## Chapter Outline

# Key Tools and Framework Interfaces

**Chapter 8 Multimodal Perception and Intelligent Decision-Making**


8.1 Image Processing Fundamentals

8.2 Recognition, Tracking, Optical Flow

8.3 Localization, Mapping, and Planning

8.4 Common Perception Sensors

8.5 Coordinate Transformations

8.6 Fusion and Control Decision-Making

**8.7 Tools and Framework Interfaces**

8.8 Experiments and Real-World Transfer

8.9 Exercises and Classroom Discussion


## 5 Units in This Section

**8.7.1 Core Development Tools**
Textbook: Sec. 8.1.3.1

**8.7.2 MATLAB/Simulink Algorithm Libraries**
Textbook: Sec. 8.1.3.2

**8.7.3 Common Open-Source Algorithm Libraries**
Textbook: Sec. 8.1.3.3

**8.7.4 Simulation Development Framework**
Textbook: Sec. 8.2.1

**8.7.5 Common Interfaces and Custom Development**
Textbook: Sec. 8.2.3

## 8.7.1 Core Development Tools: Tool Roles

# Different Tools Serve Different Layers of a Task

| Layer | Responsibility | Checkable Output |
| --- | --- | --- |
| **Runtime and communication** | Ubuntu/Linux: processes. Robot Operating System (ROS): messages. | Running nodes, topics, and data timestamps |
| **Algorithm implementation** | Python/C++, OpenCV, and deep learning frameworks | Features, target data, model weights, inference outputs |
| **Computation and reproducibility** | Numerical libraries, Ceres, Git Solving, analysis, version records | Optimal parameters, curves, traceable experiment settings |

## 8.7.1 Core Development Tools: Ubuntu and ROS

# The OS Runs Programs and Middleware Connects Modules




**Ubuntu/Linux:** an operating system (OS) runs processes and manages drivers, files, and networking. Match algorithm dependencies to its version.

**Robot Operating System (ROS):** nodes, topics, and messages connect modules. TF represents frame transforms. Messages carry sample times.

**Python interfaces:** rospy for ROS 1, rclpy for ROS 2. Each version needs compatible nodes and messages.

### 8.7.1 Core Development Tools: Languages and OpenCV

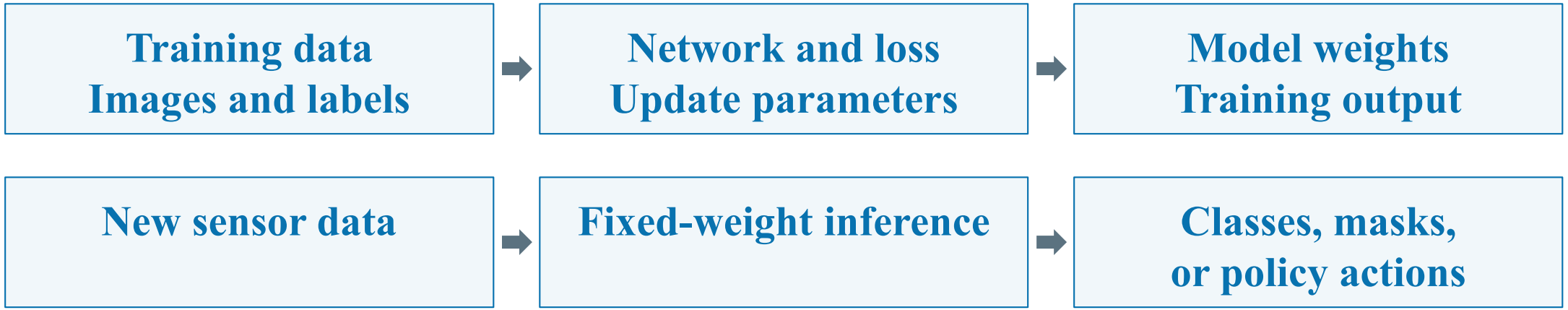
# Shared Data Conventions Connect Prototypes to Deployment

| Tool | Typical Input and Output | Integration |
|---|---|---|
| **Python** | Images, arrays, models<br>→ Prototypes and trained models | Platform Python interfaces<br>Rapid trials and integration |
| **C/C++** | Same physical data, parameters<br>→ Target runtime program | Linked libraries and ROS nodes<br>Memory / concurrency control |
| **OpenCV** | Images → filtering, features,<br>boxes, or optical flow | Python/C++: check shape.<br>BGR: blue, green, red. |

**Latency depends on algorithms, backends, and data transfer.**

### 8.7.1 Core Development Tools: Deep Learning Frameworks

## Training Produces Weights and Inference Produces Outputs




**PyTorch / TensorFlow:** network modeling, automatic differentiation, and device computation. Training and online inference use different dataflows.

**Example:** train on labeled ring images. Feed new frames to the fixed-weight detector for boxes, classes, and confidence.

### 8.7.1 Core Development Tools: Numerical Computing

## Arrays, Solvers, and Plots Answer Different Questions

| Tool | Input | Example Output |
|---|---|---|
| **NumPy** | Pixel, state, or trajectory arrays | Matrix operations and statistics |
| **SciPy** | Observations, costs, signals | Optimal / filtered results |
| **Matplotlib** | Time and numerical sequences | Errors, trajectories, heatmaps |

Horizontal errors: 3, −4, 0 px. Root mean square error (RMSE) retains magnitude.

$$\mathrm{RMSE} = \sqrt{\frac{3^2 + (-4)^2 + 0^2}{3}} \approx 2.89\ \mathrm{px}$$

### 8.7.1 Core Development Tools: Ceres Solver

# Observation Residuals Determine the Best Parameters

Ceres Solver: nonlinear least-squares optimization reduces residual cost.
A residual is prediction minus observation. Here $x$ is position, observed at 2 m and 4 m.
Uses include bundle adjustment, pose graphs, calibration, and trajectory smoothing.

$$J(x) = (x-2)^2 + (x-4)^2, \qquad x \text{ in m}$$

$$\frac{\mathrm{d}J}{\mathrm{d}x} = 4x - 12 = 0 \quad \Rightarrow \quad x^* = 3 \text{ m}$$

Squared-residual sum: 10 m² at initial 1 m, 2 m² at optimum 3 m.
Optimization reduces observation conflict. Credibility still depends on measurement quality.

Ceres residuals, parameter blocks, and bundle adjustment
https://ceres-solver.readthedocs.io/latest/nnls_tutorial.html

### 8.7.1 Core Development Tools: Git and GitHub

# An Experiment Needs a Reproducible Configuration

| Record | Contents | Reproduction Question |
|---|---|---|
| **Code and dependencies** | Git commit, library versions, Python/ROS, target architecture | Can the same runtime be built? |
| **Data and models** | Data split, random seed, weight file, and hash | Is the trained model identical? |
| **Task and results** | Config, task input, timestamps, metrics, and logs | Are operating conditions comparable? |

Git records code versions. GitHub hosts repositories and supports collaboration.
A hash summarizes file content to check that weights and data match.

## 8.7.1 Core Development Tools: Latency Calculation

# Total Frame Time Determines Control Timeliness

| Stage | Given Time | Output |
| --- | --- | --- |
| **Capture and preprocessing** | 6 + 4 ms | Current frame and image tensor |
| **Inference and planning** | 18 + 7 ms | Target data and trajectory |
| **Send control** | 2 ms | Control command |

$$T_{\text{total}} = 37\ \text{ms} > T_{30\text{Hz}} = \frac{1000}{30} \approx 33.3\ \text{ms}$$

Serial: about 27 frames/s. A sustained 30 frames/s input introduces waiting.

## Chapter Outline

# Key Tools and Framework Interfaces

**Chapter 8 Multimodal Perception and Intelligent Decision-Making**

8.1 Image Processing Fundamentals

8.2 Recognition, Tracking, Optical Flow

8.3 Localization, Mapping, and Planning

8.4 Common Perception Sensors

8.5 Coordinate Transformations

8.6 Fusion and Control Decision-Making

**8.7 Tools and Framework Interfaces**

8.8 Experiments and Real-World Transfer

8.9 Exercises and Classroom Discussion

## 5 Units in This Section

**8.7.1 Core Development Tools**
Textbook: Sec. 8.1.3.1

**8.7.2 MATLAB/Simulink Algorithm Libraries**
Textbook: Sec. 8.1.3.2

**8.7.3 Common Open-Source Algorithm Libraries**
Textbook: Sec. 8.1.3.3

**8.7.4 Simulation Development Framework**
Textbook: Sec. 8.2.1

**8.7.5 Common Interfaces and Custom Development**
Textbook: Sec. 8.2.3

## 8.7.2 MATLAB/Simulink Algorithm Libraries: Vision and Robotics Interfaces

# The Front End Extracts Observations for the System

| Toolbox | Responsibility | Minimal Input → Output |
|---|---|---|
| **Computer Vision** | Detection, features, image geometry, and object tracking | Images → boxes, features, or tracks |
| **Robotics System** | Robot models, kinematics, and robotics algorithms | Model and joint states → poses and motion |
| **ROS Toolbox** | Messages, ROS nodes, and model integration | Results → ROS messages<br>Subscriptions → model inputs |

### 8.7.2 MATLAB/Simulink Algorithm Libraries: UAVs and Navigation

# Flight Tasks and Map Planning Work Together

**Flight task**
**Select target ring**
➡
**Navigation map**
**Constrain free space**
➡
**Trajectory / waypoints**
**To controller**

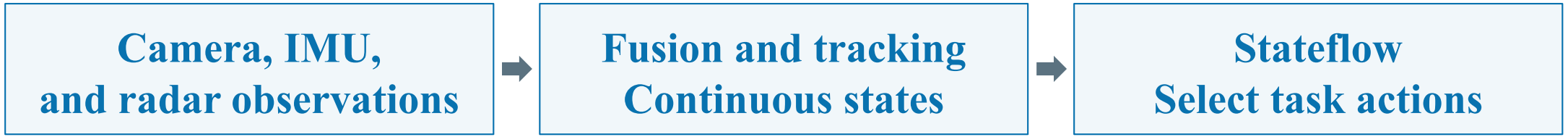
**UAV Toolbox:** UAV missions, flight scenes, and simulation. Organizes vehicles, waypoints, and mission references.

**Navigation Toolbox:** maps, path search, trajectory optimization, and cost evaluation. Produces feasible paths or trajectories.

**Example:** use the obstacle map, current pose, and target ring to plan a detour. The flight controller tracks the trajectory.

## 8.7.2 MATLAB/Simulink Algorithm Libraries: Fusion and Stateflow

# Continuous State Estimation and Discrete Task Switching




**Sensor Fusion and Tracking Toolbox:** multiple sensors and motion models estimate vehicle / target states.
IMU = inertial measurement unit.

**Stateflow:** states, events, conditions, and actions describe task sequences for takeoff, exploration, avoidance, and landing.

**Example:** switch from forward motion to avoidance below 2 m. Resume when clearance recovers and the path is feasible.

## 8.7.2 MATLAB/Simulink Algorithm Libraries: Embedded Coder

# Generated Code Still Needs Target Integration

**Supported Simulink model** → **C/C++ code Compile and link** → **ROS node or target executable**

**Embedded Coder:** generates C/C++ from supported models. Match the target compiler, libraries, and interfaces.

**Dynamic-link library (DLL):** functions called by a host program at runtime. A motion-model DLL can connect to CopterSim.

**On-board evidence:** fresh data, action outputs, and met deadlines make deployment testable.

**Generated code saves implementation work; keep interface meaning consistent.**

### 8.7.2 MATLAB/Simulink Algorithm Libraries: Reinforcement Learning and MPC

## Learned Policies and Online Optimization Produce Actions

| Method | Inputs and Action Generation | Key Checks |
|---|---|---|
| **Reinforcement Learning Toolbox** | Observations, actions, rewards<br>Train a policy, then infer online | Reward design, training coverage, new conditions |
| **Model Predictive Control Toolbox** | State, reference, prediction model → constrained actions, optimized online | Model error, feasible constraints, solve time |

Model predictive control (MPC): predicts future states to choose the current action.
Reinforcement learning (RL): learns an action policy from interaction rewards.

### 8.7.2 MATLAB/Simulink Algorithm Libraries: Model-Based Closed-Loop Example

# The Controller Computes Speed After a Task Switch

Given: obstacle at 1.6 m, switching threshold 2 m. Enter avoidance.
Current position 0 m, reference 1 m after 1 s. Avoidance speed limit: 0.5 m/s.

$$J(v) = (v\Delta t - 1)^2 + 0.2v^2, \qquad \Delta t = 1\ \mathrm{s}$$

| Candidate Speed | Cost | Constraint and Result |
|---|---|---|
| **0.5 m/s** | 0.30 | Within limit, executable |
| **1.0 m/s** | 0.20 | Exceeds limit, reject |

Task logic sets goals and constraints. Optimization chooses a feasible action.

Chapter Outline

# Key Tools and Framework Interfaces

**Chapter 8 Multimodal Perception and Intelligent Decision-Making**

8.1 Image Processing Fundamentals

8.2 Recognition, Tracking, Optical Flow

8.3 Localization, Mapping, and Planning

8.4 Common Perception Sensors

8.5 Coordinate Transformations

8.6 Fusion and Control Decision-Making

**8.7 Tools and Framework Interfaces**

8.8 Experiments and Real-World Transfer

8.9 Exercises and Classroom Discussion

## 5 Units in This Section

**8.7.1 Core Development Tools**
Textbook: Sec. 8.1.3.1

**8.7.2 MATLAB/Simulink Algorithm Libraries**
Textbook: Sec. 8.1.3.2

**8.7.3 Common Open-Source Algorithm Libraries**
Textbook: Sec. 8.1.3.3

**8.7.4 Simulation Development Framework**
Textbook: Sec. 8.2.1

**8.7.5 Common Interfaces and Custom Development**
Textbook: Sec. 8.2.3

## 8.7.3 Common Open-Source Algorithm Libraries: Algorithm Roles

# Detection, Tracking, Localization, and Planning Outputs

| Task | Textbook Algorithms | Key Output |
|---|---|---|
| **Detection and identity** | YOLO / YOLOE<br>DeepSORT | Boxes, classes, masks<br>Object identities across frames |
| **Self-localization and mapping** | VINS-Fusion, ORB-SLAM<br>FAST-LIO series | Vehicle pose, trajectory, and different map forms |
| **Motion planning** | FASTER, EGO-Planner<br>EGO-Swarm | Feasible trajectories, timing, neighbor-avoidance references |

### 8.7.3 Common Open-Source Algorithm Libraries: YOLO and YOLOE

# Class Definitions Determine How to Use a Detector

| Model Family | Input Convention | Output and Use |
|---|---|---|
| **YOLOv5 / YOLOv8**<br>**YOLO11** | Image + fixed weights<br>Training defines the classes | Classes, confidence, boxes<br>Optional masks or keypoints |
| **YOLOE** | Image + text / visual prompts<br>Also a prompt-free route | Open-vocabulary detection and segmentation.<br>Model / prompt limit scope. |

Example: prompt "ring", then check that the box marks the task's passage ring.

YOLOE prompt modes and training examples
https://github.com/THU-MIG/yoloe

### 8.7.3 Common Open-Source Algorithm Libraries: DeepSORT

# Maintaining Detection Identities Across Frames

**Detection boxes** → **Motion and appearance Association costs** → **Assignment and tracks Persistent identities**

$$C = \begin{bmatrix} 0.2 & 0.9 \\ 0.8 & 0.1 \end{bmatrix}$$

Association cost example:
Rows: tracks. Columns: new detections.
Lower cost = better match.

$$J_{\text{direct}} = 0.2 + 0.1 = 0.3 < 1.7 = 0.9 + 0.8 = J_{\text{cross}}$$

Result: retain both identities. Use prediction and appearance to reassociate after occlusion.
If neither assignment is plausible, keep a lost track rather than force an identity swap.

### 8.7.3 Common Open-Source Algorithm Libraries: VINS-Fusion

# Camera and IMU Constraints Describe One Vehicle State

| | | |
|---|---|---|
| **Mono + IMU**<br>**Stereo + IMU / stereo** | → | **Local optimization**<br>**Motion-state estimate** → **Pose and trajectory**<br>**Optional loop closure** |

**Inputs:** images and optional inertial measurement unit (IMU), camera intrinsics, camera–IMU extrinsics, time offset, and noise.

**Outputs:** vehicle position, attitude, and motion state. Optional loop closure limits long-term drift.

**Global navigation satellite system (GNSS):** a separate global fusion module can constrain the local estimate in global coordinates.

VINS-Fusion configurations and global_fusion
https://github.com/HKUST-Aerial-Robotics/VINS-Fusion

### 8.7.3 Common Open-Source Algorithm Libraries: ORB-SLAM Series

# Feature Reprojection Connects Trajectories and Maps

| System | Inputs and Capabilities | Conditions |
|---|---|---|
| **ORB-SLAM2** | Monocular, stereo, RGB-D<br>Sparse mapping / localization | Calibration, texture, parallax affect matching. Monocular vision lacks absolute metric scale. |
| **ORB-SLAM3** | Visual or visual-inertial<br>Multiple maps and loop closure | IMU timing and camera extrinsics affect fusion |

Simulated images + calibration → camera pose / sparse map. Texture / motion affect matches.

ORB-SLAM3 sensors and runtime configurations
https://github.com/UZ-SLAMLab/ORB_SLAM3

## 8.7.3 Common Open-Source Algorithm Libraries: LiDAR-Inertial Odometry

# Point Registration Complements IMU Propagation

| Algorithm | Main Mechanism | Interface Requirements |
|---|---|---|
| **FAST-LIO** | Iterated extended Kalman filter fuses LiDAR and inertial data | Point clouds, IMU, mounting extrinsics, sample times |
| **FAST-LIO2** | Direct raw-point registration<br>Incremental point-cloud map | Per-point timing and deskewing affect fast motion |
| **Faster-LIO** | Sparse incremental voxels<br>Faster map queries / updates | Pose and map outputs for planning or control |

LiDAR-inertial odometry (LIO): LiDAR + IMU estimate vehicle motion.
Check sensor conventions first, then trajectory continuity, map consistency, and latency.

### 8.7.3 Common Open-Source Algorithm Libraries: FASTER and EGO-Planner

# Unknown Space and Local Obstacles Enter Planning

| Planner | Key Problem | Input and Output |
|---|---|---|
| **FASTER** | Fast progress into unknown space with a known-free-space backup | State, map, goal → Main trajectory and stopping backup |
| **EGO-Planner** | Local collision constraints and B-spline gradient optimization | State, local map, goal → Feasible local trajectory |

Euclidean signed distance field (ESDF): distance to obstacle boundaries, with inside / outside signs.
EGO-Planner uses local optimization without building an ESDF.

### 8.7.3 Common Open-Source Algorithm Libraries: EGO-Swarm

# Shared Future Trajectories Support Distributed Avoidance

EGO-Swarm: each vehicle optimizes locally using its observations and neighbors' trajectories.
Inputs: own pose, obstacles, goals, and neighbors' timed plans.
$p_1, p_2$ are positions (m), $t$ is time (s). Minimum separation: 1 m.

$$p_1(t) = t, \quad p_2(t) = 4 - t, \quad d(t) = |4 - 2t| \text{ m}$$

| Prediction Time | 1D Separation | Interpretation |
|---|---|---|
| 1 s | 2 m | Above 1 m, acceptable for now |
| 2 s | 0 m | Meeting predicted. Replan early. |

**Neighbor plans support avoidance only with correct times and frames.**

Chapter Outline

# Key Tools and Framework Interfaces

**Chapter 8 Multimodal Perception and Intelligent Decision-Making**

8.1 Image Processing Fundamentals

8.2 Recognition, Tracking, Optical Flow

8.3 Localization, Mapping, and Planning

8.4 Common Perception Sensors

8.5 Coordinate Transformations

8.6 Fusion and Control Decision-Making

**8.7 Tools and Framework Interfaces**

8.8 Experiments and Real-World Transfer

8.9 Exercises and Classroom Discussion

## 5 Units in This Section

**8.7.1 Core Development Tools**
Textbook: Sec. 8.1.3.1

**8.7.2 MATLAB/Simulink Algorithm Libraries**
Textbook: Sec. 8.1.3.2

**8.7.3 Common Open-Source Algorithm Libraries**
Textbook: Sec. 8.1.3.3

**8.7.4 Simulation Development Framework**
Textbook: Sec. 8.2.1

**8.7.5 Common Interfaces and Custom Development**
Textbook: Sec. 8.2.3

## 8.7.4 Simulation Development Framework: Overall Framework

# Four Core Modules and Two Control Loops

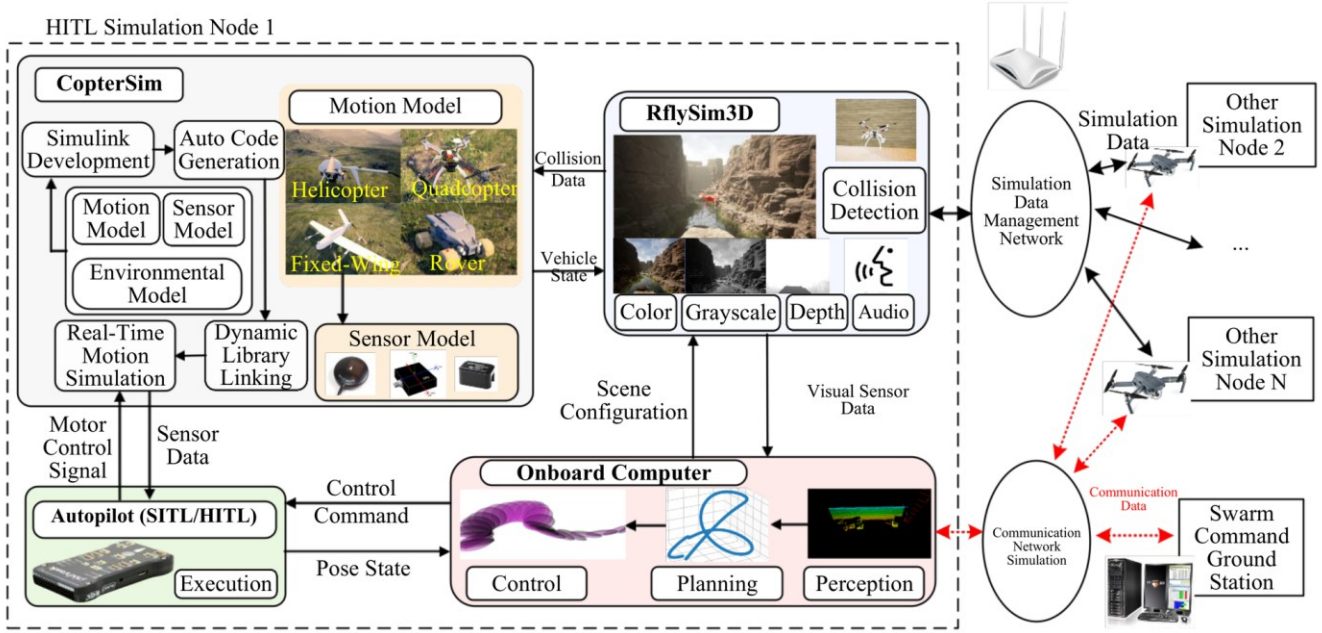



Fig. 8.8 Overall Perception and Decision-Making Simulation Framework

Lower: motors → motion → sensors. Upper: images → algorithms → external commands.
SITL/HITL = software/hardware-in-the-loop (SIL/HIL in the textbook).

### 8.7.4 Simulation Development Framework: Fig. 8.8: Upper Modules

# Motion Models and Scene Rendering Produce Different Data

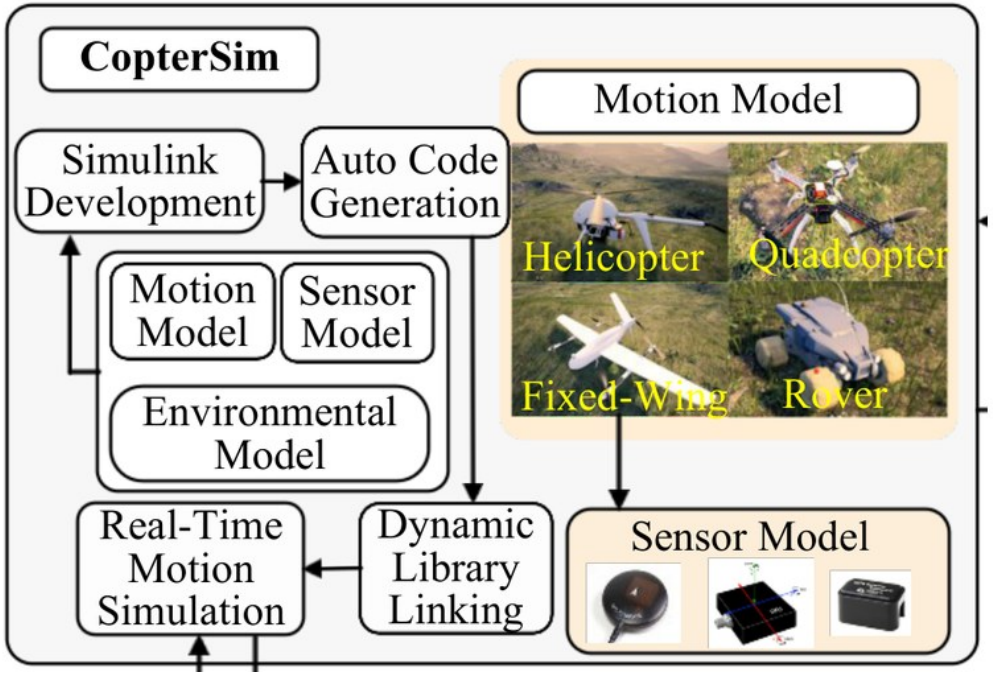



Fig. 8.8: CopterSim Motion and Sensors

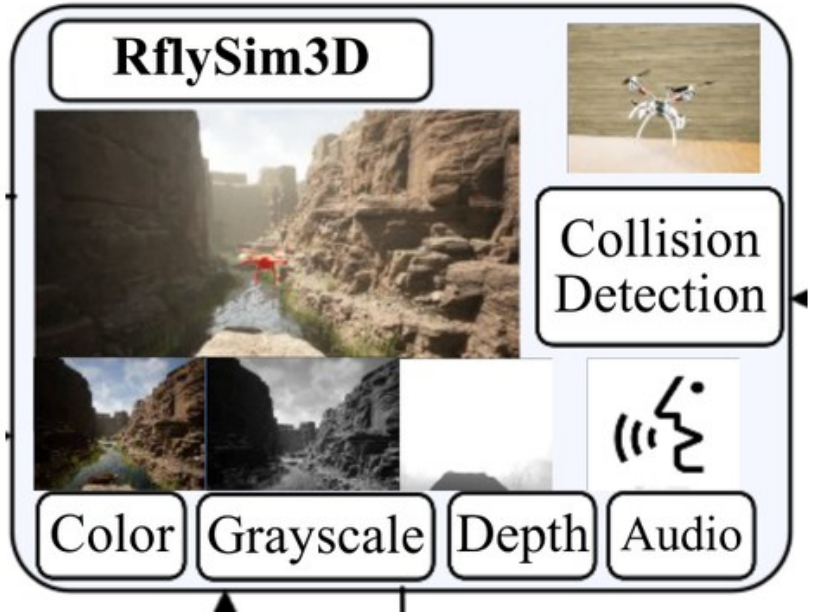



Fig. 8.8: Rendering and Collisions

Motion states go to RflySim3D. Scene collision data return to CopterSim.

### 8.7.4 Simulation Development Framework: Fig. 8.8: Lower Modules

# Onboard Algorithms and the Autopilot Close the Loop

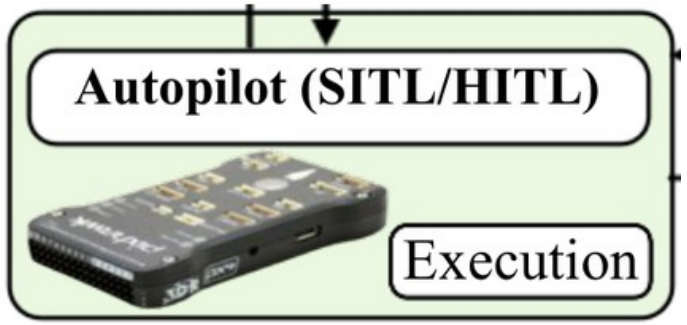



Fig. 8.8: Autopilot Execution

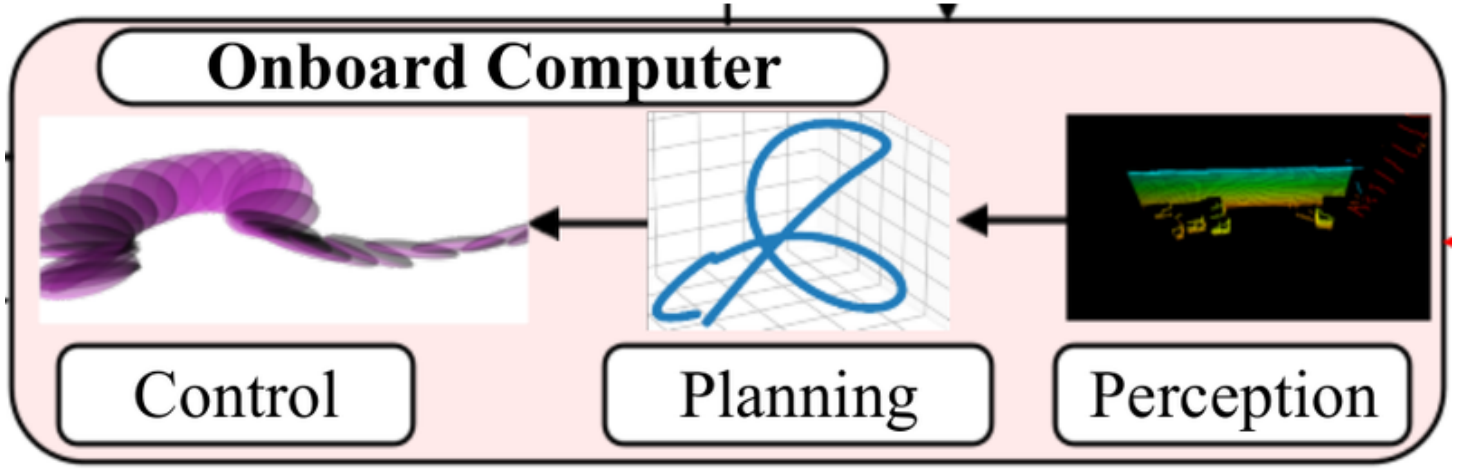


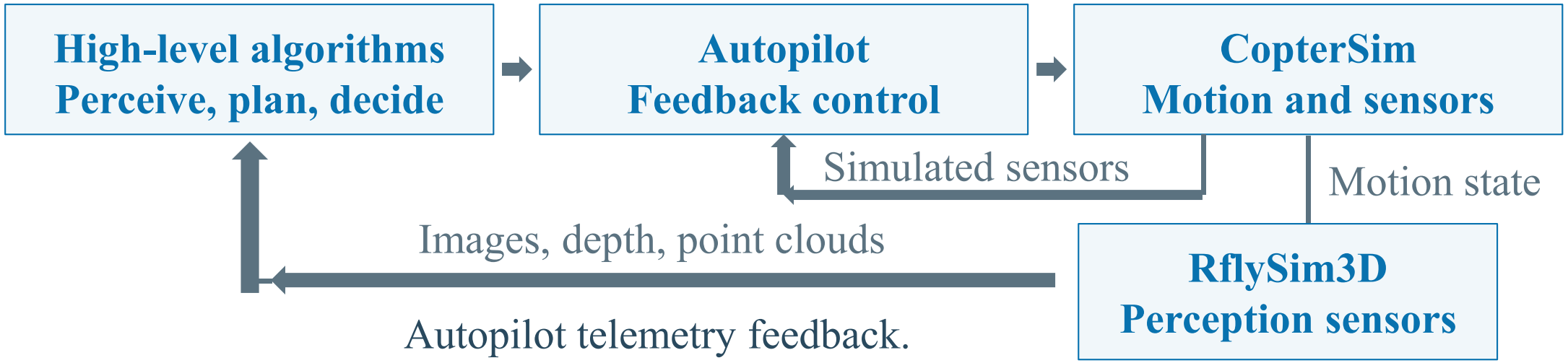

Fig. 8.8 Detail: Onboard Perception, Planning, and Control

**Commands:** onboard algorithm → autopilot → motors. CopterSim updates motion.

**States:** sensors → autopilot → pose. New states update onboard decisions.

### 8.7.4 Simulation Development Framework: Modules and Development Routes

# Dynamics, Perception, and Control Form Two Loops




**Python:** the software development kit (SDK) supports acquisition and prototypes. Shared memory supports a single host.

**Simulink / ROS:** models use data and control interfaces. ROS messages connect algorithms to external control.

### 8.7.4 Simulation Development Framework: SITL and HITL

# What Real Hardware Adds to the Test

| Aspect | Software-in-the-Loop (SITL) | Hardware-in-the-Loop (HITL) |
|---|---|---|
| **Autopilot** | PX4 software in the runtime | Real autopilot running firmware |
| **High-level algorithms** | Windows/Python or WinWSL/ROS | Real onboard computer<br>Target architecture / device libs |
| **Motion and sensing** | CopterSim and RflySim3D | Still simulated models and scenes |
| **Check** | Logic, interfaces, algorithm outputs | On-board latency, resources, communication |

### 8.7.4 Simulation Development Framework: Distributed Simulation

# Calculate Bandwidth Before Sharing Multi-Vehicle Images

Given: 4 streams, 640 × 480 × 3 channels, 30 frames/s, uint8 (1 byte/channel).
Each node: vehicle ID, sensor ID, IP, port, and time base.

$$B = 4 \times 640 \times 480 \times 3 \times 30 = 110592000 \text{ byte/s}$$

$$R = 8B = 884.736 \text{ Mbit/s}$$

This is image payload only. Headers, point clouds, and control messages add load. Images cannot use the full rated link capacity.

Send observations or results as the task requires. Weigh compression, decoding, loss, and freshness.

## 8.7.4 Simulation Development Framework: ROS Compatibility

# Sensor Forwarding and Motion Control Use Separate Interfaces

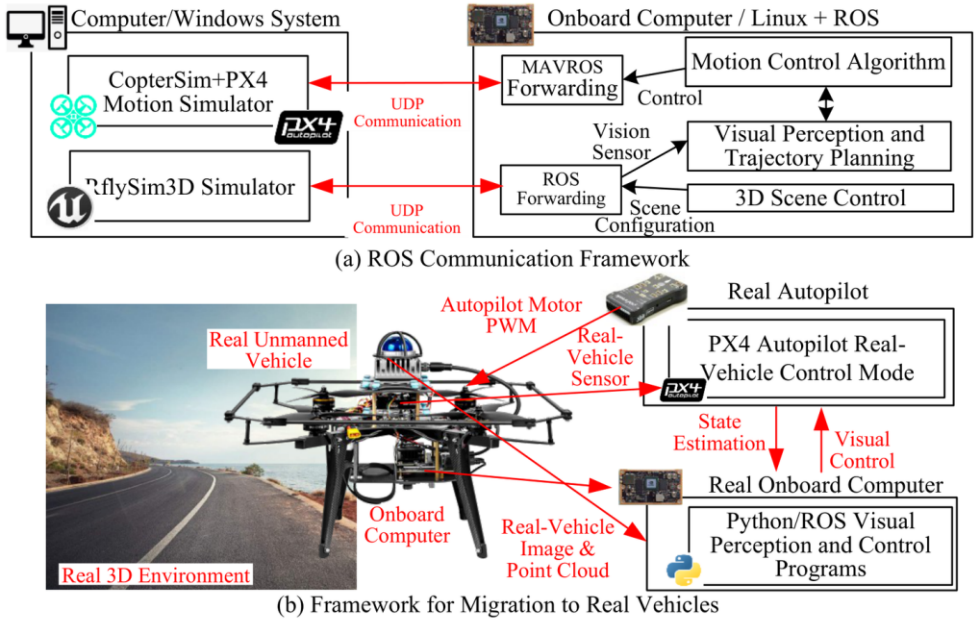



Fig. 8.9 ROS Compatibility and Sim2Real Transfer

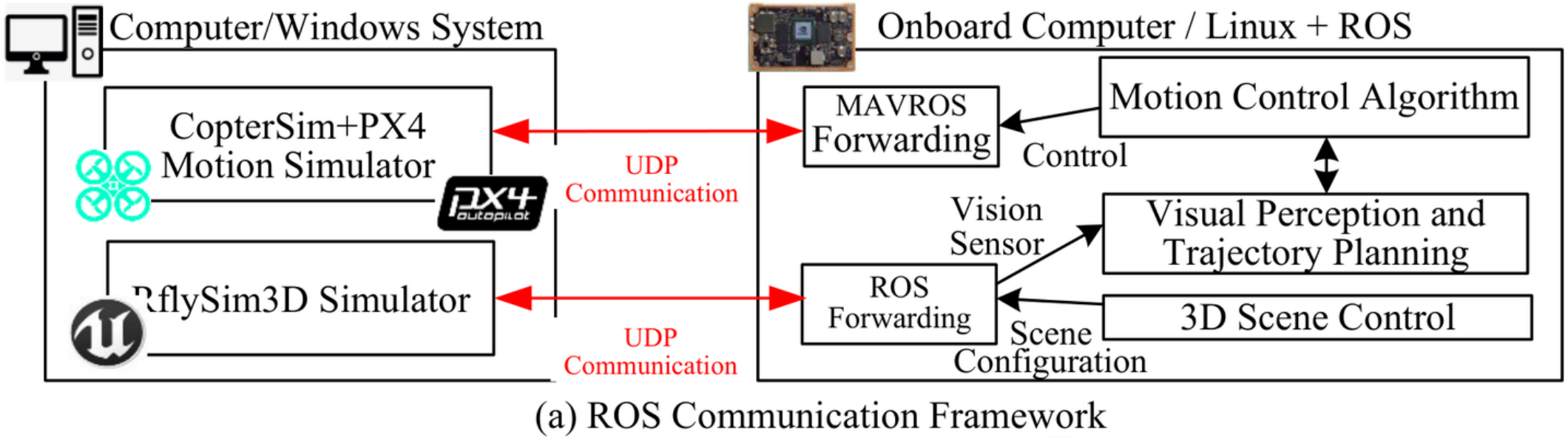
**Vision:** RflySim3D → network reception → ROS messages → perception and planning.

**Motion control:** autopilot states → MAVROS. Planned targets → MAVROS → PX4.

**Simulation-to-real transfer (Sim2Real):** real sensors replace observation sources. Real vehicles replace simulated motion.

### 8.7.4 Simulation Development Framework: Fig. 8.9(a) Detail

## Simulation Images Enter ROS and External Targets Enter PX4



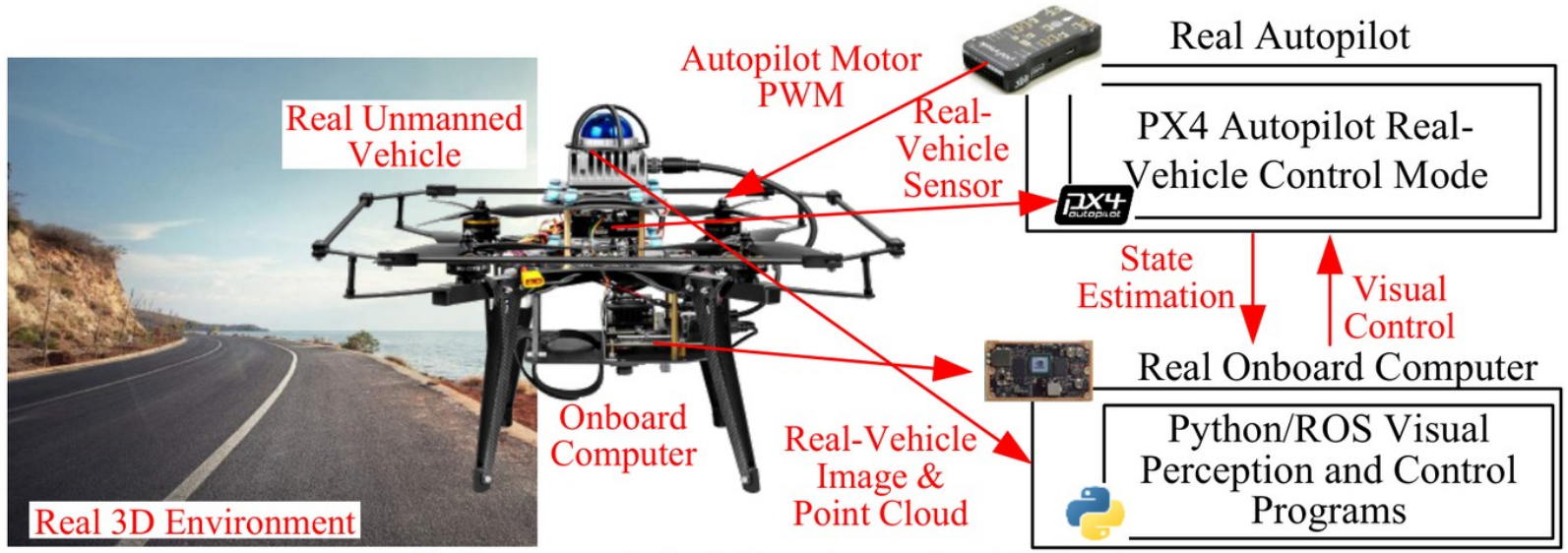

(a) ROS Communication Framework

Fig. 8.9(a) Simulation Motion, Observations, and High-Level Programs

Data reception is not control execution. Verify both data and target execution paths.

### 8.7.4 Simulation Development Framework: Fig. 8.9(b) Detail

## Real Sensors and Vehicles Replace Simulation Modules




(b) Framework for Migration to Real Vehicles

Fig. 8.9(b) Real Sensors, Vehicles, and Onboard Computers

Scan → model → augment virtual/real data → in-the-loop → real (Fig. 8.10: Unit 8.8.4).

### 8.7.4 Simulation Development Framework: Development Workflow

# Data, Decisions, and Responses Test the Closed Loop

| Observed Symptom | Working Link | Next Checks |
| --- | --- | --- |
| **Image refreshes, box stays fixed** | Sensor reception | Does the algorithm read new frames? Weights and target class. |
| **Trajectory changes, vehicle does not** | Perception and planning | External-control mode, target frame, publish rate, control path |
| **Vehicle flies through obstacles** | Motion control | Obstacle frames, planning constraints, tracking error, sample times |

Chapter Outline

# Key Tools and Framework Interfaces

**Chapter 8 Multimodal Perception and Intelligent Decision-Making**


8.1 Image Processing Fundamentals

8.2 Recognition, Tracking, Optical Flow

8.3 Localization, Mapping, and Planning

8.4 Common Perception Sensors

8.5 Coordinate Transformations

8.6 Fusion and Control Decision-Making

**8.7 Tools and Framework Interfaces**

8.8 Experiments and Real-World Transfer

8.9 Exercises and Classroom Discussion


## 5 Units in This Section

**8.7.1 Core Development Tools**
Textbook: Sec. 8.1.3.1

**8.7.2 MATLAB/Simulink Algorithm Libraries**
Textbook: Sec. 8.1.3.2

**8.7.3 Common Open-Source Algorithm Libraries**
Textbook: Sec. 8.1.3.3

**8.7.4 Simulation Development Framework**
Textbook: Sec. 8.2.1

**8.7.5 Common Interfaces and Custom Development**
Textbook: Sec. 8.2.3

### 8.7.5 Common Interfaces and Custom Development: Environment and Documentation

## Look Up Parameter Protocols and Python Methods Separately

**Environment:** first verify local capture with Windows/Python. Choose WinWSL, a VM, Linux, or an onboard computer for the ROS and hardware dependencies.

**Student documentation:** [installation directory]/RflySimAPIs/8.RflySimVision/
**API.pdf:** development guide. VisionSensorAPI.pdf: parameter protocol.

**SDK methods:** [installation directory]/RflySimAPIs/RflySimSDK/html/index.html
Continue to 4.DevRef/pysdk/vision/viscap.html for actual methods.

### 8.7.5 Common Interfaces and Custom Development: Config.json

# Sensor IDs and List Indices Have Different Roles

| Field | Meaning | Initial Setting / Reading |
|---|---|---|
| **SeqID / TargetCopter** | Sensor ID / vehicle ID | 0 / 1<br>ID ≠ list index |
| **TypeID** | Data type and decoding | 1: color. 2: depth. |
| **DataWidth / DataHeight** | Output resolution in pixels | 640 / 480 |
| **DataCheckFreq / protocol** | Update checks / transport | 30 Hz; SendProtocol[0]=0 (shared memory) |

One configured item with SeqID=7 still uses Img[0]. Check its ID with VisSensor[0].SeqID.

## 8.7.5 Common Interfaces and Custom Development: Mounting Extrinsics

# A Fixed Body Mount Needs Units and a Frame

**Vehicle reference: FRD**
**Forward-right-down**



**Camera mount**
**0.3 m forward**

TargetMountType=0: fixed to the vehicle reference point.
SensorPosXYZ=[0.3,0,0]: meters, offset along body forward.
SensorAngEular=[0,0,0]: degrees, interpreted for this mounting mode.

EularOrQuat selects Euler angles or quaternion configuration. Keep the actual key.
Camera optical, body FRD, and ground-frame conversions: Section 8.5.

### 8.7.5 Common Interfaces and Custom Development: Interface Lifecycle

# A Successful Request Still Needs a First Frame

```
import VisionCaptureApi
vis = VisionCaptureApi.VisionCaptureApi()
if not vis.jsonLoad():
    raise RuntimeError("Config load failed")
if not vis.sendReqToUE4():
    raise RuntimeError("Request failed")
vis.startImgCap()
```

**Configuration:** jsonLoad() finds Config.json beside the entry script.

**First frame:** wait for hasData[0] with a timeout. Check size and type.

Configure → request → receive → first-frame timeout → process → finally cleanup.
A True request result covers that stage only. Received data establish first-frame validity.

## 8.7.5 Common Interfaces and Custom Development: Reading New Frames

# The Processing Loop Must Detect Timestamp Changes

```
import numpy as np
if vis.hasData[0]:
    with vis.Img_lock[0]:
        frame = vis.Img[0].copy()
        stamp = np.asarray(vis.timeStmp[0]).item()
    if stamp != last_stamp:
        last_stamp = stamp
        process(frame)
```

**Image lock:** protects copying. Time updates separately. Use paired fusion messages.

**timeStmp[0]:** filters repeated times. It cannot ensure image–time pairing.

Initialize last_stamp=None before the loop. process is your own perception function. The full outer loop also needs timeouts, an exit signal, and cleanup in finally.

## 8.7.5 Common Interfaces and Custom Development: Color Images

# The RGB Sensor Type Returns a BGR Array

| Data Layer | Current Convention | Example |
|---|---|---|
| **Sensor type** | TypeID=1: color image | The RGB type name does not set array channel order. |
| **Python array** | uint8; H × W × 3; BGR | Pure red pixel = [0,0,255] |
| **Algorithm input** | Convert channels as the model requires | cv2.COLOR_BGR2RGB or cv2.COLOR_BGR2HSV |

### 8.7.5 Common Interfaces and Custom Development: Depth Data

# Mask Invalid Values Before Converting to Meters

TypeID=2: uint16, one channel. Transport can give H × W or H × W × 1.
Depth scale $u_{\mathrm{m}}$ is otherParams[2], default 0.001 m/count. Mask 65535 first.

$$d_{\mathrm{m}} = d_{\mathrm{raw}} u_{\mathrm{m}} = 2370 \times 0.001 = 2.370\ \mathrm{m}$$

| Raw Integer | Processed Value | Distance Decision |
|---|---|---|
| **2370** | Valid depth: 2.370 m | Compare with a metric threshold |
| **65535** | Invalid. Exclude with a mask. | Unknown. Use other valid data. |

Unknown ≠ free space. Resolve units and validity before algorithm input.

### 8.7.5 Common Interfaces and Custom Development: Segmentation and Point Clouds

## Rendered Labels and Algorithm Estimates Have Different Sources

| Data | Current Interface Meaning | Integration Checks |
|---|---|---|
| **Segmentation: Type 4** | uint8, BGR rendered label image | Ground-truth labels. Assess predictions separately. |
| **Point clouds: Type 7/20 etc.** | N × 4 floating-point array XYZ decoded to meters | Column 4: semantics or zero. Do not rescale the whole array. |
| **Range / optical flow** | Range object in Img[i] vis.OpticalFlowSensor | Read structure fields, not a generic image array. |

Type 7 point clouds, pods, and infrared depend on installed authorization. Check RGB and depth first.

### 8.7.5 Common Interfaces and Custom Development: Timing and Fusion

# Reading Together Does Not Mean Sampling Together

Images: timeStmp is relative time; imgStmp is mapped time.
IMU: request with sendImuReqCopterSim(). Buffer and pair on a common time base.

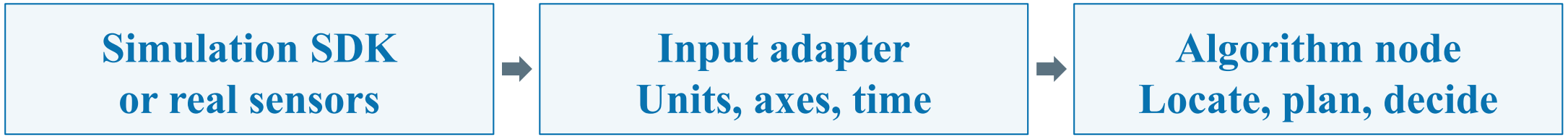
$$\Delta t = |t_{\text{imu}} - t_{\text{cam}}|, \qquad t_{\text{cam}} = 1.000\ \text{s}$$

| Given IMU Time | Camera Offset | Teaching Limit: 10 ms |
|---|---|---|
| **1.005 s** | 5 ms | Candidate for pairing |
| **1.025 s** | 25 ms | Reject / model interpolation |

imu.AlignTime(t) records the latest image time. Buffering, clock mapping, pairing / interpolation, and validity remain explicit fusion work.

### 8.7.5 Common Interfaces and Custom Development: ROS and Custom Development

# Message Conventions Connect Sensor Data to Algorithms




**Images / point clouds:** check encoding, shape, units, frame_id, and TF. A message name does not transform coordinates.

**ROS:** messages carry vision data to algorithms. MAVROS connects autopilot states and position, velocity, or attitude targets.

**Simulink:** acquisition and external-control blocks connect models. Check code generation and target support per block.

## 8.7.5 Common Interfaces and Custom Development: Control and Exit

# A Perception Stop Needs an Explicit Vehicle End Action

```
try:
    run_perception_loop()
finally:
    vis.stopRun()
    cv2.destroyAllWindows()
```

run_perception_loop is your outer loop.
Set the vehicle's end action before releasing receivers.

VisionCaptureApi: data. PX4MavCtrl: external control. Scene interfaces: objects. Evaluation: metrics.

stopRun sets a shared stop flag, closes CopterSim IMU/odometry receivers, and joins shared-memory capture. Verify image-UDP workers/sockets separately.

**Resource shutdown and the vehicle end action both belong in the control flow.**

Exercises and Discussion

# Diagnosing a Two-Camera Avoidance System

**1. IDs and new frames:** SeqID values are 7 and 9, configured in that order. Which Img indices? How do you avoid duplicate processing?

**2. Physical values:** scale 0.001 m/count. How should valid depth 2370 and invalid value 65535 enter avoidance?

**3. Execution and transfer:** 30 Hz input, 100 ms inference. What happens? What evidence is needed on an ARM onboard computer?

Answer Points

# Tracing Data Semantics to System Response

**IDs and new frames:** use Img[0] and Img[1]. Confirm VisSensor[i].SeqID, then compare sample timestamps before processing.

Depth: $2370 \times 0.001 = 2.370$ m. Mask 65535 as invalid. It is not distant free space.

**Latency and target:** 100 ms inference gives about 10 Hz. Check buffering and optimize the whole chain. Verify target builds, dependencies, fresh data, latency, and response.

Section Summary

# Task and Interface Requirements Determine Tool Choice

**Tools and models:** core tools, MATLAB toolboxes, and open-source algorithms have distinct roles. Connect them by inputs and outputs.

**Systems and evidence:** SITL checks logic; HITL adds real onboard computing and autopilot hardware. Distribution adds networks and multi-vehicle timing.

**Interface pitfalls:** an index is not SeqID; BGR is not RGB. Mask invalid depth before converting to meters. Same-loop reads are not synchronization. Closing a window is not a vehicle end action.

**Next:** Section 8.8 selects textbook experiments and tests observations, result criteria, and real-system differences.