From Simulink to HiroSim: Translating Block-Diagram Models into Reproducible Configuration

Massimo Di Pierro

2026 — technical note

Abstract

Simulink is a widely used environment for modelling dynamical systems as graphical block diagrams. HiroSim is a modular, deterministic simulation stack in which scenarios are assembled from configured model instances that exchange named channels through a shared data store. This note develops a practical translation method for an important subset of Simulink models: continuous and hybrid dynamical systems that can be written as algebraic expressions, ordinary differential equations, and simple discrete logic. For this subset, the plant can be represented in HiroSim configuration without writing a custom plugin or a flight computer.

Two built-in model types provide the key mechanisms. The `expression` model implements algebraic and discrete-time signal-flow logic; because an output may read its previous committed value, it can also represent delays, accumulators, and simple fixed-step integrators. The `state_space` model declares $\dot{y} = f(y, t)$ using derivative expressions and delegates integration to a selected numerical integrator, including adaptive Dormand–Prince `rk45`. Five worked examples — the Van der Pol oscillator, a mass–spring–damper, a bouncing ball, a thermal house model, and a filling/draining tank — show the complete Simulink construction, the corresponding HiroSim configuration, and the resulting trajectories. The comparison highlights not only the mechanical mapping between the two representations, but also the engineering consequences of storing a scenario as structured text: reviewable diffs, reproducible runs, scriptable composition, schema validation, and a tractable interface for automated or AI-assisted model generation.

1 Scope and intent

Before anything else, three statements of scope, because they shape how every later section should be read.

1. **This is a subset, not a showcase.** HiroSim is a full closed-loop simulation stack — 6-DOF rigid-body dynamics, multibody gravity, contact and constraint solvers, aerodynamics, learned gray-box models, a compiled flight computer, telemetry, and a live 3-D dashboard, exercised in examples from a lunar-injection rocket to a 100-satellite constellation to a walking hexapod. *None of that appears here.* The five models in this note are the simplest textbook systems, chosen precisely because they are unremarkable. HiroSim also ships a large catalogue of built-in model types (Appendix A) — rigid bodies, contacts, constraints, aerodynamics, sensors and actuators, reaction wheels, and more — and lets you add your own in C, C++, Rust, or Nim as runtime-loaded plugins; *this note creates none of them*, building every example from just the two built-in JSON model types (`expression` and `state_space`) plus the core `rk45` integrator. Nothing below should be read as the boundary of what HiroSim can do; it is the boundary of what *this note* sets out to prove.

2. **The claim is a useful translation subset, not universal parity.** The note covers block diagrams built from algebraic operators, ordinary differential equations, sampled logic, and simple hybrid behavior. Within that scope, the translation follows a small set of repeatable rules. Simulink also contains capabilities not reproduced here — including specialized libraries, model-reference workflows, code-generation products, exact event handling, and rich graphical tooling — so the examples should be read as a migration method, not as a proof that every Simulink artifact has a one-to-one JSON equivalent.

3. **Simulators only, not flight computers.** HiroSim normally runs control logic on an FCSL flight computer (compiled from a restricted Python dialect). Including it would compare control *stacks*, not modelling *languages*. We therefore exclude FCSL entirely and keep every plant inside the simulator's own JSON.

2 Introduction

Simulink represents a dynamical system as a graph of *blocks* — gains, sums, products, integrators, sources, and sinks — wired by signal lines and solved by a MATLAB ODE solver. It is graphical, mature, and ubiquitous. Its primary `.slx` artifact is a packaged, tool-managed format rather than a human-authored text file; although it can be inspected and compared with specialized tooling, it is not naturally reviewed or merged as line-oriented source. Running and editing models also normally depends on a licensed MATLAB and Simulink environment.

HiroSim [1] is built around a different premise: a model is *JSON*. A HiroSim scenario is a JSON file listing model instances; each publishes named channels onto a shared in-memory data store and reads other models' channels back, and a deterministic scheduler advances every model on its own fixed rate. (JSON is the on-disk format the simulator reads; a scenario may equally be authored in YAML, TOML, or Jsonnet and converted to JSON, so "JSON" throughout should be read as "the configuration.") This note asks a narrow, practical question: *given a Simulink model, how much of it can you reproduce in HiroSim using JSON alone?* The answer is: a substantial and practically useful subset, with a direct correspondence for common mathematical and stateful blocks, using two built-in, configuration-driven model types — `expression` for fixed-step block diagrams and `state_space` for ODEs advanced by a real integrator — both interpreted at runtime, with no recompilation and no code generation. That a scenario is *composed* from models by JSON wiring, rather than written as code, is also what makes the stack a dependable target for AI-assisted engineering: authoring a scenario means producing structured, schema-checkable text rather than manipulating a proprietary graphical artifact. This does not make generated models automatically correct, but it makes changes easier to constrain, review, validate, test, and reproduce.

3 The two systems at a glance

	Simulink	**HiroSim**
Model artifact	tool-managed `.slx` package	`.json` (or generated from YAML/TOML/Jsonnet)
Composition	blocks + signal lines (graphical)	models + named channels on a shared store
Authoring	graphical editor	any text editor
Time / solver	one model-wide variable- or fixed-step ODE solver	per-model fixed rate; selectable integrators (`euler`, `rk4`, `rk45`, ...)
State	block and solver states	integrated states plus committed data-store channels
Sinks	Scope / To Workspace	every channel is telemetry (live plot / recorded)
Determinism	configuration-dependent	deterministic under a fixed configuration and seed

Table 1: Simulink and HiroSim at a glance. They overlap in their ability to represent signal flow and dynamical state, but expose that structure through different authoring and execution models.

4 HiroSim in one page

A HiroSim scenario is a JSON object with a handful of top-level keys (`duration`, `clock_mode`, `telemetry_port`, ...) and a `models` array. Each entry has a `type`, a `name`, and a `config`; the `name` is the prefix under which the model publishes its channels. Models communicate only through these named channels — the data store is the signal bus. Reads from scheduled expression models see committed data-store values. Because the store is double-buffered, a self-reference — and, depending on scheduling, a cross-model dependency — observes the previously committed value. This gives an explicit sampled-data semantics that must be considered when translating feedback loops.

For this note the only model type we need is the built-in `expression` model. Its `config` holds numeric `parameters` (named constants) and an `outputs` array; each output registers one `float64` channel and gives an `expr` string, evaluated and written every tick. These expressions — and the `state_space` derivatives introduced below — are *interpreted*, not compiled: each is parsed once into an expression tree at load time and then walked allocation-free every tick by a shared engine (`fcsl_expr`), with no code generation and no recompilation. Editing a model is editing JSON. (This is the opposite choice from HiroSim's flight computer, which *compiles* to native code — and is out of scope here.) The grammar (shared with the variable server and FCSL transition guards) provides `+ - * /`, comparisons, `and`/`or`/`not`, `if(cond, a, b)`, and the builtins `abs`, `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `pow(a,b)`, `min`, `max`, and `clamp(x,lo,hi)`. A name resolves first against `parameters`, then against any data-store channel — *including the output's own channel*, read at its previous-tick value. That last property is the whole trick.

Two JSON ways to author dynamics: `expression` and `state_space`

HiroSim provides two built-in, JSON-authored ways to turn those channels into dynamics, and this note uses both. The `expression` model just described is evaluated once per scheduled tick. A self-referential output such as `"x = x + v*dt"` therefore implements a sampled accumulator, numerically equivalent to forward Euler when `v` represents a derivative. It is useful for discrete logic, simple fixed-step models, and for explaining the block mapping in §5; it should not be confused with the semantics of every Simulink Integrator configuration.

The `state_space` model instead declares an ODE $\dot{y} = f(y, t)$ as JSON derivative expressions and hands its state to a real integrator (`rk4`, adaptive `rk45`, ...). Its `config` lists `states`, each with an `init` value and a `deriv` expression; a bare state name resolves to that state's value, `time_var` (default `t`) to elapsed seconds (so Step/Sine sources read like the simulation clock), and any other name to a parameter or a DataStore channel. Crucially, the integrator evaluates those derivative expressions *at each of its stage states and stage times*, so Dormand-Prince `rk45` performs genuine adaptive high-order integration of an arbitrary — linear or nonlinear, first- or second-order — ODE, still written entirely in JSON. It is conceptually analogous to selecting a Dormand–Prince variable-step solver such as `ode45` in Simulink, and it is what every worked example in §7 uses. A Simulink Integrator block maps to a `state_space state` whose `deriv` is its input, exactly as the fixed-step mapping of §5 maps it to an `expression` self-reference. The accuracy difference is real: on $\dot{x} = -10x$ at $\Delta t = 0.1$, forward Euler collapses to zero in one step while adaptive `rk45` returns $x(0.1) = e^{-1}$ to rounding by inserting shorter internal sub-steps.

Why the integrator is a separate model. In each example config the `state_space` model names an integrator (`"integrator": "rk45"`) that is *also* declared as its own `models` entry. That is not redundancy: the integrator is a shared service, not a property of one model. Many models can name the same integrator — it gathers their states into one vector and advances them together, so coupled bodies stay consistent — and different models can name *different* integrators running side by side (a stiff subsystem on adaptive `rk45`, a cheap one on fixed `rk4`, a pure algebraic model on none). It also explains where the timestep lives: the `state_space` model has no `dt` and is never scheduled — its state is advanced entirely by the integrator, which owns the single authoritative `dt`. A scheduled model that does tick on a fixed rate takes its `dt` from its own config or, failing that, the run's top-level `defaults.advance_dt`; there are no hidden per-model default rates. HiroSim is event-driven, waking each scheduled model at its own frequency and dispatching the ready ones across a thread pool, so a 1 kHz plant and a 50 Hz controller coexist in one run. In the trivial examples of this note there is exactly *one* integrated model, so its integrator has a single subscriber — but the two-entry pattern is the general one.

5 From blocks to expressions

Integrator = self-reference. Because an expression may read its own channel's previous value, `"x = x + v*dt"` is a forward-Euler integrator: it accumulates `v` into `x` once per tick — the direct analogue of a Simulink Integrator under a fixed-step solver, and the one-tick read delay is the same z^{-1} an explicit solver introduces. A second self-reference, `"v = v + a*dt"`, gives the second integrator.

Time source = self-count. The output `"t = t + dt"` counts elapsed time (channels start at 0, so after k ticks it holds $k\,\Delta t$). From that single clock every Simulink source follows: Step `if(t`

`>= t0, A, 0)`, Ramp `A*t`, Sine `A*sin(w*t)`. HiroSim has no dedicated signal-generator model, but the self-clock supplies one in a line of JSON.

Initial conditions = first-tick guard. Expression channels initialise to 0; a non-zero initial condition is injected on the first tick with `"x = if(t < 0.5*dt, x0, x + v*dt)"`.

Simulink block	**HiroSim** `expression`
Gain, Sum, Product, Abs, trig/math	operators and builtins inside `expr`
Switch, Relational, Logical	`if(c,a,b)`, `<= > ==`, `and/or/not`
Saturation	`clamp(x, lo, hi)`
Discrete-Time Integrator / Unit Delay	output reading its previous value: `x = x + v*dt`
Step / Ramp / Sine source	self-clock `t = t + dt`, then `if/*/sin`
Initial condition	`parameters` + `if(t < 0.5*dt, x0, ...)`
Relay (with hysteresis)	`if(x < lo, 1, if(x > hi, 0, self))`
Scope / To Workspace	every channel is telemetry

Table 2: Common block-to-expression mappings. Continuous Integrator and Transfer Function blocks are more naturally translated into `state_space` states and derivatives.

Honest limitations. The correspondence is not total. (i) A self-referential `expression` integrator is fixed-step forward Euler, less accurate than a variable-step solver such as `ode45` for stiff or event-driven systems. That is exactly why every worked example here integrates not with `expression` but with the `state_space` model on adaptive `rk45` (§4), which evaluates the derivatives at each solver stage; the `expression` self-reference is kept only for the block-to-integrator mapping of Table 2, not for the runs. (ii) Hybrid resets land within one `dt` of the event rather than on an exact zero-crossing. (iii) There is no dedicated lookup-table block; a 1-D table must be approximated by an `if`-chain or a polynomial. These differences matter for demanding hybrid, stiff, or safety-critical models; they are acceptable here only because the examples are explanatory and their numerical behavior is checked against the expected solution.

6 A practical migration workflow

A reliable port is easier when the Simulink diagram is treated as an executable specification rather than translated block by block without analysis. The following sequence worked for the examples in this note.

1. **Identify continuous state.** List every continuous Integrator, state-space block, transfer function realization, and physical state. Rewrite these as a first-order system $\dot{y} = f(y, t, u)$ and place them in one or more `state_space` models.

2. **Separate algebraic and sampled logic.** Gains, sums, nonlinear functions, relays, mode logic, and sampled controllers belong in derivative expressions or scheduled `expression` models. Make every intentional delay and sample rate explicit.

3. **Define interfaces by channel name.** Replace signal lines with stable, dimensional channel names. Record units and reference frames in the scenario documentation; the JSON schema can validate shape and type, but it cannot infer physical meaning.

4. **Choose numerics deliberately.** Select the integrator, output rate, tolerances, and any event approximation from the model's stiffness and hybrid behavior. Do not use a self-referential expression accumulator merely because it is syntactically convenient.

5. **Validate behavior, not screenshots.** Compare initial conditions, steady states, invariants, event timing, and quantitative error norms against the source model or an analytic solution. Plot similarity is necessary but not sufficient.

This decomposition also scales beyond the examples. Once a hand-written ODE is replaced by a built-in rigid body, actuator, sensor, contact, or learned model, the surrounding channels and validation workflow remain the same.

7 Worked examples

Each example below gives the governing equations, the *full* Simulink construction (a MATLAB script that builds and runs the block diagram), the *full* HiroSim JSON, and a plot from running it. Every HiroSim config is a `state_space` model (§4) that declares the ODE as JSON derivative expressions, plus an explicit `rk45` integrator instance that advances it — so each is integrated by genuine adaptive Dormand-Prince RK45, the analogue of selecting `ode45` in Simulink. From an unpacked HiroSim distribution — which ships the `sim-runner` binary and its models — each runs headless with

```
sim-runner --config <name>_rk45.json
```

7.1 Van der Pol oscillator

A second-order nonlinear oscillator with a stable limit cycle. Writing it as a first-order system in the state x and velocity v:

$$x'' - \mu(1 - x^2)\,x' + x = 0, \tag{1}$$

$$\dot{x} = v, \tag{2}$$

$$\dot{v} = \mu\,(1 - x^2)\,v - x, \tag{3}$$

with $\mu = 1$ and $x(0) = 2,\ v(0) = 0$.

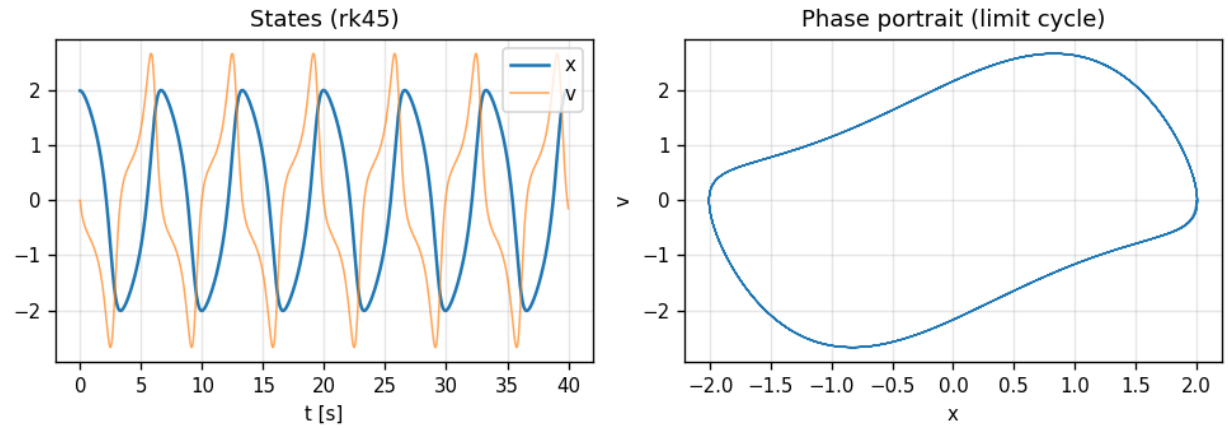

Figure 1: Simulation of `van_der_pol_rk45.json`: the state history and the phase portrait converging onto the limit cycle (amplitude ≈ 2), at $\Delta t = 0.05$.

Listing 1: Simulink model (MATLAB): `van_der_pol.m`.

```
% van_der_pol.m -- Simulink reference model for the Van der Pol oscillator.
%
% This is the *reference* implementation the HiroSim port (../van_der_pol.json)
% is compared against in the paper "Moving from Simulink to HiroSim".
%
% Governing ODE (mu = 1):
%     x'' - mu*(1 - x^2)*x' + x = 0
% As a first-order system with states x1 = x, x2 = x':
%     x1' = x2
%     x2' = mu*(1 - x1^2)*x2 - x1
% Initial conditions: x1(0) = 2, x2(0) = 0.
%
% Run in MATLAB with:  van_der_pol
% (Requires MATLAB + Simulink. MathWorks also ships this model built-in as `vdp`.)

mu = 1.0;
mdl = 'vdp_ref';
new_system(mdl);
open_system(mdl);

% --- Blocks -----------------------------------------------------------------
% Two integrators produce x2 (= x') and x1 (= x).
add_block('simulink/Continuous/Integrator', [mdl '/Int_x2'], ...
    'InitialCondition', '0', 'Position', [260 40 290 70]);   % -> x2 = x'
add_block('simulink/Continuous/Integrator', [mdl '/Int_x1'], ...
    'InitialCondition', '2', 'Position', [360 40 390 70]);   % -> x1 = x

% Nonlinear feedback term  mu*(1 - x1^2)*x2 .
add_block('simulink/Math Operations/Math Function', [mdl '/Sq'], ...
    'Function', 'square', 'Position', [360 140 390 170]);    % x1^2
add_block('simulink/Math Operations/Sum', [mdl '/OneMinusSq'], ...
    'Inputs', '+-', 'Position', [300 140 330 170]);          % 1 - x1^2
add_block('simulink/Sources/Constant', [mdl '/One'], ...
    'Value', '1', 'Position', [230 120 260 150]);
add_block('simulink/Math Operations/Product', [mdl '/Prod'], ...
    'Position', [230 200 260 230]);                          % (1-x1^2)*x2
add_block('simulink/Math Operations/Gain', [mdl '/Mu'], ...
    'Gain', num2str(mu), 'Position', [160 200 190 230]);     % mu*(...)

% x2' = mu*(1-x1^2)*x2 - x1 .
add_block('simulink/Math Operations/Sum', [mdl '/Accel'], ...
```

```
    'Inputs', '+-', 'Position', [120 40 150 70]);

add_block('simulink/Sinks/Scope', [mdl '/x'], 'Position', [440 40 470 70]);
add_block('simulink/Sinks/To Workspace', [mdl '/x_out'], ...
    'VariableName', 'x', 'SaveFormat', 'Structure With Time', ...
    'Position', [440 100 470 130]);

% --- Wiring ------------------------------------------------------------
add_line(mdl, 'Accel/1', 'Int_x2/1');
add_line(mdl, 'Int_x2/1', 'Int_x1/1');
add_line(mdl, 'Int_x1/1', 'x/1');
add_line(mdl, 'Int_x1/1', 'x_out/1');
add_line(mdl, 'Int_x1/1', 'Sq/1');
add_line(mdl, 'One/1', 'OneMinusSq/1');
add_line(mdl, 'Sq/1',  'OneMinusSq/2');
add_line(mdl, 'OneMinusSq/1', 'Prod/1');
add_line(mdl, 'Int_x2/1', 'Prod/2');
add_line(mdl, 'Prod/1', 'Mu/1');
add_line(mdl, 'Mu/1', 'Accel/1');
add_line(mdl, 'Int_x1/1', 'Accel/2');

% --- Solver + run ------------------------------------------------------
set_param(mdl, 'Solver', 'ode45', 'StopTime', '40');
out = sim(mdl);

figure; plot(x.time, x.signals.values);
xlabel('t [s]'); ylabel('x'); title('Van der Pol oscillator (Simulink, ode45)');
```

Listing 2: HiroSim config (`state_space` + `rk45`): `van_der_pol_rk45.json`.

```
{
  "duration": 40.0,
  "run_name": "van-der-pol-rk45",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
  "status_vars": ["vdp.x", "vdp.v"],
  "status_interval": 5.0,
  "plugins": ["plugins/libcore_plugin.so"],
  "models": [
    {
      "type": "state_space",
      "name": "vdp",
      "integrator": "rk45",
      "config": {
        "parameters": { "mu": 1.0 },
        "states": [
          { "name": "x", "init": 2.0, "deriv": "v" },
          { "name": "v", "init": 0.0, "deriv": "mu*(1.0 - x*x)*v - x" }
        ]
      }
    },
    {
      "type": "rk45",
      "name": "rk45",
      "depends_on": ["vdp"],
      "config": { "dt": 0.05, "atol": 1e-8, "rtol": 1e-6 }
    }
  ]
}
```

7.2 Bouncing ball (hybrid dynamics)

A ball falls under gravity and bounces at $z = 0$, losing energy ($g = 9.81$, $z(0) = 25$). A common Simulink construction uses a velocity integrator with reset logic and a position integrator constrained at the floor. The HiroSim example intentionally uses a different physical approximation: it replaces the discontinuous impact reset with a stiff penalty-contact ODE — a stiff *penalty contact* adds a spring-damper force while the ball penetrates the floor:

$$\dot{z} = v, \tag{4}$$

$$\dot{v} = -g + [\, z < 0 \,]\,(-k_c\, z - c_c\, v), \tag{5}$$

so the rebound and energy loss emerge from k_c and c_c, while adaptive `rk45` inserts shorter internal steps through contact. This is not an exact semantic translation of a reset block; it is an alternative continuous-contact model that produces comparable macroscopic behavior.

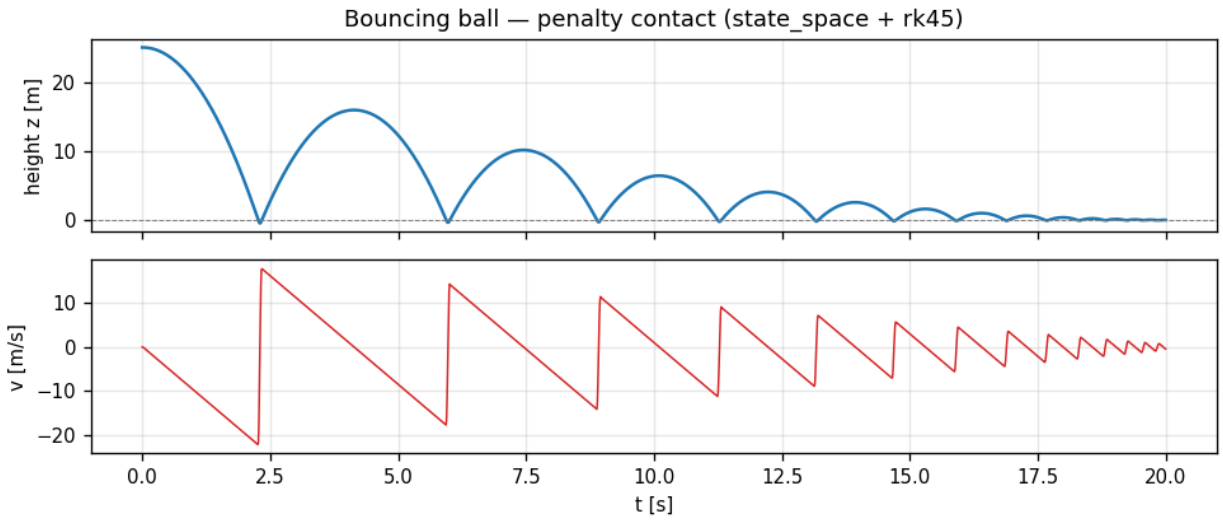

Figure 2: Simulation of `bouncing_ball_rk45.json`: the decaying sequence of bounces in height and velocity, resolved by adaptive `rk45` sub-stepping through the penalty contact.

Listing 3: Simulink model (MATLAB): `bouncing_ball.m`.

```
% bouncing_ball.m -- Simulink reference model for the bouncing ball.
%
% Reference implementation for the HiroSim port (../bouncing_ball.json), used in
% the paper "Moving from Simulink to HiroSim".
%
% Hybrid dynamics (a classic Simulink Zeno-behaviour demo, shipped as `sldemo_bounce`):
%   Continuous:   dv/dt = -g,   dx/dt = v
%   Reset at x=0: v+ = -kappa * v-      (coefficient of restitution kappa)
% Initial conditions: x(0) = 25 m, v(0) = 0 m/s.
%
% The velocity integrator has an external reset (rising edge when the ball hits
% the floor); the position integrator saturates at a lower limit of 0 and its
% "state limit hit" port triggers the velocity reset.
%
% Run in MATLAB with:  bouncing_ball

g = 9.81; kappa = 0.8; x0 = 25; v0 = 0;
mdl = 'bounce_ref';
```

```
new_system(mdl);
open_system(mdl);

% --- Blocks -----------------------------------------------------------------
add_block('simulink/Sources/Constant', [mdl '/negG'], ...
    'Value', num2str(-g), 'Position', [40 40 70 70]);

% Velocity integrator with external reset to -kappa*v at impact.
add_block('simulink/Continuous/Integrator', [mdl '/Int_v'], ...
    'InitialCondition', num2str(v0), ...
    'ExternalReset', 'rising', ...
    'InitialConditionSource', 'external', ...
    'Position', [160 40 190 90]);

% Position integrator, saturated at the floor (x >= 0). Its "hit lower limit"
% state-port event drives the velocity reset.
add_block('simulink/Continuous/Integrator', [mdl '/Int_x'], ...
    'InitialCondition', num2str(x0), ...
    'LimitOutput', 'on', 'LowerSaturationLimit', '0', 'UpperSaturationLimit', 'inf', ...
    'ShowSaturationPort', 'on', ...
    'Position', [280 40 310 90]);

% Reset value  v+ = -kappa * v-  (read v back through a gain).
add_block('simulink/Math Operations/Gain', [mdl '/negKappa'], ...
    'Gain', num2str(-kappa), 'Position', [160 140 190 170]);

add_block('simulink/Sinks/Scope', [mdl '/x'], 'Position', [380 40 410 70]);
add_block('simulink/Sinks/To Workspace', [mdl '/x_out'], ...
    'VariableName', 'x', 'SaveFormat', 'Structure With Time', ...
    'Position', [380 100 410 130]);

% --- Wiring -----------------------------------------------------------------
add_line(mdl, 'negG/1',  'Int_v/1');          % dv/dt = -g
add_line(mdl, 'Int_v/1', 'Int_x/1');          % dx/dt = v
add_line(mdl, 'Int_x/1', 'x/1');
add_line(mdl, 'Int_x/1', 'x_out/1');
add_line(mdl, 'Int_v/1', 'negKappa/1');       % read v-
add_line(mdl, 'negKappa/1', 'Int_v/3');       % external IC = -kappa*v- on reset
add_line(mdl, 'Int_x/2', 'Int_v/2');          % saturation event -> velocity reset

% --- Solver + run -----------------------------------------------------------
% Zero-crossing detection is what makes the reset land exactly at x = 0.
set_param(mdl, 'Solver', 'ode23', 'StopTime', '30', 'ZeroCrossControl', 'on');
out = sim(mdl);

figure; plot(x.time, x.signals.values);
xlabel('t [s]'); ylabel('height [m]'); title('Bouncing ball (Simulink, ode23 + zero-crossing)');
```

Listing 4: HiroSim config (`state_space` + `rk45`, penalty contact): `bouncing_ball_rk45.json`.

```
{
  "duration": 20.0,
  "run_name": "bouncing-ball-rk45",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
  "status_vars": ["ball.z", "ball.v"],
  "status_interval": 2.0,
  "plugins": ["plugins/libcore_plugin.so"],
  "models": [
    {
      "type": "state_space",
      "name": "ball",
      "integrator": "rk45",
      "config": {
        "parameters": { "g": 9.81, "kc": 2000.0, "cc": 6.3 },
        "states": [
```

```
          { "name": "z", "init": 25.0, "units": "m", "deriv": "v" },
          { "name": "v", "init": 0.0, "units": "m/s", "deriv": "0.0 - g + if(z < 0.0, 0.0 - kc*z - cc*v, 0.0)"
      }
        ]
      }
    },
    {
      "type": "rk45",
      "name": "rk45",
      "depends_on": ["ball"],
      "config": { "dt": 0.005, "dt_max": 0.005, "atol": 1e-7, "rtol": 1e-5 }
    }
  ]
}
```

7.3 Mass-spring-damper

The textbook linear second-order system, as a first-order system in x and v:

$$m\,x'' + c\,x' + k\,x = F(t), \tag{6}$$

$$\dot{x} = v, \tag{7}$$

$$\dot{v} = \frac{F - c\,v - k\,x}{m}, \tag{8}$$

with $m = 1$, $c = 0.5$, $k = 2$ (underdamped, $\zeta \approx 0.18$) and F a unit step at $t = 1\,\mathrm{s}$. The response overshoots and rings down to the static deflection $x_{ss} = F_0/k = 0.5\,\mathrm{m}$.

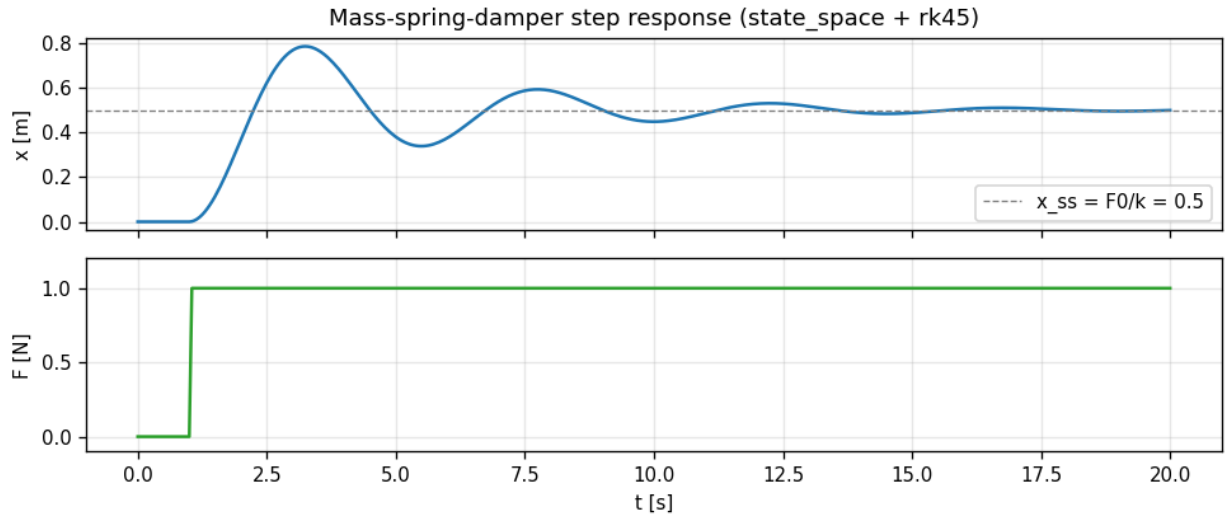

Figure 3: Simulation of `mass_spring_damper_rk45.json`: the underdamped step response settling at $x_{ss} = 0.5$, and the step forcing.

Listing 5: Simulink model (MATLAB): `mass_spring_damper.m`.

```
% mass_spring_damper.m -- Simulink reference model for a mass-spring-damper.
%
% Reference implementation for the HiroSim port (../mass_spring_damper.json),
% used in the paper "Moving from Simulink to HiroSim".
%
```

```
% Governing ODE:
%     m*x'' + c*x' + k*x = F(t)
% Rearranged for the acceleration that feeds the first integrator:
%     x'' = (F - c*x' - k*x) / m
% F(t) is a unit step applied at t = 1 s. Initial conditions x(0)=0, x'(0)=0.
% With m=1, c=0.5, k=2 the response is underdamped (zeta = 0.177) and settles
% at the static deflection x_ss = F0/k = 0.5 m.
%
% Run in MATLAB with:  mass_spring_damper

m = 1.0; c = 0.5; k = 2.0; F0 = 1.0; tstep = 1.0;
mdl = 'msd_ref';
new_system(mdl);
open_system(mdl);

% --- Blocks -----------------------------------------------------------
add_block('simulink/Sources/Step', [mdl '/F'], ...
    'Time', num2str(tstep), 'Before', '0', 'After', num2str(F0), ...
    'Position', [40 40 70 70]);

add_block('simulink/Math Operations/Sum', [mdl '/Sum'], ...
    'Inputs', '+--', 'Position', [130 40 160 90]);          % F - c*x' - k*x
add_block('simulink/Math Operations/Gain', [mdl '/InvM'], ...
    'Gain', num2str(1/m), 'Position', [200 50 230 80]);     % /m  -> x''

add_block('simulink/Continuous/Integrator', [mdl '/Int_v'], ...
    'InitialCondition', '0', 'Position', [280 50 310 80]);  % -> x'
add_block('simulink/Continuous/Integrator', [mdl '/Int_x'], ...
    'InitialCondition', '0', 'Position', [360 50 390 80]);  % -> x

add_block('simulink/Math Operations/Gain', [mdl '/DampC'], ...
    'Gain', num2str(c), 'Position', [280 140 310 170]);     % c*x'
add_block('simulink/Math Operations/Gain', [mdl '/SpringK'], ...
    'Gain', num2str(k), 'Position', [360 200 390 230]);     % k*x

add_block('simulink/Sinks/Scope', [mdl '/x'], 'Position', [440 50 470 80]);
add_block('simulink/Sinks/To Workspace', [mdl '/x_out'], ...
    'VariableName', 'x', 'SaveFormat', 'Structure With Time', ...
    'Position', [440 110 470 140]);

% --- Wiring -----------------------------------------------------------
add_line(mdl, 'F/1', 'Sum/1');
add_line(mdl, 'Sum/1', 'InvM/1');
add_line(mdl, 'InvM/1', 'Int_v/1');
add_line(mdl, 'Int_v/1', 'Int_x/1');
add_line(mdl, 'Int_x/1', 'x/1');
add_line(mdl, 'Int_x/1', 'x_out/1');
add_line(mdl, 'Int_v/1', 'DampC/1');       % feed back x'
add_line(mdl, 'DampC/1', 'Sum/2');
add_line(mdl, 'Int_x/1', 'SpringK/1');     % feed back x
add_line(mdl, 'SpringK/1', 'Sum/3');

% --- Solver + run -----------------------------------------------------
set_param(mdl, 'Solver', 'ode45', 'StopTime', '20');
out = sim(mdl);

figure; plot(x.time, x.signals.values);
xlabel('t [s]'); ylabel('x [m]'); title('Mass-spring-damper step response (Simulink, ode45)');
```

Listing 6: HiroSim config (`state_space` + `rk45`): `mass_spring_damper_rk45.json`.

```
{
  "duration": 20.0,
  "run_name": "mass-spring-damper-rk45",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
```

```
  "status_vars": ["msd.x", "msd.v"],
  "status_interval": 2.0,
  "plugins": ["plugins/libcore_plugin.so"],
  "models": [
    {
      "type": "state_space",
      "name": "msd",
      "integrator": "rk45",
      "config": {
        "parameters": { "m": 1.0, "c": 0.5, "k": 2.0, "F0": 1.0, "tstep": 1.0 },
        "states": [
          { "name": "x", "init": 0.0, "units": "m", "deriv": "v" },
          { "name": "v", "init": 0.0, "units": "m/s", "deriv": "(if(t >= tstep, F0, 0.0) - c*v - k*x) / m" }
        ]
      }
    },
    {
      "type": "rk45",
      "name": "rk45",
      "depends_on": ["msd"],
      "config": { "dt": 0.05, "atol": 1e-8, "rtol": 1e-6 }
    }
  ]
}
```

7.4 Thermal model of a house

A lumped first-order thermal model,

$$C\,\dot{T} = Q\,u - \frac{T - T_{\text{out}}}{R}, \tag{9}$$

with a thermostat relay $u \in \{0, 1\}$ (hysteresis ± 0.5°C about $T_{\text{set}} = 20$°C) and a sinusoidal outdoor temperature. It has no mechanical state — a *first-order* scalar ODE the mechanical models cannot represent, but which `state_space` states directly. The sinusoidal source is inlined in the derivative (a function of `t`, evaluated at each stage); the thermostat relay — whose memory inside the deadband is genuinely discrete — stays a small per-tick `expression` model whose `house.heat` channel appears in the plant's derivative.

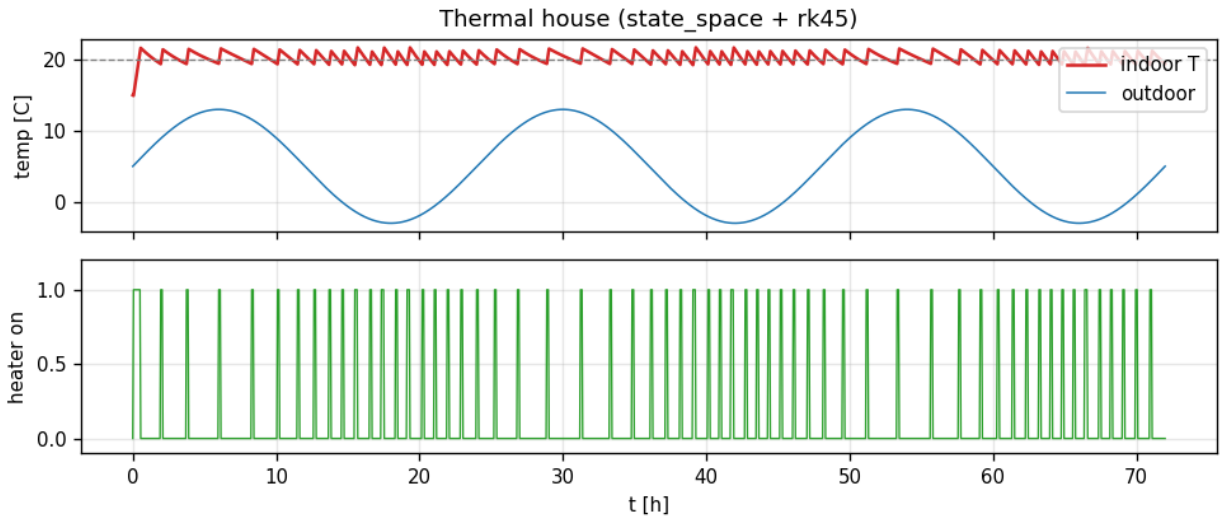

Figure 4: Simulation of `thermal_house_rk45.json`: indoor temperature holding the setpoint against the sinusoidal outdoor temperature, and the thermostat command.

Listing 7: Simulink model (MATLAB): `thermal_house.m`.

```
% thermal_house.m -- Simulink reference model for a lumped thermal house.
%
% Reference implementation for the HiroSim port (../thermal_house.json), used in
% the paper "Moving from Simulink to HiroSim". A single-capacitance simplification
% of the MathWorks `sldemo_househeat` example.
%
% First-order thermal ODE (one lumped indoor temperature T):
%     C*dT/dt = Q*u - (T - Tout)/R
% where:
%     u    = thermostat command (0/1) from a relay with hysteresis about Tset,
%     Q    = heater power, R = thermal resistance to outside, C = heat capacity,
%     Tout = outdoor temperature, a daily sinusoid Tmean + Tamp*sin(2*pi*t/period).
% Initial indoor temperature T(0) = 15 degC.
%
% (Time is in abstract "hours" mapped to sim-seconds so the daily cycle is
%  visible in a short run; period = 24.)
%
% Run in MATLAB with:  thermal_house

Tmean = 5; Tamp = 8; period = 24; Tset = 20; hyst = 0.5;
Q = 30; R = 4; C = 2; T0 = 15;
mdl = 'househeat_ref';
new_system(mdl);
open_system(mdl);

% --- Outdoor temperature (Sine source) ----------------------------------
add_block('simulink/Sources/Sine Wave', [mdl '/Tout'], ...
    'Amplitude', num2str(Tamp), 'Bias', num2str(Tmean), ...
    'Frequency', num2str(2*pi/period), 'Position', [40 140 70 170]);

% --- Thermostat: Relay with hysteresis ----------------------------------
add_block('simulink/Discontinuities/Relay', [mdl '/Thermostat'], ...
    'OnSwitchValue', num2str(Tset - hyst), 'OffSwitchValue', num2str(Tset + hyst), ...
    'OnOutputValue', '1', 'OffOutputValue', '0', 'Position', [130 30 160 60]);
add_block('simulink/Math Operations/Gain', [mdl '/HeatQ'], ...
    'Gain', num2str(Q), 'Position', [200 30 230 60]);       % Q*u

% --- Heat loss  (T - Tout)/R --------------------------------------------
```

```
add_block('simulink/Math Operations/Sum', [mdl '/dT'], ...
    'Inputs', '+-', 'Position', [200 140 230 170]);         % T - Tout
add_block('simulink/Math Operations/Gain', [mdl '/InvR'], ...
    'Gain', num2str(1/R), 'Position', [270 145 300 165]);   % /R

% --- Net heat -> dT/dt (Sum) -> /C -> Integrator -> T --------------------
add_block('simulink/Math Operations/Sum', [mdl '/Net'], ...
    'Inputs', '+-', 'Position', [330 60 360 110]);          % Q*u - loss
add_block('simulink/Math Operations/Gain', [mdl '/InvC'], ...
    'Gain', num2str(1/C), 'Position', [390 70 420 100]);    % /C
add_block('simulink/Continuous/Integrator', [mdl '/Int_T'], ...
    'InitialCondition', num2str(T0), 'Position', [450 70 480 100]);

add_block('simulink/Sinks/Scope', [mdl '/T'], 'Position', [540 70 570 100]);
add_block('simulink/Sinks/To Workspace', [mdl '/T_out'], ...
    'VariableName', 'T', 'SaveFormat', 'Structure With Time', ...
    'Position', [540 130 570 160]);

% --- Wiring --------------------------------------------------------------
add_line(mdl, 'Int_T/1', 'Thermostat/1');   % thermostat reads indoor T
add_line(mdl, 'Thermostat/1', 'HeatQ/1');
add_line(mdl, 'HeatQ/1', 'Net/1');
add_line(mdl, 'Int_T/1', 'dT/1');
add_line(mdl, 'Tout/1', 'dT/2');
add_line(mdl, 'dT/1', 'InvR/1');
add_line(mdl, 'InvR/1', 'Net/2');
add_line(mdl, 'Net/1', 'InvC/1');
add_line(mdl, 'InvC/1', 'Int_T/1');
add_line(mdl, 'Int_T/1', 'T/1');
add_line(mdl, 'Int_T/1', 'T_out/1');

% --- Solver + run --------------------------------------------------------
set_param(mdl, 'Solver', 'ode45', 'StopTime', '72');
out = sim(mdl);

figure; plot(T.time, T.signals.values);
xlabel('t [h]'); ylabel('indoor T [degC]'); title('Thermal house (Simulink, ode45)');
```

Listing 8: HiroSim config (`state_space` plant + per-tick `expression` relay + `rk45`): `thermal_house_rk45.json`.

```
{
  "duration": 72.0,
  "run_name": "thermal-house-rk45",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
  "status_vars": ["house.T", "house.heat"],
  "status_interval": 8.0,
  "plugins": ["plugins/libcore_plugin.so"],
  "models": [
    {
      "type": "expression",
      "name": "relay",
      "config": {
        "dt": 0.05,
        "parameters": { "Tset": 20.0, "hyst": 0.5 },
        "outputs": [
          { "name": "house.heat", "expr": "if(house.T < Tset - hyst, 1.0, if(house.T > Tset + hyst, 0.0, house
    .heat))" }
        ]
      }
    },
    {
      "type": "state_space",
      "name": "house",
```

```
      "integrator": "rk45",
      "config": {
        "parameters": { "PI": 3.141592653589793, "Tmean": 5.0, "Tamp": 8.0, "period": 24.0, "Q": 30.0, "R":
    4.0, "C": 2.0 },
        "states": [
          { "name": "T", "init": 15.0, "units": "degC",
            "deriv": "(Q*house.heat - (T - (Tmean + Tamp*sin(2.0*PI*t/period)))/R) / C" }
        ]
      }
    },
    {
      "type": "rk45",
      "name": "rk45",
      "depends_on": ["house", "relay"],
      "config": { "dt": 0.05, "atol": 1e-8, "rtol": 1e-6 }
    }
  ]
}
```

7.5 Tank fill and empty

Another *first-order* system — nonlinear level dynamics with a Torricelli outflow:

$$A\,\dot{h} = q_{\text{in}} - k\sqrt{h}, \tag{10}$$

with q_{in} a step-down (on until $t = 30\,\text{s}$). The single derivative expression uses the `pow` builtin and clamps the level non-negative; while inflow is on, h rises toward $h^* = 4\,\text{m}$ where $k\sqrt{h^*} = q_{\text{in}}$, then drains.

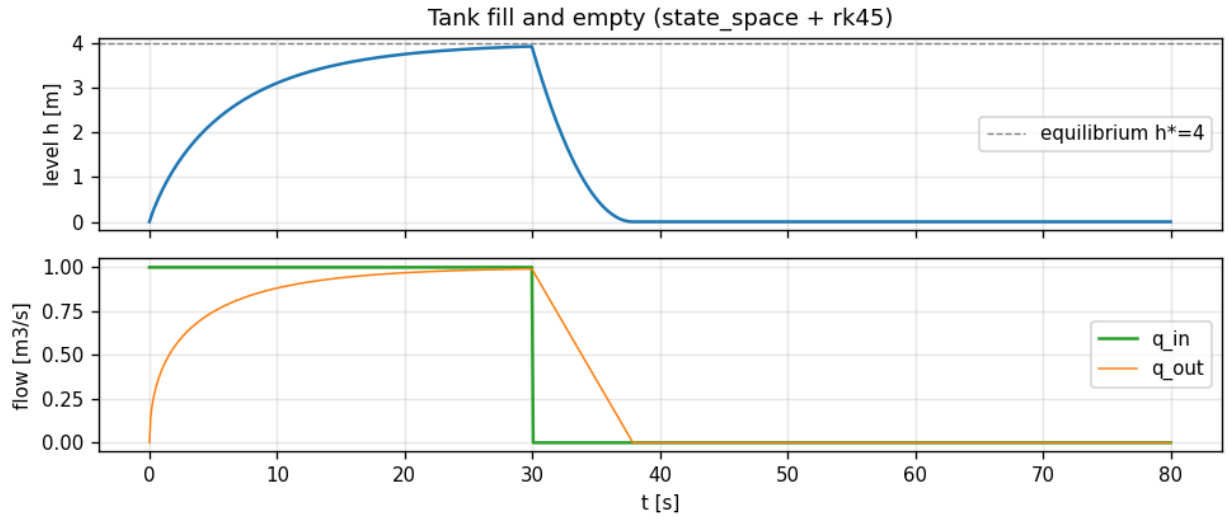

Figure 5: Simulation of `tank_level_rk45.json`: the level filling toward its equilibrium and draining after the inflow is cut, with the two flows.

Listing 9: Simulink model (MATLAB): `tank_level.m`.

```
% tank_level.m -- Simulink reference model for a filling/draining tank.
%
% Reference implementation for the HiroSim port (../tank_level.json), used in
% the paper "Moving from Simulink to HiroSim".
```

```
%
% Nonlinear first-order level dynamics (Torricelli outflow):
%     A*dh/dt = q_in - k*sqrt(h)
% i.e.  dh/dt = (q_in - k*sqrt(h)) / A , with h clamped at >= 0.
% Inflow q_in is a constant Qin until t = 30 s, then shut off (a Step down), so
% the tank fills toward its equilibrium h* (where k*sqrt(h*) = Qin, here h*=4 m)
% and then drains. Initial level h(0) = 0.
%
% Run in MATLAB with:  tank_level

A = 1.0; k = 0.5; Qin = 1.0; tfill = 30; h0 = 0;
mdl = 'tank_ref';
new_system(mdl);
open_system(mdl);

% --- Inflow: on then off (Step down at t=tfill) -------------------------
add_block('simulink/Sources/Step', [mdl '/Qin'], ...
    'Time', num2str(tfill), 'Before', num2str(Qin), 'After', '0', ...
    'Position', [40 40 70 70]);

% --- Outflow  k*sqrt(h) -------------------------------------------------
add_block('simulink/Math Operations/Sqrt', [mdl '/Sqrt'], ...
    'Operator', 'sqrt', 'Position', [280 140 310 170]);
add_block('simulink/Math Operations/Gain', [mdl '/Kout'], ...
    'Gain', num2str(k), 'Position', [220 140 250 170]);     % k*sqrt(h)

% --- Net flow -> /A -> Integrator (level, saturated >= 0) ---------------
add_block('simulink/Math Operations/Sum', [mdl '/Net'], ...
    'Inputs', '+-', 'Position', [130 40 160 90]);           % q_in - q_out
add_block('simulink/Math Operations/Gain', [mdl '/InvA'], ...
    'Gain', num2str(1/A), 'Position', [200 50 230 80]);     % /A
add_block('simulink/Continuous/Integrator', [mdl '/Int_h'], ...
    'InitialCondition', num2str(h0), ...
    'LimitOutput', 'on', 'LowerSaturationLimit', '0', 'UpperSaturationLimit', 'inf', ...
    'Position', [280 50 310 80]);                           % -> h

add_block('simulink/Sinks/Scope', [mdl '/h'], 'Position', [380 50 410 80]);
add_block('simulink/Sinks/To Workspace', [mdl '/h_out'], ...
    'VariableName', 'h', 'SaveFormat', 'Structure With Time', ...
    'Position', [380 110 410 140]);

% --- Wiring -------------------------------------------------------------
add_line(mdl, 'Qin/1', 'Net/1');
add_line(mdl, 'Net/1', 'InvA/1');
add_line(mdl, 'InvA/1', 'Int_h/1');
add_line(mdl, 'Int_h/1', 'h/1');
add_line(mdl, 'Int_h/1', 'h_out/1');
add_line(mdl, 'Int_h/1', 'Sqrt/1');   % outflow depends on current level
add_line(mdl, 'Sqrt/1', 'Kout/1');
add_line(mdl, 'Kout/1', 'Net/2');

% --- Solver + run -------------------------------------------------------
set_param(mdl, 'Solver', 'ode45', 'StopTime', '80');
out = sim(mdl);

figure; plot(h.time, h.signals.values);
xlabel('t [s]'); ylabel('level [m]'); title('Tank fill and empty (Simulink, ode45)');
```

Listing 10: HiroSim config (first-order `state_space` + `rk45`): `tank_level_rk45.json`.

```
{
  "duration": 80.0,
  "run_name": "tank-level-rk45",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
  "status_vars": ["tank.h"],
```

```
  "status_interval": 8.0,
  "plugins": ["plugins/libcore_plugin.so"],
  "models": [
    {
      "type": "state_space",
      "name": "tank",
      "integrator": "rk45",
      "config": {
        "parameters": { "A": 1.0, "kout": 0.5, "Qin": 1.0, "tfill": 30.0 },
        "states": [
          { "name": "h", "init": 0.0, "units": "m",
            "deriv": "(if(t < tfill, Qin, 0.0) - kout*pow(max(h, 0.0), 0.5)) / A" }
        ]
      }
    },
    {
      "type": "rk45",
      "name": "rk45",
      "depends_on": ["tank"],
      "config": { "dt": 0.1, "atol": 1e-8, "rtol": 1e-6 }
    }
  ]
}
```

8 Stepping it up with built-in models

The five examples above all use the `state_space` model because it maps so cleanly onto a Simulink block diagram — but that is the *teaching* choice, not how a realistic HiroSim scenario is usually built. In practice you rarely hand-write an ODE as `state_space` derivatives. You reach instead for the richer built-in models — rigid bodies with real mass and inertia, springs and joints, contact surfaces, aerodynamics, and sensors and actuators with noise and delay (Appendix A) — or you write your own in C, C++, Rust, or Nim, and you *compose* them by JSON wiring. The dynamics then emerge from the models rather than from an equation you typed. They need not even be hand-derived: HiroSim ships *learned*, trainable gray-box models — `ml_engine` and `ml_aero`, differentiable Wiener networks (stable IIR filters plus a small neural network) fit offline to data and run inline as force/torque providers — so unsteady engine thrust or aerodynamic loads can come from a *trained* model instead of an equation, while staying physically well-behaved.

As a small taste, here is a bouncing ball rebuilt from stock parts: a `rigid_body` ball with real mass and inertia, hanging from a damped `spring` tethered to a fixed anchor, bobbing under `gravity`, with a body-mounted *noisy* `imu` that measures its specific-force acceleration — all advanced by the `rk4` integrator. No ODE is written anywhere; the ball, the spring, gravity, and the sensor are separate models on the shared bus, wired by name.

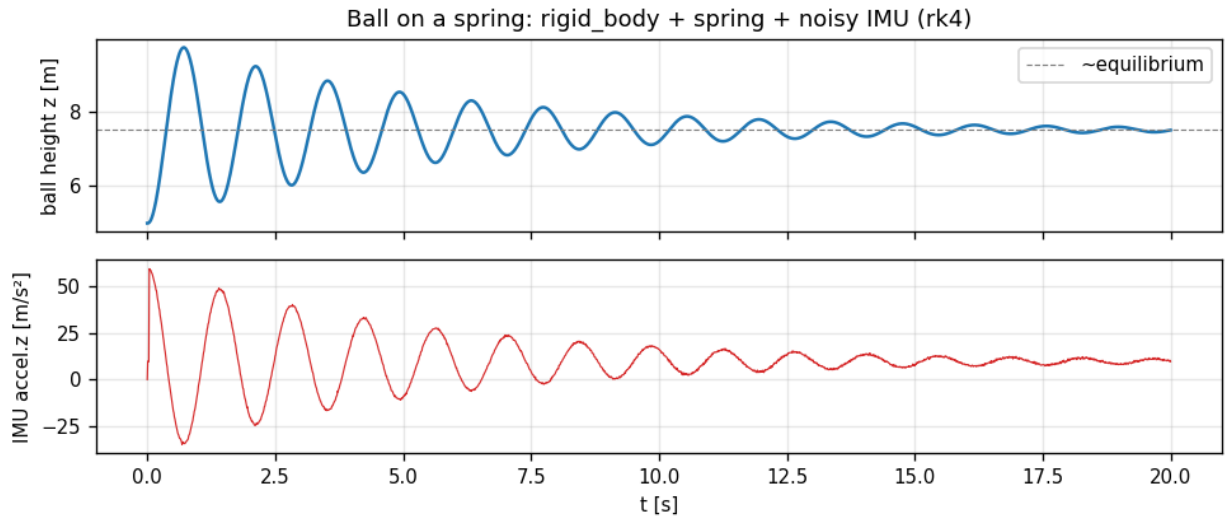

Figure 6: Simulation of `spring_ball_imu.json`: the `rigid_body` ball bobs on the `spring` and settles toward equilibrium (top); the body-mounted `imu` reports the specific-force acceleration with sensor noise, decaying to the $\approx 9.8\,\mathrm{m/s^2}$ gravity reaction at rest (bottom).

Listing 11: A composite from built-in models: `rigid_body` + `spring` + `gravity` + noisy `imu` + `rk4`: `spring_ball_imu.json`.

```
{
  "duration": 20.0,
  "run_name": "spring-ball-imu",
  "clock_mode": "fast",
  "paused_at_start": false,
  "telemetry_port": 9000,
  "status_vars": ["ball.pos.z", "ball.vel.z", "imu.accel.z"],
  "status_interval": 2.0,
  "plugins": ["plugins/libcore_plugin.so", "plugins/libexamples_plugin.so"],
  "models": [
    {
      "type": "gravity", "name": "gravity",
      "config": { "refresh_dt": 1.0, "gx": 0.0, "gy": 0.0, "gz": -9.81 }
    },
    {
      "type": "rigid_body", "name": "anchor", "integrator": "rk4",
      "config": { "mass": 1000000.0, "x": 0.0, "y": 0.0, "z": 10.0, "dt": 0.001 }
    },
    {
      "type": "rigid_body", "name": "ball", "integrator": "rk4",
      "config": {
        "mass": 1.0, "ixx": 0.1, "iyy": 0.1, "izz": 0.1,
        "x": 0.0, "y": 0.0, "z": 5.0,
        "gravity_service": "gravity.force_at", "dt": 0.001
      }
    },
    {
      "type": "spring", "name": "spring", "depends_on": ["anchor", "ball"],
      "config": { "k": 20.0, "rest_length": 2.0, "damping": 0.4, "ball_a": "anchor", "ball_b": "ball", "dt":
     0.001 }
    },
    {
      "type": "imu", "name": "imu", "order": 50,
      "config": {
        "dt": 0.01,
```

```
        "position": "ball.pos", "attitude": "ball.quat", "angular_rate": "ball.omega",
        "frame": "body",
        "accel": { "gaussian_noise": { "stddev": 0.3 } },
        "gyro": { "gaussian_noise": { "stddev": 0.005 } },
        "seed": 42
      }
    },
    {
      "type": "rk4", "name": "rk4", "depends_on": ["spring", "imu"],
      "config": { "dt": 0.001 }
    }
  ]
}
```

Two things carry over from here to the simple `state_space` examples — they are the same kind of object to the simulator. First, *everything is telemetry*: every channel each model publishes — `ball.pos.z`, `imu.accel.z`, the spring force, the attitude quaternion — is recorded and observable exactly as in §9, whether the model is a two-line `state_space` ODE or a 6-DOF rigid body. Second, a scenario is *commandable*: add a `variable_server` model and you can pause, read, and retune any of it live — change the spring's stiffness `k`, perturb the ball's state — over the same Python client. Nothing about the workflow changes as the models get more capable; only the models do.

9 Observing a run: the dashboard and the Python client

HiroSim ships a configurable browser *dashboard* (the telemetry visualizer) for watching a simulation: it renders live telemetry in real time — 2-D plots, 3-D trajectories, gauges, and log panels, arranged by a JSON dashboard config — and replays recorded runs the same way. Figure 7 shows it replaying the ball-on-a-spring run of §8: four plot widgets and a sim-control panel, driven by a few lines of dashboard JSON, with a transport bar to scrub the recording.

Figure 7: The HiroSim dashboard replaying the `spring_ball_imu` run: the ball's height and velocity, the noisy accelerometer, and the noisy gyro, each a `plot` widget over a telemetry channel, plus a pause/resume command panel and the playback scrubber. The whole layout is a small JSON file (`examples/visualizer-configs/spring_ball_imu.json`).

For the figures in this note we go the other way and drive everything from the HiroSim Python client, `simclient`. From a Python session — interactive, a script, or a notebook — it starts the simulator, connects to the running sim to read and command it over the variable server, pulls the recorded telemetry back as `numpy` arrays, and plots with matplotlib. The workflow, end to end:

Listing 12: Start a run, command it, read telemetry as `numpy`, and plot — with `simclient`.

```
import subprocess, asyncio
import numpy as np, matplotlib.pyplot as plt
from simclient import Live, Reader

# 1. START A RUN. The receiver records every channel to output/data/<id>/*.npy;
#    sim_runner runs the scenario.
recv = subprocess.Popen(["telemetry_receiver", "--port", "9000", "--id", "vdp",
                         "--data-dir", "output/data", "--idle-timeout", "2"])
sim  = subprocess.Popen(["sim_runner", "--config",
                         "papers/simulink-to-hirosim/examples/van-der-pol/van_der_pol_rk45.json",
                         "--set", "telemetry_port=9000"])

# 2. CONNECT + COMMAND the running sim over its variable server (add a
```

```
#    `variable_server` model to the config to enable this; it listens on 7002).
async def command():
    live = Live(command_host="127.0.0.1", command_port=7002)
    await live.pause()                    # freeze the clock
    x_now = await live.get("vdp.x")       # read any channel's live value
    await live.set("vdp.x", value=1.0)    # ... or write it (retune / perturb)
    await live.resume()                   # let it run on
    return x_now.value
asyncio.run(command())                    # in a notebook, just `await command()`

sim.wait(); recv.wait()                   # deterministic + finite: it ends on its own

# 3. READ TELEMETRY back as numpy arrays.
r = Reader("output/data", "vdp")
x = r.numeric("vdp.x")["values"]          # -> numpy.ndarray
v = r.numeric("vdp.v")["values"]
t = np.linspace(0.0, 40.0, len(x))        # state_space has no clock channel

# 4. PLOT.
fig, ax = plt.subplots()
ax.plot(t, x, label="x"); ax.plot(t, v, label="v")
ax.set_xlabel("t [s]"); ax.legend()
```

With the same executable, configuration, initial conditions, seed, and runtime settings, the run is reproducible. Bit-for-bit identity can additionally depend on platform and floating-point implementation details.

10 Discussion

What porting to HiroSim buys. The model becomes structured text: it lives naturally in version control, can be reviewed line by line, and can be generated or transformed by ordinary tooling. Language models can assist with such edits, but generated scenarios still require schema checks, dimensional review, numerical tests, and engineering validation. Runs are deterministic and reproducible from a seed. There is no license gate on opening or running a model. And the same simulator that runs these tutorials runs full closed-loop vehicle scenarios, so a toy model and a flight scenario share one engine.

What is genuinely different. Simulink is graphical; a wiring diagram communicates topology at a glance in a way a JSON array does not. On the numerics, Simulink's variable-step solvers and exact zero-crossing location have a counterpart here: the worked examples integrate with adaptive `rk45` via the `state_space` model, whose derivative expressions the solver evaluates at every stage. The `expression` model's fixed-step forward Euler is the simpler, plugin-free option — and the most direct way to *read* a Simulink block diagram, since an Integrator becomes a self-reference — which is why the mapping of §5 is built on it; `state_space` is the upgrade when accuracy matters. Both are just JSON.

What FCSL adds — and why these examples don't need it. Everything in this note is a *plant*: a system of equations, with any controller written inline as part of the model. That is what a Simulink model of a dynamical system usually is, and it is all these examples need. HiroSim's next tier up is the *flight computer*, authored in **FCSL** — a restricted Python dialect that compiles to native code. FCSL is where real control and autonomy live: sensor filtering and state estimation (e.g. Kalman / complementary filters), guidance and trajectory logic, PID and state-feedback control laws, mode/fault state machines. Its point is portability: the *same* compiled flight-computer binary runs against the simulator and, through a hardware-abstraction

layer, against real sensors and actuators — so a controller is validated in simulation and then carried onto a vehicle without being rewritten (software- to hardware-in-the-loop). None of that is needed to *reproduce a Simulink block diagram*: a plant needs no flight computer, so we left FCSL out. When you do want closed-loop control that will fly — an autopilot, a landing sequence, a robot's motion planner — FCSL is where it goes.

Reading this note correctly. As stated in Section 1, the five examples are the floor of the demonstration, not the ceiling of the tool. They establish parity on the simplest systems and hand you the translation rules; scaling those rules up — to 6-DOF dynamics, contact, aerodynamics, learned models, and compiled flight software — is what the rest of HiroSim is for, and is the subject of the HiroSim white paper [1].

For a Simulink user. If you use Simulink for equation-based plant models and sampled control logic, much of that work can be represented in HiroSim — the same integrators, sources, gains, summing junctions, and feedback loops — in a *simpler, text-based syntax* that a person, or a language model, can read, diff, and edit. And you can do considerably more without leaving it: a deep catalogue of ready-made physics models, custom models in C, C++, Rust, or Nim, and *learned* gray-box models fit from data — for the systems a block diagram cannot express (§8, Appendix A); telemetry on *every* channel, observed live in a configurable browser dashboard or pulled into `numpy` and commanded from a Python client (§9); and a compiled flight computer (FCSL) for control software that moves from simulation onto real hardware unchanged. The five toy examples are the on-ramp; the road keeps going.

11 Conclusion

A Simulink model and its HiroSim counterpart are two encodings of the same object: a signal-flow graph of stateful arithmetic operators. Once an Integrator is understood as a state whose derivative is its input — an `expression` self-reference for fixed-step work, or a `state_space deriv` handed to `rk45` for adaptive accuracy — a source as a function of a self-incrementing clock, and a Scope as an always-on telemetry channel, the translation is mechanical, and it lands in JSON. The five worked examples, shown in both systems and reproducible from their configuration, demonstrate a narrower and more useful claim: common block-diagram plants can be translated into HiroSim by separating algebraic and discrete logic from continuously integrated state. The resulting model is not always structurally identical to its Simulink source, but its assumptions, interfaces, solver choice, and runtime behavior become explicit in reviewable configuration.

Appendix A HiroSim model catalogue

For reference, the model types HiroSim ships today, each with a one-line description. Which are available in a given run depends on the plugins the config loads (the `core` and `examples` plugins, plus a few built into the runner); the worked examples in this note use only `state_space`, `expression`, and the core `rk45` integrator.

Integrators and core physics (`core` plugin).

- `euler`, `rk2`, `rk4`, `verlet`, `ab4`, `rk45` — fixed- and adaptive-step numerical integrators that advance any model's registered state.
- `gravity` — uniform or central-body gravity service.
- `frame` — ECI/ECEF/geodetic (WGS-84) coordinate-transform service.
- `aero` — atmospheric drag force on a body.
- `rigid_body` — 3-D rigid body (quaternion attitude), with optional geodetic launch initial conditions.
- `ground_contact` — penalty contact surface (plane/sphere/WGS-84): spring-damper normal plus optional Coulomb friction.
- `tire_friction` — rolling-contact wheel friction coupling spin to ground reaction under one friction circle.
- `distance_constraint`, `ball_joint`, `hinge_joint`, `weld_joint` — rigid-body constraints.
- `rigid_constraint_solver` — PGS solver consumed by the constraints above.
- `foot_lock` — pins a foot's contact point on touchdown, releases it on liftoff.
- `mass_properties` — drives a target rigid body's mass, centre of mass, and inertia from a tracked channel (e.g. a depleting propellant level).
- `vehicle_state` — aggregates the pose, velocity, and mass of a set of rigid bodies into vehicle-level channels (a multi-body assembly read as one vehicle).
- `proximity_stop` — ends the run when a vehicle reaches, or travels past, a target body.
- `tcp_command` — TCP JSON command endpoint.
- `zenoh_bus` — Zenoh pub/sub bridge (built with `-d:zenoh`).

Vehicles, robots, and I/O (`examples` plugin).

- `ball`, `spring` — point mass and Hooke spring.
- `sensor`, `gps`, `imu`, `actuator`, `thruster` — hardware-shaped I/O models (noise, bias, transport delay, saturation, slew).
- `tutorial_ball` — wall-bouncing demo body.
- `cart_pendulum` — reduced-coordinate cart-pole.
- `robot_arm` — 6-DOF serial manipulator on a mobile base (reduced coordinates).
- `joint_motor` — torque actuator across a `hinge_joint` for maximal-coordinate robots.
- `gravity_compensator` — per-joint gravity-hold feed-forward for an articulated robot.
- `computed_torque` — full inverse-dynamics feed-forward for an articulated robot.

- `conveyor` — kinematic belt carrying parts along at constant speed.
- `kinematic_gripper` — kinematic grasp carrying a workpiece with the end-effector.
- `part_box` — depleting grid of supply parts a feeder arm picks from.
- `reaction_wheel`, `solar_panel`, `laser_link` — momentum-wheel attitude actuator; solar-power capture; laser power downlink.
- `kinematic_arm`, `debris_collector` — forward-kinematics render of a 2-link arm; debris capture/stow manager.
- `lifting_wing` — wing with lift, drag, and stabilising moments.

The `ml` plugin adds `ml_engine` and `ml_aero` — *learned*, trainable gray-box models (a differentiable biquad-Wiener core: stable IIR filters plus a small MLP), fit offline to data and run inline as force/torque providers (engine thrust / mass-flows / shaft speed, and aerodynamic coefficients).

Built-in and configuration-level (always available).

- `expression` — JSON `fcsl_expr` output channels, self-advancing (fixed-step forward Euler); no recompile.
- `state_space` — JSON ODE $\dot{y} = f(y, t)$ integrated by `rk4`/`rk45`/…, derivatives evaluated at each solver stage.
- `variable_server` — runtime get/set/eval command server on the DataStore.
- `composite` — config-loader directive that inlines another JSON file at this position.

Beyond these, individual applications register their own model types: an example application ships them in its own plugin (the lander demo, for instance, provides a variable-mass vehicle and a native descent controller), and every FCSL flight-computer program becomes a model type in the `fcsl` plugin (the type name is the program's filename stem). Both are application-specific and out of this note's scope.

References

[1] M. Di Pierro, *HiroSim: A Modular, Deterministic Simulation Stack for Closed-Loop Control-Software Development*, technical white paper, 2026.