Seminar Smart Systems 2026 Drone & Co.

Markus Nöth Richard Schröder Mariella Arlt Filip Müller Christian Model
Sebastian Nagles Hasan Yildiz Steffen Krutzsch Macro Schmieg Toni Kostov Arndt Balzer
{markus.noeth | richard.schroeder | mariella.arlt | filip.mueller | christian.model.1}@study.thws.de
{sebastian.nagles | hasan.yildiz | steffen.krutzsch | marco.schmieg.1 | toni.kostov}@study.thws.de
{arndt.balzer}@thws.de
Faculty of Computer Science and Business Information Systems (FIW)
Technical University of Applied Sciences Wuerzburg-Schweinfurt (THWS)
Sanderheinrichsleitenweg 20, 97074 Würzburg, Germany

***Abstract*—This combined seminar paper summarizes core smart-system concepts for drone applications. The individual mini papers focus on simultaneous localization and mapping, Kalman filtering, orientation estimation, PID control, and swarm intelligence. Together, these topics describe how autonomous drones can perceive, estimate, control, and coordinate their behavior in dynamic environments.**

***Index Terms*—SLAM, Kalman Filter, orientation, PID, swarm intelligence, drones, ROS2, RViz, Gazebo**



Fig. 1. Crazyflie 2.1 Brushless

I. SLAM

Authors: Markus Nöth, Richard Schröder

Abstract

Simultaneous localization and mapping, or SLAM, is the process of determining a robot's location while also creating a map of the surrounding environment. It is needed when a robot has to move in an area that it has never seen before or only partly knows, especially when it cannot use external positioning systems such as GPS. This paper first explains SLAM and its typical processing steps. It then discusses common methods for estimating positions, building maps, handling errors, closing loops, and planning paths. The last part discusses the Crazyflie 2.1 Brushless platform with optical flow and Time-of-Flight sensors. The sensor configuration supports indoor relative localization and conservative occupancy mapping. Sparse range measurements, limited computation, and weak place-recognition evidence restrict the accuracy of the map and the reliability of loop closure compared with dense LiDAR or feature-rich camera systems.

A. Introduction

Autonomous robots need to know where they are and what the environment around them looks like. These tasks are known as localization and mapping. Localization means estimating the robot's position and orientation. Mapping means building a representation of the environment. This representation can include walls, obstacles, free space, rooms, or landmarks.

SLAM combines both tasks. The key issue is often described as a chicken-and-egg problem. A robot needs a map to know where it is, but it also needs to know where it is to build a correct map. If the pose estimate is wrong, new measurements are inserted into the wrong place in the map. If the map is wrong, the localization will also be worse later on. Because of this, SLAM is both a mapping and a localization method [1]–[3].

SLAM is useful in many robotic systems. Indoor mobile robots, drones, vacuum robots, logistics robots, autonomous vehicles, and planetary rovers may all require SLAM when they move in areas where a reliable map or localization system is not available. In these situations, the robot uses onboard sensors to estimate its own position. It then uses this information to create a map. This map can be used later to help the robot navigate and plan its path [2], [4].

The SLAM problem can be written in a simple probabilistic way as follows:

$$p(x_{0:t}, m \mid z_{1:t}, u_{1:t}),$$

where $x_{0:t}$ is the robot's trajectory up to time t, m is the map, $z_{1:t}$ are the sensor measurements, and $u_{1:t}$ are the control or motion inputs. This notation shows that the trajectory and the map are estimated together from all available motion and sensor information [1], [3].

Modern SLAM systems normally consist of several algorithms that work together. The front end processes the sensor data, finds useful information, and identifies possible connections between current measurements and earlier map elements. A back end estimates the most consistent robot poses and the map state from these constraints. Therefore, SLAM is better viewed as a pipeline, i. e. a set of algorithms that the robot follows while it is moving [2], [5].

B. Basic SLAM Workflow

A typical SLAM system runs in cycles. Each cycle uses new sensor data to update the estimated pose and map. The updated pose and map then become the input for the next cycle. The exact steps will depend on the sensor setup, but many systems follow similar steps.

1) Sensor Data Acquisition: The first step is to collect raw data from one or more sensors. Some examples of sensors include cameras, LiDAR scanners, IMUs, wheel encoders, depth sensors, and Time-of-Flight sensors. These sensors provide information about the robot's movement and the surrounding environment. In a ground robot, wheel odometry can measure how far the robot has moved. In an aerial robot, IMU data and optical flow can help estimate motion. Range sensors can measure how far the robot is from walls, floors, ceilings, or other obstacles [2], [6].

2) Data Preprocessing: Raw sensor data contains noise, timing errors, and invalid values. Preprocessing undistorts camera images, removes invalid values and noisy points, synchronizes timestamps, and applies sensor calibration. Later matching and estimation stages depend on these corrections; inaccurate measurements produce incorrect map updates or pose corrections [2], [6].

3) Feature Extraction and Data Association: Feature extraction converts sensor data into observations used for estimation. Camera-based systems detect visual features, LiDAR systems identify edges, planes, or scan structures, and occupancy-grid systems convert range measurements into free and occupied cell updates. Some systems also select keyframes to represent the trajectory and map with fewer stored poses.

After extracting this information, the system must decide which new measurements go with which known map elements. This is called data association [2], [7].

4) Pose Estimation: Pose estimation determines where the robot is and what its orientation is. A pose can be written as position (x, y, z) and orientation $(roll, pitch, yaw)$. Many systems first predict the new pose from the movement information and then correct this prediction using sensor observations.

Wheel odometry predicts movement on a ground robot, an IMU supplies acceleration and rotation measurements, a camera compares consecutive images, and range sensors compare measured distances with the current map. Errors accumulate in the prediction over time. Observations of the environment constrain that drift [1], [6].

5) Map Update: Once the system has estimated the current pose, it adds new information to the map. In a landmark map, this is done by adding or updating the positions of landmarks. In a point-cloud map, this means adding measured 3D points. In an occupancy grid, the system marks cells as free or occupied. The quality of the map update depends on the pose estimate. If the robot's pose is wrong, the map will be distorted [3], [8].

6) Local Optimization: Many systems improve recent poses and nearby map elements after a first estimate has been made. This can reduce local errors by optimizing the last few keyframes, reducing reprojection error in a camera system, or improving the fit between recent scans and the local map. Local optimization does not always fix global drift, but it does make the most recent part of the trajectory more consistent [2], [5].

7) Loop Closure and Map Optimization: Loop closure checks whether the robot has returned to a place that it has already visited. If the system detects a loop closure correctly, it can add a new constraint between the current pose and an older pose. This constraint can fix the error caused by drift over the whole trajectory. Graph-based SLAM systems are well suited for this because they represent poses as nodes and measurements as constraints between nodes. Adding a loop closure makes the graph easier to handle. This helps the full map and trajectory to be more consistent. But adding a wrong loop closure will corrupt the map [1], [2], [5], [7].

C. Common SLAM Types

SLAM systems can be grouped either by their estimation backend or by their main sensor modality. Table I gives a short overview of some common types.

EKF SLAM and Graph SLAM mainly describe the estimation method, while Visual SLAM and LiDAR-SLAM mainly describe the sensor setup [2], [5]–[7], [9].

D. Position Estimation Methods

Position estimation can use self-motion measurements, environment-relative measurements, or both. Self-motion estimation asks: how did the robot move? Environment-relative estimation is how the environment moved compared to the robot, or how the current observation matches the map.

Odometry and IMU measurements are valuable because they are available at high rates and can predict motion quickly. Their main weakness is drift. Even a small error in each step adds up over time. After a long flight or drive, the estimated position can be far away from the true position.

Sensors that are sensitive to the environment can help reduce this problem. A LiDAR scanner, time-of-flight (ToF) sensor, ultrasonic sensor, depth camera, or standard camera observes the surroundings. These observations can be compared with previous ones or with a map. If the robot recognizes structures it has seen before, it can adjust its estimate of the pose. Dead reckoning uses the robots previous pose and measured motion, for example from wheel encoders or an IMU, to estimate its current pose. As it does not use environmental observations for correction, small errors accumulate over time. However, SLAM can reduce this drift by detecting landmarks or previously visited places and correcting the map with this new knowledge. [1], [2]. In probabilistic SLAM, the joint estimation of the current pose and the map is often written as a recursive belief update. A compact form is

$$\begin{aligned}&\mathrm{bel}_t(x_t, m)\\ &= \eta\, p(z_t \mid x_t, m) \int p(x_t \mid x_{t-1}, u_t)\, \mathrm{bel}_{t-1}(x_{t-1}, m)\, dx_{t-1}.\end{aligned}$$

where the prediction model $p(x_t \mid x_{t-1}, u_t)$ describes the expected motion, and the measurement model $p(z_t \mid x_t, m)$

TABLE I: Common SLAM types.

Type	Main idea	Typical limitation
EKF SLAM	Uses an Extended Kalman Filter to update a joint state estimate of robot pose and map features.	Linearization errors and wrong covariance estimates can make the result inconsistent.
Graph SLAM	Represents poses or landmarks as nodes and measurements as constraints between them. The graph is optimized after new constraints, especially loop closures.	Depends strongly on correct constraints and loop-closure validation.
Visual SLAM	Uses camera images, visual features, and keyframes to estimate motion and build a map.	Sensitive to lighting, motion blur, texture, and feature matching quality.
LiDAR-SLAM	Uses laser or range scans for scan matching, mapping, and loop closure.	Needs suitable geometry and can require more sensor hardware than a lightweight ToF setup.

TABLE II: Overview of position estimation methods.

Method	Principle	Quality	Main problems	Advantage	Category
Odometry	Measures movement using motor, wheel, or drive-unit data	Low to medium	Drift over time, wheel slip	Often available	Self-motion
IMU	Uses acceleration and angular velocity	Medium	Noise and integration drift	Cheap and fast	Self-motion
LiDAR, ToF, Ultrasound	Measures distance to external surfaces	Good when geometry is visible	Needs extra hardware and suitable surfaces	Uses external references	Environment-relative
Camera	Compares images or visual features	Depends strongly on environment	Sensitive to lighting, texture, and computation cost	Rich visual information	Environment-relative

describes how likely the current sensor measurement is for a given pose and map. The normalization constant η ensures that the belief integrates to one. This form shows the two main parts of state estimation: prediction and correction [1], [3].

A more complete overview of different methods can be seen in table II.

1) Monte Carlo Position Estimation: Monte Carlo position estimation represents a guess about the robot pose with many weighted particles. Each particle is one possible robot position. In the prediction step, the particles are moved with the motion model. In the correction step, each particle is weighted by how well its expected sensor readings match the real measurements. During resampling, likely particles are kept and unlikely particles are discarded.

This particle representation can handle several possible pose hypotheses at the same time, which is useful when the start position is uncertain or when measurements are ambiguous. Its quality depends on the number of particles, the motion model, and the sensor model [8], [10].

2) State Estimation and SLAM: The Crazyflie state estimator combines IMU, optical-flow, and downward range measurements to estimate position, velocity, and attitude. This estimate supplies the local motion data used by the SLAM system. It neither estimates environmental landmarks nor corrects the trajectory when the drone revisits a previous location.

The state estimator acts as the odometry source for the SLAM frontend. Range measurements are attached to estimated poses to form local scans. The backend compares selected scans and adds constraints when two poses represent the same location. These constraints correct drift that local state estimation alone cannot remove [2], [11].

E. Map Representations

In SLAM, a map is a navigation-oriented environment representation. It stores information in a way that supports localization, planning, or control. Different SLAM systems use different map types depending on the task and the sensors.

1) Two-Dimensional Maps: In 2D SLAM, the most common representations are occupancy grids, probabilistic occupancy grids, feature maps, topological maps, and semantic maps.

An occupancy grid divides the environment into cells. Each cell is marked as free, occupied, or unknown. A probabilistic occupancy grid stores the probability that each cell is occupied. This helps when sensor data is unreliable. A feature or landmark map stores detected landmarks instead of full free-space information. A topological map stores places and connections as a graph. A semantic map adds labels such as walls, doors, or rooms [2], [3], [8].

Occupancy grids support navigation by distinguishing free cells from blocked cells and can be used directly by grid-based path-planning algorithms. Memory demand grows quadratically with the linear dimensions of the mapped environment and with the inverse of the cell size, so larger environments and smaller grid cells rapidly increase memory usage.

2) Three-Dimensional Maps: In 3D mapping, common representations include point clouds, octree-based occupancy maps, TSDF maps, and ESDF maps. A point cloud stores measured 3D points. An octree map divides space into levels,

and it can represent occupied and free 3D cells more efficiently than a full grid. A TSDF map stores the signed distance to the nearest surface within a limited area around it, which helps reconstruct smooth surfaces. An ESDF maps stores the Euclidean distance from each cell to the nearest surface, making it especially useful for navigation and collision avoidance [2], [4].

Dense 3D maps require greater sensor coverage, memory, and processing capacity than 2D occupancy maps. A small robot with sparse range sensing cannot observe enough of the environment to support a detailed reconstruction. Its map representation must match the available measurements, computing resources, and navigation task.

3) Point-Based and Occupancy Maps: Projecting range endpoints into world coordinates produces a point-based map. It shows where the sensors detected surfaces, but it does not distinguish free, occupied, and unobserved space.

An occupancy grid also evaluates the path from the sensor to the measured endpoint. Cells crossed by a valid sensor ray are updated towards free space, while the endpoint is updated towards occupied space. Repeated measurements are combined through a probabilistic update, commonly represented in log-odds form [3], [8].

The experimental result in Fig. 3 is a point-based map. An occupancy grid is discussed as a possible onboard representation, but it was not implemented in the presented experiment.

4) Occupancy Grid Update: A range measurement can update an occupancy grid using ray tracing, a computer graphics technique. The cells between the robot and the measured obstacle are updated as more likely to be free. The cell near the endpoint is updated as more likely to be occupied. A common way to represent these updates is to use log-odds:

$$L_t(c) = L_{t-1}(c) + \log \frac{p(c \text{ occupied} \mid z_t, x_t)}{1 - p(c \text{ occupied} \mid z_t, x_t)} - L_0(c),$$

where $L_t(c)$ is the log-odds occupancy value of cell c at time t, z_t is the current measurement, x_t is the robot pose, and $L_0(c)$ is the prior log-odds value. Repeated measurements increase or decrease the confidence that a cell is occupied [3], [8].

F. Typical SLAM Problems

Errors from odometry or IMU integration accumulate when no external observation constrains the pose estimate. A trajectory may remain locally consistent while its position and heading diverge from the actual path over time. Measurements inserted at these faulty poses also distort the map.

Sensor errors differ by measurement principle. IMU measurements contain noise and bias, while ToF sensors have range limits and depend on surface properties and ambient conditions. Cameras require sufficient texture and illumination. LiDAR and ultrasound measurements also contain noise and outliers. A SLAM estimator must represent these uncertainties and reject measurements that are inconsistent with the remaining observations [2], [6].

Wind moves a drone from its commanded path, and rotor-induced airflow causes additional disturbances near walls or the ground. Dust and rain degrade some sensor measurements. People and moving furniture produce observations that violate the static-map assumption. Changes in the environment also reduce the agreement between current measurements and an older map [2].

Perceptual aliasing occurs when different places produce similar observations. Two corridors with similar geometry, for example, can yield range measurements that pass a place-recognition test. Accepting such a match as a loop closure adds an incorrect constraint and destroys the map [2], [12].

G. Typical Solutions

Sensor fusion combines measurements from sources such as an IMU, optical flow sensor, range sensor, camera, or wheel encoder. Their error characteristics and observation directions differ, so one source can constrain state variables that another source cannot observe. Fusion only improves the estimate when the measurement models include sensor uncertainty, bias, timing, and calibration errors [2], [6].

Filtering estimates the robot state from a motion model and uncertain measurements while rejecting invalid values and outliers. The Extended Kalman Filter applies this prediction-and-update scheme after linearizing nonlinear models. Large linearization errors can make an EKF-based SLAM estimate inconsistent [9], [12].

Calibration determines sensor offsets, scale factors, relative poses, and time alignment. Errors in these parameters introduce systematic errors that filtering cannot remove as random noise. During estimation, uncertainty weighting assigns less influence to imprecise measurements, and outlier rejection removes observations that conflict with the remaining data [2].

GPS, motion capture, and lighthouse systems provide absolute or externally referenced pose corrections that bound drift. Indoor operation may exclude GPS, while experimental infrastructure may be unavailable outside a laboratory. External references can supply ground truth or occasional corrections under those conditions, but onboard estimation remains necessary between updates.

Loop closure constrains long-term drift by connecting the current pose to an earlier pose at the same location. Graph optimization then distributes the resulting correction across the trajectory and map. Unlike odometry alone, SLAM retains environmental observations that support these non-consecutive constraints [1], [2], [7].

H. Loop Closure

1) Keyframes and Pose Graphs: Processing every recorded pose as a graph node would increase the graph size without adding the same amount of new information. SLAM systems select keyframes at positions where they record a new scan or viewpoint. Each keyframe stores a pose

$$\mathbf{x}_i = \begin{bmatrix} x_i & y_i & \theta_i \end{bmatrix}^T.$$

An edge between two nodes stores the measured relative transformation

$$\mathbf{z}_{ij} = \begin{bmatrix} \Delta x_{ij} & \Delta y_{ij} & \Delta\theta_{ij} \end{bmatrix}^T.$$

Sequential edges connect poses that follow each other in time. A loop-closure edge connects two non-consecutive poses that refer to the same location. This edge exposes accumulated error because the sequential trajectory and the loop-closure measurement cannot both be satisfied without adjusting the poses [11].

A full pose-graph optimizer estimates the poses by minimizing the weighted errors of all graph edges:

$$\mathbf{X}^* = \arg\min_{\mathbf{X}} \sum_{(i,j)\in\mathcal{E}} \mathbf{e}_{ij}(\mathbf{X})^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}(\mathbf{X}),$$

where $\mathbf{e}_{ij}$ is the difference between the measured and predicted relative pose. The information matrix $\mathbf{\Omega}_{ij}$ determines how strongly an edge affects the result. Measurements with lower uncertainty receive a higher weight [11].

The prototype presented in this work does not solve this weighted nonlinear optimization problem. After a loop closure is accepted, the measured position and heading correction are distributed over the intermediate poses. The following sections refer to this method as simplified pose-graph correction because it applies the correction principle without optimizing all graph constraints.

2) Scan Matching and Loop-Closure Validation: Scan matching estimates the rigid transformation between two sets of range measurements. Given two scans $\mathcal{P}$ and $\mathcal{Q}$, the alignment searches for a rotation $\mathbf{R}$ and a translation $\mathbf{t}$ that minimize the mean squared distance between corresponding points. The correspondence set $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{Q}$ contains point pairs $(\mathbf{p}, \mathbf{q})$ that are treated as observations of the same surface:

$$E(\mathbf{R}, \mathbf{t}) = \frac{1}{|\mathcal{C}|} \sum_{(\mathbf{p},\mathbf{q})\in\mathcal{C}} \|\mathbf{p} - (\mathbf{R}\mathbf{q} + \mathbf{t})\|^2 .$$

A low alignment error does not prove that both scans were recorded at the same location. Similar walls, corners, or boxes can produce similar range patterns. Loop-closure detection separates candidate generation from geometric validation. Candidate generation selects scans that could represent the same place; validation checks their alignment, overlap, and consistency with the estimated trajectory. An incorrect constraint can deform the complete map, while a rejected constraint leaves the existing drift unchanged [11].

In the implemented prototype, the first and final keyframes form the loop-closure candidate because the autonomous flight returns to its starting area. The scans are aligned automatically, while the user performs the final acceptance.

Loop closure checks whether the current place has already been visited. In camera-based systems, this can be done by comparing the current image with older keyframes. With LiDAR systems, the current scan can be compared with older scans or with a map. In landmark-based systems, the robot can compare the landmarks it sees with landmarks it knows about.

When a valid loop closure is found, it creates a new constraint in the estimation problem. Without this constraint, the trajectory may slowly drift away from the true path. With this constraint, the estimator knows that two parts of the trajectory should meet. Optimizing the graph can spread the correction over the whole trajectory, instead of only changing the most recent pose [2], [5].

The estimated poses and map elements are connected by constraints. If the robot returns to the start but the estimated path does not close, the new loop-closure constraint adjusts the map and path to make them more consistent. The correction does not completely eliminate the error, but it can reduce it if the loop closure is correct.

An incorrect loop closure causes the optimizer to force unrelated parts of the map together. Candidate acceptance must include geometric agreement and consistency with the estimated trajectory. Uncertainty weighting and outlier-resistant loss functions reduce the influence of inconsistent constraints that pass this validation [2], [12].

I. Map-Based Path Planning

A map supports navigation by representing free, occupied and unknown space. Once the robot has a representation of its surroundings, it can plan a path to a goal. In an occupancy grid, path planning can be treated as graph search. Each free cell can be a node, and neighboring free cells can be connected by edges.

Dijkstra's algorithm computes shortest paths in graphs with non-negative edge costs. A* adds a heuristic estimate of the remaining cost to the goal. With an informative admissible heuristic, A* expands fewer nodes than Dijkstra's algorithm while retaining optimality.

When all free cells have the same cost, A* and Dijkstra often produce paths that pass very close to walls. Therefore, a cost heatmap is created, assigning a cost to each cell, with higher costs near obstacles. The algorithms then minimise the total cost, resulting in a safer path rather than the shortest path.

Unknown or partly mapped environments require the planner to assign a traversal policy to unobserved cells. A conservative planner treats them as blocked, whereas an exploration planner selects frontiers between known free space and unknown space. Small flying robots also require local obstacle avoidance because a global plan becomes unsafe when an obstacle appears after planning or is absent from the map. Live range measurements provide this local response while the map-based planner supplies the longer route.

For local planning and obstacle avoidance, the global path is converted into short-term motion commands. The Dynamic Window Approach (DWA) evaluates velocity commands that are feasible within the robot's dynamic limits, meaning its maximum speed, turning rate, acceleration, and braking capability, and selects the command that best balances progress toward the goal, obstacle clearance, and motion constraints. Model Predictive Control (MPC) instead solves a short-horizon constrained optimization problem, so robot dynamics, obstacle constraints, and reference-path tracking can be handled together. DWA is often simpler and more reactive, while MPC is more flexible but usually needs more computation [13], [14].

J. Evaluation and Validation

A SLAM system should be evaluated with clear criteria. It is not enough to show that a map looks reasonable. The

trajectory, the map, the navigation behavior, the computational effort, and robustness should be evaluated separately. Table III lists the different metrics and their interpretations.

Experiments should cover environments with different SLAM failure modes. A simple room works well for first tests, a messy room can make obstacle mapping difficult, and a corridor tests drift along a long direction and repeated wall geometry. An open space tests few geometric constraints, repeated structures test perceptual aliasing and false loop closure.

Ablation tests isolate the effect of each sensor by comparing IMU-only estimation, IMU with optical flow, IMU with optical flow and downward ToF, and the full sensor set. Reporting loop closure and offboard optimization both enabled and disabled separates their effects from those of the local estimator [2], [15].

When possible, the evaluation should use external ground truth. Motion capture or a high-quality reference map can provide independent measurements. If full ground truth is unavailable, partial references such as known wall distances, known start and end positions, or manually measured room dimensions can be used.

K. Crazyflie 2.1 Brushless as a SLAM Platform

The Crazyflie 2.1 Brushless was equipped with an IMU, a Flow Deck v2, and a Multi-ranger Deck. This creates a small indoor flight platform that can estimate motion and measure distances in several directions. The Flow Deck v2 provides downward optical flow and downward ToF distance. The Multi-ranger Deck provides ToF distances to the front, back, left, right, and up. Together these sensors give information about movement relative to the ground and about nearby obstacles.

The pose estimate can be described with a state vector such as

$$x_t = [p_t, \theta_t, v_t]^\top,$$

where p_t is position, θ_t is orientation, and v_t is velocity. IMU data support prediction and attitude estimation. Optical flow measures horizontal motion relative to the ground. Downward ToF provides height over the ground. These measurements can be fused in a estimator such as an EKF. The resulting state estimate gives a relative pose estimate for the drone.

The directional ToF sensors can update an occupancy map. Each range measurement can be treated as a ray. The cells along the ray are probably free, and the endpoint is likely occupied if the sensor detects an obstacle. They are especially useful for measuring walls and obstacles in the horizontal plane. The upward and downward directions can help identify where the ceiling and floor are. This matches a conservative 2D or limited 2.5D occupancy representation.

Figure 1 separates fast local estimation from slower map correction. IMU and optical flow support continuous state estimation, while sparse ToF rays update the occupancy map. That map supplies constraints and free-space information for path planning. Live ToF readings feed reactive obstacle avoidance when the map is incomplete or delayed.

L. Crazyflie-Specific Limitations and Constraints

The Crazyflie setup permits small-scale indoor SLAM experiments but limits the claims that the results can support. Table IV relates each hardware or sensing constraint to its effect on estimation and mapping.

These constraints restrict the platform to small indoor experiments, conservative obstacle mapping, and resource-aware SLAM research. Unlike a larger UAV with a 2D LiDAR, 3D LiDAR, RGB-D camera, or more powerful onboard computer, the Crazyflie records only six range beams. It supports relative localization and occupancy-based mapping under controlled conditions, but the current sensor configuration cannot produce dense reconstructions or enough place-recognition evidence for reliable global loop closure [2], [15].

M. Experimental Pose-Graph Mapping

A small pose-graph mapping prototype was implemented to test the Crazyflie sensor configuration. A pose graph represents selected drone poses as nodes. Edges store the measured motion between consecutive nodes. When two non-consecutive nodes represent the same physical location, an additional loop-closure edge can be added. This constraint is used to correct drift in the recorded trajectory [11].

1) Setup and Data Recording: The experiment used a Crazyflie 2.1 Brushless with a Flow Deck v2 and a Multi-ranger Deck. The Flow Deck provides optical-flow measurements and a downward distance measurement. The Multi-ranger Deck measures distances in the front, back, left, right, and upward directions [17]–[19].

The flight controller communicated with a laptop through a Crazyradio USB dongle. Flight control and data recording were implemented in Python with the Crazyflie Python library [20]. Position, velocity, attitude, and range measurements were stored in JSON files. The evaluation was performed offboard after the flight.

The test took place in an approximately $5\,\mathrm{m} \times 5\,\mathrm{m}$ room. A box with a footprint of approximately $1\,\mathrm{m} \times 1\,\mathrm{m}$ was placed in the center. The Crazyflie followed an autonomous square trajectory of approximately $2\,\mathrm{m} \times 2\,\mathrm{m}$ around the box. Figure 2 shows the test area and the flight sequence.

Before the flight, 100 measurements were recorded for each horizontal range sensor at a reference distance of $0.5\,\mathrm{m}$. The target flight height was $0.35\,\mathrm{m}$ and the movement speed was $0.35\,\mathrm{m/s}$.

A keyframe was created at the starting position, at each corner, and after returning to the start. This produced five pose nodes and four sequential edges. At each keyframe, the drone rotated by 360° at $25^\circ/\mathrm{s}$. Pose and range values were recorded throughout the rotation and during the movement between the keyframes.

Each horizontal range value was converted into a point in the world coordinate system:

$$\mathbf{p}_{i,s} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} + r_{i,s} \begin{bmatrix} \cos(\psi_i + \alpha_s) \\ \sin(\psi_i + \alpha_s) \end{bmatrix},$$

TABLE III: Evaluation criteria for SLAM systems.

Evaluation aspect	Example metric	Interpretation
Trajectory quality	Absolute trajectory error, relative pose error, return-to-start error	Shows whether the pose estimate is accurate locally and globally
Map quality	Agreement with reference map, obstacle-boundary error, free-space correctness	Shows whether the map is useful and not only visually plausible
Navigation quality	Goal success rate, collision rate, path length, replanning count	Shows whether the map and localization support safe motion
Efficiency	CPU load, update rate, memory use, communication load	Shows whether the method fits the available platform resources
Robustness	Failure rate across surfaces, lighting, geometry, and disturbance cases	Shows whether the system works beyond one easy test scenario

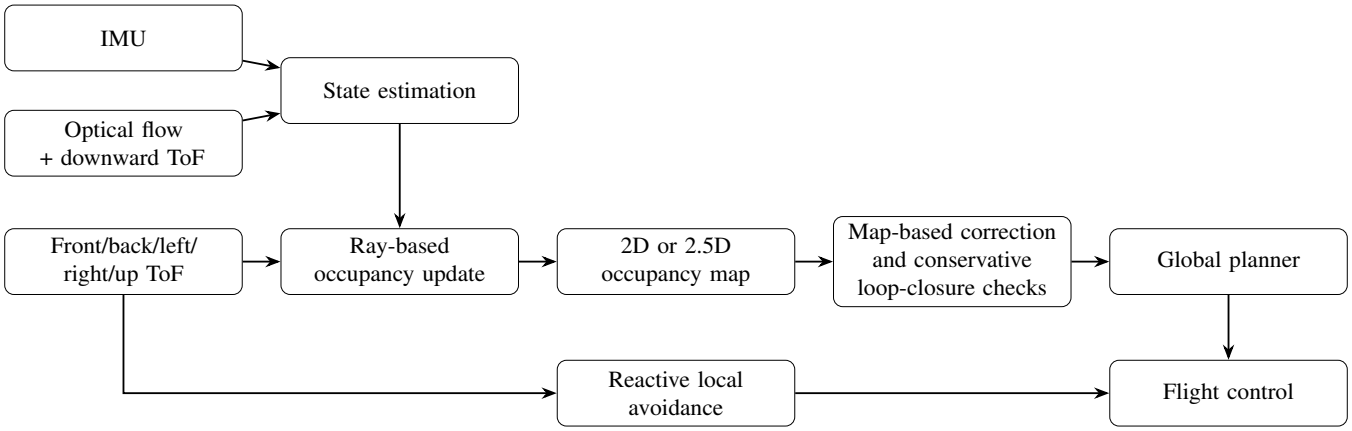


Fig. 1: Conceptual SLAM and navigation architecture for the Crazyflie platform with optical flow and sparse Time-of-Flight sensing.

TABLE IV: Crazyflie-specific SLAM limitations and implications.

Limitation or constraint	Implication
The sensing is sparse and directional.	Six ToF beams do not provide a dense scan or a rich image. The resulting map should represent occupancy rather than claim a dense reconstruction.
Optical flow depends on the ground.	Motion estimation quality depends on floor texture, lighting, height estimate, and motion quality.
ToF measurements depend on surface and range conditions.	Range values should be filtered and interpreted with uncertainty. A single beam gives only a narrow observation of the environment.
The platform has limited payload, computation, and flight time.	Onboard algorithms should be lightweight. Heavy graph optimization or dense 3D mapping is better treated as optional or offboard [15], [16].
Loop-closure evidence is weak compared with LiDAR or camera SLAM.	Loop closure should be conservative. False loop closures can deform the map, and sparse range signatures can be ambiguous [2], [7].
Open or repetitive environments provide few unique constraints.	Similar corridors, symmetric rooms, and large open spaces can make localization ambiguous and increase drift [2], [12].
The most suitable map is limited in detail.	A 2D occupancy grid or limited 2.5D map is better aligned with the sensor geometry than a full dense 3D reconstruction [8], [15].

where (x_i, y_i, ψ_i) denotes the estimated drone pose at time step i, $r_{i,s}$ is the range measured by sensor s, and α_s is the mounting angle of sensor s relative to the drone body frame.

2) Measurement Processing: The recorded data was corrected before comparing the keyframes. The median error measured during sensor calibration was subtracted from each horizontal range value. Measurements with a roll or pitch angle above $8°$, or a velocity above $0.65\,\mathrm{m/s}$, were excluded from the corrected result.

Roll and pitch were used to correct the horizontal component of the range measurements. Yaw values were smoothed with a moving average over five samples. During each keyframe scan, the recorded positions were referenced to the position at the start of the scan. This reduces point dispersion caused by changes in the position estimate during the rotation. It also assumes that the Crazyflie should rotate at a fixed position.

3) Scan Matching and Loop Closure: The first and final keyframes were selected as a loop-closure candidate. Both scans were converted into local coordinates relative to their keyframe poses.

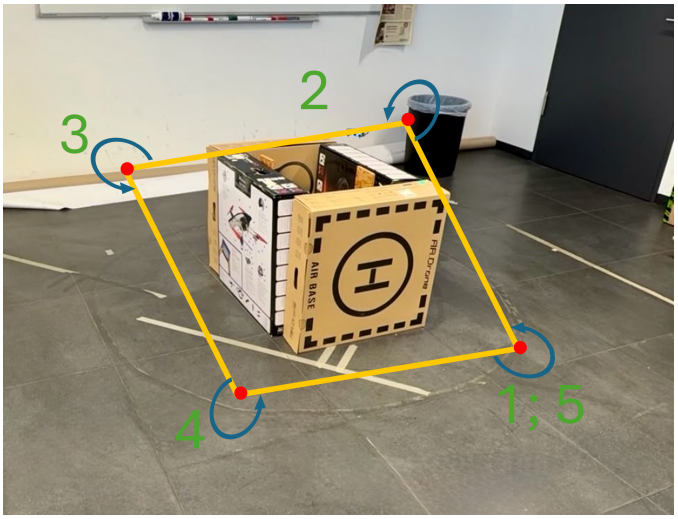


Fig. 2: Test environment and autonomous flight sequence. The numbered positions mark the five keyframes. The first and fifth keyframes were recorded at the same physical location. A 360° scan was performed at every keyframe.

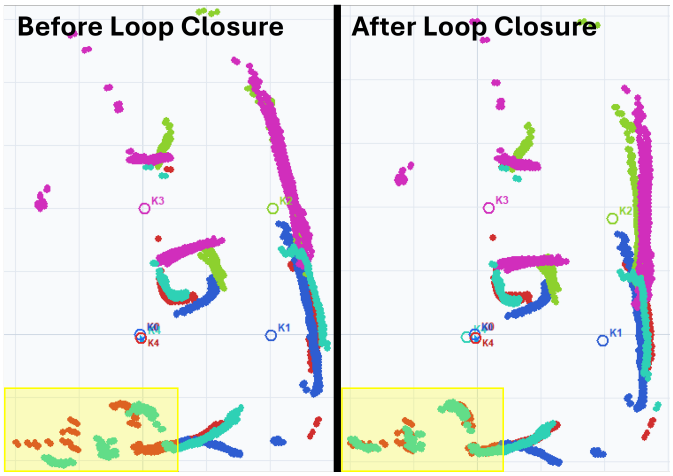


Fig. 3: Projected Multi-ranger measurements before and after preprocessing and the accepted loop-closure correction. The corrected result aligns the final keyframe with the initial keyframe and distributes the correction over the recorded trajectory.

The alignment first searches for the relative rotation. The second scan is tested in steps of 5° between −180° and 180°. The best result is then checked again in steps of 1°. The initial translation is estimated from the difference between the centers of both point sets.

For every candidate transformation, each point in the second scan is assigned to its nearest point in the first scan. Associations with a distance above 0.35 m are discarded. The mean distance of the best 80 percent of the remaining associations is used as the matching score.

The program displays both scans as an overlay. The calculated rotation and translation can be adjusted before the user accepts or rejects the loop closure. Manual confirmation was kept because the four horizontal range sensors can produce similar measurements at different positions.

After confirmation, the final keyframe is aligned with the initial keyframe. The position and heading correction are distributed over the intermediate keyframes and recorded samples according to their position in the flight sequence. This follows the rubber-band interpretation of pose graphs, but it is not a nonlinear least-squares optimization of the complete graph.

4) Recorded Result: The evaluated run contained five keyframes and 1,900 recorded state samples. Of these, 533 samples were recorded during the movement between keyframes. According to the Crazyflie state estimate, the final keyframe was approximately 5.3 cm from the initial keyframe. The difference in yaw was approximately 1.1°.

The Crazyflie did not remain at a fixed position during the keyframe scans. The largest estimated horizontal displacement during the individual 360° rotations ranged from approximately 3.8 cm to 5.6 cm. These movements spread measurements that should belong to a single scan position.

The result shows that the recorded measurements can represent the shape of the test object and that the first and final scans can be aligned. The map still contains scattered points caused by hover movement, missing range values, narrow sensor beams, and errors in the estimated pose.

The accepted loop-closure candidate contained 424 point associations, with a mean nearest-neighbor distance of 0.015m. The applied correction was 0.141m and −8.7° see Fig. 3.

No external tracking system was available, so the reported position errors refer to the Crazyflie state estimate rather than ground-truth errors. Onboard mapping was also not tested. The current implementation records and processes all data on the laptop. A small two-dimensional occupancy grid is a possible next step, but its memory use and update rate still need to be measured.

N. Conclusion

SLAM is the combined problem of estimating robot pose and building a map. It is needed whenever a robot must move autonomously in an unknown or partly known environment. A typical SLAM system collects sensor data, preprocesses it, extracts useful information, estimates pose, updates the map, refines recent estimates, detects loop closures, and uses the resulting map for planning.

Different sensors and map representations lead to different forms of SLAM. Odometry and IMU-based estimates are fast but drift over time. Cameras, LiDAR, ToF, and other range sensors can observe the environment and reduce drift, but they have their own noise and failure cases. Occupancy grids are simple and support navigation, while dense 3D maps need more sensor coverage and computation.

For the Crazyflie 2.1 Brushless with Flow Deck v2 and Multi-ranger Deck, the available payload and sensor coverage favor a lightweight SLAM pipeline. IMU, optical flow, and

ToF data supply pose updates, while sparse ToF rays update an occupancy grid used for planning and reactive avoidance. Four horizontal range beams cannot provide the geometric detail available from a dense LiDAR or camera system. Evaluation must account for the resulting drift, ambiguous range patterns, and weak loop-closure evidence.

AI disclosure

Markus and Richard, the authors of this section, used ChatGPT (OpenAI) to assist with drafting, revising and preparing selected text passages, LaTeX code and source references in the SLAM section. All AI-assisted content was reviewed, verified and revised by the authors, who take full responsibility for its accuracy, originality and quality. This disclosure only applies to the SLAM section.

II. Kalman Filter

Authors: Steffen Krutzsch, Christian Model

A. Introduction

Autonomous flight requires a drone to continuously know its own state, meaning its position, velocity, and orientation, with high accuracy and low latency. However, onboard sensors such as accelerometers, gyroscopes, and range finders are inherently noisy and subject to drift, making direct measurements unreliable on their own. A naive approach of simply trusting raw sensor data leads to rapid accumulation of errors, rendering stable autonomous flight impossible. The Kalman filter addresses this fundamental problem by fusing a mathematical model of the system's dynamics with incoming sensor measurements, producing a statistically optimal state estimate under the assumption of Gaussian noise [21]. Originally derived for linear systems, the filter has since been extended to handle the non-linear dynamics that are inherent to real-world platforms such as quadrotors. These non-linear variants, most notably the Extended Kalman Filter (EKF) and the Unscented Kalman Filter (UKF), form the foundation of modern onboard state estimation. This section introduces the theoretical foundations of the Kalman filter, motivates the need for its non-linear extensions, and examines their concrete implementation on the Crazyflie 2.1 Brushless platform, where the choice of estimator directly depends on the attached sensor configuration.

B. Historical Context

The Kalman filter was introduced in 1960 by Rudolf E. Kalman in his seminal paper *A New Approach to Linear Filtering and Prediction Problems* [21], published in the ASME Journal of Basic Engineering. Although the theoretical contribution was immediately recognised as significant, its practical potential became apparent only when NASA researchers began searching for a solution to one of the most demanding navigation problems of the era, namely guiding a crewed spacecraft to the Moon and back.

At NASA's Ames Research Center, engineers faced significant challenges in navigation for the circumlunar mission and lacked suitable estimation tools. Stanley Schmidt, chief of the Dynamic Analysis Branch at Ames, identified Kalman's algorithm as a candidate solution and recognised its immense potential for spaceflight. The challenge was considerable: the spacecraft's trajectory was sensitive to a multitude of disturbances, and the Moon itself was a moving target requiring continuous trajectory correction over several days. Furthermore, the engineers had to calculate precise navigation data in real-time despite the extremely limited resources of the Apollo guidance computer, which only possessed 4 KB of RAM. The Kalman filter was particularly well-suited for this constraint due to its recursive, resource-efficient nature.

Because the classic 1960 Kalman filter functions exclusively for linear systems—where physical relationships are purely proportional—it was insufficient for the real world, as spaceflight is inherently non-linear. Consequently, Stanley Schmidt and his engineering team at the Ames Research Center developed the Extended Kalman Filter (EKF) [22] to address these non-linearities. This adaptation allowed the filter to effectively process the noisy sensor data, filtering out the noise to produce a continuously improved estimate of the actual flight path, which made the lunar landing possible.

The Extended Kalman filter that was deployed on Apollo 11 worked fast, continuously generating new best estimates based on the most recent previous state without requiring large amounts of memory. The success of the Apollo programme demonstrated that optimal state estimation under uncertainty was not merely a theoretical concept but an indispensable engineering tool [23]. It has since found applications in GPS receivers, autonomous vehicles, weather forecasting, and, as discussed in this paper, miniature autonomous quadrotors.

C. Mathematical Preliminaries

Three mathematical concepts recur throughout the derivation of the Kalman filter and its non-linear extensions. They are briefly reviewed here.

Gaussian Distribution: A scalar random variable x is Gaussian (normally distributed) with mean μ and variance σ^2, written $x \sim \mathcal{N}(\mu, \sigma^2)$, if its probability density is given by

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{1}$$

A Gaussian distribution is completely described by these two parameters. This property is the reason the Kalman filter only needs to track a mean vector and a covariance matrix instead of a full probability distribution.

Covariance Matrix: For a random vector $\mathbf{x} \in \mathbb{R}^n$ with mean $\boldsymbol{\mu}$, the covariance matrix generalises the scalar variance and is defined as

$$\boldsymbol{\Sigma} = \mathrm{E}\big[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top\big] \tag{2}$$

Its diagonal entries Σ_{ii} are the variances of the individual components, and its off-diagonal entries Σ_{ij} describe how strongly components i and j vary together. Every covariance matrix is symmetric and positive semi-definite, a property that the filter equations must preserve at every step.

Jacobian Matrix: For a differentiable vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix collects all first-order partial derivatives,

$$\mathbf{J} = \frac{\partial f}{\partial \mathbf{x}}, \qquad J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{3}$$

The Jacobian provides the best linear approximation of f around a point $\mathbf{x}_0$ in the sense of a first-order Taylor expansion, $f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{J}(\mathbf{x} - \mathbf{x}_0)$. The Extended Kalman Filter in Section II-F1 relies on exactly this approximation.

D. Core Idea: Optimal State Estimation

The Kalman filter solves a fundamental problem: estimating a system's state (position, velocity) from two imperfect information sources. A mathematical model predicts the next state based on physics, but it is never perfect due to unmodelled disturbances. A sensor measures the true state, but its readings are noisy. Simply trusting one source alone is suboptimal.

The elegant solution: *fuse both optimally*, weighting each according to its current reliability. Both the model prediction and sensor measurement can be represented as Gaussian distributions with a mean (best estimate) and variance (uncertainty). The Kalman filter computes their intersection, producing a fused estimate that lies between the two sources, weighted inversely by their variances. Crucially, the fused estimate's uncertainty is always smaller than either source alone [24].

Figure 4 visualises this: the model prediction (blue Gaussian) and sensor measurement (red Gaussian) are fused into an optimal estimate (green Gaussian) with lower uncertainty. This process repeats recursively, with minimal memory overhead, making the algorithm both elegant and computationally efficient.

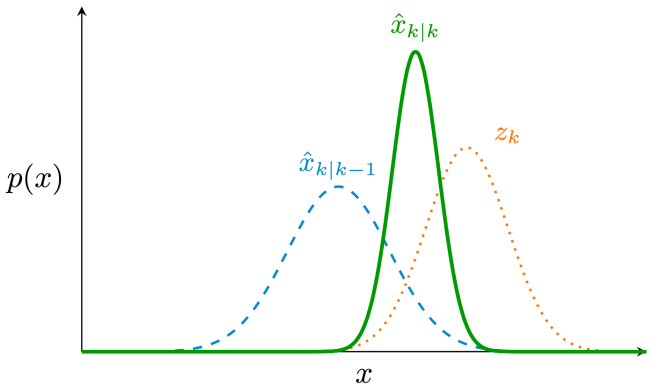


Fig. 4: Intuitive illustration of the Kalman filter using Gaussian distributions. The model prediction (blue) and sensor measurement (red) are both represented as normal distributions, each with a mean and variance. The Kalman filter computes their optimal fusion (green), which has a tighter variance and a mean that lies between the two sources, weighted by their relative uncertainty. This visual representation underscores that the filter is fundamentally an optimal combination of imperfect information sources.

The Kalman filter is a recursive algorithm that produces optimal state estimates for linear systems subject to Gaussian noise [25]–[27]. It operates in two alternating phases, *prediction* and *update*, shown in Figure 5, which together allow the filter to continuously refine its estimate as new measurements arrive, without storing the full history of observations. The filter's practical value lies in its computational efficiency, since at each time step only the most recent estimate and its associated uncertainty need to be retained [26].

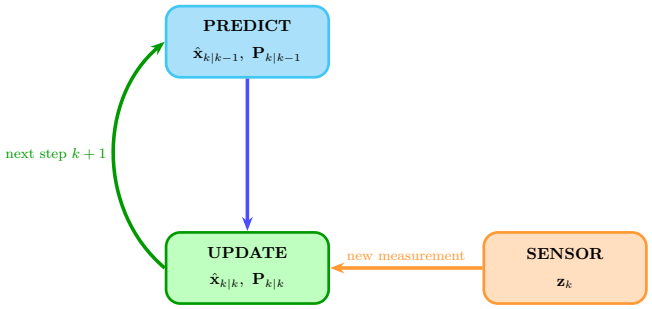


Fig. 5: The recursive predict-and-update cycle of the Kalman filter. At each time step the prior estimate is propagated forward via the process model and subsequently corrected by the incoming measurement.

E. Mathematical Formulation of the Linear Kalman Filter

The filter assumes a linear, discrete-time system governed by two equations: a process model (state evolution) and a measurement model (sensor observations).

The **process model** describes how the system state evolves over time according to

$$\mathbf{x}_k = \mathbf{F}\,\mathbf{x}_{k-1} + \mathbf{B}\,\mathbf{u}_k + \mathbf{w}_k, \qquad \mathbf{w}_k \sim \mathcal{N}(\mathbf{0},\, \mathbf{Q}) \tag{4}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the state vector at time step k, $\mathbf{F} \in \mathbb{R}^{n \times n}$ is the state transition matrix encoding the system dynamics, $\mathbf{B}$ maps the control input $\mathbf{u}_k$ onto the state space, and $\mathbf{w}_k$ is zero-mean Gaussian process noise with covariance $\mathbf{Q} \in \mathbb{R}^{n \times n}$.

As an illustrative example, consider a state vector simplified to position and velocity ($n = 6$), deliberately omitting the orientation angles ϕ, θ, ψ, so that

$$\mathbf{x} = \begin{bmatrix} p_x & p_y & p_z & v_x & v_y & v_z \end{bmatrix}^\top \tag{5}$$

This simplification keeps the measurement functions linear and allows the standard Kalman filter to be applied without modification. On the Crazyflie a fuller state vector additionally includes the orientation angles roll, pitch, and yaw, whose measurement functions are inherently non-linear. The concrete composition and dimension of this state vector depend on the attached deck configuration and the selected estimator, and such orientation-dependent measurements necessitate the Extended Kalman Filter discussed in Section II-F1.

The **measurement model** relates the true state to the sensor output through

$$\mathbf{z}_k = \mathbf{H}\,\mathbf{x}_k + \mathbf{v}_k, \qquad \mathbf{v}_k \sim \mathcal{N}(\mathbf{0},\, \mathbf{R}) \tag{6}$$

where $\mathbf{z}_k \in \mathbb{R}^p$ is the measurement vector, $\mathbf{H} \in \mathbb{R}^{p\times n}$ is the observation matrix, and $\mathbf{v}_k$ is zero-mean Gaussian measurement noise with covariance $\mathbf{R} \in \mathbb{R}^{p\times p}$. The observation matrix $\mathbf{H}$ acts as a bridge between the two spaces. Left-multiplication by $\mathbf{H}$ projects an n-dimensional state vector into the p-dimensional measurement space, enabling the difference $\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_k^-$ to be computed. For a sensor that measures only height, for example, $\mathbf{H} = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0\end{bmatrix}$ selects p_z from the state vector. Both noise processes are assumed to be mutually uncorrelated and independent across time steps. The covariance matrices $\mathbf{Q}$ and $\mathbf{R}$ are design parameters that encode the engineer's relative trust in the model versus the sensors [24].

Table V summarises the dimensions of every quantity appearing in the filter equations and the space in which it lives.

TABLE V: Dimensions and spaces of all filter quantities. n denotes the state dimension and p the measurement dimension. On the Crazyflie both n and p vary with the deck configuration and the active sensor.

Symbol	Dimension	Space
$\mathbf{z}_k$	$p \times 1$	measurement
$\hat{\mathbf{x}}_{k\|k-1}$	$n \times 1$	state
$\mathbf{H}$	$p \times n$	bridge: state $\rightarrow$ meas.
$\mathbf{P}_{k\|k-1}$	$n \times n$	state
$\mathbf{R}$	$p \times p$	measurement
$\mathbf{H}\mathbf{P}\mathbf{H}^\top$	$p \times p$	measurement
$\mathbf{K}_k$	$n \times p$	bridge: meas. $\rightarrow$ state
$\tilde{y}_k$	$p \times 1$	measurement
$\mathbf{K}_k\tilde{y}_k$	$n \times 1$	state
$\mathbf{I} - \mathbf{K}_k\mathbf{H}$	$n \times n$	state

Prediction Step: Given the previous estimate $\hat{\mathbf{x}}_{k-1}$ and covariance $\mathbf{P}_{k-1}$, the filter propagates both quantities forward in time using the system model,

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}\,\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}\,\mathbf{u}_k \tag{7}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}\,\mathbf{P}_{k-1|k-1}\,\mathbf{F}^\top + \mathbf{Q} \tag{8}$$

The subscript notation $(\cdot)_{k|k-1}$ denotes an *a priori* quantity, that is an estimate formed at time k before incorporating the measurement at time k. The covariance update in Equation (8) takes the form $\mathbf{F}\mathbf{P}\mathbf{F}^\top$ rather than simply $\mathbf{F}\mathbf{P}$ because covariance matrices are quadratic forms and must remain symmetric after transformation. Left-multiplying by $\mathbf{F}$ and right-multiplying by $\mathbf{F}^\top$ is the matrix analogue of scaling a scalar variance by $a^2 = a \cdot a$. The additive term $\mathbf{Q}$ accounts for process noise such as wind gusts, motor imbalances, and modelling inaccuracies. As a consequence, $\mathbf{P}_{k|k-1}$ grows at every prediction step, reflecting the increasing uncertainty that accumulates in the absence of a corrective measurement.

Update Step: Once the new measurement $\mathbf{z}_k$ is received, the filter computes the **innovation**

$$\tilde{y}_k = \mathbf{z}_k - \mathbf{H}\,\hat{\mathbf{x}}_{k|k-1} \tag{9}$$

The term *innovation* is deliberate, because it quantifies the information genuinely new to the filter, namely the discrepancy between what the sensor measured and what the model predicted. If the drone were exactly where the model expected, the innovation would be zero and the measurement would contribute nothing.

The **Kalman gain** is then computed as

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\,\mathbf{H}^\top\Big(\mathbf{H}\,\mathbf{P}_{k|k-1}\,\mathbf{H}^\top + \mathbf{R}\Big)^{-1} \tag{10}$$

The term $\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^\top$ projects the state-space covariance into the measurement space using the same two-sided transformation as in Equation (8), so that it can be added directly to $\mathbf{R}$, which also lives in the measurement space (see Table V). The resulting sum is the total uncertainty of the innovation. The structure of the gain is easiest to understand in the scalar case ($n = p = 1$, $H = 1$), where Equation (10) reduces to

$$K = \frac{P_{k|k-1}}{P_{k|k-1} + R} = \frac{\text{model uncertainty}}{\text{total uncertainty}} \tag{11}$$

Because both $P_{k|k-1}$ and R are non-negative, the scalar gain always lies in the closed interval $[0,\, 1]$, that is, between zero and one inclusive. When R is large relative to $P_{k|k-1}$, the sensor is noisy and K approaches zero. The correction term $K\tilde{y}_k$ then vanishes and the estimate stays close to the model prediction. When $P_{k|k-1}$ is large relative to R, the model is uncertain and K approaches one. The corrected estimate then moves almost entirely to the sensor reading [24]. In the multidimensional case this intuition carries over, but the upper limit is no longer the number one or the identity matrix. As the prior covariance grows without bound, the gain $\mathbf{K}_k$ approaches the Moore–Penrose pseudoinverse $\mathbf{H}^+$ of the observation matrix, so that the updated estimate reproduces the measurement exactly in the observed subspace while leaving unobserved state directions unchanged.

The state estimate and covariance are then corrected according to

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k\,\tilde{y}_k \tag{12}$$

$$\mathbf{P}_{k|k} = \big(\mathbf{I} - \mathbf{K}_k\,\mathbf{H}\big)\,\mathbf{P}_{k|k-1} \tag{13}$$

The factor $(\mathbf{I} - \mathbf{K}_k\mathbf{H})$ in Equation (13) is an $n \times n$ matrix, since $\mathbf{K}_k$ is $n \times p$ and $\mathbf{H}$ is $p \times n$, so that their product is $n \times n$. It represents the fraction of the prior uncertainty that survives after the measurement. The *a posteriori* covariance $\mathbf{P}_{k|k}$ is strictly smaller than $\mathbf{P}_{k|k-1}$, confirming that each new measurement reduces uncertainty [26].

Figure 5 summarises the predict-update cycle, and Figure 6 demonstrates the filter on a one-dimensional height-estimation example.

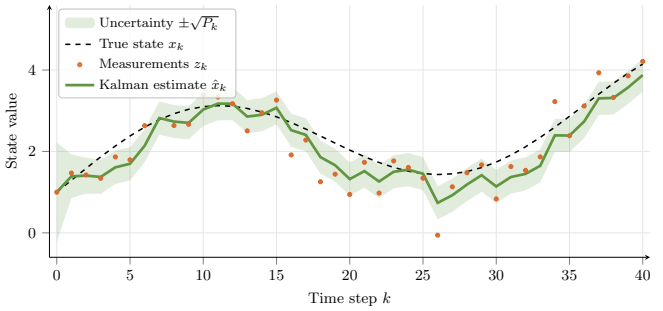


Fig. 6: Illustrative example of the Kalman filter estimating a one-dimensional state from noisy measurements. The Kalman estimate (green) tracks the true state while smoothing the noisy measurements (orange). The shaded band indicates the one-sigma uncertainty $\pm\sqrt{P_k}$ around the estimate, which is initially large and shrinks as measurements accumulate.

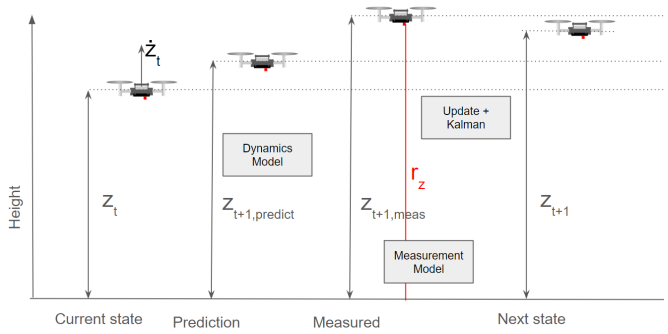


Fig. 7: One-dimensional height-estimation example for the Crazyflie, shown across four phases. Starting from the current state, the model produces a prediction, the ToF sensor delivers the noisy measurement z_t with noise r_z, and both are fused into the corrected next state. Adapted from [28].

Numerical Example: One-Dimensional Height Estimation: To make the filter equations concrete, we trace a single predict-and-update cycle for a Crazyflie hovering over a flat surface. Figure 7 illustrates the four phases of this cycle, which the following calculation works through numerically. The simplified two-dimensional state vector $\mathbf{x} = [z,\, \dot{z}]^\top$ collects height and vertical velocity. The ToF sensor on the Flow Deck measures height only, giving $\mathbf{H} = \begin{bmatrix}1 & 0\end{bmatrix}$ and scalar measurement noise $R = 0.01\,\mathrm{m}^2$. The unit of R is square metres because R is a variance, and the unit of a variance is always the square of the unit of the underlying quantity. A ToF sensor with a standard deviation of $\sigma_z = 0.1\,\mathrm{m}$ therefore corresponds to $R = \sigma_z^2 = 0.01\,\mathrm{m}^2$.

Initial conditions.

$$\hat{\mathbf{x}}_{t|t} = \begin{bmatrix}1.00\\0.20\end{bmatrix}\ \mathrm{m,\ m/s}, \qquad \mathbf{P}_{t|t} = \begin{bmatrix}0.04 & 0.00\\0.00 & 0.01\end{bmatrix},$$

$$\mathbf{F} = \begin{bmatrix}1 & \Delta t\\0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0.1\\0 & 1\end{bmatrix}, \qquad \mathbf{Q} = \begin{bmatrix}0.001 & 0\\0 & 0.001\end{bmatrix}$$

Predict step. The state prediction multiplies out to

$$\hat{\mathbf{x}}_{t+1|t} = \mathbf{F}\,\hat{\mathbf{x}}_{t|t} = \begin{bmatrix}1 & 0.1\\0 & 1\end{bmatrix}\begin{bmatrix}1.00\\0.20\end{bmatrix} = \begin{bmatrix}1\cdot 1.00 + 0.1\cdot 0.20\\0\cdot 1.00 + 1\cdot 0.20\end{bmatrix} = \begin{bmatrix}1.02\\0.20\end{bmatrix}$$

The covariance prediction is carried out in three stages. First the left multiplication,

$$\mathbf{F}\,\mathbf{P}_{t|t} = \begin{bmatrix}1 & 0.1\\0 & 1\end{bmatrix}\begin{bmatrix}0.04 & 0\\0 & 0.01\end{bmatrix} = \begin{bmatrix}0.04 & 0.001\\0 & 0.01\end{bmatrix}$$

then the right multiplication by $\mathbf{F}^\top$,

$$\mathbf{F}\,\mathbf{P}_{t|t}\,\mathbf{F}^\top = \begin{bmatrix}0.04 & 0.001\\0 & 0.01\end{bmatrix}\begin{bmatrix}1 & 0\\0.1 & 1\end{bmatrix} = \begin{bmatrix}0.0401 & 0.001\\0.001 & 0.01\end{bmatrix}$$

and finally the addition of the process noise,

$$\mathbf{P}_{t+1|t} = \mathbf{F}\,\mathbf{P}_{t|t}\,\mathbf{F}^\top + \mathbf{Q} = \begin{bmatrix}0.0411 & 0.0010\\0.0010 & 0.0110\end{bmatrix}$$

The height variance $\sigma_z^2 = \mathbf{P}_{1,1}$ grew from 0.04 to 0.0411, reflecting the uncertainty that accumulates without a corrective measurement.

Update step (sensor reading $z_{\mathrm{meas}} = 1.10\,\mathrm{m}$). The innovation compares the measurement with the predicted height,

$$\tilde{y} = z_{\mathrm{meas}} - \mathbf{H}\,\hat{\mathbf{x}}_{t+1|t} = 1.10 - 1.02 = 0.08\,\mathrm{m}$$

For the Kalman gain, the numerator and the innovation covariance are computed separately. The numerator projects the covariance onto the measured state,

$$\mathbf{P}_{t+1|t}\,\mathbf{H}^\top = \begin{bmatrix}0.0411 & 0.0010\\0.0010 & 0.0110\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0.0411\\0.0010\end{bmatrix}$$

and the innovation covariance is the scalar

$$\mathbf{H}\,\mathbf{P}_{t+1|t}\,\mathbf{H}^\top + R = 0.0411 + 0.01 = 0.0511$$

so that the gain becomes

$$\mathbf{K} = \begin{bmatrix}0.0411\\0.0010\end{bmatrix}\cdot\frac{1}{0.0511} = \begin{bmatrix}0.804\\0.020\end{bmatrix}$$

The state correction adds the weighted innovation to the prediction,

$$\hat{\mathbf{x}}_{t+1|t+1} = \hat{\mathbf{x}}_{t+1|t} + \mathbf{K}\,\tilde{y} = \begin{bmatrix}1.02\\0.20\end{bmatrix} + \begin{bmatrix}0.804\\0.020\end{bmatrix} \cdot 0.08$$
$$= \begin{bmatrix}1.02 + 0.0643\\0.20 + 0.0016\end{bmatrix} = \begin{bmatrix}1.084\\0.202\end{bmatrix}$$

For the covariance correction, the product $\mathbf{KH}$ and the remaining fraction $\mathbf{I} - \mathbf{KH}$ are formed first,

$$\mathbf{KH} = \begin{bmatrix}0.804\\0.020\end{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix} = \begin{bmatrix}0.804 & 0\\0.020 & 0\end{bmatrix},$$
$$\mathbf{I} - \mathbf{KH} = \begin{bmatrix}0.196 & 0\\-0.020 & 1\end{bmatrix}$$

and the posterior covariance follows as

$$\mathbf{P}_{t+1|t+1} = (\mathbf{I} - \mathbf{KH})\,\mathbf{P}_{t+1|t} = \begin{bmatrix}0.0081 & 0.0002\\0.0002 & 0.0110\end{bmatrix}$$

The Kalman gain $K_z = 0.804$ indicates that the filter trusts the ToF sensor strongly, which is consistent with its low measurement noise $R = 0.01\,\mathrm{m}^2$ relative to the predicted variance $\sigma_z^2 = 0.0411\,\mathrm{m}^2$. The corrected height $1.084\,\mathrm{m}$ lies closer to the measurement than to the prediction, and the height variance drops sharply from 0.0411 to $0.0081\,\mathrm{m}^2$, confirming that the update step reduces uncertainty.

F. Non-linear Extensions

The linear Kalman filter relies on two fundamental assumptions, namely that the process model $\mathbf{F}$ and observation model $\mathbf{H}$ are both linear, and that all noise is Gaussian. For many real-world systems these assumptions are violated. A quadrotor such as the Crazyflie is a prime example. Its orientation is naturally described by Euler angles, which enter the dynamics non-linearly via trigonometric functions such as sine and cosine (e.g., in the rotation matrices), and the relationship between height and an optical-flow measurement is governed by a division, which is an inherently non-linear operation [29].

Figures 8 and 9 illustrate the consequence. A Gaussian input distribution pushed through a linear function remains Gaussian, as shown in Figure 8. Pushed through a non-linear function, it becomes skewed and non-Gaussian, as shown in Figure 9. Applying the standard Kalman filter to such systems would require linearising the models globally, introducing errors that accumulate over time and can destabilise the estimator.

Two extensions have been developed to address this limitation. The *Extended Kalman Filter* linearises the system locally at each time step, while the *Unscented Kalman Filter* avoids linearisation altogether by propagating a small set of carefully chosen sample points through the exact non-linear functions [29], [31].

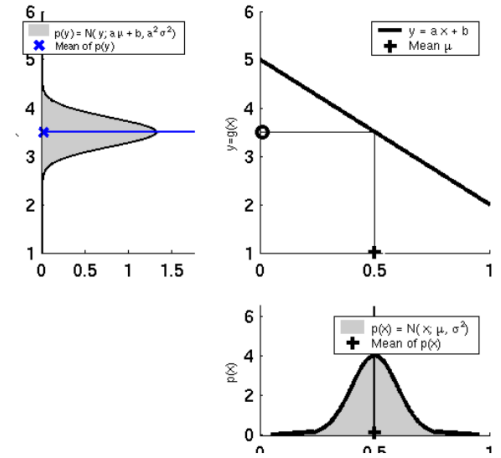


Fig. 8: Linear transformation of a Gaussian. An input density $p(x)$ passed through $g(x) = ax + b$ yields a Gaussian output with shifted mean and scaled variance. Adapted from [30].

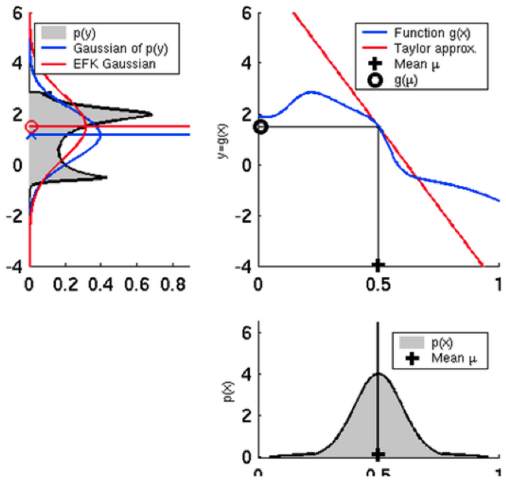


Fig. 9: Non-linear transformation of a Gaussian. The same input passed through a non-linear $g(x)$ yields a skewed, non-Gaussian output (black). The EKF linearises g at μ via its Jacobian (red), producing a Gaussian approximation that diverges from the true distribution. Adapted from [30].

1) Extended Kalman Filter (EKF): The EKF generalises the linear Kalman filter to non-linear systems by replacing the constant matrices $\mathbf{F}$ and $\mathbf{H}$ with non-linear functions f and h, and linearising these functions around the current state estimate at every time step [27], [29]. The system model becomes

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k, \qquad \mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{14}$$

The EKF computes the Jacobian matrices (Equation (3)) of f and h evaluated at the current estimate,

$$\mathbf{F}_k = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k-1|k-1}}, \qquad \mathbf{H}_k = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k|k-1}} \tag{15}$$

These Jacobians replace $\mathbf{F}$ and $\mathbf{H}$ in the covariance equations, while the state propagation itself uses the original non-linear functions. The full EKF equations are

$$\hat{\mathbf{x}}_{k|k-1} = f(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k) \tag{16}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \, \mathbf{P}_{k-1|k-1} \, \mathbf{F}_k^\top + \mathbf{Q} \tag{17}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \, \mathbf{H}_k^\top \big(\mathbf{H}_k \, \mathbf{P}_{k|k-1} \, \mathbf{H}_k^\top + \mathbf{R}\big)^{-1} \tag{18}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \big(\mathbf{z}_k - h(\hat{\mathbf{x}}_{k|k-1})\big) \tag{19}$$

$$\mathbf{P}_{k|k} = \big(\mathbf{I} - \mathbf{K}_k \, \mathbf{H}_k\big) \, \mathbf{P}_{k|k-1} \tag{20}$$

A key subtlety is that the innovation in Equation (19) uses the true non-linear function $h(\hat{\mathbf{x}}_{k|k-1})$ rather than the linearised $\mathbf{H}_k \hat{\mathbf{x}}_{k|k-1}$, which improves accuracy. Only the covariance updates rely on the linearised Jacobians [27], [32].

On the Crazyflie, the optical-flow measurement model does not depend linearly on the state. Instead, it combines the observed image motion with the current flight height, so that the same pixel displacement corresponds to a different physical velocity depending on altitude. In addition, rotational motion of the drone contributes to the measurement and must be compensated for in the model. As a result, the flow observation is inherently non-linear in the state variables, which is exactly why the EKF is required when the Flow Deck is attached [28], [33].

The ToF height model is also non-linear, because the measured distance depends on the orientation angles through trigonometric terms such as cosine [34]. In practice, these sensor models are sufficiently coupled to the state that the standard linear Kalman filter is no longer adequate.

The EKF's main limitation is that the Jacobian-based linearisation introduces approximation errors when the non-linearities are strong or when the uncertainty is large, as the first-order Taylor expansion may poorly represent the true propagated distribution [29], [32].

2) Unscented Kalman Filter (UKF): The UKF addresses the core limitation of the EKF. Rather than linearising the system via Jacobians, it approximates the state distribution itself [29], [31]. The underlying principle, stated by Julier and Uhlmann, is that it is easier to approximate a probability distribution than an arbitrary non-linear function [31].

In the prediction step, the filter deterministically places a small set of $2n+1$ sample points, called *sigma points*, around the current mean. One point sits exactly on the mean, and the remaining $2n$ points are placed symmetrically along the principal axes of the covariance ellipse, so that the point set as a whole reproduces the mean and the covariance of the prior distribution. For a Crazyflie full-pose state with $n = 9$, this yields 19 sigma points per step. Every sigma point is then propagated individually through the exact non-linear process function f, without any linearisation. The predicted mean and covariance are finally recovered as weighted averages over the transformed points. The update step follows the same structure as in the linear filter, with the required covariances likewise estimated from the transformed sigma points. At no point is a Jacobian computed. The non-linear functions f and h are treated as black boxes and simply evaluated at each sigma point [29].

Figure 10 illustrates the procedure, showing how the weighted sigma points recover the true mean and spread of the transformed distribution, whereas an EKF-style linearisation captures neither.

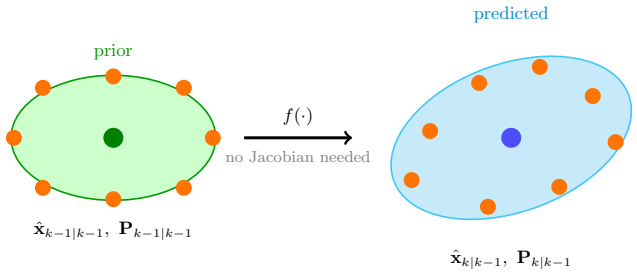


Fig. 10: Illustration of the unscented transform in a single plot. Sigma points (orange) are placed deterministically around the prior mean according to the prior covariance $\mathbf{P}_{k-1}$ (green ellipse). After propagation through the non-linear process function, indicated by the central arrow, the weighted sigma points reconstruct the predicted distribution (cyan ellipse), capturing the shifted mean and the rotated, enlarged spread that a linearisation-based approach would miss.

3) Comparative Summary: Table VI summarises the key characteristics of each estimator.

TABLE VI: Comparison of Kalman filter variants.

	Linear KF	**EKF**	**UKF**
System model	Linear	Non-linear	Non-linear
Noise assumption	Gaussian	Gaussian	Gaussian
Linearisation	Exact	Jacobian	None
Accuracy (non-lin.)	Fails	1st order	2nd–3rd order
Jacobian required	No	Yes	No
Computational cost	$O(n^2)$	$O(n^2)$	$O(n^3)$
Crazyflie use	—	Flow/Loco/ Lighthouse	Loco Positioning

The UKF achieves higher approximation accuracy at the cost of increased computation, requiring $\mathcal{O}(n^3)$ operations due to the Cholesky decomposition, compared to $\mathcal{O}(n^2)$ for the EKF [29]. On the Crazyflie, the EKF is the standard choice for most deck configurations, as it provides a good balance between accuracy and real-time feasibility. The UKF is available as an experimental alternative for Loco Positioning scenarios, where UWB range measurements introduce stronger non-linearities [31].

G. Implementation on the Crazyflie

The Crazyflie 2.1 Brushless runs a modular firmware that supports multiple state estimators simultaneously, selecting the appropriate one at boot time based on the detected hardware configuration [28]. By default, without any expansion deck

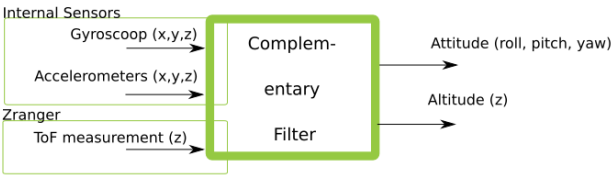


Fig. 11: Block diagram of the complementary filter on the Crazyflie. Gyroscope, accelerometer, and time-of-flight inputs are fused into attitude and altitude estimates without estimating horizontal position. Source: [35].

attached, the firmware activates the *complementary filter*, a lightweight algorithm that fuses gyroscope and accelerometer data to estimate attitude (roll, pitch, yaw) and altitude only, without estimating horizontal position. Figure 11 shows its block structure, in which the gyroscope, accelerometer, and time-of-flight inputs are combined into attitude and altitude outputs.

As soon as one of the following deck configurations is detected, the firmware automatically switches to the EKF.

- The **Flow Deck v2** fuses optical flow and time-of-flight height to estimate velocity and horizontal position, enabling stable hover without external infrastructure.
- The **Loco Positioning Deck** fuses UWB ranging measurements from fixed anchors to estimate absolute 3D position.
- The **Lighthouse Deck** fuses lighthouse sweep angles for high-precision absolute 3D position and attitude.

Figure 12 shows how the EKF integrates these deck inputs. The internal sensors and the available decks, namely the Flow Deck, the Loco Positioning Deck, the Lighthouse Deck, and an external motion-capture system, all feed into the EKF, which outputs attitude, position, and velocity estimates.

The switch is handled automatically through the `requiredEstimator` field in each deck's driver [28]. Table VII provides an overview. Note that adding the Multi-Ranger Deck to any configuration does not affect the selected estimator, as its distance measurements are used for obstacle avoidance rather than state estimation.

TABLE VII: State estimator selection on the Crazyflie 2.1 Brushless depending on the attached expansion deck.

Configuration	Estimator
No deck	Complementary
Z-Ranger Deck v2	Complementary
Flow Deck v2	EKF
Loco Positioning Deck	EKF / UKF (experimental)
Lighthouse Deck	EKF

The exact composition and dimension of the state vector estimated by the EKF depend on the attached deck configuration

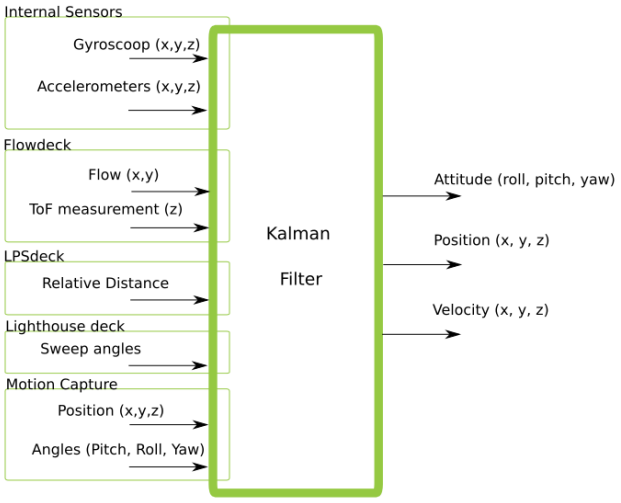


Fig. 12: Block diagram of the Extended Kalman Filter on the Crazyflie. Internal sensors together with the Flow Deck, Loco Positioning Deck, Lighthouse Deck, and motion-capture inputs are fused into attitude, position, and velocity estimates. Source: [35].

and the selected estimator rather than being fixed. A typical full-pose configuration estimates the following nine-dimensional state vector.

$$\mathbf{x} = \begin{bmatrix} p_x & p_y & p_z & v_x & v_y & v_z & \phi & \theta & \psi \end{bmatrix}^\top \tag{21}$$

Other configurations may estimate fewer or additional quantities, so this particular vector is only one representative example.

In the firmware, the IMU is sampled at 1000 Hz, but the accumulated accelerometer and gyroscope data drive the EKF prediction step at a fixed rate of 100 Hz [36]. Update steps are triggered asynchronously whenever a sensor delivers a new measurement, with the Flow Deck providing optical-flow readings at roughly 100 Hz. Each sensor supplies a single scalar measurement, so `kalmanCoreScalarUpdate()` is called once per measurement instead of performing a joint vector update. This is mathematically equivalent and avoids any matrix inversion beyond a scalar division, which improves numerical stability on the 32-bit microcontroller [37]. The covariance update uses the numerically stable Joseph form

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\, \mathbf{P}_{k|k-1}\, (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)^\top + \mathbf{K}_k\, \mathbf{R}\, \mathbf{K}_k^\top \tag{22}$$

which guarantees that $\mathbf{P}_{k|k}$ remains symmetric and positive semi-definite even under floating-point rounding errors. In addition, the implementation explicitly re-symmetrises the covariance matrix after every update and clamps its entries to fixed minimum and maximum bounds as a further safeguard against numerical degradation [37].

H. Demonstration using the Crazyflie

To demonstrate the EKF in operation, the optical-flow measurements of a Crazyflie 2.1 Brushless equipped with a Flow Deck v2 were logged during a structured hover flight and compared against the EKF prior prediction. The flight consisted of two phases. In the first half the drone performed three forward–backward manoeuvres along the body x-axis, and in the second half it performed three left–right manoeuvres along the body y-axis.

Figure 13 shows both axes of the optical-flow signal. The upper subplot (Axis X) captures the response to the forward–backward motion. The raw measurement (orange, `kalman_pred.measNX`) exhibits strong spikes during acceleration and deceleration, while the EKF prior $H\hat{\mathbf{x}}^{-}$ (green, `kalman_pred.predNX`), computed from the filtered velocity estimate projected into measurement space, follows the underlying motion smoothly and suppresses high-frequency sensor noise. The lower subplot (Axis Y) shows the complementary response to the left–right motion, where the same filtering behaviour is visible. During phases in which the drone moves predominantly along one axis, the orthogonal axis remains close to zero in both the raw and filtered signal, confirming that the EKF correctly decouples the two flow components. Overall, the plots illustrate the core function of the Kalman filter, namely fusing noisy sensor data with a motion model to produce a smooth, reliable state estimate in real time.

I. Conclusion

This paper traced the Kalman filter from its 1960 origins through the linear formulation, its non-linear extensions, and their concrete implementation on the Crazyflie 2.1 Brushless. The linear filter is optimal only under linear-Gaussian assumptions, which quadrotor dynamics and sensor models violate. The EKF restores applicability through local linearisation and is the firmware's default estimator for full-pose deck configurations. The UKF offers higher accuracy in strongly non-linear regimes such as UWB ranging, at additional computational cost. A numerical example illustrated how the predict step grows the covariance matrix and the update step shrinks it, and how the Kalman gain automatically balances trust between model and sensor. A short flow-deck demonstration showed the filter rejecting high-frequency noise while tracking the underlying motion, confirming that an algorithm first published in 1960 remains central to modern autonomous flight.

AI-Usage Disclosure for this Section

The authors of this section, Christian and Steffen, used Claude (Anthropic) to assist with drafting and revising text passages, generating LaTeX figure and table code, and locating and verifying firmware source references within the Kalman Filter section of this paper. All AI-generated content was reviewed, fact-checked, and edited by the authors, who take full responsibility for the accuracy, correctness, and originality of this section. This disclosure applies only to the Kalman Filter section. The authors make no claim about the use of generative AI in other parts of this paper.

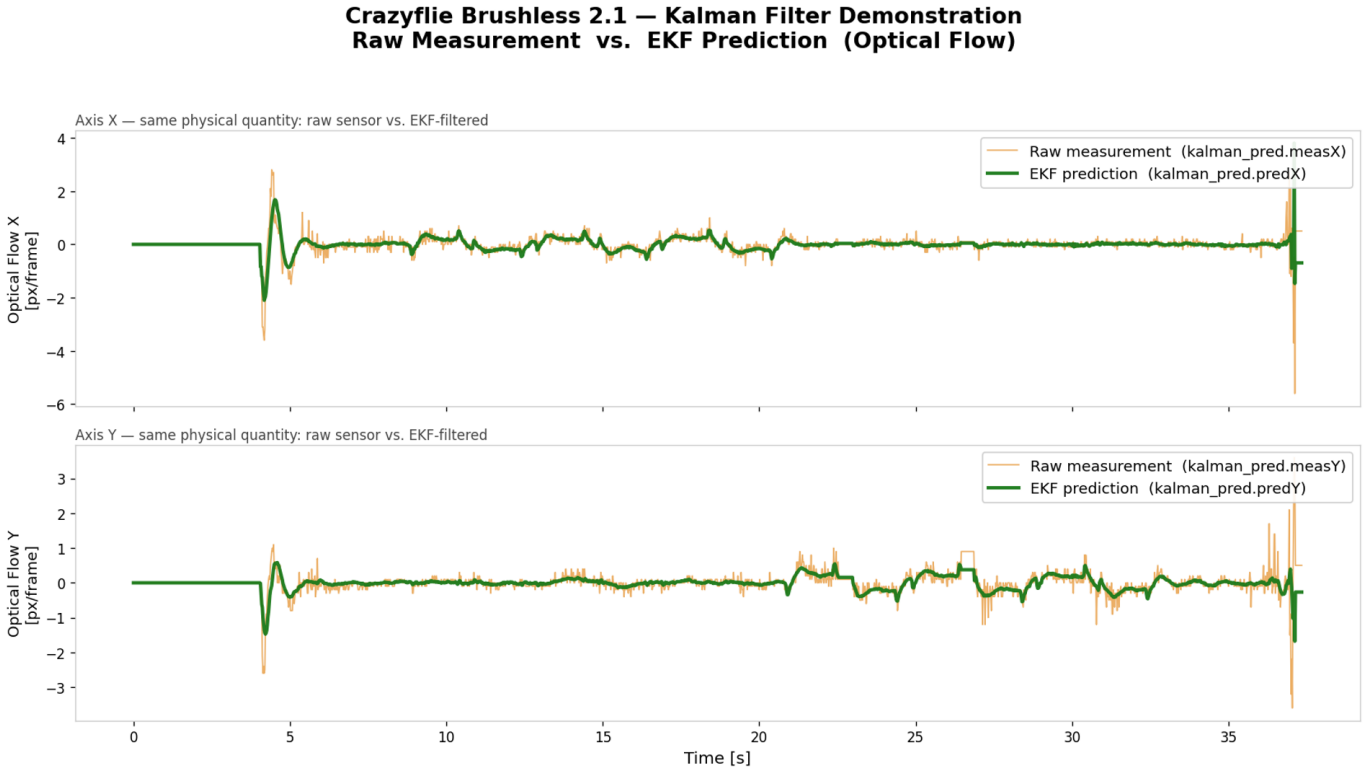


Fig. 13: Crazyflie 2.1 Brushless demonstration: raw optical-flow measurement versus EKF-filtered prediction for the x- and y-axis during a hover flight.

III. Orientation

Authors: Mariella Arlt, Filip Müller

***Abstract*—Spatial orientation describes how an object is rotated with respect to a reference frame. In robotics, virtual reality and quadrotor flight, this concept is not only a geometric description but also a practical engineering problem: the chosen representation determines how rotations are stored, combined, interpolated and interpreted. This paper introduces the most common orientation representations used in three-dimensional systems: rotation matrices, Euler angles, axis–angle representation and quaternions. Their mathematical structure is derived from simple vector rotations and then related to typical drone attitude updates. Special emphasis is placed on the singularity of Euler angles known as gimbal lock and on the reason quaternions avoid this problem while still using only four scalar values.**

***Index Terms*—spatial orientation, rotation matrices, Euler angles, gimbal lock, axis-angle, quaternions, quadrotors**

A. Introduction

Spatial orientation is the task of describing how an object is aligned in space. A drone, for example, does not only have a position; it also has a heading, a tilt and a roll angle. A virtual camera in a game engine, a robot arm joint and an inertial measurement unit all face the same mathematical issue: the system must store an orientation, update it when a rotation occurs and apply it to vectors such as body axes, velocities or sensor measurements.

The difficulty is that there is no single best representation for all use cases. Euler angles are easy for humans to read, but they are order-dependent and can cause gimbal lock. Rotation matrices are direct and robust for transforming vectors, but they require nine numbers and their orientation is less intuitive. Axis–angle notation is geometrically clear, but it is less convenient for repeated composition. Quaternions are compact, numerically stable and widely used in robotics and computer graphics, but their four-dimensional notation is initially less intuitive.

This paper provides an overview of the four methods for representing spatial orientation, explains how they are used, and highlights the differences between them.

B. Transformation

A transformation describes a geometric change of an object in space. It is used to mathematically represent changes in an object's position, orientation, or other geometric properties. The most important types of transformations are translations, rotations, scaling, and reflections.

For describing the motion of rigid bodies, translations and rotations are of particular importance. A translation changes the position of an object, whereas a rotation changes its orientation. Together, these two transformations provide a complete description of an object's pose in three-dimensional space.

Transformations are commonly described using concepts from linear algebra. In particular, rotations can be represented by matrices that define the orientation of an object relative to a reference coordinate system. If only the orientation changes

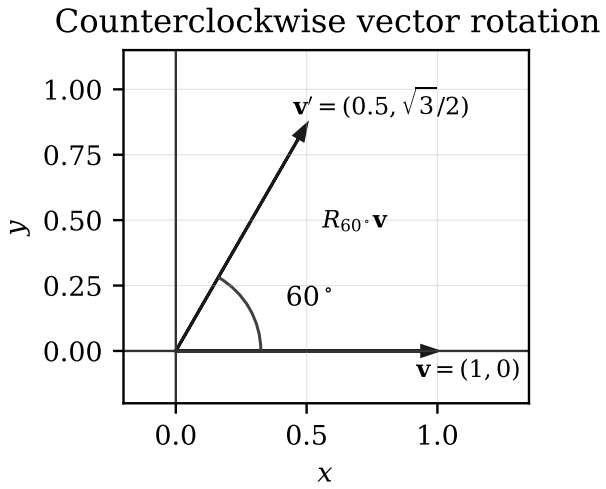


Fig. 14: A 2D vector rotation by $60°$. The rotated vector is obtained by multiplying the original vector with $R_{60°}$.

while the position remains fixed, the transformation reduces to a pure rotation.

The mathematical representation of rotations plays a fundamental role in many fields, including robotics, computer graphics, aerospace engineering, and autonomous systems. In these applications, the orientation of objects must be represented, stored, and processed accurately.

Since this paper focuses on orientation in space, the following sections concentrate on rotation and introduce several methods for representing three-dimensional orientations.

C. Rotation Matrices

A rotation matrix represents the orientation of an object as a matrix that can be used to rotate vectors in two- or three-dimensional space.

1) Two-Dimensional Rotation: In two dimensions, a counterclockwise rotation by an angle α is described by

$$R_\alpha = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}. \tag{23}$$

Multiplying a vector by this matrix rotates the vector around the origin. For example, rotating $\mathbf{v} = (1, 0)^T$ by $60°$ gives

$$R_{60°}\mathbf{v} = \begin{bmatrix} \cos 60° \\ \sin 60° \end{bmatrix} = \begin{bmatrix} 0.5 \\ \sqrt{3}/2 \end{bmatrix}. \tag{24}$$

The inverse matrix $R_\alpha^{-1} = R_{-\alpha} = R_\alpha^T$ rotates the vector clockwise by the same angle.

2) Three-Dimensional Rotation: In three dimensions, each coordinate axis has its own rotation matrix describing rotations about that axis:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, \tag{25}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{26}$$

$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{27}$$

A complete orientation can be produced by multiplying these matrices. Here is the roll–pitch–yaw convention used,

$$R = R_z(\psi)R_y(\theta)R_x(\phi), \tag{28}$$

where yaw corresponds to a rotation about the z-axis, pitch about the y-axis, and roll about the x-axis. Although the convention is written as yaw–pitch–roll (ZYX), the rotations are applied in the order roll $\rightarrow$ pitch $\rightarrow$ yaw, since matrix multiplication acts from right to left.

The order is essential: matrix multiplication is not commutative, so changing the order generally changes the final orientation.

Rotation matrices are attractive because they can be applied directly to vectors, preserve length and angles, and do not suffer from gimbal lock. Mathematically, a valid rotation matrix satisfies

$$R^T R = I, \qquad \det(R) = 1. \tag{29}$$

Their disadvantages are practical: nine values must be stored, repeated multiplication can slowly break orthogonality due to floating-point error, and a matrix is harder for humans to interpret than three labeled angles.

D. Euler Angles and Gimbal Lock

Euler angles describe a 3D orientation using three sequential rotations. In the common roll–pitch–yaw convention, roll is a rotation around the body or coordinate x-axis, pitch around the y-axis and yaw around the z-axis. This makes the representation highly readable. For example, "roll $30°$, pitch $45°$, yaw $60°$" is easier to understand than a 3×3 matrix.

However, Euler angles are not a vector and cannot be applied directly to a point. They first need to be converted into rotation matrices using an agreed rotation order. They also suffer from ambiguity: the same final orientation may be represented by different angle triples. The most serious issue is *gimbal lock*.

Gimbal lock occurs when two of the three rotation axes align. For roll–pitch–yaw, this happens at pitch $= \pm 90°$. At that point, yaw and roll no longer describe independent directions. A short matrix example makes this visible. At the critical pitch angle, the pitch matrix becomes

$$R_y(90°) = \begin{bmatrix} \cos 90° & 0 & \sin 90° \\ 0 & 1 & 0 \\ -\sin 90° & 0 & \cos 90° \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}. \tag{30}$$

Inserted into the full orientation matrix from (28), this gives

$$R_{GL} = \begin{bmatrix} 0 & \sin(\phi-\psi) & \cos(\phi-\psi) \\ 0 & \cos(\phi-\psi) & -\sin(\phi-\psi) \\ -1 & 0 & 0 \end{bmatrix}. \quad (31)$$

The matrix no longer depends on roll ϕ and yaw ψ separately, but only on their difference $\phi - \psi$. The orientation itself still exists, but the coordinate chart used by Euler angles becomes singular. As a result, small physical changes can cause large numerical changes in the angles, and one degree of freedom is effectively lost in the parameterization.

This is why Euler angles are useful for user interfaces, logging and simple explanations, but less suitable as the internal representation for continuous 3D attitude control.

This is why Euler angles are useful for user interfaces, logging and simple explanations, but less suitable as the internal representation for continuous 3D attitude control.

E. Axis–Angle and Rotation Vectors

Another common way to describe a three-dimensional rotation is the **axis–angle representation**. Instead of storing an orientation as a matrix or three Euler angles, a rotation is represented by two quantities: a rotation axis and a rotation angle. The axis specifies the line about which the rotation is performed, while the angle specifies how far the object is rotated around that axis. Together, these two parameters uniquely describe a single rotation.

The rotation axis is represented by a normalized vector

$$\mathbf{a} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T, \qquad |\mathbf{a}| = 1, \quad (32)$$

where only the direction of the vector is relevant. Its magnitude has no influence on the represented rotation. Although the coordinate axes are frequently used as rotation axes, any arbitrary axis in three-dimensional space can be chosen.

A vector can be rotated directly about the axis by the angle using **Rodrigues' rotation formula**,

$$\mathbf{v}' = \mathbf{v}\cos\omega + (\mathbf{a}\times\mathbf{v})\sin\omega + \mathbf{a}(\mathbf{a}\cdot\mathbf{v})(1-\cos\omega), \quad (33)$$

which computes the rotated vector without first constructing a rotation matrix.

Compared with Euler angles, the axis–angle representation does not suffer from gimbal lock and requires only four values to describe a rotation: three for the axis and one for the angle. It is also closely related to quaternions, since an axis–angle representation can be converted directly into a quaternion. However, combining multiple rotations is less straightforward than with rotation matrices, as rotations cannot simply be added together.

F. Quaternions

1) Complex Numbers as 2D Rotations: Before introducing quaternions, it is useful to briefly consider how complex numbers can represent rotations in two dimensions. A complex number can be written as

$$z = x + yi, \quad (34)$$

where the real part x describes the horizontal component and the imaginary part y describes the vertical component in the complex plane. A point or vector in 2D can therefore be interpreted as a complex number.

A rotation by an angle θ can be performed by multiplying the complex number with a unit complex number

$$r = \cos(\theta) + i\sin(\theta). \quad (35)$$

For example, rotating the vector $(1, 0)$ by 90° corresponds to multiplying the complex number $z = 1 + 0i$ with

$$r = \cos(90^\circ) + i\sin(90^\circ) = i. \quad (36)$$

The result is

$$z' = rz = i \cdot 1 = i. \quad (37)$$

This corresponds to the vector

$$(1, 0) \to (0, 1). \quad (38)$$

Thus, multiplication by a unit complex number rotates a vector in the 2D plane without explicitly using a rotation matrix.

2) From Complex Numbers to Quaternions: Complex numbers are not introduced because quaternions work exactly the same way. Instead, they provide an intuitive stepping stone: they show that rotations can be represented algebraically. Quaternions use the same general idea, but extend it to 3D by encoding both a rotation axis and a rotation angle.

In 2D, one imaginary unit i is sufficient because there is only one fixed plane of rotation. Every 2D rotation happens inside the same plane. In 3D, however, a rotation is not only described by an angle, but also by the axis around which the rotation happens. This axis can point in any direction in space.

Quaternions extend complex numbers with three imaginary basis elements i, j and k. A quaternion is written as

$$\mathbf{q} = w + xi + yj + zk, \quad (39)$$

where w is the scalar part and $(x, y, z)^T$ is the vector part. For rotations, the vector part is related to the rotation axis, while the scalar part is related to the rotation angle.

The multiplication rules of the imaginary basis elements are

$$i^2 = j^2 = k^2 = ijk = -1. \quad (40)$$

These rules make quaternion multiplication non-commutative. This is important because 3D rotations are also non-commutative: changing the order of rotations can change the final orientation.

For two quaternions $\mathbf{q}_1 = (w_1, \mathbf{u}_1)$ and $\mathbf{q}_2 = (w_2, \mathbf{u}_2)$, the product can be written as

$$\mathbf{q}_1 \otimes \mathbf{q}_2 = \big(w_1 w_2 - \mathbf{u}_1 \cdot \mathbf{u}_2,\ w_1\mathbf{u}_2 + w_2\mathbf{u}_1 + \mathbf{u}_1 \times \mathbf{u}_2\big). \quad (41)$$

The dot product and cross product in this equation show why quaternions are well suited for 3D rotations: they combine scalar angle information with vector axis information. However, a quaternion only represents a pure rotation if it is a unit quaternion. The next subsection explains how such a unit quaternion is built from an axis and an angle.

3) Building a Unit Quaternion: For rotation, quaternions must be unit quaternions. A unit quaternion satisfies

$$\|\mathbf{q}\|^2 = w^2 + x^2 + y^2 + z^2 = 1. \tag{42}$$

Given a normalized axis $\mathbf{a} = (a_x, a_y, a_z)^T$ and desired rotation angle ω, the corresponding rotation quaternion is

$$\mathbf{q} = \begin{bmatrix} \cos(\omega/2) & a_x \sin(\omega/2) & a_y \sin(\omega/2) & a_z \sin(\omega/2) \end{bmatrix}^T. \tag{43}$$

The vector part therefore encodes the rotation axis, scaled by $\sin(\omega/2)$, while the scalar part encodes the angle through $\cos(\omega/2)$. The half-angle is essential because the quaternion rotates a vector through a sandwich product: the quaternion acts from the left and its inverse acts from the right. Together they produce the full physical rotation angle ω.

For a unit quaternion, the inverse is easy to compute:

$$\mathbf{q}^{-1} = \begin{bmatrix} w & -x & -y & -z \end{bmatrix}^T. \tag{44}$$

This simple inverse is one reason unit quaternions are efficient in real-time systems. Normalization is still important, because numerical integration and repeated multiplication can cause drift. A common correction is

$$\mathbf{q} \leftarrow \frac{\mathbf{q}}{\|\mathbf{q}\|}. \tag{45}$$

4) Rotating a Vector: To rotate a vector with a quaternion, the vector is first converted into a pure quaternion,

$$\mathbf{p} = (p_x, p_y, p_z)^T \quad \rightarrow \quad P = [0, p_x, p_y, p_z]. \tag{46}$$

The rotated vector is then

$$P' = \mathbf{q} \otimes P \otimes \mathbf{q}^{-1}, \tag{47}$$

where $\otimes$ denotes quaternion multiplication and $\mathbf{q}^{-1}$ is the inverse. For unit quaternions, the inverse is simply the conjugate,

$$\mathbf{q}^{-1} = \begin{bmatrix} w & -x & -y & -z \end{bmatrix}^T. \tag{48}$$

5) Quaternion from Two Vectors: A practical construction is to compute the quaternion that rotates one vector onto another. Let $\mathbf{v}_1$ and $\mathbf{v}_2$ be normalized vectors. The rotation axis and angle are

$$\mathbf{a} = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{\|\mathbf{v}_1 \times \mathbf{v}_2\|}, \tag{49}$$

$$\omega = \operatorname{atan2}(\|\mathbf{v}_1 \times \mathbf{v}_2\|, \mathbf{v}_1 \cdot \mathbf{v}_2). \tag{50}$$

The quaternion is then built using (**??**). For the example $\mathbf{v}_1 = (1, 0, 0)^T$ and $\mathbf{v}_2 = (0, 1, 0)^T$,

$$\mathbf{a} = (0, 0, 1)^T, \qquad \omega = 90^\circ, \tag{51}$$

which gives

$$\mathbf{q} = \begin{bmatrix} 0.7071 & 0 & 0 & 0.7071 \end{bmatrix}^T. \tag{52}$$

Applying (47) rotates $(1, 0, 0)^T$ onto $(0, 1, 0)^T$.

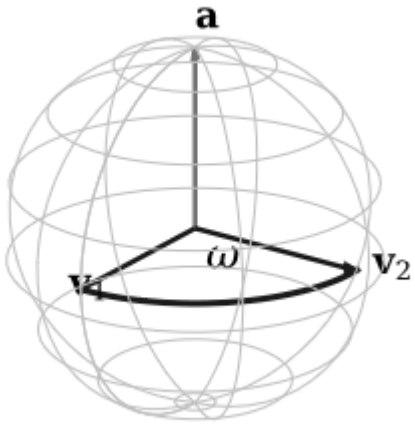


Fig. 15: A quaternion can be constructed from a rotation axis and a rotation angle. For $\mathbf{v}_1 = (1, 0, 0)^T$ and $\mathbf{v}_2 = (0, 1, 0)^T$, the axis is $\mathbf{a} = (0, 0, 1)^T$ and the angle is 90°.

6) Orientation Update and Composition: The vector rotation formula (47) should not be confused with an orientation update. If $\mathbf{q}_{current}$ stores the current attitude of a drone and $\mathbf{q}_\Delta$ describes a newly measured or commanded rotation, the updated attitude is obtained by quaternion composition:

$$\mathbf{q}_{new} = \mathbf{q}_\Delta \otimes \mathbf{q}_{current}. \tag{53}$$

This equation means: first interpret the existing orientation, then apply the new rotation according to the selected frame convention. Some software libraries use the reverse order depending on whether the incremental rotation is expressed in the world frame or in the local body frame. The important principle is that quaternion multiplication composes rotations in the same way matrix multiplication composes rotation matrices.

For a drone, $\mathbf{q}_{current}$ may come from an estimator that fuses gyroscope and accelerometer measurements. If the controller commands an additional yaw rotation of 90° around the z-axis, the increment is

$$\mathbf{q}_\Delta = \begin{bmatrix} 0.7071 & 0 & 0 & 0.7071 \end{bmatrix}^T. \tag{54}$$

Starting from the identity orientation $[1, 0, 0, 0]^T$, the updated orientation is exactly $\mathbf{q}_\Delta$. From a non-trivial current attitude, the multiplication in (53) produces a different quaternion representing the composed orientation.

G. Comparison of Representations

Table VIII summarizes the practical trade-offs from the presentation.

TABLE VIII: Comparison of common 3D rotation representations.

Representation	Values	Practical properties
Euler angles	3	Intuitive and compact, but order-dependent, ambiguous and vulnerable to gimbal lock.
Axis–angle	4	Intuitive for one fixed-axis rotation and no gimbal lock, but rotations do not combine by direct addition.
Rotation matrix	9	Direct vector transformation and no gimbal lock, but more memory, less readable and may require re-orthogonalization.
Quaternion	4	Compact, stable composition and no gimbal lock, but less intuitive, requires normalization and has sign ambiguity.

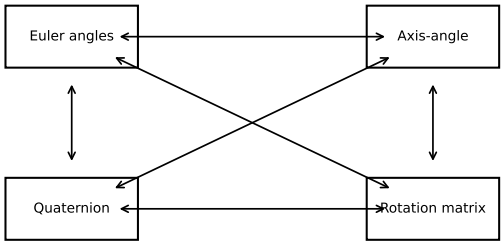


Fig. 16: All common 3D orientation representations describe the same physical rotation and can be converted into one another.

One additional property of quaternions is sign ambiguity. The quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same physical orientation. This is not a problem for rotation itself, but it matters when comparing logged values or interpolating orientations. Implementations commonly choose the sign that keeps consecutive quaternions close in dot product.

H. Application to Drone Attitude Control

In a drone, attitude describes the current orientation of the vehicle. The controller compares the desired attitude with the current attitude and computes an attitude error. In quaternion notation, this can be written as

$$\mathbf{q}_{error} = \mathbf{q}_{desired} \otimes \mathbf{q}_{current}^{-1}. \tag{55}$$

The controller also uses angular velocity from the gyroscope,

$$\boldsymbol{\omega} = \begin{bmatrix} p & q & r \end{bmatrix}^T, \tag{56}$$

where p is roll rate, q is pitch rate and r is yaw rate. The controller then calculates motor commands that reduce the attitude error and damp the angular velocity.

This explains why drones often use several representations simultaneously. The software can store attitude as a quaternion, convert it to roll–pitch–yaw values for telemetry and debugging, and convert it to a rotation matrix when vectors or coordinate frames must be transformed in control calculations.

I. Conclusion

The main 3D rotation representations each have a distinct role:

- **Rotation matrices** are robust and directly transform vectors and coordinate frames. However, they require nine values and are difficult to interpret visually.
- **Euler angles** are compact and human-readable. However, the rotation sequence must be clearly defined, and the representation suffers from gimbal lock.
- **Axis–angle representation** describes one rotation by a fixed axis and an angle. It is intuitive and closely related to Rodrigues' rotation formula, but it is less convenient for combining multiple rotations.
- **Quaternions** provide a compact and nonsingular way to store and update orientations. They are well suited for continuous rotation composition, but they are less intuitive, must be normalized, and have a sign ambiguity.

In practice, this shows that not just one representation is used. Instead, different representations are combined depending on whether the orientation needs to be stored, displayed, or used for control calculations.

IV. Control Systems, Sensor Fusion, Attitude Control, and Actuation in Unmanned Aerial Vehicles

Authors: Sebastian Nagles, Toni Kostov

A. Abstract

This part of the document examines the fundamental principles that prevent a drone from falling. It begins with the closed-loop control architecture that continuously corrects deviations from a desired state, followed by the sensor suite that provides the necessary state measurements and the fusion algorithms that combine them. The document then introduces the Roll-Pitch-Yaw (RPY) convention used to describe drone orientation, and the cascaded attitude control architecture — including rate control, rotational control computation, and motor mixing — that translates orientation errors into motor commands. The PID controller, the workhorse of UAV stabilisation, is analysed in terms of its three components and their respective trade-offs. Finally, the brushed and brushless DC motors that physically execute the computed commands are described. Throughout, the Crazyflie nano-quadrotor serves as the concrete reference platform.

B. Introduction

Consumer drones have become a familiar sight — delivering parcels, filming weddings, inspecting infrastructure — yet the engineering that keeps them airborne is far from obvious. A multirotor is, by its physical nature, an unstable system: remove the electronics and it falls immediately. What prevents this is not a mechanical trick but a continuous, high-speed conversation between sensors, algorithms, and motors, repeated roughly a thousand times per second. Understanding this conversation

requires working across several disciplines at once. Control theory provides the mathematical framework for turning a measurement error into a corrective action. MEMS sensor technology determines how accurately that error can even be measured. Signal processing fuses multiple imperfect sensor streams into a single reliable estimate. Rigid-body dynamics govern how a torque applied at a motor propagates into a change in the vehicle's orientation. And electromechanics dictates how fast and precisely a motor can execute the computed command.

None of these layers is sufficient on its own, and none can be fully understood in isolation from the others. This section of the document traces the complete chain — from the physical sensor to the spinning propeller — using the Crazyflie nano-quadrotor as a concrete reference platform. The goal is not to provide an exhaustive treatment of any single topic, but to give the reader a coherent picture of how these disciplines connect to produce stable autonomous flight.

C. Control Systems

1) What is a Control System?: A *control system* is an arrangement of components designed to regulate a physical process so that a desired output is maintained despite disturbances. In the context of a drone, the *process* is the rigid-body dynamics of the vehicle; the desired output is a target attitude, altitude, or position; and the disturbances are wind gusts, motor asymmetries, and sensor noise.

Every control system involves three functional stages [38]:

1) **Input:** a reference or setpoint specifying the desired state (e.g., "hover at 1 m altitude with level attitude").
2) **Processing:** a controller that computes the required actuator command.
3) **Output:** the resulting physical action (motor speeds) that drives the process toward the setpoint.

2) Open-Loop Control: In an *open-loop* system, the controller issues commands based solely on the input; it does not measure whether the desired output has actually been achieved. A simple example is a microwave oven: the user sets the time and power level, and the oven runs for the specified duration regardless of whether the food is cooked.

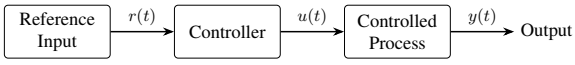


Fig. 17: Block diagram of an open-loop control system. The output is not fed back to the controller.

Open-loop control is attractive for its simplicity and low cost, but it offers no error correction. Any model inaccuracy, manufacturing tolerance, or external disturbance will cause the actual output to deviate from the desired value with no mechanism to compensate. For a drone, whose aerodynamic environment is constantly changing, open-loop control alone is entirely insufficient.

3) Closed-Loop Control: A *closed-loop* (or *feedback*) control system measures the actual output, compares it with the desired setpoint, and feeds the resulting *error* back into the controller to drive the error toward zero. This architecture is the cornerstone of virtually all practical control applications [39].

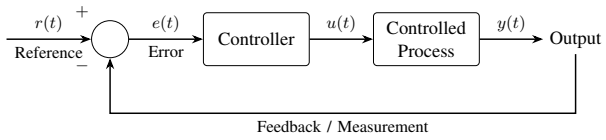


Fig. 18: Block diagram of a closed-loop control system. The measured output is subtracted from the reference to form an error signal that drives the controller.

4) Closed-Loop Control Applied to a Drone: Figure 19 illustrates the closed-loop architecture of the Crazyflie quadrotor. The desired state (e.g., "hover at 1 m") is compared with measurements from the onboard sensors. The difference — the error — is processed by a PID (Proportional-Integral-Derivative) controller implemented in the Crazyflie firmware, which computes the required motor speeds. Each motor then drives a propeller, generating thrust and torque that alter the drone's physical state. The updated state is continuously re-measured by the sensors, closing the loop.

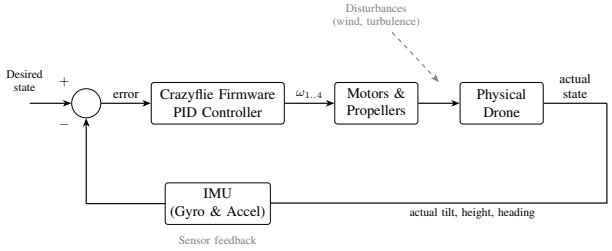


Fig. 19: Closed-loop control architecture of the Crazyflie quadrotor. Sensor measurements are fed back to the firmware controller, which continuously adjusts motor speeds to minimise the error between the desired and actual state.

D. Sensors

"The control loop is only as good as its measurements." This principle underscores why the sensor suite is as critical as the controller itself. A drone must estimate its orientation, acceleration, altitude, and optionally its absolute position to close the feedback loop. Different sensor modalities contribute different quantities; none is sufficient on its own.

1) Gyroscope:

a) Measurement Principle: A gyroscope measures the *angular rate* (rate of rotation) around each of the three body axes: roll, pitch, and yaw. The output is expressed in degrees per second ($^\circ/\mathrm{s}$).

Modern miniaturised gyroscopes — such as those used in the Crazyflie's BMI088 IMU — exploit the *Coriolis effect*. A Micro-Electro-Mechanical Systems (MEMS) gyroscope contains a microscale proof mass that is driven to oscillate at a known frequency. When the sensor experiences an external angular rate $\dot{\theta}$, the Coriolis force $\mathbf{F}_c = -2m\,\boldsymbol{\Omega} \times \dot{\mathbf{x}}$ acts perpendicular to both the oscillation velocity $\dot{\mathbf{x}}$ and the angular

velocity vector $\boldsymbol{\Omega}$, causing a secondary displacement of the proof mass. Here m is the mass of the proof mass, $\boldsymbol{\Omega}$ is the externally applied angular velocity vector (the quantity being measured), and $\dot{\mathbf{x}}$ is the velocity of the proof mass along its driven oscillation axis. This displacement is detected capacitively and converted to an angular rate.

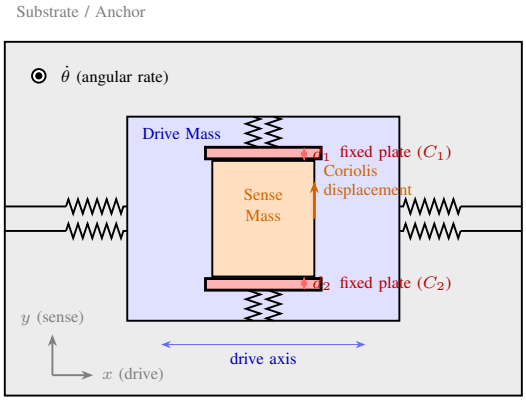


Fig. 20: Schematic cross-section of a MEMS gyroscope. The oscillating proof mass experiences a Coriolis-induced displacement perpendicular to its drive motion when an angular rate is applied; this displacement is read out as a change in capacitance.

b) Limitations: MEMS gyroscopes exhibit *drift*: a slowly accumulating error in the estimated angle caused by tiny, persistent measurement biases. To understand why, it is important to recall that the gyroscope does not directly measure the drone's angle — it only measures how fast that angle is currently changing (the angular rate in $^\circ/\mathrm{s}$). To obtain the actual angle, the firmware must maintain a running total: at every time step, it takes the current angular rate, multiplies it by the elapsed time, and adds the result to the previously stored angle. For example, if the gyroscope reads $3^\circ/\mathrm{s}$ for two seconds, the firmware adds $3 \times 2 = 6^\circ$ to the stored angle.

The problem arises because real gyroscopes are never perfectly accurate — there is always a tiny residual reading even when the drone is completely still, caused by manufacturing imperfections, temperature effects, and electronic noise. Suppose the sensor has a bias of just $0.02^\circ/\mathrm{s}$. This error gets added into the running total at every single time step. After one minute the angle estimate is already off by $0.02 \times 60 = 1.2^\circ$, and after ten minutes it has drifted by 12° — enough to cause a serious control error. This is analogous to navigating with a compass that always points 1° too far to one side: after a short walk the error is barely noticeable, but after a long journey it has led you far off course. For this reason, gyroscope output must be fused with complementary measurements, as described in Section IV-D6.

2) Accelerometer:

a) Measurement Principle: An accelerometer measures *specific force*: the sum of linear acceleration and gravitational acceleration along each axis. The output unit is metres per second squared ($\mathrm{m/s^2}$).

A MEMS accelerometer contains a proof mass suspended by flexible beams between two sets of fixed capacitor plates — one set on each side of the mass, forming two capacitors with capacitances C_1 and C_2. At rest, the proof mass sits midway between the two sets of plates, so $C_1 = C_2$. When an external acceleration a is applied, the mass displaces by a distance δ along the sense axis: it moves closer to one set of plates and farther from the other, increasing one capacitance while decreasing the other ($C_1 \neq C_2$). This capacitance difference $C_1 - C_2$ is measured electronically and is proportional to the displacement δ, which in turn is proportional to the applied acceleration a.

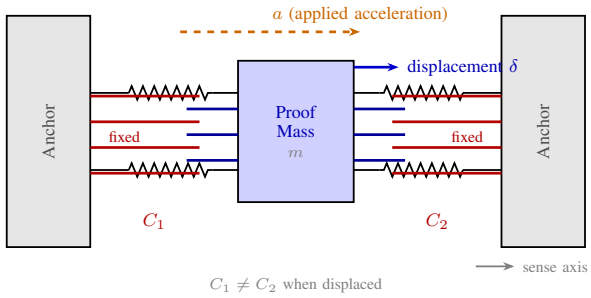


Fig. 21: Schematic cross-section of a MEMS accelerometer.

b) Use in Attitude Estimation: Unlike the gyroscope, the accelerometer does not need a running total to estimate the drone's angle. Instead of measuring a rate that must be accumulated over time, it directly measures the forces acting on the sensor at each instant — including the gravitational force, which always points straight downward. By looking at the direction of this measured gravity vector, the firmware can compute the tilt angle freshly and independently at every time step, with no dependence on previous measurements. There is nothing accumulating, so there is nothing to drift.

At rest or during slow manoeuvres, the dominant component of the accelerometer output is indeed the gravity vector, and the tilt angle can be estimated from:

$$\phi = \arctan\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right), \quad \theta = \arctan\left(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}\right) \tag{57}$$

where ϕ is the roll angle, θ is the pitch angle, and a_x, a_y, a_z are the accelerometer outputs along each axis.

The downside of this direct measurement is that whenever the drone accelerates horizontally, that acceleration adds to the measured force vector and temporarily shifts its apparent direction away from true gravity, making the angle estimate unreliable during aggressive manoeuvres. The moment the drone stops accelerating, however, the very next reading is correct again — completely unaffected by what came before.

3) Inertial Measurement Unit (IMU): An Inertial Measurement Unit combines a gyroscope and an accelerometer into a single package. The Crazyflie uses the **Bosch BMI088**, a 16-bit six-axis IMU consisting of a triaxial accelerometer and a

triaxial gyroscope. The BMP388 pressure sensor is co-located on the same board to provide altitude information. Together these constitute the primary state-estimation platform for the Crazyflie.



Fig. 22: The Bosch BMI088 used in Crazyflie 2.1 Brushless

4) Barometer:

a) Measurement Principle: Atmospheric pressure decreases with altitude in a well-characterised manner described by the international standard atmosphere. A MEMS barometer — such as the Bosch BMP388 — measures absolute pressure using a thin, elastic silicon diaphragm. One side of the diaphragm is exposed to ambient pressure; the other side is sealed at a known reference pressure. The diaphragm and a nearby fixed electrode together form a parallel-plate capacitor with capacitance C. Applied pressure deflects the diaphragm, changing the spacing between the diaphragm and the fixed electrode, and therefore changing the capacitance: $C = \varepsilon_r \varepsilon_0 \frac{A}{d}$ where C is the resulting capacitance, ε_r is the relative permittivity of the dielectric, $\varepsilon_0 = 8.854 \times 10^{-12}$ F/m is the permittivity of free space, A is the electrode area, and d is the plate separation (i.e., the gap between the diaphragm and the fixed electrode). The measured capacitance change is thus proportional to the applied pressure.

b) Limitations: Barometric altitude is subject to drift due to weather-related pressure changes and turbulence. Short-term accuracy is typically ± 0.5–1 m; it is insufficient as a standalone height sensor during aggressive flight. Its key advantage, however, is that it works in any environment — it requires no surface below the drone, no lighting, and no external infrastructure. On the Crazyflie, the barometer provides a continuous absolute altitude estimate that other sensors can use as a reference, particularly during indoor flight where GPS is unavailable.

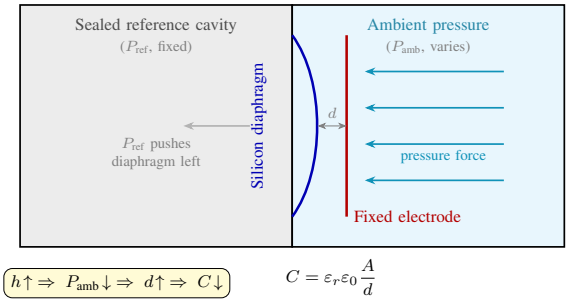


Fig. 23: Cross-section of a MEMS capacitive barometer. Ambient pressure deflects the silicon diaphragm toward the fixed electrode, reducing the plate separation d and increasing the measured capacitance C. Because atmospheric pressure falls predictably with altitude, C provides an indirect altitude measurement.

5) Position Sensors: Depending on whether the operating environment is outdoor or indoor, position can be estimated using different methods.

a) GPS (Outdoor): The Global Positioning System (GPS) determines position by *trilateration*: measuring the signal travel time from satellites whose orbits are precisely known, and computing the position as the intersection of the corresponding distance spheres. In three-dimensional space, position has three unknowns (x, y, z), which would seem to require only three satellites. However, there is a fourth unknown: the receiver's own clock error. GPS works by measuring how long a signal takes to travel from the satellite to the receiver, but even a clock error of one microsecond translates to roughly 300 m of position error, since radio signals travel at the speed of light. A dedicated atomic clock on the receiver would solve this, but the hardware is far too large and expensive for a small drone. Instead, a fourth satellite is used: four range measurements give four equations, which allow the receiver to solve simultaneously for its three spatial coordinates *and* its clock offset, cancelling the clock error entirely without any dedicated timing hardware. Civilian GPS offers accuracy of approximately 1–3 m horizontally. Its primary drawback is that satellite signals are attenuated by rooftops and walls, making GPS unreliable indoors or in urban canyons.

b) Optical Flow (Indoor): An optical flow sensor consists of a downward-facing camera combined with a cross-correlation algorithm that tracks the motion of texture patterns in consecutive image frames. Its primary output is horizontal velocity: by measuring how many pixels the ground texture has shifted between two frames, and combining that with the known altitude, the sensor computes how fast the drone is moving sideways. This makes optical flow most useful for stabilising horizontal position indoors, where GPS is unavailable and the IMU alone accumulates velocity errors too quickly to maintain a hover.

As a secondary capability, optical flow can also estimate changes in altitude: the faster the ground texture grows in the camera image, the closer the drone is to the surface. This relative height estimate is, however, less reliable than the barometer's absolute pressure reading — it requires a textured surface and adequate lighting to work at all, and it fails completely over featureless or uniformly coloured floors, in darkness, or when the ground is reflective.

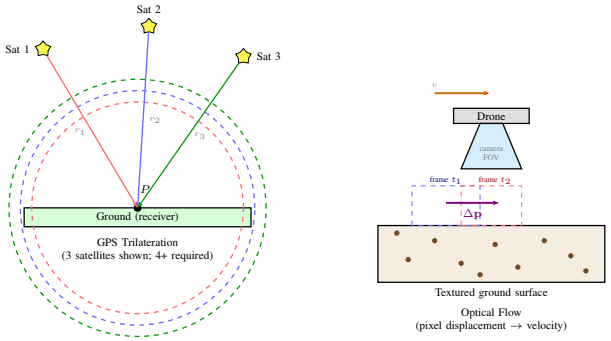


Fig. 24: Position sensing modalities. *Left:* GPS trilateration — the receiver at P computes its position from the ranges r_1, r_2, r_3 to satellites with known orbits. *Right:* Optical flow — a downward camera tracks the shift $\Delta\mathbf{p}$ of ground-texture features between successive frames to estimate horizontal velocity.

6) Sensor Data Fusion: As the preceding sections show, every sensor on the Crazyflie has a specific strength and a specific failure mode. The gyroscope is fast and precise on short timescales but drifts over time. The accelerometer provides an absolute gravity-anchored angle reference but is disturbed during motion. The barometer gives a reliable absolute altitude but is noisy on short timescales and sensitive to atmospheric conditions. Optical flow measures horizontal velocity and relative altitude changes with high precision but requires a textured, well-lit surface below the drone. No single sensor provides a complete, noise-free estimate of the drone's state — and crucially, the failure modes of these sensors are *complementary*: where one sensor is unreliable, another tends to remain accurate. The flight controller therefore combines multiple sensor streams through *data fusion*, exploiting these complementary characteristics to obtain estimates that are more accurate and more robust than any individual sensor could achieve alone.

a) Why Fuse the Gyroscope and Accelerometer?: The gyroscope and accelerometer are fused because neither one alone can provide an accurate, stable angle estimate, yet together they can. The gyroscope measures *angular rate* (how fast the angle is changing), not the angle itself. To obtain an angle, the firmware maintains a running total of these rate measurements over time — but as explained in Section **??**, any tiny bias in the sensor gets added into this running total at every time step, causing the angle estimate to drift further and further from reality. The accelerometer, by contrast, measures the direction of the gravitational force and can compute the tilt angle directly and independently at every time step, with no running total and therefore no drift. Its weakness is that any horizontal acceleration of the drone adds to the measured force vector, temporarily making the angle estimate unreliable during manoeuvres.

The solution is to use the gyroscope for fast, short-term tracking of angle changes (where it is accurate before drift has time to accumulate) and to use the accelerometer to periodically correct the long-term error (where its independent gravity reference cancels the gyro's accumulated bias). This is exactly the role of the complementary filter below.

b) Complementary Filter: The simplest fusion approach is the *complementary filter*. It combines the two sensors by weighting each according to the timescale at which it is reliable:

$$\hat{\phi} = \alpha \left(\hat{\phi}_{k-1} + \dot{\phi}_{\text{gyro}} \cdot \Delta t \right) + (1 - \alpha)\, \phi_{\text{accel}} \tag{58}$$

where $\hat{\phi}$ is the current filtered angle estimate, $\hat{\phi}_{k-1}$ is the angle estimate from the previous time step, $\dot{\phi}_{\text{gyro}}$ is the angular rate measured by the gyroscope, Δt is the time elapsed since the previous step, and ϕ_{accel} is the tilt angle computed directly from the accelerometer's gravity vector. The first term updates the angle estimate using the gyroscope rate, just as the firmware would do without fusion — but instead of trusting it fully, it is weighted by α. The second term pulls the estimate back toward the accelerometer's independent angle reading, weighted by $(1 - \alpha)$.

The factor $\alpha \in (0, 1)$ is a design parameter chosen by the engineer, not computed automatically. A value close to 1 (typically 0.98) means the gyroscope dominates on each time step — the estimate tracks fast rotations faithfully — while the small $(1 - \alpha) = 0.02$ weight on the accelerometer is enough to slowly correct the gyro's accumulated drift over many time steps. The value is chosen through tuning. Once set, α does not change during flight in a basic complementary filter — it is a fixed design choice. The Kalman filter, described next, can be understood as a mathematically optimal version of this idea that computes the equivalent of α automatically and adjusts it in real time based on the measured noise level of each sensor.

c) Kalman Filter: For more demanding applications, the *Kalman filter* provides an optimal recursive estimator in the presence of Gaussian noise [40]. It maintains a probabilistic belief over the drone's state vector (position, velocity, orientation, sensor biases) and updates it as new measurements arrive. The Extended Kalman Filter (EKF) generalises this to nonlinear systems such as drone dynamics. The Crazyflie firmware uses an EKF that fuses IMU, barometer, and optical flow (or UWB) measurements. A detailed explanation of the Kalman filter is available in Section II-F1.

E. Attitude Control

1) Introduction to Attitude Control: Attitude control is one of the most fundamental subsystems in unmanned aerial vehicles (UAVs), especially in multirotor drones such as quadcopters. Its main purpose is to maintain and regulate the orientation of the drone during flight. Without an effective attitude control system, a drone would quickly become unstable and uncontrollable. Modern UAVs operate in highly dynamic environments and must constantly react to disturbances such as wind, sudden motion changes, or payload variations. To achieve stable flight, the control system continuously measures the current orientation of the vehicle, compares it with the desired orientation, and generates corrective actions through the motors. Attitude control operates as one of the fastest feedback loops in the UAV control architecture, typically running at

higher frequencies than outer-loop controllers such as position and velocity control. In many multirotor systems, position and velocity controllers operate at lower frequencies (e.g., around 100 Hz), while attitude and rate controllers run at higher frequencies (e.g., several hundred Hz). This allows the UAV to react rapidly to changes in orientation and external disturbances. Because of this, it represents one of the fastest and most critical feedback loops in the entire flight control architecture.

2) Definition of Attitude (Roll, Pitch, Yaw): The term attitude refers to the orientation of the drone relative to a reference frame, usually the world frame or inertial frame. In UAV systems, attitude is typically described using three rotational angles that are often referred to as RPY (Roll–Pitch–Yaw):

- **Roll (ϕ):** rotation around the longitudinal axis, producing a left or right tilt.
- **Pitch (θ):** rotation around the lateral axis, producing a forward or backward tilt.
- **Yaw (ψ):** rotation around the vertical axis, changing the drone's heading direction.

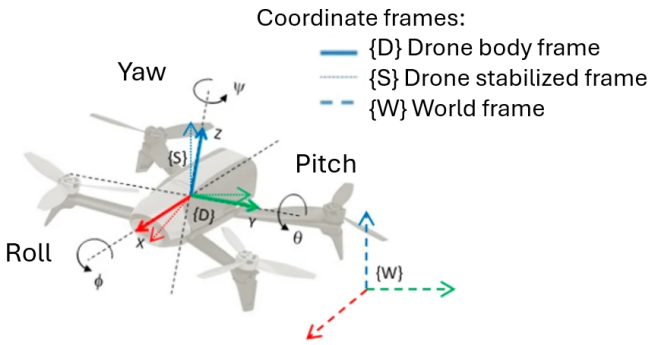


Fig. 25: Roll, pitch, and yaw angles describing the attitude of a multirotor UAV.
Adapted from: [41]

As illustrated in Figure 25, the attitude of a UAV describes its orientation in space and can be represented by three rotational degrees of freedom around the vehicle's body axes.

In UAV systems, two reference frames are commonly used to describe motion: the world frame, which provides a fixed reference relative to the Earth, and the body frame, which is attached to the drone and moves together with it. Figure 25 shows the relationship between these two reference frames, where the world frame remains stationary while the body frame rotates according to the UAV's orientation. The attitude of the UAV describes the orientation of the body frame relative to the world frame. Therefore, changes in roll, pitch, and yaw represent rotations of the vehicle's local coordinate system with respect to the fixed external reference frame. This relationship allows the UAV's orientation to be described and controlled independently of its position in the environment.

Euler angles are frequently used because they are intuitive and easy to visualize. An important characteristic of Euler-angle representations is that the order of rotations matters. Rotations in three-dimensional space are not commutative, meaning that applying roll, pitch, and yaw in different sequences produces different final orientations. For this reason, UAV systems typically follow a specific rotation convention, such as yaw–pitch–roll or roll–pitch–yaw, depending on the implementation [42]. Although Euler angles offer a geometrically intuitive representation of orientation, advanced UAV systems often use quaternions internally to avoid singularities such as gimbal lock and to provide smoother mathematical representations of orientation.

The drone's attitude directly influences its movement. For example, a forward pitch causes the drone to accelerate forward, while a roll angle causes lateral motion. Therefore, controlling orientation is essential for controlling position and trajectory.

3) Purpose of Attitude Control: The objective of the attitude controller is to minimize the error between the desired orientation and the measured orientation of the UAV. To achieve this, the controller receives the current orientation from onboard sensors and a desired orientation generated by the pilot, an autopilot, or higher-level control algorithms. Based on the difference between these two values, it computes corrective control actions that are ultimately translated into motor commands to stabilize the vehicle and guide it toward the desired attitude.

For example, if the drone tilts too far to the right, the controller modifies the thrust distribution among the motors to generate a compensating roll torque that brings the drone back to the desired orientation. Attitude control therefore acts as the bridge between high-level flight objectives and low-level motor actuation. Higher-level systems may command actions such as "move forward" or "maintain position," but these commands are ultimately translated into desired orientations that the attitude controller must achieve.

4) Cascaded Control Architecture: Most UAVs use a cascaded or hierarchical control architecture. Instead of controlling everything in a single loop, the system is divided into multiple layers with different responsibilities and operating frequencies. Outer loops handle slower objectives such as position and velocity, while inner loops handle faster dynamics such as attitude and angular rates. The most common structure consists of:

- A position controller
- A velocity controller
- An attitude controller
- A rate controller
- A control allocation or motor mixing stage

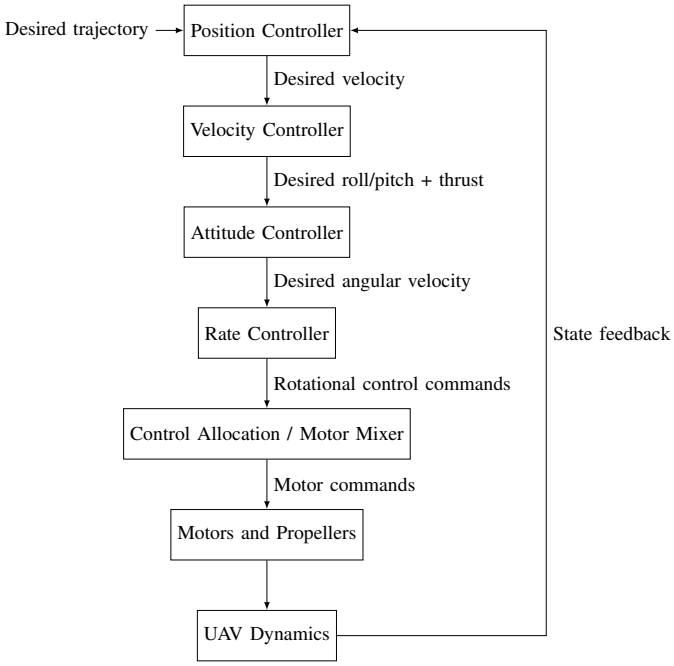


Fig. 26: Simplified cascaded control architecture for a multirotor UAV.

Figure 26 illustrates the information flow within a typical cascaded UAV control architecture. The outer-loop position controller generates desired velocity references, which are tracked by the velocity controller. Based on the velocity error, the velocity controller computes the desired roll and pitch angles together with the required total thrust. These attitude references are passed to the attitude controller, which generates desired angular velocity references. The rate controller tracks these angular velocities and produces the rotational control commands required to achieve the desired motion. Finally, the control allocation stage converts the thrust and rotational control commands into individual motor commands. The motors and propellers then generate the forces and moments acting on the UAV, while sensor measurements continuously close the feedback loop by providing updated state information.

a) Position Controller: The position controller represents the outermost layer of the cascaded control architecture. It compares the desired position of the UAV with its estimated current position and computes the desired translational velocity required to reduce the position error. Because the position of the vehicle changes relatively slowly, this controller typically operates at a lower update frequency than the inner control loops.

b) Velocity Controller: The velocity controller receives the desired velocity generated by the position controller and compares it with the estimated current velocity of the UAV. Based on the velocity error, it computes the desired roll and pitch angles required to generate horizontal motion, together with the total thrust required to control the vertical motion. These outputs are then passed to the attitude controller and the control allocation stage.

c) Attitude Controller: The attitude controller regulates the orientation of the UAV. It receives the desired attitude generated by higher-level controllers, such as the position and velocity controllers, and compares it with the current estimated attitude. It compares the desired attitude with the current estimated attitude and generates desired angular rates as output. For example, if the drone should pitch forward by 10 degrees, the outer loop determines how quickly the vehicle should rotate to achieve that target. The output of this stage is therefore not motor commands directly, but desired angular velocity references for the rate controller.

d) Rate Controller: The rate controller regulates the angular velocity of the UAV. It compares the desired angular velocity generated by the attitude controller with the measured angular velocity obtained directly from the gyroscope. Based on this error, the controller generates the rotational control commands required to achieve the desired rotational motion. The exact form of these control commands depends on the selected rate controller. Classical PID controllers typically generate roll, pitch, and yaw control commands directly, whereas model-based controllers such as the Brescianini or Lee controllers explicitly compute the torques required to achieve the desired rotational behavior [43], [44].

The rate controller is considered one of the innermost control loops because it directly regulates the angular velocity of the UAV and operates closest to the physical actuation of the motors. Although different control loops may run at different update frequencies depending on the UAV platform, the rate controller remains the lower-level rotational controller because it directly stabilizes angular motion, while the attitude controller generates the desired angular velocity references. This cascaded structure follows the natural dynamics of rotational motion. The orientation of the UAV evolves as a result of changes in angular velocity, making angular velocity an effective variable for fast stabilization in feedback control. The underlying physical behavior of the system is governed by Newton's second law for rotation,

$$\tau = I \cdot \alpha \tag{59}$$

where τ is the applied torque acting on the system, responsible for producing rotational motion about the vehicle's center of mass, measured in Nm, I is the moment of inertia, representing the resistance of the vehicle to changes in angular motion and depending on the mass distribution relative to the axis of rotation, measured in $\mathrm{kg\,m^2}$, and α is the angular acceleration, defined as the rate of change of angular velocity, measured in $\mathrm{rad/s^2}$. This equation describes how applied torques lead to angular acceleration of the vehicle.

Therefore, the rate controller is responsible for the fast stabilization of the UAV's rotational motion by regulating angular velocity, while the attitude controller determines the desired orientation. This separation improves stability and allows rapid disturbances to be corrected before they significantly affect the vehicle attitude.

5) Rotational Dynamics and Torque: Torque is the rotational equivalent of force. In UAV rotational dynamics, torques determine how the drone rotates around its axes. Although some control approaches explicitly compute torque commands, the drone does not directly actuate torque. Instead, the control system adjusts motor speeds to indirectly generate the required rotational effects. In some model-based control approaches, torque can appear as an intermediate variable used to derive motor commands. The system determines how much rotational acceleration is necessary to reduce the attitude or rate error and translates that requirement into equivalent torques. For a quadcopter, different torque directions are achieved through specific motor speed adjustments:

- Roll control is achieved by differential thrust between left and right motors. By increasing the thrust on one side and decreasing it on the opposite side, a torque is generated due to the offset between the applied force and the center of mass.
- Pitch control is achieved by differential thrust between front and rear motors. Similar to roll, the difference in thrust creates a moment around the corresponding axis due to the lever arm between the motors and the vehicle center of mass.
- Yaw control is achieved through reaction torques generated by the propellers. According to Newton's third law, the motor applies a torque to rotate the propeller, and the propeller applies an equal and opposite torque back onto the motor and the drone body. By using counter-rotating propellers, these reaction torques cancel during steady operation. A difference in motor speeds creates an imbalance in reaction torques, producing a net yaw rotation.

These physical effects ultimately determine the rotational behavior of the UAV.

6) Control Allocation or Motor Mixing: Once the desired total thrust and rotational control commands have been computed, they must be converted into individual motor commands. This process is known as control allocation or motor mixing. The mixer computes the required speed for each motor based on the desired collective thrust and rotational control inputs. This stage is essential because the drone cannot directly command forces or torques. Instead, it can only control the rotational speed of its motors. The mixer determines how each motor must accelerate or decelerate to achieve the desired combination of lift and rotational behavior in roll, pitch, and yaw.

$$\omega_i = f(T, u_{\text{roll}}, u_{\text{pitch}}, u_{\text{yaw}}) \tag{60}$$

In this expression, ω_i denotes the rotational speed command for motor i, T represents the desired collective thrust, and u_{roll}, u_{pitch}, and u_{yaw} are the rotational control commands generated by the attitude rate controller for the roll, pitch, and yaw axes, respectively. The function $f(\cdot)$ represents the motor mixing algorithm that maps these control commands to the individual motor speed commands.

In a quadcopter, the mixer determines how each motor must adjust its speed to produce the desired total thrust and rotational effects around the roll, pitch, and yaw axes. The resulting commands are sent to the Electronic Speed Controllers (ESCs), which regulate the electrical power delivered to the motors.

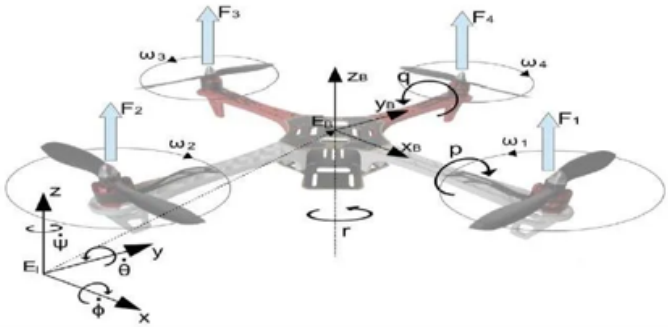


Fig. 27: Motor mixing process converting thrust and rotational commands into individual motor outputs.
Source: [45]

Figure 27 shows the quadrotor configuration and the rotation direction of each propeller. These characteristics determine how individual motor thrusts influence the overall translational and rotational behavior of the vehicle in roll, pitch, and yaw. The motor mixing stage uses this relationship to convert the desired total thrust and rotational control commands into individual motor commands.

7) Key Design Principle of UAV Control Loops: Modern UAV controllers are designed as cascaded multi-rate systems, where higher-level objectives (such as position tracking) are handled at lower frequencies, while lower-level stabilization tasks (such as attitude and angular rate control) operate at higher frequencies. This separation allows the system to simultaneously achieve stability and responsiveness.

8) Importance of Attitude Control in UAV Stability: Attitude control is one of the most critical components of any UAV flight control system. It is responsible for maintaining stability, enabling maneuverability, and ensuring that higher-level navigation objectives can be achieved safely. Because multirotor drones are inherently unstable systems, continuous attitude correction is required throughout the entire flight. Even small orientation errors can quickly grow and destabilize the vehicle if not corrected immediately. The high-speed feedback architecture, combined with accurate sensing, state estimation, and motor control, allows modern UAVs to maintain stable and precise flight behavior under varying conditions. As a result, attitude control forms the foundation upon which all other autonomous drone capabilities are built, including position control, trajectory tracking, obstacle avoidance, and autonomous navigation.

F. PID Controllers in UAV Systems

1) Introduction to PID Control: A PID (Proportional–Integral–Derivative) controller is one of the most widely used feedback control strategies in engineering and robotics. It is

designed to continuously correct the behavior of a dynamic system by comparing a desired reference value (setpoint) with the actual measured output. The difference between these two values is called the error, and the PID controller generates a control signal based on this error to reduce it over time. Mathematically, the control law is defined as:

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt} \tag{61}$$

Each term contributes differently to the overall behavior of the system, allowing the controller to balance responsiveness, accuracy, and stability [46].

2) Why PID Instead of Other Controllers?: In control theory, several alternative approaches exist beyond PID control. However, PID remains dominant in practical UAV systems due to its simplicity and robustness. Other control strategies include:

- **On/Off control**, which operates through simple switching logic and is suitable for basic systems such as thermostats but is too coarse for continuous dynamics like flight control.
- **State feedback and LQR (Linear Quadratic Regulator)**, which provide mathematically optimal solutions but require accurate system models that are often difficult to obtain for real UAV dynamics.
- **Model Predictive Control (MPC)**, which predicts future system behavior over a time horizon but is computationally expensive for high-frequency embedded control loops.
- **Adaptive and fuzzy control**, which can handle uncertainty and changing dynamics but are more complex to design and tune.

Despite the existence of these advanced methods, PID control is widely used because it offers a strong balance between performance and simplicity, especially in real-time embedded systems.

3) Structure of a PID Controller: A PID controller consists of three independent components, each addressing a different aspect of system behavior [47].The gains K_P, K_I, and K_D determine how strongly each PID component influences the control output. A higher gain increases the contribution of the corresponding term, producing a stronger corrective action for the same error signal. However, excessive gains can reduce stability and lead to oscillations or undesirable behavior.

a) Proportional Term (P): The proportional term reacts directly to the current error:

$$u_P = K_P e(t) \tag{62}$$

It provides an immediate correction proportional to the magnitude of the error. A larger error results in stronger corrective action. If the gain is too high, it can also lead to oscillations or instability. The proportional term improves the response speed of the system, but it may not eliminate the error completely. Since the control action is proportional to the current error, it becomes very small as the system approaches the setpoint. In real systems, a small non-zero error is often needed to balance constant disturbances such as gravity, friction, or load effects. This remaining difference is known as steady-state error, and it persists even after the system has settled.

b) Integral Term (I): The integral term accumulates the error over time:

$$u_I = K_I \int_0^t e(\tau)d\tau \tag{63}$$

Its main purpose is to eliminate steady-state error by correcting long-term bias in the system. Even small persistent errors are gradually accumulated and compensated. However, the integral term can slow down the system and may introduce overshoot. It is also associated with a common issue known as integral windup, where the accumulated error becomes excessively large when actuators are saturated.

To mitigate integral windup, practical implementations typically include anti-windup mechanisms. The most common approaches are clamping (limiting the integral term when actuator saturation is reached) or back-calculation methods, which adjust the accumulated integral based on the difference between the computed control signal and the actual saturated output. These strategies prevent excessive error accumulation and help maintain controller stability during saturation conditions.

c) Derivative Term (D): The derivative term is based on the rate of change of the error:

$$u_D = K_D \frac{de(t)}{dt} \tag{64}$$

It provides predictive behavior by reacting to how quickly the error is changing. This helps damp oscillations and improves system stability. However, the derivative term is sensitive to noise in real sensor measurements, which can introduce unwanted high-frequency fluctuations in the control signal. In addition, when the derivative is computed directly on the error signal, a sudden change in the reference input can cause a phenomenon known as *derivative kick*, where the control output exhibits a sharp transient spike. For this reason, practical implementations often compute the derivative based on the measured state rather than the error signal.

4) PID in Control System Design: From a control engineering perspective, the goal of a feedback controller is to ensure that a system remains stable, accurate, and responsive under disturbances and changing conditions. The three PID components contribute to these objectives in different ways:

- The proportional term improves responsiveness.
- The integral term eliminates steady-state error.
- The derivative term enhances stability and damping.

Together, they form a balanced control structure that can handle a wide range of dynamic behaviors without requiring an exact mathematical model of the system. It is also possible to obtain different combinations of the PID (for example PI or PD) by setting the corresponding gain term to zero. The influence of the individual PID gains on the system response is illustrated in Fig. 28.

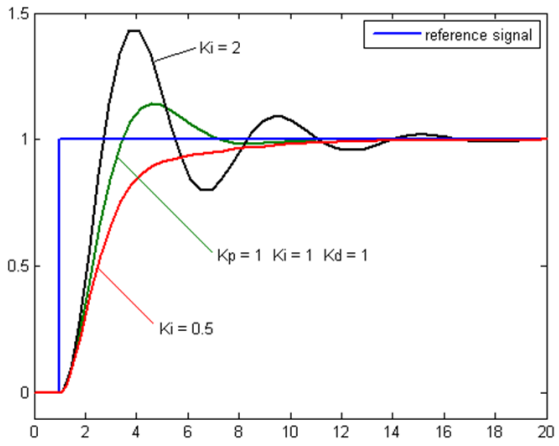


Fig. 28: Effect of different PID gains on system response
Source: [48]

5) Interpretation of the s and $\frac{1}{s}$ Operators in PID Control: The PID controller in the Laplace domain is commonly expressed as

$$G_{PID}(s) = K_p + \frac{K_i}{s} + K_d s \tag{65}$$

The symbols s and $\frac{1}{s}$ used in control theory originate from the Laplace transform, which converts differential equations in the time domain into algebraic equations in the Laplace domain. This transformation greatly simplifies the analysis and design of dynamic systems by replacing differentiation and integration with simple algebraic equations in the Laplace domain [49].

For a time-domain signal $x(t)$ with Laplace transform $X(s)$, the following properties hold under zero initial conditions:

$$\mathcal{L}\left\{\frac{dx(t)}{dt}\right\} = sX(s), \tag{66}$$

$$\mathcal{L}\left\{\int_0^t x(\tau)\,d\tau\right\} = \frac{X(s)}{s}. \tag{67}$$

These expressions show that differentiation in the time domain is represented by multiplication by s, while integration is represented by division by s. Consequently, the symbols s and $\frac{1}{s}$ should not be interpreted as physical quantities, but rather as algebraic representations of these two operations in the Laplace domain [50].

Within a PID controller, the continuous-time control law is expressed as

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt}, \tag{68}$$

where the proportional term depends on the instantaneous error, the integral term accumulates past errors, and the derivative term responds to the rate of change of the error.

Applying the Laplace transform to the control law yields

$$U(s) = \left(K_p + \frac{K_i}{s} + K_d s\right) E(s), \tag{69}$$

from which the PID transfer function is obtained as

$$G_{PID}(s) = \frac{U(s)}{E(s)} = K_p + \frac{K_i}{s} + K_d s. \tag{70}$$

Although the PID controller is commonly analyzed and designed using its Laplace-domain representation, practical digital implementations do not explicitly perform Laplace transforms during operation. Instead, the controller executes discrete-time approximations of the integral and derivative terms using numerical summation and finite differences. Consequently, the s-domain formulation serves primarily as a mathematical framework for controller modeling, analysis, tuning, and stability assessment, while the implemented controller operates entirely in the time domain.

6) Discrete-Time Implementation of the PID Controller: Although the PID controller is defined in continuous time, practical implementations in embedded systems operate in discrete time. This is due to the fact that digital controllers, such as those used in microcontrollers and UAV flight systems, process signals at discrete sampling intervals rather than as continuous functions of time.

Let T_s denote the sampling period of the digital controller, and let $k \in \mathbb{Z}_{\geq 0}$ be the discrete-time index representing the k-th sampling instant. The continuous-time error signal $e(t)$ is therefore sampled at discrete instants $t = kT_s$, leading to the discrete-time signal

$$e[k] = e(kT_s). \tag{71}$$

In this formulation, T_s defines the time interval between two consecutive controller updates, while k simply counts how many sampling periods have elapsed since the start of the control process. The continuous-time PID controller

$$u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \frac{de(t)}{dt} \tag{72}$$

is approximated in discrete time as follows.

The integral term is implemented using a numerical accumulation (rectangular approximation):

$$\int e(t)\,dt \approx \sum_{i=0}^{k} e[i]\,T_s \tag{73}$$

which leads to the recursive form:

$$I[k] = I[k-1] + e[k]\,T_s \tag{74}$$

The formulation indicates that the integral at the current sampling instant, denoted by $I[k]$, is obtained by adding the accumulated integral from the previous sample, $I[k-1]$, to the current error $e[k]$ multiplied by the sampling period T_s. Here, k denotes the discrete sampling index, while T_s represents the time interval between two consecutive controller executions.

The derivative term is approximated using a finite difference:

$$\frac{de(t)}{dt} \approx \frac{e[k] - e[k-1]}{T_s} \tag{75}$$

where $e[k]$ and $e[k-1]$ denote the current and previous sampled values of the error signal, respectively. This approximation estimates the rate of change of the error by dividing the difference between two consecutive samples by the sampling period.

Thus, the discrete-time PID control law becomes:

$$u[k] = K_p e[k] + K_i I[k] + K_d \frac{e[k] - e[k-1]}{T_s} \tag{76}$$

This formulation is widely used in embedded control systems due to its computational efficiency and suitability for real-time execution on microcontrollers. It directly replaces continuous-time integration and differentiation with algebraic operations based on sampled measurements [49].

In UAV systems such as the Crazyflie 2.1 Brushless, this discrete-time implementation is used in the onboard flight controller [44]. In practice, the firmware does not assume a perfectly constant sampling period T_s, but instead computes the actual elapsed time between consecutive control iterations, denoted as $\Delta t = t[k] - t[k-1]$. This value is then used directly in the numerical implementation of the controller, replacing T_s in both the integral and derivative terms.

Consequently, the integral is updated as a time-weighted accumulation,

$$I[k] = I[k-1] + e[k]\Delta t, \tag{77}$$

while the derivative is computed using a finite difference approximation,

$$\frac{de(t)}{dt} = \frac{e[k] - e[k-1]}{\Delta t}. \tag{78}$$

This approach improves numerical accuracy in real-time embedded systems, since it accounts for small variations in execution timing inherent to microcontroller-based control loops.

Despite being implemented in discrete time, the design and tuning of these controllers are typically performed using the Laplace-domain representation, which provides a convenient analytical framework for stability analysis and system design.

7) Why PID is Important in UAV Systems: PID controllers are especially important in UAV applications because they provide a simple yet highly effective method for stabilizing inherently unstable systems such as multirotor drones. Their main advantages include simplicity of implementation, low computational cost, real-time capability for high-frequency control loops, robustness to modeling inaccuracies, and proven performance in real-world systems. These characteristics make PID controllers widely used in UAV flight control systems, particularly for attitude stabilization, motor control, and altitude regulation.

In multirotor drones, PID controllers are commonly applied to several control tasks, including roll, pitch, and yaw stabilization, angular rate control, motor speed regulation, and altitude control. Through these applications, PID controllers play a central role in maintaining stable flight and ensuring smooth responses to user commands or higher-level autopilot instructions.

G. Separate Controllers and Superposition

1) Separate Controllers in UAV Systems:

a) Concept Overview: In most UAV flight control architectures, control is not handled by a single monolithic controller. Instead, the system is decomposed into multiple separate controllers, each responsible for regulating a specific aspect of the drone's motion or state. Typical examples include independent controllers for roll, pitch, and yaw, as well as additional loops for altitude and position control. Each of these controllers computes its own correction signal based on the error between a desired reference value and the current measured state. This modular structure is a fundamental design choice in UAV control systems, allowing complex flight dynamics to be managed in a structured and scalable way.

b) Motivation for Using Separate Controllers: A quadcopter is a highly coupled and nonlinear system that must simultaneously control multiple degrees of freedom, including attitude stabilization (roll, pitch, yaw), angular motion, altitude regulation, and horizontal position control.

Attempting to design a single unified controller that handles all these variables at once would lead to a highly complex system that is difficult to tune, computationally expensive, and often less robust in practice. By separating the problem into smaller control loops, each subsystem becomes significantly easier to analyze, tune, and stabilize. Each controller focuses only on specific dynamic behavior, which simplifies both design and implementation.

c) Control Cascaded Architecture: Most UAVs implement hierarchical or cascaded control architecture, where different control loops operate at different levels of abstraction and frequency. At the highest level, slower outer loops manage global objectives such as position or attitude. At lower levels, faster inner loops handle rapid dynamic stabilization such as attitude rate control. A typical structure can be described as follows:

1) The outer loop receives high-level commands such as desired position or orientation.
2) It computes intermediate references such as desired angles or velocities.
3) The inner loop tracks these references by regulating angular rates and generating rotational control commands.
4) Finally, motor commands are produced to actuate the system.

This hierarchical decomposition allows each layer to operate at an appropriate time scale, improving both stability and responsiveness.

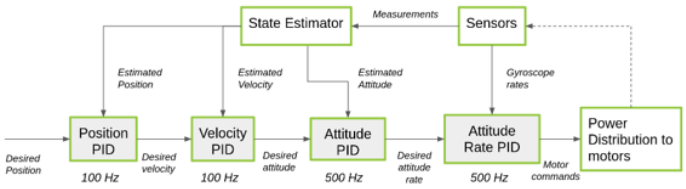


Fig. 29: Hierarchical cascaded control architecture of Crazyflie 2.1 brushless flight controller.
Source: [43]

The cascaded structure shown in Figure 29 illustrates how different control layers are organized hierarchically. Higher-level controllers, such as velocity and position control, operate at lower update frequencies and generate references for faster inner loops. The inner loops, including attitude and attitude rate control, operate at higher frequencies to stabilize the vehicle dynamics and generate motor commands through the motor mixing stage. In many real-world implementations, including platforms such as the Crazyflie 2.1 Brushless, this cascaded control structure is a standard approach for flight control.

In addition to the classical PID-based loops for attitude and rate stabilization, more advanced control strategies are also integrated into the flight stack. In particular, the Crazyflie control architecture can also include the Mellinger controller, which extends the cascaded control approach with a geometric control formulation for accurate trajectory tracking and smooth motion generation, and the INDI (Incremental Nonlinear Dynamic Inversion) controller, which improves robustness by relying on incremental control updates that reduce the dependence on an exact system model and provide better disturbance rejection. Together, these controllers extend the basic cascaded PID structure, allowing the system to achieve both stable low-level flight and high-performance trajectory execution under real-world uncertainties [43].

d) Advantages of Separate Controllers: Using separate controllers provides several important benefits in UAV system design:

- **Modularity:** Each control loop can be designed and analyzed independently.
- **Easier tuning:** PID parameters or control laws can be adjusted per axis or subsystem without affecting the entire system.
- **Improved stability:** Decomposing the system reduces coupling complexity and simplifies stability analysis.
- **Scalability:** Additional control layers (such as position hold or navigation) can be added without redesigning the entire controller.
- **Independent optimization:** Each axis or subsystem (roll, pitch, yaw, altitude) can be optimized individually for performance.

2) Superposition in UAV Control Systems:

a) Concept Overview: In multirotor UAV control systems, different control loops regulate different aspects of the vehicle motion. Instead of directly controlling individual motors, the control system generates higher-level control objectives such as total thrust and rotational control contributions around the roll, pitch, and yaw axes. These contributions are combined in the control allocation or motor mixing stage, which maps the desired thrust and rotational effects into individual motor commands. This process allows multiple control objectives, such as maintaining altitude while simultaneously changing attitude, to be achieved at the same time. The combined control input can be represented conceptually as:

$$u = u_{\text{thrust}} + u_{\text{roll}} + u_{\text{pitch}} + u_{\text{yaw}} \tag{79}$$

This additive structure is implemented in practice by quadrotor motor mixing algorithms, where the desired force and torque contributions are distributed among the individual rotors. The Crazyflie firmware, for example, performs this operation in its Power Distribution module, where thrust, roll, pitch, and yaw control inputs are combined to calculate the required motor outputs [44].

b) Why Superposition Works: The possibility of combining these control contributions comes from the physical relationship between rotor thrusts and the forces and moments generated by the vehicle. Each rotor produces an individual thrust force, and the combination of these forces determines the total thrust and rotational moments acting on the quadrotor.

Because forces and torques can be expressed as the sum of the contributions generated by each rotor, the desired thrust and attitude control inputs can be converted into individual motor commands through a control allocation or mixing matrix. This relationship between rotor forces and vehicle moments is commonly used in quadrotor dynamic models and motor mixing algorithms [51].

In practice, the controller does not directly command individual motors. Instead, it generates desired collective thrust and rotational control efforts around the roll, pitch, and yaw axes. The motor mixer then distributes these commands among the available motors so that the resulting rotor forces generate the required vehicle motion.

For example, increasing the thrust of motors on one side while reducing the thrust of motors on the opposite side creates a torque that produces a roll motion, while maintaining the overall thrust level can preserve altitude. By combining these individual motor contributions, the UAV can perform multiple actions simultaneously, such as climbing while changing attitude. This principle is fundamental for motor mixing, simultaneous attitude and altitude stabilization, and multi-axis motion control.

H. Actuators: Brushed and Brushless Motors

The flight controller's output is a set of motor speed commands. These commands are executed by electric motors that convert electrical power into rotational kinetic energy, driving the propellers. Two motor technologies are common in UAV applications: brushed Direct Current (DC) motors and brushless DC (BLDC) motors.

1) Brushed DC Motors:

a) Operating Principle: A brushed DC motor consists of a wound coil assembly (the *rotor* or *armature*) that rotates within a fixed permanent magnet assembly (the *stator*). Electrical current is delivered to the rotating coils through a *commutator* — a segmented copper ring attached to the rotor shaft — via stationary carbon contacts called *brushes*.

As current flows through a coil, the resulting electromagnetic field interacts with the stator's permanent magnetic field, generating a torque that drives rotation (Lorentz force). The commutator mechanically switches the current direction in each coil at the appropriate rotor position, ensuring that the torque always acts in the same direction and rotation is sustained.

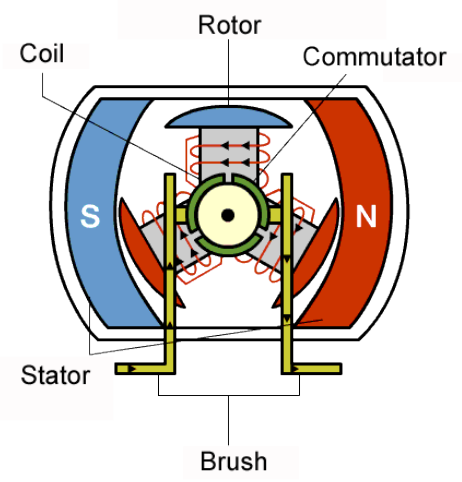


Fig. 30: Cross-section of a brushed DC motor (view along the shaft axis). The stator consists of two fixed permanent-magnet poles (S and N) that create a magnetic field across the air gap. The rotor sits between the poles and carries coils of wire connected to a segmented commutator. Two fixed carbon brushes press against the commutator and supply it with current.
Source: https://botland.de/content/230-die-wahl-des-richtigen-motors-dc-vs-schrittmotor-vs-servo

b) Characteristics and Limitations: Brushed motors are simple and inexpensive to drive — speed control requires only a variable voltage or Pulse-Width Modulation (PWM) signal, where motor speed is regulated by varying the fraction of time the voltage is switched on — making them suitable for small, low-cost platforms such as the Crazyflie. However, the physical contact between brushes and commutator introduces friction, wear, and electrical arcing, limiting lifespan and efficiency, particularly at high rotation rates.

2) Brushless DC (BLDC) Motors:

a) Operating Principle: A brushless DC motor eliminates the commutator and brushes by inverting the conventional motor topology: the permanent magnets are on the *rotor*, and the wound coils are fixed to the *stator*. Current commutation — the switching of current between stator coil phases — is performed electronically by a dedicated motor controller, the *Electronic Speed Controller* (ESC).

The stator holds a set of fixed coils arranged around the rotor. The ESC energises these coils in a specific sequence, creating a rotating electromagnetic field that the permanent-magnet rotor follows. The timing of this switching is determined either by Hall-effect position sensors or, in sensorless designs, by detecting the back-EMF (back Electromotive Force — the voltage generated by the spinning rotor itself, which acts as a signal for the rotor's current position) induced in the un-energised coils.

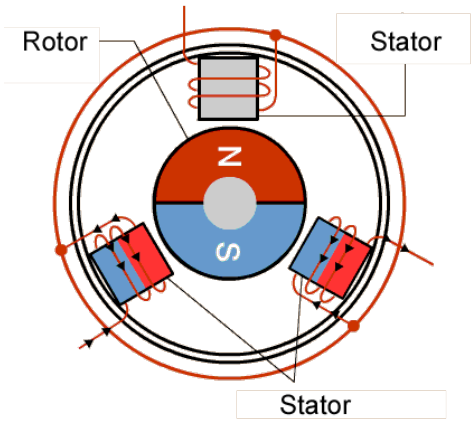


Fig. 31: Cross-section of a brushless outrunner motor (view along the shaft axis). The rotor is a permanent magnet (N/S) at the centre and is free to rotate. The stator is fixed and consists of coil windings arranged around the rotor; the example shown here has three coils, each wired to one phase of the Electronic Speed Controller (ESC). The ESC energises the coils in sequence, producing a rotating magnetic field that the permanent-magnet rotor follows. Because commutation is handled electronically by the ESC rather than mechanically by brushes and a commutator, no sliding contacts are needed.
Source: https://doncenmotor.com/what-are-brushless-dc-motors/

3) Brushed and Brushless Motors on the Crazyflie: The Crazyflie product line offers essentially the same nano-quadrotor airframe in two motor configurations, making the trade-off between the two technologies concrete: the standard Crazyflie 2.1+ uses small brushed coreless motors, while the Crazyflie 2.1 Brushless replaces them with brushless motors and integrated ESCs. The brushed variant keeps the simplicity described in Section IV-H — no dedicated motor controller hardware, low manufacturing cost, and low weight — but this comes at the cost of performance: a flight time of around 7 minutes, and brush wear that limits motor lifespan and continuous-operation reliability.

The brushless variant trades that simplicity for tangible gains: flight time rises to roughly 10 minutes, the maximum recommended payload more than doubles (from about 15 g to 40 g), and the motors run cooler and more smoothly thanks to precise electronic commutation. The downside is a more complex and expensive design, since each motor needs its own

ESC and driver electronics, consuming extra cost and circuit board space.

I. Conclusion

The stability of a UAV in flight is not a passive property of its hardware but an active achievement of tightly coupled subsystems working in concert at high speed. A closed-loop control architecture continuously corrects deviations from the desired state, compensating for disturbances that would cause an uncontrolled vehicle to crash within moments. A diverse sensor suite — gyroscopes, accelerometers, barometers, and position sensors — provides the measurements that make feedback possible, with fusion algorithms combining their complementary characteristics into a reliable state estimate. The Roll-Pitch-Yaw convention provides a common language for describing orientation, while the cascaded attitude control architecture translates orientation errors into angular rate commands and ultimately into individual motor commands through the motor mixing stage. The PID controller, independently instantiated for each controlled axis and combined via superposition, delivers the right balance of simplicity, robustness, and performance for the demanding real-time requirements of UAV stabilisation. Finally, electric motors — whether brushed or brushless — physically execute the computed commands, converting electrical signals into the thrust and torque that keep the vehicle aloft.

Together, these elements answer the central question: drones do not fall because they continuously measure, compute, and correct — running the full control loop at approximately 1000 Hz.

V. Swarm Intelligence

Authors: Marco Schmieg, Hasan Yildiz

Swarm intelligence studies how coordinated group behavior can emerge from simple local rules. For drones, swarm concepts can be used for distributed exploration, formation control, coverage tasks, and cooperative sensing.

A common idea is that each agent follows local interaction rules such as separation, alignment, and cohesion. Similar principles were introduced in early work on flocking behavior [52]. Optimization-based swarm methods, such as particle swarm optimization, also influenced the field [53].

A. Introduction

Swarm intelligence (SI) describes the collective behavior emerging from local interactions among decentralized, autonomous agents that operate without centralized control [54]. Applied to unmanned aerial vehicles (UAVs), SI enables drone fleets to cooperate on complex tasks such as search and rescue, area coverage, and surveillance, adapting in real time to unpredictable and dynamic environments [55], [56]. Compared to single-platform systems, UAV swarms provide inherent fault tolerance, scalability, and parallel task execution, making them increasingly relevant for safety-critical and large-scale missions [57].

B. Role Models from Nature

The principles underlying swarm intelligence are rooted in biological systems that have evolved highly efficient collective behaviors over millions of years [54]. Paradigmatic examples include bird flocking, fish schooling, ant colony foraging, and honeybee communication, each characterized by decentralized decision-making based solely on local information and simple interaction rules [58]. These natural models directly inspired the core algorithmic frameworks applied to drone swarms. Particle Swarm Optimization (PSO), derived from the movement dynamics of birds and fish, Ant Colony Optimization (ACO), modeled on pheromone-based trail reinforcement, and the Artificial Bee Colony (ABC) algorithm, replicating the role-based foraging strategy of honeybees [56], [57].

C. Basic Concepts

Swarm intelligence in UAV systems is built on the principle that complex, coordinated behavior emerges from simple local rules applied independently by each drone, without any centralized authority [54], [57]. Three fundamental capabilities define swarm operation: *formation flight*, which keeps drones at defined relative positions; *scalable expansion*, which allows the swarm to grow or shrink without redesigning the system; and *distributed target search*, in which each drone follows local decision heuristics based on gradient information and neighbor proximity [56]. Each agent perceives only a limited neighborhood, exchanges state information with nearby drones, and updates its behavior accordingly, enabling the collective to adapt to dynamic environments without global knowledge [54], [58].

D. Centralised vs. Decentralised Control

A centralised architecture relies on a ground station or lead drone that maintains global knowledge and dispatches explicit commands to every agent, offering high coordination quality but introducing a critical single point of failure and severely limiting scalability [55], [57]. Decentralised architectures grant each drone full autonomy based solely on local sensor data and communication with neighboring agents, requiring neither a global map nor a central coordinator [54]. The governing phenomenon is *emergence*: individually simple agents collectively exhibit intelligent system-level behavior, analogous to a murmuration of starlings where no single bird directs the flock yet coherent, collision-free motion arises spontaneously [58].

E. Algorithms

Reynolds Boids (1987): Craig W. Reynolds introduced the Boids model based on three minimal local steering rules applied by each agent i with position $\mathbf{x}_i$, velocity $\mathbf{v}_i$, and neighborhood $\mathcal{N}_i = \{j \mid \|\mathbf{x}_j - \mathbf{x}_i\| < r\}$ [59].

Separation prevents collisions by repelling agent i from nearby neighbors:

$$\mathbf{f}_{sep,i} = -\sum_{j \in \mathcal{N}_i} \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|^2} \tag{80}$$

Alignment matches the agent's heading to the average velocity of its neighborhood:

$$\mathbf{f}_{ali,i} = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \mathbf{v}_j - \mathbf{v}_i \tag{81}$$

Cohesion steers the agent toward the local center of mass:

$$\mathbf{f}_{coh,i} = \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \mathbf{x}_j - \mathbf{x}_i \tag{82}$$

The combined steering update is a weighted superposition of all three components:

$$\mathbf{v}_i^{new} = \mathbf{v}_i + w_s\, \mathbf{f}_{sep,i} + w_a\, \mathbf{f}_{ali,i} + w_c\, \mathbf{f}_{coh,i} \tag{83}$$

where w_s, w_a, and w_c are scalar weights tuned per application [59]. These rules remain the foundational model for decentralised formation control in UAV swarms.

Artificial Potential Fields (APF): Introduced by Khatib [60], APF models navigation as movement along the negative gradient of a scalar potential field $U(\mathbf{q})$ composed of an attractive component toward the goal $\mathbf{q}_{goal}$ and a repulsive component from obstacles:

$$U_{att}(\mathbf{q}) = \frac{1}{2}\, \xi\, \|\mathbf{q} - \mathbf{q}_{goal}\|^2 \tag{84}$$

$$U_{rep}(\mathbf{q}) = \begin{cases} \dfrac{1}{2}\, \eta \left(\dfrac{1}{\rho(\mathbf{q})} - \dfrac{1}{\rho_0} \right)^2 & \text{if } \rho(\mathbf{q}) \le \rho_0 \\ 0 & \text{otherwise} \end{cases} \tag{85}$$

where $\rho(\mathbf{q})$ is the distance to the nearest obstacle and ρ_0 is the influence radius. The resulting motion command is:

$$\mathbf{F}(\mathbf{q}) = -\nabla U(\mathbf{q}) = \mathbf{F}_{att}(\mathbf{q}) + \mathbf{F}_{rep}(\mathbf{q}) \tag{86}$$

This force vector can be evaluated in real time onboard each drone, making APF directly applicable to decentralised UAV navigation in cluttered environments [60].

Swarm Gradient Bug Algorithm (SGBA): McGuire et al. developed SGBA specifically for resource-constrained nano-drones that carry no onboard map and operate under strict weight and power budgets [61]. Implemented as a finite state machine with the behaviors *go-to-goal*, *wall-following*, and *inter-agent avoidance*, the algorithm was experimentally validated in an indoor search-and-rescue scenario in which six Crazyflie quadrotors collectively explored an office environment while communicating locally to maximise coverage and avoid collisions [61].

F. Future Prospects

UAV swarm research is rapidly advancing towards AI-driven coordination, replacing handcrafted rules with learned policies. A prominent direction is multi-agent deep reinforcement learning (MARL) combined with transformer-based attention: the HTransRL framework models swarm coordination as a decentralised partially observable Markov decision process (Dec-POMDP) and uses a hybrid transformer architecture to capture inter-agent dependencies, achieving a successful arrival rate exceeding 90 % in worst-case corridor scenarios [62]. Graph Neural Networks (GNNs) combined with transformer-enhanced Double Deep Q-Networks further enable structured inter-agent communication, yielding 90 % goal collection and 100 % area coverage in dynamic environments while remaining lightweight enough for onboard execution [63]. Beyond reinforcement learning, Large Language Model (LLM) integration is emerging as a new coordination paradigm: the SwarmGPT system uses LLM-generated waypoints to produce collision-free drone formations in real time, illustrating the potential of natural language as a high-level swarm command interface [64].

VI. Conclusion

In conclusion, the five topics show how autonomous drones combine sensing, estimation, orientation, control, and cooperation. SLAM and Kalman filtering help the drone understand its position and environment despite noisy sensor data. Orientation representations and PID control make stable flight possible, while swarm intelligence extends these ideas to multiple drones working together. Overall, these methods form the foundation for reliable autonomous drone systems, although limitations such as sensor accuracy, processing power, and communication remain important challenges.

List of Figures

1 Conceptual SLAM and navigation architecture for the Crazyflie platform with optical flow and sparse Time-of-Flight sensing. 7

2 Test environment and autonomous flight sequence. The numbered positions mark the five keyframes. The first and fifth keyframes were recorded at the same physical location. A 360° scan was performed at every keyframe. 8

3 Projected Multi-ranger measurements before and after preprocessing and the accepted loop-closure correction. The corrected result aligns the final keyframe with the initial keyframe and distributes the correction over the recorded trajectory. . . . 8

4 Intuitive illustration of the Kalman filter using Gaussian distributions. The model prediction (blue) and sensor measurement (red) are both represented as normal distributions, each with a mean and variance. The Kalman filter computes their optimal fusion (green), which has a tighter variance and a mean that lies between the two sources, weighted by their relative uncertainty. This visual representation underscores that the filter is fundamentally an optimal combination of imperfect information sources. 10

5 The recursive predict-and-update cycle of the Kalman filter. At each time step the prior estimate is propagated forward via the process model and subsequently corrected by the incoming measurement. 10

6 Illustrative example of the Kalman filter estimating a one-dimensional state from noisy measurements. The Kalman estimate (green) tracks the true state while smoothing the noisy measurements (orange). The shaded band indicates the one-sigma uncertainty $\pm\sqrt{P_k}$ around the estimate, which is initially large and shrinks as measurements accumulate. 12

7 One-dimensional height-estimation example for the Crazyflie, shown across four phases. Starting from the current state, the model produces a prediction, the ToF sensor delivers the noisy measurement z_t with noise r_z, and both are fused into the corrected next state. Adapted from [28]. 12

8 Linear transformation of a Gaussian. An input density $p(x)$ passed through $g(x) = ax + b$ yields a Gaussian output with shifted mean and scaled variance. Adapted from [30]. 13

9 Non-linear transformation of a Gaussian. The same input passed through a non-linear $g(x)$ yields a skewed, non-Gaussian output (black). The EKF linearises g at μ via its Jacobian (red), producing a Gaussian approximation that diverges from the true distribution. Adapted from [30]. . . 13

10 Illustration of the unscented transform in a single plot. Sigma points (orange) are placed deterministically around the prior mean according to the prior covariance $\mathbf{P}_{k-1}$ (green ellipse). After propagation through the non-linear process function, indicated by the central arrow, the weighted sigma points reconstruct the predicted distribution (cyan ellipse), capturing the shifted mean and the rotated, enlarged spread that a linearisation-based approach would miss. 14

11 Block diagram of the complementary filter on the Crazyflie. Gyroscope, accelerometer, and time-of-flight inputs are fused into attitude and altitude estimates without estimating horizontal position. Source: [35]. 15

12 Block diagram of the Extended Kalman Filter on the Crazyflie. Internal sensors together with the Flow Deck, Loco Positioning Deck, Lighthouse Deck, and motion-capture inputs are fused into attitude, position, and velocity estimates. Source: [35]. 15

13 Crazyflie 2.1 Brushless demonstration: raw optical-flow measurement versus EKF-filtered prediction for the x- and y-axis during a hover flight. 17

14 A 2D vector rotation by $60°$. The rotated vector is obtained by multiplying the original vector with $R_{60°}$. 18

15 A quaternion can be constructed from a rotation axis and a rotation angle. For $\mathbf{v}_1 = (1, 0, 0)^T$ and $\mathbf{v}_2 = (0, 1, 0)^T$, the axis is $\mathbf{a} = (0, 0, 1)^T$ and the angle is $90°$. 20

16 All common 3D orientation representations describe the same physical rotation and can be converted into one another. 21

17 Block diagram of an open-loop control system. The output is not fed back to the controller. . . . 22

18 Block diagram of a closed-loop control system. The measured output is subtracted from the reference to form an error signal that drives the controller. 22

19 Closed-loop control architecture of the Crazyflie quadrotor. Sensor measurements are fed back to the firmware controller, which continuously adjusts motor speeds to minimise the error between the desired and actual state. 22

20 Schematic cross-section of a MEMS gyroscope. The oscillating proof mass experiences a Coriolis-induced displacement perpendicular to its drive motion when an angular rate is applied; this displacement is read out as a change in capacitance. 23

21 Schematic cross-section of a MEMS accelerometer. 23

22 The Bosch BMI088 used in Crazyflie 2.1 Brushless 24

23 Cross-section of a MEMS capacitive barometer. Ambient pressure deflects the silicon diaphragm toward the fixed electrode, reducing the plate separation d and increasing the measured capacitance C. Because atmospheric pressure falls predictably with altitude, C provides an indirect altitude measurement. 24

24 Position sensing modalities. *Left:* GPS trilateration — the receiver at P computes its position from the ranges r_1, r_2, r_3 to satellites with known orbits. *Right:* Optical flow — a downward camera tracks the shift $\Delta\mathbf{p}$ of ground-texture features between successive frames to estimate horizontal velocity. 25

25 Roll, pitch, and yaw angles describing the attitude of a multirotor UAV. 26

26 Simplified cascaded control architecture for a multirotor UAV. 27

27 Motor mixing process converting thrust and rotational commands into individual motor outputs. . 28

28 Effect of different PID gains on system response 30

29 Hierarchical cascaded control architecture of Crazyflie 2.1 brushless flight controller. 32

30 Cross-section of a brushed DC motor (view along the shaft axis). The stator consists of two fixed permanent-magnet poles (S and N) that create a magnetic field across the air gap. The rotor sits between the poles and carries coils of wire connected to a segmented commutator. Two fixed carbon brushes press against the commutator and supply it with current. 33

31 Cross-section of a brushless outrunner motor (view along the shaft axis). The rotor is a permanent magnet (N/S) at the centre and is free to rotate. The stator is fixed and consists of coil windings arranged around the rotor; the example shown here has three coils, each wired to one phase of the Electronic Speed Controller (ESC). The ESC energises the coils in sequence, producing a rotating magnetic field that the permanent-magnet rotor follows. Because commutation is handled electronically by the ESC rather than mechanically by brushes and a commutator, no sliding contacts are needed. 33

References

[1] H. Durrant-Whyte and T. Bailey, "Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms,"

[2] C. Cadena, L. Carlone, H. Carrillo, *et al.*, "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016, ISSN: 1552-3098, 1941-0468. DOI: 10.1109/TRO.2016.2624754. [Online]. Available: http://ieeexplore.ieee.org/document/7747236/ (visited on 06/01/2026).

[3] C. Stachniss, J. J. Leonard, and S. Thrun, "Simultaneous Localization and Mapping," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds., Cham: Springer International Publishing, 2016, pp. 1153–1176, ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_46. [Online]. Available: https://doi.org/10.1007/978-3-319-32552-1_46 (visited on 06/01/2026).

[4] D. M. Rosen, K. J. Doherty, A. T. Espinoza, and J. J. Leonard, "Advances in Inference and Representation for Simultaneous Localization and Mapping," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 4, pp. 215–242, Volume 4, 2021 May 3, 2021, ISSN: 2573-5144. DOI: 10.1146/annurev-control-072720-082553. [Online]. Available: https://www.annualreviews.org/content/journals/10.1146/annurev-control-072720-082553 (visited on 06/01/2026).

[5] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "An Updated Tutorial on Graph-Based SLAM," in *Robotics Goes MOOC: Knowledge*, B. Siciliano, Ed., Cham: Springer Nature Switzerland, 2025, pp. 255–285, ISBN: 978-3-319-74095-9. DOI: 10.1007/978-3-319-74095-9_8. [Online]. Available: https://doi.org/10.1007/978-3-319-74095-9_8 (visited on 06/01/2026).

[6] C. Chen, H. Zhu, M. Li, and S. You, "A Review of Visual-Inertial Simultaneous Localization and Mapping from Filtering-Based and Optimization-Based Perspectives," *Robotics*, vol. 7, no. 3, p. 45, Sep. 2018, ISSN: 2218-6581. DOI: 10.3390/robotics7030045. [Online]. Available: https://www.mdpi.com/2218-6581/7/3/45 (visited on 06/01/2026).

[7] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1271–1278. DOI: 10.1109/ICRA.2016.7487258. [Online]. Available: https://ieeexplore.ieee.org/document/7487258 (visited on 06/01/2026).

[8] G. Grisetti, C. Stachniss, and W. Burgard, "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, Feb. 2007, ISSN: 1552-3098. DOI: 10.1109/TRO.2006.889486. [Online]. Available: http://ieeexplore.ieee.org/document/4084563/ (visited on 06/01/2026).

[9] J. A. Castellanos, J. Neira, and J. D. Tardós, "Limits to the consistency of EKF-based SLAM," *IFAC Proceedings Volumes*, IFAC/EURON Symposium on Intelligent Autonomous Vehicles, Lisbon, Portugal, 5-7 July 2004, vol. 37, no. 8, pp. 716–721, Jul. 1, 2004, ISSN: 1474-6670. DOI: 10.1016/S1474-6670(17)32063-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017320633 (visited on 06/01/2026).

[10] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte Carlo localization for mobile robots," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 2, Detroit, MI, USA: IEEE, 1999, pp. 1322–1328, ISBN: 978-0-7803-5180-6. DOI: 10.1109/ROBOT.1999.772544. [Online]. Available: http://ieeexplore.ieee.org/document/772544/ (visited on 06/01/2026).

[11] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010. DOI: 10.1109/MITS.2010.939925.

[12] S. Huang and G. Dissanayake, "A critique of current developments in simultaneous localization and mapping," *International Journal of Advanced Robotic Systems*, vol. 13, no. 5, p. 1 729 881 416 669 482, Sep. 1, 2016, ISSN: 1729-8806. DOI: 10.1177/1729881416669482. [Online]. Available: https://doi.org/10.1177/1729881416669482 (visited on 06/01/2026).

[13] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, Mar. 1997, ISSN: 10709932. DOI: 10.1109/100.580977. [Online]. Available: http://ieeexplore.ieee.org/document/580977/ (visited on 07/02/2026).

[14] B. Brito, B. Floor, L. Ferranti, and J. Alonso-Mora, "Model Predictive Contouring Control for Collision Avoidance in Unstructured Dynamic Environments," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 4459–4466, Oct. 2019, ISSN: 2377-3766. DOI: 10.1109/LRA.2019.2929976. [Online]. Available: https://ieeexplore.ieee.org/document/8768044 (visited on 07/02/2026).

[15] S. Karam, F. Nex, O. Karlsson, *et al.*, "MICRO AND MACRO QUADCOPTER DRONES FOR INDOOR

MAPPING TO SUPPORT DISASTER MANAGEMENT," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. V-1-2022, pp. 203–210, May 17, 2022, ISSN: 2194-9042. DOI: 10.5194/isprs-annals-V-1-2022-203-2022. [Online]. Available: https://isprs-annals.copernicus.org/articles/V-1-2022/203/2022/isprs-annals-V-1-2022-203-2022.html (visited on 06/01/2026).

[16] R. Eyvazpour, M. Shoaran, and G. Karimian, "Hardware implementation of SLAM algorithms: A survey on implementation approaches and platforms," *Artificial Intelligence Review*, vol. 56, no. 7, pp. 6187–6239, Jul. 1, 2023, ISSN: 1573-7462. DOI: 10.1007/s10462-022-10310-5. [Online]. Available: https://doi.org/10.1007/s10462-022-10310-5 (visited on 06/01/2026).

[17] Bitcraze AB, *Crazyflie 2.1 brushless*, https://www.bitcraze.io/products/crazyflie-2-1-brushless/, Accessed: 2026-06-30.

[18] Bitcraze AB, *Flow deck v2*, https://www.bitcraze.io/products/flow-deck-v2/, Accessed: 2026-06-30.

[19] Bitcraze AB, *Multi-ranger deck*, https://www.bitcraze.io/products/multi-ranger-deck/, Accessed: 2026-06-30.

[20] Bitcraze AB, *Crazyflie python library*, https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/, Accessed: 2026-06-30.

[21] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[22] G. L. Smith, S. F. Schmidt, and L. A. McGee, "Application of statistical filter theory to the optimal estimation of position and velocity on board a circumlunar vehicle," National Aeronautics and Space Administration, Washington, D.C., Tech. Rep. NASA TR R-135, 1962.

[23] M. S. Grewal and A. P. Andrews, "Applications of Kalman filtering in aerospace 1960 to the present," *IEEE Control Systems Magazine*, vol. 30, no. 3, pp. 69–78, 2010.

[24] M. R. Ananthasayanam, M. S. Mohan, N. Naik, and R. M. O. Gemson, "A heuristic reference recursive recipe for adaptively tuning the Kalman filter statistics part-1: Formulation and simulation studies," *Sādhanā*, vol. 41, no. 12, pp. 1473–1490, 2016. DOI: 10.1007/s12046-016-0562-z.

[25] V. Bonato, R. Peron, D. F. Wolf, J. A. M. De Holanda, E. Marques, and J. M. P. Cardoso, "An FPGA implementation for a Kalman filter with application to mobile robotics," in *2007 Symposium on Industrial Embedded Systems (SIES)*, 2007, pp. 148–155. DOI: 10.1109/SIES.2007.4297329.

[26] P. T. L. Pereira, G. Paim, P. U. L. D. Costa, E. A. C. D. Costa, S. J. M. De Almeida, and S. Bampi, "Architectural exploration for energy-efficient fixed-point Kalman filter VLSI design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 7, pp. 1402–1415, 2021. DOI: 10.1109/TVLSI.2021.3075379.

[27] M. B. Rhudy, R. A. Salguero, and K. Holappa, "A Kalman filtering tutorial for undergraduate students," *International Journal of Computer Science & Engineering Survey (IJCSES)*, vol. 8, no. 1, pp. 1–18, 2017. DOI: 10.5121/ijcses.2017.8101.

[28] K. McGuire, *Explaining Kalman filters with the Crazyflie*, Bitcraze Blog, 2024. [Online]. Available: https://www.bitcraze.io/2024/01/explaining-kalman-filters-with-the-crazyflie/.

[29] N. B. F. Da Silva, D. B. Wilson, and K. R. L. J. Branco, "Performance evaluation of the Extended Kalman Filter and Unscented Kalman Filter," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 733–741. DOI: 10.1109/ICUAS.2015.7152356.

[30] C. Stachniss, *Kalman filter and extended kalman filter*, Accessed: 2026-07-05, University of Bonn, Photogrammetry & Robotics Lab, 2021. [Online]. Available: https://www.ipb.uni-bonn.de/html/teaching/photo12-2021/2021-pho2-14-ekf.pptx.pdf.

[31] R. Van Der Merwe and E. A. Wan, "The square-root unscented Kalman filter for state and parameter-estimation," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2001, pp. 3461–3464.

[32] M. Rhudy, Y. Gu, J. Gross, S. Gururajan, and M. R. Napolitano, "Sensitivity analysis of extended and unscented Kalman filters for attitude estimation," *Journal of Aerospace Computing, Information and Communication*, vol. 10, no. 3, pp. 131–143, 2013. DOI: 10.2514/1.54899.

[33] Bitcraze AB. "Mm_flow.c — optical-flow measurement model, crazyflie firmware." Commit d96d2ab, Bitcraze AB. (2026), [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/d96d2abd2905ac5d32056170b1e79ceacf521daf/src/modules/src/kalman_core/mm_flow.c (visited on 07/02/2026).

[34] Bitcraze AB. "Mm_tof.c — time-of-flight measurement model, crazyflie firmware." Commit d96d2ab, Bitcraze AB. (2026), [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/d96d2abd2905ac5d32056170b1e79ceacf521daf/src/modules/src/kalman_core/mm_tof.c (visited on 07/02/2026).

[35] Bitcraze, *State estimation — crazyflie firmware documentation*, Bitcraze Documentation, Accessed: 2026-06-28, 2024. [Online]. Available: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/state_estimators/.

[36] Bitcraze AB. "Estimator_kalman.c — crazyflie firmware source code." Commit d96d2ab, Bitcraze AB. (2026), [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/d96d2abd2905ac5d32056170b1e79ceacf521daf/src/modules/src/estimator/estimator_kalman.c (visited on 07/02/2026).

[37] Bitcraze AB. "Kalman_core.c — crazyflie firmware source code." Commit d96d2ab,

Bitcraze AB. (2026), [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/d96d2abd2905ac5d32056170b1e79ceacf521daf/src/modules/src/kalman_core/kalman_core.c (visited on 07/02/2026).

[38] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 7th. Pearson, 2015.

[39] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008. [Online]. Available: https://www.cds.caltech.edu/~murray/amwiki.

[40] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960. DOI: 10.1115/1.3662552.

[41] M. Guillén, S. García, R. Barea, *et al.*, "A multisensorial simultaneous localization and mapping (SLAM) system for low-cost micro aerial vehicles in GPS-denied environments," *Sensors*, vol. 17, no. 4, 2017.

[42] A. Janota, V. Šimák, D. Nemec, and J. Hrbček, "Improving the precision and speed of Euler angles computation from low-cost rotation sensor data," *Sensors*, vol. 15, no. 3, pp. 7016–7039, 2015. DOI: 10.3390/s150307016.

[43] Bitcraze AB. "Controllers in the Crazyflie," Bitcraze AB. (), [Online]. Available: https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/controllers/ (visited on 07/05/2026).

[44] Bitcraze AB. "Power_distribution_quadrotor.c — Crazyflie firmware source code," Bitcraze AB. (), [Online]. Available: https://github.com/bitcraze/crazyflie-firmware (visited on 07/05/2026).

[45] A. Bousbaine, I. K. Ibraheem, A. J. Humaidi, *et al.*, "Design and implementation of a robust 6-DOF quadrotor controller based on Kalman filter for position control," in *Mobile Robot: Motion Control and Path Planning*, ser. Studies in Computational Intelligence, vol. 1090, Cham: Springer, 2023, pp. 193–214.

[46] M. J. Willis, "Proportional-integral-derivative control," Department of Chemical and Process Engineering, University of Newcastle, Tech. Rep., 1999.

[47] National Instruments. "The PID controller & theory explained," National Instruments. (), [Online]. Available: https://www.ni.com/en/shop/labview/pid-theory-explained.html (visited on 07/05/2026).

[48] A. Al-Maliki and K. Iqbal, "FLC-based PID controller tuning for sensorless speed control of DC motor," in *2018 International Conference on Information and Technology (ICIT)*, 2018, pp. 169–174. DOI: 10.1109/ICIT.2018.8352171.

[49] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd ed. Reading, MA: Addison-Wesley, 1994.

[50] MIT OpenCourseWare, *Laplace transform: Solving initial value problems*, Accessed: 2026-07-06, 2011. [Online]. Available: https://ocw.mit.edu/courses/18-03sc-differential-equations-fall-2011/pages/unit-iii-fourier-series-and-laplace-transform/laplace-transform-solving-initial-value-problems/.

[51] S. Bouabdallah, P. Murrieri, and R. Siegwart, "Design and control of an indoor micro quadrotor," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 5, 2004, pp. 4393–4398.

[52] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 1987, pp. 25–34. DOI: 10.1145/37401.37406.

[53] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948. DOI: 10.1109/ICNN.1995.488968.

[54] I. D. Couzin, "Collective intelligence in animals and robots," *Nature Communications*, vol. 16, no. 1, p. 9574, 2025. DOI: 10.1038/s41467-025-65814-9.

[55] A. Phadke, F. A. Medrano, C. N. Sekharan, and T. Chu, "An analysis of trends in UAV swarm implementations in current research: Simulation versus hardware," *Drone Systems and Applications*, vol. 12, 2024. DOI: 10.1139/dsa-2023-0099.

[56] A. A. Kareem, A. J. Abid, D. A. Hammood, *et al.*, "A bio-inspired swarm UAV framework integrating thermal sensing and optimization-based coordination for efficient search and rescue operations," *Scientific Reports*, vol. 15, no. 1, p. 3361, 2025. DOI: 10.1038/s41598-025-33223-z.

[57] M. A. Ali, A. Maqsood, U. Athar, and S. Sial, "Benchmarking control strategies for UAV swarms: Centralized, decentralized, or federated reinforcement learning," *Aerospace Science and Technology*, vol. 170, p. 111 539, 2026. DOI: 10.1016/j.ast.2025.111539.

[58] A. Nitti, M. D. de Tullio, I. Federico, and G. Carbone, "A collective intelligence model for swarm robotics applications," *Nature Communications*, vol. 16, no. 1, p. 6572, 2025. DOI: 10.1038/s41467-025-61985-7.

[59] C. W. Reynolds, "Flocks, herds, and schools: A distributed behavioral model," 4, vol. 21, 1987, pp. 25–34. DOI: 10.1145/37402.37406.

[60] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986. DOI: 10.1177/027836498600500106.

[61] K. N. McGuire, C. de Wagter, K. Tuyls, H. J. Kappen, and G. C. H. E. de Croon, "Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment," *Science Robotics*, vol. 4, no. 35, eaaw9710, 2019. DOI: 10.1126/scirobotics.aaw9710.

[62] L. Yu, Z. Li, N. Ansari, and X. Sun, "Hybrid transformer based multi-agent reinforcement learning for multiple unpiloted aerial vehicle coordination in air corridors," *IEEE Transactions on Mobile Computing*, vol. 24, no. 6, pp. 5482–5495, 2025. DOI: 10.1109/TMC.2025.3532204.

[63] M. Elrod, N. Mehrabi, R. Amin, *et al.*, "Graph based deep reinforcement learning aided by transformers for multi-agent cooperation," vol. 2504.08195, 2025. DOI: 10.48550/arXiv.2504.08195.

[64] C. Jimenez-Romero, A. Yegenoglu, and C. Blum, "Multi-agent systems powered by large language models: Applications in swarm intelligence," *arXiv*, vol. 2503.03800, 2025. DOI: 10.48550/arXiv.2503.03800.