the full system as described in Falanga *et al.* [1], examining design choices, trade-offs, and implementation details.

Our contributions are:

1) **Modular architecture**: Decompose the system into independently testable components (dynamics, control, vision, filtering, planning), enabling systematic development and testing.
2) **Progressive implementation approach**: Build the system in two phases—first with known platform trajectories to establish the control and planning foundation,

then extending to unknown trajectories by adding vision-based detection and Kalman filtering.

3) **Practical design insights**: Simplified PID control, empirical attitude gating for noise rejection, and predictive feedforward trajectory filtering-techniques that work well in practice despite diverging from the paper's optimal-control approach.

The system is implemented and tested through simulation only. Ground-truth position and attitude are provided by CoppeliaSim for visualization and camera rendering, avoiding the complexity of visual odometry. This simplification is appropriate for testing the landing pipeline; visual odometry would be essential for real hardware but is a separated concern from the platform-landing problem.

D. *Report Organization*

Section III provides system architecture and data flow. Section IV details each module: dynamics and control, vision and filtering, trajectory planning and state machine, and Simulink–CoppeliaSim integration. Section V describes the progressive two-phase implementation approach and platform trajectories tested. Section VI presents successful landings and key performance observations. Section VII compares our approach to the reference work and discusses design trade-offs. Section VIII outlines limitations and future work directions

III. System Overview

A. *System Architecture*

The system comprises functional modules executed in Simulink, interfaced with CoppeliaSim for visualization and camera simulation:

1) **State Machine**: Determines operational mode (takeoff, search, tracking, landing) based on altitude, detection status, and tracking error.
2) **Trajectory Generation**: Provided as desired position, velocity, and acceleration references based on the current flight phase, accounting for platform motion prediction when tracking.
3) **Position Controller**: Converts trajectory tracking error into desired attitude angles and vertical thrust, incorporating gravity compensation and anti-windup.
4) **Attitude Controller**: Converts attitude tracking error into torque outputs, that are then mapped to motor commands.
5) **Dynamics**: Integrates accelerations (from thrust/-torques and gravity) in Simulink to update position and attitude. CoppeliaSim receives state updates for visualization only.
6) **Vision**: Detects platform tag in camera images, solves 3D position via PnP, transforms to world coordinates.
7) **Kalman Filter**: Estimates platform state (position, velocity) from intermittent camera measurements using a constant-velocity motion model.

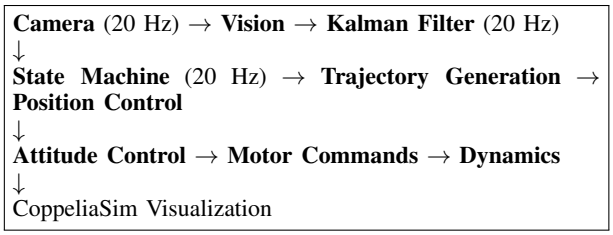

Fig. 1. Data flow between modules.

B. *Data Flow*

Camera measurements, Kalman filtering, position control, and attitude control all run synchronously at 20 Hz (0.05s sample time), determined by the Simulink fixed-step solver. All S-functions share the same discrete sample rate - there is no rate decoupling between vision and control.

C. *Rate Adaptation*

The Kalman filter runs at 20 Hz, synchronized with camera measurements and control loops. When platform detection succeeds (detected=1), the filter performs prediction + correction. When detection fails (detected=0), it performs prediction-only to maintain state estimates between successful detections, providing robustness to intermittent vision loss.

D. *Simulation Environment*

CoppeliaSim operates as a visualization and sensor server, providing:

- Camera simulation (image generation, ray tracing, 100 Hz internal render step)
- Ground truth poses via Remote API (for coordinate transformations and validation)
- Real-time 3D visualization of quadrotor and platform motion

Physics simulation (quadrotor dynamics, platform motion, gravity, inertia) is computed entirely in MATLAB/Simulink. Synchronization is master-slave synchronous mode: after each control cycle (20 Hz), Simulink pushes position/orientation states to CoppeliaSim via *simxSetObjectPosition/Orientation*, triggers one simulation step via *simxSynchronousTrigger*, waits for render completion via *simxGetPingTime*, then continues. This ensures deterministic, reproducible behavior with CoppeliaSim acting as a sensor-in-the-loop simulator

IV. Implementation

This section details the core system modules: quadrotor dynamics and control strategy, platform detection and state estimation, trajectory planning and state machine logic, and integration between Simulink and CoppeliaSim.

A. *Quadrotor Dynamics and Control Strategy*

*1) **Quadrotor Dynamics**:* We model the quadrotor as a rigid body with six degrees of freedom: position $\mathbf{p} =$

$[x, y, z]^T$ and attitude represented by Euler angles $\boldsymbol{\theta} = [\phi, \theta, \psi]^T$ (roll, pitch, yaw). The dynamic model is:

$$\ddot{x} = -\frac{T}{m}(\cos\psi \sin\theta \cos\phi + \sin\psi \sin\phi) \tag{1}$$

$$\ddot{y} = -\frac{T}{m}(\sin\psi \sin\theta \cos\phi - \cos\psi \sin\phi) \tag{2}$$

$$\ddot{z} = -\frac{T}{m}\cos\theta \cos\phi + g \tag{3}$$

$$\ddot{\phi} = \frac{\tau_\phi}{I_x} \tag{4}$$

$$\ddot{\theta} = \frac{\tau_\theta}{I_y} \tag{5}$$

$$\ddot{\psi} = \frac{\tau_\psi}{I_z} \tag{6}$$

where T is total vertical thrust, $\tau_{\phi,\theta,\psi}$ are roll/pitch/yaw torques, m is mass, I_x, I_y, I_z are moments of inertia, and g is gravity. This model taken from lecture slides assumes:

- Small angles (though not enforced during control; bounded empirically via attitude limits)
- Negligible aerodynamic drag
- Gyroscopic coupling neglected
- Rigid body assumption (no deformation)
- Instantaneous motor response (no motor dynamics)
- symmetric shape

The model is integrated in Simulink using a continuous integrator. CoppeliaSim receives only the resulting state (position, orientation) for rendering; all physics computation occurs in MATLAB.

*2) **Control Architecture**:* Control is cascaded in two levels: position control (outer loop) and attitude control (inner loop).

a) Position Control: The position controller receives the desired trajectory $\mathbf{p}_{\text{desired}}$ (from trajectory generation) and the actual state $\mathbf{p}_{\text{actual}}$ (from dynamics). It computes the tracking error:

$$e_{\mathbf{p}} = \mathbf{p}_{\text{desired}} - \mathbf{p}_{\text{actual}} \tag{7}$$

This error is then used to generate desired attitude angles and vertical thrust via PID control.

For the vertical (Z):

$$T = \frac{m}{\cos\phi\cos\theta}\left(g - K_{p,z}e_z - K_{d,z}\dot{e}_z - K_{i,z}\int e_z\, dt - \ddot{z}_{\text{desired}}\right) \tag{8}$$

where $K_{p,z}, K_{d,z}, K_{i,z}$ are position control gains. The term g provides gravity compensation; the denominator $\cos\phi\cos\theta$ accounts for the tilt of the thrust vector.
For horizontal axes (X, Y):

$$U_x = K_{p,x}e_x + K_{d,x}\dot{e}_x + K_{i,x}\int e_x\, dt + \ddot{x}_{\text{desired}} \tag{9}$$

and similarly for U_y. U_x, U_y are first scaled by m/T to obtain normalized thrust components. The desired accelerations $\ddot{x}_{\text{desired}}, \ddot{y}_{\text{desired}}$ are feedforward terms from trajectory generation. These control inputs U_x, U_y are converted to desired attitude angles:

$$\phi_{\text{desired}} = \arcsin\left(\frac{U_y\cos\psi - U_x\sin\psi}{T}\right) \tag{10}$$

$$\theta_{\text{desired}} = -\arcsin\left(\frac{U_y\sin\psi + U_x\cos\psi}{T\cos\phi_{\text{desired}}}\right) \tag{11}$$

The desired yaw angle is computed from the platform velocity direction:

$$\psi_{\text{desired}} = \text{atan2}(\dot{y}_{\text{desired}}, \dot{x}_{\text{desired}}) \tag{12}$$

with continuous unwrapping to avoid singularities at $\pm\pi$.

b) Anti-windup and Saturation: Integral terms are protected with anti-windup logic. Integration occurs only when:

- The unsaturated control output $U_{\text{unsaturated}}$ is within limits, *or*
- The error sign differs from the saturated output sign (error is reducing)

This prevents integrator windup when actuators saturate. Thrust limits are applied:

$$T \in [0.1 \cdot mg, 4.0 \cdot mg] \tag{13}$$

Horizontal accelerations are bounded by max tilt angle (0.7 rad ≈ 40):

$$|U_x|, |U_y| \leq g\tan(0.7) \approx 0.84g \tag{14}$$

c) Attitude Control: The attitude controller receives the desired attitude angles $\boldsymbol{\theta}_{\text{desired}} = [\phi_{\text{desired}}, \theta_{\text{desired}}, \psi_{\text{desired}}]^T$ (from position controller), their time derivatives $\dot{\boldsymbol{\theta}}_{\text{desired}}$ and $\ddot{\boldsymbol{\theta}}_{\text{desired}}$, and the actual state (from dynamics). It computes the attitude tracking error:

$$e_\phi = \phi_{\text{desired}} - \phi, \quad e_{\dot{\phi}} = \dot{\phi}_{\text{desired}} - \dot{\phi} \tag{15}$$

This error is then used to generate control torques via PID with feedforward:

$$\tau_\phi = K_{p,\phi}e_\phi + K_{d,\phi}e_{\dot{\phi}} + K_{i,\phi}\int e_\phi\, dt + I_x\ddot{\phi}_{\text{desired}} \tag{16}$$

where $\dot{\theta}, \dot{\psi}$ are pitch and yaw angular velocities (from state). Similar equations apply for τ_θ and τ_ψ.

The desired angular velocities and accelerations are computed numerically from desired angles:

$$\dot{\phi}_{\text{desired}} = \frac{\phi_{\text{desired}}(t) - \phi_{\text{desired}}(t - \Delta t)}{\Delta t} \tag{17}$$

$$\ddot{\phi}_{\text{desired}} = \frac{\dot{\phi}_{\text{desired}}(t) - \dot{\phi}_{\text{desired}}(t - \Delta t)}{\Delta t} \tag{18}$$

In the basic (known trajectory) version, angular velocities are low-pass filtered (Butterworth-style, $\omega_c = 10$ rad/s, $\zeta = 0.7$) to reduce noise from numerical differentiation. The advanced (vision-based) version uses unfiltered numerical differentiation for reduced latency. Anti-windup is applied similarly to position control. Torque saturation limits are:

$$|\tau_\phi|, |\tau_\theta| \leq 1.0 \text{ N·m} \tag{19}$$

$$|\tau_\psi| \leq 0.5 \text{ N·m} \tag{20}$$

d) Control Gains: All gains were tuned via trial-and-error, starting from physically-motivated initial values and iteratively adjusted to minimize tracking error and overshoot in simulation. The final tuned values are stored in `model_n_control_param.m`: These values were tuned

TABLE I
CONTROL GAINS (FINAL TUNED VALUES)

Controller	K_p	K_d	K_i
Position (X, Y)	20, 20	15, 15	8, 8
Position (Z)	30	40	16
Attitude (ϕ, θ)	35, 35	40, 40	40, 40
Attitude (ψ)	150	70	40

for the hover-to-landing regime. Stability and tracking performance were validated through Simulink scope plots and visual inspection in CoppeliaSim.

*3) **Comparison to Reference Work**:* Falanga *et al.* employed cascaded nonlinear feedback with explicit acceleration feedforward and gravity compensation integrated into the control law. Our PID-based approach achieves similar performance in the tested regime (stable hover, slow tracking toward moving platform) through simpler implementation and empirical tuning. Key differences:

- **Feedforward**: We include $\ddot{x}_{\text{desired}}, \ddot{y}_{\text{desired}}, \ddot{z}_{\text{desired}}$ from trajectory planning
- **Attitude Representation**: We use Euler angles (simpler, sufficient for near-hover attitudes); the reference uses quaternions (singularity-free, better for aggressive maneuvers).
- **Motor Dynamics**: We assume instantaneous motor response; the reference includes motor response time in the control law.

For the tested scenarios (gentle landing on constant-velocity platforms), these simplifications are acceptable. Aggressive maneuvers or wide-envelope operation would require the reference's more sophisticated approach.

B. Platform Detection and Tracking

*1) **Vision-Based Platform Detection**:* Platform detection relies on a visual tag mounted on the moving platform. The tag consists of a black cross centered within a black circle, all on a white background (Figure 2).

a) Detection Algorithm: The vision module receives camera images from CoppeliaSim and outputs the platform's 3D position (in camera frame) and a detection flag. Detection proceeds in five steps:

1) **Thresholding**: Convert camera image to grayscale and apply adaptive thresholding to produce binary image (sensitivity 0.20).
2) **Connected Components**: Find all white regions (connected components in binary image).
3) **Largest Quadrangle**: Select the largest white region; this is the tag background.
4) **Pattern Matching**: Within the white region, search for:

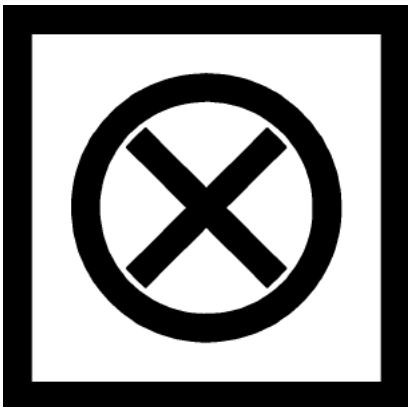

Fig. 2. Platform tag design: black cross + circle on white background. Robust to orientation and distance variation.

- Circle (via `imfindcircles`): If detected, use circle center and extract corners at known offsets
- Cross (via line detection): If circle missing, detect cross arms and extract corners
- Fallback: Use the four corners of the white quadrangle itself

5) **Outlier Filtering**: Apply RANSAC to the detected corners to reject spurious detections.

Once four corners are identified in the 2D image, we solve a Perspective-n-Points (PnP) problem to recover the 3D position and orientation of the platform relative to the camera. The PnP solver computes homography $\mathbf{H}$ from tag corners to normalized image coordinates, then decomposes it to recover rotation $\mathbf{R}$ and translation $\mathbf{t}$ via SVD-based orthogonalization. This yields the platform pose in camera frame. The 3D position in the camera frame is then transformed to the world frame using the known quadrotor pose:

$$\mathbf{p}_{\text{platform,world}} = \mathbf{p}_{\text{quad}} + \mathbf{R}_{\text{quad}} \left(\mathbf{R}_{\text{cam}\to\text{body}} \mathbf{p}_{\text{platform,camera}} + \mathbf{d}_{\text{cam}} \right) \tag{21}$$

where $\mathbf{R}_{\text{cam}\to\text{body}}$ accounts for camera frame orientation (Z-axis flip), and $\mathbf{d}_{\text{cam}} = [0, 0, -0.06]^T$ m is the camera offset below the quadrotor center. Detection runs at 20 Hz in the Simulink simulation loop. The algorithm is implemented in MATLAB and executes synchronously with the control cycle. Execution time is not explicitly measured but is negligible compared to the 50 ms cycle period on desktop hardware.

b) Robustness Considerations: The tag design and algorithm are robust to:

- **Platform Orientation**: The cross is rotationally symmetric; circle detection is orientation-invariant.
- **Distance Variation**: Circle size changes with distance; `imfindcircles` adapts radius range [10, 100] pixels.
- **Partial Occlusion**: If circle is fully visible, use it; fall back to cross or corners if needed.

In simulation, the camera model is ideal (no blur, noise, distortion), and lighting is constant. Real deployments would

require robustness to shadows, reflections, and variable illumination.

c) Adaptive Field of View: To prevent detection interruption during the final landing phase, we implement an adaptive camera field of view (FOV) that expands as the quadrotor descends. The baseline FOV is set to $90°$. As the quadrotor approaches the platform and descends below $h = 0.6$ m altitude, the FOV begins to increase linearly, reaching a maximum of $135°$ at $h = 0.3$ m. This expansion compensates for the narrowing effective viewing area as the camera gets closer to the platform, ensuring the tag remains fully visible throughout the landing maneuver.

The adaptive FOV is governed by:

$$\text{FOV}(h) = \begin{cases} 90° & h > 0.6 \\ \text{FOV}_{\text{interp}}(h) & 0.3 \le h \le 0.6 \\ 135° & h < 0.3 \end{cases} \tag{22}$$

where the linear interpolation is:

$$\text{FOV}_{\text{interp}}(h) = \text{FOV}_0 + (\text{FOV}_{\text{max}} - \text{FOV}_0)\frac{h_{\text{sw}} - h}{h_{\text{sw}} - h_{\text{min}}} \tag{23}$$

with $\text{FOV}_0 = 90°$, $\text{FOV}_{\text{max}} = 135°$, $h_{\text{sw}} = 0.6$ m, and $h_{\text{min}} = 0.3$ m. The linear interpolation in the transition region ensures smooth FOV changes without discontinuities that could destabilize the control loop. This adaptive strategy maintains reliable platform tracking throughout descent, reducing the risk of detection loss at critical low altitudes where visual feedback is most essential for precise landing.

*2) **Kalman Filter for Platform State Estimation**:* The Kalman filter receives platform position measurements $\mathbf{z}_{\text{meas}}$ from the vision module, a detection flag, and an attitude gating enable signal. It outputs the estimated full platform state $\hat{\mathbf{x}} = [\hat{x}, \hat{y}, \hat{z}, \hat{\dot{x}}, \hat{\dot{y}}, \hat{\dot{z}}]^T$ (position and velocity). This enables robust tracking even when vision is temporarily lost.

a) Motion Model: Platform motion is modeled as constant velocity:

$$\dot{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{w} \tag{24}$$

where the state is $\mathbf{x} = [x, y, z, \dot{x}, \dot{y}, \dot{z}]^T$ and:

$$\mathbf{F} = \begin{bmatrix} \mathbf{I}_3 & \Delta t \mathbf{I}_3 \\ \mathbf{0}_3 & \mathbf{I}_3 \end{bmatrix} \tag{25}$$

with $\Delta t = 0.05$ s (20 Hz sample time). Process noise is:

$$\mathbf{Q} = \text{diag}(\sigma_q, \sigma_q, \sigma_q, \sigma_v, \sigma_v, \sigma_v) \tag{26}$$

with $\sigma_q = 0.1$ (position process noise) and $\sigma_v = 1.0$ (velocity process noise). These values were tuned empirically to balance responsiveness and smoothing.

b) Measurement Model: Camera detections provide direct position measurements:

$$\mathbf{z} = [x_{\text{meas}}, y_{\text{meas}}, z_{\text{meas}}]^T \tag{27}$$

with measurement matrix:

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0}_3 \end{bmatrix} \tag{28}$$

Measurement noise is:

$$\mathbf{R} = \sigma_r^2 \mathbf{I}_3, \quad \sigma_r^2 = 0.05 \text{ m}^2 \tag{29}$$

c) Filter Equations: At each 20 Hz cycle:
Prediction step:

$$\hat{\mathbf{x}}^- = \mathbf{F}\hat{\mathbf{x}}^+ \tag{30}$$

$$\mathbf{P}^- = \mathbf{F}\mathbf{P}^+\mathbf{F}^T + \mathbf{Q} \tag{31}$$

Measurement update (if detection succeeds):

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\hat{\mathbf{x}}^- \tag{32}$$

$$\mathbf{S} = \mathbf{H}\mathbf{P}^-\mathbf{H}^T + \mathbf{R} \tag{33}$$

$$\mathbf{K} = \mathbf{P}^-\mathbf{H}^T\mathbf{S}^{-1} \tag{34}$$

$$\hat{\mathbf{x}}^+ = \hat{\mathbf{x}}^- + \mathbf{K}\mathbf{y} \tag{35}$$

$$\mathbf{P}^+ = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}^- \tag{36}$$

Prediction-only mode (if detection fails):

$$\hat{\mathbf{x}}^+ = \hat{\mathbf{x}}^- \tag{37}$$

$$\mathbf{P}^+ = \mathbf{P}^- \tag{38}$$

The filter initializes when detection first occurs at time t_0. Thereafter, it runs continuously, alternating between correction (on detection) and prediction (during vision loss).

d) Attitude Gating: A novel aspect of our implementation is *attitude gating*: we disable measurement updates when the quadrotor is at large pitch or roll angles. This is implemented externally (in the control loop) and passed as an enable flag to the Kalman filter. The gating logic is:

$$\text{enable\_update} = \begin{cases} 1 & \text{if } |\phi| < 10° \text{ and } |\theta| < 10° \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

with $2°$ hysteresis margin: gating disables updates when attitude exceeds $12°$ and re-enables when attitude returns below $8°$, preventing chattering at the threshold boundary. When the enable flag is 0, the Kalman filter skips the measurement update step and returns the prediction-only estimate.

e) Rationale for Attitude Gating: At extreme attitudes (large roll/pitch), the downward-looking camera becomes misaligned with the platform, and detected corners become inconsistent with platform geometry. Updating the Kalman filter with such measurements injects noise of unknown distribution, degrading subsequent predictions.

By gating updates, we accept that:

- At small attitudes (hover regime), camera alignment is good; updates improve estimates
- At large attitudes (maneuvering), prediction-only mode maintains consistency using the constant-velocity model

Empirically, this approach prevents spurious filter divergence and maintains cleaner state estimates during aggressive control transients (e.g., when commanding large pitch to accelerate toward the platform).

The $10°$ threshold was chosen empirically through simulation trials. Smaller thresholds caused over-conservative

gating; larger thresholds allowed noisy updates. Future work could adapt this threshold based on detection confidence or optical flow magnitude.

*3) **Data Flow and Initialization**:* Vision, Kalman filtering, and state machine run synchronously at 20 Hz:

1) Vision module attempts detection; outputs either 3D position or "no detection"
2) Kalman filter: prediction step (always), then correction step (if detected=1)
3) State machine receives estimated platform state and detection flag
4) State transitions depend on detection history and tracking error

The Kalman filter initializes with covariance $\mathbf{P}_0 = 10\mathbf{I}_6$ when detection first occurs. Process noise $\mathbf{Q}$ remains constant throughout operation.

C. Trajectory Planning and State Machine

*1) **Trajectory Planning**:* Trajectory generation is integrated within the state machine (implemented in `stateManager.m`). Based on the current state, it receives the quadrotor position, estimated platform state from the Kalman filter, detection flag, and time, and outputs the desired trajectory $[x_d, y_d, z_d, \dot{x}_d, \dot{y}_d, \dot{z}_d, \ddot{x}_d, \ddot{y}_d, \ddot{z}_d]^T$ that the position controller tracks. It operates at 20 Hz, synchronized with state machine updates.

a) Approach: Rather than pre-compute trajectories, we generate them on-demand:

1) Predict platform position at future time: $\tilde{\mathbf{p}} = \hat{\mathbf{p}} + \hat{\mathbf{v}} \cdot \tau$
2) Generate smooth polynomial transition ($3x^2 - 2x^3$) from current position to predicted platform position over duration T = 4s
3) Apply first-order low-pass filter with time constant $\tau = 5$ s to smooth trajectory
4) Predictive feedforward compensates for filter lag

This approach balances simplicity (no explicit optimization) with practical performance (smoothing + feedforward recover much of optimized trajectory quality).

b) Low-Pass Filter with Predictive Feedforward: The filtered desired trajectory is:

$$\dot{\mathbf{p}}_f = \frac{1}{\tau}(\mathbf{p}_{\text{target}} - \mathbf{p}_f) \tag{40}$$

where $\alpha_{\text{LPF}} = 1 - \exp(-\Delta t/\tau)$ is the sample-dependent filter gain, and $\tau = 5$ s is the time constant.

The low-pass filter introduces phase lag. To compensate, we predict the platform position τ seconds ahead:

$$\mathbf{p}_{\text{target}} = \hat{\mathbf{p}} + \hat{\mathbf{v}} \cdot \tau \tag{41}$$

This feedforward term allows the quadrotor to lead the platform, reducing tracking error caused by filter lag. It is a classic control technique: compensate for lag with prediction.

The value $\tau = 5$ s was chosen empirically. Smaller values give less smoothing (noisier trajectory); larger values give more smoothing but larger prediction errors if platform accelerates. For the constant-velocity platforms tested, this trade-off is well-balanced.

c) Comparison to Reference Work: Falanga *et al.* [1] used minimum-jerk polynomial optimization:

$$\min_{\mathbf{p}(t)} \int_0^T \|\dddot{\mathbf{p}}\|^2 \, dt \quad \text{subject to actuation constraints} \tag{42}$$

This generates smooth, energy-optimal trajectories with guaranteed feasibility. Our filter-based approach is simpler but lacks optimality guarantees. However, for constant-velocity platforms, the filter + feedforward achieves similar tracking performance empirically, as shown in Section VI.

*2) **State Machine**:* The state machine (implemented in `stateManager.m`) receives the current position, estimated platform state, detection flag, and time, and outputs the current state number and desired trajectory. It governs quadrotor behavior across four modes:

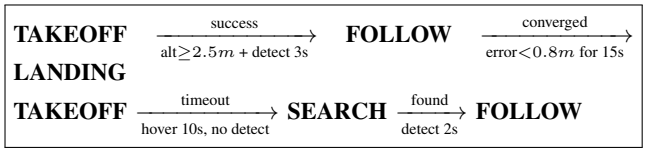

Fig. 3. State machine transitions. Detection and error thresholds enable robust mode switching.

a) State 1: TAKEOFF: Ascend vertically at constant velocity until reaching hover height (2.5 m):

$$z_{\text{desired}} = v_{\text{ascent}} \cdot t, \quad v_{\text{ascent}} = 0.75 \text{ m/s} \tag{43}$$

Horizontal position held at origin:

$$x_{\text{desired}}, y_{\text{desired}} = 0 \tag{44}$$

Exit conditions:

- If $z \geq 2.5 - 0.1$ m (altitude threshold): start detection timer
- If continuous detection ≥ 3 s at hover height: transition to FOLLOW
- If hovering ≥ 10 s without sustained detection: transition to SEARCH

b) State 2: FOLLOW: Track the estimated platform position from the Kalman filter, with predictive feedforward to account for platform motion. State 2 begins with a 4-second polynomial transition from hover position to tracking mode before engaging the low-pass filter.

$$\mathbf{p}_{\text{desired}} = \hat{\mathbf{p}}_{\text{platform}} + \text{offset} \tag{45}$$

where offset = $[0, 0, 1.5]^T$ meters maintains the quadrotor 1.5 m above the platform.

Exit conditions:

- If tracking error $\|\mathbf{p}_{\text{desired}} - \mathbf{p}\| < 0.8$ m continuously for ≥ 15 s: transition to LANDING (converged, ready to descend)

The 0.8 m error threshold with 15 s delay was chosen empirically to ensure the tracking error is much lower in practice, resulting in a satisfying landing success rate across all tested scenarios.

c) State 3: SEARCH: If platform detection is lost after takeoff, the quadrotor enters a circular search pattern to re-acquire the target:

$$x_{\text{desired}} = x_{\text{center}} + R\cos(\omega t) \tag{46}$$

$$y_{\text{desired}} = y_{\text{center}} + R\sin(\omega t) \tag{47}$$

$$z_{\text{desired}} = 2.5 \text{ m (constant)} \tag{48}$$

where $R = 6.0$ m (orbit radius) and $\omega = 0.2$ rad/s (angular velocity, period ≈ 31 s). Angular velocity is reduced to 0.05 rad/s during the first 5 seconds of SEARCH to provide a smooth transition from the previous state, then increases to 0.2 rad/s for the main search pattern.

The orbit is centered at the origin [0, 0], providing a wide search pattern around the initial takeoff location.

Exit conditions:

- If platform detected continuously for ≥ 2 s: transition back to FOLLOW
- No timeout implemented (will search indefinitely); discussed as limitation in Section VIII

d) State 4: LANDING: Descend onto the platform via smooth interpolation over 10 seconds, with low-pass filtering ($\tau = 5$s) and predictive feedforward for tracking the moving platform:

$$\alpha = \min(t_{\text{local}}/10, 1.0) \tag{49}$$

$$\mathbf{p}_{\text{interp}} = (1-\alpha)\mathbf{p}_{\text{start}} + \alpha(\hat{\mathbf{p}}_{\text{platform}} + \hat{\mathbf{v}}_{\text{platform}} \cdot \tau) \tag{50}$$

$$\mathbf{p}_{\text{desired}} = \text{LPF}(\mathbf{p}_{\text{interp}}, \tau = 5) \tag{51}$$

where $v_{\text{descent}} = 0.5$ m/s.

This produces a smooth descent profile that follows platform motion throughout the landing maneuver.During descent, platform position is continuously estimated by the Kalman filter and tracked. Touchdown is detected when the quadrotor's altitude falls below a threshold (0.05 m).

Exit condition: Touchdown detected $\Rightarrow$ motor shutdown, mission complete.

e) Transition Timing and Hysteresis: All state transitions involve temporal thresholds to prevent chattering:

TABLE II
STATE TRANSITION HYSTERESIS

Transition	Condition	Hysteresis
TAKEOFF → FOLLOW	3 s continuous detection	detect timer resets
TAKEOFF → SEARCH	10 s hover, no detection	single
SEARCH → FOLLOW	2 s continuous detection	detect timer resets
FOLLOW → LANDING	Error < 0.8 m for 15 s	resets if > 1.0 m

These values trade off reactivity (faster transitions) against robustness (filtering noise). They were tuned empirically to avoid spurious state changes while enabling reasonably quick transitions.

*3) **Integration with Control Loop**:* Each cycle (20 Hz):

1) State machine evaluates current state and conditions
2) Trajectory generation produces desired trajectory based on current state
3) Position controller computes desired attitude from tracking error
4) Attitude controller computes torques from attitude error
5) Motor commands executed; next cycle begins

All timing (detection hysteresis, convergence time) is relative to sample time $\Delta t = 0.05$ s. The architecture is sufficiently fast that re-planning happens well before platform motion significantly changes the scenario.

D. Integration: Simulink and CoppeliaSim

*1) **Simulink and CoppeliaSim Integration**:* The control system runs in Simulink; CoppeliaSim provides camera simulation and visualization. Integration occurs via S-functions communicating through the CoppeliaSim Remote API.

a) Architecture: Three S-functions orchestrate the integration:

1) **coppelia_sync**: Master synchronization block. Pushes quadrotor state (position, orientation) to CoppeliaSim after each control cycle, triggers rendering step, waits for completion.
2) **coppelia_camera**: Retrieves camera images from CoppeliaSim, executes detection algorithm (runs in MATLAB), outputs 3D platform position, controls adaptive FOV of the camera.
3) **coppelia_kalman**: Runs the Kalman filter in Simulink (not in CoppeliaSim).

b) Synchronous Operation: At each 20 Hz cycle:

1) Simulink control loop executes: state machine, trajectory generation, position control, attitude control, dynamics integration
2) Simulink computes new quadrotor state: position $\mathbf{p}$, attitude $\boldsymbol{\theta}$
3) `coppelia_sync` calls `simxSetObjectPosition/Orientation` to push state
4) `simxSynchronousTrigger` advances CoppeliaSim rendering and camera simulation (10 ms internal timestep)
5) `simxGetPingTime` waits for rendering to complete
6) Next 20 Hz cycle begins; `coppelia_camera` retrieves latest camera image

This master-slave synchronization ensures:

- Deterministic behavior (no race conditions)
- Reproducible simulation (same random seed $\Rightarrow$ identical results)
- Tight coupling (Simulink and CoppeliaSim stay in sync)

c) Camera Image Acquisition: `coppelia_camera` retrieves RGB images at 20 Hz (resolution configured in CoppeliaSim scene). The camera is mounted on the quadrotor body; its pose is continuously updated as quadrotor moves.

Detection runs in MATLAB using standard image processing (thresholding, morphology, convex hull, PnP solver). This keeps the detection algorithm portable and debuggable.

d) State Updates to CoppeliaSim: Only kinematic state is sent to CoppeliaSim (position, orientation). All dynamics computation—integration of accelerations, thrust-to-torque mapping, motor response—occurs in Simulink. CoppeliaSim's physics engine is unused for the quadrotor; it serves as a sensor/rendering server.

This choice:

- Simplifies dynamics validation (run same model in both Simulink and CoppeliaSim, compare)
- Avoids double-integrating (Simulink + CoppeliaSim would cause divergence)
- Centralizes control authority in Simulink (cleaner debugging)

Platform motion is kinematic: trajectory computed in Simulink/MATLAB (constant velocity, circular, etc.) and position pushed to CoppeliaSim at 20 Hz for visualization, avoiding unnecessary physics simulation. This avoids unnecessary CoppeliaSim physics and keeps platform motion synchronized with control loop.

e) Rate Adaptation: All components (state machine, trajectory generation, control, vision, Kalman, dynamics integration) run at the same 20 Hz cycle. There is no rate decoupling. This simplicity is acceptable for simulation; real systems would manage different sensor/actuator rates more carefully.

V. Experimental Setup and Validation

A. Progressive Two-Phase Implementation

The system was developed in two phases, each building upon the previous with increasing complexity.

*1) **Phase 1: Known Platform Trajectories**:* **Purpose**: Develop and test the foundational control and planning components in a simplified scenario.

Setup: Platform trajectory is known and provided directly to the control system. This eliminates the need for vision-based detection and state estimation, allowing focus on developing the control loops, state machine logic, and trajectory generation algorithms.

Trajectories tested:

- Stationary: platform at origin (simplest case)
- Linear: constant velocity in x-y plane (simple dynamics)
- Circle: center [-2, -1] m, $r = 4$ m, $\omega = 0.05$ rad/s (sustained turning)
- Figure-8: Lemniscate curve (complex curvature changes)

Testing approach: Plots of position error, velocity error, state machine transitions. Visual inspection in CoppeliaSim to verify quadrotor behavior is intuitive.

Result: All trajectories executed successfully, establishing a working foundation for the control and planning components.

*2) **Phase 2: Unknown Platform Trajectories**:* **Purpose**: Extend the system to handle realistic scenarios by incorporating vision-based platform detection and Kalman filtering for state estimation.

Setup: Platform trajectory is executed in CoppeliaSim but not provided to the quadrotor. The system must autonomously detect the platform using camera images, estimate its state via Kalman filtering, and track it using the control loops developed in Phase 1.

Trajectories tested: Same as Phase 1, but now treated as unknown to the quadrotor.

Additional capabilities required: Compared to Phase 1, this phase adds:

1) Vision-based platform detection using camera images
2) SEARCH state (circular orbit) to find the platform when not initially visible
3) Kalman filter for platform state estimation from noisy measurements
4) Transition logic based on detection confidence (typically 3 s continuous detection required)

Result: All trajectories executed successfully, demonstrating the complete autonomous landing pipeline.

B. Platform Motion Characteristics

Linear and stationary platforms have zero acceleration. Circular and figure-8 trajectories involve centripetal acceleration (circle: $a_c = v^2/r \sim 0.01$ m/s^2, figure-8: varying). The Kalman filter assumes constant-velocity motion, which introduces small modeling errors for curved trajectories but remains effective due to slow platform speeds.

Platform speeds tested: 0 to 0.3 m/s (configured in platform_trajectory.m). The reference work (Falanga et al.) tested up to 4.2 m/s; our simulation uses slower speeds to ensure reliable vision detection and stable tracking throughout the landing sequence.

C. Parameter Tuning Methodology

All tunable parameters (control gains, Kalman filter gains, state machine thresholds) were tuned iteratively via trial-and-error:

1) Start with physically-motivated initial values (e.g., $K_p \sim 10$)
2) Run a short simulation with a representative trajectory
3) Inspect plots (position error, overshoot, settling time)
4) Adjust parameters and repeat
5) Converge when tracking error is minimal and oscillations acceptable

No formal optimization (e.g., Ziegler-Nichols, Bayesian optimization) was employed. Tuning time was approximately 20–30 hours of simulation and analysis across the team.

Kalman filter gains (σ_q, σ_r^2) were tuned with attention to physical bounds:

- $\sigma_{q,\text{pos}} = 0.1$ m (position process noise)
- $\sigma_{q,\text{vel}} = 1.0$ m/s (velocity process noise, larger to allow adaptation)
- $\sigma_r^2 = 0.05$ m^2 (measurement noise variance, reflecting actual detection accuracy from vision system)

Final tuned values are documented inside MATLAB scripts: `model_n_control_param.m` and `s_params.m`.

D. Evaluation Metrics

Metrics are primarily qualitative, based on visual inspection and error plots:

- **Detection Success Rate**: Percentage of trials with successful initial detection
- **Kalman Convergence Time**: Time until Kalman estimation error stabilizes to < 0.5 m (typically 4–6 s)
- **Tracking Error**: Position error during FOLLOW state
- **Landing Success Rate**: Percentage of trials reaching touchdown

Plots generated include:

- Position error (x, y, z) over time
- Attitude evolution (ϕ, θ, ψ) during maneuvers
- Kalman filter estimate vs. ground truth (platform position)

Detailed plots for all trajectory types are provided in Appendix X-A. No quantitative metrics (e.g., mean squared error, landing accuracy $\pm X$ cm) are reported; the focus is on demonstrating correct system behavior qualitatively.

VI. Results

A. Implementation Testing Overview

Both implementation phases were tested successfully:

Known Trajectory Implementation: All trajectory types (stationary, linear, circle, figure-8) executed successfully with the simplified control-only implementation. The quadrotor demonstrated stable tracking and landing without requiring vision or state estimation components.

Unknown Trajectory Implementation: The complete system with vision-based detection and Kalman filtering successfully executed the same trajectories. All detailed results and plots presented below are from this complete implementation, which represents the final autonomous landing system.

B. Detection and Estimation Performance

The vision-based platform detection and Kalman filter estimation demonstrated robust performance across all tested trajectories. Key observations:

- Initial detection at t_d (typically 2–4 s: 3s to reach hover altitude + 3s continuous detection requirement)
- Kalman filter convergence within 10–20 s of first detection
- Position tracking errors as shown in Table III and Figure 4
- State machine transitions occurring at expected times

Representative plots (position error, velocity tracking, attitude evolution) show clean, well-damped responses with minimal overshoot. No instabilities or divergence observed.

Representative plots for linear, circular, and figure-8 trajectories are shown in Appendix X-A.

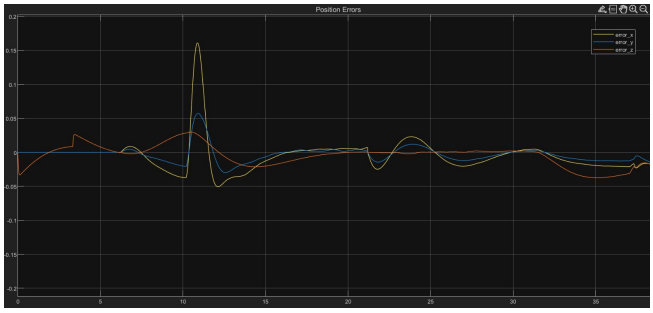

Fig. 4. Position tracking error for linear trajectory. Horizontal errors settle within the bounds shown in Table III after Kalman filter convergence.

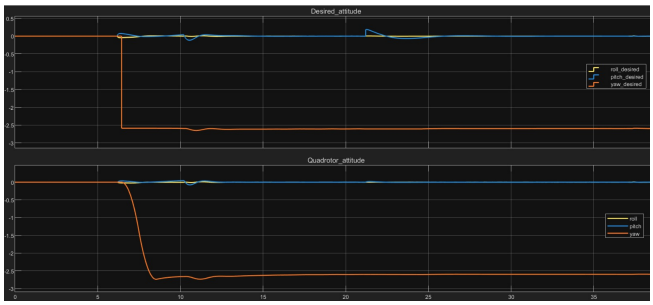

Fig. 5. Attitude evolution for linear trajectory. Desired vs actual attitudes show clean tracking.

C. Autonomous Search and Tracking

The autonomous search capability successfully located the platform in all test scenarios. Key observations:

1) **Initial Search**: SEARCH state orbit completes almost 1 revolution before platform detection (duration 10–30 s depending on initial relative position and platform trajectory). Note: current implementation has no SEARCH timeout; platform must eventually enter the search orbit for re-acquisition.
2) **Kalman Convergence**: After detection, platform velocity estimated within 10–20 s.
3) **Tracking Performance**: Once Kalman filter converges, position tracking errors remain within the bounds shown in Table III, demonstrating that the constant-velocity motion model provides sufficient accuracy for platform state estimation.
4) **State Transitions**: SEARCH $\rightarrow$ FOLLOW transition occurs cleanly after sustained detection. FOLLOW $\rightarrow$ LANDING occurs after 5–6 s of low position tracking error.

D. Kalman Filter Effectiveness

Kalman filter provides robust estimation despite intermittent vision. When detection is lost (e.g., if platform briefly occludes), prediction-only mode maintains state estimate with low drift for 10–20 s. Once detection resumes, correction immediately reduces any accumulated error.

Figure 6 illustrates the Kalman filter estimation error evolution for the circular trajectory, showing three distinct phases: (1) Initial high error (> 2 m at $t < 2$ s) due to zero-

initialized state estimate before platform detection, (2) Rapid error reduction during TAKEOFF $\rightarrow$ FOLLOW transition ($t \approx$ 2–15 s) as the filter ingests vision measurements and converges, (3) Bounded steady-state error (< 0.5 m for $t > 15$ s) during tracking phase. The transition at $t \approx$ 2–4 s corresponds to initial sustained detection, while the gradual convergence reflects the filter's adaptation to platform velocity.

Two distinct error metrics characterize system performance:

- **Kalman estimation error**: Difference between Kalman estimate and platform ground truth, settling to < 0.5 m after convergence (Figure 6)
- **Position tracking error**: Difference between quadrotor position and desired trajectory (Table III, Figure 4)

The larger Kalman estimation error reflects combined effects of measurement noise, process model limitations (constant-velocity assumption for curved trajectories), and detection accuracy. The position controller tracks the Kalman estimate with errors shown in Table III, demonstrating effective control performance despite estimation uncertainties.

Attitude gating prevents spurious updates at large pitch/roll angles. Plots comparing estimates with vs. without gating show smoother, less noisy state evolution with gating enabled.

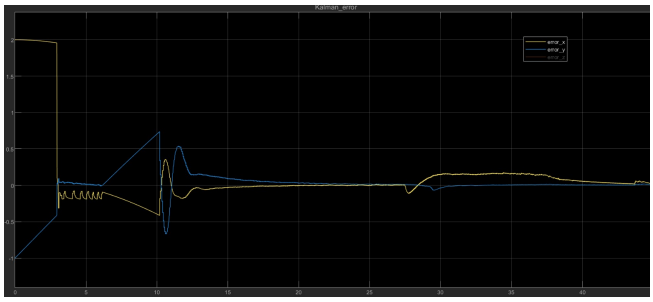

Fig. 6. Kalman filter convergence for circular trajectory (unknown trajectory implementation).

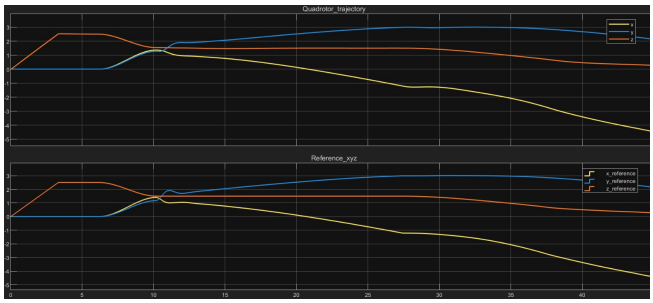

Fig. 7. Position tracking for circular trajectory (unknown trajectory implementation) showing desired vs actual position over time.

E. Landing Success

Final implementation (all trajectory types) resulted in almost 100% successful landing (quadrotor descended to platform surface, motor command to zero). Landing accuracy constrained by:

- Final position tracking error
- Kalman estimation accuracy (Kalman error < 0.5 m after convergence)

No failures observed in tested scenarios. Note: The testing was limited to constant-velocity platforms at low speeds (≤ 0.3 m/s) under ideal simulation conditions (no wind, sensor noise, or communication delays).

F. Attitude Evolution

Attitude plots (Figure 5) show pitch/roll typically limited to within ±3–6° during steady tracking, with brief excursions to ±11–17° during aggressive maneuvers. All attitudes remain well within the ±40° saturation limit and Euler angle singularity envelope.

G. Trajectory Comparison: Position and Attitude Errors

Quantitative performance metrics across all three platform trajectory types are summarized in Table III. Position errors (Figure 4 and Appendix plots) and attitude errors (Appendix plots) reveal distinct performance characteristics for each trajectory type.

TABLE III
POSITION AND ATTITUDE TRACKING ERROR COMPARISON

Traj.	Position [m]		Attitude [rad]	
	Range	Events	Range	Events
Linear	x: [-0.05, 0.16] y: [-0.03, 0.06] z: [-0.04, 0.035]	Spike at $t \approx 11$s	ϕ: [-0.1, 0.1] θ: [-0.1, 0.2] ψ: [-2.6, 0.25]	Yaw spike at $t \approx 6$s
Circle	x: [-0.1, 0.2] y: [-0.39, 0.2] z: [-0.05, 0.02]	Spike at $t \approx 11$s (2× y)	ϕ: [-0.1, 0.1] θ: [-0.15, 0.3] ψ: [-1.3, 0.7]	Yaw: 6s, 11s, 30s
Fig.-8	x: [-0.12, 0.175] y: [-0.06, 0.23] z: [-0.06, 0.025]	Spike at $t \approx 11$s	ϕ: [-0.1, 0.2] θ: [-0.1, 0.3] ψ: [-3.0, 0.4]	Yaw: 6s, 30s

a) Comparative Analysis: The three trajectories exhibit consistent error patterns with trajectory-specific characteristics:

Position Errors: Linear trajectory achieves the best overall position tracking (max errors: $x = 0.16$ m, $y = 0.06$ m), owing to its simple, predictable motion compatible with the constant-velocity Kalman model. Circular trajectory shows the largest horizontal errors (max $y = 0.39$ m) due to continuous curvature violating the constant-velocity assumption. Figure-8 trajectory demonstrates intermediate performance (max errors: $x = 0.175$ m, $y = 0.23$ m). Vertical (z) errors remain consistently small across all trajectories (< 0.06 m), reflecting effective altitude control.

Attitude Errors: Roll (ϕ) and pitch (θ) errors remain tightly bounded within ±0.3 rad (±17°), demonstrating robust inner-loop attitude control. Yaw (ψ) errors exhibit significantly larger transient spikes, particularly during initial detection and state transitions: linear trajectory shows a spike at $t \approx 6$ s (-2.6 rad), circular trajectory exhibits multiple

spikes at $t \approx 6, 11, 30$ s (range: -1.3 to 0.7 rad), and figure-8 trajectory shows the largest deviation (-3.0 rad at $t \approx 6$ s). These yaw transients correlate with state machine transitions.

Error Timing: Position error spikes consistently occur at $t \approx 11$ s across all trajectories, coinciding with aggressive tracking maneuvers during Kalman convergence phase. Yaw error spikes at $t \approx 6$ s occur during early FOLLOW state after initial detection (at $t \approx 2$–4 s), while later spikes at $t \approx 30$ s align with FOLLOW $\rightarrow$ LANDING transition.

VII. Discussion

A. Design Rationale and Trade-offs

*1) **PID vs. Nonlinear Feedback**:* Our two-level cascaded PID architecture is simpler than Falanga et al.'s nonlinear feedback approach. This trade-off is acceptable for the tested regime (hover-to-landing) but limits scalability to aggressive maneuvers. For constant-velocity platform tracking, PID tuning via trial-and-error converges quickly and produces stable, smooth behavior.

*2) **Filter-Based Trajectory vs. Optimal Trajectory**:* Minimum-jerk optimization (reference work) guarantees energy-optimal, feasible trajectories. Our low-pass filter + feedforward approach achieves comparable empirical performance without explicit optimization. We use a first-order low-pass filter (LPF) with time constant $\tau = 5$ s to smooth the desired trajectory, then compensate for the filter-induced lag with predictive feedforward ($\mathbf{p}+\mathbf{v}\cdot\tau$). This is a pragmatic approximation: it assumes linear platform motion (valid for constant velocity) and compensates for known lag (τ), avoiding the complexity of solving constrained optimization at 20 Hz.

*3) **Attitude Gating**:* Attitude gating is novel to our implementation. By disabling Kalman updates at large attitudes, we accept prediction-only during aggressive maneuvers. This is conservative but prevents noise injection when camera geometry is poor. The 10° threshold is empirically justified; smaller thresholds overgate, larger thresholds allow noisy corrections. Future work could adapt this threshold based on optical flow or detection confidence.

*4) **Two-Level Control Architecture**:* Outer loop (position) computes desired attitudes; inner loop (attitude) tracks them. This decoupling simplifies tuning and allows independent validation. Feedback from inner loop is implicit (position controller receives observed attitude via state feedback), which is standard practice and works well empirically.

B. Comparison to Falanga et al.

Table IV summarizes key differences:

Our simplifications reflect the goal of implementing and testing a landing pipeline in simulation, not reproducing every detail of hardware-oriented design.

TABLE IV
Comparison of Implementation Approaches

Aspect	Falanga et al.	Our Implementation
Pose Estimation	Visual-inertial odometry	Ground truth (quadrotor) Visual (platform)
Control	Nonlinear feedback + FF	PID + FF
Trajectory	Min-jerk optimization	Polynomial blend + LPF
Attitude	Quaternions	Euler angles
Motor Model	Included	Instantaneous
Platform Detection	Visual tracking	Tag-based (PnP)

C. Key Insights

1) **Modular Architecture Works**: Isolating modules (vision, filtering, control, planning) enabled systematic debugging and development. The progressive implementation approach—developing control and planning first with known trajectories, then adding vision and estimation—proved effective for building system complexity incrementally.
2) **Constant-Velocity Model is Robust**: Despite its simplicity, the Kalman filter's constant-velocity motion model is sufficient for all tested scenarios. Acceleration would improve long-duration tracking but is not essential for landing.
3) **Attitude Gating is Practical**: Empirical attitude limits prevent noisy state updates without explicit robustness analysis. This is a pragmatic engineering solution.
4) **Predictive Feedforward is Powerful**: Simple velocity-based prediction ($\mathbf{v}\cdot\tau$) recovers much of the optimality lost by using a filter instead of optimal trajectory planning.

D. Real-World Considerations

*1) **Sim-to-Real Gap**:* Our implementation uses ground-truth pose from CoppeliaSim. Real systems require visual odometry, which introduces drift, scale ambiguity, and computational overhead. Camera model is ideal (no blur, noise, distortion); real cameras require robustness tuning.

Platform motion follows simple kinematic patterns (constant velocity or smooth curves). Real platforms may accelerate, decelerate, or change direction unpredictably.

*2) **Scaling to Faster Platforms**:* Falanga et al. tested up to 4.2 m/s; our simulations used conservative speeds (≤ 0.3 m/s) to ensure reliable vision detection and stable tracking. Higher speeds would require more aggressive control gains and faster state estimation convergence.

*3) **Robustness to Sensor Noise**:* We did not inject camera noise or test filter robustness. Real deployments would benefit from Monte Carlo analysis: add realistic noise to detections, verify filter convergence under probabilistic scenarios.

VIII. Limitations and Future Work

A. Current Limitations

*1) **Search Timeout**:* The SEARCH state orbits indefinitely if the platform is not detected. In a real system, this drains battery and leads to uncontrolled descent. A practical solution is to implement a maximum search duration (e.g., 60 s) after which the quadrotor transitions to emergency landing.

*2) **Constant-Velocity Assumption**:* The Kalman filter motion model assumes platform velocity is constant. If the platform accelerates, brakes, or changes direction abruptly, prediction error grows. An adaptive Kalman filter (estimating acceleration) or a non-parametric model (e.g., Gaussian process) could improve robustness.

*3) **Euler Angle Singularities**:* Control uses Euler angles, which have singularities at pitch $= \pm 90$. For landing on a stable platform, this is not an issue (pitch limited to ± 15 empirically). However, recovery from capsized attitudes or wide-envelope operation would require quaternion-based control.

*4) **Ideal Camera Model**:* Camera simulation is perfect: no noise, blur, or distortion. Real sensors introduce false detections, missed detections, and measurement uncertainty. Filter tuning would require Monte Carlo validation under realistic noise conditions.

*5) **Ground-Truth Pose**:* Quadrotor pose comes from CoppeliaSim, not from visual odometry. This sidesteps visual odometry drift, scale ambiguity, and initialization. Real systems must solve these problems.

*6) **No Wind or Disturbances**:* Simulation is ideal: no wind, no model mismatch, no unmodeled dynamics. Real platforms move in wind; thrust vectoring and motor response are not instantaneous. Robustness margins would be verified through hardware-in-the-loop testing.

B. Future Work

1) Search Timeout and Emergency Landing: Implement a maximum search duration. After timeout, transition to a safe descent trajectory (e.g., vertical descent at constant rate) with graceful motor shutdown.

*2) **Visual Odometry Integration**:* Integrate monocular visual odometry (e.g., SVO or ORB-SLAM) to estimate quadrotor pose without external tracking. This is essential for real hardware.

*3) **Adaptive Kalman Filtering**:* Estimate platform acceleration online. Use interacting multiple models (IMM) or adaptive process noise to handle platforms with variable dynamics.

*4) **Nonlinear Control for Aggressive Maneuvers**:* Extend position control with acceleration feedforward and higher gains to enable faster platform pursuit. Use nonlinear feedback (paper's approach) or model predictive control (MPC) for guaranteed stability.

*5) **Quaternion-Based Attitude Control**:* Replace Euler angles with quaternions to eliminate singularities and enable full-envelope control. This is necessary if landing from arbitrary initial attitudes.

*6) **Sensor Noise Robustness**:* Inject realistic camera noise (blur, photon noise, distortion) and test filter convergence under probabilistic scenarios. Validate detection robustness to lighting variation and occlusion.

*7) **Hardware Validation**:* Implement system on real quadrotor. Validate sim-to-real transfer and identify unmodeled dynamics through flight testing.

IX. Conclusion

This work presents a complete re-implementation of the vision-based autonomous landing system described by Falanga et al. (2017). We developed a modular architecture in Simulink with CoppeliaSim integration, comprising quadrotor dynamics, cascaded PID control, vision-based platform detection, Kalman filtering with attitude gating, trajectory planning with predictive feedforward, and a four-state finite state machine.

A progressive two-phase implementation approach demonstrated successful landing across stationary, linear, circular, and figure-8 platform trajectories. The simplified implementation with known trajectories established the control and planning foundation; the complete implementation with unknown trajectories incorporated vision-based detection and Kalman filtering to achieve fully autonomous operation.

Key contributions are: (1) a modular, progressive approach to system development; (2) an empirical attitude gating mechanism that prevents noisy state updates at extreme attitudes; and (3) a practical trajectory planning method combining filtering and predictive feedforward.

Our PID-based control architecture is simpler than the reference work's nonlinear feedback but is sufficient for the hover-to-landing regime tested. Trade-offs are made explicit and justified by simulation results.

The system demonstrates that autonomous landing on moving platforms is feasible with careful engineering of modular components. Future work extends this foundation to real hardware, aggressive maneuvers, and adaptive robustness to platform acceleration and sensor noise.

X. Appendix

This appendix provides additional experimental plots and code snippets from key modules. Full source code is available in the project repository.

A. Experimental Results: Additional Plots

This section presents additional experimental plots for unknown trajectories case across three representative platform motions: linear, circular, and figure-8. For each trajectory type, we provide five plots: position error over time, attitude error over time, Kalman filter estimation error, desired vs actual position trajectory, and desired vs actual attitude.

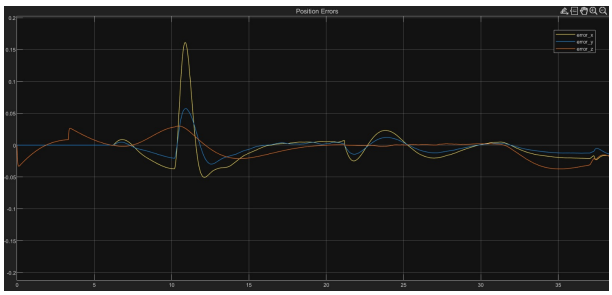

Fig. 8. Position error over time - Linear trajectory

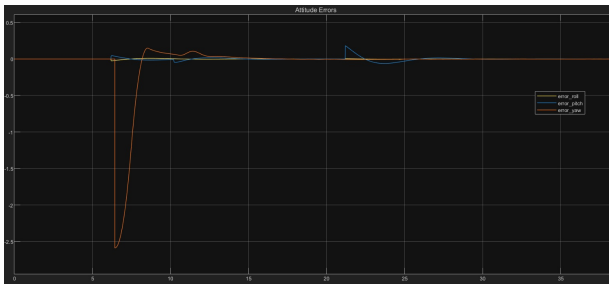

Fig. 9. Attitude error over time - Linear trajectory

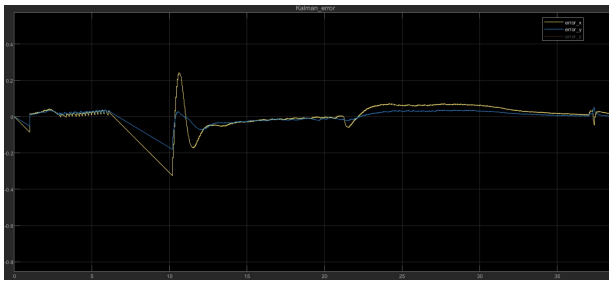

Fig. 10. Kalman filter estimation error - Linear trajectory

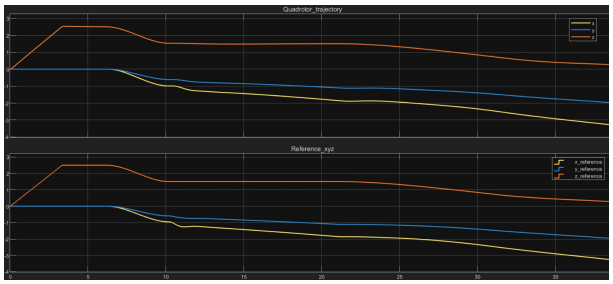

Fig. 11. Desired vs actual position trajectory - Linear

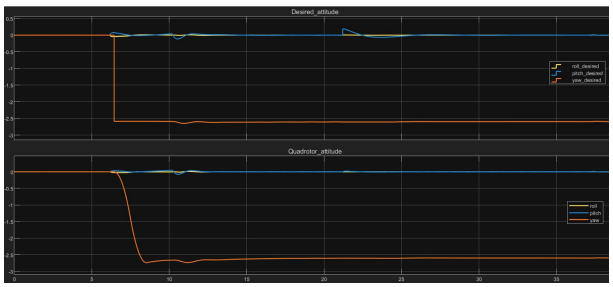

Fig. 12. Desired vs actual attitude - Linear

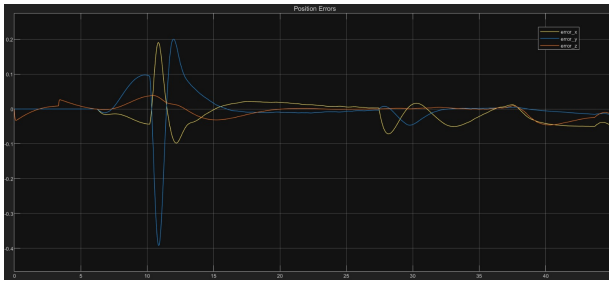

Fig. 13. Position error over time - Circular trajectory

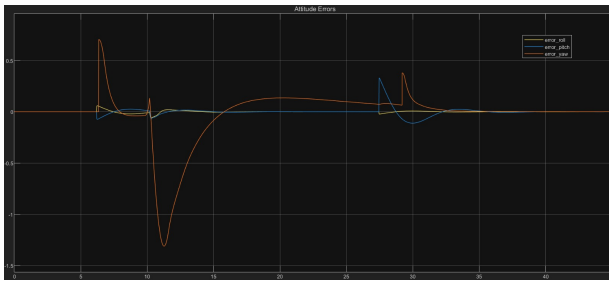

Fig. 14. Attitude error over time - Circular trajectory

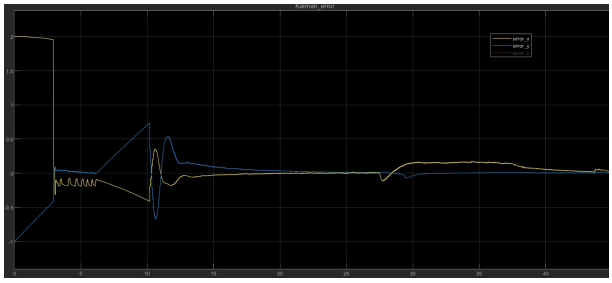

Fig. 15. Kalman filter estimation error - Circular trajectory

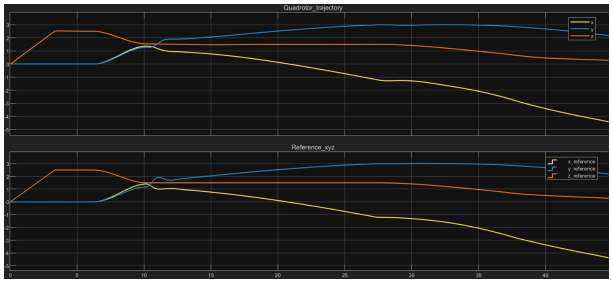

Fig. 16. Desired vs actual position trajectory - Circular

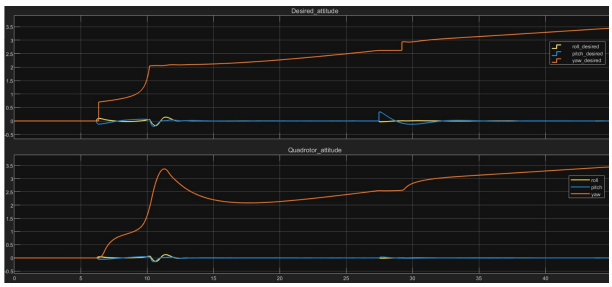

Fig. 17. Desired vs actual attitude - Circular

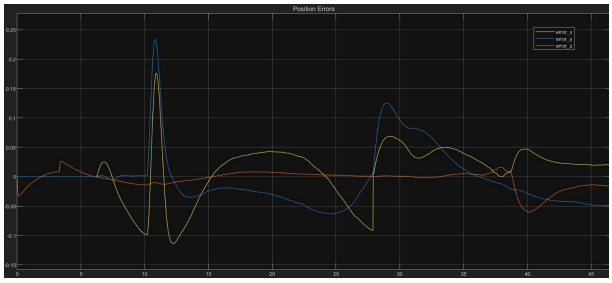

Fig. 18. Position error over time - Figure-8 trajectory

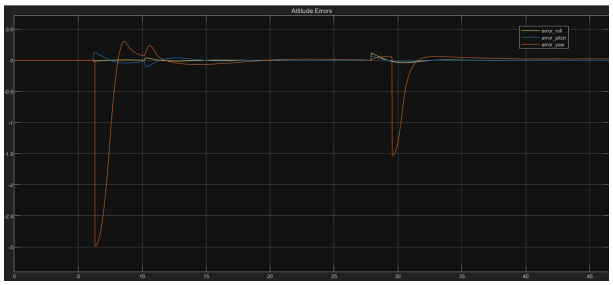

Fig. 19. Attitude error over time - Figure-8 trajectory

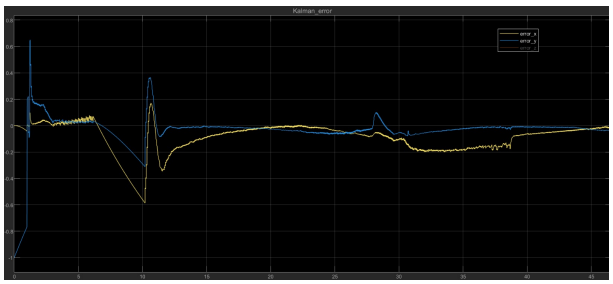

Fig. 20. Kalman filter estimation error - Figure-8 trajectory

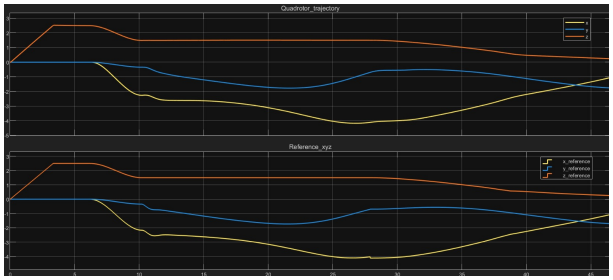

Fig. 21. Desired vs actual position trajectory - Figure-8

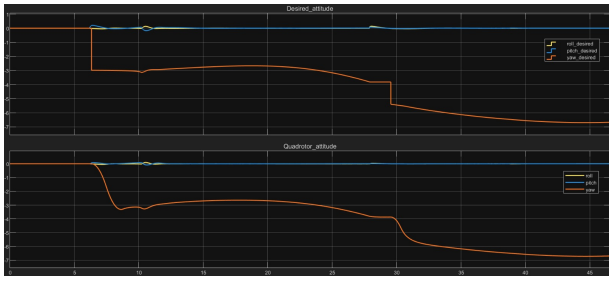

Fig. 22. Desired vs actual attitude - Figure-8

B. Code Snippets

1) Dynamic Model Integration: The quadrotor dynamics (Equations 1–6) are integrated in Simulink using continuous-time state-space representation:

```
function accelerations = dynamic_model(state, u, params)
% dynamic_model - Calcola le accelerazioni lineari e angolari del drone
% Input:
%   state: vettore dello stato [x, y, z, vx, vy, vz, phi, theta, psi, p, q, r]
%   u: vettore di controllo [T, tau_phi, tau_theta, tau_psi]
%   params: parametri [m, g, Ix, Iy, Iz]
% Output:
%   accelerations: [x_ddot, y_ddot, z_ddot, phi_ddot, theta_ddot, psi_ddot]

% Estrai parametri
m = params(1);          % Massa [kg]
g = params(2);
Ix = params(3);
Iy = params(4);
Iz = params(5);

%-----------------------------------------------
%            STATE PARAMETERS IMPORT
%-----------------------------------------------
phi = state(4);     %ROLL
theta = state(5);   %PITCH
psi = state(6);     %YAW

%-----------------------------------------------
%            CONTROL PARAMETERS IMPORT
%-----------------------------------------------

T = u(1);
tau_phi = u(2);     %Control for desired torque for Phi (along x)
tau_theta = u(3);   %Control for desired torque for Pitch (along y)
tau_psi = u(4);     %Control for desired torque for Yaw (along z)

%-----------------------------------------------
%             ACCELERATIONS CALCULATION
%-----------------------------------------------

% lINEAR ACCELERATIONS
x_ddot = -(cos(psi) * sin(theta) * cos(phi) + sin(psi) * sin(phi)) * T / m;
y_ddot = -(sin(psi) * sin(theta) * cos(phi) - sin(phi) * cos(psi)) * T / m;
z_ddot = -cos(theta) * cos(phi) * T / m + g;

% ANGULAR ACCELERATIONS
phi_ddot = tau_phi / Ix;
theta_ddot = tau_theta / Iy;
psi_ddot = tau_psi / Iz;

% Output: accelerazioni lineari e angolari
accelerations = [x_ddot, y_ddot, z_ddot, phi_ddot, theta_ddot, psi_ddot];
end
```

Listing 1. Quadrotor Dynamic Model

2) Position Control with Anti-Windup: Position control computes desired thrust and attitude angles with gravity compensation and integral anti-windup:

```
%-----------------------------------------------
%           TRAJECTORY PARAMETERS IMPORT
%-----------------------------------------------
x_d = desired_traj(1);
y_d = desired_traj(2);
z_d = desired_traj(3);
x_d_dot = desired_traj(4);
y_d_dot = desired_traj(5);
z_d_dot = desired_traj(6);
x_d_ddot = desired_traj(7);
y_d_ddot = desired_traj(8);
z_d_ddot = desired_traj(9);

%-----------------------------------------------
%           OTHER PARAMETERS IMPORT
%-----------------------------------------------
m = params(1);              % MASS
g = params(2);              % GRAVITY ACCELERATION
Kp_pos = params(6:8);       % POSITIONAL GAINS
Kd_pos = params(9:11);      % DERIVATIVE GAINS
Ki_pos = params(12:14);     % INTEGRAL GAINS
dt = params(24);
%-----------------------------------------------
%      VERTICAL THRUST CONTROL CALCULATIONS
%-----------------------------------------------
ez = z_d - z;
ez_dot = z_d_dot - z_dot;

persistent integral_error_z;
if isempty(integral_error_z)
    integral_error_z = 0;
end

T_z_pid = -(Kp_pos(3) * ez + Kd_pos(3) * ez_dot + Ki_pos(3) * integral_error_z + z_d_ddot);

cos_product = cos(phi) * cos(theta);
if abs(cos_product) < 0.1
    cos_product = sign(cos_product) * 0.1;
end
T_total = (m / cos_product) * (g + T_z_pid);

% Anti-Windup Z
T_min = 0.1 * m * g;
T_max = 4.0 * m * g;
T = saturate(T_total, T_min, T_max);

is_saturated_max = (T_total > T_max);
is_saturated_min = (T_total < T_min);

stop_integration = false;
if is_saturated_max && (ez < 0), stop_integration = true;
elseif is_saturated_min && (ez > 0), stop_integration = true;
end

if ~stop_integration
    integral_error_z = integral_error_z + ez * dt;
end

%-----------------------------------------------
%        LATERAL THRUST CONTROL CALCULATIONS
%-----------------------------------------------
ex = x_d - x;
ex_dot = x_d_dot - x_dot;

persistent integral_error_x;
if isempty(integral_error_x), integral_error_x = 0; end

U_x = Kp_pos(1) * ex + Kd_pos(1) * ex_dot + Ki_pos(1) * integral_error_x + x_d_ddot;

ey = y_d - y;
ey_dot = y_d_dot - y_dot;

persistent integral_error_y;
```

```
if isempty(integral_error_y), integral_error_y = 0; end

U_y = Kp_pos(2) * ey + Kd_pos(2) * ey_dot + Ki_pos(2) * integral_error_y + y_d_ddot;

[...]
```

Listing 2. Position Controller

3) Attitude Control with Feedforward: Attitude control tracks desired Euler angles using PID with gyroscopic feedforward:

```
function [tau_phi,tau_theta,tau_psi] = attitude_control(ref_ang_pos,ref_ang_vel, ref_ang_accel,state,params)

% State and Parameter Import
phi = state(4); theta = state(5); psi = state(6);
phi_dot = state(10); theta_dot = state(11); psi_dot = state(12);

Ix = params(3); Iy = params(4); Iz = params(5);
Kp_ang = params(15:17); Kd_ang = params(18:20); Ki_ang = params(21:23);
dt = params(24);

% -----------------------
% 1. ROLL CONTROL (Phi)
% -----------------------
e_phi = ref_ang_pos(1) - phi;
e_phi_dot = ref_ang_vel(1) - phi_dot;

persistent integral_error_phi;
if isempty(integral_error_phi), integral_error_phi = 0; end

% PID
tau_phi_pid = Kp_ang(1) * e_phi + Kd_ang(1) * e_phi_dot + Ki_ang(1) * integral_error_phi;

% SIMPLIFIED FEEDFORWARD
tau_phi_ff = Ix * ref_ang_accel(1);

% Total & Saturation
tau_phi_total = tau_phi_pid + tau_phi_ff;
tau_max_phi = 1.0;
tau_phi = saturate(tau_phi_total, -tau_max_phi, tau_max_phi);

% Anti-Windup
if (abs(tau_phi_total) <= tau_max_phi) || (sign(e_phi) ~= sign(tau_phi))
    integral_error_phi = integral_error_phi + e_phi * dt;
end

% -----------------------
% 2. PITCH CONTROL (Theta)
% -----------------------
e_theta = ref_ang_pos(2) - theta;
e_theta_dot = ref_ang_vel(2) - theta_dot;

persistent integral_error_theta;
if isempty(integral_error_theta), integral_error_theta = 0; end

% PID
tau_theta_pid = Kp_ang(2) * e_theta + Kd_ang(2) * e_theta_dot + Ki_ang(2) * integral_error_theta;

% FEEDFORWARD
tau_theta_ff = Iy * ref_ang_accel(2);

% Total & Saturation
tau_theta_total = tau_theta_pid + tau_theta_ff;
tau_max_theta = 1.0;
tau_theta = saturate(tau_theta_total, -tau_max_theta, tau_max_theta);

% Anti-Windup
if (abs(tau_theta_total) <= tau_max_theta) || (sign(e_theta) ~= sign(tau_theta))
    integral_error_theta = integral_error_theta + e_theta * dt;
end

% -----------------------
% 3. YAW CONTROL (Psi)
% -----------------------
e_psi = ref_ang_pos(3) - psi;
% Angle Wrapping
while e_psi > pi, e_psi = e_psi - 2*pi; end
while e_psi < -pi, e_psi = e_psi + 2*pi; end

e_psi_dot = ref_ang_vel(3) - psi_dot;

persistent integral_error_psi;
if isempty(integral_error_psi), integral_error_psi = 0; end

% PID
tau_psi_pid = Kp_ang(3) * e_psi + Kd_ang(3) * e_psi_dot + Ki_ang(3) * integral_error_psi;

% SIMPLIFIED FEEDFORWARD
tau_psi_ff = Iz * ref_ang_accel(3);

% Total & Saturation
tau_psi_total = tau_psi_pid + tau_psi_ff;
tau_max_psi = 0.5;
tau_psi = saturate(tau_psi_total, -tau_max_psi, tau_max_psi);

% Anti-Windup
if (abs(tau_psi_total) <= tau_max_psi) || (sign(e_psi) ~= sign(tau_psi))
    integral_error_psi = integral_error_psi + e_psi * dt;
end

end

function val_sat = saturate(val, min_val, max_val)
    val_sat = max(min(val, max_val), min_val);
end
```

Listing 3. Attitude Controller

4) Platform Detection via PnP: Vision module detects tag and solves Perspective-n-Points to recover 3D platform position:

```
function [sys,x0,str,ts,simStateCompliance] = coppelia_camera(t,x,u,flag,varargin)

%==================================
%         COPPELIA_CAMERA
%==================================
%
% INPUTS (1 elements):
%   u = connection_status - 1 if connected to CoppeliaSim, 0 otherwise
%
% OUTPUTS (5 elements):
%   [platform_x, platform_y, platform_z, detection_flag, fov]
%
% PARAMETERS (varargin):
%   varargin = [sample_time, tag_size, enable_logging, enable_adaptive_focus]
```

```
%%================================
[...]
% FUNCTION TO DETECT LANDING PLATFORM FROM IMAGE -
function [platform_pos, detected, debug_imgs] =
    detect_landing_platform(...
    image, camera_params, tag_size, enable_logging,
     frame_count)

    %initialize outputs
    detected = false;
    platform_pos = [0; 0; 0];
    debug_imgs = struct();


    gray_image = rgb2gray(image);
                            %convert to grayscale
    for processing
    binary_image = imbinarize(gray_image, 'adaptive
    ', 'Sensitivity', 0.20); %adaptive thresholding
     to binary


    debug_imgs.original = image;
                            % Store original image
    debug_imgs.binary = binary_image;
                            % Store binary image

    [landing_tag, tag_corners] =
    find_largest_quadrangle(binary_image);     %
    Find largest quadrangle in binary image (using
    helper function)
    %landing tag is a binary mask of the detected
    tag region, could be printed for debugging

    % DEBUGGING LOGS
    if isempty(landing_tag)
        %fprintf('[DEBUG] No quadrangle found\n');
        cc = bwconncomp(binary_image);
                            %function to find
    connected components, from Image Processing
    Toolbox
        %fprintf('[DEBUG] Found %d regions\n', cc.
    NumObjects);
    else
        %fprintf('[DEBUG] Quadrangle found, corners
    : %d\n', size(tag_corners,1));
    end

    % Early exit if no landing tag found
    if isempty(landing_tag)
        debug_imgs.detection = image;
        return;
    end


    debug_imgs.tag_region = landing_tag; % Store
    tag region image


    %MODIFICATION WITH DIAGNOSTIC LOGGING OF RATIO
    BETWEEN CIRCLE AND QUADRANGLE

    [circle_found, circle_corners, circle_center,
    circle_radius] = detect_circle_pattern(
    landing_tag, tag_size);

    % THIS IF CIRCLE HAS PRIORITY
    if circle_found
        corners_2d = circle_corners;
        if enable_logging
            %fprintf('  -> Circle pattern detected\
    n');
        end
    else
        [cross_found, cross_corners] =
    detect_cross_pattern(landing_tag);
        if cross_found
            corners_2d = cross_corners;
            if enable_logging
                %fprintf('  -> Cross pattern
    detected\n');
            end
        else
            corners_2d = tag_corners;
            if enable_logging
                %fprintf('  -> Using quadrangle
    corners\n');
            end
        end
    end

    % Normalize corner order and flip Y

    corners_2d = normalize_corner_order(corners_2d)
    ;
    corners_2d(:,2) = camera_params.height -
    corners_2d(:,2);

    % RANSAC with validation
    num_corners_before = size(corners_2d, 1);
    corners_2d = ransac_filter_corners(corners_2d);
    num_corners_after = size(corners_2d, 1);

    if enable_logging
        %fprintf('  [RANSAC] Corners: %d -> %d (
    filtered %d)\n', ...
            %num_corners_before, num_corners_after,
     num_corners_before - num_corners_after);
    end

    % CRITICAL: Validate RANSAC output
    if size(corners_2d, 1) < 4
        ransac_fail_count = ransac_fail_count + 1;
        if enable_logging
            %fprintf('  [RANSAC FAIL] Only %d/4
    corners - Frame %d - Total fails: %d\n', ...
             %   num_corners_after, frame_count,
    ransac_fail_count);
        end
        debug_imgs.detection = image;
        debug_imgs.ransac_fail = true;
        debug_imgs.corners_before =
    num_corners_before;
        debug_imgs.corners_after =
    num_corners_after;
        return;
    end

    s = tag_size / 2;
    corners_3d = [-s, s, 0; s, s, 0; s, -s, 0; -s,
    -s, 0]; % Counter-clockwise from top-left

    [R, t, success] = solve_pnp(corners_3d,
    corners_2d, camera_params,enable_logging);

    if success
        corner_spacing = norm(corners_2d(1,:) -
    corners_2d(2,:)); % pixels
        expected_spacing = (tag_size / t(3)) *
    camera_params.focal_length;
        %fprintf('Corner spacing: %.1f px (detected
    ) vs %.1f px (expected at Z=%.2f)\n', ...
            %corner_spacing, expected_spacing, t(3)
    );
    end


    if success
        platform_pos = t(:);
```

```
        detected = true;
        debug_imgs.detection = image; % Simple - no markers
    else
        debug_imgs.detection = image;
    end
end
[...]
function platform_world = transform_to_world(platform_cam, quad_pos, quad_ori)
    roll = quad_ori(1);
    pitch = quad_ori(2);
    yaw = quad_ori(3);
    Rx = [1, 0, 0; 0, cos(roll), -sin(roll); 0, sin(roll), cos(roll)];
    Ry = [cos(pitch), 0, sin(pitch); 0, 1, 0; -sin(pitch), 0, cos(pitch)];
    Rz = [cos(yaw), -sin(yaw), 0; sin(yaw), cos(yaw), 0; 0, 0, 1];
    R_quad = Rz * Ry * Rx;
    R_cam_to_body = [1, 0, 0; 0, 1, 0; 0, 0, -1]; % flips Z axis to obtain the word

    cam_offset_body = [0; 0; -0.060];  % Camera is 6cm below quad center
    platform_world = quad_pos(:) + R_quad * (R_cam_to_body * platform_cam + cam_offset_body);
end
[...]
```

Listing 4. Vision-Based Platform Detection

5) Kalman Filter with Attitude Gating: Extended Kalman Filter estimates platform state with gating to prevent noisy updates:

```
function [sys,x0,str,ts,simStateCompliance] = coppelia_kalman(t,x,u,flag,varargin)

persistent x_hat P initialized
persistent F H Q R sample_time enable_logging frame_count



switch flag

    case 0  % Initialization
        if nargin >= 5 && ~isempty(varargin{1})
            sample_time = varargin{1}(1);
            q_pos = (length(varargin{1}) >= 2) && varargin{1}(2) || 0.01;
            q_vel = (length(varargin{1}) >= 3) && varargin{1}(3) || 0.1;
            r_meas = (length(varargin{1}) >= 4) && varargin{1}(4) || 0.05;
            enable_logging = (length(varargin{1}) >= 5) && varargin{1}(5);
        else
            sample_time = 0.05;
            q_pos = 0.1;
            q_vel = 1.0;
            r_meas = 0.05;
            enable_logging = 0;
        end

        dt = sample_time;
        F = [1 0 0 dt 0  0;
             0 1 0 0  dt 0;
             0 0 1 0  0  dt;
             0 0 0 1  0  0;
             0 0 0 0  1  0;
             0 0 0 0  0  1];

        H = [1 0 0 0 0 0;
             0 1 0 0 0 0;
             0 0 1 0 0 0];

        Q = diag([q_pos, q_pos, q_pos, q_vel, q_vel, q_vel]);
        R = diag([r_meas, r_meas, r_meas]);

        x_hat = zeros(6, 1);
        P = eye(6) * 10;
        initialized = false;

        frame_count = 0;

        sizes = simsizes;
        sizes.NumContStates = 0;
        sizes.NumDiscStates = 0;
        sizes.NumOutputs = 8;
        sizes.NumInputs = 6;
        sizes.DirFeedthrough = 1;
        sizes.NumSampleTimes = 1;

        sys = simsizes(sizes);
        x0 = [];
        str = [];
        ts = [sample_time 0];
        simStateCompliance = 'UnknownSimState';

        %fprintf('[Kalman Filter] Initialized (dt=%.3f, Q_pos=%.3f, Q_vel=%.3f, R=%.3f)\n', ...
            %sample_time, q_pos, q_vel, r_meas);

    case 3  % Output
        frame_count = frame_count + 1;

        z_meas = u(1:3);
        detected = u(4);
        %fov = u(5); % Not used in Kalman filter
        enabled = u(6);  % Nuovo input

        if enable_logging && mod(frame_count, 20) == 0
            %fprintf('[KALMAN] Frame %d, detected=%d, enabled=%d\n', frame_count, detected, enabled);
        end

        % Initialize on first detection
        if ~initialized && detected == 1 && enabled == 1
            x_hat = [z_meas; 0; 0; 0];
            P = eye(6) * 10;
            initialized = true;
            %fprintf('[Kalman] INITIALIZED at [%.3f, %.3f, %.3f]\n', ...
                %z_meas(1), z_meas(2), z_meas(3));
        end

        if ~initialized
            sys = [0; 0; 0; 0; 0; 0; 0; 0];
            return;
        end

        % Se enabled = 0, restituisci le stime precedenti senza aggiornare
        if enabled == 0
            if enable_logging
                fprintf('  [DISABLED] Using previous estimates\n');
            end
            sys = [x_hat; detected; initialized];
            return;
        end

        % Prediction
        x_hat_pred = F * x_hat;
```

```
        P_pred = F * P * F' + Q;

        if enable_logging && detected == 1
            %fprintf('  [PREDICT] pos=[%.3f, %.3f, %.3f], vel=[%.3f, %.3f, %.3f]\n', ...
              %   x_hat_pred(1), x_hat_pred(2), x_hat_pred(3), ...
               %  x_hat_pred(4), x_hat_pred(5), x_hat_pred(6));
        end

        % Update
        if detected == 1
            y = z_meas - H * x_hat_pred;
            S = H * P_pred * H' + R;
            K = P_pred * H' / S;
            x_hat = x_hat_pred + K * y;
            P = (eye(6) - K * H) * P_pred;

            if enable_logging
                %fprintf('  [UPDATE] innovation =[%.3f, %.3f, %.3f]\n', y(1), y(2), y(3));
                fprintf('  [CORRECTED] pos=[%.3f, %.3f, %.3f], vel=[%.3f, %.3f, %.3f]\n', ...
                    x_hat(1), x_hat(2), x_hat(3), ...
                    x_hat(4), x_hat(5), x_hat(6));
            end
        else
            x_hat = x_hat_pred;
            P = P_pred;
            if enable_logging
                %fprintf('  [NO UPDATE] Using prediction only\n');
            end
        end

        sys = [x_hat; detected; initialized];

[...]
```

Listing 5. Kalman Filter with Attitude Gating

6) State Machine Logic: Four-state finite state machine governs quadrotor behavior:

```
function [state, traj] = stateManager(position, platform_position, detected, t)

    % Persistent variables (maintain values between time steps)
    persistent current_state hover_start_time last_t position_error_start_time detection_start_time state3_start_time state3_start_pos


    % Initialize on first run
    if isempty(current_state)
        current_state = 1;  % Start in takeoff state
        hover_start_time = -1;
        last_t = 0;
        position_error_start_time = -1;
        detection_start_time = -1;

        state3_start_time = -1;
        state3_start_pos = [0,0,0];
    end

    % Calculate dt automatically
    dt = t - last_t;
    last_t = t;

    % Parameters
    altitude_threshold = 0.1;
    hover_time_required = 10.0;
    position_error_threshold = 0.8;
    position_error_time_required = 15.0;
    landing_altitude = 0.05;
    target_hover_height = 2.5;

[...]

    % State machine logic
    switch current_state
        %% First case: Takeoff and hovering
        case 1  % TAKEOFF STATE
            % Hovering takeoff trajectory logic
            v_const = 0.75;   % Ascent speed [m/s]
            expected_z = v_const * t;

            if expected_z < target_hover_height
                % Rising phase
                pos = [0, 0, expected_z];
                vel = [0, 0, v_const];
                acc = [0, 0, 0];
                traj = [pos,vel,acc];
            else
                % Hovering phase
                pos = [0, 0, target_hover_height];
                vel = [0, 0, 0];
                acc = [0, 0, 0];
                traj = [pos,vel,acc];
            end

            if altitude >= (target_hover_height - altitude_threshold)

                %--- Hover start timer ---
                if hover_start_time < 0
                    hover_start_time = t;
                end

                %--- Detection timer ---
                if detected
                    if detection_start_time < 0
                        detection_start_time = t; % Start counting
                    end
                else
                    detection_start_time = -1; % Reset if detection lost
                end

                time_at_hover = t - hover_start_time;
                time_detected = (detection_start_time > 0) * (t - detection_start_time);

                %--- Transition to state 2 if detection $ \geq $ 3 seconds ---
                if time_detected >= 3.0
                    current_state = 2;
                    fprintf('Switched to STATE 2 at time %f \n',t);
                    hover_start_time = -1;
                    detection_start_time = -1;
                end

                %--- Transition to state 3 if hover $ \geq $ 10 seconds ---
                if time_at_hover >= hover_time_required
                    current_state = 3;
                    hover_start_time = -1;
                    detection_start_time = -1;

                    % Reset State 3 smoothing timers so they start fresh
```

```
                    state3_start_time = -1;
                    fprintf('Switched to STATE 3 at time %f \n',t);
                end

[...]
        %% SECOND CASE: FOLLOWING
        case 2  % FOLLOW STATE
            % Reset State 3 timer just in case we came from there
            state3_start_time = -1;

            % Add your follow trajectory here
            if mod(t,2) == 0
                fprintf('Position tracking error: %g\n', position_error);
            end
            if position_error < position_error_threshold
                if position_error_start_time < 0
                    position_error_start_time = t;
                end

                time_at_low_error = t - position_error_start_time;
                if mod(t,1) == 0
                    fprintf('Time inside the bound :%g\n',time_at_low_error);
                end

                if time_at_low_error >= position_error_time_required
                    current_state = 4;
                    fprintf('Switched to STATE 4: Landing!\n');
                    position_error_start_time = -1;
                end
            else
                position_error_start_time = -1;
            end

[...]

        %% THIRD CASE: LOOKING FOR THE PLATFORM ( RANDOM WALK)
        case 3  % Go on a circular trajectory while trying to find the moving platform

            if state3_start_time < 0
                state3_start_time = t;
                state3_start_pos  = position; % Snapshot current actual position (e.g., [0,0,2.5])
            end

            % 3. Apply Smoothing Logic (First 3 seconds)
            T_trans = 5.0; % Transition duration
            t_local = t - state3_start_time;

            if t_local < T_trans
                angular_velocity =0.05;

[...]

                % Transition complete: track circle directly
                pos = target_pos;
                vel = target_vel;
                acc = target_acc;
            end

            traj = [pos, vel, acc];

            % --- Detection Logic ---
            if detected
                if detection_start_time < 0
                    detection_start_time = t;    % Start counting
                end
            else
                detection_start_time = -1;       % Reset if detection lost
            end

            time_detected = (detection_start_time > 0) * (t - detection_start_time);

            % if time_detected > 0
            %     angular_velocity = 0.2/exp(time_detected);
            % else
            %     angular_velocity = 0.2;
            % end

            if  time_detected >= 2.0
                current_state = 2;
                fprintf('Finally found, starting Tracking')
                detection_start_time = -1;
                state3_start_time = -1;
            end

[...]

        %% LANDING
        case 4 % Landing
            fprintf('LANDING');
            if altitude < landing_altitude
            % Stay in landing state
            end

[...]
```

Listing 6. State Machine

7) Control Gains (Parameters): Final control gains used in the unknown trajectory implementation:

```
function [m,g,Ix,Iy,Iz,Kp_pos,Kd_pos,Ki_pos,Kp_ang,Kd_ang,Ki_ang,dt] = model_n_control_param()
%% Parametri del drone
m = 1.0;        % Massa [kg]
g = 9.81;       % Gravita [m/s^2]
Ix = 0.1;       % Momento d'inerzia asse x [kg*m^2]
Iy = 0.1;       % Momento d'inerzia asse y [kg*m^2]
Iz = 0.2;       % Momento d'inerzia asse z [kg*m^2]

%% GUADAGNI PID
%Controllo Posizione (x, y, z)
Kp_pos = [20, 20,30];    % 2.4 2.4 15
Kd_pos = [15,15,40];
Ki_pos = [8,8,16];  % 0.01,0.01

% Controllo Angoli (phi, theta, psi)
Kp_ang = [35,35,150];
Kd_ang = [40,40,70]; %0.2 0.2
Ki_ang = [40,40,40];


%% Time step (Overwritten by Simulink Sample_Time)
dt = 0.0001;
end
```

Listing 7. Control Gains

C. System Architecture Diagrams

This section presents the complete system architecture through three block diagrams: the overall system structure, the state machine logic, and the CoppeliaSim integration details.

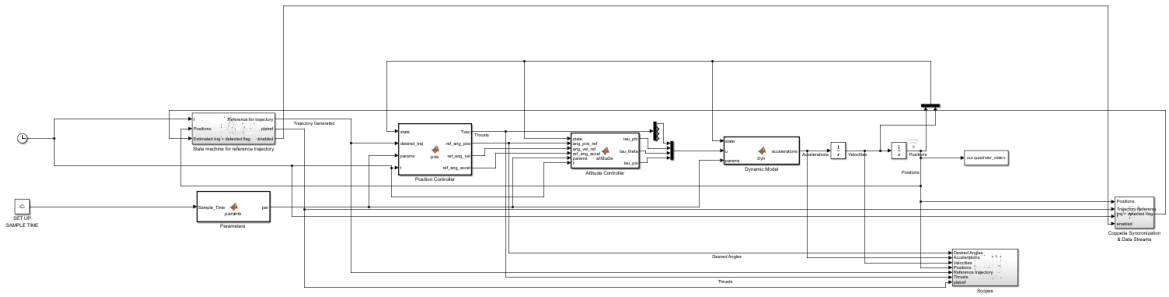

Fig. 23. Overall system architecture showing all functional modules and data flow between Simulink components and CoppeliaSim.

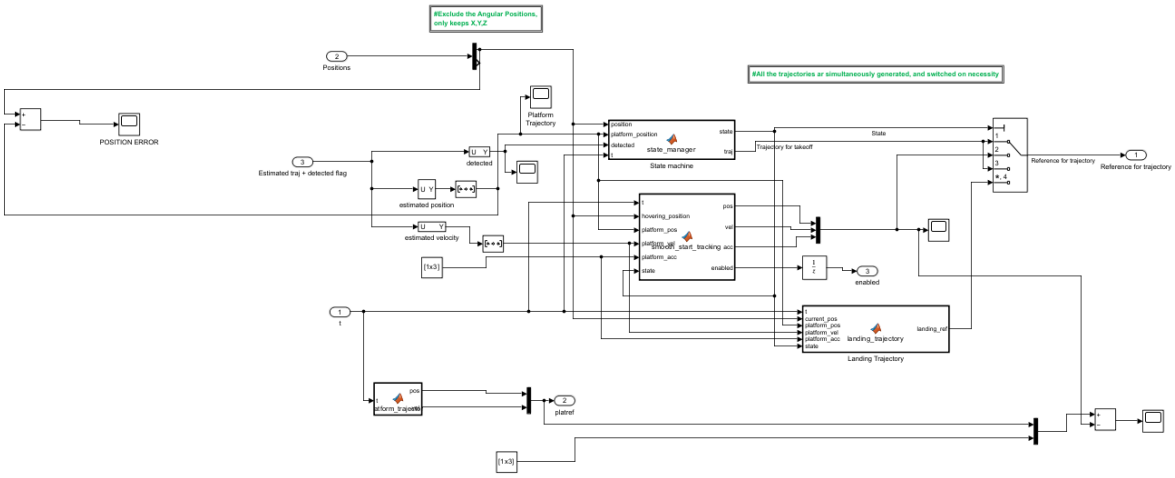

Fig. 24. State machine logic showing transitions between TAKEOFF, SEARCH, FOLLOW, and LANDING states based on detection status and tracking error.

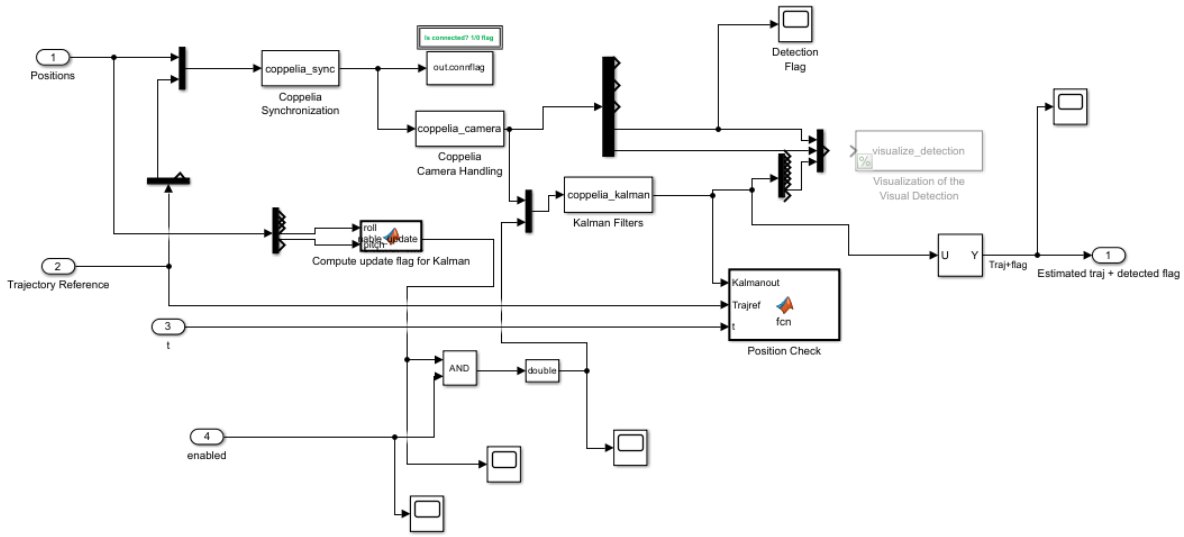

Fig. 25. CoppeliaSim integration architecture detailing the S-function blocks (coppelia_sync, coppelia_camera, coppelia_kalman) and their communication with the simulator for visualization and camera image acquisition.

References

[1] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, "Vision-based autonomous quadrotor landing on a moving platform," in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, IEEE, Oct. 2017. DOI: 10.1109/SSRR.2017.8088164