Intelligent Unmanned System Control Practice

Chapter 2 Experimental Platform Configuration

Author: Xunhua Dai, Central South University

Website: https://faculty.csu.edu.cn/daixunhua

Email: dai.xh@csu.edu.cn

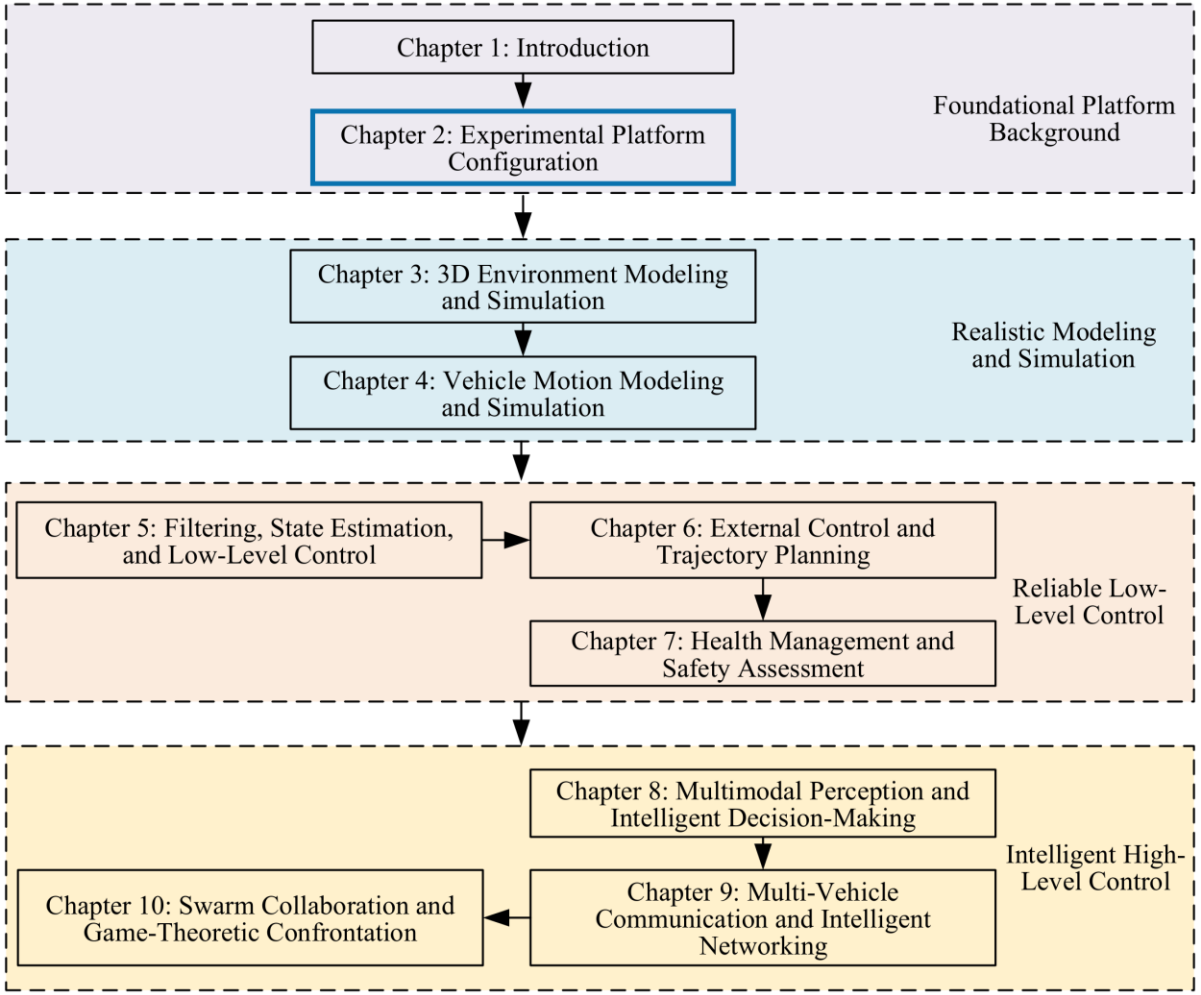
RflySim: http://rflysim.com

Companion Paper



Dai, Xunhua, Jinhu Tu, and Quan Quan. "RflySim ToolChain: a rapid development and validation toolchain for intelligent unmanned swarm systems." Journal of Systems Engineering and Electronics 36.4 (2025): 1077-1093.

Textbook is coming soon...



Chapter Learning Objectives

What You Can Do After Chapter 2

1. Explain the roles and connections of the main tools
2. Install the platform and run a first simulation
3. Compare simulation modes and identify interface inputs and outputs
4. Reproduce an example and explain the result of a change

Understand tool data flow to judge whether changes truly improve system behavior.

Chapter Outline

Key Tools: From Runtime Environments to Closed-Loop Simulation

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.1.1 Development Basics and Runtime Environments

Textbook: Sec. 2.1.2.2, 2.1.2.3, 2.1.2.7

2.1.2 Model and Algorithm Development

Textbook: Sec. 2.1.2.1, 2.1.2.3, 2.1.2.7

2.1.3 System Components and Communication

Textbook: Sec. 2.1.2.4–2.1.2.6

2.1.4 Simulation Tools and Development Routes

Textbook: Sec. 2.1.2.1, 2.1.2.8

2.1.1 Development Basics and Runtime Environments: Tool Categories

Five categories in the toolchain

- **Runtime environments:** Windows and Ubuntu host programs. Windows Subsystem for Linux (WSL) hosts a simulated Linux onboard environment, including PX4 firmware builds and ROS development.
- **Languages and libraries:** Python and C/C++ implement programs; libraries such as OpenCV provide ready-made functions, including image processing.
- **Development tools:** MATLAB analyzes numerical values and data; Simulink builds system models from blocks and signal connections.
- **Interfaces and collaboration:** An application programming interface (API) defines how to call functions.

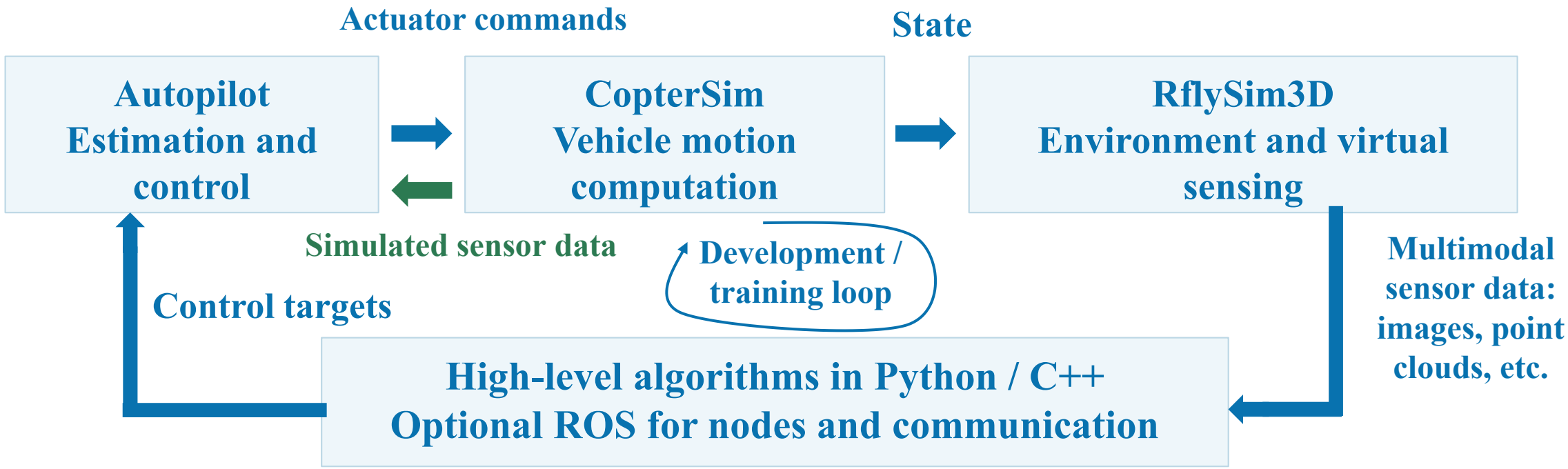
Robot Operating System (ROS) is a software framework for coordinating modules.

- **Simulation and autopilots:** RflySim connects vehicle models, autopilots such as PX4, a ground station, high-level algorithms, and 3D scenes in a loop to test system behavior virtually.

Environments host programs; algorithms process data; communication closes the simulation loop.

2.1.1 Development Basics and Runtime Environments: A First Look at the Simulation Loop

RflySim connects models, control, and scenes



Interfaces connect motion, control, the environment, and high-level algorithms in a simulation loop.

Building training simulations and developing algorithms need many tools → RflySim toolchain

2.1.1 Development Basics and Runtime Environments: Operating Systems

Which programs run on Windows and Linux?

- **Windows:** Runs motion simulation, 3D rendering, control algorithms, and ground-station programs; handles simulation setup, display, and interaction.
- **Ubuntu:** A Linux distribution for developing, building, and running the supplied autopilot and ROS programs and deploying onboard algorithms.
- **Cross-system collaboration:** Programs exchange data through interfaces, each using its own libraries and toolchain.



Ubuntu: a Linux distribution

Onboard deployment: Both Linux systems must still match in processor architecture and drivers.

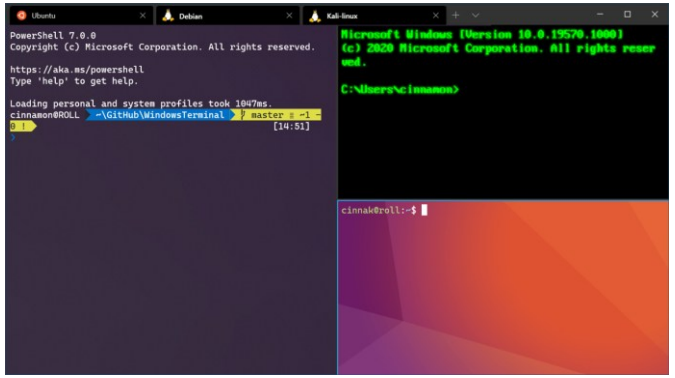
Windows system and development documentation
https://learn.microsoft.com/zh-cn/windows/

Ubuntu setup and command-line tutorials
https://ubuntu.com/tutorials

Both systems can share a PC; programs, dependencies, and device architectures must match.

2.1.1 Development Basics and Runtime Environments: WSL and Docker

WSL provides Linux; Docker packages applications



Windows and Linux commands in one terminal

- **WSL:** Runs Linux programs on Windows. WSL1 uses a compatibility layer; WSL2 runs a Linux kernel.
- **Docker:** An image stores an application and its dependencies; a container is a running instance. Containers in the same environment share a kernel.
- **Combined use:** Linux containers can run through WSL2. WSL containers (WSLC) have been in public preview since 2026-06.

Image → container instance

WSL installation and configuration
https://learn.microsoft.com/windows/wsl/

Introduction to Docker images and containers
https://docs.docker.com/get-started/

WSL runs Linux programs; Docker images reuse apps and dependencies. They can work together.

2.1.1 Development Basics and Runtime Environments: More on Systems and Environments

Processor architectures and WinWSL

Note: Everyday development computers typically use the x86-64 processor architecture, also called AMD64. Phones and onboard computers face strict power constraints, so most use ARM64. In both names, “64” denotes a 64-bit system word length.

Note: The RflySim toolchain includes the WinWSL build environment, based on WSL 1/2 and Ubuntu 22.04. It supports Docker containerization to simplify transferring algorithms to real hardware.

Architecture determines executable compatibility; WSL hosts Linux and Docker packages applications.

2.1.1 Development Basics and Runtime Environments: Optional: Cross-Architecture Builds

x86 Windows PC → ARM64 Onboard Computer

Development computers simplify debugging; onboard boards face size, power, and compute limits. Common x86-64 PCs and ARM64 boards have different instruction sets. Even with Ubuntu on both, programs and libraries must be built for the target architecture.

- **Buildx:** Docker Buildx specifies the target platform and uses ARM64-compatible base images and dependencies to build a target image.
- **Two common routes:** QEMU emulates target instructions to run build steps, but may be slower. A cross-compiler generates target machine code on the development computer.
- **Onboard execution:** Programs need target-system libraries, and image capture needs camera drivers. Image input, algorithm output, and processing latency together show deployment performance.

Docker builds for multiple architectures: https://docs.docker.com/build/building/multi-platform/

Builds target an architecture; onboard runs also need system libraries and device drivers.

2.1.1 Development Basics and Runtime Environments: Command-Line Basics

The roles of terminals, CMD, and BAT

- **Terminal window:** Provides input and display space for CMD, PowerShell, or a Linux shell.
- **Command interpreter (shell):** CMD interprets Windows commands; PowerShell and Linux shells each have their own syntax.
- **Batch scripts:** A BAT file stores CMD commands to set directories and parameters and launch programs,
  helping reproduce an experiment's startup conditions.
- **Running Python:** Use cd to enter the script directory, then run python ShootBall3.py.
  The shell starts the interpreter to execute the script.

Using Windows Terminal
https://learn.microsoft.com/windows/terminal/

CMD command reference (SS64)
https://ss64.com/nt/

Terminals host interaction; shells interpret launch commands; Python interpreters run algorithm scripts.

2.1.1 Development Basics and Runtime Environments: Languages and Libraries

Python for experiments; libraries for algorithms

Languages and libraries in the red-ball example

Layer	Role in the example	What to locate
Python main program	Organizes sampling, decisions, and loops	ShootBall3.py
Numerical / vision libraries	Store images, segment targets, compute centroids	NumPy, OpenCV
RflySim API	Acquire images, read state, send motion targets	Image acquisition and control interfaces
C/C++ and compiled libraries	Implement low-level control or fast computation	Firmware, drivers, library functions

Python documentation: https://docs.python.org/3/ C++ docs: https://learn.microsoft.com/cpp/

Python: rapid algorithm development and testing; C/C++: often onboard optimization and deployment.

2.1.1 Development Basics and Runtime Environments: Getting Started with Python

Python programs, interpreters, and packages

- **Language and interpreter:** Python code is stored in .py files and executed by an interpreter. Functions and loops organize experiments.
- **Packages:** import brings in functions. NumPy handles arrays, OpenCV processes images, and platform APIs connect to simulation.
- **Development advantages:** Interactive trials and existing libraries simplify algorithm checks. Time-consuming operations can call compiled implementations.
- **Environment isolation:** venv isolates package versions. Use the supplied interpreter for platform examples. Check its path with sys.executable; python -m pip installs packages for that interpreter.

Python syntax and program structure
https://docs.python.org/zh-cn/3/tutorial/

Introduction to NumPy arrays
https://numpy.org/learn/

Interpreters run the main program; packages supply functions; environments select dependencies.

2.1.1 Development Basics and Runtime Environments: C/C++ and Dynamic Libraries

Implementing algorithms in C/C++

Think: With very limited onboard computing power, how would you optimize a small unmanned aircraft and make tradeoffs?

- **C:** A general-purpose language that organizes data and computation with functions, arrays, and structures. Pointers represent memory addresses; access bounds and object lifetimes affect correctness.
- **C++ extensions:** Classes, templates, and the standard library organize code, encapsulate model state, and reuse algorithms. C++ is closely related to C, but they are not fully compatible.
- **Selection criteria:** Native code and explicit data layouts help optimize repeated computations and integrate existing model libraries. Actual speed still depends on the algorithm, implementation, and target hardware.
- **Typical uses:** Autopilot firmware, drivers, motion models, and ROS nodes can directly reuse existing native C/C++ code and libraries.

C++ syntax and exercises (LearnCpp): https://www.learncpp.com/

C/C++ supports native-code reuse and optimization for firmware, drivers, and models.

2.1.1 Development Basics and Runtime Environments: C/C++ and Dynamic Libraries

The roles of editors, compilers, and build tools

- **Editing and debugging:** An editor handles code; an integrated development environment combines editing, building, and debugging. VS Code's C/C++ extension still needs a compiler. Installing it does not install the complete toolchain.
- **Compilation and linking:** Compilers turn source into object code; linkers combine object files and resolve external references. Common compiler toolchains include MSVC, GCC, and Clang.
- **Project configuration:** CMake reads CMakeLists.txt to identify sources, dependencies, and targets, then generates a build system. It does not replace a C++ compiler.
- **Building:** Ninja, Make, or MSBuild invokes tools according to dependencies. Rebuilding after a source change usually handles only affected parts. Then run and debug the new output.

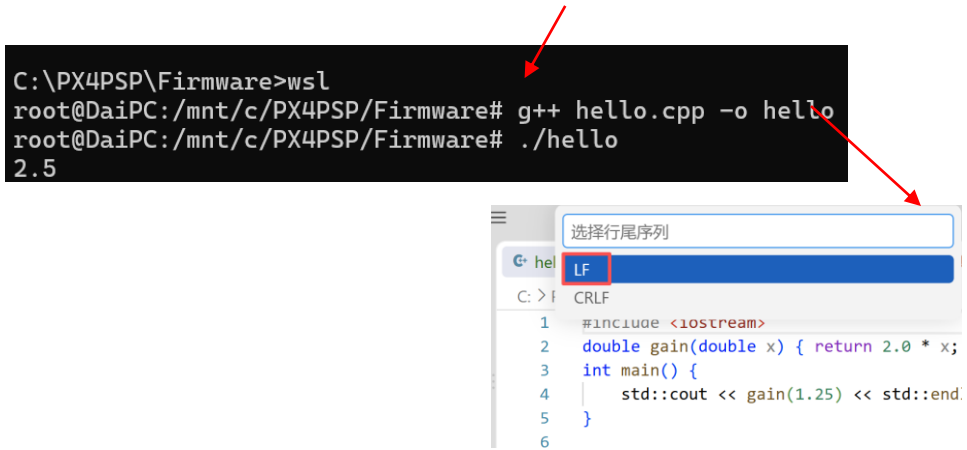
CMake build tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Build settings specify targets; build tools schedule steps; compilers and linkers create outputs.

2.1.1 Development Basics and Runtime Environments: A Minimal C++ Exercise

How a gain function produces a running result

```
#include <iostream>
double gain(double x) { return 2.0 * x; }
int main() {
    std::cout << gain(1.25) << std::endl;
}
```



- **Environment:** Install MSVC C++ tools; open the x64 Native Tools Command Prompt.
- **Compile and run:** Save hello.cpp; run cl /EHsc hello.cpp, then hello.exe.
- **Check:** Expect 2.5. Change 1.25 to -2.0, rebuild, and expect -4.

MSVC builds: https://learn.microsoft.com/cpp/build/building-on-the-command-line

Source defines computation; build outputs execute it. Rebuild after changes and check results.

2.1.1 Development Basics and Runtime Environments: C/C++ and Dynamic Libraries

Source files, executables, and dynamic libraries

- **Source and header files:** .c/.cpp files contain implementations; .h files usually share declarations. A declaration specifies how to call a function; its definition must also enter the build.
- **Preprocessing and compilation:** Expand headers and macros; check syntax and types and translate code. Assembly produces .obj/.o object files, not complete programs.
- **Linking:** Combine object files and libraries, resolve external references, and create an executable or dynamic library. Missing function definitions often cause link errors.
- **Runtime loading:** At program startup or library loading, the system must find dynamic libraries and their dependencies. Missing dependencies can prevent execution even after successful compilation and linking.

Clang compilation and linking: https://clang.llvm.org/docs/UsersManual.html
GCC compilation stages and options: https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html

Compilers create object code; linkers resolve function references; loaders load runtime libraries.

2.1.1 Development Basics and Runtime Environments: More on Languages and Builds

Language choices, Linux builds, and cross-compilation

Note: Python excels at rapid algorithm implementation and verification, but falls short of C/C++ in reliability, execution speed, and system resource management. C/C++ is suited to final product deployment.

Note: You can also compile and run programs on Linux with gcc or g++. Look up the details and practice in RflySim's WSL environment.

Basic steps: Save hello.cpp with LF line endings. Run g++ hello.cpp -o hello in the terminal to build an executable, then run ./hello.

Note: Code compiled with gcc on an x86 PC cannot run on ARM; use a cross-compiler such as arm-none-eabi-gcc. This lets WSL build firmware that runs on PX4.

Python speeds algorithm checks; cross-architecture C/C++ deployment needs the right compiler.

2.1.1 Development Basics and Runtime Environments: C/C++ and Dynamic Libraries

Executables and dynamic libraries: EXE, DLL, .so

- **Executable (EXE):** A common Windows program output. At startup, the operating system creates a process and enters the program's entry point. It may contain algorithms or use dynamic libraries.
- **Dynamic-link library (DLL):** A Windows shared-code module loaded by a host, which calls its exported interfaces. A native model DLL normally neither starts as a standalone application nor automatically creates a separate process.
- **Linux shared library (.so):** Serves a role similar to a DLL. Its binary format and platform interfaces differ; changing the extension cannot turn it into a loadable Windows DLL.
- **Two linking methods:** Static linking includes required library code in the output. Dynamic linking uses shared modules at runtime, simplifying reuse and update distribution, but updates must preserve interface compatibility.

Windows DLL fundamentals: https://learn.microsoft.com/windows/win32/dlls/dynamic-link-libraries

EXEs start as programs; compatible hosts load DLLs and .so libraries.

2.1.1 Development Basics and Runtime Environments: Distinguishing Library Files

Static libraries, import libraries, and runtime libraries

- **Static library:** An archive of object files, commonly .a or .lib. Linking includes the needed code in the output.
- **Import library:** On Windows, a .lib may instead be a DLL import library. It lets the linker resolve exported symbols, without containing an implementation that replaces the entire DLL.
- **Runtime dependencies:** A program linked to a DLL through an import library still needs that DLL and its dependencies at runtime. Distributing only the .lib is usually insufficient.
- **Explicit loading:** Interfaces such as ctypes load a DLL and look up functions directly. Correct parameter and return types and calling conventions are still required.

CMake library types: https://cmake.org/cmake/help/latest/command/add_library.html

Static library: object code; import library: link data; DLL: runtime implementation.

2.1.1 Development Basics and Runtime Environments: Dynamic-Library Compatibility

DLL compatibility: interfaces, architecture, dependencies

- **Interface contract:** Exported function names and parameter and return types define the call. Structure layouts, array lengths, and memory-release ownership must also agree to avoid incorrect data or crashes.
- **Processor architecture:** Host process and library need compatible architectures. A typical x64 process cannot directly load an x86 DLL; a 64-bit operating system does not make every process 64-bit.
- **Runtime dependencies:** A DLL may need other dynamic libraries and C/C++ runtimes. A missing-module error with the main file present can indicate a missing dependency or incorrect search path.
- **Cross-language interfaces:** C++ implementations are usually wrapped in explicit C-style functions. Parameter types, array lengths, and memory-release rules must still agree.

DLL exports and name decoration: https://learn.microsoft.com/cpp/build/reference/decorated-names

DLL calls need a clear interface, matching architecture, and accessible runtime dependencies.

2.1.1 Development Basics and Runtime Environments: A Minimal Dynamic-Library Exercise

Exporting the gain function as a Windows DLL

```
// gain.cpp
extern "C" __declspec(dllexport)
double gain(double x) { return 2.0 * x; }
```

- **Build environment:** Open MSVC's x64 Native Tools Command Prompt and build in the file's directory.
- **Build command:** cl /LD /MD /EHsc gain.cpp /link /OUT:gain.dll
- **Interface:** extern "C" uses C linkage; dllexport exports the function; /LD selects a DLL target.
- **Output scope:** gain.dll demonstrates a function call; it is not a CopterSim motion model.

Creating and using an MSVC DLL
https://learn.microsoft.com/cpp/build/walkthrough-creating-and-using-a-dynamic-link-library-cpp

Define exports and build targets; load the DLL with a matching host architecture.

2.1.1 Development Basics and Runtime Environments: More on Dynamic Libraries

Static-library tradeoffs and Linux shared libraries

Note: Static linking reduces runtime dependencies on the linked library, but increases code size and requires relinking for updates. Dynamic libraries simplify sharing and partial upgrades, but risk missing dependencies.

Note: In WSL, remove __declspec(dllexport), keep extern "C", then run

```
g++ -fPIC -shared gain.cpp -o gain.so
```

This produces gain.so, which other programs can call.

Build and load Windows DLLs and Linux .so separately; library type affects dependencies.

2.1.1 Development Basics and Runtime Environments: C/C++ and Dynamic Libraries

Calling a C/C++ computation module from Python

```
import ctypes
from pathlib import Path
lib = ctypes.CDLL(str(Path("gain.dll").resolve()))
lib.gain.argtypes = [ctypes.c_double]
lib.gain.restype = ctypes.c_double
print(lib.gain(1.25))
```

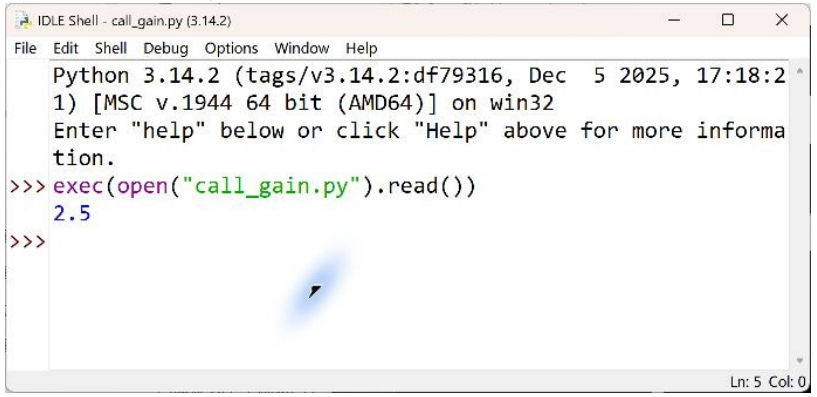


Figure: call_gain.py output in IDLE

- **Run conditions:** Save as call_gain.py and run in the gain.dll directory with x64 Python.
- **Result and distinction:** Prints 2.5. This is a function call inside the Python process; it uses no network and starts no simulation.

ctypes types and calling conventions: https://docs.python.org/3/library/ctypes.html

ctypes calls native functions in Python's process; parameter and return types define data interpretation.

2.1.1 Development Basics and Runtime Environments: The Command Line and Startup Scripts

Startup scripts, interpreters, and algorithms: separate roles

Stage	Red-ball example file / command	What to check
Start simulation	ShootBall3SITL.bat	Scene, motion model, and control software ready
Prepare interpreter	Python38Run.bat	Window confirms the Python3.8 environment is set
Run algorithm	python ShootBall3.py	Mask window opens and tracking starts after 25 s

- **Read scripts:** A batch (BAT) script launches programs and passes parameters; the terminal shows commands and errors. This example uses software-in-the-loop (SITL; SIL in the textbook).
- **Save conditions:** Reproducibility depends on scripts, models, dependencies, and algorithm versions.

BAT sets startup conditions; interpreters run scripts; algorithms map observations to motion targets.

2.1.1 Development Basics and Runtime Environments: Version Control

Git saves experiment changes as traceable versions



Branches separate trials; merging combines changes

- **Version control:** Git records file-change history so you can compare differences and restore recorded versions.
- **Commits and branches:** A commit records a project snapshot. Branches separate approaches; merging combines approved changes.
- **Experiment records:** Save code versions together with parameters, scenes, initial states, and results to support repeated comparisons.

Git command reference
https://git-scm.com/docs

GitHub repository hosting
https://github.com/

Gitee repository hosting
https://gitee.com/

Commits save versions; branches separate trials; experiment conditions link versions to results.

2.1.1 Development Basics and Runtime Environments: Getting Started with Git

Git records changes; hosting platforms share projects

- **Git and hosting:** Git keeps version history locally. GitHub and Gitee host remote repositories for collaboration and access to source code.
- **Inspect before editing:** git status shows which files changed; git diff shows the changes. Confirm exactly what this change includes.
- **Save a version:** git add selects files; git commit saves a local commit without automatically uploading it.
- **Minimal exercise:** Run git init in a copy of an example and commit a baseline. Change one threshold and commit again, record the scene parameters, and inspect history with git log.

Pro Git in Chinese: commits and branches
https://git-scm.com/book/zh/v2

Interactive Git branching exercises
https://learngitbranching.js.org/

Save code commits with experiment conditions to explain why results change.

Chapter Outline

Key Tools: From Runtime Environments to Closed-Loop Simulation

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.1.1 Development Basics and Runtime Environments

Textbook: Sec. 2.1.2.2, 2.1.2.3, 2.1.2.7

2.1.2 Model and Algorithm Development

Textbook: Sec. 2.1.2.1, 2.1.2.3, 2.1.2.7

2.1.3 System Components and Communication

Textbook: Sec. 2.1.2.4–2.1.2.6

2.1.4 Simulation Tools and Development Routes

Textbook: Sec. 2.1.2.1, 2.1.2.8

2.1.2 Model and Algorithm Development: Overview of Task Relationships

Models, controllers, and vision algorithms share a loop

Development object	Inputs and tools	Output and role in the loop
Motion model	Parameters, actuator commands; Simulink/C++	Model DLL → CopterSim state
Controller	State, targets; Simulink/C++	Control program → autopilot actuator commands
Vision algorithm	Images; OpenCV, Python/C++	Target information → high-level control targets
Optional: training	Samples; PyTorch/TensorFlow	Trained model → vision inference
As needed: acceleration	Model; graphics processing unit (GPU), CUDA/TensorRT	Faster inference → high-level algorithm

Models handle motion, controllers actuation, vision observations; train and accelerate as needed.

2.1.2 Model and Algorithm Development: Modeling and Simulation

MATLAB analyzes data; Simulink organizes models

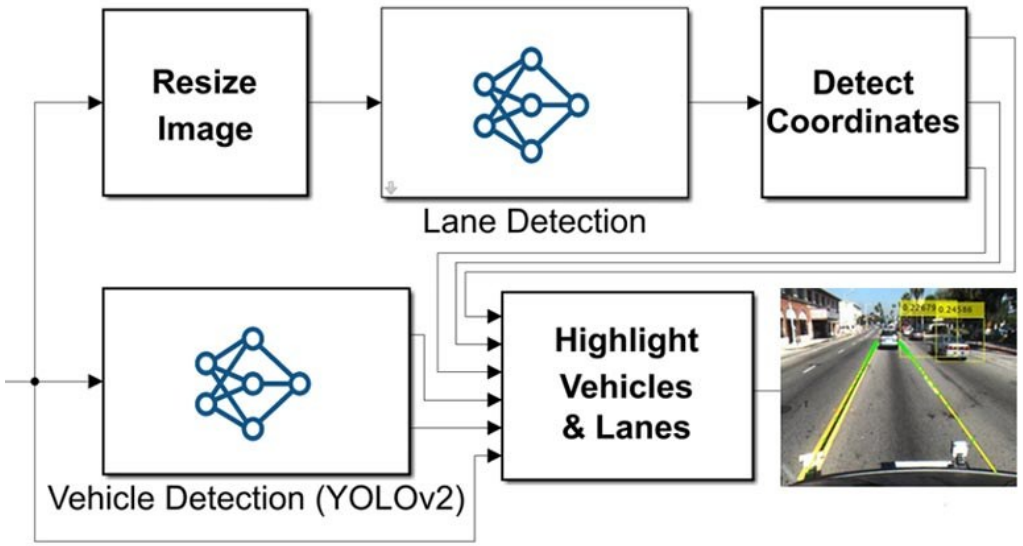


Figure: Lane and vehicle detection data flow

- **MATLAB:** Uses numerical computation and scripts to prepare inputs, tune parameters, and analyze errors and results.
- **Simulink:** Represents operations as blocks and data flow as signal lines, advancing the simulation over time.
- **Diagram:** The upper branch detects lanes, the lower branch detects vehicles, and the results are overlaid on the original image.

MATLAB handles numbers and scripts; Simulink uses blocks and signals for system simulation.

2.1.2 Model and Algorithm Development: Getting Started with Modeling Tools

First tasks in MATLAB and Simulink

- **Two tools:** MATLAB analyzes data with matrices and scripts;
  Simulink advances dynamic systems with blocks, signals, and solvers.
- **Model-based design (MBD):** One model supports design, simulation, and testing.
  Code generation converts supported blocks to C/C++, then compilation produces autopilot firmware or a model DLL.
- **Minimal exercise:** Plot a sine wave with MATLAB's plot. In Simulink, connect Sine Wave→Gain→Scope;
  changing the gain from 1 to 2 doubles the amplitude.

Interactive MATLAB introductory course
https://matlabacademy.mathworks.com/

Simulink modeling and simulation tutorials
https://www.mathworks.com/help/simulink/

Numerical scripts analyze data; block models reveal how system behavior changes over time.

2.1.2 Model and Algorithm Development: Models and Generated Outputs

Where motion models and controllers run

Plant

Motion model	Dynamic-link library (DLL)	CopterSim
Forces, torques, and motion state	Library with the required interface	Loads model and computes motion

Control algorithm

Controller model	Generate and compile code	Autopilot program
State input and control output	Toolchain and target settings	Executes the control algorithm

Model interface: CopterSim sends actuator commands → library computes and returns motion state.

Motion models describe the plant; control code implements the control law. Their destinations differ.

2.1.2 Model and Algorithm Development: C/C++ and Dynamic Libraries

How CopterSim loads a model DLL

- **Experiment entry:** In Chapter 2 basic experiment e2_DLL-Load, open Exp1_MinModelTemp_SITL.bat with a text editor.
- **Model selection:** DLLModel is set to Exp1_MinModelTemp, corresponding to Exp1_MinModelTemp.dll.
- **Import chain:** The startup script copies the model DLL to CopterSim/external/model and passes the model name to CopterSim.exe.
- **Runtime roles:** The model DLL updates the motion state through the platform interface; CopterSim schedules calls; RflySim3D updates the scene from that state.

CopterSim model DLL configuration and import: https://rflysim.com/doc/zh/3.Soft/coptersim.html

Model DLLs compute motion, CopterSim calls them, and 3D programs display state.

2.1.2 Model and Algorithm Development: Getting Started with OpenCV

OpenCV capabilities, advantages, and language interfaces

- **Capabilities:** The Open Source Computer Vision Library (OpenCV) provides image I/O, filtering, color processing, geometric transforms, camera calibration, feature detection, and other ready-made algorithms.
- **Language interfaces:** C++ and Python. Python supports interactive trials; C++ integrates with native projects.
- **Classical methods:** Thresholding and geometric measurement need no training samples. Visible intermediate results help establish a detection baseline.
- **Conditions:** Rules are sensitive to lighting and occlusion. Interfaces must agree on color, dimensions, and data types; measure processing speed in practice.

OpenCV modules and example source code: https://github.com/opencv/opencv

Ready-made algorithms and C++/Python ease vision work; reliable detection depends on data and scenes.

2.1.2 Model and Algorithm Development: Getting Started with OpenCV

Image arrays and color channels in OpenCV

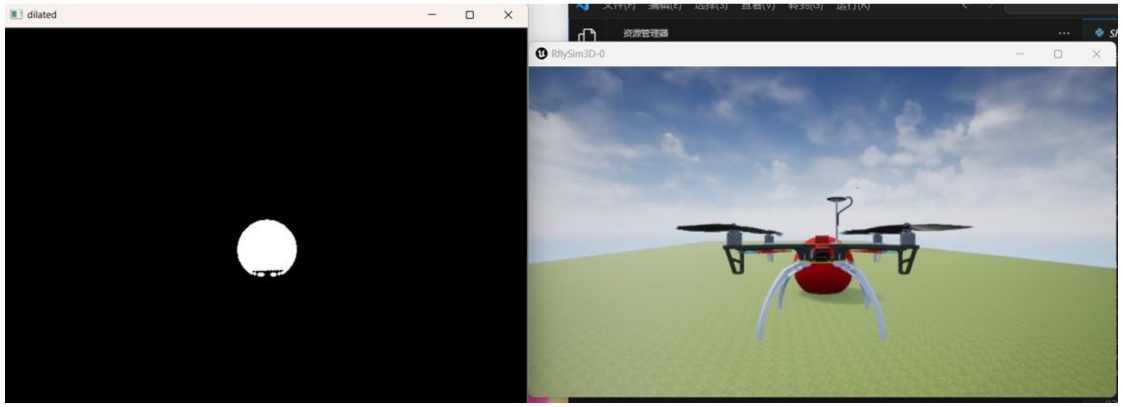
- **Image loading:** cv2.imread returns a NumPy array, or None for an incorrect path. Check the result before processing.
- **Array structure:** Python images are usually NumPy arrays. Color images use height, width, and channels; grayscale images are usually 2D. Index pixels by row, then column.
- **Color channels:** Each channel in an 8-bit color image ranges from 0 to 255. Default color loading uses blue-green-red (BGR); other interfaces may use red-green-blue (RGB).
- **Limits:** The interface specifies channel order; a filename does not. Convert explicitly when changing order. Pixel coordinates are not directly world positions.

Pixels, channels, and arrays: https://docs.opencv.org/4.x/d3/df2/tutorial_py_basic_ops.html

Image arrays use row, column, and channel indices; interfaces define color-channel order.

2.1.2 Model and Algorithm Development: Vision and Learning Frameworks

Different detection methods for the same vision task



Color segmentation → area and centroid

OpenCV

TensorFlow

- **OpenCV method:** Select and clean pixels by color, then compute target area and center.
- **Learning method:** Train a model on samples with PyTorch/TensorFlow, then recognize targets in new images.
- **Shared outputs:** Target position, extent, or class feeds subsequent perception, decision-making, and motion control.
- **Tradeoffs:** Thresholding avoids training; learned methods depend on samples that cover the actual scene conditions.

Color detection uses hand-written rules; learned models acquire detection capability from data.

2.1.2 Model and Algorithm Development: Getting Started with OpenCV

How color segmentation produces a target center

- **Color selection:** Convert the image to hue, saturation, and value (HSV), then threshold the target color to obtain a binary mask: white foreground, black background.
- **Noise cleanup:** Morphological opening removes small noise regions; closing fills small holes. Kernel size determines which details remain and must be chosen for the target size.
- **Center output:** Extract and filter contours by area, then use image moments to compute the centroid. Also output area and a validity flag for tracking or error computation.
- **Failure criteria:** Return invalid if no contour qualifies or area is zero. Lighting changes, occlusion, and similar-colored backgrounds interfere with detection. A 2D center contains no depth yet.

HSV color segmentation: https://docs.opencv.org/4.x/df/d9d/tutorial_py_colorspaces.html

Color selection, morphological cleanup, and contour measurement yield a target center.

2.1.2 Model and Algorithm Development: Getting Started with OpenCV

Color-segmentation stages and invalid detections

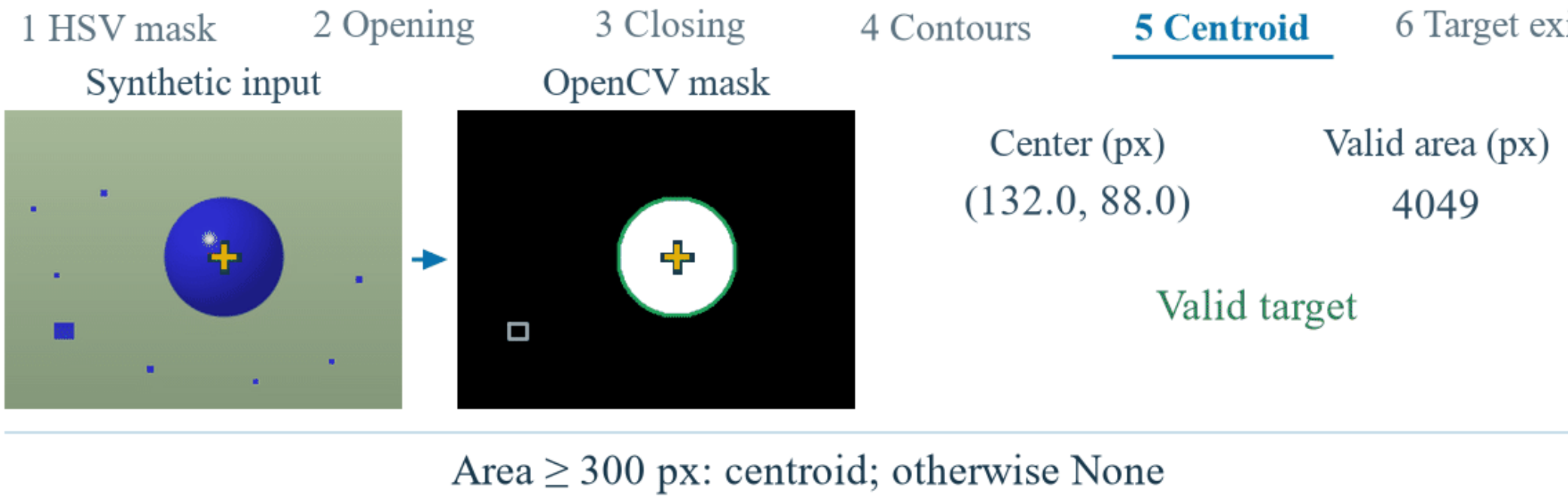


Figure: Step-by-step OpenCV processing of a synthetic image (teaching illustration)

Report missing targets explicitly so visual control cannot reuse invalid centroids as new observations.

2.1.2 Model and Algorithm Development: A Minimal OpenCV Exercise

Image input, measurement, and failure checks

- **Check input:** Generate an image with one blue rectangle in code.
  If loading a file instead, check that the result is not None, then print shape and dtype.
- **Inspect stages:** For an 8-bit OpenCV image converted to HSV, H ranges from 0 to 179, and S and V from 0 to 255. Select blue and compare the original, mask, and center marker.
- **Check results:** Rectangle corners (80,60) and (140,100) should give a center near (110,80), in pixels.
- **Handle failure:** An all-black image must produce an invalid flag. With no contour or zero area, do not compute a centroid or reuse an old observation.

Mask morphology: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html

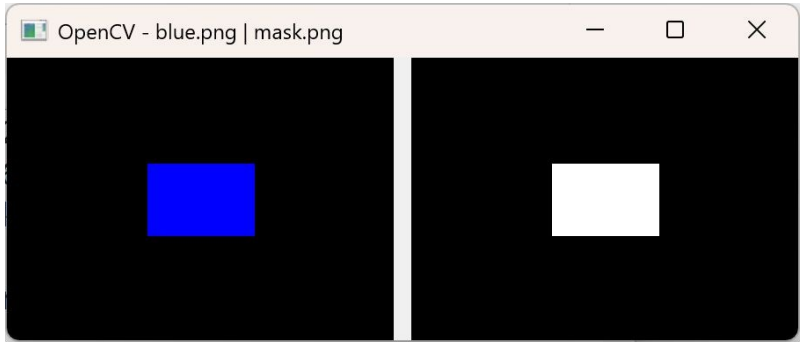
Check intermediate results and failures on static images before processing simulation image streams.

2.1.2 Model and Algorithm Development: A Minimal OpenCV Exercise

Generating an image and extracting a blue mask

```
import cv2
import numpy as np
img = np.zeros((160, 220, 3), dtype=np.uint8)
cv2.rectangle(img, (80, 60), (140, 100), (255, 0, 0), -1)
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv, (110, 100, 100), (130, 255, 255))
cv2.imwrite("blue.png", img)
cv2.imwrite("mask.png", mask)
```

Figure: OpenCV output windows



- **Result:** This directory contains the blue-rectangle image and white-rectangle mask. Next, measure the center.

Drawing functions: https://docs.opencv.org/4.x/dc/da5/tutorial_py_drawing_functions.html

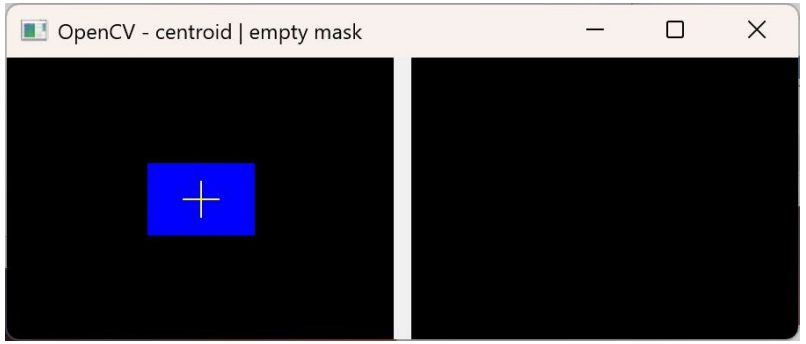
Color matches are only candidates; check whether their area and position are credible.

2.1.2 Model and Algorithm Development: Minimal OpenCV Exercise

Compute the Centroid and Test the No-Target Branch

```
def center(mask):
    m = cv2.moments(mask, binaryImage=True)
    if m["m00"] == 0:
        return None
    return (m["m10"] / m["m00"], m["m01"] / m["m00"])
print(center(mask))
print(center(np.zeros_like(mask)))
```

Figure: Centroid and Empty-Mask Window



- Continue directly after the previous page's code. Check the outputs for these two inputs.
- **Blue rectangle:** divide first moments by area to obtain the centroid; expect (110.0, 80.0).
- **All-black mask:** return None when the area is zero to mark the observation as invalid.

Moments and centroids: https://docs.opencv.org/4.x/dd/d49/tutorial_py_contour_features.html

2.1.2 Model and Algorithm Development: Optional: Learned Vision

Training Frameworks, Inference, and Vision Libraries

- **Training:** PyTorch and TensorFlow provide tensor operations, automatic differentiation, and model components;
  update parameters using samples and a loss function.
- **Inference:** at deployment, a trained model processes new images and outputs classes, bounding boxes, or masks, usually without updating parameters.
- **OpenCV support:** read and resize images, convert colors, and display results; its deep neural network (DNN) module can also run supported models.
- **Learning sequence:** first run inference on one image and check the inputs and outputs, then connect live images and motion control.

PyTorch tensors and model training
https://pytorch.org/tutorials/

TensorFlow training and prediction
https://www.tensorflow.org/tutorials

Training learns parameters; inference yields target data; vision libraries pre/postprocess and display.

2.1.2 Model and Algorithm Development: Optional: Inference Acceleration

Parallel Computing and Model Inference Acceleration

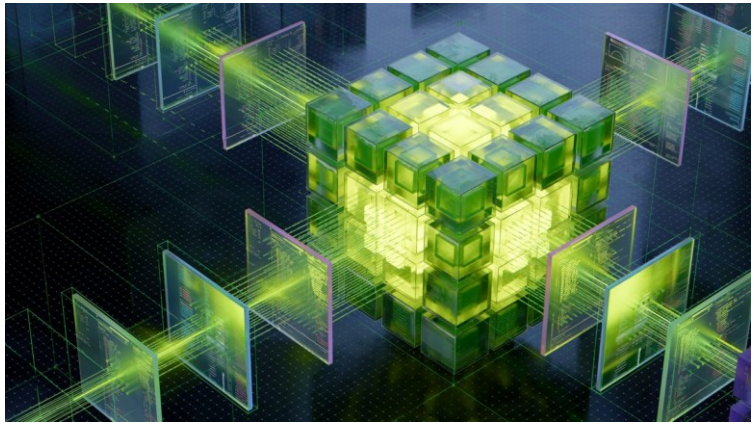


Figure: Parallel GPU Computation

- **Parallel hardware:** a graphics processing unit (GPU) handles many parallel operations; CUDA provides NVIDIA's GPU computing platform.
- **Inference optimization:** TensorRT optimizes and executes trained models to turn new images into results such as detections.
- **Total latency:** preprocessing, data transfer, model inference, and postprocessing together determine output speed.

Preprocessing → Data Transfer → Model Inference → Postprocessing

CUDA docs: https://docs.nvidia.com/cuda/

TensorRT docs: https://docs.nvidia.com/tensorrt/

Parallel hardware and optimized inference cut compute time; total delay includes transfer/processing.

2.1.2 Model and Algorithm Development: Parallel Computing / Faster Inference (Optional)

Code Examples: GPU Computing, Inference, and Training

CUDA parallel computing examples: https://github.com/NVIDIA/cuda-samples

Start with device queries, vector operations, and data transfers to understand cooperation between the host and GPU.

TensorRT inference examples: https://github.com/NVIDIA/TensorRT

Use the samples directory to study model inputs, execution engines, and outputs; check versions and runtime dependencies.

PyTorch training examples: https://github.com/pytorch/examples

Start with small examples such as image classification to trace data, models, loss, and parameter updates. Read the README first.

TensorFlow models and examples: https://github.com/tensorflow/models

Choose a model example for your task. Check dependencies and data, then distinguish the training and inference entry points.

Faster model computation shortens inference only; the full vision chain also has costs such as data transfer.

Chapter Outline

Key Tools: From Runtime Environments to Closed-Loop Simulation

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.1.1 Development Basics and Runtime Environments

Textbook: Sec. 2.1.2.2, 2.1.2.3, 2.1.2.7

2.1.2 Model and Algorithm Development

Textbook: Sec. 2.1.2.1, 2.1.2.3, 2.1.2.7

2.1.3 System Components and Communication

Textbook: Sec. 2.1.2.4–2.1.2.6

2.1.4 Simulation Tools and Development Routes

Textbook: Sec. 2.1.2.1, 2.1.2.8

2.1.3 System Components and Communication: Roles in the Ecosystem

What Are Pixhawk, PX4, and a Ground Station?





Autopilot is this book's general term; UAV users commonly call it a flight controller.

Component	Layer	Role in the System
Pixhawk	Hardware standards and ecosystem	Compatible boards provide computation, sensing, and outputs
PX4 / **ArduPilot**	Autopilot software	State estimation, control, missions, and failsafes
QGroundControl	Ground station software	Configuration, mission planning, and monitoring
MAVLink	Messaging protocol	Defines command, state, and other message formats

Hardware runs autopilot software; ground stations use message interfaces for setup and monitoring.

2.1.3 System Components and Communication: Autopilot Ecosystem Basics

Resources for Hardware, Firmware, and Ground Stations

Pixhawk hardware standards and ecosystem: https://pixhawk.org/

Identify the control board, sensor inputs, and actuator outputs. Understand the hardware's role first.

PX4 vehicle configuration and control: https://docs.px4.io/main/zh/

Start with Getting Started to learn about vehicle types, firmware, state estimation, and control.

ArduPilot vehicle manuals: https://ardupilot.org/

Choose the manual for the vehicle type to understand shared capabilities and configuration differences.

QGC setup, planning, and monitoring: https://docs.qgroundcontrol.com/

Use the interface overview to distinguish configuration, mission planning, and runtime observation.

Hardware supplies compute/I/O; firmware implements the autopilot; ground stations configure/observe.

2.1.3 System Components and Communication: Onboard Computing

Autopilot and Onboard Computer: Their Roles

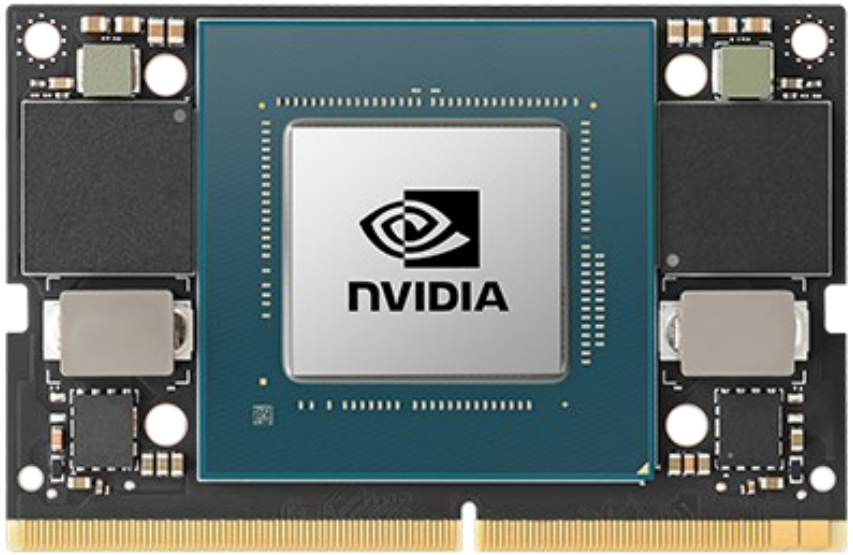


Figure: NVIDIA Jetson Onboard Computing Module

- **Autopilot:** receives sensor data, estimates state, runs low-level control, and outputs actuator commands.
- **Onboard computer:** processes images and point clouds for recognition, path planning, and mission logic.
- **Red-ball tracking:** the upper layer computes velocity targets from the target centroid and area; the autopilot controls vehicle motion.
- **Feedback and constraints:** position, velocity, and mode return to the upper layer; compute capacity, power, cooling, and interfaces constrain hardware selection.

Perception/planning set motion targets; autopilot uses them and state estimates for actuator commands.

2.1.3 System Components and Communication: Ground Station

Read State, Request Actions, and Observe the Response

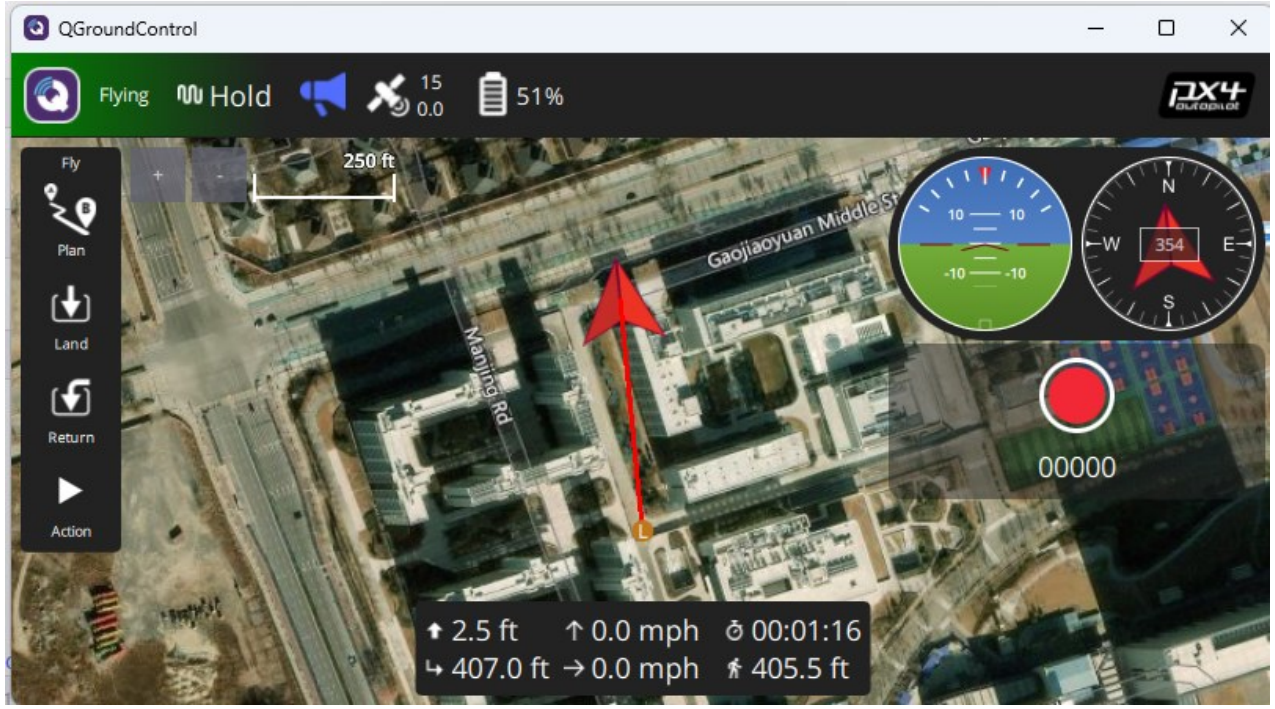


Figure: QGroundControl (QGC) Fly View

- **Status bar:** connection, mode, battery, and other operating states appear at the top.
- **Action panel:** takeoff and return requests are sent here; execution depends on the autopilot's state.
- **Map and instruments:** the map shows position; the instruments show attitude.

The ground station unites system state, operator commands, and vehicle response in one view.

2.1.3 System Components and Communication: Ground Station Workflow

QGC Supports Mission Preparation, Operation, and Review

- **Parameter configuration:** set vehicle type, sensors, and control parameters supported by the autopilot; values should match the actual vehicle and simulation task.
- **Mission planning:** edit waypoints and mission items on the map; check altitude, sequence, and the final action before uploading.
- **Runtime monitoring:** observe position, mode, and battery telemetry; compare targets with actual responses and inspect messages when problems occur.
- **Log review:** logs store state and control information over time. Align targets, actual responses, and mode changes to locate the source of response lag.

QGC mission planning and interface source
https://github.com/mavlink/qgroundcontrol

Online PX4 flight-log analysis
https://logs.px4.io/

Ground stations organize setup, missions, and observation; autopilots estimate state and control motion.

2.1.3 System Components and Communication: Messaging Protocol

MAVLink Defines Message Meaning; the Link Carries It



Upper-Level Program / Ground Station	Serial or Network Link	Autopilot
Encode control targets ➡	Carry MAVLink messages ➡	Decode and execute control

Feedback: attitude, position, velocity, mode, and other state

- **Message content:** a velocity-target message includes the coordinate frame, vx/vy/vz (m/s), and target vehicle ID.
- **Transport channel:** match the port and baud rate for serial links, or the address and port for network links.
- **State feedback:** the autopilot reports actual motion and mode; the upper layer updates targets accordingly.

Message formats define value meanings; transport channels deliver messages to the destination program.

2.1.3 System Components and Communication: Inter-Process Communication

Library Calls and Inter-Process Communication

- **Programs and processes:** a program is executable code; running it creates a process. Processes usually have separate address spaces.
- **Library calls:** a host process loads a DLL and calls its functions; parameter and return-value conventions define data exchange.
- **Exchange between processes:** inter-process communication (IPC) lets independent programs exchange data or synchronize;
  common methods include pipes, sockets, and shared memory.
- **In this platform:** CopterSim calls a model DLL for computation within the host process; exchanging images and state among programs still requires communication interfaces.

Python socket communication: https://docs.python.org/3/library/socket.html

Function calls connect code within a process; IPC connects independently running programs.

2.1.3 System Components and Communication: Transport Channels

Choosing Serial, Network, or Shared Memory

Channel	Transport Mechanism	Uses and Tradeoffs in Unmanned Systems
Serial	Device link carrying bytes	Connect a computer to an autopilot; match baud rate and format at both ends.
User Datagram Protocol (UDP)	Datagram boundaries kept No delivery/order guarantee	RflySim components exchange state/commands; app handles loss and freshness.
Transmission Control Protocol (TCP)	Reliable, ordered byte stream Can retransmit lost data	For data that must arrive in full; retransmission may add waiting time.
Shared Memory	Local processes share a buffer	Red-ball images: RflySim3D → Python; manage buffers and synchronize reads/writes.

TCP: reliable and ordered; UDP: app handles loss; shared memory suits large data on one computer.

2.1.3 System Components and Communication: Transport Channels

UDP and TCP Delivery Under the Same Packet Loss

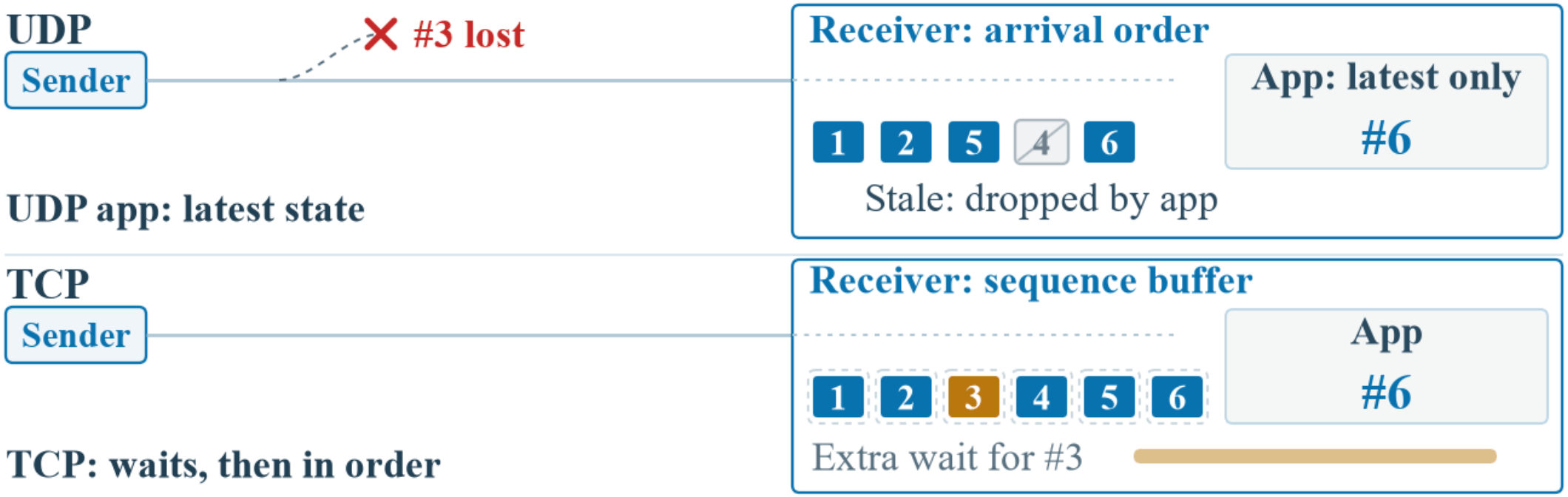


Figure: Same Network Conditions—Packet 3 Lost, Packet 4 Delayed (Teaching Illustration)

With equal packet loss, the UDP app keeps the latest state; TCP fills gaps before ordered delivery.

2.1.3 System Components and Communication: ROS Basics

ROS Nodes and Publish–Subscribe Communication

- **Organization:** ROS provides communication libraries, development tools, and a package ecosystem for independently developing and reusing perception, planning, and control modules.
- **Node roles:** a node is a computational unit with one logical responsibility. Nodes can work on one or several computers; camera capture, detection, and display can be separate.
- **Publish–subscribe:** publishers send messages on named topics; subscribers receive by topic and type. One image source can feed detection, display, and recording at once.
- **Message model:** messages have typed fields. An image includes dimensions, encoding, byte data, capture time, and coordinate frame so programs interpret it consistently.

ROS 2 Jazzy introductory tutorials
https://docs.ros.org/en/jazzy/

ROS 1 node and topic tutorials
https://wiki.ros.org/ROS/Tutorials

ROS connects nodes with typed messages so multiple programs can share observations and results.

2.1.3 System Components and Communication: ROS Concepts

How ROS Organizes Robot Programs

ROS: a robot software framework running on top of the operating system

A node is a functional unit, not a process; several nodes can share one process.



Concept	Purpose	Example
Topic	Publish and subscribe by message type	Camera images, odometry state
Service	One request and response	Query configuration, trigger an operation
Action	Long task with feedback and cancellation	Navigate to a target position
Parameter	Store configurable node values	Frame rate, thresholds, coordinate settings

ROS organizes functions into nodes and uses topics, services, and actions for different interactions.

2.1.3 System Components and Communication: ROS Concepts

Message Timing for Topics, Services, and Actions

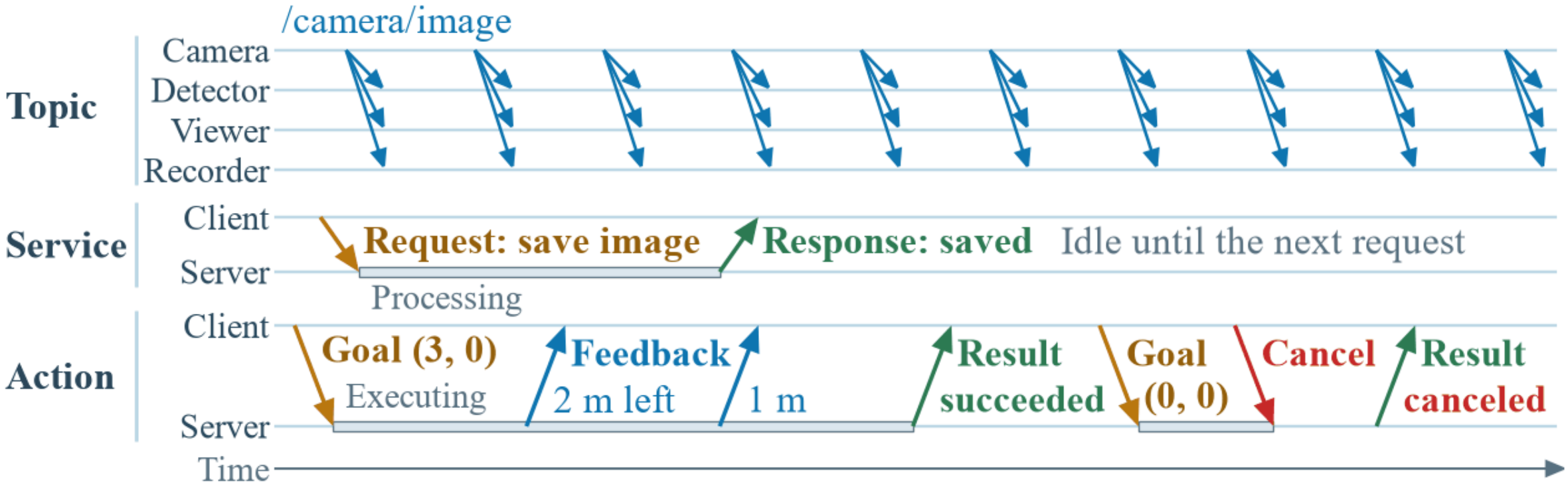


Figure: Three Interactions on One Timeline; Message Transit Is Slowed Down (Teaching Illustration)

2.1.3 System Components and Communication: ROS Versions

ROS 1 and ROS 2: Shared Concepts, Different Implementations

Aspect	ROS 1	ROS 2
Core concepts	Nodes, topics, services, etc.	Retains concepts and extends capabilities
Communication and discovery	Master usually coordinates discovery Nodes exchange messages directly	ROS Middleware (RMW) interface Commonly Data Distribution Service (DDS)
Build ecosystem	Usually catkin and related tools	Usually ament and colcon
Migration needs	Legacy package dependencies and message interfaces	Client APIs, package builds, and communication settings

RMW separates communication back ends; DDS middleware provides pub/sub and discovery.

catkin and ament support package builds; colcon coordinates builds across multiple packages.

Both ROS generations keep nodes/messages; communication implementations and build tools change.

2.1.3 System Components and Communication: ROS Basics

Match the ROS Version to the Platform Environment

- **Existing courses:** reproduce experiments in the validated environment. ROS 1 Noetic reached end of official maintenance in May 2025; Ubuntu 22.04 onward has no official ROS 1 packages.
- **New projects:** choose a stable ROS 2 release for your dependencies, such as Jazzy on Ubuntu 24.04 or Humble on 22.04, and check support dates.
- **Platform connection:** WinWSL hosts Linux and ROS; the image interface publishes observations, and nodes process them before connecting to the course control interface.
- **Communication compatibility:** topics, types, and networking define the data path. Quality of service (QoS) also sets delivery policies; best-effort publication cannot meet a reliable-reception requirement.

ROS 2 releases and target platforms
https://www.ros.org/reps/rep-2000.html

RflySim's accompanying WinWSL environment
https://rflysim.com/doc/zh/3.Soft/winwsl.html

Reuse validated course dependencies; for new work, match a stable release to the OS and packages.

2.1.3 System Components and Communication: ROS Basics

ROS Visualization and Topic Recording

- **Communication checks:** use the command line to list nodes and topics and inspect message types, values, and rates. rqt_graph shows who publishes and who subscribes.
- **Visualization:** RViz displays images, point clouds, and coordinate transforms from input data. Incorrect fixed frames or timestamps can misalign the display.
- **Recording and playback:** ROS 1 uses rosbag; ROS 2 uses ros2 bag. Save topic data and replay it over time to retune repeatedly, share observations, and compare algorithms.
- **Reproduction limits:** a bag saves selected messages, not the whole simulation world or every parameter. Retain images, camera parameters, transforms, and algorithm settings as the experiment requires.

RViz 3D visualization source (ROS 2)
https://github.com/ros2/rviz

ROS 1 recording and communication source
https://github.com/ros/ros_comm

RViz displays spatial messages; bags save observations for repeated algorithm processing/comparison.

2.1.3 System Components and Communication: Minimal ROS Exercise

Discover, Inspect, and Replay Messages

- **Environment:** use a separate Jazzy installation with turtlesim. In each new Bash terminal, run source /opt/ros/jazzy/setup.bash, then start the simulation and keyboard nodes.
- **List and inspect:** in ROS 2, use ros2 topic list, info, and echo to inspect topics and data; ROS 1 uses rostopic.
- **Record and replay:** record /turtle1/pose; use a new directory name for each recording. Stop the source before playback to receive the same pose messages again.
- **Meaning:** /turtle1/pose publishes pose observations; /turtle1/cmd_vel receives velocity commands. This exercise replays observations.

ROS 2 record/replay examples
https://github.com/ros2/rosbag2

ROS 1 turtlesim source (Noetic)
https://github.com/ros/ros_tutorials/tree/noetic-devel

Pose topics report motion results; velocity topics carry motion commands. The flow directions differ.

2.1.3 System Components and Communication: ROS 2 Command Exercise

Inspect Topics, Record, and Replay with Jazzy

01 Two terminals **Start separately**	ros2 run turtlesim turtlesim_node ros2 run turtlesim turtle_teleop_key
02 New terminal **Inspect / subscribe**	ros2 topic info /turtle1/pose ros2 topic echo /turtle1/pose
03 New terminal **Record; move; stop**	ros2 bag record -o pose_demo /turtle1/pose Move with the keyboard; press Ctrl+C in the recording terminal to stop.
04 Stop sources **Replay / observe**	ros2 bag play pose_demo Stop the simulation and keyboard nodes first; keep echo running to observe.

ROS 2 communication examples (Jazzy branch): https://github.com/ros2/demos/tree/jazzy

Pose recordings repeat the same observations to debug subscribers and compare algorithms.

2.1.3 System Components and Communication: Adapter Libraries

Using MAVLink in a Program

Adapter libraries handle message transmission and reception, connecting application functions to a MAVLink link.

Tool	Main Use	Suitable Tasks
Pymavlink	Handle MAVLink messages directly in Python	Inspect messages and implement specific exchanges
MAVSDK	Use higher-level APIs for telemetry, actions, and missions	Quickly implement supported application functions
MAVROS	Bridge ROS and MAVLink systems	Connect an autopilot to a ROS node network

Choose a MAVLink interface level for message inspection or higher-level function calls.

2.1.3 System Components and Communication: Communication Resources

MAVLink Messaging and Adapter-Library Examples

MAVLink message fields and protocol: https://mavlink.io/en/

Read message definitions: fields, units, and coordinate frames; distinguish targets from actual state.

Pymavlink send/receive tutorials: https://mavlink.io/en/mavgen_python/

Start with connection and message-reception examples to understand message types and timeouts.

MAVSDK connection and telemetry tutorials: https://mavsdk.mavlink.io/

Start with QuickStart and telemetry examples to relate higher-level APIs to underlying messages.

MAVROS node and topic source: https://github.com/mavlink/mavros

Read the README and topic interfaces first; use ROS nodes to connect to autopilot messages.

One protocol can offer multiple language/framework interfaces; choose for your application's needs.

2.1.3 System Components and Communication: External Control: MAVLink Example

The Upper Layer Sets Targets; the Autopilot Controls Motion

Vision Algorithm	→	PX4 Autopilot	→	Vehicle / Motion Model
Centroid and area Compute motion targets		State estimation and control Generate actuator commands		Motion response Produces a new image

Observation feedback: image-center error, target area, and actual motion state

- **Velocity interface:** SendVelFRD(vx, vy, vz, yawrate): velocity in m/s, yaw rate in rad/s.
- **Directions:** forward, right, down (FRD); vz < 0 requests ascent.
- **External control:** Offboard mode continuously receives targets; the autopilot handles low-level control.

Visual error becomes a motion target; the autopilot turns that target into actuator commands.

2.1.3 System Components and Communication: External-Control Mode

Offboard Target Streams and Autopilot Response

- **Control roles:** the external program supplies position or velocity targets; PX4 uses state estimates to compute control outputs. Actual velocity depends on controller and vehicle response.
- **Mode conditions:** position and velocity control need valid position and velocity estimates, respectively. Establish the external-control keepalive stream before entry and maintain it afterward.
- **Link loss:** MAVLink target messages also provide keepalive; ROS 2 sends OffboardControlMode separately. A keepalive timeout exits the mode and triggers the configured action.
- **Stale targets:** a background thread can keep sending a cached target after vision updates stop. Check image timestamps; on timeout, replace the old target and enter the predefined hold or landing procedure.

PX4 Offboard requirements: https://docs.px4.io/main/en/flight_modes/offboard

If background sending outlasts vision updates, the vehicle may keep following an old target.

2.1.3 System Components and Communication: External-Control Mode

Why Old Targets May Persist When Vision Stops Updating

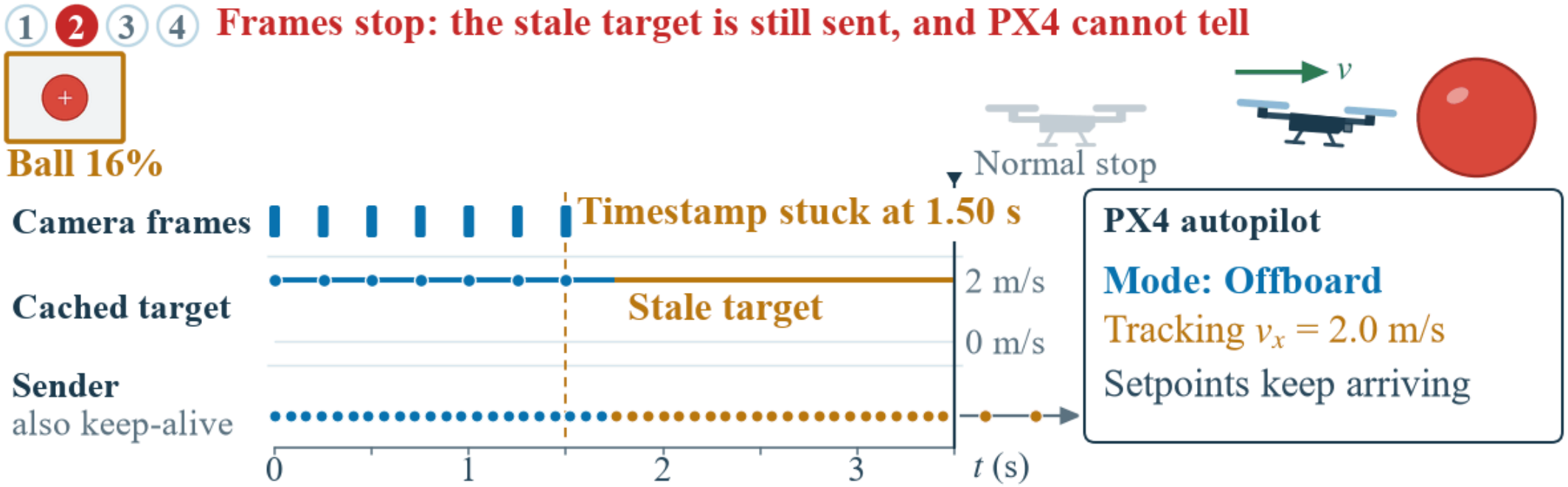


Figure: Four Cases from the Same Starting State (Simplified Simulation, Not Measurements)

Compare frame times and send markers: continued packets do not prove continued vision updates.

Chapter Outline

Key Tools: From Runtime Environments to Closed-Loop Simulation

Chapter 2 Experimental Platform Configuration

2.1 Key Tools

2.2 Installation and First Simulation

2.3 Architecture and Workflows

2.4 API and SDK Interfaces

2.5 Examples and Experimental Routes

2.6 Exercises and Discussion

4 Units in This Section

2.1.1 Development Basics and Runtime Environments

Textbook: Sec. 2.1.2.2, 2.1.2.3, 2.1.2.7

2.1.2 Model and Algorithm Development

Textbook: Sec. 2.1.2.1, 2.1.2.3, 2.1.2.7

2.1.3 System Components and Communication

Textbook: Sec. 2.1.2.4–2.1.2.6

2.1.4 Simulation Tools and Development Routes

Textbook: Sec. 2.1.2.1, 2.1.2.8

2.1.4 Simulation Tools and Development Routes: In-the-Loop Simulation

Software-in-the-Loop vs. Hardware-in-the-Loop

Compare in-the-loop modes, then scene and physics tools, and finally choose a route by task.

Aspect	Software-in-the-Loop (SITL; textbook: SIL)	Hardware-in-the-Loop (HITL; textbook: HIL)
Control program	Runs in a computer environment	Runs on real target hardware
Plant	Computer-simulated model	Computer-simulated model
Main use	Rapid algorithm and functional iteration	Onboard execution, timing, and interface integration
Not covered	Real hardware timing and the physical environment	Real propulsion, environment, and whole-vehicle behavior

SITL runs control software on a computer; HITL places real control hardware in the closed loop.

2.1.4 Simulation Tools and Development Routes: Scene and System Simulation

3D Engines and Robot Simulation Platforms

- **Unreal Engine:** a general-purpose 3D engine using geometry, materials, lighting, and cameras to generate visual observations.
- **Unity:** a general-purpose 3D engine; the optional ML-Agents toolkit connects scenes to policy training.
- **Gazebo and Webots:** robot system simulators connecting vehicle models, virtual sensors, and control programs.
- **RflySim roles:** CopterSim computes motion state; RflySim3D uses it to update the scene and camera images.
- **Red-ball loop:** motion changes the target's image position and size; the vision program updates control targets accordingly.

3D engines focus on scenes and interaction; robot simulation also connects physics, sensors, and control.

2.1.4 Simulation Tools and Development Routes: 3D Scenes

Roles of Unreal Engine and RflySim3D

- **Scene engine:** Unreal Engine (UE) organizes geometry, materials, lighting, and cameras to render images.
- **Platform integration:** RflySim3D uses UE for visualization and virtual sensors; it receives vehicle poses and updates the scene.
- **Motion computation:** CopterSim computes vehicle motion. State drives the display, and the vision program reads camera images.
- **Perception effects:** lighting changes can affect color segmentation; model parameters change motion first, then the images.

RflySim3D scenes and interfaces: https://rflysim.com/doc/zh/3.Soft/3d-eng.html

UE scenes, materials, and cameras: https://dev.epicgames.com/documentation/unreal-engine/

Motion models supply vehicle state; 3D engines turn scene and camera conditions into usable observations.

2.1.4 Simulation Tools and Development Routes: Optional: Scenes and Robot Simulation

Getting Started with Scenes and Robot Simulation

UE scene-building courses: https://dev.epicgames.com/community/learning

Select Unreal Engine in the courses to learn object, level, material, and camera settings.

Unity editor and interaction courses: https://learn.unity.com/

Start with the editor and basic projects to learn scenes, scripts, and interaction.

Gazebo model and sensor tutorials: https://gazebosim.org/docs/

Use Getting Started to learn worlds, models, sensors, and control interfaces.

Webots robot and controller tutorials: https://cyberbotics.com/doc/guide/index

Use Tutorials to learn how robots, environments, and controllers connect.

Choose one scene-building or robot-simulation tool for your task; complete its smallest working example.

2.1.4 Simulation Tools and Development Routes: Optional: Robot Learning

Roles of Robot Learning Tools

In reinforcement learning, for example, a policy selects actions and the training algorithm updates it using rewards.

Tool	Main Role	Typical Tasks
MuJoCo / MJX	Multibody dynamics and contact MJX supports parallel simulation	Joint control, contact motion, and policy training
Isaac Sim / Lab	Sim builds simulation environments Lab organizes learning tasks	Robot perception and motion control
Genesis	Rigid- and soft-body physics, etc. Build robot interaction environments	Arm manipulation and rigid–soft interaction

Simulation supplies post-action state and observations; task rewards guide policy learning.

2.1.4 Simulation Tools and Development Routes: Optional: Robot Learning Resources

Resources for Physics Simulation and Policy Training

MuJoCo models and MJX documentation: https://mujoco.readthedocs.io/

Start with Overview and model examples; use MJX for related parallel simulation workflows.

Isaac Sim scene and sensor tutorials: https://docs.isaacsim.omniverse.nvidia.com/

Use Getting Started to understand scenes, robot models, and sensors.

Isaac Lab task and policy-training tutorials: https://isaac-sim.github.io/IsaacLab/

Use Tutorials to learn task environments, observations, actions, rewards, and training interfaces.

Genesis entity and simulation tutorials: https://genesis-world.readthedocs.io/

Start with basic User Guide examples to understand entities, physical properties, and simulation steps.

Simulation gives state changes and observations; training frameworks organize tasks and policy updates.

2.1.4 Simulation Tools and Development Routes: Development Routes

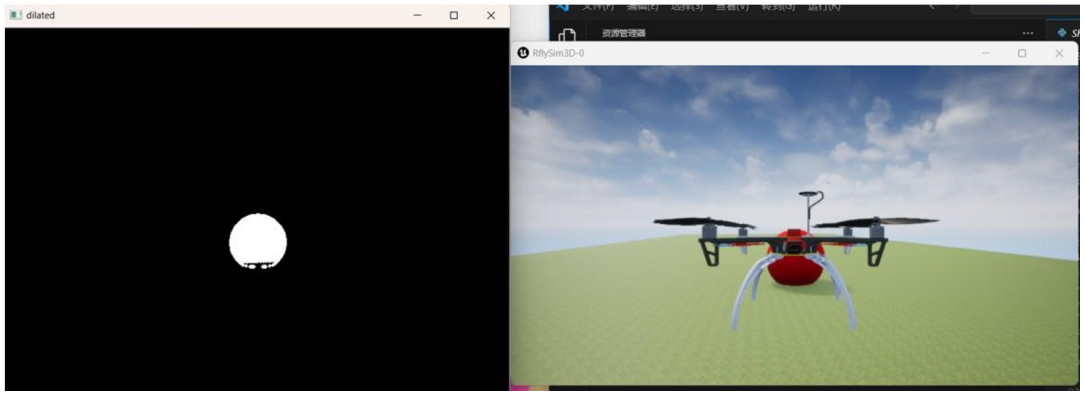
Choose Tools and Deliverables by Development Task

Task	Main Tools	Deliverables
Reproduce basic simulation	Unified launcher, CopterSim, ground station	Run record and state interpretation
Modify visual tracking	Python, OpenCV, platform APIs	Detection/control scripts and comparison results
Modeling and control design	MATLAB, Simulink, compilation tools	Model DLL or control program
Integrate robot modules	Linux, ROS, and adapter interfaces	Nodes, messages, and integration settings
Learned perception and decision-making	Training frameworks, simulation, deployment tools	Data, models/policies, and evaluation

Identify whether the change concerns observation, motion, or control, then select the corresponding tools.

2.1.4 Simulation Tools and Development Routes: Connecting the Example

Red-Ball Tracking: From Target Data to Motion



The white mask marks the red-ball region

- **Acquire observations:** VisionCaptureApi reads camera images.
- **Locate the target:** calc_centroid computes its area and centroid.
- **Generate actions:** controller converts error into velocity targets.
- **Close the loop:** SendVelFRD sends targets; simulation returns new images.

Example index: Chapter 1 → Basic Experiments → e7_RunPythonProject

Vision extracts target error; control changes motion; new images feed back the tracking outcome.

2.1.4 Simulation Tools and Development Routes: Single-Parameter Comparison

Change One Threshold, Predict the Motion, Then Test It

Compare max_prop in ShootBall3.py: the target-area fraction threshold

Aspect	Case A: 0.30	Case B: 0.20
Initial conditions	Same red ball, vehicle position, and camera settings	Use the same conditions as Case A
Command rule	Stop the forward command at 30% target area	Stop the forward command at 20% target area
Prediction to test	Record the stop position as the baseline	Decelerate earlier and stop farther from the ball

Record target fraction, velocity targets before/after smoothing, and actual motion on one timeline to distinguish command from response.

Area threshold zeros the forward target; command smoothing and vehicle response affect deceleration.

2.1.4 Simulation Tools and Development Routes: Single-Parameter Comparison

Area Threshold: Trigger Time and Stopping Position

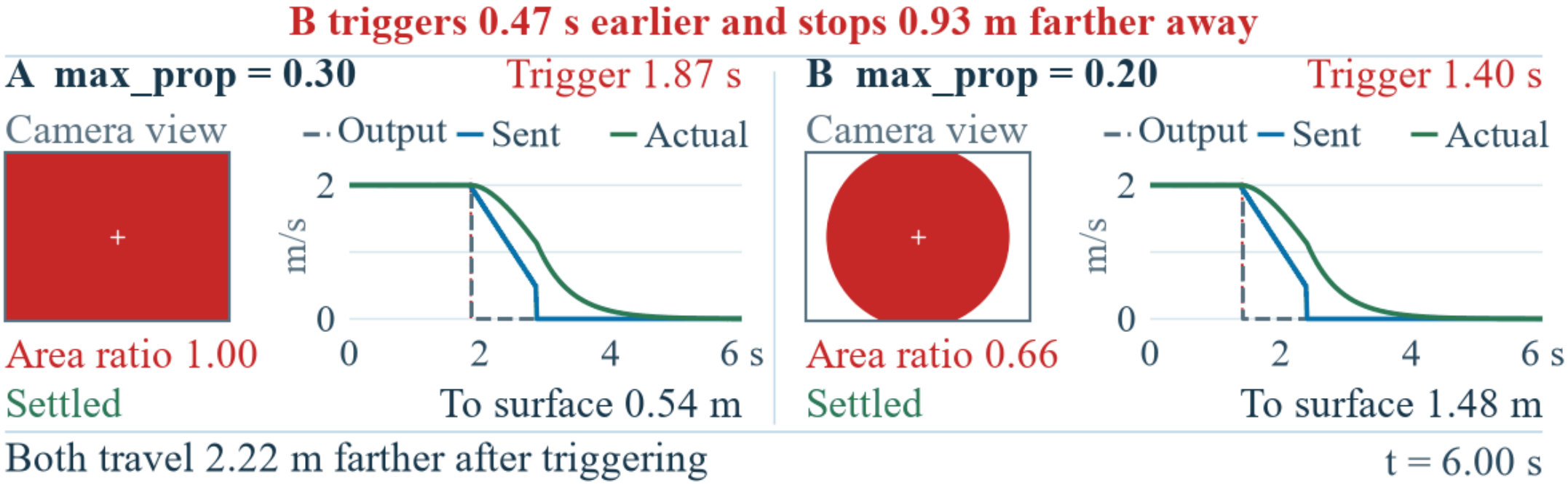


Figure: Only the Area Threshold Changes—A: 0.30, B: 0.20 (Simplified Simulation, Not Measurements)

Compare three curves: a lower threshold ends the forward command sooner; the vehicle then slows.

2.1.4 Simulation Tools and Development Routes: Official Learning Resources

Use Official Resources to Answer a Specific Question

Current Question	What to Check First	Resource
How does the example run?	Readme, startup file, expected behavior	RflySim documentation
How is a function used?	Parameter types, return values, minimal example	Python / OpenCV
How is a target interpreted?	Message fields, coordinate frames, mode support	MAVLink / PX4
How can the research expand?	Physics, sensors, interfaces, training tasks	Simulation and robot learning documentation

RflySim course and experiment index: https://rflysim.com/doc/zh/

Read the course example first, then look up the functions, messages, and operating conditions it uses.

2.1.4 Simulation Tools and Development Routes: Learning in Stages

Reproduce, Modify Algorithms, and Develop Further

- **Reproduce first:** run an example through the unified launcher, inspect state, and explain the script's role and experimental result.
- **Then modify the algorithm:** learn Python variables, functions, and arrays; process one image, then explain how OpenCV output affects control targets.
- **Add foundations by task:** learn C++ and DLLs for model development, or ROS and message interfaces for robot-module integration.
- **Extend the research last:** study scene creation, GPU inference, and policy training as needed; record inputs, outputs, operating conditions, and comparison results.

RflySim feature demonstrations and case videos
https://rflysim.com/doc/zh/0.Show/videos.html

ML-Agents policy training (optional)
https://unity-technologies.github.io/ml-agents/

Add tools gradually for the current task; check understanding with a small observable result at each step.

Reflection and Transition

Can You Explain the Tools and Data in Red-Ball Tracking?

- **Tool roles:** who generates and processes the red-ball images? Which parts of the response belong to PX4 and CopterSim?
- **Interfaces:** if color segmentation is replaced with a learned model, which input/output conventions must remain consistent?
- **Results:** after max_prop triggers and the velocity target reaches zero, the vehicle still moves. What should you observe?
- **Learning order:** which tools are needed now to reproduce this Python example, and which can wait for extensions?

Next: 2.2 Installation and First Simulation

Only by tracing images, targets, and motion feedback can you truly grasp the example's division of work.