4.9 Modeling Tools and Deployment

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

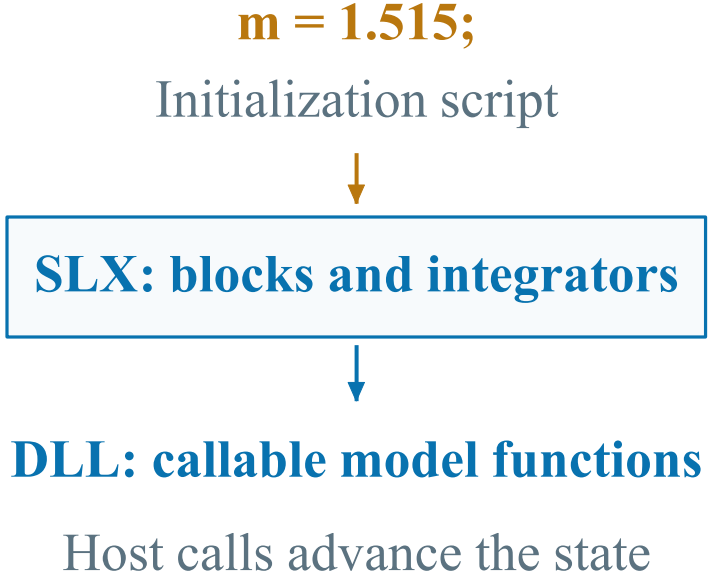
4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.1 Tools and the Execution Chain: Equations, Project, and Host

From Equations to Execution: Scripts, Models, and Hosts

- MATLAB scripts fix parameters, initial values, and inputs. Simulink expresses equations with blocks and signals; its solver advances the states.
- SLX stores model connections and solver settings. C/C++ code is compiled and packaged for the platform as a dynamic-link library (DLL), called by a host.
- For example, change mass in the initialization script, run, and save logs. Changing only the 3D appearance does not change forces or motion.



Equations define computation, the project makes it reproducible, and the host executes the built model.

4.9.1 Tools and the Execution Chain: System Modeling Tools

Choose Model Libraries for the Task

- Aerospace Blockset: White-box blocks for six-degree-of-freedom (6-DoF) rigid-body motion, environment, and sensors. Much of the RflySim PSP model library is based on it.
- RflySim libraries package motors, force/torque synthesis, ground-contact models, and platform outputs. Parameter interfaces support identification and refinement from experiment data.
- Simscape: Connected white-box physical components for tasks such as mechanical grasping, payload release, and energy management that need detailed multibody constraints.
- UAV Toolbox: Primarily point-mass, gray-box models for path planning and navigation. Vehicle Dynamics Blockset targets ground vehicles.

Decide which physical processes must be retained, then choose components that express them.

4.9 Modeling Tools and Deployment

4.9.1 Tools and the Execution Chain: Platform Data Flow

Trace the Loop Through Actuator Commands and Measurements

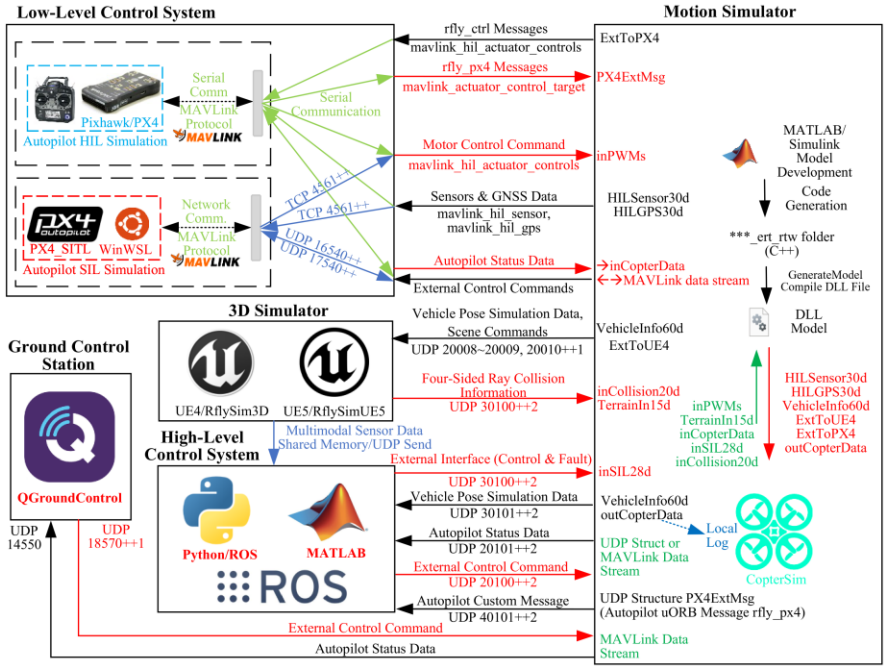


Fig. 4.11 Motion-model exchange with other modules

- **Loop:** PX4 sends actuator values through inPWMs to the DLL in CopterSim; computed measurements return through HILSensor30d and HILGPS30d.
- **Connections:** In software-in-the-loop (SITL; SIL in the textbook), PX4 runs on the PC using Transmission Control Protocol (TCP) ports from 4561. In hardware-in-the-loop (HITL; HIL in the textbook), PX4 runs on the autopilot over serial.
- **External programs:** Python uses User Datagram Protocol (UDP) series 20100 for commands, 20101 for state reception, and 30101 for ground truth.

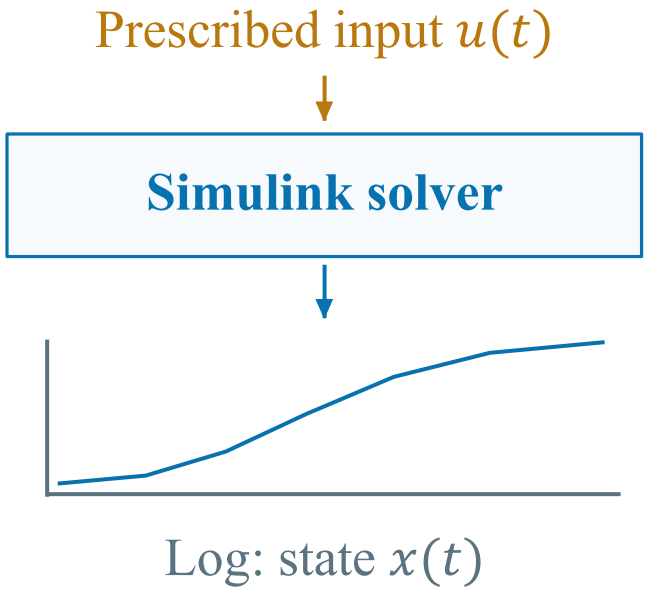
Follow model development → DLL → CopterSim → autopilot.

4.9.1 Tools and the Execution Chain: Offline Solution

Offline Route: Check the Plant First

- The Simulink solver advances offline simulation time. Inputs come from prescribed signals or data replay; initially, check the plant without an autopilot.

- Record motor speed, resultant force and torque, position, and attitude to locate errors in actuators, force synthesis, or rigid-body motion.
- Test settings determine stop time and logging period. Animation is a viewing window; interpretable time series should remain available with the display off.

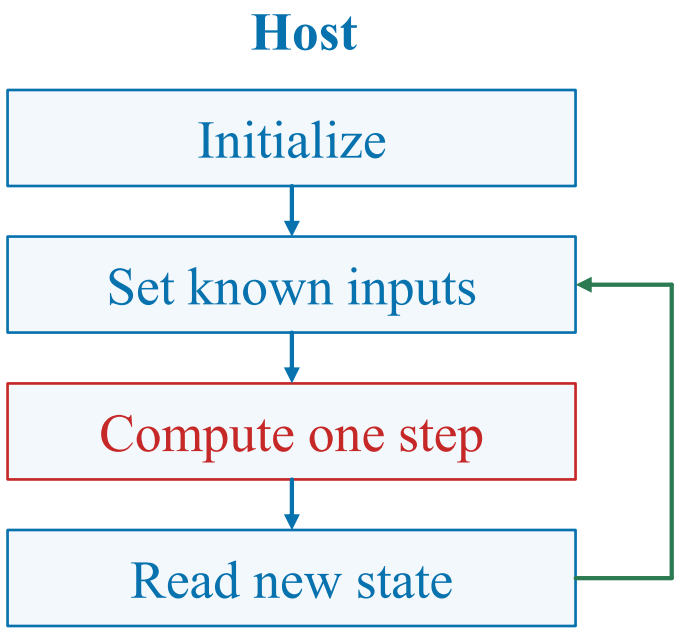


Offline tests apply known inputs directly to the model to trace physical causes.

4.9.1 Tools and the Execution Chain: Host Calls

DLL Route: The Host Makes the Calls

- Code generation converts model structure into a program; compilation produces a DLL. Typical entry points initialize, set inputs, advance one step, and read outputs.
- The host decides when to call these functions and provides data and runtime context. Copying a DLL into a directory alone produces no new simulation state.
- An interface name works only if the current DLL exports it and its internal connections support it. Callable capabilities depend on the compiled model content.



The DLL stores computation functions; the host organizes loading, initialization, input, and stepping.

4.9.1 Tools and the Execution Chain: Closed-Loop Execution

CopterSim Route: Locate Dynamics and the Autopilot

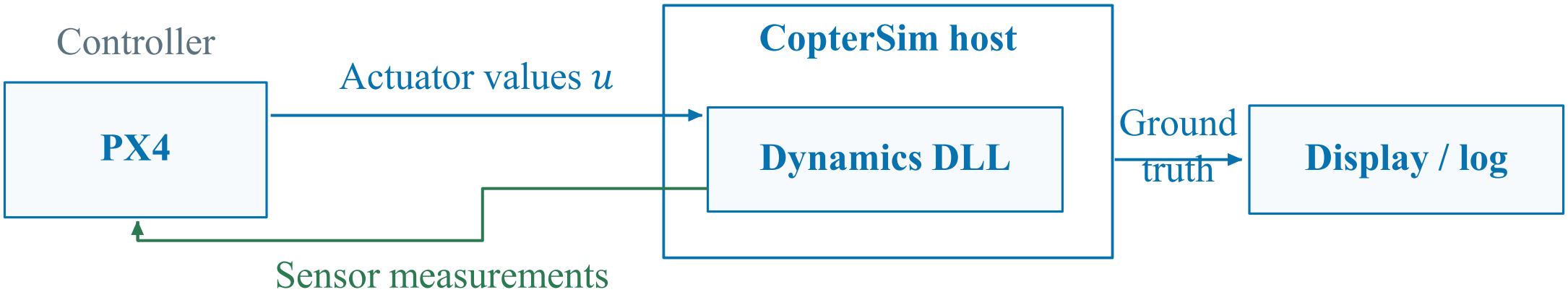


Figure: Measurements return to PX4; ground truth supports observation

- **CopterSim loads the DLL:** actuator values enter; model measurements return to PX4.
- SITL: PX4 on the PC. HITL: PX4 on autopilot hardware. The 3D display uses ground truth.

CopterSim advances dynamics; PX4 computes actuator outputs from measurements.

4.9.1 Tools and the Execution Chain: Two Python Roles

Two Python Routes: Network Interaction and Direct Loading

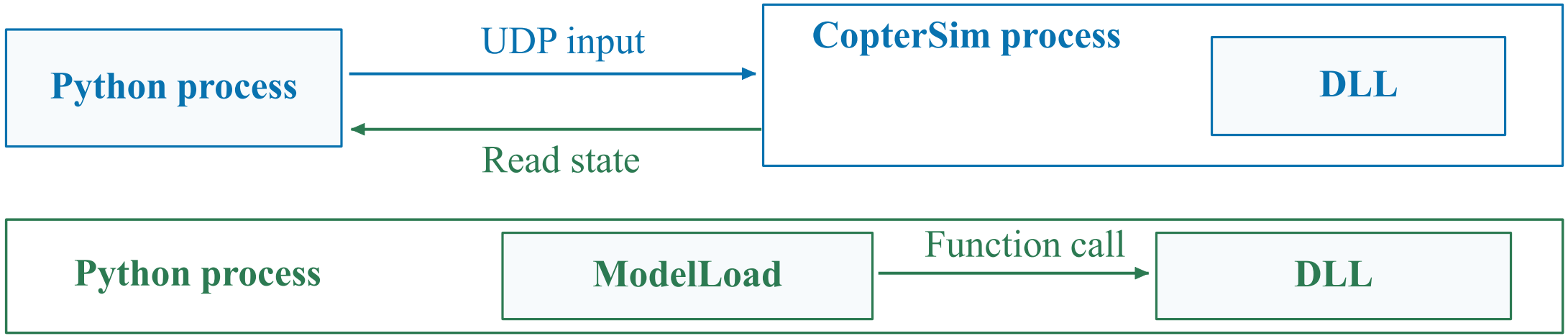


Figure: UDP crosses processes; direct loading calls within one process

- UDP communicates across processes; CopterSim advances the simulation.
- ModelLoad steps an in-process DLL. Integrated DLLs include a controller (Unit 4.5.6).

Network interaction connects to an existing instance; direct loading advances the DLL inside Python.

4.9.1 Tools and the Execution Chain: Generation and Transfer

Detail: Actuators, Measurements, and Code Generation

- Actuator values from the autopilot enter inPWMs; sensor outputs return to PX4 through communication.
- At right, Simulink generates code that compiles into a DLL; CopterSim loads it for computation.
- Read platform ports with field definitions in Section 4.10. Data with the same name can have different indices at different transport layers.

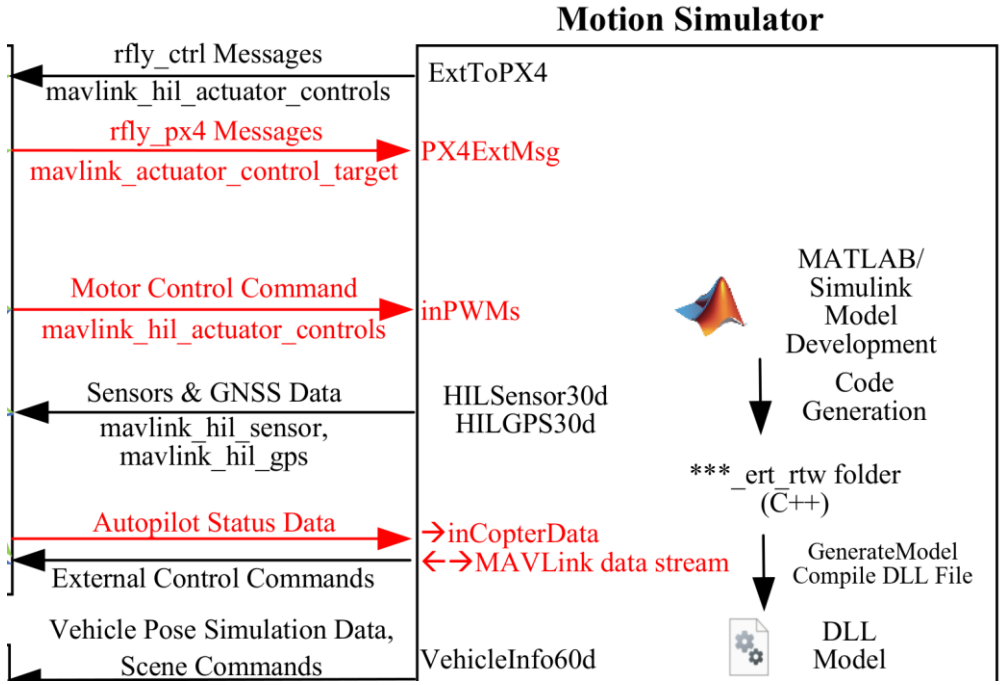


Fig. 4.11 (right): Motion-model I/O and code generation

Actuator input and measurement return have distinct paths; trace the actual interfaces.

4.9.1 Tools and the Execution Chain: In-the-Loop Connections

Detail: Two In-the-Loop Connections for Low-Level Control

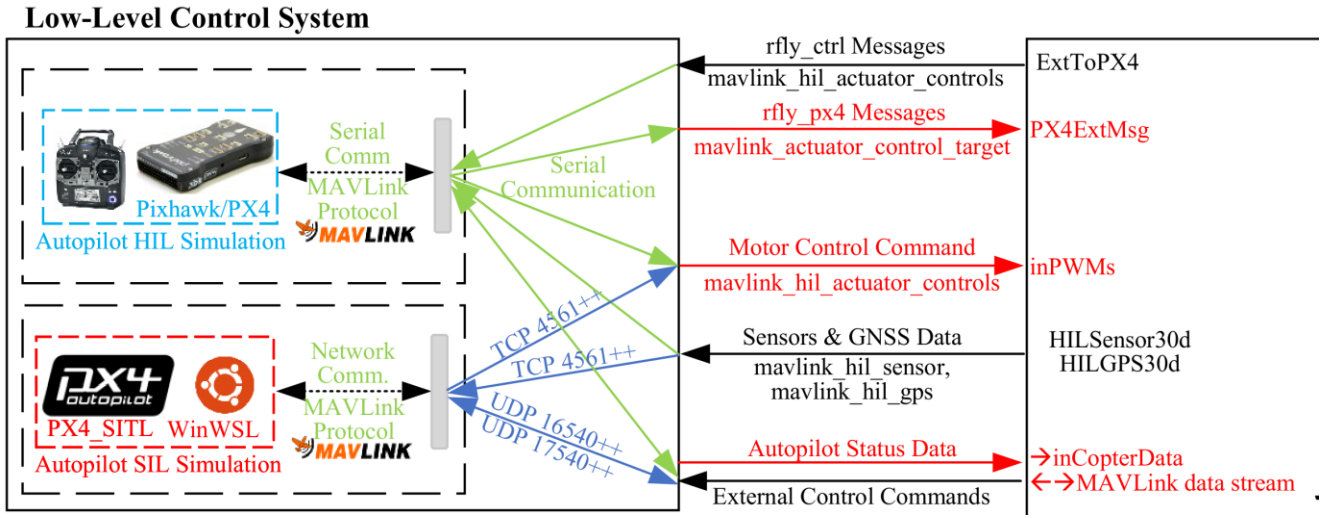


Fig. 4.11 (upper left): Low-level control and motion simulator

- **Read colors:** Green is HITL serial; blue is SITL networking. Red enters the motion simulator; black returns to the autopilot.

- **HITL:** Pixhawk autopilot over serial, using MAVLink.
- **SITL:** PX4 on the PC (WinWSL), using TCP 4561++.
- **Actuators:** mavlink_hil_actuator_controls → inPWMs.
- **Measurements:** HILSensor30d, HILGPS30d → hil_sensor, hil_gps messages.
- **State:** → inCopterData; SITL also uses UDP 16540++, 17540++.

4.9 Modeling Tools and Deployment

4.9.1 Tools and the Execution Chain: External Ports

Detail: UDP Ports for 3D, Ground Station, and High-Level Control

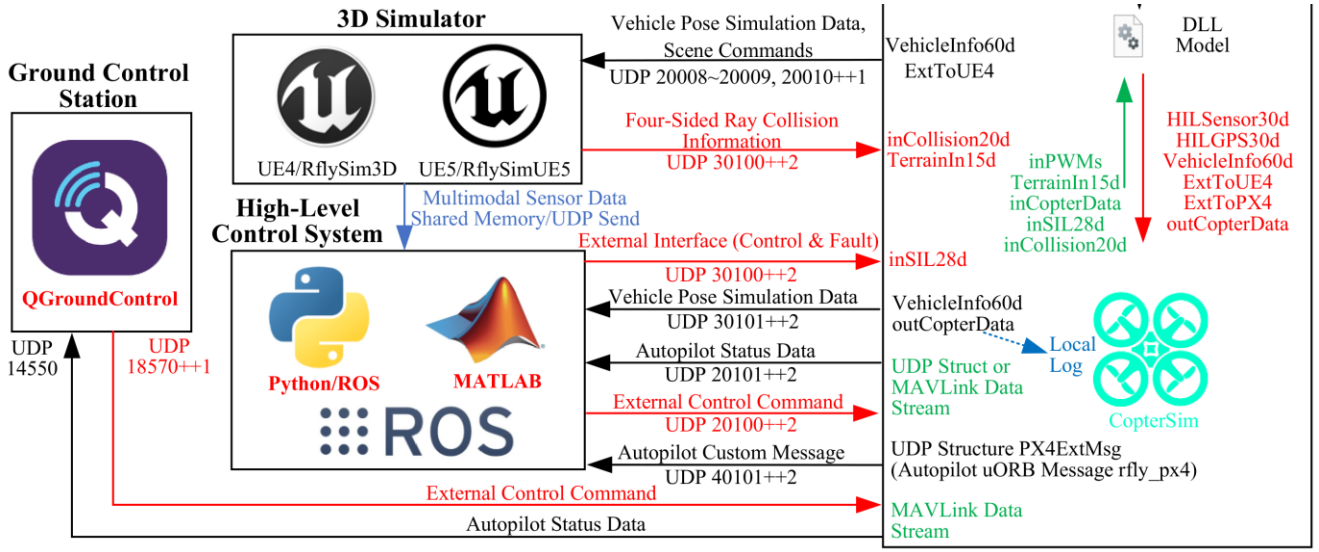


Fig. 4.11 (bottom): 3D simulator, ground station, high-level control

- **Read colors:** Red enters the motion simulator; black leaves it. "++2" means add 2 to the port for each vehicle ID increment.

- **3D simulator:** Receives poses and scene commands (UDP 20008~20009, 20010++1); returns collision and terrain to inCollision20d, TerrainIn15d.
- **High-level control:** inSIL28d uses UDP 30100++2; external commands use 20100++2.
- **Readback:** Ground truth uses UDP 30101++2; autopilot state uses 20101++2.
- **Ground station:** QGroundControl (QGC) receives state over UDP 14550 and sends commands over 18570++1.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.2 MuJoCo Multibody Modeling: Joints and Contact (Optional)

A Physics Engine Builds Joint and Contact Equations from the Model

- **Single rigid body:** Begin multicopter, fixed-wing, or ground-vehicle body motion with one rigid-body equation. Suspended loads, tethers, and trailers also need relative states and constraints.
- **Multiple joints:** Manipulators, suspended loads in Unit 4.8.4, and tilt mechanisms connect rigid bodies through joints. Inertia and gravity couple through joint angles; more joints make hand-derived equations longer.
- **Contact:** Landing, grasping, and walking involve intermittent, friction-limited contact forces. A multibody physics engine builds equations and handles contact from the model description.
- **Model inputs:** Body masses and inertias, joint degrees of freedom, actuator application points, and contact parameters. Check motion, constraint forces, and contact forces against the problem conditions.

Optional environment: Python + MuJoCo, independent of RflySim example entry points.

4.9.2 MuJoCo Multibody Modeling: Dynamics Equations (Optional)

MuJoCo Uses One Dynamics Form for Many Multibody Models

$$M(q)\,\dot{v} + c(q,v) = \tau + J^{\mathrm{T}} f$$

- **Inertia matrix:** $M(q)$ is joint-space inertia, computed for the current configuration by the composite rigid-body algorithm; it is always invertible.
- **Bias force:** $c(q,v)$ is qfrc_bias, combining Coriolis, centrifugal, and other velocity terms with gravity, computed by recursive Newton–Euler.
- **Applied forces and constraints:** τ sums passive, actuator, and external forces. Contact and limit forces f enter joint space through J^{T}; J is the constraint Jacobian.
- **Generalized coordinates:** q is in qpos; v is in qvel. A free joint uses quaternion attitude: 7 qpos entries, 6 qvel entries. A single free joint recovers this chapter's rigid-body equations.

Given the MuJoCo model format (MJCF), the engine computes inertia, bias, and constraint forces at each state.

4.9.2 MuJoCo Multibody Modeling: Role and Data Structures (Optional)

MJCF Defines the Model; MuJoCo Updates Its State Step by Step

- **Role:** MuJoCo means Multi-Joint dynamics with Contact, a general-purpose physics engine maintained by Google DeepMind and open-sourced under Apache-2.0 (as of 2026-09).
- **Format:** MJCF is MuJoCo's Extensible Markup Language (XML) model format. Nested tags specify bodies, joints, shapes, actuators, and sensors; loading compiles them into a model.
- **Data:** MjModel stores fixed structure and parameters after loading. MjData stores time-varying state (qpos, qvel, ctrl) and intermediate results.
- **Loop:** Each mj_step computes forces and acceleration, then integrates one step. Default: semi-implicit Euler, updating velocity before position, with timestep 0.002 s.

Describe the model once; each step computes forces from the current state and advances it.

4.9.2 MuJoCo Multibody Modeling: MJCF Elements (Optional)

Nested MJCF Defines a Kinematic Tree with Actuators and Sensors

worldbody: Fixed scene root; default gravity is 9.81 m/s^2 along $-z$.

body: A rigid body with mass and inertia. Nested bodies form the kinematic tree.

joint: Child-body freedom relative to its parent: hinge rotation, slide translation, ball rotation, or free 6-DoF motion.

geom: Collision and display shape. Without mass, density 1000 kg/m^3 determines mass and inertia.

site: A fixed point on a body, with no collision or mass; often a force or measurement point.

Top-level elements: Outside worldbody; reference a joint or site by name.

actuator: For example, motor multiplies ctrl by transmission gear and acts on a joint or site.

sensor: Reads position, velocity, force, or other simulation quantities into MjData.sensordata.

4.9.2 MuJoCo Multibody Modeling: Pendulum and Integrators (Optional)

A Pendulum Uses body, joint, and geom; Integration Sets Accuracy

- **Model:** One body under worldbody; one hinge rotational DOF; a 1 kg sphere on a massless 1 m rod.
- **Period:** A 60° release gives 2.154 s, matching the exact result. Large amplitude adds about 7% over the small-angle formula.
- At 0.01 s timestep, semi-implicit Euler has 0.073 J energy error; fourth-order Runge–Kutta (RK4) has only about 5×10^{-8} J.

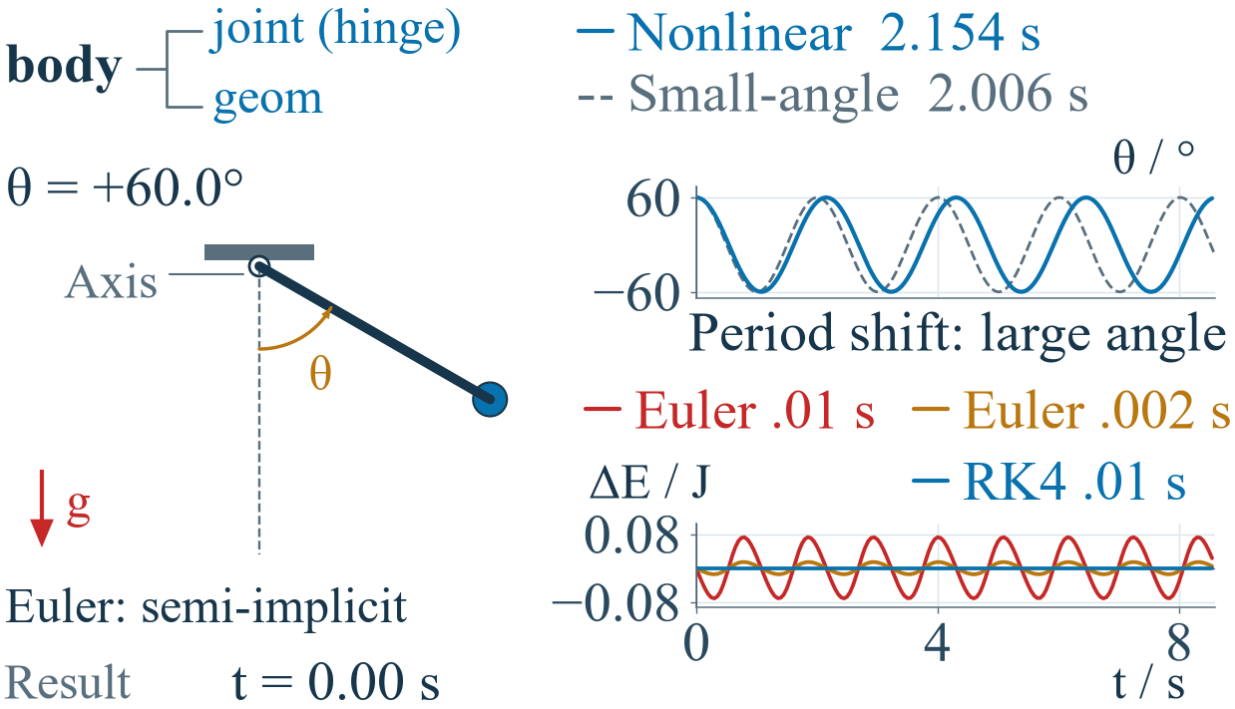


Figure: Pendulum angle and integrator energy error (MuJoCo)

Model structure sets the equations; integrator and timestep set numerical accuracy.

4.9.2 MuJoCo Multibody Modeling: Minimal Python Example (Optional)

Simulate a Pendulum with MjModel, MjData, and mj_step

```
import mujoco
XML = """<mujoco><worldbody><body>
<joint type="hinge" axis="0 -1 0"/>
<geom type="sphere" pos="0 0 -1" size="0.05" mass="1"/>
</body></worldbody></mujoco>"""
model=mujoco.MjModel.from_xml_string(XML) # const
data = mujoco.MjData(model)  # state
data.qpos[0] = 1.0472  # release from rest at 60°
for _ in range(500):  # default 0.002 s step; 1 s total
        mujoco.mj_step(model, data)
print(round(data.time, 2), round(data.qpos[0], 4))
```

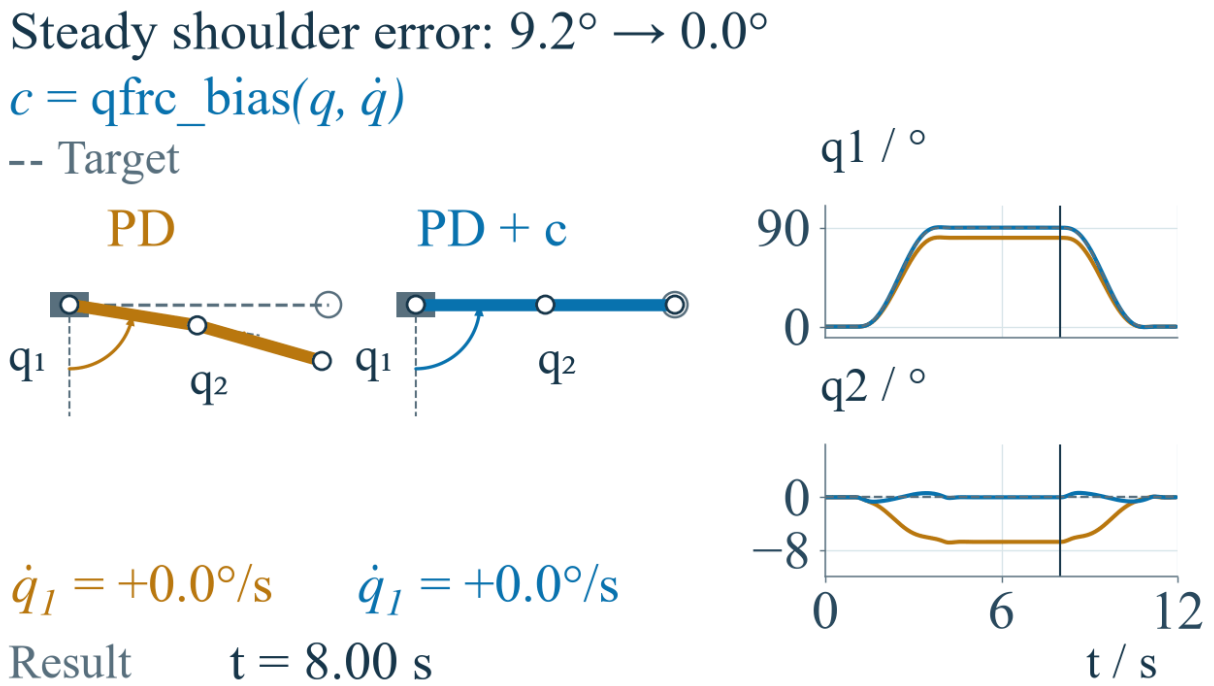
- **Try:** Set option to integrator="RK4", timestep="0.01", and run 100 iterations. The angle output is −1.0221.

- **Load:** from_xml_string compiles MJCF into MjModel; MjData stores states for that model.
- **Initialize and step:** Release from rest at 60°. Default semi-implicit Euler, timestep 0.002 s; 500 mj_step calls advance 1 s.
- **Result:** 1.0 -1.0228. At 1 s, the pendulum is about −58.6° on the other side, near −60° at half a period.

4.9.2 MuJoCo Multibody Modeling: Two-Link PD and Bias Compensation (Optional)

qfrc_bias Compensation Removes the Error Needed to Hold Gravity

- **Comparison:** Identical two-link arms. Left uses proportional–derivative (PD) control only; right also adds qfrc_bias compensation.
- **PD only:** At rest, only $K_p e$ balances gravity torque τ_g, so $e \approx \tau_g / K_p$. Shoulder: $9.8 \div 60 \approx 0.16$ rad$\approx 9.4°$.
- Add current-state qfrc_bias each step. Motors directly supply (9.8, 2.5) N · m gravity torques; steady-state error is 0.



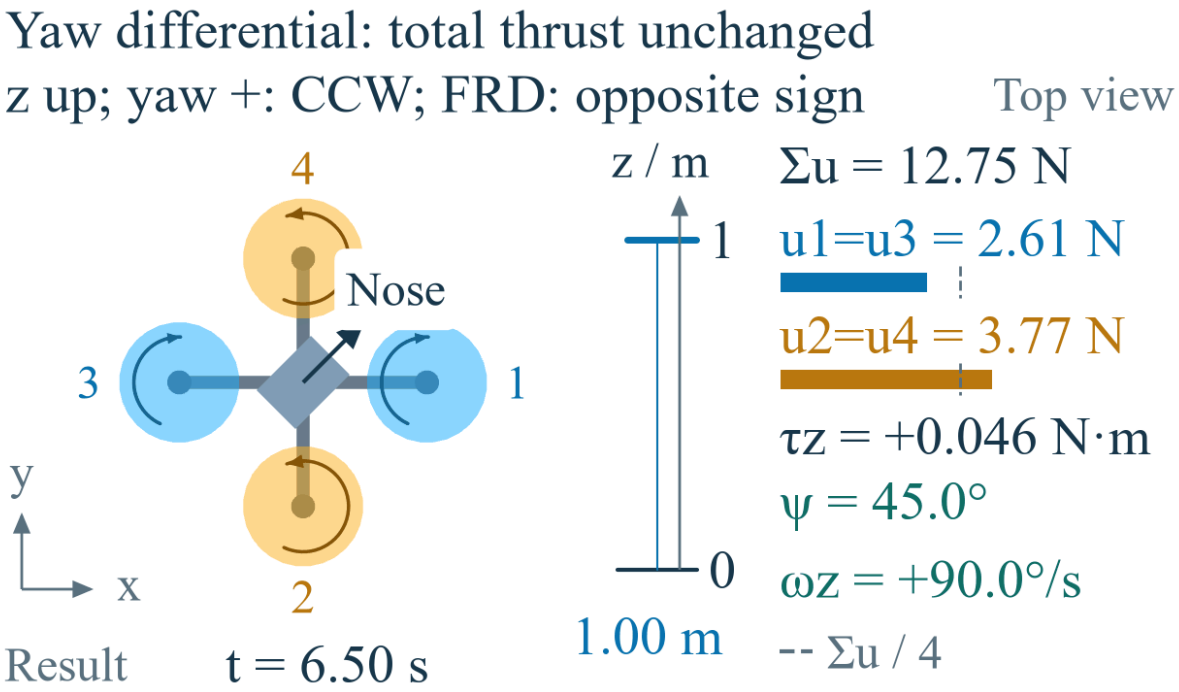
Two-link arms: PD vs. PD + bias compensation (MuJoCo sim.)

Ideal bias compensation supplies gravity torque, so PD need not rely on steady-state error for balance.

4.9.2 MuJoCo Multibody Modeling: Free-Joint Quadcopter (Optional)

site and gear Turn One Control into Thrust and Reaction Torque

- **Body:** One free-joint rigid body: 7 qpos, 6 qvel (6-DoF); z points up, opposite forward-right-down (FRD).
- **Rotors:** Motor at each rotor-center site, gear=(0,0,1,0,0,$\pm k$). Control u_i gives thrust u_i and reaction torque $\pm k u_i$.
- **Thrust:** Hover needs $mg/4 = 3.19$ N per rotor. Opposite diagonal changes $\pm \Delta$ cancel roll and pitch, leaving only $\tau_z = 4k\Delta$.



Free-joint quadcopter: thrust/reaction torque (MuJoCo sim.)

The engine combines moments from rotor positions and directions; the controller must still allocate inputs.

4.9.2 MuJoCo Multibody Modeling: Learning Resources (Optional)

Continue MuJoCo Modeling with Official Docs and Models

- **Overview:** MuJoCo, MJCF, and the roles of MjModel and MjData
  https://mujoco.readthedocs.io/en/stable/overview.html
- **Element reference:** All attributes and defaults for body, joint, geom, site, actuator, and more
  https://mujoco.readthedocs.io/en/stable/XMLreference.html
- **Computation:** Dynamics, constraints, contact solvers, and Euler/RK4 integration
  https://mujoco.readthedocs.io/en/stable/computation/index.html
- **Programming:** Python binding installation (pip install mujoco) and functions such as mj_step
  https://mujoco.readthedocs.io/en/stable/python.html
- **Model library:** MuJoCo Menagerie manipulators, legged robots, Skydio X2 quadcopter, and more
  https://github.com/google-deepmind/mujoco_menagerie

Start with the overview, consult the manual to write models, then modify an existing library model.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.3 Minimal Templates and Reproducible Projects: Vehicle Framework

A General Vehicle Framework: Trace Thrust to Motion States

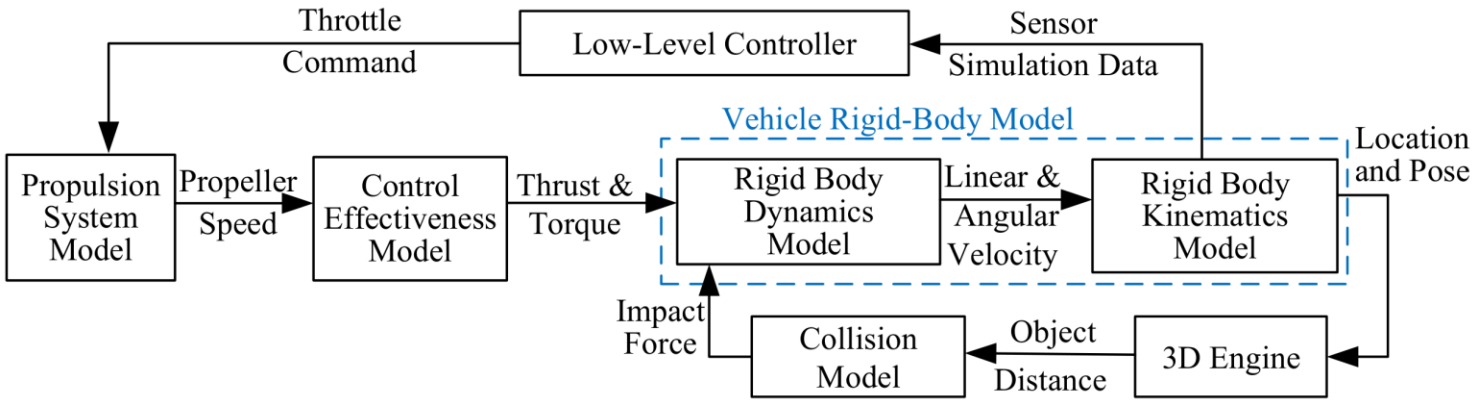


Figure 4.12 General Vehicle Control Model Framework

- **Main path:** Throttle → speed → thrust/torque → velocity/angular rate → position/attitude.
- **Collision branch:** The 3D engine supplies object distances; collision forces return to rigid-body dynamics.
- **Dashed box:** Rigid-body plant.
- **Loop:** Simulated sensor data → low-level controller.

Module boundaries must have interpretable inputs, outputs, and states.

4.9.3 Minimal Templates and Reproducible Projects: Project Files

M0 Files: Model, Initialization, Packaging, Launch, and Outputs

- Directory: [installation directory]/RflySimAPIs/4.RflySimModel/1.BasicExps/e0_MinModelTemp/. Deployment baseline M0 is Exp1_MinModelTemp.slx; read Readme.pdf first.

```
e0_MinModelTemp/
  Readme.pdf
  Exp1_MinModelTemp.slx
  Exp1_MinModelTemp_init.m
  GenerateModelDLLFile.p
  Exp1_MinModelTemp_SITL.bat / _HITL.bat
  Exp1_MinModelTemp.dll
  MulticopterModel.zip
```

- **Model and initialization:** SLX defines equations and root ports; _init.m stores parameters and initial state.
- **Packaging and launch:** The .p file packages the DLL; two batch files launch SITL and HITL.
- **Released outputs:** Load the DLL directly; the zip packages generated code.

Reproducible deployment requires matching versions of the model, initialization, and runtime artifact.

4.9.3 Minimal Templates and Reproducible Projects: Root Ports

M0 Has Two Inputs and Three Outputs

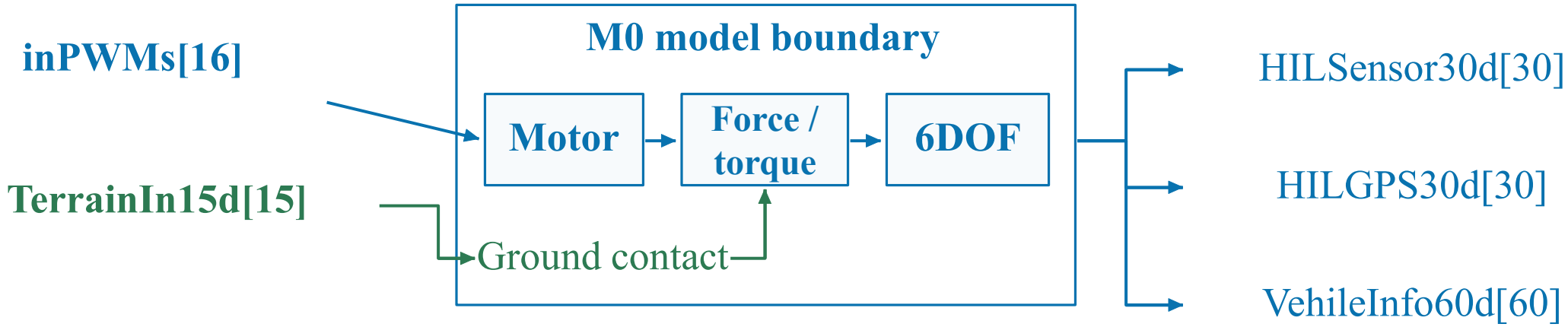


Figure: Actuator and terrain inputs produce state and observation outputs; VehileInfo60d is the source spelling

- 16 actuator inputs and 15 terrain fields; the first terrain field is ground height.
- The 6DOF bus contains vectors and matrices. Select signals before reading the three output groups.

Root ports define the agreed data boundary between model and host.

4.9.3 Minimal Templates and Reproducible Projects: Actual Connections

Trace One Signal Through to the State

- Follow inPWMs into Motor Model, inspect MotorRads angular speed, then enter Force and Moment Model for resultant force and torque.
- 6DOF1 advances motion states and sends position, velocity, attitude, and more to SensorOutput and 3DOutput1. Ground-contact forces feed back through the contact model.
- The 6DOF output is a bus of vectors and matrices, not a fixed 13-element array. Expand its bus selector before naming logged signals and units.

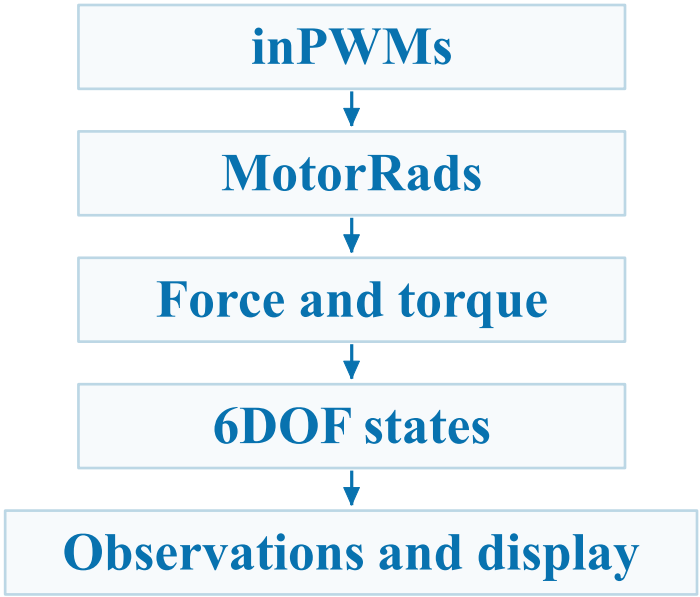


Figure: M0 actuator-input computation path

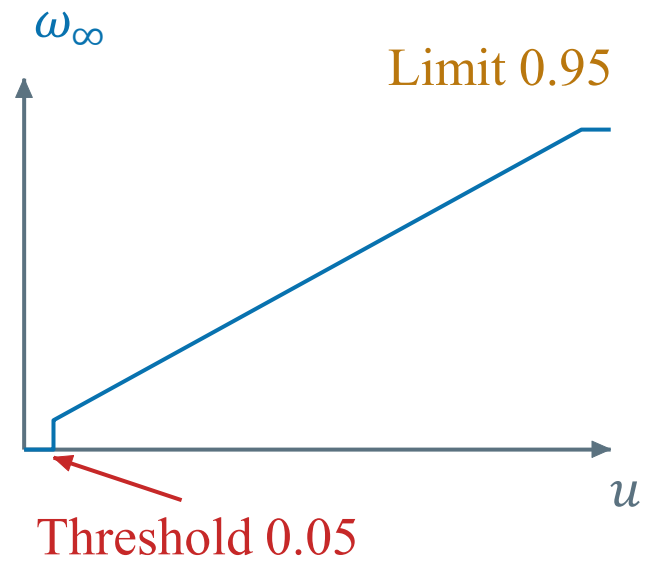
Follow actual connections for causality, then record results through output ports.

4.9.3 Minimal Templates and Reproducible Projects: Input Scale

Actuator Inputs Are Not Pulse Widths in Microseconds

- M0's first four channels are normalized motor inputs; servos commonly use −1 to 1. Entering 1500 clips to 0.95, near full throttle.
- Clip input to 0–0.95. At or above 0.05, set target speed linearly; below it, use zero. Actual speed always follows first-order dynamics.
- The 16-element port reserves expansion. Current M0 has four motors, with positions and rotation directions set by the layout.

$$u_s = \min(0.95, \max(0, u))$$



Schematic: linear map above threshold

A port named for pulse-width modulation (PWM) still requires its actual normalized data scale.

4.9.3 Minimal Templates and Reproducible Projects: Parameters and Initial State

Parameters Define the Model; Initial Values Define the Start

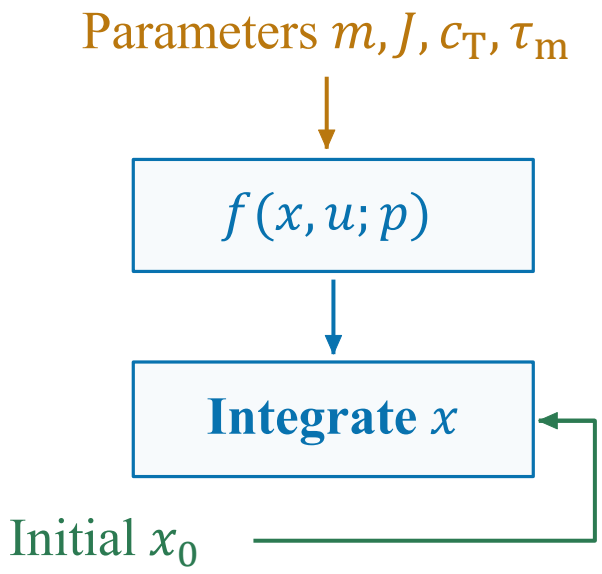


Figure: Model properties and experiment starting point

- M0 mass is 1.515 kg. Inertia diagonal entries are 0.0211, 0.0219, 0.0366 kg·m². They set acceleration under the same force and torque.
- Position, velocity, attitude, angular rate, and motor speed are initial states. Equal inputs from rest or an existing velocity give different position curves.
- The M0 script starts position, velocity, and angular rate at zero. For free-motion tests, declare a new height and contact treatment in the teaching copy.

Changing physical parameters and changing the starting state affect responses in different ways.

4.9.3 Minimal Templates and Reproducible Projects: Two Templates

D0 Explains Equations; M0 Starts Deployment

- **Location:** Both templates are under [installation directory]/RflySimAPIs/4.RflySimModel/, with different purposes.

M0: Deployment starting point

- **Directory:** 1.BasicExps/e0_MinModelTemp
- **Files:** Exp1_MinModelTemp.slx + _init.m
- **Generation:** Embedded C++; retains platform root ports

D0: Equation reference

- **Directory:** 2.AdvExps/e12_ModelTempId/3.ModelingTemp/code
- **Files:** dynamics.slx + Init.m; run Init.m first
- **Generation:** Default C, inlined parameters

- D0 is not an M0 copy with matching parameters: mass, motor map, and initial height differ. Compare static thrust using the mapping equations.

Identify each template's source and purpose before interpreting its default-parameter effects.

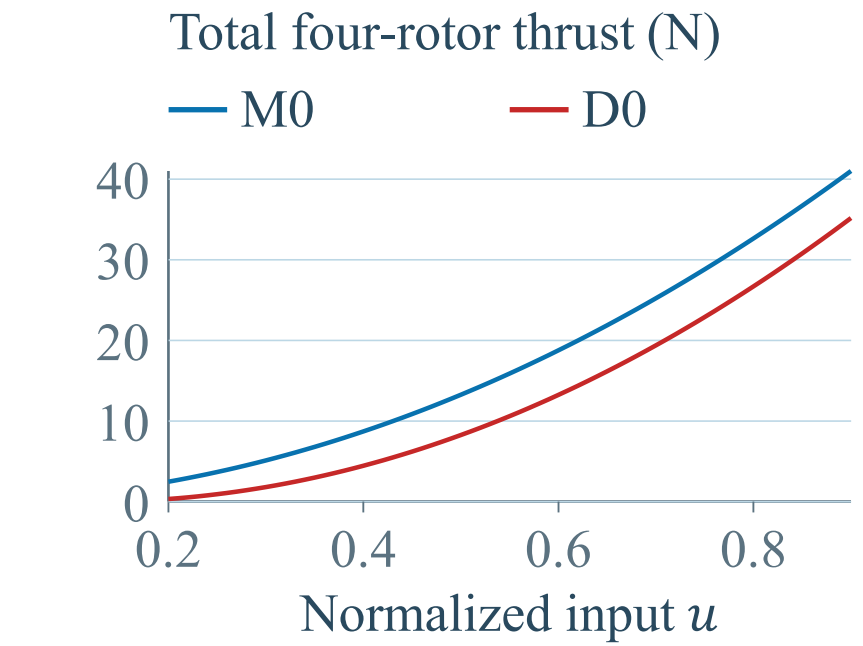
4.9.3 Minimal Templates and Reproducible Projects: Matched Conditions

Why M0 and D0 Curves Cannot Be Compared Directly

- M0: Mass 1.515 kg, map $\omega = 842.1u + 22.83$.
  D0: Mass 1.4 kg, map $\omega = 1148u - 141.4$.

- Speed ω and intercepts are in rad/s; input u is dimensionless. Thrust coefficients are 1.681×10^{-5} and 1.105×10^{-5} respectively, in N·s²/rad².

- Initial z_D is 0 and −100 m respectively. Equal input also gives different total thrust. Align parameters, initial states, and model boundaries before comparing.



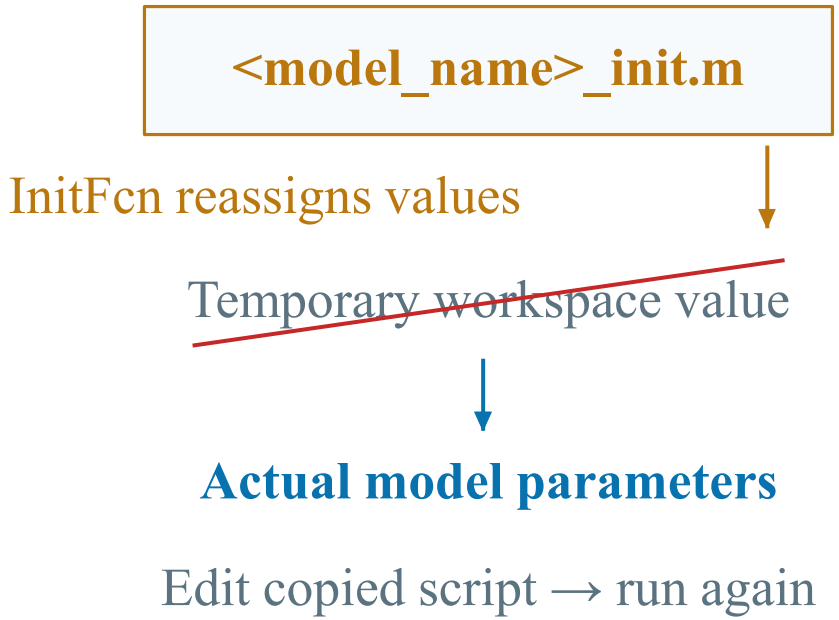
Static thrust differs at equal input

Different parameters and initial states prevent attributing trajectory differences directly to implementation.

4.9.3 Minimal Templates and Reproducible Projects: Initialization Callback

When Does the Initialization Script Run?

- M0's initialization callback (InitFcn) runs the matching _init.m, reassigning parameters and initial states and overwriting temporary workspace edits.
- Edit _init.m in a teaching copy and check its callback. A test wrapper can instead disable the copied callback and inject fixed parameters.
- For D0, run Init.m in code first. It contains clear: save data, then check actual mass, initial state, and time constant.



Model name, working directory, and variable workspace jointly affect initialization.

4.9.3 Minimal Templates and Reproducible Projects: Output Boundaries

Distinguish Ground Truth, Measurements, and Estimates

Ground truth: $a_D = 0$

↓

Level and stationary: $f_{b,z} = -g$

Sampling, noise, and encoding

↓

PX4 fuses data into estimates

These three data types are not interchangeable

- Ground truth is the computed motion state used to check force–motion relations. VehileInfo60d provides ground truth and display data.
- Measurements come from states through sensor models, with noise, sampling, and encoding. Accelerometer specific force differs from Earth-frame acceleration.
- Estimates are states obtained by PX4 measurement fusion, affected by filtering and initialization. State which data layer is being read before comparing.

Different data outputs from one aircraft answer different questions.

4.9.3 Minimal Templates and Reproducible Projects: Fixed Test Conditions

Define the Experiment Before Pressing Run

- Copy the project and fix parameters, initial states, north-east-down (NED)/FRD frames, and environment switches. Finite stop time, input sequence, and logs define the experiment.
- Current M0 uses ode4, 1 ms, 10 s. D0 defaults to infinite duration; run Init.m, replace zero inputs, and set a finite stop time first.
- Offline checks: m0_offline_checks under 4.RflySimModel/1.BasicExps/e6_ModelingLearningLab/05_M0Checks/ (read Readme.pdf first). Full workflow: Unit 4.11.2.

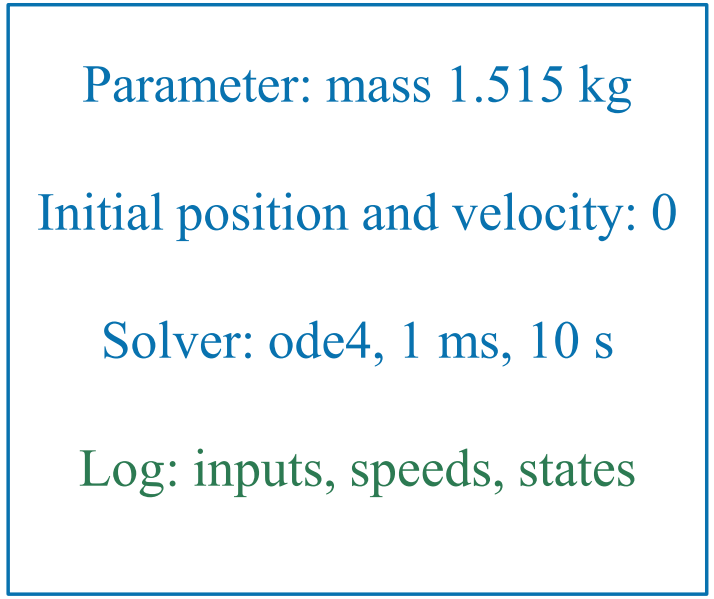


Figure: Example M0 experiment card

Fix inputs, initial values, time, and logged quantities so the result can be assessed.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.4 Numerical Time and Scheduling: Solver Settings

The Timestep Sets How Often the State Advances

- Timestep h is the interval between integration times. Fixed steps suit periodic generated-code calls; oversized steps may miss rapid changes.
- On the Solver page, check Fixed-step, ode4, and 0.001 s. M0 uses fourth-order Runge–Kutta.
- Stop time is also a test condition. Check timestep accuracy for the actual motor and contact conditions.

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, t), \qquad t_{k+1} = t_k + h$$

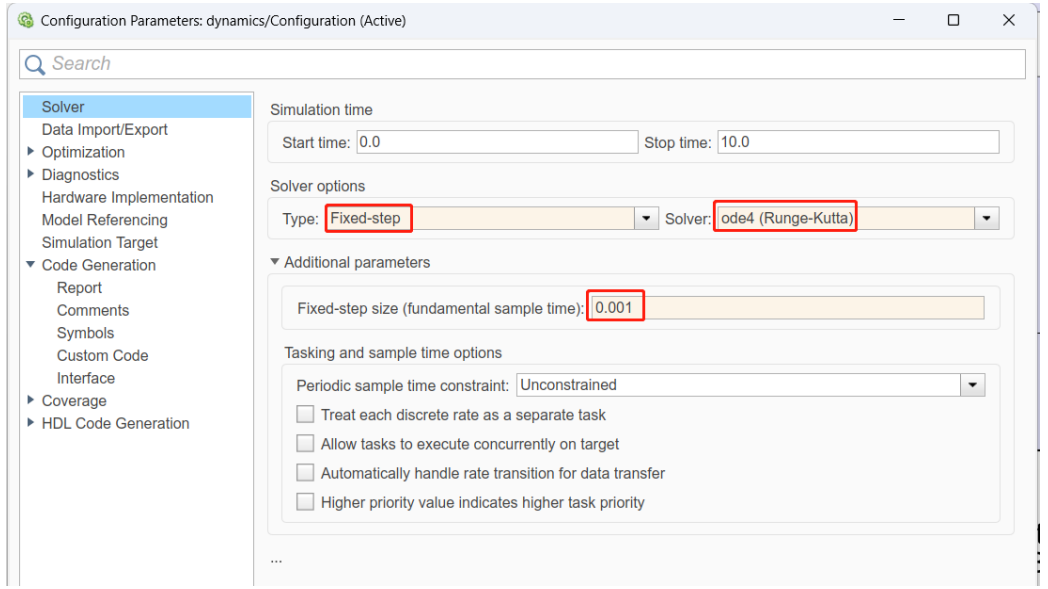


Figure: Solver settings (reference screenshot, example dynamics; M0 uses the same settings)

Finite timesteps turn continuous equations into a numerical trajectory.

4.9.4 Numerical Time and Scheduling: Multirate Tasks

One Simulation Can Have Several Periods

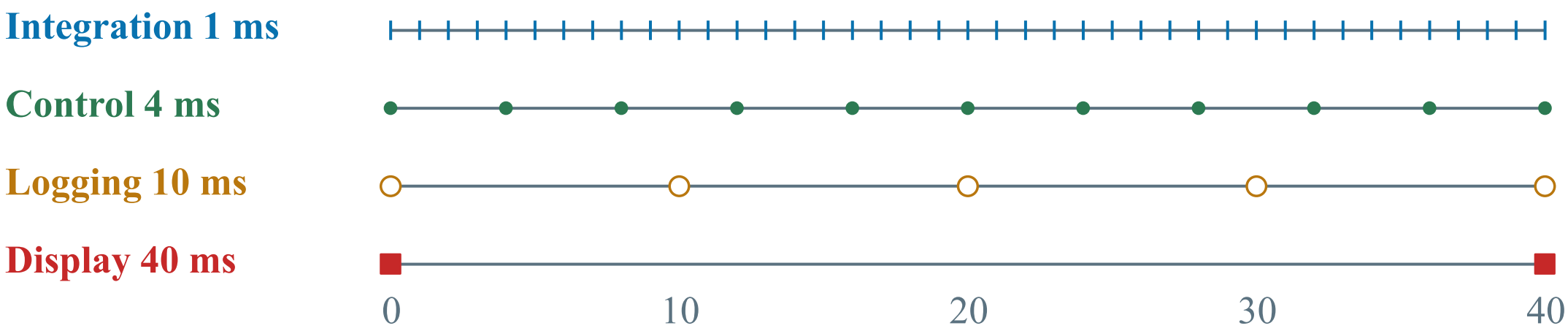


Figure: Multirate timing schematic (horizontal axis in ms)

- Hold commands between control updates; logging and display have their own schedules.
- One 40 ms display interval contains 40 integration steps. Reading a cache does not advance physics.

Integration, held inputs, logging, and display refresh perform different jobs.

4.9.4 Numerical Time and Scheduling: Different State Update Rates

Motion Advances While the Display Cache Updates at Its Own Rate

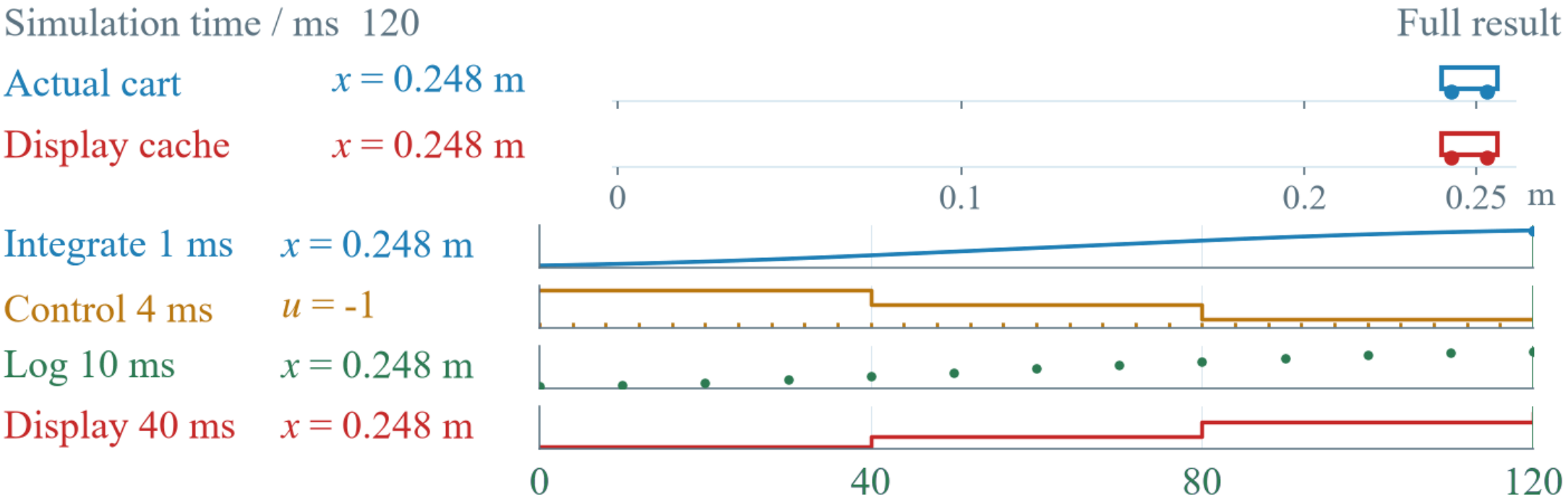


Figure: Four schedules for the same motion (position in m; time in ms)

Integration, held inputs, logging, and display refresh perform different jobs.

4.9.4 Numerical Time and Scheduling: Simulation and Wall Time

Record Simulation Time and Wall Time Separately

- Simulation time belongs to the state equations; wall time is elapsed real time. Offline computation can run faster or slower than the physical process.
- Real-time factor is simulation-time increment divided by wall-time increment. Compare trajectories by simulation time and performance by wall time.
- Direct DLL calls advance integer base steps. For positive requested duration, truncate duration/step to an integer. Actual advance is Nh; the substep remainder is not executed.

$$\rho = \frac{\Delta t_{\text{sim}}}{\Delta t_{\text{wall}}}$$

$$N = \text{int}(\Delta t / h)$$

Seconds of model evolution differ from seconds elapsed on the computer.

4.9.4 Numerical Time and Scheduling: Fast Dynamics and Contact

Why Contact and Fast Dynamics Need Timestep Checks

- Ground contact can produce large spring and damping forces over short times. Oversized steps can skip contact, causing penetration, rebound, or numerical oscillation.
- A smaller motor time constant puts more response within one step. Recheck with half the step: the short M0 baseline in Unit 4.11.2 differs by about 10^{-8} m between 1 ms and 0.5 ms.
- For long attitude integration, also check quaternion norm or rotation-matrix orthogonality. A smooth position curve does not establish every internal state constraint.

$$\|\mathbf{q}\| \approx 1, \qquad R^T R \approx I$$

Faster changes require stronger evidence that finite steps resolve the important process.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.5 Generating C++ from SLX: Minimal Traceable Example

Begin Code Generation with One Integrator

- **Entry:** 4.RflySimModel/0.ApiExps/ 2.UserDefinedC++/2.GenC++/demo.slx; read Readme.pdf first.
- **Mapping:** Uin passes through Integrator to become Yout; X0 supplies the initial state. The model has one dynamic relation.
- **Prediction and criterion:** In a teaching copy, input 1, X0=0, and run 1 s. Yout should rise linearly to 1; its error from 1 at 1 s should be at numerical-error scale.

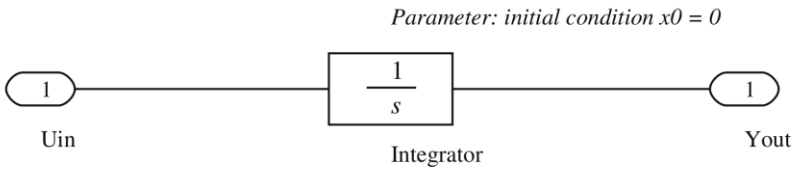


Figure: GenC++ integrator example

Use an integrator with a known solution to map input, state, and output to generated code.

4.9.5 Generating C++ from SLX: Port and State Configuration

Fix Types, Dimensions, and Initial Values Before Generating Interfaces

- **Ports:** Set Uin to double, dimension 1, and do the same for Yout. Names, types, dimensions, units, and call timing together define interface conventions.
- **Initial state:** Set the integrator initial value to X0 and model InitFcn to X0=0. Rerunning invokes the callback and may overwrite temporary workspace edits.
- **Check:** Save a teaching copy and set a finite stop time. Record port names, double type, scalar dimensions, and initial value together.

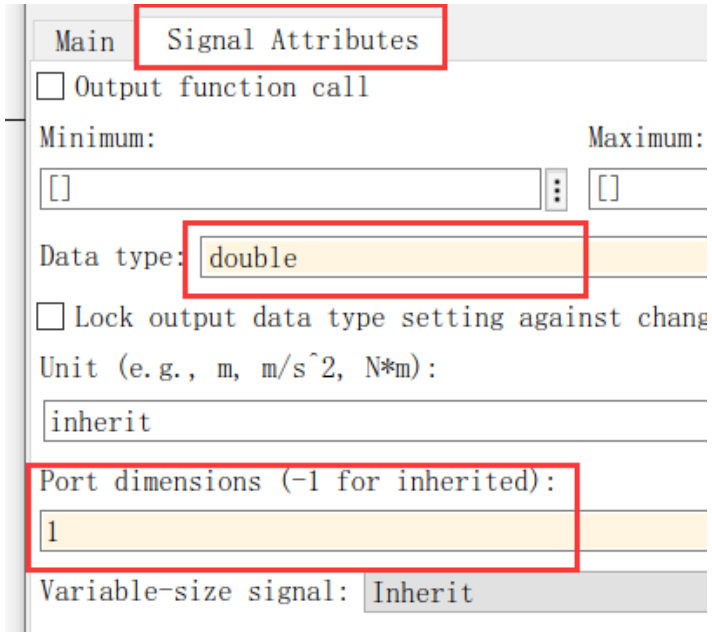


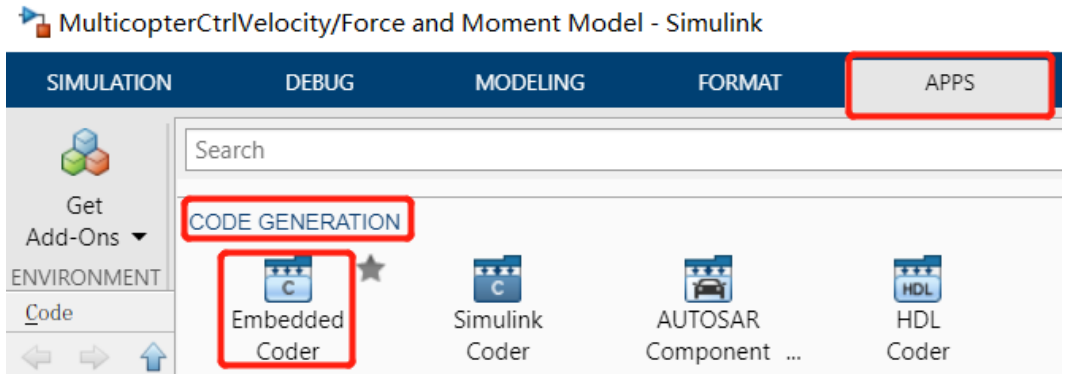
Figure: Scalar port configuration (reference detail)

Types, dimensions, and initialization tell the caller how to provide input and start a calculation.

4.9.5 Generating C++ from SLX: Prerequisites and Build Entry

Check Generation Prerequisites, Then Build in Embedded Coder

- **Dependencies:** Update the model and resolve missing parameters or libraries first. M0 uses ert.tlc, the Embedded Coder embedded real-time target, to generate C++.
- **Compiler:** Confirm with mex -setup C++ in MATLAB. Use the tool environment configured in Chapter 2.
- **Change of example:** demo explains U, Y, X0, and integrator state. The following settings target an M0 teaching copy; screenshots come from reference examples.



Embedded Coder entry (reference screenshot)

- **Build:** APPS → CODE GENERATION → Embedded Coder generates program files for the current M0 teaching copy.
- **Run:** Executes the current model to obtain a trajectory. Menu locations vary by version.

4.9.5 Generating C++ from SLX: Target and Language

Check Target, Language, and Packaging Together

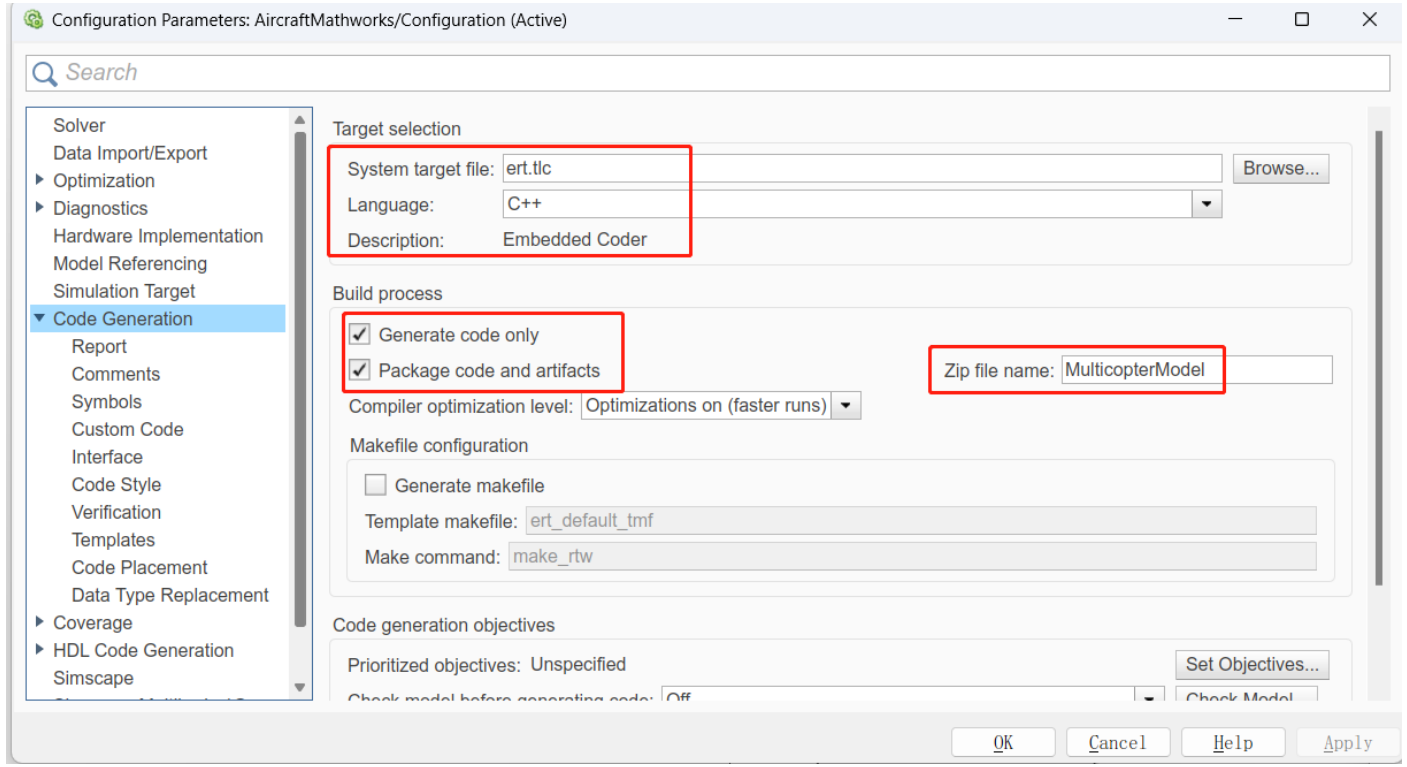


Figure: Target language and packaging—application programming interface (API) reference

- **Target and language:** The reference uses ert.tlc and C++; M0's saved settings match.
- **Generation and packaging:** Select "Generate code only" and "Package code and artifacts." Package name: MulticopterModel.
- **Match entry points:** Later steps use this name. Changing only the filename or language can break that match.

4.9.5 Generating C++ from SLX: Continuous States and Parameter Access

Continuous-State Support and Accessible Parameters

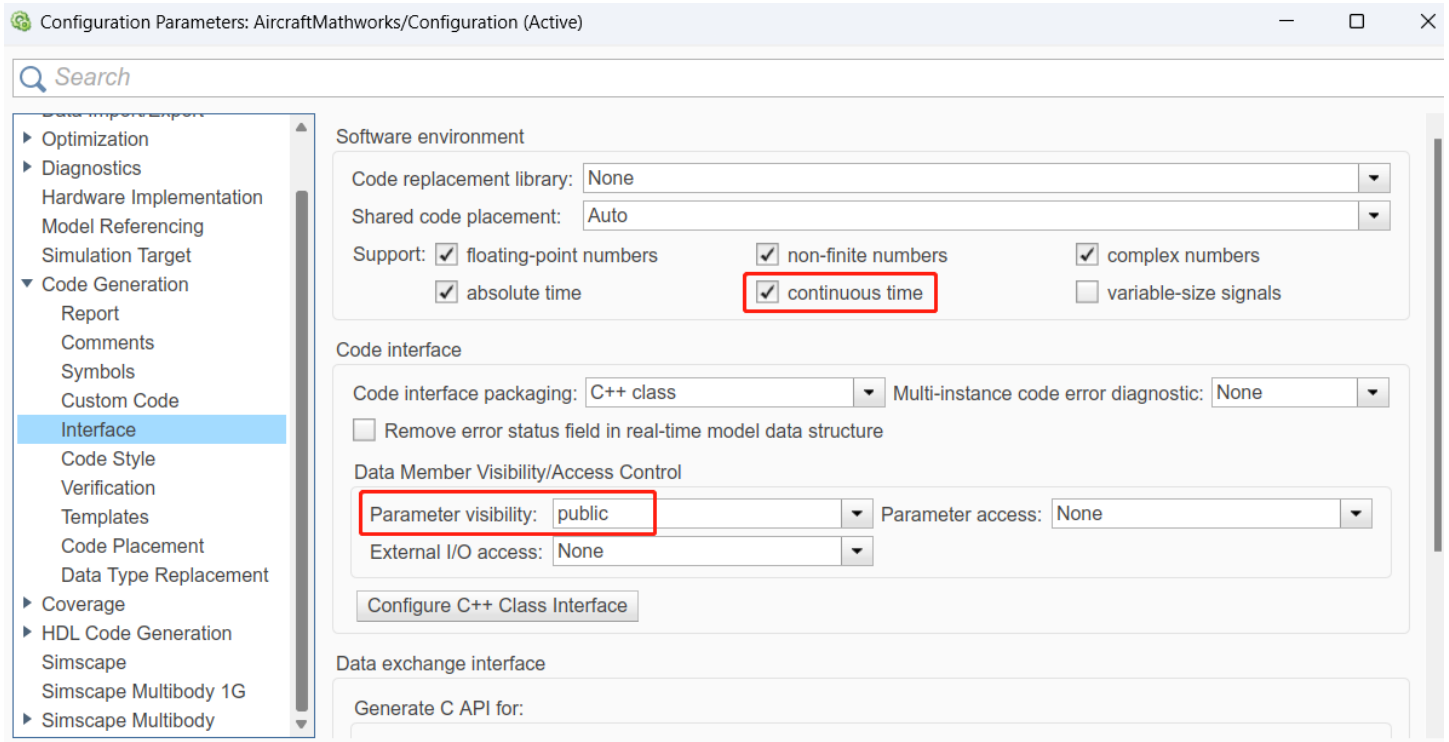


Figure: Continuous states and parameter access (API-reference screenshot)

- **continuous time:** Models with continuous integration states require continuous-time support; otherwise the model and generation target conflict.
- **public parameters:** Code can read/write generated class parameters. Changing mass from Python also requires the DLL to export the corresponding setter.

4.9.5 Generating C++ from SLX: Generated Class Boundary

Code: Identify U, Y, and P in the Generated Class

- **Files:** Extract e0_MinModelTemp/MulticopterModel.zip and open Exp1_MinModelTemp_ert_rtw. Headers declare interfaces; CPP implements them; ert_main.cpp demonstrates calls.

```
class MulticopterModelClass {
public:
  ExtU_Exp1_MinModelTemp_T Exp1_MinModelTemp_U;
  ExtY_Exp1_MinModelTemp_T Exp1_MinModelTemp_Y;
  static P_Exp1_MinModelTemp_T Exp1_MinModelTemp_P;
  void initialize();
  void step();
  static void terminate();
};
```

- **U:** Instance root inputs.
- **Y:** Instance root outputs; internal states are not listed.

- **P is static:** Multiple instances do not automatically get independent parameter copies. Check generation settings and wrappers for isolation.

4.9.5 Generating C++ from SLX: Model Lifecycle

Three Functions Start, Advance, and End the Model

- **initialize:** Set model time, continuous states, and initial conditions. Initialize once at the start of each independent test; do not reset every step.
- **step:** Compute outputs and states for one base step. M0 includes ode4 intermediate stages; one step is more than one formula evaluation.
- **terminate:** End the instance's run. The host must also manage its own threads, logs, and external resources.

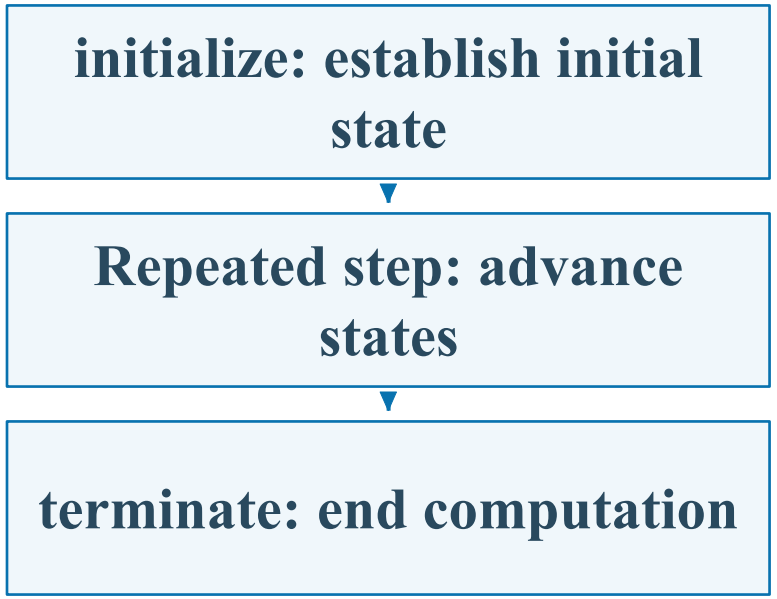


Figure: Function-call order for one model experiment

4.9.5 Generating C++ from SLX: Call Order and Logging

Code: Initialize Once, Then Advance a Finite Number of Steps

```
MulticopterModelClass plant;
plant.initialize();
for (int k = 0; k < 1000; ++k) {
  // Fill all U inputs here according to interface conventions
  plant.step();
  const auto& y = plant.Exp1_MinModelTemp_Y;
  // Save y with its model time here
}
plant.terminate();
```

- **Start:** initialize is outside the loop, called once per independent test.
- **Each iteration:** Fill all U → step → read Y. Timestamp with exposed model time; check first and last sample times.
- **Stop:** 1000 × 1 ms = 1 s of model time, then terminate, independent of wall-time waiting. See ert_main.cpp and the template for building, terrain initialization, and logging.

4.9.5 Generating C++ from SLX: Reusing Source Code

Connect Existing C Functions to the Model Computation Chain

- **Example:** Max.c and Square.c in 4.RflySimModel/0.ApiExps/2.UserDefinedC++/1.CallC/code/ provide maximum and square functions, called from mfunc.slx through MATLAB Function.
- **Generation route:** coder.cinclude declares the header; coder.ceval calls C functions. The build must locate source and header files; data types must match uint16_t parameters.
- **Distinction:** An external call running only in the MATLAB host does not automatically become deployable model code. First test inputs 2 and 3: maximum 3, square 9.

```
coder.cinclude('Typedef.h');
y = coder.ceval('GetMaxValue', u1, u2);
y = coder.ceval('SquareOperation', y);
```

Align function signatures, data types, and build dependencies before including external algorithms in generated code.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

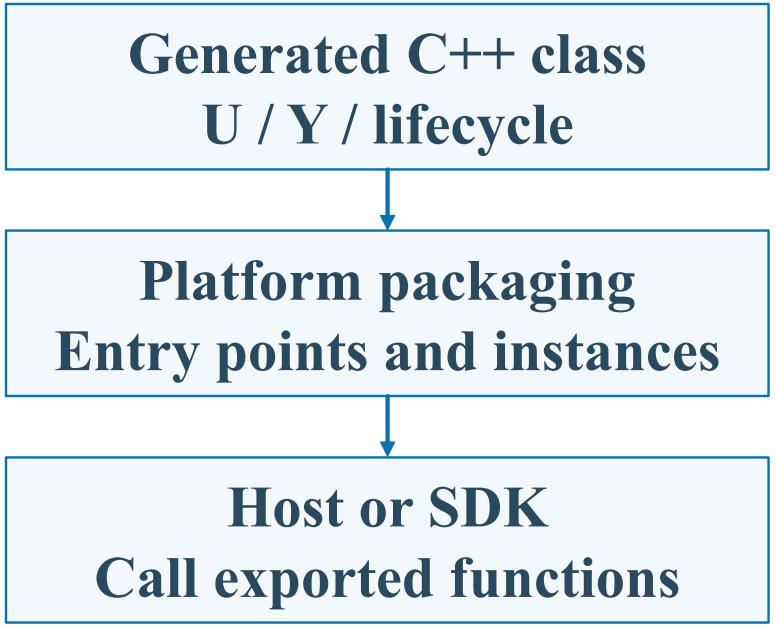
4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.6 Packaging, Deployment, and Loading: Callable Artifacts

A Platform Wrapper Connects the Generated Class to a Loadable DLL

- **Generated class:** Members and methods are directly usable only after external code knows the class definition and links the corresponding implementation.
- **Platform wrapper:** A loadable DLL must expose entry points, transfer arrays, and manage instances under platform conventions. CopterSim DLLs must follow them.
- **Existing route:** M0 uses GenerateModelDLLFile for platform packaging. Direct-loading examples use the software development kit (SDK) to call DLL exports; inspect the actual artifact for wrapper content.



Generated code → platform DLL

Generated code computes; the platform wrapper provides interface conventions recognized by the host.

4.9.6 Packaging, Deployment, and Loading: Artifacts and Build Evidence

Visible Results from Generated Files to the DLL

- After Build, check this model's _ert_rtw directory and package. Run GenerateModelDLLFile in the project directory.
- Check the new DLL's name, timestamp, and hash; save the build log. Confirm success from the newly generated file.
- The figure shows a release-documentation success message. Use current logs and new files for this machine's result.

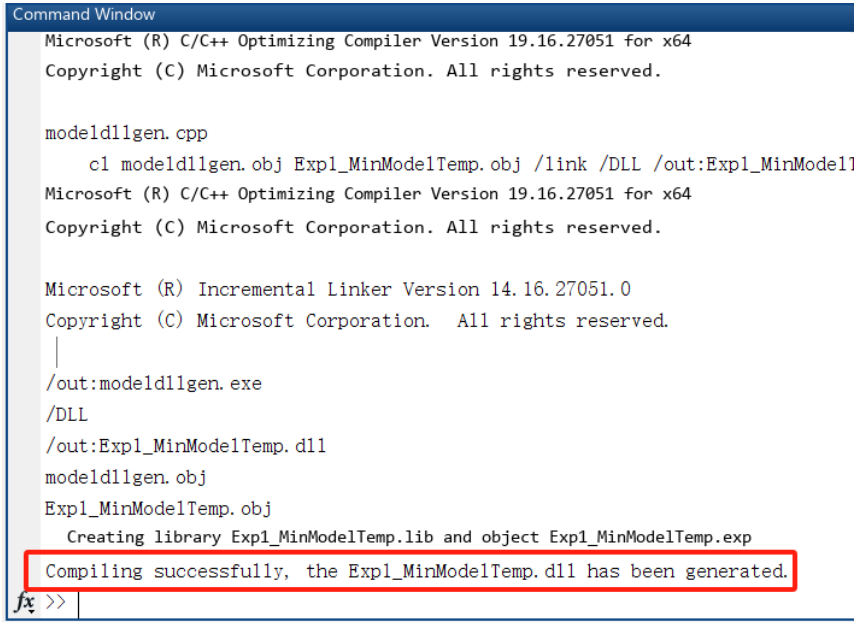


Figure: Reference example of a successful DLL-generation message

Code generation and DLL generation are successive but distinct steps.

4.9.6 Packaging, Deployment, and Loading: Deployment Files

DLLModel Selects the Dynamics File to Load

```
REM Set use DLL model name or not, use number index or name string
REM This option is useful for simulation with other types of vehicles instead of multicopters
set DLLModel=Exp1_MinModelTemp
```

Figure: DLLModel load name (example reference screenshot)

- The script sets DLLModel to Exp1_MinModelTemp, copies its DLL to CopterSim/external/model, and loads that name.
- Record source and destination DLL hashes to exclude old files. Changing airframe or 3D appearance does not replace the dynamics DLL.

A change enters the execution chain only when launch settings match the deployed file.

4.9.6 Packaging, Deployment, and Loading: Launch Configuration

Configure Airframe, Simulation Mode, and Appearance Separately

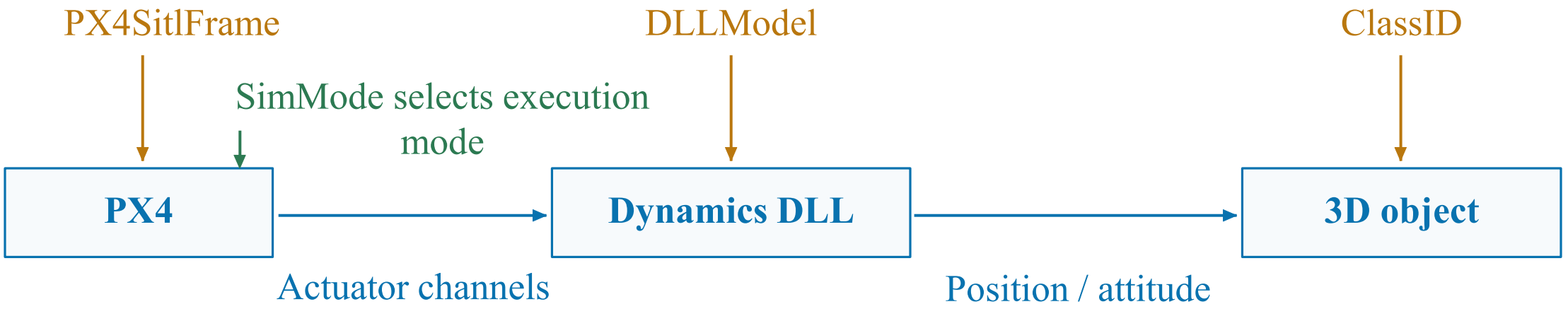


Figure: Configuration fields address different runtime objects and do not substitute for one another

- SimMode=2 selects SITL. The iris quadcopter airframe references a mixer and a proportional–integral–derivative (PID) parameter set.
- DLL layout sets forces; ClassID sets display. Retune PID in QGC after changing structure or mass.

Control channels, execution mode, and visual shape must agree but have different meanings.

4.9.6 Packaging, Deployment, and Loading: GUI and Simulation Scene

Configure Multicopters in CopterSim; Match the Scene to the Vehicle

- **GUI:** Set multicopter layout, dimensions, and weight in CopterSim model configuration, then click “Calculate” and “Store and use parameters.” No DLL import is needed. To change equations or vehicle type, generate and import a DLL as shown here.
- **Scene and initial position:** Match them to vehicle type; fixed-wing aircraft need a takeoff runway. M0 defaults to zero ground slope and supports flat maps only.
- **Multiple vehicles:** Put initial points and operating areas in a BAT launch script, or generate a coordinated configuration with SimCreate. See “Experiment Organization” in Section 2.4.
- **Documentation:** Readme.pdf in the CopterSim and SimCreate subdirectories under [installation directory]/RflySimAPIs/2.RflySimUsage/0.ApiExps/e1_RflySimSoftwareReadme/

4.9.6 Packaging, Deployment, and Loading: Launch Environment

Save Running Experiments Before Launching

- Save the current model, logs, and other CopterSim experiments. Check installation paths, DLL name, mode, airframe, and vehicle count in the launch script.
- The current script copies model files and terminates existing CopterSim processes. On a shared workstation, finish or save affected experiments before running it.
- Launch SITL as instructed and enter 1 vehicle. Inspect copying and startup messages. Elevate privileges only as required by installation and workstation settings.

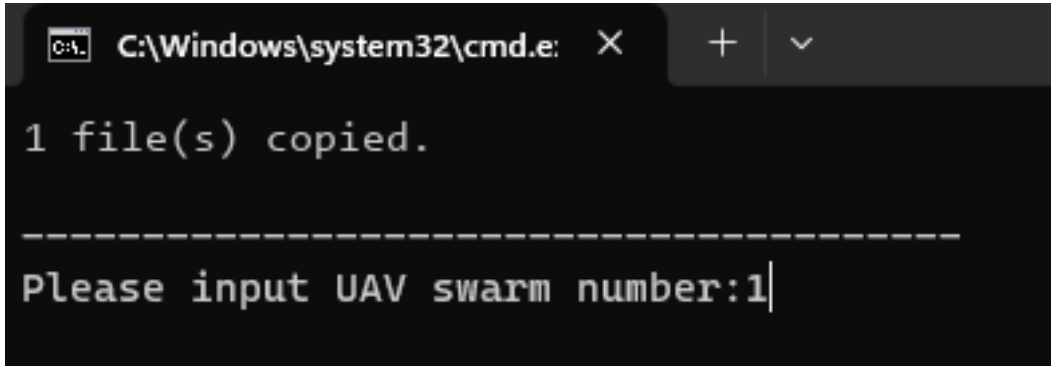


Figure: Vehicle-count prompt (example reference screenshot)

A batch file deploys files and coordinates programs; launching has actual side effects.

4.9.6 Packaging, Deployment, and Loading: Initialization Evidence

After Initialization Is Ready, Read a New State

- The reference shows “CopterSim/PX4 EKF 3D Fixed” at 1/1: the instance and extended Kalman filter (EKF) state-estimator initialization are ready.
- Then check vehicle ID, position, attitude, and increasing time before applying a small known input. Ground-truth change should match the input and model directions.



Figure: Initialization ready (reference screenshot)

Readiness shows the observation chain has started; motion validation also needs time and input evidence.

4.9.6 Packaging, Deployment, and Loading: M0 Quick Reference

M0 Deployment: Entry, Steps, Criteria, and Common Failures

- **Directory and entry:** Read Readme.pdf, then use the SLX and Exp1_MinModelTemp_SITL.bat; [installation directory]/RflySimAPIs/4.RflySimModel/1.BasicExps/e0_MinModelTemp/
- **Steps:** Build the SLX → run GenerateModelDLLFile for Exp1_MinModelTemp.dll → run the SITL batch file as administrator and enter 1.
- **Side effects:** The script copies the DLL to CopterSim/external/model and terminates existing CopterSim processes. Save other experiments first.
- **Criteria:** RflySim3D shows "CopterSim/PX4 EKF 3D Fixed:1/1"; CopterSim reports GPS 3D fixed and EKF initialized. Then take off through QGC.
- **Common failures:** Missing mex compiler configuration causes build failure. "Cannot load library file" often means PX4PSP needs updating. Check hashes if an old DLL still runs.

4.9.6 Packaging, Deployment, and Loading: In-the-Loop Modes

How This Chapter Uses SITL and HITL

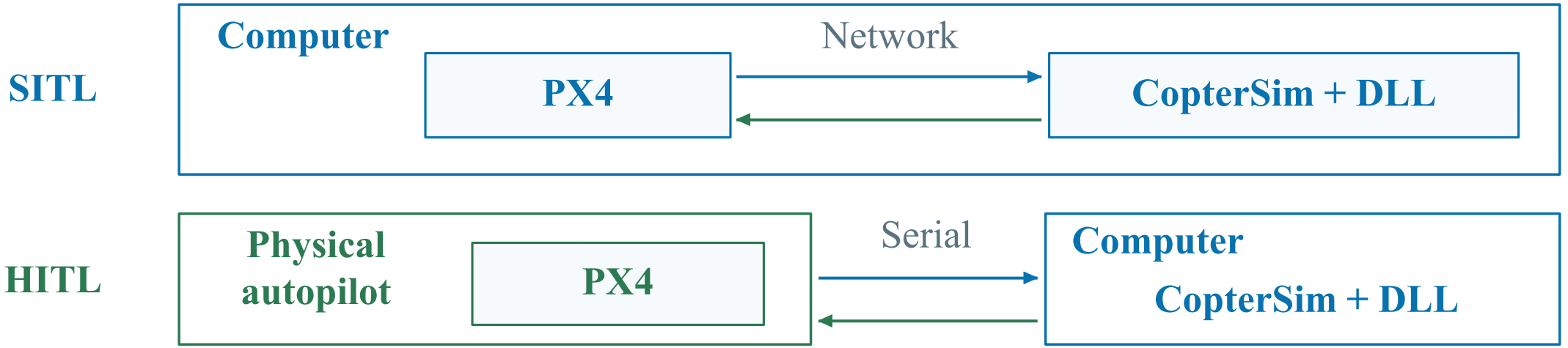


Figure: Dynamics stays on the computer; the controller changes location

- Use existing controllers here to check models and interfaces; Chapter 5 develops control laws.
- SITL: Class-wide reproduction. HITL also needs matching firmware, serial, and hardware timing.

Validate in a shared software environment first, then extend communication and timing checks with hardware.

Chapter Outline

Modeling Tools, Code Generation, and Deployment

Chapter 4
Vehicle Motion Modeling and Simulation

4.1 Modeling Fundamentals and Kinematics
4.2 Rigid-Body Dynamics and Solvers
4.3 Propulsion, Forces, and Observations
4.4 Trim, Linearization, and Dynamics
4.5 Multicopter Modeling and Simulation
4.6 Measurement and Static Identification
4.7 Dynamic Identification and Refinement
4.8 Typical Vehicles and Extensions
4.9 Modeling Tools and Deployment
4.10 API and SDK Modeling Interfaces
4.11 Examples and Integrated Practice
4.12 Exercises and Course Projects

7 Units in This Section

4.9.1 Tools and the Execution Chain
Textbook: Secs. 4.2.1 and 4.2.2.3

4.9.2 MuJoCo Multibody Modeling
Optional: Joints and contact forces

4.9.3 Minimal Templates and Reproducible Projects
Textbook: Secs. 4.2.2.2 and 4.2.3

4.9.4 Numerical Time and Scheduling
Timestep, multiple rates, and wall time

4.9.5 Generating C++ from SLX
Textbook: Secs. 4.2.3 and 4.2.6

4.9.6 Packaging, Deployment, and Loading
Textbook: Secs. 4.2.3 and 4.2.4

4.9.7 Implementation Consistency and Troubleshooting
SLX vs. dynamic-link library (DLL), same conditions

4.9.7 Implementation Consistency and Troubleshooting: Levels of Success

Three Levels of Success Need Different Evidence

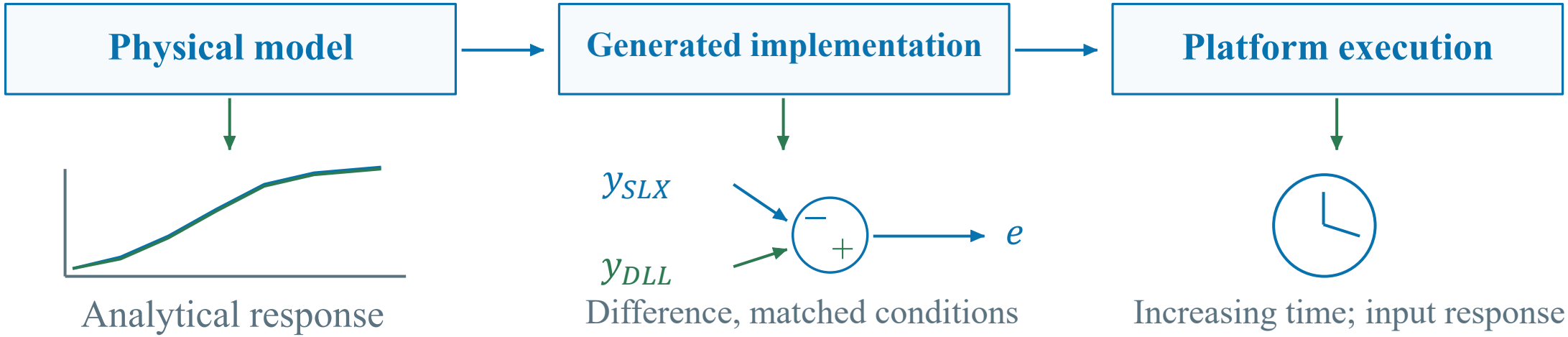


Figure: Attach the relevant evidence at each stage

- **Physical model:** Use analytical responses to check force and motion.
- **Implementation:** Match conditions and system boundaries before comparing SLX and DLL.
- **Platform execution:** Check increasing model time and state response to known inputs.

- Validate real-vehicle accuracy separately with independent data in Sections 4.6 and 4.7.

4.9.7 Implementation Consistency and Troubleshooting: Matched Boundaries

Give SLX and DLL Exactly the Same Experiment

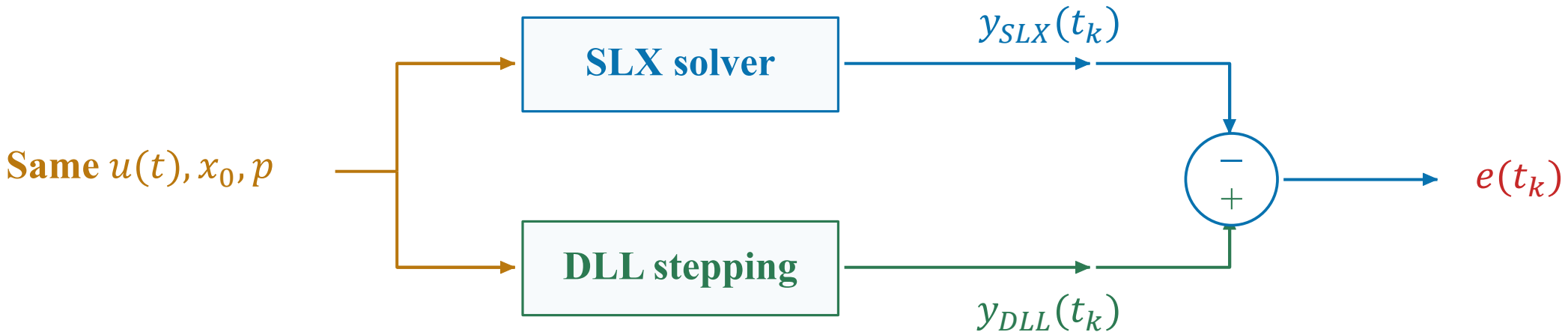


Figure: Two parallel paths sharing experiment conditions

- Match parameters, initial states, inputs, and environment; read the same quantity at equal simulation times.
- An integrated DLL contains a controller. Add a matching-boundary test entry before comparing with bare SLX.

Output differences test implementation only when system boundaries and conditions match.

4.9.7 Implementation Consistency and Troubleshooting: Time and Error

Align Data Before Computing Differences

- Save simulation time in each log, then choose common sample times. State any interpolation method; check input-switch alignment especially near steps.
- Compute differences and maximum absolute error for each output in its physical unit. Do not merge position and angular-rate errors into a unitless score.
- Error shape suggests causes: fixed offset—initial state; step spikes—sample alignment; accumulating drift—timestep and parameters. With matched solver and step, expect roundoff-scale differences.

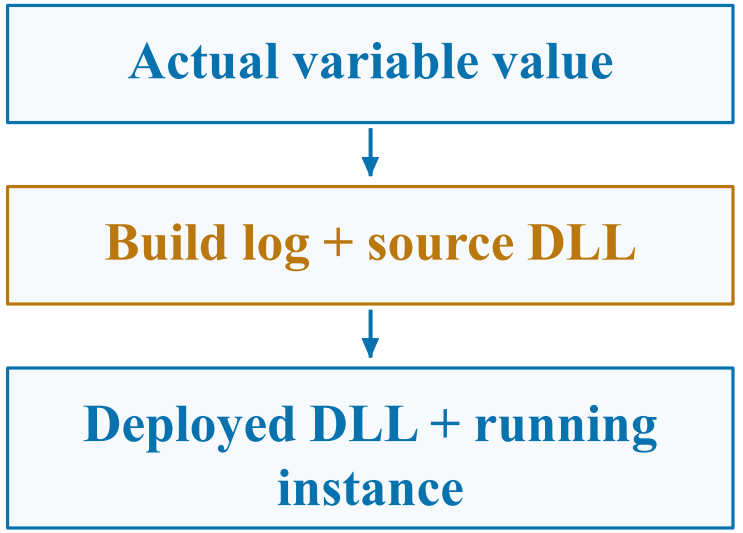
$$e_y(t_k) = y_{\mathrm{DLL}}(t_k) - y_{\mathrm{SLX}}(t_k)$$

Equal array row numbers may represent different times; compute error at equal physical times.

4.9.7 Implementation Consistency and Troubleshooting: Ineffective Parameters

What to Check When a Parameter Change Has No Effect

- **No change after editing:** Read actual model parameters, build log, source/deployed DLL hashes, then confirm the running instance.
- Wrong direction: Check motor channels, rotation directions, and airframe. Fixed position offset: Check origin and initial state. Accumulating error: Check time and parameters.
- Velocity targets pass through the integrated DLL's internal controller, unlike bare dynamics inputs. Section 4.10 identifies boundaries through interface conventions.



One build hash through deployment

If a parameter change produces no response, check the test condition, then the running model.

4.9.7 Implementation Consistency and Troubleshooting: Dependencies and Build Failures

Separate Missing-Library Errors from Compiler Errors

Cannot load library 'pixhawk_slib_adv' referenced by 'pixhawk_slib_adv/CopterForceModel'.
Component: Simulink | Category: Block diagram error

Error message: e0_MinModelTemp/Readme.pdf, FAQ Q2

- A failed block load in pixhawk_slib_adv calls for checking RflySim toolbox paths and versions first. The Readme cites an outdated toolbox as a common cause; do not tune physical parameters.
- If model update succeeds but C++ compilation fails, check compiler selection and the full build log. After repair, revalidate from the failed stage and save results.

Read the first informative error and troubleshoot model dependencies separately from the build environment.

4.9.7 Implementation Consistency and Troubleshooting: Stale States

What to Check When a Display Has No New State

- An open window shows only that a program started. Check increasing ground-truth time, position response to input, and the vehicle ID being read.
- Without stepping, a direct DLL may keep returning its initial state. For a network route, check that the target instance and received data are updating.
- A zero array can also be a fallback for a missing function. Validate exported support, effective input, and time increments together before treating output as actual state.

Check display, cache, and model time separately to establish that simulation is advancing.

4.9.7 Implementation Consistency and Troubleshooting: Development Deliverables

Deliver a Traceable Model Experiment Package

- Continue the offline baseline in Sections 4.5 and 4.11. Save M0 source, parameters, initial states, inputs, and three-layer logs; state physical conditions for analytical comparison. Accept when another person can reproduce the curve.
- Save solver settings, timestep comparisons, and error calculations. If generation is available, include build record, DLL identity, launch settings, and loading evidence. Accept when the deployed DLL hash matches the build record and model time increases.
- If loading a released DLL, identify its source. State missing conditions for unavailable routes; submit physical analysis and interface evidence separately. Every conclusion must point to its supporting log or file.

Deliver model, inputs, logs, and judgments together so the development process can be reviewed.

Class Exercises

Use These Checks to Explain Four Observations

- **Exercise 1:** Change teaching-copy mass from 1.515 kg to 1.8 kg, redeploy, and hover with PX4. Height barely changes. Did the edit take effect? Which quantity should you inspect first?
- **Exercise 2:** In matched SLX/DLL tests, errors are a nearly constant 0.2 m offset, one spike at the t=2 s input step, or linear growth over time. What should you check first for each?
- **Exercise 3:** A script writes 1500 to the first four inPWMs channels. What happens? Approximately what normalized input does M0 need to hover?
- **Exercise 4:** An integrated DLL position response differs greatly from bare M0 SLX output. Does this establish a code-generation error?

Reference Points

Check Conditions and Boundaries, Then Trace Evidence by Layer

- **Exercise 1:** PX4 compensates for mass, so hover height can remain unchanged. Inspect motor input: with M0 and g=9.81 m/s², hover input should rise from about 0.53 to 0.58 (teaching calculation). If input is unchanged, check InitFcn overwrite, build log, source/deployed DLL hashes, and running instance.
- **Exercise 2:** Fixed offset—initial state and origin. Step spike—sample and switching-time alignment. Linear growth—row-based subtraction causing time mismatch first, then timestep and parameters.

- **Exercise 3:** 1500 clips to 0.95, giving about 45.5 N total thrust, roughly three times the 14.9 N weight. The aircraft rises rapidly; hover needs only about 0.53 (teaching calculation).

- **Exercise 4:** No. The integrated DLL includes a controller and takes velocity targets, so its boundary differs from bare SLX. Add a test wrapper with matching root inputs, then compare after time alignment.

Section Summary

Take the Model Reliably from SLX to Deployment

- **Method:** Fix parameters, initial states, inputs, and model time. Then retain evidence at every stage: SLX → generated C++ class → platform-wrapped DLL → host loading.
- **Key judgments:** Build produces program files; Run produces trajectories. Check deployment by DLL hash and increasing model time. Check implementation with matching boundaries and time-aligned differences.
- **Input and parameter pitfalls:** Writing 1500 μs pulse width to normalized inPWMs; editing workspace mass only for InitFcn to overwrite it.
- **Deployment pitfalls:** Leaving an old DLL in place; treating Run as Build and assuming new code was generated.
- **Comparison pitfalls:** Subtracting by array row; comparing an integrated controller DLL directly with bare SLX.