4.10 API and SDK Modeling Interfaces

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.1 Choosing an Interface for the Task: Interface Responsibilities

API and SDK Entry Points by Experimental Task

- An application programming interface (API) specifies call names, parameters, and behavior. A software development kit (SDK) provides wrappers, documentation, and examples.
- Start from the same initial state, change one supported input, and read a new state. The input's physical meaning depends on the current model's internal connections.

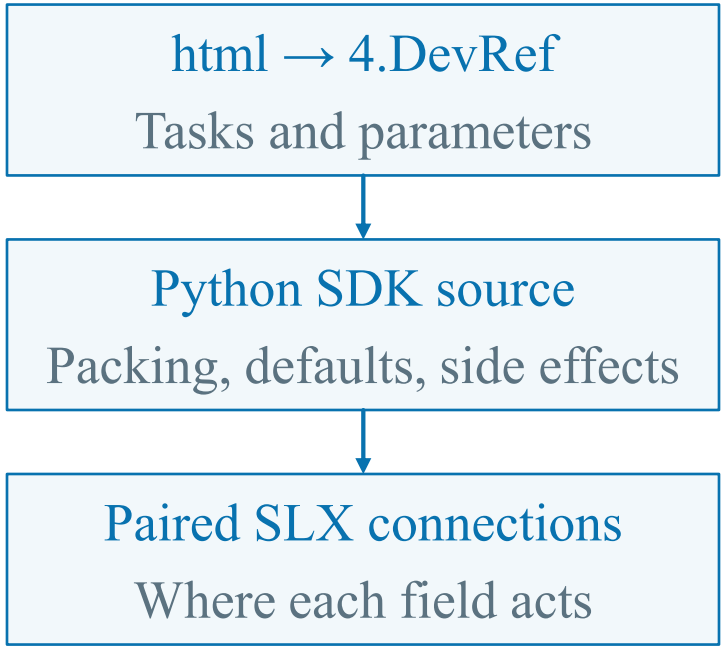
Task	Entry point	Execution location
Set initial state and scene	ReqCopterSim	Request goes to running CopterSim
Send extended inputs; read state	DllSimCtrlAPI network class	CopterSim advances; Python uses User Datagram Protocol (UDP)
Advance an exact number of steps	Direct loading with ModelLoad	Dynamic-link library (DLL) runs in Python

An interface defines data exchange; the development kit packages those conventions into callable tools.

4.10.1 Choosing an Interface for the Task: Local Documentation and Source

Local Documentation Leads to the Actual Implementation

- Open [installation directory]/RflySimAPIs/RflySimSDK/html/index.html, which redirects to 4.DevRef/index.html. Under Python SDK, use the ctrl category to find the interface page.
- Read the task and parameter descriptions, then open the corresponding Python source. Check defaults, packing, return values, and construction or call side effects.
- Finally open the paired SLX in 0.ApiExps and follow input connections to each field's use. The same SDK function may act differently in different models.



Documentation guides the search; current source and the paired model establish the interface contract.

4.10.1 Choosing an Interface for the Task: Python Environment

The Python Environment Supplied with the Installation

```
Python3.8 environment has been set with openCV+pymavlink+numpy+pyulog etc.
You can use pip or pip3 command to install other libraries
Put Python38Run.bat into your code folder
Use the command: 'python XXX.py' to run the script with Python
```

Figure: Entry to the supplied Python terminal

- Run Python38Run.bat in the example directory to use the supplied interpreter. Record the Python version, SDK source, and model file.
- For import failures, check interpreter and SDK paths first. For DLL load failures, check architecture, dependencies, and location. These are not all network failures.

Keep interpreter, module paths, and model files in one traceable environment.

4.10.1 Choosing an Interface for the Task: Different Roles in One Module

Two Classes in One Module Have Different Boundaries

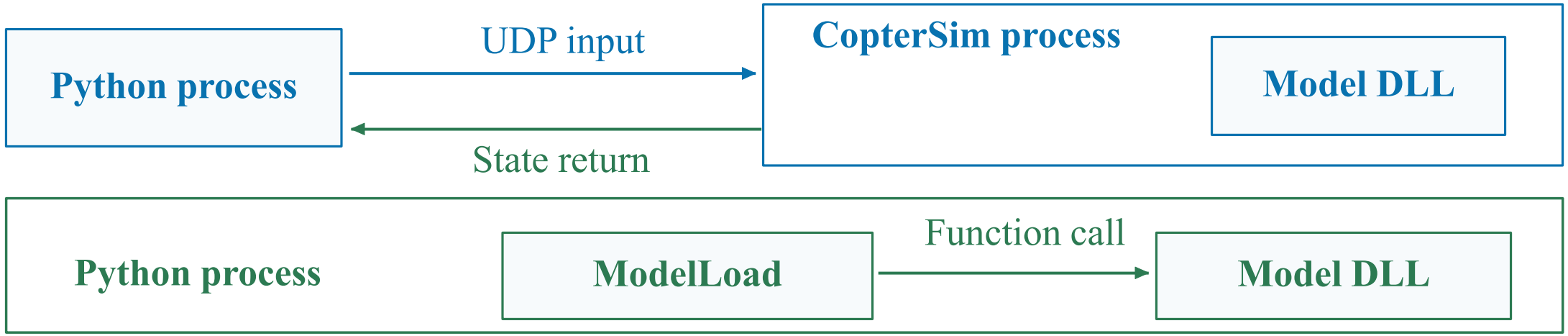


Figure: Cross-process transfer above; direct DLL calls below

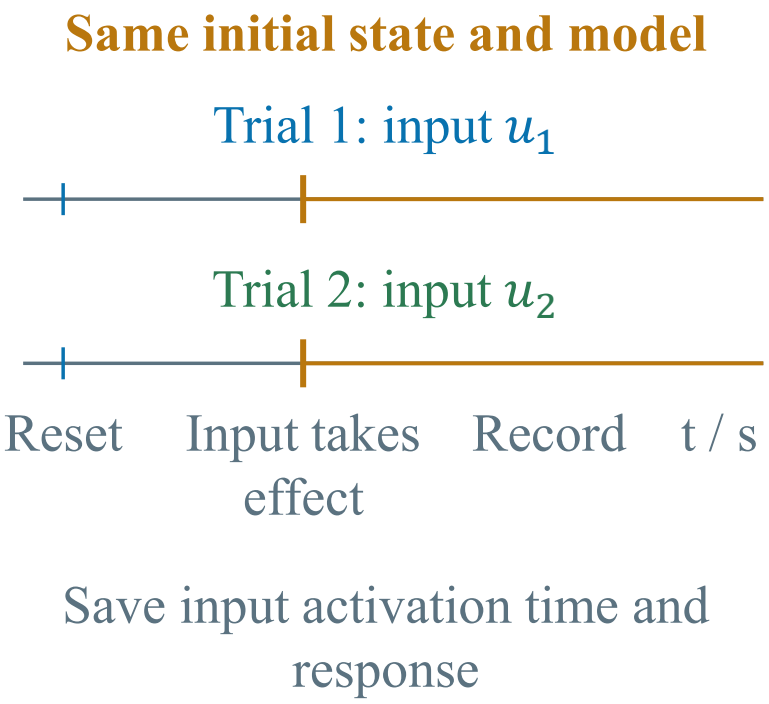
- The network class exchanges UDP data with CopterSim, which advances the model.
- ModelLoad loads inside Python. MultiCopterDll is its subclass with an embedded controller.

The network class sends data; ModelLoad loads and advances the model.

4.10.1 Choosing an Interface for the Task: A Short Experiment

One Short Experiment Answers One Interface Question

- A configuration trial tests whether the initial state took effect. An input trial tests whether a field reached its intended connection. A timing trial tests whether a call advanced the model state.
- Change one supported configuration or input at a time and state the expected effect. Changing model, map, communication mode, and control command together hides causality.
- Save call parameters, input activation time, timestamped outputs, and final status. A normal-looking aircraft screenshot alone cannot establish a correct data path.



Fix the model and initial state, then match input, observation, and success criterion.

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.2 Ports, Fields, and Units: Interface Contract Checks

Structure, Coordinates, and Time in Model Port Values

- The interface contract specifies inputs, outputs, units, frames, rates, and timing between modules. Object, dimensions, and type identify whose data it is and how it is organized.
- The frame specifies directions and origin; units specify scale. Sections of the same array may use different frames and units.
- Time specifies when a sample is valid. Save the model timestamp separately from the program's reception time.

Object	**Vehicle 1**
Representation	**Array / bus**
Frame and units	**North–East–Down (NED), m**
Validity time	**Model timestamp**

Table: Checks and values for this example

Object, dimensions, type, frame, units, and time jointly define data meaning.

4.10.2 Ports, Fields, and Units: Vectors and Buses

Root Ports, Vectors, and Buses

- Minimal model M0, used from Section 4.3, has root inputs inPWMs[16] and TerrainIn15d[15], and root outputs HILSensor30d, HILGPS30d, and the 60-element VehileInfo60d.
- Vectors distinguish elements by index; buses distinguish signals by name. The six-degree-of-freedom (6DOF) bus includes velocity and position vectors, an attitude matrix, and other signals.
- Bus signal count differs from state dimension. A 30-element array does not mean 30 identical sensors. Expand each field needed for this task before connecting it.

Ports connect data channels; buses group diverse signals by name.

4.10.2 Ports, Fields, and Units: The Direct 60-Element Array

Common Fields in the Direct 60-Element Output

- Data3D.update parses the direct 60-element output. Python slices start at 0 and exclude the right endpoint.
- Position is 6:9, containing 3 values. Prefer named out3Ddata attributes instead of rewriting indices in each script.
- The table lists common fields only. Another parser reads network packets with different offsets.

2	Model time	runnedTime
3:6	NED velocity	VelE
6:9	NED position	PosE
9:12	Euler angles	AngEuler
12:16	Quaternion	AngQuatern
16:24	Motor speed	MotorRPMS
27:30	Body angular rate	RateB

Table: Python slices, meanings, and out3Ddata attributes

Use defined DLL-vector indices and prefer the parsed named attributes.

4.10.2 Ports, Fields, and Units: Network Packet Layout

Network Packets and DLL Arrays Need Different Indices

- Time is at index 2 in the direct 60-element vector. Network truth packets unpack as 4i24f7d: 4 ints, 24 floats, 7 doubles. Leading fields include checksum and ID, placing time elsewhere.
- The SDK uses Data3D.update and updateUdp for these two representations. When parsing raw packets, check message length, format, and checksum fields.
- Named attributes unify access to both representations. Before parsing, still check the source interface contract. Custom output uses a separate 32-element message.

Direct DLL: 60 elements



Index 2

Network: 4i24f7d



Network packing can change leading fields and ordering for the same physical quantity.

4.10.2 Ports, Fields, and Units: Root Ports in Two Templates

Two Fixed-Wing Templates Have Different Port Names and Orders

e2_FixWingModelCtrl	
In 1	inPWMs
In 2	inCopterData
In 3	TerrainIn15d
Out 1	HILSensor30d
Out 2	HILGPS30d
Out 3	VehileInfo60d

e5_FixedWingManuscriptMdl	
In 1	inPWMs
In 2	TerrainZ
In 3	inCopterData
Out 1	MavHILSensor
Out 2	MavHILGPS
Out 3	MavVehile3DInfo

Contract: Match the selected model's inputs/outputs, units, frames, rates, and timing.

Adapt to the actual interface contract rather than copying connections from the previous model.

4.10.2 Ports, Fields, and Units: Ground-Vehicle Root Output Order

Different Ground-Vehicle Output Orders Need Explicit Mapping

Common inputs: inPWMs / inCopterData / TerrainIn15d

Model	Out1	Out2	Out3
Differential drive	HILSensor30d	HILGPS30d	VehileInfo60d
Ackermann	VehileInfo60d	HILSensor30d	HILGPS30d

Truth output: Out3 for differential drive; Out1 for Ackermann. Check name, dimension, and physical meaning of each port before connecting.

Similar responsibilities with different indices can still misconnect sensors and truth states.

4.10.2 Ports, Fields, and Units: Normalized Inputs and Physical Quantities

Normalized Actuator Inputs Differ from Wheel Speed and Steering Angle

- Template actuator inputs inPWMs are normalized. Values such as 12 rad/s, 2 m/s, or 26.565° require the internal mapping to determine wheel speed, steering angle, and driving force.
- The local angular rate in differential-drive wheel-speed conversion is positive for left turns. Explicitly convert its sign at the interface to this chapter's NED convention, positive for right turns.
- The Ackermann branch uses steering, drive, and fixed scale factors. MotorRads is only a variable name; confirm each element's units from its calculation.

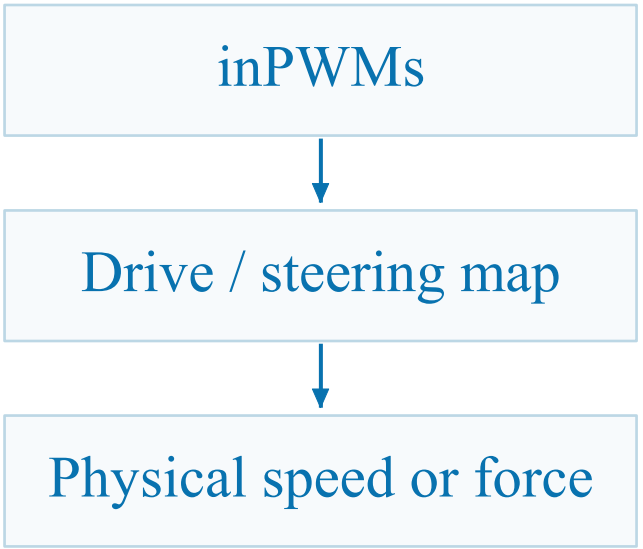


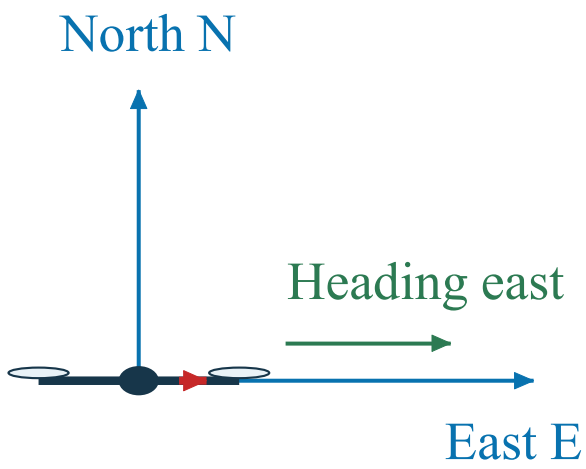
Figure: Read the mapping before assigning units

Map input scale and sign through the selected template before assigning physical meaning.

4.10.2 Ports, Fields, and Units: Coordinate Frames and Positive Directions

NED and FRD in Interface Values

- NED means North, East, Down. Position uses m and velocity m/s. Upward motion at 1 m/s is NED velocity (0,0,−1).
- Forward–Right–Down (FRD) follows body axes. With the nose east, forward velocity is eastward over the ground. The same array cannot be sent directly as NED.
- Check the position origin too. In the current implementation, ModelLoad.SendPosNED adds InitPos, so local and global position readings may differ.



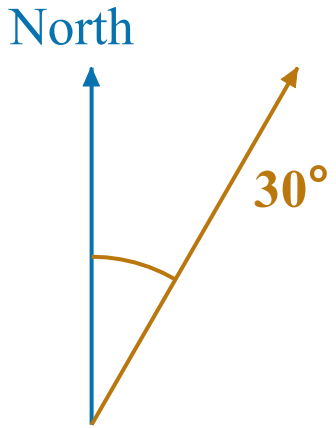
$$\mathbf{v}_{NED} = [0,0,-1]^T \text{ m/s}$$

Directions and origin must be explicit for a number to specify one physical command.

4.10.2 Ports, Fields, and Units: Angle Units

Initialization and State Angles Use Different Units

- XYyaw and XyzRPYaw initial-state requests use degrees. Model Euler angles and Data3D.AngEuler use radians.
- Requesting 30° heading should return about 0.5236 rad. Convert first, then confirm with the first new sample.
- Body angular rate uses rad/s and differs from attitude angle. Keep quantity, direction, and units in log column names.



$$\psi_{rad} = \psi_{deg}\frac{\pi}{180}, \qquad 30^\circ = 0.5236 \text{ rad}$$

Readback 0.5236 rad

Configuration uses degrees and state output uses radians. Convert at the interface boundary.

4.10.2 Ports, Fields, and Units: Encoding Scales

Motor Speed and GPS Encodings Also Need Scale Conversion

- Force calculations use rad/s; MotorRPMS uses rpm. 470 rad/s is about 4488.2 rpm. Convert before comparing.
- Global Positioning System (GPS) data encodes latitude/longitude multiplied by 10^7, altitude in mm, and velocity in cm/s. Decode before comparing with model states.
- Altitude 50 m is 50000 mm; 1 m/s is 100 cm/s. Also distinguish NED down from upward altitude and check their origins.

$$n_{rpm} = \frac{60}{2\pi}\omega_{rad/s}$$

Example: Unit conversion

470 rad/s × 60/(2π)
≈ 4488.2 rpm

50 m × 1000
= 50000 mm

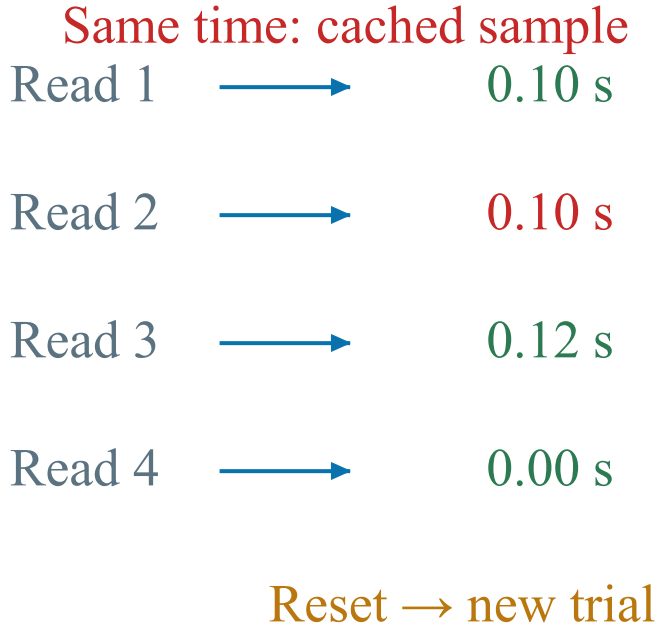
1 m/s × 100
= 100 cm/s

Label physical quantities, display units, and transmitted integer scales separately.

4.10.2 Ports, Fields, and Units: Recognizing New Samples

A New State Needs a New Model Timestamp

- runnedTime is model runtime in seconds. Save it with every record and compare it with the previous sample to detect duplicates.
- Wall-clock reception time helps analyze network and scheduling delay but cannot replace simulation time for trajectory alignment. Save both separately to explain differences.
- Reset can return simulation time to its start. Create a new trial ID; requiring time to increase monotonically across resets would reject valid data.

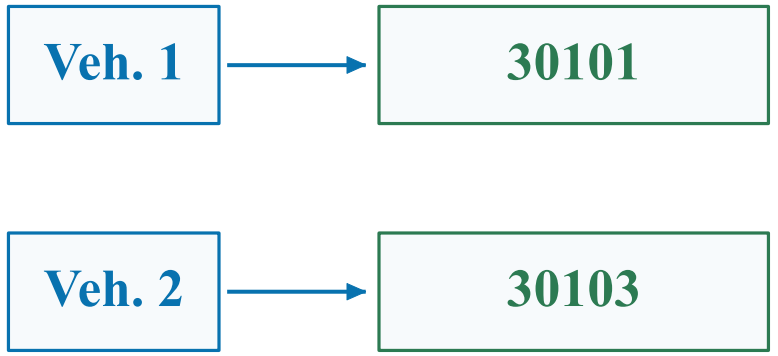


Changing simulation time indicates a new sample; reception time records when the program read it.

4.10.2 Ports, Fields, and Units: Network Object IDs

Network Truth Data Needs an Extra Object-ID Check

- The current updateUdp parser updates motion and time but does not write Data3D.CopterID. The attribute can retain its construction default of 0.
- Confirm the target with the discovery list, receive port, and object ID in the raw packet. Treat out3Ddata.CopterID only as a reference.
- Listen on 30101 for vehicle 1 and 30103 for vehicle 2. When parsing raw packets, check checksum, object ID in the message, and sender address too.



CopterID may still be 0

Confirm the target using discovery and data endpoints; a default field may remain unchanged.

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.3 Configuration and Reproducible Initialization: Target Discovery

Instance Discovery Comes Before Configuration

- ReqCopterSim listens for heartbeats and builds instance records during construction. getAllCopterID and getSimIpList expose IDs and addresses.
- getSimIpID searches and tries a broadcast query, with fallback behavior if unsuccessful. A printable IP does not automatically prove discovery of the target.
- Start software-in-the-loop (SITL; SIL in the textbook; introduced in 4.9), confirm the target ID, then create the request object. If discovery fails, check runtime status and network adapter first.

```
Start listening CopterSim heartbeat Msg ...
End listening CopterSim heartbeat.
Got 1 CopterSim on the LAN.
[(1, '192.168.1.100')]
192.168.1.100
Sending online-mode reset command
```

Figure: Discovered IDs and IPs (example documentation screenshot)

Discovery lists current instances; a default IP is only an addressing fallback.

4.10.3 Configuration and Reproducible Initialization: Discovery Code

Short Code: Checking the Discovery List

- Start the target SITL instance first. Stop configuration if discovery fails.

```
import ReqCopterSim
req = ReqCopterSim.ReqCopterSim()
vehicle_id = 1
if vehicle_id not in req.getAllCopterID():
    raise RuntimeError("Target not discovered")
print(req.getSimIpList())
print(req.getSimIpID(vehicle_id))
```

- **Target ID:** getAllCopterID must contain 1 before configuration continues.
- **Returned address:** Check the ID before reading the IP list and target IP. An address fallback does not establish discovery.

4.10.3 Configuration and Reproducible Initialization: Example Entry Point

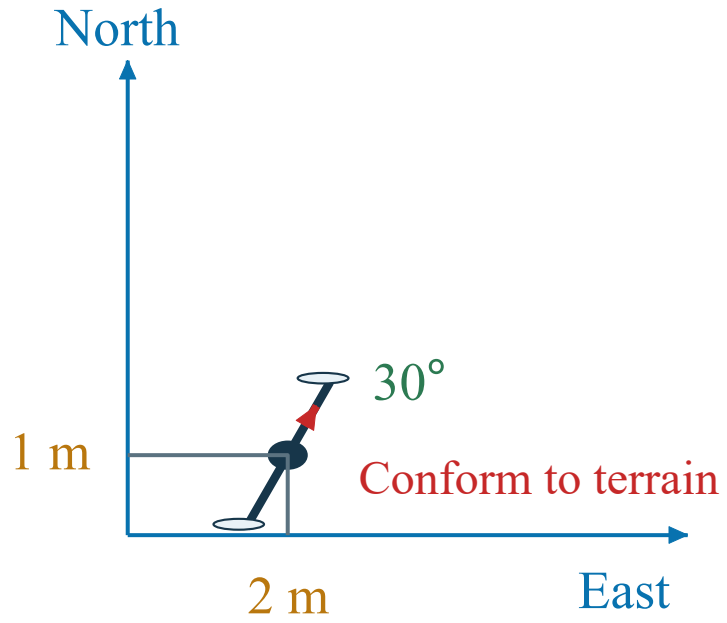
Example: Initial State and Scene Reset with ReqCopterSim

- **Directory:** [installation directory]/RflySimAPIs/4.RflySimModel/0.ApiExps/4.InitAPI/1.ReqCopterSim/. Read its Readme.pdf first.
- **Entry:** MulticopterNoCtrl_SITLRun.bat, Python38Run.bat, testReqCopterSim.py.
- **Steps:** Copy HexarotorModelCTRL.dll to [installation directory]/CopterSim/external/model. Run SITLRun.bat as administrator and enter 1. After initialization, run python testReqCopterSim.py in the Python38Run.bat terminal.
- **Expected:** UDP_Mode changes from UDP_Full to MAVLink_Full; x, y, yaw initial values change and the vehicle jumps in Unreal Engine (UE); the model becomes a hexarotor; Grasslands changes to VisionRing.
- **Side effects:** The bat first terminates running CopterSim instances. Pressing any key to exit closes CopterSim, QGroundControl (QGC), and RflySim3D. The script finally arms and flies in Offboard to z=−20 m.
- **Checks:** If model switching fails, check the target DLL in external/model. If discovery is empty, check running SITL and the network adapter.

4.10.3 Configuration and Reproducible Initialization: Planar Initial-State Request

One XY and Heading Reset Request

- After discovering instance 1, request 1 m north, 2 m east, and 30° heading. This entry point does not explicitly set height.
- Height is unspecified by this request. Source defaults place the vehicle on terrain with zero roll and pitch. Read the first new sample to confirm position and heading.
- With PX4, also wait for estimator initialization. The vehicle continues moving after its initial-state reset.



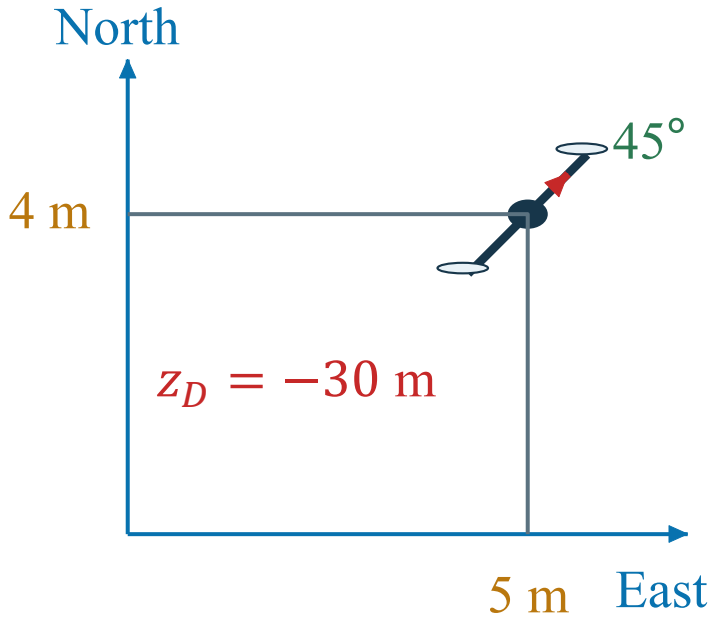

```
req.sendReSimXYyaw(1, [1, 2, 30])
```

A single request changes only its declared initial state, simplifying before-and-after comparison.

4.10.3 Configuration and Reproducible Initialization: Full Initial-State Request

An Explicit Entry Point for Height and Attitude

- [4,5,−30] is an NED position 30 m above the local origin, rather than altitude 30 m above sea level. The attitude request still uses degrees.
- Check initialization with the earliest new sample. Without thrust supporting its weight, the aircraft descends from its airborne initial state; reset does not hold position.



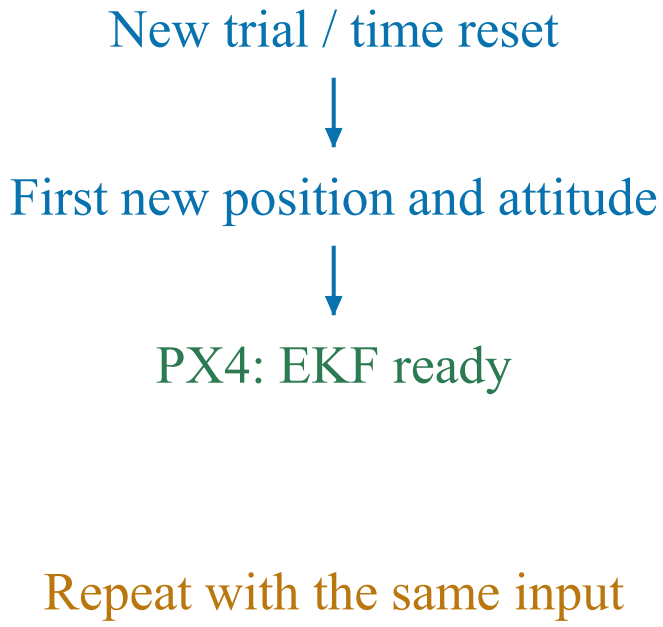

```
req.sendReSimXyzRPYaw(1, [4, 5, -30], [0, 0, 45])
```

XYZ and roll, pitch, yaw use separate vectors in meters and degrees, respectively.

4.10.3 Configuration and Reproducible Initialization: Reset Readback

Three Signals Confirm Different Parts of a Reset

- Create a new trial ID after reset and check the first new position and attitude. Increasing time alone only shows that the model is advancing.
- With PX4, wait for extended Kalman filter (EKF) readiness. Bound the wait and save the last state on timeout.
- Repeat twice from the same initial state and input; compare on shared time. With noise, fix random conditions or compare statistics.



A new trial, the new initial state, and estimator readiness answer different questions.

4.10.3 Configuration and Reproducible Initialization: Model and Map Configuration

Managing Maps, Models, and the GPS Origin

- Changing a DLL changes model structure or parameters. Changing a map may change terrain and reference environment. Both affect repetition and must be logged with control commands.
- Set the GPS origin with sendReGPSOrin(ID, [latitude°, longitude°, altitude m]), with altitude positive upward. Use sendReSimMapName and sendReSimDllName for maps and DLLs.
- If a configuration entry point is unavailable, restart the baseline with the same startup configuration. Matched initial state, input, and recording conditions still support repeatability practice.

Change one comparison-affecting setting at a time and record its source.

4.10.3 Configuration and Reproducible Initialization: Two Initialization Entry Points

Startup Parameters and Network Resets During Operation

- **Template parameters:** Rows 1–3 are in *_init.m (2.AdvExps/e1_MaxModelTemp), loaded at compilation and partly overwritten at CopterSim startup. Row 4 is a runtime request.

Entry point	Units	When effective / who overwrites it
ModelInit_PosE [x, y, z]	m, NED	Startup: x and y from the CopterSim interface; z from map terrain height
ModelInit_AngEuler	rad	Startup: roll and pitch from this value; yaw from the CopterSim interface
ModelParam_GPSLatLong and envAltitude	degrees; m, positive down	Startup: GPS origin; map txt coordinates can overwrite it
sendReSimXYyaw, sendReGPSOrin, sendReSimXyzRPYaw	m and degrees	Runtime: Ask CopterSim to reset; confirm with the first new sample

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

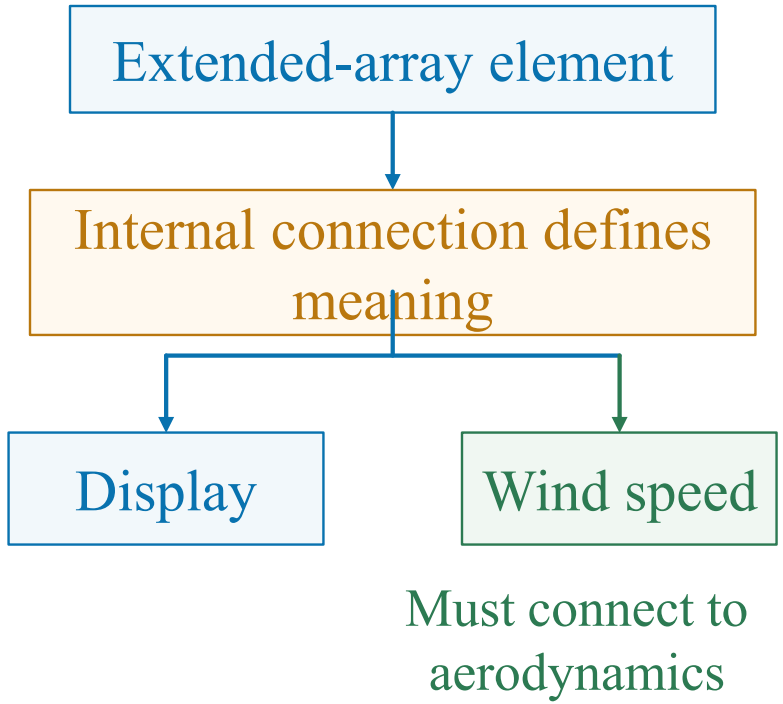
4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.4 External Inputs and Logs: Where an Input Acts

Internal Connections Give External Inputs Their Meaning

- A send function organizes type and length. The current SLX decides whether a value changes mass, wind, a fault, or display.
- Record both the received port value and the affected internal quantity. For display type, inspect 3DType; for wind, inspect relative airspeed and aerodynamic force.
- If the input changes but force does not, check connections, units, and enable conditions. Exposing a tunable parameter does not automatically create a network write path.
- **Other channels:** FaultParamAPI.FaultInParams receives fault parameters; inCollision20d receives collision information.



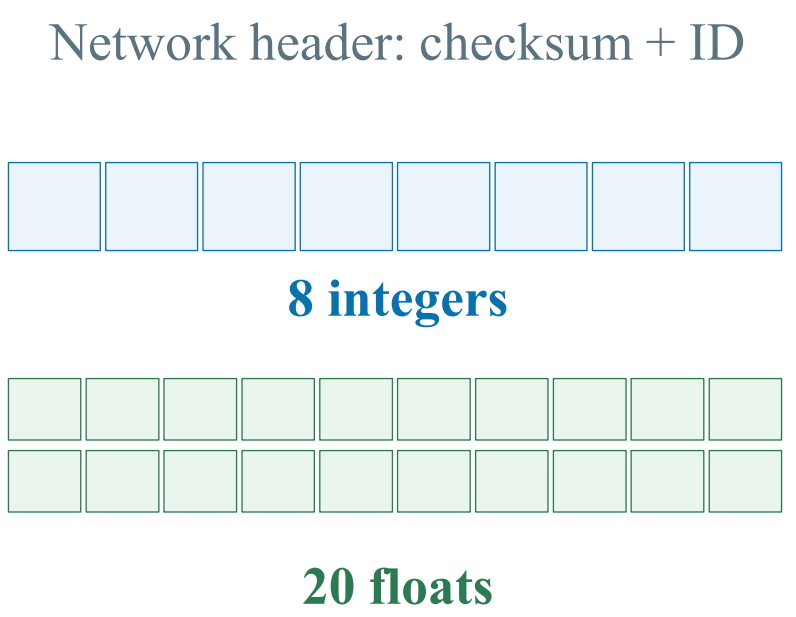
After a value reaches a port, identify which model element it affects.

4.10.4 External Inputs and Logs: Extended Input Capacity

What sendSILIntFloat Actually Sends

- sendSILIntFloat pads/truncates to 8 integers and 20 floats, prepends checksum and object ID, then packs 10i20f: 10 ints and 20 floats.
- Input port is $30100 + 2(ID - 1)$: 30100 for vehicle 1, 30102 for vehicle 2. Truth returns on the next odd port.
- Truth reception: Listen on 30101 for vehicle 1 and 30103 for vehicle 2, each input port + 1.

$$P_{in} = 30100 + 2(ID - 1)$$



The protocol defines delivery; model connections define effects.

4.10.4 External Inputs and Logs: Example Entry Point

Example: Display Changes Through inSIL Extended Inputs

- **Definition:** inSIL is the model DLL's external integer/float input port.
- **Directory:** [installation directory]/RflySimAPIs/4.RflySimModel/0.ApiExps/11.inSILAPI/1.InSILIntsFloats/3.inSILIntsFloats_py/. Read its Readme.pdf first.
- **Entry:** MulticopterNoCtrl_SITLRun.bat, Python38Run.bat, inSIL.py.
- **Steps:** Run SITLRun.bat as administrator and enter 1. Run python inSIL.py in the Python38Run.bat terminal. After SLX changes, recompile and run GenerateModelDLLFile.p to generate the DLL.
- **Expected:** Setting silInt[0] to 5, 8, 10, then 50 changes the RflySim3D display to a hexarotor, octorotor, four-arm octorotor, then a car.
- **Side effects:** The bat first terminates running CopterSim. Pressing any key to exit closes CopterSim, QGC, and RflySim3D.
- **Checks:** For unchanged appearance, first check that inSILInts and inSILFloats are both connected in MulticopterNoCtrl.slx, then check CopterID and input port 30100.

4.10.4 External Inputs and Logs: Send Code

Short Code: Explicitly Setting the First Extended Input

- Use the inSIL example's SITL instance and send first element 5.

```
from DllSimCtrlAPI import DllSimCtrlAPI

link = DllSimCtrlAPI(CopterID=1)
ints = [0] * 8
floats = [0.0] * 20
ints[0] = 5
floats[0] = 5.0
link.sendSILIntFloat(ints, floats)
```

- **Contract:** Target 1; 8 integers, 20 floats. First values 5 and 5.0; all others zero.
- **Model use:** The first integer feeds 3DType and changes appearance in the 3D scene.

4.10.4 External Inputs and Logs: Display-Type Connection

The Integer Example Feeds Display Type

- **Integer path (red box):** inSILInts[1] → Selector → 3DType. Element [1] is Python index 0.
- **Float path:** inSILFloats elements 1–16 → Selector → double → ExtToUE4.

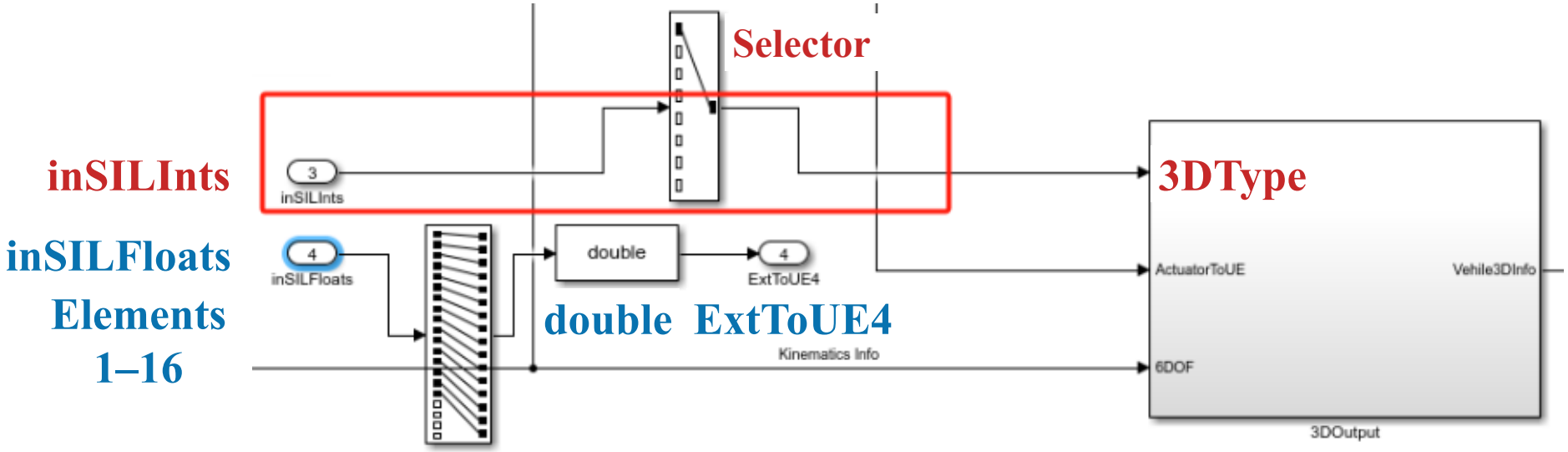


Figure: Actual extended-input connections in the example SLX (detail)

Follow the send index into the actual SLX connection to establish the expected effect.

4.10.4 External Inputs and Logs: Float Connections

The Current Model Defines the Float Example

- The current example has 20-element inSILFloats. Its first 16 elements are selected, converted to double, and sent to ExtToUE4 as external display data.
- ExtToUE4 passes these values to RflySim3D for display or effects, without entering force calculations. The first integer separately feeds 3DType.
- Wind injection requires connecting the relevant component to relative airflow or aerodynamic force inside the model, then adding unit, direction, and response checks.

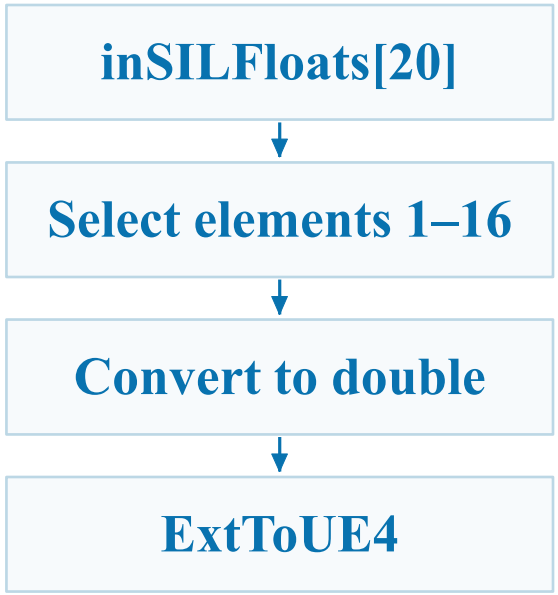


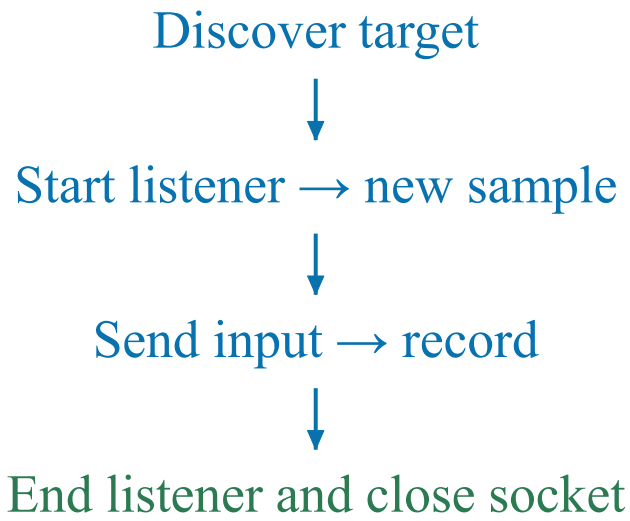
Figure: Actual extended float-input path

The actual SLX establishes current field destinations more reliably than an old screenshot.

4.10.4 External Inputs and Logs: Reading Motion State

Starting and Ending Truth Reception Must Be Paired

- Start CopterSim and discover the target. InitTrueDataLoop binds the return port and starts a thread. Then confirm a new sample.
- Send one supported field and record input, model time, and output. CopterSim advances dynamics; Python does not call Dllstep.
- End under a finite condition. EndTrueDataLoop stops reception and closes the socket. Resolve port occupancy before retrying a conflict.



The receive thread refreshes attributes; reading them does not advance CopterSim.

4.10.4 External Inputs and Logs: Reading Motion State

Short Code: Motion Readback and New-Sample Detection

```
import time, DllSimCtrlAPI
link = DllSimCtrlAPI.DllSimCtrlAPI(CopterID=1)
link.InitTrueDataLoop()
rows, t_end = [], time.time() + 5
try:
    while time.time() < t_end:
        d = link.out3Ddata
        if d.runnedTime != (rows[-1][0] if rows else 0):
            rows.append([d.runnedTime, *d.PosE])
        time.sleep(0.005)
finally:
    link.EndTrueDataLoop()
```

- **New sample:** Append only when runnedTime differs from the previous row; compare the first row against default 0. Split trials after resets.
- **Finite ending:** The 5 s limit ends the loop. EndTrueDataLoop in finally stops the thread and closes the socket.

4.10.4 External Inputs and Logs: Custom Output

outCopterData Exposes Internal Quantities

- outCopterData is a 32-element double output. The example uses 1:32 to check order; these are not 32 motion states.
- CopterSim1.csv is for vehicle 1. Check added rows, column count, and the sequence 1 to 32. File existence does not establish active sampling.
- The SDK saves the same 32 values in outCopterVect, a separate message from out3Ddata. Save field definitions and units when adapting the log.

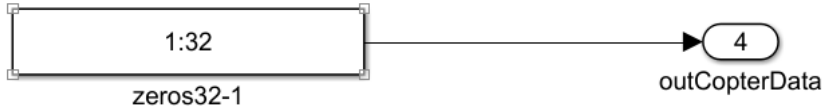


Figure: Checking custom output with a known sequence (example screenshot)

Custom output records selected internal signals; the model defines each field's meaning.

4.10.4 External Inputs and Logs: Example Entry Point

Example: outCopterData Custom Output and Logs

- **Directory:** [installation directory]/RflySimAPIs/4.RflySimModel/0.ApiExps/8.OutCopterData/. Read its Readme.pdf first.
- **Entry:** Exp2_MaxModelTempSITL.bat. LogSelectModel in Exp2_MaxModelTemp.slx defines fields; the example uses 1 to 32.
- **Steps:** Create an empty CopterSim1.csv in [installation directory]/CopterSim. Run Exp2_MaxModelTempSITL.bat as administrator and enter 1. After a simulation interval, press any key in the command window to end.
- **Expected:** CopterSim1.csv gains rows with columns 1 to 32. UDP also sends these values to the 30101 port series; the SDK stores them in outCopterVect.
- **Side effects:** The bat first terminates running CopterSim and closes CopterSim, QGC, and RflySim3D on exit. Use a new file or back up the old log for each trial.
- **Checks:** If empty, check the file is in CopterSim and its numbered name matches CopterID. Field changes require recompilation and a new DLL.

4.10.4 External Inputs and Logs: Custom Log Fields

Internal Signals as an Identification Record

- **Check the channel first:** Use the 1:32 example in 0.ApiExps/8.OutCopterData to verify output order. After confirming new rows, replace constants with the required internal signals.
- **Define fields:** For example, model time, actual normalized input, motor angular speed, body torque, and body angular rate. Give each column a name, unit, frame, and sampling location.
- **Keep synchronization:** Output all quantities at one model instant. Display refresh and packet reception times cannot directly replace integrator simulation time.

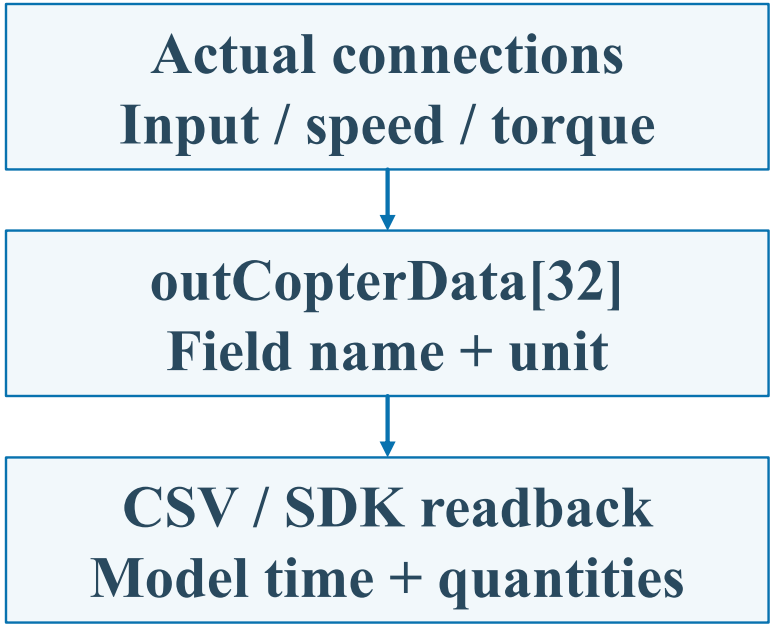


Figure: Logged internal signals

Field definitions turn internal physical quantities into reproducible data rather than unnamed numbers.

4.10.4 External Inputs and Logs: From Logs to Identification

An Input Port Defines the System to Identify

Actual motor input → u_norm → **Drive + motor** → omega_rad_s → **Angular speed**

- **Actuator:** Use the actual normalized motor input, rather than a position target.

Actual net body torque → $\mathbf{M}_B$ / N·m → **Rigid-body rotation** → $\boldsymbol{\omega}_B$ / rad/s → **Body angular rate**

- **Body:** Net torque includes damping and other external effects; do not subtract damping twice.

Velocity target → $\mathbf{v}_{\mathrm{ref}}$ / m/s → **Controller + plant** → $\mathbf{v}$ / m/s → **Actual velocity**

- **Closed loop:** The controller is in the path. Label this boundary explicitly (methods in Section 4.7).

Figure: Different log inputs define different identification boundaries

Define the input-output boundary before attributing parameters to a physical process.

4.10.4 External Inputs and Logs: Checks Before Algorithm Input

Time and Validity Checks Before Processing a Log

- **File contract:** Export the three defined columns to trial.csv: t_s, u_norm, omega_rad_s. This model experiment generates the file.

- **Inspect data first:** Check sample count, finite values, and increasing time. Before uniformly sampled identification, also check approximately consistent sample intervals. Then analyze input and response (methods in 4.7).
- **Processing rules:** Preserve raw logs when duplicates, out-of-order samples, or gaps occur. Describe selection and alignment, then save a separate algorithm-input file.

Establish valid time and data before asking an algorithm to explain the input-response relation.

4.10.4 External Inputs and Logs: Checks Before Algorithm Input

Code: Log Time and Finite-Value Checks

```
import numpy as np
a = np.genfromtxt(
    'trial.csv', delimiter=',', names=True)
t, u, w = a['t_s'], a['u_norm'], a['omega_rad_s']
assert t.size > 2
assert np.isfinite([t, u, w]).all()
assert np.all(np.diff(t) > 0)
print('Samples, duration:', t.size, t[-1] - t[0])
```

- **Sample interval:** Increasing time is not uniform sampling. Check intervals before identification and retain raw logs.

- **t.size > 2:** At least 3 values.
  If fewer, check the export.
- **isfinite:** Time, input, and speed must be finite.
  Check NaN and Inf.
- **diff(t) > 0:** Strictly increasing time. For repeats or rollback, check trials and ordering.

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.5 Direct Loading and Time Advancement: Construction and the Actual Starting State

Prerequisites for the Direct-DLL Example

- **Directory:** [installation directory]/RflySimAPIs/4.RflySimModel/0.ApiExps/12.DllModelImport/4.ModelLoadMannual/ Entry: ModelLoadApiTest.py and Python38Run.bat.
- ModelLoad uses Python's standard-library ctypes to load the DLL. CreateVehicle then sets position, attitude, and terrain. Construction already initializes and advances once.
- Record actual time and state after construction as the starting point. Do not repeat CreateVehicle. Dependencies must match the environment.

ctypes loads DLL
↓
CreateVehicle initializes
↓
First advance
↓
Read actual start

Running without CopterSim still requires model files, maps, and matching libraries.

4.10.5 Direct Loading and Time Advancement: Exported Function Support

Export Checks for the Current DLL

- DLL exports differ. The current SDK often uses hasattr checks and may skip a call or return defaults if absent. Verify support through exported functions.
- For stepwise execution, check at least Dllstep, the actual step size, and state output. Stop this experiment if a required function is missing; do not interpret a zero array as a stationary state.
- If DllGetStep0 is absent, the SDK falls back to 0.001 s. That does not prove a real 1 ms DLL step. Obtain evidence from exports or generation settings.

Exported function	Role
Dllstep	Advance one step
DllGetStep0	Actual step size
DlloutVehileInfo60d	60-element output

Default value ≠ support evidence

A Python method can exist even when a DLL lacks its corresponding computational entry point.

4.10.5 Direct Loading and Time Advancement: Code for the Starting State

Short Code: Reading the Actual Post-Initialization State

- Create the paired integrated model and check base-step and truth exports.

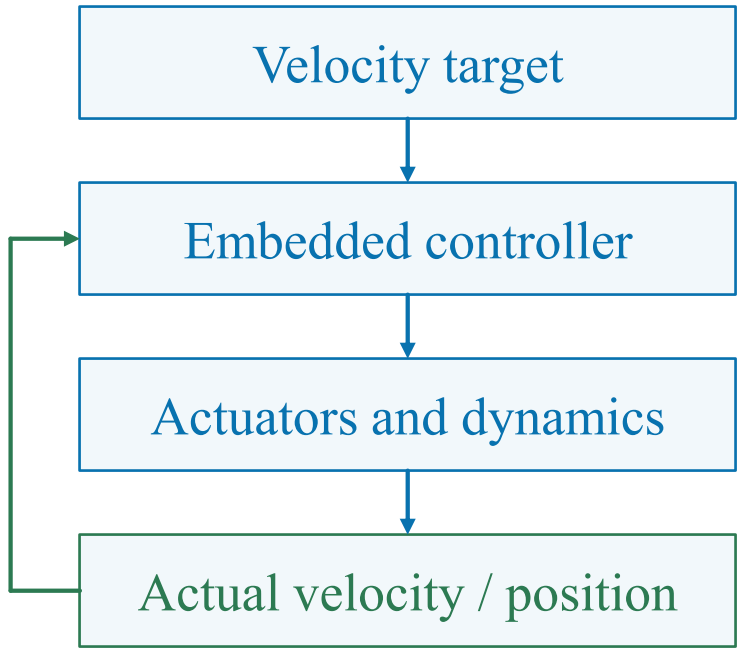
```
import DllSimCtrlAPI
model = DllSimCtrlAPI.MultiCopterDll()
required = ["Dllstep", "DllGetStep0",
            "DlloutVehileInfo60d"]
if not all(hasattr(model.dll, f) for f in required):
    raise RuntimeError("Required DLL export missing")
print(model.out3Ddata.runnedTime)
print(model.out3Ddata.PosE)
```

- **Support:** Dllstep, DllGetStep0, and the 60-element truth export must all exist. Stop this experiment if any is missing.
- **Actual start:** Construction already initializes and advances once. Save runnedTime and PosE; do not assume zero height.

4.10.5 Direct Loading and Time Advancement: Velocity Targets in an Integrated Model

Velocity Targets Pass Through the Embedded Controller

- SendVelNED(0,0,−1) requests upward velocity 1 m/s. The embedded controller converts state error to actuator action, then dynamics computes actual motion.
- Actual velocity approaches the target with lag. Displacement is usually less than target velocity times advanced time. Read VelE and PosE for the actual response.
- The response includes controller and dynamics. For bare-motor or rigid-body identification, use a test input acting directly on actuators instead.



Commands specify desired motion; closed-loop response determines actual speed and displacement.

4.10.5 Direct Loading and Time Advancement: Integer Time Steps

How ModelUpdate Converts Time to Steps

- ModelUpdate advances $N = \mathrm{int}(\Delta t/h)$ steps without storing a remainder. With $h = 1$ ms, 0.5 ms advances 0 steps; 10.5 ms advances 10.
- Use manual integer stepping for quantitative comparisons and confirm actual advancement from returned model time.



Actual: 0 steps



Actual: 10 steps = 10 ms

Remaining 0.5 ms is discarded

The current implementation advances integer steps and does not carry a fractional remainder into the next call.

4.10.5 Direct Loading and Time Advancement: Integer-Step Code

Short Code: Advancing 1000 Steps Before Readback

- Reuse model from the starting-state code. Apply an upward velocity target and advance 1000 steps.

```
h, t0 = model.DllGetStep0(), model.out3Ddata.runnedTime
model.SendVelNED(0, 0, -1)
for _ in range(1000):
    model.Dllstep()
model.Updata3Doutput()
print("expected dt", 1000 * h)
print("actual dt", model.out3Ddata.runnedTime - t0)
print(model.out3Ddata.PosE)
```

- **Request:** NED(0,0,-1) targets 1 m/s upward. Advance 1000 base steps.
- **Expected:** For h=1 ms, actual dt is 1 s. VelE[2] approaches −1 m/s with lag; climb is usually less than 1 m.

4.10.5 Direct Loading and Time Advancement: Published Example Output

Actual Readback in the Published Example

- **Commands:** SendVelNED(0,0,−10), then ModelUpdate(1); SendVelNED(10,0,0), then ModelUpdate(10). Print localPos, PosE, and VelE after each. This target is −10 m/s; the 1000-step code uses −1 m/s.

```
[1.6666424061912851e-16, 4.180508356564208e-17, -5.6306766265134165] [1.6666424061912851e-16, 4.180508356564208e-17, -13
.680676817248 28] [5.737159294107992e-17, 1.7120228701808106e-16, -9.866896579793057]
[78.56886111450616, -2.3218608213043204e-15, -9.955108932450404] [78.56886111450616, -2.3218608213043204e-15, -18.005109
123185267] [9.959637020888362, -4.772923161087532e-16, 0.02391732914028788]
```

Figure: State after manual advancement (runtime output in the example documentation)

- **Readback:** After 1 s, local z=−5.63 m and vertical velocity −9.87 m/s; acceleration continues.
  The two z positions differ by about 8.05 m, from terrain height in the starting InitPos.

The example returns state responses rather than simply echoing target commands.

4.10.5 Direct Loading and Time Advancement: Automatic Updates and Integration Rate

Model Time Advanced by One Automatic Update

- **Calls:** StartAutUpdate defaults to outer updates at about 30 Hz and requests ModelUpdate from wall-clock intervals and a multiplier. This refresh rate is separate from integration; one update can contain many steps.
- **Example:** If an update requests 0.0337 s with a 0.001 s base step, the current ModelUpdate truncates to 33 steps, advancing 0.033 s.
- **Recording:** Save model and wall-clock times before and after each update. Use automatic playback for trends; prefer explicit integer steps for quantitative repeated trials.

$$N = \left\lfloor \frac{0.0337}{0.001} \right\rfloor = 33, \qquad \Delta t_{\text{model}} = Nh = 0.033\,\text{s}$$

Record display pacing, requested advancement, and actual model time separately.

Chapter Outline

Reproducible Model Experiments with API and SDK

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

6 Units in This Section

4.10.1 Choosing an Interface for the Task
Textbook: Sec. 4.2.6

4.10.2 Ports, Fields, and Units
Textbook: Secs. 4.2.3, 4.2.5.2–4.2.5.3

4.10.3 Configuration and Reproducible Initialization
Textbook: Sec. 4.2.3

4.10.4 External Inputs and Logs
Textbook: Sec. 4.2.3

4.10.5 Direct Loading and Time Advancement
Textbook: Sec. 4.2.4

4.10.6 Lifecycle and Fault Isolation
Experiment lifecycle, cleanup, and unchanged-output diagnosis

4.10.6 Lifecycle and Fault Isolation: Network Host Route

The Complete Network-Experiment Lifecycle

- Four stages: Start and discover, confirm start, advance and record, then stop and clean up. See the short motion-readback and new-sample code.
- End after a finite simulation duration or sample count. Close only resources created for this trial; then save logs and runtime configuration. Compare both routes below.

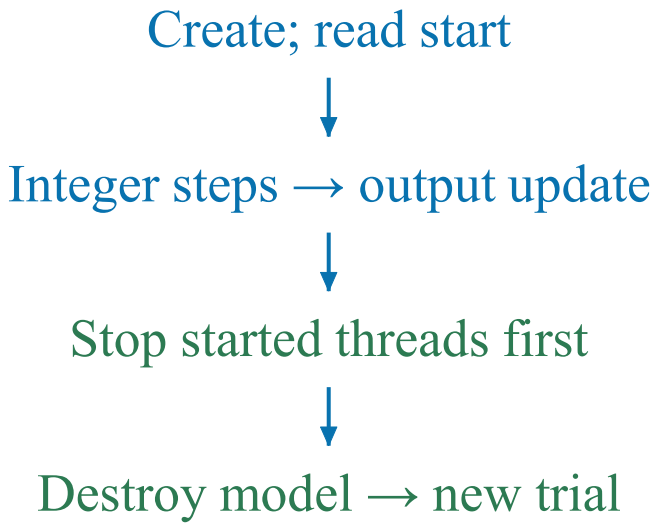
Route	Start and discover	Confirm start	Advance and record	Stop and clean up
Network host	Start CopterSim; discover target	InitTrueDataLoop; wait for runnedTime update	CopterSim advances; send input and record	EndTrueDataLoop closes socket
Direct loading	Create model; check exports	Read starting runnedTime and PosE	Integer Dllstep calls; Updata3Doutput	Stop started threads; DllDestroyModel

Match actual input, model time, and received response in a network experiment.

4.10.6 Lifecycle and Fault Isolation: Direct-Loading Route

Ending and Repeating a Direct-Loading Experiment

- After creation, check exports, actual start, and input boundary. Manually advance a known number of steps, update output, and save actual model time.
- Call StopAutUpdate first only if automatic updating was started. Then DllDestroyModel releases internal model resources.
- Clean up network and display objects created for this trial. Confirm initial state again next time; residual velocity is not a matched-condition repeat.



Stop started threads, destroy the model, and establish an explicit initial state for the next trial.

4.10.6 Lifecycle and Fault Isolation: Cleanup After Errors

Code: Destroying the Model After Manual Stepping

```
try:
    model.SendVelNED(0, 0, -1)
    for _ in range(1000):
        model.Dllstep()
    model.Updata3Doutput()
    print(model.out3Ddata.runnedTime)
    print(model.out3Ddata.PosE)
finally:
    model.DllDestroyModel()
```

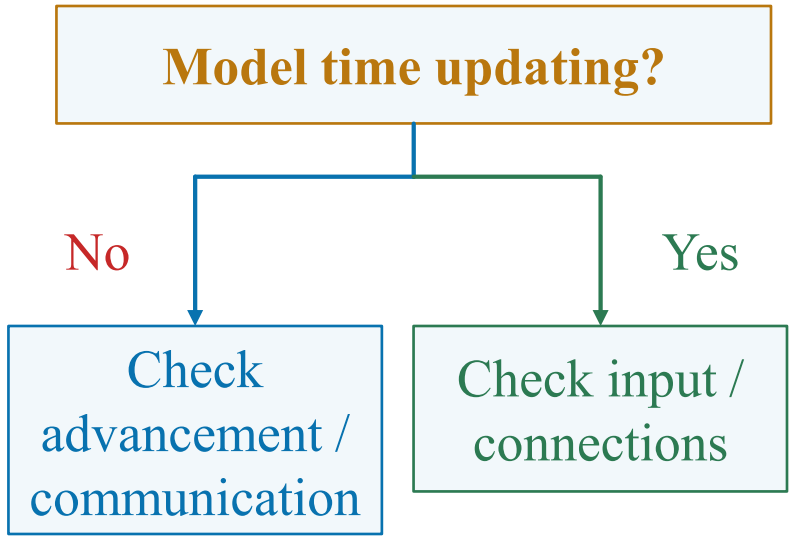
- **Prerequisite:** model comes from the actual-start readback code.
- **Ending:** finally runs on normal completion or error. This manual example has no automatic thread, so destroy the model directly.

- **Other routes:** StopAutUpdate before destruction if its thread was started. Creators clean up network and display objects separately. Read the actual initial state again next trial.

4.10.6 Lifecycle and Fault Isolation: Locating an Ineffective Input

Diagnosis When Sending Completes but Output Stays Unchanged

- **Unchanged time:** Check instance advancement, endpoints, and receive thread. Updating time without input arrival: Check ID, port, fields, and internal connections.
- **Wrong direction or fixed scale factor:** Check NED/FRD, degrees/radians, rpm/rad/s, and GPS integer scales. Convert once.
- If state seems constrained to a target, inspect the integrated controller boundary.



For direction and scale, check frame and units

Find the first missing evidence along endpoint, port reception, and internal connections.

4.10.6 Lifecycle and Fault Isolation: Interface Stage Experiment

Stage Experiment A3: A Complete Evidence Chain

- **Choose one A3 route:** Send a defined network extension field, or give an integrated DLL a single-axis velocity target. Predict first, then record actual input, time, and response.

- Keep model, parameters, and initial state fixed. Change one input across two trials. Submit a field table, short program, and shared-time-axis response curves.
- A3 is Chapter 4's third stage experiment. For the sequence, see Section 4.3, "Stage Experiment A1: Actuators and Force/Torque Synthesis." Distinguish source facts, predictions, documented examples, and local records.

Network route

First inSIL element → 3DType → RflySim3D appearance

Direct route

SendVelNED → 1000 steps → Read VelE and PosE

Change one input only

An explicit initial state and aligned input and response make an interface result interpretable.

Reference Answers

Stage Experiment A3: Predictions and Self-Check Criteria

- **Network display:** Setting the first inSILInts value to 5 displays a hexarotor in RflySim3D (example documentation). The current network parser does not update vehicleType; validate appearance in the scene.
- **Network state:** Increasing runnedTime only confirms instance advancement. Split trials after reset.
- **Logging route:** Growing CopterSim1.csv rows belong to outCopterData, not an inSIL success criterion.
- **Direct time:** For h=1 ms and 1000 steps, expect actual dt≈1000h=1 s. If low, check h and truncation.
- **Direct motion:** Expect VelE[2] to approach −1 m/s with lag; climb in 1 s is usually less than 1 m.
- **Report:** Attribute each conclusion to source code, documentation, prediction, or local records. Plot both trials on one model-time axis.

Section Summary

Section Summary: Method, Key Judgments, and Common Errors

- **Method:** Choose an entry point by task → check object, frame, units, and time → change one input → read new samples and log model time → clean up in reverse creation order.
- **Key judgments:** Sending success means data was sent. SLX connections define input meaning. Integrated model DLLs include a controller. A changed runnedTime marks a new sample.
- **Common errors:** Check these six points against your own code and records.
- **Indices:** The 60-element array and network packets have different offsets; keep their parsers separate.
- **Units:** Convert degrees/radians and rpm/rad/s once at the boundary.
- **Time:** Save simulation and wall-clock time separately; split after reset.
- **ID:** Network updates leave out3Ddata.CopterID at default 0.
- **Truncation:** ModelUpdate uses int(Δt/h) and discards remainders.
- **Cleanup:** Stop started threads before DllDestroyModel.