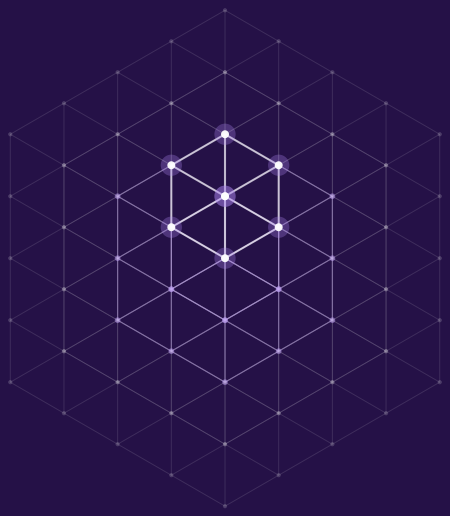
The Ballistics Lab Handbook

Interactive Exterior Ballistics with Lattice



Alex Jokela

Lattice v0.6.4 + engine v0.37.0

The Ballistics Lab Handbook
Interactive Exterior Ballistics with Lattice

Covers the Ballistics Lab on Lattice v0.6.4
and ballistics-engine v0.37.0.
Part IV additionally notes v0.30.1 where the two differ.

First edition, 2026.

Contents

Part I

The Laboratory

Chapter 1

What the Lab Is

1.1 A question, and the shape of it

You are prone behind a 6.5 Creedmoor at a mountain range. The board says 1,200 meters of altitude, twelve degrees, 880 hectopascals. You have dialed for 600 yards, and the shot went low. The flag at 400 is showing more than it was.

The questions that follow are never single questions. They arrive as a family:

- What is my drop at 600 with this air?
- If I re-zero at 200 instead of 100, what happens to everything else?
- That flag is eight miles per hour, not four. What is the hold now?
- What if the muzzle velocity I chronographed last spring is thirty feet per second optimistic?

Every one of those questions shares almost all of its inputs with the others. The bullet does not change. The rifle does not change. The station pressure does not change between the second question and the third. What changes is one term, and what you want is the effect of that one term on everything downrange.

That is the shape of the work: **a mostly-fixed context, perturbed**. It is the reason this book is about a REPL and not about a calculator.

The design note that started this project states it in one line: *the session is the rifle-plus-environment; commands are perturbations.* Hold onto that sentence. It is the entire architectural argument, and everything in Part I is an elaboration of it.

Vocabulary we will use from here on

Exterior ballistics is the flight of the projectile between the muzzle and the target: gravity, drag, wind, and rotation of the earth. Interior ballistics (what happens in the barrel) and terminal ballistics (what happens in the target) are different subjects.
Drag model — a published reference drag curve, indexed by Mach number, that stands in for a family of bullet shapes. The engine ships G1, G2, G5, G6, G7, G8, GI, GS and RA4; the JSON protocol the Lab speaks accepts `G1`, `G6`, `G7` and `G8`.
Ballistic coefficient (BC) — a dimensionless number saying how much less your bullet slows down than the reference bullet of the chosen drag model. A BC is meaningless without naming its model: 0.243 G7 and 0.243 G1 are entirely different bullets.
Zero — the distance at which the sight line and the trajectory are made to agree. Every drop number in this book is relative to some zero.
Dope — the table of sight corrections you actually dial or hold, by distance. "Data On Previous Engagements" if you like acronyms; a range card if you do not.
Mil — a milliradian, exactly 0.001 radian. At 100 yards, one mil subtends 3.6 inches. **MOA** — a minute of angle, 1/60 of a degree, about 1.047 inches at 100 yards; the engine's printed tables deliberately use the round constant 3438 rather than the exact 3437.7467.
Density altitude (DA) — the ISA altitude at which the air would have the density you are actually shooting in. One number that folds pressure, temperature and humidity together.

1.2 Three near-misses

Before the Lab existed there were three tools on this machine that each got part of the way there. None of them has the shape described above, and the reasons are instructive — they are the requirements list, written backwards.

1.2.1 Near-miss one: the engine's own command line

The `ballistics-engine` binary is a serious piece of work: thirty subcommands, a two-hundred-flag `trajectory` verb, nine drag tables, an ICAO atmosphere to 84 km. Everything the Lab can compute, the CLI can compute.

It is also stateless by construction. Here is the 140 gr load from the opening paragraph, asked once:

Listing 1.1: One question, fully re-specified

```
$ ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
      --drag-model g7 --sight-height 1.8 --twist-rate 8 \
      --auto-zero 100 --max-range 1000 \
      --wind-speed 8 --wind-direction 270
+========================================+
|         TRAJECTORY RESULTS             |
+========================================+
| Max Range:           532.24 yd         |
| Max Height:            1.72 yd         |
| Zero Angle:          0.0678°           |
| Time of Flight:       0.688 s          |
| Impact Velocity:    1990.09 fps        |
| Impact Energy:      1230.95 ft-lb      |
+========================================+
| Stability (SG):        1.85            |
| Status:            STABLE              |
+========================================+

Bullet struck ground at 532 yd
```

Read the fourth line again. We asked for 1,000 yards and got 532. Ground-impact detection is on by default, the default bore height is 60 inches, and the bullet reached the dirt before it reached the range we asked for. Nothing lied to us — the last line says so — but the number that looks like an answer is not the answer to the question we thought we asked.

The default that truncates

`--ignore-ground-impact` defeats ground termination on the engine CLI. Without it, `--max-range 1000` is a request, not a promise. The Lab surfaces the same physics through a field named `termination`, which reads `max_range` or `ground_threshold` on every single run — you never have to remember to look.

Add the flag and ask a second question — the wind picked up, eight to fifteen:

Listing 1.2: The second question is the first question retyped

```
$ ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 \
      --drag-model g7 --sight-height 1.8 --twist-rate 8 \
      --auto-zero 100 --max-range 1000 --ignore-ground-impact \
      --wind-speed 15 --wind-direction 270
+========================================+
|         TRAJECTORY RESULTS             |
+========================================+
| Max Range:          1000.00 yd         |
| Max Height:            1.72 yd         |
| Zero Angle:           0.0678°          |
| Time of Flight:        1.513 s          |
| Impact Velocity:    1450.89 fps        |
| Impact Energy:       654.28 ft-lb      |
+========================================+
| Stability (SG):         1.85            |
| Status:             STABLE              |
+========================================+
```

Two things went wrong, and neither is a bug.

First, twelve flags were retyped to change one of them. That is not a keystroke complaint; it is a correctness complaint. Every retype is an opportunity to drop `--sight-height` and silently model a different rifle.

Second — and this is the sharper one — the answer to the question we asked is *not in the output*. We changed the wind. The summary block does not report windage at all. Run at 8 mph, run at 15 mph, and the two boxes differ in the sixth significant figure of impact velocity. To see the thing we changed we would need `--full`, a table, and our own eyes doing the subtraction.

There is a third, quieter problem. Neither of those commands mentions altitude, temperature or pressure, so both ran at the ICAO sea-level standard: 59 °F, 29.92 inHg, 50 % relative humidity. The mountain range from the opening paragraph is at 1,200 m and 880 hPa. The CLI has no memory of where you are standing. It will happily answer the wrong question, correctly, forever.

None of this is a criticism of the engine. A stateless command is exactly the right shape for a build script, a batch job, or the thirty other subcommands that answer a self-contained question — recoil energy, power factor, gyroscopic stability, a drag curve dump. The engine is not trying to be a laboratory. It is trying to be a solver you can call from anything, and it succeeds so thoroughly that the Lab calls it from Lattice.

1.2.2 The engine's own answer: saved profiles

The engine does have persistent storage. Its `profile save` subcommand writes one JSON file per load under the directory `~/.ballistics/profiles`, and most subcommands will read it back:

Listing 1.3: Stored loads

```
$ ballistics profile list
Saved Profiles:
+--------------------+--------+-------+--------+----------+----------+
| Name               | Vel    | BC    | Mass   | Diameter | Drag     |
+--------------------+--------+-------+--------+----------+----------+
| LABBOOK            |   2700 | 0.243 |  175.0 |    0.308 | G7       |
| SUBDEMO            |   1050 | 0.270 |  220.0 |    0.308 | G7       |
+--------------------+--------+-------+--------+----------+----------+
```

This removes the retyping, and it is genuinely useful. It does not turn the CLI into a session. A profile is a *load*, stored on disk, that a later command may consult. There is no history, no notion of a previous run, nothing to compare against, and no provenance attached to the answer you get back. And it carries its own trap:

Listing 1.4: Two flags named `--profile`, with different types

```
$ ballistics trajectory --profile LABBOOK
error: the following required arguments were not provided:
  --profile-row <NAME>

Usage: ballistics trajectory --profile-row <NAME> --profile <FILE>

$ ballistics trajectory --saved-profile LABBOOK
Loaded saved profile 'LABBOOK'
Effective values: velocity=2700.0 (trued=2700.0), BC=0.243 (trued=0.2430)
```

On `trajectory`, `--profile` means a CSV file and `--saved-profile` means a stored name; on every other subcommand `--profile` means the stored name. Part IV covers the whole profile system properly. The point here is narrower: storage is not state, and a named load is not a session.

1.2.3 Near-miss two: the browser demo terminal

The engine has also been compiled to WebAssembly and wrapped in a terminal-looking page, so that a curious visitor can type a command and watch a trajectory appear. The design note that preceded

the Lab describes its mechanism precisely: the page calls a `runCommand(string)` entry point once per line, with no session between calls.

That is a demo, and an effective one. It is not a laboratory. Each line is an independent process-equivalent; there is nothing to perturb because there is nothing that persists. It is the CLI's statelessness with a nicer font.

An honest note on this example

The demo page is described here as the design document described it. It is not present in the engine checkout this book was verified against, so unlike every other claim in these chapters, this one is reported at second hand and is labelled as such.

1.2.4 Near-miss three: the general LATTICE REPL

LATTICE ships its own interactive shell. It has exactly the property the first two lack — state that survives between lines:

Listing 1.5: `clat`, the general language shell

```
$ ./clat
Lattice v0.6.4 — crystallization-based programming language
Copyright (c) 2026 Alex Jokela. BSD 3-Clause License.
Type expressions to evaluate. Ctrl-D to exit.

let bc = 0.315
bc * 2
=> 0.63
projectile("x", 1, 2, 3, "G7")
error: undefined variable 'projectile'
```

`bc` survived from one line to the next. That is the right substrate. What it does not have is any of the domain: no `projectile`, no unit that knows a grain from a kilogram, no engine, no dope table, no rendering. It is a general language shell, and a general language shell is a starting point rather than a destination.

1.2.5 The requirements, read off the three failures

- **State between lines** — from the `clat` REPL.
- **A ballistics vocabulary** — which `clat` lacks.
- **A real solver** — which only the engine has.

- **Provenance on every answer** — so that "which engine, which solver, which assumptions" is never a memory exercise.
- **Termination and defaults surfaced, not buried** — so that the 532-yard surprise announces itself.

The Lattice Ballistics Lab is what you get when you satisfy all five.

1.3 What the Lab is

The Ballistics Lab is an interactive laboratory written in ordinary Lattice. Thirty-six source files, 14,038 lines, split evenly between eighteen library and application modules and eighteen test modules. It adds no intrinsic to the language, no native extension, and no C ABI dependency. If you can build `clat`, you can run the Lab; there is nothing else to compile.

It also does not compute ballistics. Not one line of trajectory integration lives in Lattice. The Lab's job is everything *around* the integration. It obtains the integration by running `ballistics-engine` as a child process — one process per solve — speaking a strict JSON protocol called **solve-json v1**.

Listing 1.6: The entire engine contract

```
      request JSON on stdin
                |
                v
  ballistics solve-json          <-- fixed literal argv, no shell
                |
                v
 exactly one JSON envelope on stdout
```

That is the whole transport. A Lattice `Experiment` value is converted into a request `Map`, serialized, and written to the child's standard input. The child writes exactly one envelope back. The Lab validates every field of that envelope before it constructs a single domain value, and only then does the session state advance.

The spawn goes through Lattice's `exec_argv` builtin with the literal argument vector `["solve-json"]`. No user value ever reaches argv, and no shell is involved, so an experiment name containing `$(rm -rf ~)` is a string and stays a string. Part V proves that empirically, with a test that tries to make the Lab create a file through a crafted argument and then asserts the file was not created.

1.3.1 Everything else is LATTICE

Around that one narrow seam, the Lab supplies:

- **Units** — eight nominally distinct SI-backed quantity types, so a muzzle velocity cannot be passed where a wind speed is required.
- **A domain** — validated, immutable `Projectile`, `Rifle`, `Atmosphere`, `Wind`, `Shot`, `SolverConfig` and `Experiment` values, with relational checks between them.
- **A protocol encoder** — which reads only canonical SI fields, so display metadata cannot leak onto the wire.
- **A response decoder** — which rejects anything the protocol did not promise.
- **Analysis** — checked interpolation, bounded resampling, and three programmable table builders: dope, wind card, comparison.
- **Rendering** — deterministic terminal strings, in metric or imperial, that never touch I/O.
- **A session** — the mutable state, behind exactly one reference, with transactional updates.
- **Persistence and artifacts** — strict versioned JSON documents for experiments and for verified runs.
- **A command surface** — eleven colon commands, parsed as inert data.

1.4 What the Lab is not

Five statements worth making early, because each one saves a wrong expectation.

It is not a solver. No trajectory arithmetic happens in LATTICE. If the engine is absent, the Lab starts, shows you the experiment, parses your commands, and fails at `:solve` with a structured error. We will see exactly that failure in Chapter 2.

It is not a second configuration language. There is no `:set` command. Changing the rifle, the load, the air or the wind is done by typing ordinary LATTICE. The README states this as a design thesis rather than an omission: experiment changes "remain visible ordinary Lattice calls rather than a second `:set` language." The colon commands can solve, query, render, save, load and reset — they cannot mutate the experiment.

It is not a sandbox. `exec_argv` runs the configured executable with the LATTICE process's own permissions and environment. The Lab guarantees that *your data* never becomes *code*; it does not guarantee anything about an engine binary you pointed it at.

It is not networked. No module in the Lab performs network access. The engine child is the only process it ever spawns. An air-gapped machine runs the Lab exactly as a connected one does.

It is not in the browser. The process backend needs real process execution and real terminal input, and WebAssembly builds of LATTICE have neither.

1.5 The architecture that was proposed, and the one that shipped

The design note that opens this project proposes something different from what you are about to use. It argues for a native LATTICE extension written in Rust — a `ballistics.dylib` installed under `~/.lattice/ext/` — statically linking the engine crate and binding its Rust API directly, with a LATTICE prelude on top supplying the session and the command vocabulary.

That extension was not built

Everything in this book describes the **subprocess** architecture: the Lab runs `ballistics solve-json` as a child process and speaks solve-json v1 over stdin and stdout. There is no dynamic library, no `lat_ext_init`, and no linked engine. If you have read the design note, read it as history — an alternative that was considered carefully and not taken.

The proposal is worth a page anyway, because its reasoning explains the shape of what shipped.

The design note's central technical finding is what it calls the *C-FFI coverage cliff*. The engine exports exactly eight `extern "C"` symbols. Those cover a single trajectory, a zero angle, a Monte Carlo run and a version string — roughly a quarter of what a serious laboratory needs. Everything requiring a variable-length array is missing from the flat C struct vocabulary entirely: wind segments, atmosphere segments, velocity-keyed BC segments, custom drag tables, powder-temperature curves. Binding the C ABI means proposing new engine exports the first week you want segmented wind.

Binding the Rust API from a Rust extension avoids that cliff, which is why the note chose it. The cost is a per-platform native build product, a dynamic library that must be kept in step with both the language runtime and the engine crate, and a data boundary that force-copies every value across it.

The subprocess route pays a different price and buys a different thing. The price is process spawn cost — measured, on the machine this book was written on, at roughly a fortieth of a second for a whole launch-solve-quit cycle, of which the engine's own solve is about three milliseconds. The purchase is that **the seam is a documented contract rather than an ABI**. solve-json v1 is explicit SI, versioned, closed-schema, and reproducible: the response carries a `resolved_request` in which every default is materialized as a concrete value. Two engine builds four minor versions apart produce byte-identical Lab output on the same experiment — we will run that experiment in Chapter 2 and diff it.

A contract you can print is a different kind of object from a symbol table you must match. That is the trade, made deliberately.

1.6 Why LATTICE, for a shooter who does not care

You can use the Lab without learning LATTICE, and this book will not force you to. But one language feature shows through the user interface often enough to be worth a page, because it explains why the Lab behaves the way it does when something goes wrong.

Phase: , , and

LATTICE tracks, in the type system, whether a value may still change.
A **flux** binding is mutable — you can assign to it. A **fix** binding is not. `freeze(value)` performs a one-way transition: the value becomes a **crystal**, deeply immutable, and stays that way. There is a matching `thaw()` that produces a mutable copy, and `clone()` for an independent one.
The distinction that matters here is between a value that is merely *declared* unchanging and one that has been *crystallized*. The second is enforced all the way down.

The Lab uses this to draw a line between draft data and finished data.

Every domain value you build — a projectile, a rifle, an atmosphere, a complete experiment — is frozen by its constructor. Once you hold an `Experiment`, nothing can quietly reach in and change the bullet weight. Every trajectory the engine returns is frozen before you see it. Every analysis table is built mutable and then frozen exactly once, at the top, so that a table is either complete or it does not exist.

The one deliberately mutable thing is the session itself, and it lives behind a single `Ref` — a reference cell. LATTICE passes values by value, so without that cell a function that "updated the session" would update a copy and throw it away. With it, every mutation is one write to one place, performed only after the complete next state has been built. That is why a failed solve leaves your experiment exactly as it was, and why a failed load does not leave you with half a rifle.

The design note went further and imagined the phase system carrying the draft-versus-verified distinction all the way to the range: a dope card stays mutable while you are truing it, and a `freeze` marks it as verified, with the language refusing to print field dope from an unverified draft. The shipped Lab implements the data half of that idea — there is a `VerifiedRun` document, and it authenticates itself by checking that it is a crystal — and leaves the refusal to the user. Part V takes that apart.

1.7 Five invariants you will meet everywhere

These recur in every part of the Lab. You do not need them to use it, and you will recognize them everywhere once you have read them once.

1. **SI at the wire, display metadata at the edge.** Every quantity carries a canonical SI value *and* a display value with its unit. Only the SI value crosses the process boundary; only the

display metadata is persisted. A ballistic mil is exactly one milliradian, everywhere, with no rounding.

2. **Failure is a value, not a throw — at module seams.** A failed solve, a bad command, a missing file: each comes back as a `LaboratoryError` carrying a code, a message, and a JSON path. The Lab throws internally, during validation; the seam modules catch and normalize. Nothing bounces you out of the session.

3. **Construct-then-publish.** Session mutation builds a complete next state and publishes it with exactly one write. A failed solve, a failed load, a failed validation leaves the prior state untouched — not partially updated.

4. **Freeze the graph once, at the top.** Children stay mutable during construction; a single `freeze()` takes ownership of the finished value. Where the Lab deliberately does *not* freeze, the source says why in a comment.

5. **Fail closed.** Anything the protocol did not promise — a banner on stdout, trailing JSON, an unknown field, a non-finite number, an exit code that disagrees with the envelope's status — becomes `invalid_engine_response`. There is no lenient path.

1.8 What a session actually looks like

Enough architecture. Here is the Lab answering the opening paragraph's questions, with the setup elided; Chapter 3 builds it keystroke by keystroke.

Listing 1.7: One solve, two queries, one perturbation (run summaries elided)

```
:solve
:at 1000 yd
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +50.43 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
sessions.replace_winds(lab, [wind(mph(15), deg(270))])
:solve
:at 1000 yd
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.13 ft-lb
  drop    : +300.82 in
  windage : +92.45 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

One line changed the wind. Nothing else was retyped. The bullet, the rifle, the 1,200 m of altitude, the 880 hPa, the 44 degrees of latitude with Coriolis enabled — all of it persisted, and the answer to the question we asked is on the screen: 50.43 inches of drift becomes 92.45 inches.

Note also what did *not* change. The drop is +300.82 inches in both runs, to the hundredth. A pure beam wind at these ranges moves the bullet sideways and leaves the vertical alone, and the Lab shows you that rather than making you infer it. Notice, too, that the drift did not scale linearly: eight to fifteen miles per hour is a factor of 1.875, and 50.43 inches times 1.875 would be 94.6, not 92.45. The trajectory itself shifted slightly, so the lag time changed. That two-inch discrepancy is physics, not rounding, and it is exactly the kind of thing a perturbation-shaped tool makes visible.

Three sign conventions to know before you read any number

The Lab prints a legend under everything it renders, and the legend means what it says. There are three conventions and they do not all point the same way.
Drop is positive below. `drop : +300.82 in` is 300.82 inches *below* the line of sight. Read drop as depth, not as height.
Windage is positive right. A bullet pushed to the shooter's right reports a positive windage; the wind above comes from 270°, the shooter's left, so it does.
Come-up is positive up. In a dope table, a positive come-up is elevation you dial *into* the sight. Chapter 3 watches all three of these behave on a real trajectory, and Part III works out the datum that drop is measured from.

1.9 How to read this book

This handbook has two readers and does not want to condescend to either.

If you came for the ballistics, Parts I through IV are yours. Chapter 2 gets the Lab running and verifies it. Chapter 3 walks a complete first session. Part II works the session surface in depth — solving, sampling, dope, wind cards, comparison, units, persistence. Part III is the physics the engine actually models, and how to ask for it. Part IV is the engine underneath, for when you want to drive it directly or read its JSON.

If you came for the Lattice, Part V is a case study of a 14,038-line application: how the domain is modelled, how the process backend validates a hostile response, how rendering stays deterministic, how the whole thing is tested. It is not an appendix. Nothing there requires you to know ballistics, and the ballistics chapters do not require you to know Lattice.

Every transcript in this book was produced by running the Lab. Every engine command shown was executed. Where two code paths disagree, or a default is doing something you did not ask for, the book says so and shows the evidence. A ballistics book that tidies its output is a ballistics book you cannot trust with a 1,000-yard shot.

Chapter 2

Installing and Launching

2.1 Two build products, one script

The Lab needs exactly two executables, and it ships as neither of them.

1. `clat` — the LATTICE runtime, built from the LATTICE checkout. This is the program that will actually run the Lab.

2. `ballistics` — the engine, built from the `ballistics-engine` Rust crate. This is the program that will actually solve trajectories.

The Lab itself is source. It lives under `examples/ballistics_lab/` in the LATTICE repository: thirty-six `.lat` files, no build step, nothing compiled, nothing installed. When you launch it you are running LATTICE source directly.

Why the Lab is not packaged

Distribution from the examples tree is the current mechanism. The LATTICE package registry serves single-file source packages, and the Lab is eighteen interdependent modules with root-relative import paths; multi-file registry packaging is planned but not shipped. Until it lands, a source checkout is the supported installation, and this chapter describes that path.

2.2 Building the engine

The engine is a Rust crate. From a checkout beside your LATTICE checkout:

Listing 2.1: Building the engine

```
$ cd ../ballistics-engine
$ cargo build --release
```

That produces `target/release/ballistics`. You can leave it there — the launcher knows to look in a sibling checkout — or install it somewhere on your PATH.

Confirm two things before moving on. First, that the binary runs and reports a version:

Listing 2.2: Engine identity

```
$ ./target/release/ballistics --version
ballistics 0.37.0
```

Ask the binary you just built, by path, rather than the bare name `ballistics`. A machine that has been building this engine for a while usually has an older copy installed somewhere on PATH, and the bare name will find that one instead — which is the whole subject of Section 2.11.

Second — and this is the check the launcher itself performs — that the binary provides the `solve-json` subcommand, and that its help text identifies the explicit-SI v1 interface:

Listing 2.3: The capability probe

```
$ ./target/release/ballistics solve-json --help
Solve one explicit-SI v1 JSON request from stdin and write one JSON envelope to stdout

Usage: ballistics solve-json [OPTIONS]

Options:
  -u, --units <UNITS>
          Unit system for input/output (metric or imperial)
```

An engine that does not print the phrase *explicit-SI v1 JSON request* is not usable by the Lab. It might be an older `ballistics`, or an unrelated program with the same name; either way the launcher will skip it and say so.

The units flag does nothing here

`solve-json` accepts `-u`/`--units` because every subcommand does, and then ignores it. The VI protocol is always SI in both directions. Passing `-u metric` returns a byte-identical summary.

2.3 Building `clat`

From the LATTICE checkout root:

Listing 2.4: Building the runtime

```
$ make all
$ ./clat --version
Lattice v0.6.4
```

That is the whole build. The Lab requires no extension, no `--prelude` flag, no dynamic library, and no configuration file. If `make all` succeeds and `./clat --version` answers, the LATTICE half of the installation is finished.

Version pairing

This book was verified against LATTICE **v0.6.4** and engine **0.37.0**. The machine it was written on also keeps an older **0.32.0** installed on PATH, which is why the sessions in Parts I–III name the 0.37.0 build explicitly instead of trusting discovery to find it. Where a chapter shows an engine version in output, it is the version the run actually reported — see Section 2.11 for why two different engine versions appear in this book and what changes between them.

2.4 The launcher

You could start the Lab by hand. You should not, and Section 2.9 shows why. The supported entry point is a 240-line POSIX shell script in the LATTICE checkout:

Listing 2.5: Starting the Lab

```
$ sh scripts/ballistics-lab.sh
ballistics-lab: backend=stack-vm, engine=/Users/alexjokela/.local/bin/ballistics
Ballistics laboratory — Lattice v0.6.4
Engine: /Users/alexjokela/.local/bin/ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

ballistics>
```

The script does four things and then gets out of the way:

1. Resolves a `clat` executable.
2. Resolves a `ballistics` executable, probing each candidate for solve-json v1 support.
3. Changes directory to the LATTICE checkout root, because the Lab's import paths are root-relative.
4. Replaces itself with `clat examples/ballistics_lab/ballistics-repl.lat <engine>`, via `exec`, so the process you end up talking to is `clat` itself.

That first line goes to standard error

`ballistics-lab: backend=..., engine=...` is written to *stderr*, not stdout. It will not appear in a redirected capture of the session, and it is the fastest way to confirm which engine you are actually talking to. Every transcript in this book was captured with stdout only, so that line appears in this chapter and nowhere else.

Listing 2.6: The launcher's own help

```
$ sh scripts/ballistics-lab.sh --help
Usage: ballistics-lab [OPTIONS]

Launch the self-hosted Lattice Ballistics Lab.

Options:
  --backend NAME   Select stack-vm (default) or regvm
  --stack-vm       Use the default stack VM
  --regvm          Use the register VM
  --clat PATH      Use this clat executable
  --engine PATH    Use this ballistics executable
  -h, --help       Show this help

Environment:
  BALLISTICS_LAB_BACKEND  Backend name; defaults to stack-vm
  CLAT                    clat executable name or path
  BALLISTICS_ENGINE       ballistics executable name or path
  BALLISTICS_ENGINE_VERSION
                          Optional exact engine version required by the lab

Discovery is deterministic. Explicit options override environment variables.
Otherwise the launcher checks the checkout, PATH, Cargo target directories,
and a sibling ballistics-engine checkout.
```

2.5 Deterministic engine discovery

That last paragraph deserves expansion, because "which engine am I actually running?" is the question behind most confusing ballistics output.

For `clat`, the order is:

1. `--clat PATH`
2. the `CLAT` environment variable
3. `./clat` in the checkout root
4. `clat` on `PATH`

For the engine, an explicit override is a hard requirement — if you name an engine and it is missing, non-executable, or fails the capability probe, the launcher exits rather than quietly falling back to something else:

1. `--engine PATH`
2. the `BALLISTICS_ENGINE` environment variable

With no override, the launcher searches exactly five places, in order, and takes the first candidate that passes the solve-json v1 probe:

1. `ballistics` on `PATH`;
2. `$CARGO_TARGET_DIR/release/ballistics`, then `$CARGO_TARGET_DIR/debug/ballistics`;
3. the sibling checkout `../ballistics-engine/target/release/ballistics`, then its `debug` equivalent, relative to the LATTICE checkout;
4. `target/release/ballistics`, then `target/debug/ballistics`, under *the caller's* working directory;
5. `$HOME/.cargo/bin/ballistics`.

Nothing else on the filesystem is searched. Here is the fall-through, live. On this machine `ballistics` lives in `~/.local/bin`, so step one normally wins — and wins with the stale 0.32.0 copy. Remove that from `PATH` and step three takes over, reaching the sibling checkout and the newer engine this book is written against:

Listing 2.7: Discovery falling through to the sibling checkout

```
$ env PATH=/usr/bin:/bin sh scripts/ballistics-lab.sh
ballistics-lab: backend=stack-vm,
     engine=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Ballistics laboratory — Lattice v0.6.4
Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

:solve
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
```

Discovery is convenience, not policy

If it matters which engine solved a number — and in a ballistics notebook it always matters — do not rely on the search order. Name the engine with `--engine` or `BALLISTICS_ENGINE`, and pin its version with `BALLISTICS_ENGINE_VERSION` (Section 2.7). Every run summary and every analysis table then carries the version it was solved with, so the notebook proves itself.

The probe itself never solves. It runs `solve-json --help` once per candidate and reads the output. The Lab does not start the configured engine until you type `:solve`.

2.6 Choosing the LATTICE backend

LATTICE has more than one execution engine of its own. The Lab supports two:

- **StackVM** — the default, a stack-based bytecode VM.
- **RegVM** — a register-based bytecode VM.

Select with `--backend stack-vm`, `--backend regvm`, the shorthands `--stack-vm` and `--regvm`, or the environment variable `BALLISTICS_LAB_BACKEND`.

Listing 2.8: Selecting RegVM

```
$ sh scripts/ballistics-lab.sh --regvm
ballistics-lab: backend=regvm, engine=/Users/alexjokela/.local/bin/ballistics
```

The tree-walking interpreter, `--tree-walk`, is a LATTICE diagnostic tool and is deliberately *not* a supported Lab backend. The launcher refuses anything that is not one of the two:

Listing 2.9: Backend validation happens before anything is launched

```
$ sh scripts/ballistics-lab.sh --backend treewalk
ballistics-lab: unknown backend 'treewalk' (expected stack-vm or regvm)

$ sh scripts/ballistics-lab.sh --regvm --stack-vm
ballistics-lab: conflicting backend selections: regvm and stack-vm
```

Both exit with status 2 without starting `clat`.

2.6.1 Do the two backends agree?

They must, and on this machine they do, byte for byte. Three lines of input — a `:solve` on the shipped reference experiment, and a ten-row dope table over that same solve — run through both VMs. Call the file `session.txt`; Section 2.11 feeds it to two different engines later in this chapter.

Listing 2.10: `session.txt`

```
:solve
:dope 100 1000 100 yd
:quit
```

Listing 2.11: Same session, both VMs

```
$ sh scripts/ballistics-lab.sh         < session.txt > out_stack.txt
$ sh scripts/ballistics-lab.sh --regvm < session.txt > out_regvm.txt
$ diff out_stack.txt out_regvm.txt
$ shasum -a 256 out_stack.txt out_regvm.txt
56a07c785ce68d9ce0246c98ea193dcd3a3c426ad42b77662c5380f7b54f88cf  out_stack.txt
56a07c785ce68d9ce0246c98ea193dcd3a3c426ad42b77662c5380f7b54f88cf  out_regvm.txt
```

`diff` printed nothing. The hashes match.

Be precise about what that proves. The trajectory integration happens inside the Rust engine, in the same child process either way, so this is not a statement about floating-point ODE arithmetic. What it demonstrates is that the two LATTICE VMs agree on request construction, unit conversion,

interpolation, table assembly, and fixed-point number formatting — which is the part written in Lattice, and therefore the part that could have diverged.

The Lab's own position on this is worth stating: behaviour that differs between StackVM and RegVM is defined as a *Lattice defect*, not a Lab bug.

2.7 Naming and pinning the engine

Two environment variables control the engine, and they do different jobs.

`BALLISTICS_ENGINE` chooses *which* executable to run. It accepts a path or a bare name; a bare name is resolved through `PATH`:

Listing 2.12: Naming an engine explicitly

```
$ BALLISTICS_ENGINE=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics \
      sh scripts/ballistics-lab.sh
ballistics-lab: backend=stack-vm,
     engine=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Ballistics laboratory — Lattice v0.6.4
Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics

:solve
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
```

`BALLISTICS_ENGINE_VERSION` declares which version you *require*. It is a reproducibility control, not a selector: it does not change which binary runs, it changes what happens when the binary that ran turns out to be the wrong one. A successful solve from any other version becomes a structured error carrying both versions:

Listing 2.13: A pin that does not match

```
$ BALLISTICS_ENGINE_VERSION=0.37.0 sh scripts/ballistics-lab.sh
:solve
Error
  code: incompatible_engine_version
  message: expected engine version '0.37.0', got '0.32.0'
  exit code: 0
```

That is the stale `PATH` copy answering: we pinned the version we built and let discovery choose the binary, and discovery chose the other one.

Read `exit code: 0` carefully. The engine succeeded. It produced a valid v1 envelope, exit status zero, a complete trajectory — and the Lab threw the result away because it was not the version you asked for. That is the behaviour you want in a notebook whose numbers you will later stake a shot on.

Name the binary and pin the version together — the selector and the assertion — and the pin is silent:

Listing 2.14: A pin that matches

```
$ BALLISTICS_ENGINE=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics \
      BALLISTICS_ENGINE_VERSION=0.37.0 sh scripts/ballistics-lab.sh
:solve
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1000.00 yd
```

That pair is the invocation the rest of this book uses. Either half alone leaves a hole: the selector without the pin runs whatever is at that path today, and the pin without the selector is what produced the error above.

Leaving the pin unset disables the version check and nothing else. Strict schema-v1 validation of the response still runs in full.

`engine_version` is opaque

The Lab never parses the version string, never compares it as a semantic version, and never infers capability from it. It is compared for exact string equality against your pin, or ignored. A pin of `0.37` will not match `0.37.0`.

2.8 Verifying the installation

Five checks, in increasing order of thoroughness. Run at least the first three before trusting a number.

2.8.1 Check one: the engine answers

Name the engine once, in a shell variable, and the remaining checks all address the same binary. On a machine with more than one `ballistics` that is not fussiness — it is the difference between verifying your installation and verifying somebody else's.

Listing 2.15: Engine identity and capability

```
$ ENGINE=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics
$ $ENGINE --version
ballistics 0.37.0
$ $ENGINE solve-json --help | head -1
Solve one explicit-SI v1 JSON request from stdin and write one JSON envelope to stdout
```

2.8.2 Check two: the protocol round-trips

This is the smallest complete v1 request. All nine top-level members are required; six of them may be empty objects.

Listing 2.16: `probe.json` — a minimal solve-json v1 request

```
{"schema_version":1,
 "projectile":{"mass_kg":0.01134,"diameter_m":0.00782,
               "drag_model":"G7","ballistic_coefficient":0.243},
 "rifle":{"muzzle_velocity_mps":823.0},
 "shot":{"max_range_m":300.0},
 "atmosphere":{},"wind":{},"solver":{},"effects":{},
 "sampling":{"interval_m":150.0}}
```

Listing 2.17: Solving it

```
$ $ENGINE solve-json < probe.json
{"schema_version": 1, "engine_version": "0.37.0", "status": "ok"}
{"actual_range_m": 300.0, "maximum_height_m": 0.0,
 "time_of_flight_s": 0.4119407650866686,
 "terminal_speed_mps": 644.895553000399,
 "terminal_energy_j": 2358.0978551658445,
 "stability_factor": 1.7547882257792367, "termination": "max_range"}
```

(The engine writes one compact single-line envelope; the extract above is the envelope's identity fields and its `summary` object, pulled out for readability. Part IV reads a whole envelope field by field.)

If that works, the engine half of the installation is sound independently of Lattice.

2.8.3 Check three: the Lab starts and solves

Listing 2.18: The end-to-end smoke test

```
$ printf ':solve\n:quit\n' | sh scripts/ballistics-lab.sh --engine "$ENGINE"
ballistics-lab: backend=stack-vm,
     engine=/Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Ballistics laboratory — Lattice v0.6.4
Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1000.00 yd
  maximum height   : +1.62 in
  time of flight   : 1.71 s
  terminal velocity: 1145.96 ft/s
  terminal energy  : 510.20 ft-lb
  stability factor : 3.80
  spin drift       : not reported
Assumptions
  - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
  - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
  - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
  - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
  - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
  - default_applied: Constant vertical wind speed defaulted to 0 m/s. [$.wind.vertical_speed_mps]
  - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
  - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
  - default_applied: Enhanced spin drift defaulted to disabled. [$.effects.enhanced_spin_drift]
Warnings
  none
Quit requested.
```

That is the shipped reference experiment — a 175 gr G7 match load at 1,000 yards — solved by your engine, and that is the whole of what it prints: the run summary, nine assumptions, an empty

warnings block, and the exit line. If you see that, the installation is complete. Chapter 3 explains every line of it.

Nine assumptions, not ten

Each `default_applied` line names a field the experiment left unset and the value the engine chose. There were ten until the Lab began sending `target_height_m`; the engine no longer has to default the target height, so that line is gone. Chapter 13 explains what that field does and why it matters to your dope.

Feeding the Lab from a pipe

The Lab reads through LATTICE's `input()`, which uses GNU readline. When standard input is a pipe rather than a terminal, the `ballistics>` prompt is suppressed. That is why the piped transcripts in this book contain no prompts, while a live session shows them. Both are the same program.

2.8.4 Check four: the hermetic test suite

The Lab has eighteen test modules. This target runs fourteen of them, plus the executable reference experiment, on *both* LATTICE VMs, then the end-to-end REPL transcript test once. It needs no engine at all — it launches a portable fake engine, itself written in LATTICE, as a real child process:

Listing 2.19: `make test-ballistics-lab`, abridged

```
$ make test-ballistics-lab
=== ballistics_lab backend --stack-vm ===
units: 19 tests passed
ballistics_lab domain: all tests passed
ballistics_lab process backend: all tests passed
ballistics_lab analysis: all tests passed
ballistics_lab session: all tests passed
ballistics_lab persistence: all tests passed
ballistics_lab render: all tests passed
ballistics_lab command parser: all tests passed
ballistics_lab commands: all tests passed
=== ballistics_lab backend --regvm ===
...
=== ballistics_lab REPL transcripts ===
stack-vm: ballistics laboratory transcript passed
regvm: ballistics laboratory transcript passed
```

It also runs a 412-line contract test for the launcher script itself, which exercises discovery, relocation, and paths containing spaces without invoking a real `clat` or engine:

Listing 2.20: The launcher's own test suite

```
$ make test-ballistics-lab-launcher
Running POSIX ballistics Lab launcher tests...
  ok: default stack-vm and repository-local discovery
  ok: CLI discovery and backend precedence with spaced paths
  ok: environment discovery and backend precedence
  ok: long and shorthand backend selection
  ok: relative PATH entries survive the bundle-root chdir
  ok: help, VM-only backend validation, and option errors do not launch clat
  ok: invalid explicit overrides fail without fallback
  ok: incompatible PATH engines are skipped but explicit engines fail
  ok: clat exit status is preserved
Results: 9 launcher test groups passed
```

2.8.5 Check five: your engine, against the checked fixtures

The strongest check. It solves six checked request/response fixture pairs against *your* engine and compares the results, on both VMs:

Listing 2.21: Cross-interface conformance

```
$ make test-ballistics-lab-engine \
      BALLISTICS_ENGINE="$ENGINE" \
      BALLISTICS_ENGINE_VERSION=0.37.0
=== ballistics_lab real engine --stack-vm ===
ballistics_lab real engine: 0.37.0 passed
=== ballistics_lab real engine --regvm ===
ballistics_lab real engine: 0.37.0 passed
=== ballistics_lab real-engine REPL transcripts ===
stack-vm: real-engine ballistics REPL transcript passed
regvm: real-engine ballistics REPL transcript passed
=== ballistics_lab cross-interface conformance --stack-vm ===
conformance calm-air: ok (engine 0.37.0)
conformance crosswind: ok (engine 0.37.0)
conformance segmented-wind: ok (engine 0.37.0)
conformance zeroed: ok (engine 0.37.0)
conformance representative: ok (engine 0.37.0)
conformance early-termination: ok (engine 0.37.0)
ballistics_lab conformance: all fixtures passed
=== ballistics_lab cross-interface conformance --regvm ===
conformance calm-air: ok (engine 0.37.0)
...
ballistics_lab conformance: all fixtures passed
```

`BALLISTICS_ENGINE` is required by this target and it fails loudly without it. `BALLISTICS_ENGINE_VERSION` is optional; omitting it drops the version pin while keeping full schema validation.

2.9 Running without the launcher

Nothing stops you from starting the Lab directly. The engine can come from the command line, from the environment, or — as a last resort — from the bare name `ballistics` resolved through `PATH`:

Listing 2.22: Three ways to name the engine without the launcher

```
$ ./clat examples/ballistics_lab/ballistics-repl.lat /absolute/path/to/ballistics

$ BALLISTICS_ENGINE=/absolute/path/to/ballistics \
      ./clat examples/ballistics_lab/ballistics-repl.lat

$ ./clat examples/ballistics_lab/ballistics-repl.lat
Ballistics laboratory — Lattice v0.6.4
Engine: ballistics
```

The precedence inside the application is: `argv[1]`, then `BALLISTICS_ENGINE`, then the literal string `"ballistics"`. Note that the third form starts perfectly happily and prints `Engine: ballistics` — because the application performs no discovery and no probe of its own. If that name does not resolve, you find out at `:solve`:

Listing 2.23: The failure the launcher exists to prevent

```
$ env PATH=/usr/bin:/bin ./clat examples/ballistics_lab/ballistics-repl.lat
Ballistics laboratory — Lattice v0.6.4
Engine: ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

:solve
Error
  code: process_error
  message: exec_argv: failed to spawn 'ballistics': No such file or directory
```

The working-directory trap

Every Lab import path is written relative to the repository root, like `"examples/ballistics_lab/units"`. Start `clat` from anywhere else, even with an absolute path to the script, and it fails immediately:

```
$ cd /tmp
$ /path/to/lattice/clat /path/to/lattice/examples/ballistics_lab/ballistics-repl.lat \
      /path/to/ballistics
vm error: import: cannot find 'examples/ballistics_lab/units.lat'
```

The launcher `cd`s to the checkout root before `exec`, which is why it works from any directory:

```
$ cd /tmp
$ sh /path/to/lattice/scripts/ballistics-lab.sh
ballistics-lab: backend=stack-vm, engine=/Users/alexjokela/.local/bin/ballistics
Ballistics laboratory — Lattice v0.6.4
```

2.10 When it does not work

Every message below was produced by an actual failed invocation. All of the launcher's failures exit with status 2 and none of them start `clat`.

2.10.1 The launcher cannot find `clat`

```
ballistics-lab: clat was not found; build the checkout with 'make' or set CLAT
```

Run `make all` in the LATTICE checkout, or point `CLAT` at a runtime.

2.10.2 The launcher cannot find an engine

```
ballistics-lab: ballistics was not found; install it, build ../ballistics-engine,
or set BALLISTICS_ENGINE
```

Build the sibling engine checkout, or export `BALLISTICS_ENGINE`.

2.10.3 A candidate exists but is skipped

```
ballistics-lab: ignoring incompatible engine /some/path/ballistics
(no solve-json v1 command)
```

That candidate is an older or unrelated `ballistics`. Discovery continues past it. Upgrade it or override the path.

2.10.4 A named engine is wrong

An explicit engine never falls back. Two distinct failures:

Listing 2.24: Named engine failures

```
$ sh scripts/ballistics-lab.sh --engine nosuchengine
ballistics-lab: ballistics executable 'nosuchengine' was not found or is not executable

$ sh scripts/ballistics-lab.sh --engine /bin/ls
ballistics-lab: ballistics executable '/bin/ls' does not provide the required
solve-json v1 command
```

The second is the interesting one: `/bin/ls` exists and is executable, and the launcher still refuses it, because it failed the capability probe rather than the existence check.

2.10.5 Failures that happen inside the Lab

Three you may meet on a fresh install, all of which leave the session usable:

- `process_error` — the engine could not be spawned, timed out, or exceeded an output limit.
- `incompatible_engine_version` — a successful solve from an engine that is not the pinned version.
- `invalid_engine_response` — the engine ran and produced something the protocol did not promise: a banner on stdout, truncated or concatenated JSON, an unknown field, a non-finite number, or an exit code that disagrees with the envelope's status.

That last one is worth internalizing now. The Lab has no lenient mode. There is no flag that says "accept it anyway."

2.11 A word about engine versions

This machine has two engines: a stale `0.32.0` installed on `PATH`, and the `0.37.0` this book is written against, built in the sibling checkout. The `PATH` copy was installed once and never refreshed; the checkout tracks development, so it moves. That is not tidy, and it turns out to be the most useful demonstration in this chapter.

Take the shipped reference experiment, solve it and print a ten-row dope table, once against each engine, and diff the two captures:

Listing 2.25: Five minor versions apart, on the same experiment

```
$ sh scripts/ballistics-lab.sh                    < session.txt > out_v032.txt
$ sh scripts/ballistics-lab.sh --engine "$ENGINE"  < session.txt > out_v037.txt
$ diff out_v032.txt out_v037.txt
2c2
< Engine: /Users/alexjokela/.local/bin/ballistics
---
> Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
7c7
<   engine           : 0.32.0
---
>   engine           : 0.37.0
46c46
<     engine: 0.32.0 (schema 1)
---
>     engine: 0.37.0 (schema 1)
```

Three lines differ, and all three are identity, not physics. The banner path, the run summary's `engine` field, and the same field inside the dope table's provenance block. Every trajectory number — ten rows of drop, come-up, velocity, energy, time and Mach, plus the whole run summary — is character-for-character identical.

That is what the v1 compatibility rules are for. Removing a field, changing a unit, changing a sign, renaming an enum value or changing a field's meaning all require a v2. Version 1 grows only by addition, and a new response field must be *omitted* whenever its feature is inactive, so that a response which does not exercise the new feature is byte-identical to one from before the field existed.

Five releases separate these two binaries, and they were not idle ones. Between them, 0.33 through 0.37 added the truing bridge, put wind shear on the `solve-json` surface, and reworked how a dope card reports a range the load cannot reach. None of that moves this experiment, because none of it happens unless you ask for it: the new request fields are opt-in, and the response fields that answer them are omitted when they are not in play. The clean diff is the compatibility rule doing its job, not a coincidence.

Do not read that as "all versions agree"

The two engines agree on *this* experiment, which uses only long-settled parts of the protocol. Newer engines add fields, and newer physics changes numbers where the physics changed. The versions in this book are the versions each run actually reported. If your engine reports something else, expect the identity lines to differ, and check the numbers yourself before assuming they do not.

With both executables built, both probes answering, and the reference experiment solving, you have a laboratory. Chapter 3 puts a rifle in it.

Chapter 3

Your First Session

This chapter is one session, start to finish. We launch the Lab, look at what it shipped with, replace it with a real rifle, solve, query a distance, build a dope table, perturb the wind, and save the experiment to disk. Every line of output below came out of that session.

Transcript conventions

The Lab's prompt is `ballistics>` , and a continuation prompt `...>` appears while a multi-line block is incomplete. Both are suppressed when standard input is a pipe, which is how these transcripts were captured, so what you see below is the input line followed directly by its output. The startup banner is shown once, at the beginning, and elided afterwards.

3.1 Launching

Listing 3.1: Starting the Lab

```
$ sh scripts/ballistics-lab.sh --engine \
      /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Ballistics laboratory — Lattice v0.6.4
Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

ballistics>
```

We name the engine rather than letting discovery pick one, for the reason Section 2.11 sets out: this machine also carries an older `ballistics` on `PATH`, and a session whose numbers you intend to keep should say which binary produced them. Every session in this book is launched that way.

Three lines, and each is a fact worth having.

The first names the LATTICE runtime. The second names the engine executable the Lab will run — *the path, not the version*, because the Lab has not started the engine yet and will not until you ask it to solve something. The third tells you the two kinds of input the Lab accepts: colon commands, and LATTICE.

That second kind is the one that makes this a laboratory rather than a menu. Anything that does not begin with a colon is evaluated as LATTICE source, and it persists. Declarations, functions, multi-line blocks — all of it stays available for the rest of the session.

3.2 `:help` — the whole surface

Listing 3.2: The complete command surface

```
:help
Commands:
:help
:show
:solve
:at <distance> <unit>
:dope <start> <stop> <interval> <unit>
:wind-card <start> <stop> <interval> <unit>
:compare
:save "<path>"
:load "<path>"
:reset
:quit
```

Eleven commands. That is the entire colon vocabulary, and it will not grow while you are reading this book.

Look at what is missing: there is no `:set`. Nothing in that list can change the rifle, the bullet, the air or the wind.

Why there is no :set

This is a design decision, stated in the Lab's own documentation: rifle, projectile, atmosphere and other experiment changes "remain visible ordinary Lattice calls rather than a second :set language."
The alternative — a :set projectile.bc 0.315 mini-language — would need its own grammar, its own validation, its own error messages, and its own documentation, all shadowing constructors that already exist and already validate. Instead the colon commands do the things that are awkward to type as expressions (solve, tabulate, render, save) and LATTICE does the rest.

3.3 :show — what shipped in the box

The Lab starts with a real experiment already loaded. :show prints it in two parts. First, the session:

Listing 3.3: The session block

```
:show
Ballistics laboratory session
  active experiment: 175 gr match at 1,000 yards
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none
```

Six facts about the state of the laboratory itself:

- **active experiment** — the name of the loaded experiment.
- **backend** — process-json-v1: the process-per-solve backend speaking solve-json schema 1. This is the only backend the shipped Lab constructs, though the seam accepts others.
- **display** — your unit preferences, and the decimal precision every rendered number will use. Imperial by default: yards, feet per second, foot-pounds, mils.
- **history** — runs retained, out of the limit. Zero of fifty.
- **current run / previous run** — the last two successful solves. Both empty; we have not solved anything.

Then the experiment:

Listing 3.4: The shipped reference load — bullet and rifle

```
Experiment: 175 gr match at 1,000 yards
  projectile           : 175 gr match
  mass                 : 175.00 gr
  diameter             : 0.31 in
  length               : 1.24 in
  drag model           : G7
  ballistic coefficient: 0.24
  rifle                : 24-inch match rifle
  sight height         : 1.50 in
  muzzle height        : 0.00 in
  twist rate           : 8.00 in
  twist direction      : right
```

A 175 grain match bullet, 0.308 inches across, 1.24 inches long, referenced to the G7 drag curve with a BC of 0.243, out of a rifle with a 1.5-inch sight height and an 8-inch right-hand twist.

Rendered precision is not stored precision

`diameter : 0.31 in` and `ballistic coefficient: 0.24` are the display preference at work: two decimals, applied to everything. The stored values are 0.308 inches and 0.243. Nothing has been rounded in the experiment; only in the rendering. Raise the precision — Part II shows how — and the digits come back.

Listing 3.5: The shipped reference load — air, shot, and solver

```
altitude             : 273.40 yd
temperature          : 59.00 F
pressure             : 29.92 inHg
humidity             : 50.00 %
latitude             : +45.00 deg
muzzle velocity      : 2700.00 ft/s
maximum range        : 1000.00 yd
zero distance        : 100.00 yd
muzzle angle         : default
aim azimuth          : default
shot azimuth         : default
shooting angle       : default
cant angle           : default
target height        : default
ground threshold     : default
solver               : rk45
time step            : default
sampling interval    : 10.00 yd
Magnus               : default
Coriolis             : on
enhanced spin drift  : default
wind[0]              : 10.00 mph from 90.00 deg
```

Four things in that block are worth stopping on.

`default` is not a value. It means the experiment does not supply that field, so the engine will apply its documented default and say so. There is a real difference between omitting `cant angle` and setting it to zero: the first produces an assumption notice in the response, the second appears in the request.

`wind[0] : 10.00 mph from 90.00 deg`. Wind uses the *wind-from* convention everywhere in this system. Zero degrees is a headwind, 90 is wind from the shooter's right, 180 is a tailwind, 270 is from the left. This one is worth over-learning; getting it backwards inverts every windage number you will ever read.

`Coriolis : on`, and a latitude. The shipped experiment enables Earth-rotation effects, which is why it also supplies `latitude : +45.00 deg`. The two are coupled by a validation rule: asking for Coriolis without a latitude is rejected before any request is built.

`altitude : 273.40 yd`. That reads oddly, and it is not a mistake in the number — 250 metres is 273.40 yards. The renderer applies your single distance preference to every length, including vertical ones. If you prefer altitudes in feet you will need a metric or custom profile; there is no separate altitude unit.

3.4 Building a rifle

The shipped experiment is a demonstration. Let us replace it with the rifle from Chapter 1: a 6.5 mm match load, at altitude, in a left-hand wind.

Each component is replaced by calling a constructor and handing the result to the session. The Lab imports all twenty-five unit constructors and all fifteen domain constructors into the prompt's environment unqualified, so you can write `projectile(...)` and `gr(140)` directly. The module aliases — `units`, `domain`, `sessions`, `analysis`, `persistence`, `render`, `commands`, `artifacts` and the rest — are there too, for everything else.

Listing 3.6: The bullet

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140),
    inches(0.264), 0.315, "G7", inches(1.34)))
```

Read the arguments: a name, a mass, a diameter, a ballistic coefficient, the drag model that BC is referenced to, and a length.

The units are not decoration. `gr(140)` constructs a `Mass`; `inches(0.264)` constructs a `Distance`. They are distinct types carrying a canonical SI value and your display preference together. Hand a velocity to a parameter that wants a mass and the constructor rejects it — there is no coercion, and no way for feet per second to be silently read as metres per second.

The BC is a bare number, deliberately: a ballistic coefficient is dimensionless. The drag model beside it is what gives it meaning. The length is optional, and supplying it matters — the engine will otherwise estimate a length from mass and diameter and tell you it did so, and length-dependent physics such as spin drift and Magnus require it.

Listing 3.7: The rifle

```
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in", inches(1.8),
    m(0), inches(8), "right"))
```

A name, sight height above the bore, muzzle height above the ground, twist rate, and twist hand. Sight height is the one that silently ruins dope if you guess it. Twist rate and direction feed stability and spin drift.

Listing 3.8: The air

```
sessions.replace_atmosphere(lab, atmosphere(m(1200), celsius(12),
    hpa(880), 0.35, deg(44)))
```

Station altitude, temperature, station pressure, relative humidity as a fraction, and geodetic latitude. Every one of those is a real measurement you can take at the firing line.

Station pressure, not the airport's

The pressure field is *absolute station pressure* — what a barometer reads where you are standing. A weather report or an altimeter setting is QNH, corrected to sea level, and at 1,200 m those differ enormously: 880 hPa here versus something near 1013 on the report. Entering a sea-level-corrected value as station pressure overstates air density and understates your drop.

Listing 3.9: The wind

```
sessions.replace_winds(lab, [wind(mph(8), deg(270))])
```

Winds are an array, because the engine also accepts downrange segments. One constant wind is an array of one. `deg(270)` is wind *from* 270 — from the shooter's left, pushing the bullet right.

Listing 3.10: The shot

```
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100)))
```

Muzzle velocity, maximum range, zero distance. `shot` has seven more optional parameters — muzzle angle, aim azimuth, shot azimuth, shooting angle, cant angle, target height and ground threshold — and a rule: *zero distance and muzzle angle are mutually exclusive.* You either tell the engine where you zeroed and let it solve the elevation, or you tell it the elevation directly. Supplying both is rejected.

Listing 3.11: The solver

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), nil, true, nil))
```

Method, fixed time step, sampling interval, Magnus, Coriolis, enhanced spin drift. We keep adaptive RK45, pass `nil` for the time step (RK45 owns its own step and warns if you supply one), sample every 25 yards, leave Magnus off, turn Coriolis on, and leave enhanced spin drift off.

Sampling interval is a resolution knob with a hard ceiling

25 yards over 1,200 yards is 49 samples. The protocol caps a response at 10,000 samples and *errors* rather than thinning: ask for a 0.1 yard interval over 1,200 yards and you get a `resource_limit`, not a coarser table. Interpolated queries such as `:at` do not need a fine grid; a comparison over the raw sample set does.

3.5 The name that did not change

Now look at what `:show` says after all six replacements:

Listing 3.12: Everything changed except the label

```
:show
Ballistics laboratory session
  active experiment: 175 gr match at 1,000 yards
  ...
Experiment: 175 gr match at 1,000 yards
  projectile          : 140 gr ELD-M
  mass                : 140.00 gr
  diameter            : 0.26 in
```

A lab notebook that mislabels every page

The experiment is still called `175 gr match at 1,000 yards` while describing a 140 gr ELD-M. Every `replace_*` call rebuilds the whole experiment and carries the existing name forward, because the name belongs to the experiment, not to any one component.

This matters more than it looks. That name is printed on every run summary and in the provenance block of every dope table, wind card and comparison you export. A session built entirely from `replace_*` calls will label every artifact with the name of a load you are not shooting.

The fix is to replace the experiment as a whole. Fetch the current one, and rebuild it with a new name and the same components:

Listing 3.13: Renaming

```
let cur = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("140 gr ELD-M at 1,200 yd",
    cur.projectile, cur.rifle, cur.atmosphere, cur.winds, cur.shot, cur.solver))
:show
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none
```

That is also a first look at reading the experiment as data. `cur` is an ordinary LATTICE value with fields you can inspect, and `cur.projectile.mass.si_value` is a number in kilograms. Nothing about the session is hidden behind an opaque handle.

Name it first

The habit that avoids the whole problem: build the complete experiment with its name up front and install it in one `replace_experiment` call, then use `replace_*` only for the perturbations you are actually studying.

3.6 `:solve`

Now we run it. `:solve` is the only command that starts a process.

Listing 3.14: The run summary

```
:solve
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.37.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1200.00 yd
  maximum height   : +1.85 in
  time of flight   : 1.85 s
  terminal velocity: 1396.15 ft/s
  terminal energy  : 605.84 ft-lb
  stability factor : 2.06
  spin drift       : not reported
```

Behind that block: the experiment was converted to a solve-json v1 request, serialized, and written to the standard input of a freshly spawned `ballistics solve-json`. The child returned one JSON envelope. Every field of it was validated — the resolved request, every notice, every sample, the summary, and a cross-check that the last sample agrees with the summary to within $10^{-10} + 10^{-9}$ relative — before any of it became a value you can see. On the machine this book was written on, a complete launch, solve and quit cycle measures 0.042 seconds of wall time, of which the engine's own solve is about three milliseconds.

Reading the summary:

- **engine : 0.37.0** — reported by the engine, recorded on this run, and carried into every table built from it.
- **verified : no** — the normal state. "Verified" is a specific thing in this system: a run promoted into an immutable, portable artifact document. It is not a quality judgement, and Part II shows how to make one.
- **termination : max_range** — *we got the range we asked for*. The alternative you will meet is `ground_threshold`, meaning the bullet reached the ground first. This is the field that makes the CLI's 532-yard surprise from Chapter 1 impossible to miss.
- **maximum height : +1.85 in** — greatest height above the horizontal plane through the *muzzle*, not height above the line of sight. The sight sits 1.8 inches above the muzzle, so a bullet that peaks a few hundredths of an inch over the sight line reads a hair over 1.8. Chapter 5 pins that identity down with numbers.

- **stability factor : 2.06** — the Miller gyroscopic stability factor at the muzzle. The conventionally quoted floor is somewhere between 1.4 and 1.5; the engine's own `stability` subcommand reports the minimum twist for $S_g = 1.5$ and for $S_g = 1.0$.

- **spin drift : not reported** — the engine reports a spin drift figure only when the enhanced spin-drift effect is enabled, which we left off. When it is on, that number is already folded into windage; it is not something to add.

3.6.1 The assumptions block

Immediately below the summary, the Lab prints what the engine assumed:

Listing 3.15: Nine assumptions on a fully specified experiment

```
Assumptions
  - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
  - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
  - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
  - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
  - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
  - default_applied: Constant vertical wind speed defaulted to 0 m/s. [$.wind.vertical_speed_mps]
  - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
  - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
  - default_applied: Enhanced spin drift defaulted to disabled. [$.effects.enhanced_spin_drift]
Warnings
  none
```

Nine assumptions on an experiment we specified carefully. That is not a complaint from the engine; it is a receipt. Each line names the field, the value applied, and its JSON path, in deterministic request-field order.

This is the difference between a tool that answers and a tool that shows its work. If tomorrow you wonder whether that run modelled a cant, the run itself tells you: cant angle defaulted to 0 rad. And because assumptions travel with every table built from the run, the receipt survives export.

One field that is *not* on that list is worth naming, because its absence is the interesting part. The v1 request has a `target_height_m` field: the height above the ground that the zero is solved to. Leave it out and the engine defaults it to zero — a target sitting on the dirt — and says so with a tenth assumption line. The Lab fills the field in for you, at the height of your line of sight, so the zero is solved where you actually aimed. We watch that convention behave in Section 3.8, and Chapter 5 takes it apart properly.

Assumptions versus warnings

An **assumption** says "you left this out and here is what I used." A **warning** says "you supplied something I could not honour as given." Engine 0.37.0 defines six of them. Three are as old as the v1 protocol: `partial_wind_coverage` (a segmented wind deck ends short of the solve range), `experimental_effect` (Magnus or enhanced spin drift is on), and `rk45_time_step_ignored` (you passed a fixed step to an adaptive integrator). Three more arrived across 0.33 to 0.36, as the engine grew surfaces that can accept a request and then not fully honour it: `zero_distance_elevation_not_resolved` (you gave both a zero distance and a muzzle angle, so the angle was used directly and the elevation search never ran), `bc5d_drag_model_coerced` (a BC5D table was asked for a drag model outside its G1/G7 planes, and the lookup was typed as G1), and `wind_shear_model_not_modeled` (a shear model was accepted and echoed back in the resolved request, but no profile behind it applies on this solve path, so the wind was left unchanged).

That last one is the shape to learn, because it is the one that looks like success. The model you asked for appears in `resolved_request` whether or not it did anything; the warning is the only thing that separates "applied, and it moved the numbers" from "applied, and it could not." Our run has none of the six.

3.7 `:at` — asking one question

A trajectory is a lot of numbers. `:at` pulls one distance out of it.

Listing 3.16: A single observation

```
:at 600 yd
Observation
  distance: 600.00 yd
  time    : 0.77 s
  velocity: 1997.90 ft/s
  energy  : 1240.63 ft-lb
  drop    : +81.08 in
  windage : +16.27 in
  Mach    : 1.80
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Read the legend on the last line, because both halves of it matter. The drop is positive, meaning 81.08 inches *below* the line of sight. The windage is positive, meaning right, which is what a wind from 270 degrees — from the left — should do. The Mach number is 1.80; when it falls between 1.2 and 0.8 the `flags` field will start reporting `transonic`.

`:at` interpolates. It is not restricted to sample points, and it accepts any of `m`, `km`, `yd`, `ft`, `in` — then answers in *your display unit*, not the unit you asked in:

Listing 3.17: Ask in metres, get yards

```
:at 500 m
Observation
  distance: 546.81 yd
  time    : 0.70 s
  velocity: 2056.04 ft/s
  energy  : 1313.91 ft-lb
  drop    : +63.91 in
  windage : +13.33 in
  Mach    : 1.85
  flags   : none
```

500 metres is 546.81 yards, and that is what comes back. The query unit selects the distance; the display preference selects the presentation.

It also refuses to invent an answer. Earlier in this same session — before we had solved anything at all — the same command produced this instead:

Listing 3.18: No run, no answer

```
:at 300 yd
Error
  code: command_state_error
  message: at: no current run is available
  path: $.session.current_run
```

A code, a message, and a JSON path naming the piece of session state that was missing. The session is untouched, the prompt returns, and you carry on. That is the pattern for every failure in the Lab: errors are values that get rendered, not exceptions that end the session.

3.8 Where drop is measured from

Before we build a table, let us settle the legend that printed under that observation:

```
Signs: drop + below/- above; windage + right/- left
```

Two claims, and both are measurable. *Drop is positive below* — read the number as a depth, not as a height. *Windage is positive right* — read it as an offset from where you aimed. A dope table is a list of drops, so it is worth three minutes to watch the convention behave.

One: move the zero, in the Lab

Take the same rifle and zero it at 300 yards instead of 100, then ask at the muzzle, at 100, and at 300. With a 300-yard zero, a bullet at 100 yards is unambiguously *above* the sight line:

Listing 3.19: A 300-yard zero, queried at the muzzle, 100 and 300 yards

```
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(300)))
:solve
:at 0 yd
Observation
  distance: 0.00 yd
  drop    : +1.80 in
Signs: drop + below/- above; windage + right/- left
:at 100 yd
Observation
  distance: 100.00 yd
  drop    : -4.19 in
Signs: drop + below/- above; windage + right/- left
:at 300 yd
Observation
  distance: 300.00 yd
  drop    : +0.00 in
Signs: drop + below/- above; windage + right/- left
```

Three numbers, three facts.

At the muzzle the drop is $+1.80$ inches — exactly the sight height we entered. That is not a coincidence and it is not an offset the Lab adds. The bore *is* 1.8 inches below the line of sight at the muzzle, so a bullet sitting in it is 1.8 inches low.

At 100 yards, with the zero out at 300, the bullet is climbing and reads -4.19 — negative, which the legend calls "above". That is the mid-range rise, and it is the only place in a normal trajectory where drop goes negative.

At 300 yards — the zero — the drop is $+0.00$. The bullet is on the line of sight, which is what a zero means.

Two: the request the Lab sends

That last one only happens because the Lab tells the engine where the target is. The v1 protocol has a field, `target_height_m`, that says how far above the ground the zero should be solved to; omit it and the engine puts the target on the dirt. The Lab fills it in with the height of your line of sight — muzzle height plus sight height — so the zero lands where you were looking.

You can read that straight off the run. The engine echoes back every value it actually resolved, and the Lab keeps the echo:

Listing 3.20: The engine's own echo of what it resolved

```
let rq = sessions.current_run(lab).resolved_request
json_stringify(rq.get("rifle"))
{"twist_direction":"right","muzzle_velocity_mps":826.00800000000004,"muzzle_height_m":0,"
     sight_height_m":0.045719999999999997,"twist_rate_m_per_turn":0.20319999999999999}
json_stringify(rq.get("shot"))
{"cant_angle_rad":0,"muzzle_angle_rad":0.0023437500000000003,"max_range_m":1097.28,"
     zero_distance_m":274.31999999999999,"aim_azimuth_rad":0,"shooting_angle_rad":0,"
     ground_threshold_m":-100,"shot_azimuth_rad":0,"target_height_m":0.045719999999999997}
```

`sight_height_m` and `target_height_m` are the same number, 0.04572 m, because `muzzle_height_m` is zero and $0 + 0.04572$ is the height of the sight line above the ground. The zero is solved to the line of sight rather than to the ground under it, which is why the drop at the zero distance is 0.00 and not one sight height.

Three: the Lab does not touch the number

Once the engine has answered, the decoder maps the field straight through, with no negation and no offset:

Listing 3.21: `process_backend.lat`, sample decoding

```
return observation(
m(object.get("distance_m")),
object.get("time_s"),
mps(object.get("speed_mps")),
joules(object.get("energy_j")),
m(object.get("drop_m")),
m(object.get("windage_m")),
object.get("mach"),
clone(object.get("flags"))
)
```

Read drop as depth

`drop` is positive *downward*, measured from the line of sight. A drop of +300.82 inches at 1,000 yards means the bullet is 300.82 inches *below* where you are looking — which is what the word says. It is one sight height at the muzzle, exactly zero at the zero distance, negative wherever the bullet rides above the sight line, and growing everywhere past that.
The rule for a first session: **bigger drop, further below**, and the elevation you have to dial is in the come-up column next to it.

3.9 `:dope` — the first table

The sign check moved our zero to 300 yards, so put it back and solve again — then ask for ten rows from 100 to 1,000:

Listing 3.22: A dope table

```
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100)))
:solve
:dope 100 1000 100 yd
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       300.00 |    +12.58 |         +1.16 |         2338.37 |        1699.50 |     0.36 | 2.10
       400.00 |    +28.10 |         +1.95 |         2221.35 |        1533.66 |     0.49 | 2.00
       500.00 |    +50.68 |         +2.82 |         2107.93 |        1381.04 |     0.63 | 1.90
       600.00 |    +81.08 |         +3.75 |         1997.90 |        1240.63 |     0.77 | 1.80
       700.00 |   +120.20 |         +4.77 |         1890.96 |        1111.37 |     0.93 | 1.70
       800.00 |   +169.05 |         +5.87 |         1786.81 |         992.32 |     1.09 | 1.61
       900.00 |   +228.81 |         +7.06 |         1685.30 |         882.78 |     1.26 | 1.52
      1000.00 |   +300.82 |         +8.36 |         1586.32 |         782.12 |     1.45 | 1.43
Signs: drop + below/- above; come-up + correction up/- correction down
```

The arguments are start, stop, interval and unit. The drop column climbs the way drop climbs; velocity falls; energy falls faster, because it goes as velocity squared; the Mach column tells you how much supersonic margin is left. At 1,000 yards this load is still at Mach 1.43, comfortably clear of the transonic region.

The come-up column is the one you dial. It starts at +0.00 at the zero and climbs monotonically to +8.36 mil at 1,000 yards, and the legend says which way: positive means the sight goes *up*.

You can check the arithmetic without leaving the prompt. A mil is exactly one milliradian, so at these angles the come-up in mils is the drop divided by the distance, times a thousand:

Listing 3.23: Checking the arithmetic in the same session

```
fn mils(drop_in: Number, range_yd: Number) -> Number {
    return drop_in / (range_yd * 36.0) * 1000.0
}
mils(300.82, 1000.0)
8.35611
```

8.35611, rounding to the +8.36 in the table. That is also your first multi-line LATTICE definition at the prompt, and `mils` will still be there for the rest of the session.

A second opinion, from the engine itself

The engine has its own table generator — the `range-table` subcommand — reached through a completely different code path, flags on a subcommand instead of solve-json, and it prints elevation in mils. Asked the same question with the same air and no wind, it returns −0.000, 1.951, 3.754 and 8.357 mil at 100, 400, 600 and 1,000 yards, against the Lab's +0.00, +1.95, +3.75 and +8.36. All ten rows agree. Chapter 5 prints both tables in full and shows the invocation.

3.9.1 Provenance travels with the table

Below every table, the Lab prints where it came from:

Listing 3.24: The provenance block

```
Provenance
  [0] 140 gr ELD-M at 1,200 yd
    engine: 0.37.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
    Assumptions
      - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
      - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
      - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
      - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
      - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
      - default_applied: Constant vertical wind speed defaulted to 0 m/s.
     [$.wind.vertical_speed_mps]
      - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
      - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
      - default_applied: Enhanced spin drift defaulted to disabled.
     [$.effects.enhanced_spin_drift]
    Warnings
      none
```

The series name, the engine version, the schema, the solver method, whether the run was verified, how it terminated, the range actually reached, and every assumption and warning. A dope table printed six months ago can still tell you which engine produced it and what it assumed.

The `[0]` is a series index. A dope table has one series; a wind card or a comparison can have several, and each row carries the index of the series it belongs to.

3.10 Perturbing

Here is the payoff for all the setup. The flag picked up — eight to fifteen. One line:

Listing 3.25: A single perturbation, end to end (run summaries elided)

```
:at 1000 yd
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +50.43 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
sessions.replace_winds(lab, [wind(mph(15), deg(270))])
:solve
:at 1000 yd
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.13 ft-lb
  drop    : +300.82 in
  windage : +92.45 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Nothing was retyped. The bullet, the rifle, the 1,200 m of altitude, the 880 hPa of station pressure, the 44 degrees of latitude with Coriolis enabled, the 25-yard sampling — all of it persisted, and the answer to the question we actually asked is the second-to-last line: 50.43 inches of drift becomes 92.45.

Three details in that output reward a second look.

The drift did not scale with the wind. 8 to 15 mph is a factor of 1.875, and $50.43 \times 1.875 = 94.6$, not 92.45. Crosswind drift is *approximately* linear in wind speed but not exactly so, because the stronger wind slightly alters the trajectory and therefore the lag time. That two-inch difference is real physics, and it is visible only because we could change one term and hold everything else.

The drop did not move at all. $+300.82$ inches in both runs, to the hundredth. A pure beam wind moves the bullet sideways.

The energy moved in the last printed digit — 782.12 to 782.13 ft-lb. That is not noise. A crosswind makes the bullet fly at a very slight yaw, and the flight path is marginally different: on the wire the speed at 1,000 yards goes from 483.50956593249703 to 483.51177349938041 m/s, a change too small to

show at two decimals in feet per second but large enough to show in the energy. The Lab reports it rather than hiding it.

The command that compares runs for you

Because we solved twice, the session now holds a current run *and* a previous one, and `:compare` will align them and print the deltas. It takes no arguments and emits the whole sampled trajectory, two rows per distance, so save it for when you want it. Part II gives it a chapter.

3.11 Saving, and leaving

`:save` writes the *experiment* — not the run, not the history — to a strict versioned JSON document:

Listing 3.26: Saving the experiment

```
:save "/tmp/eldm.json"
Saved experiment to /tmp/eldm.json
```

1,282 bytes, and readable by anything that reads JSON:

Listing 3.27: The first 400 bytes of the saved document

```
{"experiment":{"rifle":{"twist_direction":"right","name":"Tikka T3x CTR 24 in",
"sight_height":{"unit":"in","value":1.8},"muzzle_height":{"unit":"m","value":0},
"twist_rate":{"unit":"in","value":8}},"atmosphere":{"pressure":{"unit":"hPa",
"value":880},"relative_humidity":0.34999999999999998,"temperature":{"unit":"C",
"value":12},"latitude":{"unit":"deg","value":44},"altitude":{"unit":"m",
"value":1200}},"name":"140 gr ELD-M at 1,200 yd", ...
```

Every quantity is stored as the value and the unit you entered it in — 880 and `"hPa"`, 1.8 and `"in"` — not as SI. The canonical SI form is reconstructed and cross-checked on load, so a hand-edited file that lies about its own numbers is rejected rather than believed.

Note `0.34999999999999998`. The Lab writes numbers in a round-trip-safe form rather than a pretty one, because a document that reloads as a slightly different experiment is worse than an ugly document.

`:load` does not clear the run

Loading an experiment replaces the experiment and nothing else. The current run, the previous run and the history all survive — which means `:at`, `:dope`, `:wind-card` and `:compare` will keep answering from the *old* trajectory, while `:show` reports the newly loaded rifle.
The safe idiom is `:load`, then `:reset`, then `:solve`. Part II demonstrates the trap with numbers.

Finally:

Listing 3.28: Leaving

```
:quit
Quit requested.
```

Ctrl-D does the same thing, printing a blank line instead of the message. The process exits with status 0 either way — including after a session full of errors, because a laboratory that dies when you mistype is not a laboratory.

3.12 What we did not do

That was one pass through the Lab's main loop. Several things were skipped deliberately, and each has a chapter waiting:

- **Wind cards.** `:wind-card` tabulates windage and hold across a distance band, with the same provenance discipline.
- **Comparison.** `:compare` aligns two runs on a common distance grid and reports deltas — with the previous run as the baseline, so deltas read *current minus previous*.
- **Display preferences.** Everything above was imperial with two decimals because that is the default. Metric, and higher precision, are one call away.
- **Segmented wind.** A wind that changes downrange is an array of winds with strictly increasing boundaries.
- **Verified runs.** Promoting a run into a portable, immutable artifact with an embedded resolved request.
- **JSON and CSV export.** Not reachable from the colon commands at all — it is a source-level capability, with provenance repeated on every CSV row so a Python reader keeps it without a sidecar.

- **The physics.** What the engine actually models, what each effect is worth in inches, and when to turn it on.

You now have a running laboratory, a rifle in it, a dope table that agrees with the engine's own, and three sign conventions you can read off a legend without guessing. Part II takes the session apart properly.

Part II

Working the Session

Chapter 4

The Session Model

A ballistics solver that forgets what you told it is a calculator. A ballistics solver that remembers is a laboratory. The difference is state, and the BALLISTICS LAB holds a very particular, very small amount of it.

This chapter is about that state: what the Lab keeps between commands, how you read it, how you change it, what survives a `:reset`, and — for the reader who came for the LATTICE rather than the ballistics — how a language whose values are passed by value manages to have a mutable session at all.

How transcripts are printed in this book

Every transcript in this chapter was produced by feeding lines to `scripts/ballistics-lab.sh` on standard input and capturing standard output. The Lab reads its prompt through GNU readline, which suppresses the `ballistics>` prompt when standard input is a pipe — so the output blocks below contain no prompts. Where it matters, the lines you would type are shown in their own listing, labelled as input. Nothing has been re-typed, re-aligned, or rounded. The engine used throughout is the `ballistics-engine` binary on `PATH`, which self-reports `0.32.0`. Every `:solve` in this book prints that version back to you, and you should check it against yours before trusting a number.

4.1 One session, seven slots

When the Lab starts, it builds exactly one session and binds it to the top-level name `lab`. Everything the Lab remembers lives inside it.

The session's shape is declared in `session.lat`. There are two structs: a public wrapper and a private state record.

Listing 4.1: The session wrapper and its private state (`session.lat:40-52`)

```
struct LabSessionState {
    experiment: any,
    backend: any,
    display: any,
    history_limit: Int,
    history: Array,
    current_run: any,
    previous_run: any
}

export struct LabSession {
    state: flux any
}
```

Seven slots. That is the whole of the Lab's memory:

`experiment`
: The complete description of the shot — projectile, rifle, atmosphere, winds, shot geometry, solver configuration — as a single frozen value. This is the input to every solve.

`backend`
: How to reach a solver. In the shipped Lab this is a process backend named `process-json-v1` that spawns the `ballistics-engine` executable once per solve.

`display`
: Your unit and precision preferences. These affect rendering only; they never cross the wire.

`history_limit`
: How many past runs to retain.

`history`
: The retained runs, oldest first.

`current_run`
: The most recent successful run.

`previous_run`
: The one before it. This is what `:compare` compares against.

Notice what is *not* there. There is no trajectory cache keyed by distance, no open connection, no child-process handle, no scratch directory, no undo stack. The header comment on `session.lat` is explicit about the process point:

A LabSession owns only Lattice values and backend configuration. In particular, it never retains a child-process handle: process_backend's closure remains process-per-solve.

That single sentence explains a great deal of the Lab's behaviour. The engine is a pure function invoked as a subprocess. It has no session, so the Lab cannot lean on one. Whatever you want remembered, the Lab must remember in LATTICE.

The subprocess bridge, not a library

The design notes in the LATTICE repository sketch a native Rust extension — a dynamic library loaded into the interpreter with a C ABI. That is *not* what shipped. The Lab drives `ballistics-engine` as an ordinary child process through `exec_argv()`, writing one JSON request to the child's standard input and reading one JSON envelope back. The dylib remains an unbuilt alternative. The chapter on the process backend covers the shipped mechanism in full.

4.2 Reading the session: :show

`:show` is the Lab's status line. It takes no arguments and it prints two blocks: the session header, then the active experiment in full.

Start the Lab with no setup at all and ask:

Listing 4.2: Input

```
:show
:quit
```

Listing 4.3: : show on a fresh session (banner elided)

```
Ballistics laboratory session
  active experiment: 175 gr match at 1,000 yards
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none

Experiment: 175 gr match at 1,000 yards
  projectile           : 175 gr match
  mass                 : 175.00 gr
  diameter             : 0.31 in
  length               : 1.24 in
  drag model           : G7
  ballistic coefficient: 0.24
  rifle                : 24-inch match rifle
  sight height         : 1.50 in
  muzzle height        : 0.00 in
  twist rate           : 8.00 in
  twist direction      : right
  altitude             : 273.40 yd
  temperature          : 59.00 F
  pressure             : 29.92 inHg
  humidity             : 50.00 %
  latitude             : +45.00 deg
  muzzle velocity      : 2700.00 ft/s
  maximum range        : 1000.00 yd
  zero distance        : 100.00 yd
  muzzle angle         : default
  aim azimuth          : default
  shot azimuth         : default
  shooting angle       : default
  cant angle           : default
  target height        : default
  ground threshold     : default
  solver               : rk45
  time step            : default
  sampling interval    : 10.00 yd
  Magnus               : default
  Coriolis             : on
  enhanced spin drift  : default
  wind[0]              : 10.00 mph from 90.00 deg
```

The first block is the seven slots, rendered. `history : 0/50` is *retained runs* over *history limit*. `current run` and `previous run` read `none` until you solve.

The second block is the active experiment. Every line of it is a value you can change, and Section 4.4 shows how.

4.2.1 Reading the experiment block

A few lines deserve immediate comment, because they surprise people.

`default` is not a value; it is an absence. `muzzle angle`, `aim azimuth`, `cant angle` and the rest read `default` because the corresponding field is `nil` in the domain value — the Lab did not supply it, so it is simply omitted from the request and the engine picks its own default. Chapter 5 shows what the engine chose: every omitted field comes back as a `default_applied` assumption on the solve.

`Coriolis : on` is not a default. The shipped reference experiment enables Coriolis deliberately, and supplies the `latitude : +45.00 deg` that Coriolis requires. Those two facts are linked by a validation rule you will meet in Section 4.6.

`altitude : 273.40 yd` reads oddly, and it is not a bug. The reference experiment's altitude is 250 m. The display profile's distance unit is yards, and the renderer applies that unit to every distance in the experiment — including the vertical one. 250 m is 273.40 yd. If you would rather read altitude in metres, switch the whole profile to metric (Section 4.7); there is no per-field unit override.

Reading a diameter of `0.31 in`

The reference projectile is a .308, and `:show` prints `0.31 in` because the display precision is two decimals. The stored value is exact — 0.00782 m — and the request that reaches the engine carries all of it. Precision is a rendering choice, not a storage choice. Raise it with `sessions.set_display_preferences` if you need to see more.

4.3 There is no `:set`

Here is the design decision that shapes the rest of this chapter. The Lab's colon-command surface is:

Listing 4.4: `:help`

```
Commands:
:help
:show
:solve
:at <distance> <unit>
:dope <start> <stop> <interval> <unit>
:wind-card <start> <stop> <interval> <unit>
:compare
:save "<path>"
:load "<path>"
:reset
:quit
```

Eleven commands. Not one of them changes the experiment. There is no `:set velocity 2710`, no `:rifle`, no `:wind`.

That is deliberate. The `README` states the reasoning plainly: rifle, projectile, atmosphere, and other experiment changes remain visible ordinary LATTICE calls rather than a second `:set` language. A colon command is a *query* or a *lifecycle* operation. Everything else is programming.

The consequence is that the Lab has one input surface for reading and a different one for writing, and the writing surface is the whole language.

4.4 Mutating the session

Any line that does not begin with `:` at a fresh prompt is LATTICE source, evaluated against the same top-level environment that holds `lab`. The session module is bound as `sessions`, and it exports a family of replacement functions:

Call	Replaces
`sessions.replace_projectile(lab, v)`	the projectile
`sessions.replace_rifle(lab, v)`	the rifle
`sessions.replace_atmosphere(lab, v)`	the atmosphere
`sessions.replace_winds(lab, v)`	the wind array
`sessions.replace_shot(lab, v)`	the shot
`sessions.replace_solver(lab, v)`	the solver configuration
`sessions.replace_experiment(lab, v)`	the whole experiment
`sessions.set_backend(lab, v)`	the backend
`sessions.set_display_preferences(lab, v)`	display units and precision
`sessions.set_history_limit(lab, n)`	the history bound

The constructors that build the values — `projectile`, `rifle`, `atmosphere`, `wind`, `shot`, `solver_config`, `experiment` — are imported unqualified into the REPL environment, as are the unit helpers `gr`, `inches`, `m`, `yd`, `fps`, `mph`, `deg`, `celsius` and `hpa`. You call them by bare name.

Here is a complete rifle setup — a 6.5 mm match load at moderate altitude — typed at the prompt:

Listing 4.5: Input: building a load, component by component

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140), inches(0.264), 0.315, "G7",
    inches(1.34)))
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in", inches(1.8), m(0), inches(8), "right"))
sessions.replace_atmosphere(lab, atmosphere(m(1200), celsius(12), hpa(880), 0.35, deg(44)))
sessions.replace_winds(lab, [wind(mph(8), deg(270))])
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100)))
sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), nil, true, nil))
```

Six lines, six commits. Read them left to right and the argument order is the same story every time: a name, then quantities with their units attached.

Ballistic coefficient (BC) and drag model

A **drag model** is a reference drag curve for a standard projectile shape — `G1` (flat-base, blunt), `G7` (boat-tail, long ogive), and two others the protocol accepts. The **ballistic coefficient** is the single number that scales that reference curve to your bullet: drag enters the equations as C_d/BC. A BC is therefore meaningless without the model it was measured against. `0.315` on `G7` and `0.315` on `G1` describe two very different bullets, and the Lab makes you name both.

Sight height, twist rate, wind direction

Sight height is the vertical distance from the bore axis to the optical axis of the sight — for most scoped bolt rifles, between 1.5 and 2.0 inches. It matters more than beginners expect; Section 5.6 shows exactly how much.
Twist rate is the barrel length per one full rotation of the bullet: `inches(8)` is a 1:8″ twist. It is a distance, and the Lab types it as one.
Wind direction in the Lab, as everywhere in this engine, is a *wind-from* bearing relative to the shot: 0° is a headwind, 90° is wind from the shooter's right, 270° is wind from the left.

4.4.1 The name does not follow

Run `:show` after those six lines and the header says something wrong:

Listing 4.6: : show after replacing every component

```
Ballistics laboratory session
  active experiment: 175 gr match at 1,000 yards
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none

Experiment: 175 gr match at 1,000 yards
  projectile           : 140 gr ELD-M
  mass                 : 140.00 gr
  diameter             : 0.26 in
  length               : 1.34 in
  drag model           : G7
  ballistic coefficient: 0.32
  rifle                : Tikka T3x CTR 24 in
  sight height         : 1.80 in
  muzzle height        : 0.00 in
  twist rate           : 8.00 in
  twist direction      : right
  altitude             : 1312.34 yd
  temperature          : 53.60 F
  pressure             : 25.99 inHg
  humidity             : 35.00 %
  latitude             : +44.00 deg
  muzzle velocity      : 2710.00 ft/s
  maximum range        : 1200.00 yd
  zero distance        : 100.00 yd
  muzzle angle         : default
  aim azimuth          : default
  shot azimuth         : default
  shooting angle       : default
  cant angle           : default
  target height        : default
  ground threshold     : default
  solver               : rk45
  time step            : default
  sampling interval    : 25.00 yd
  Magnus               : default
  Coriolis             : on
  enhanced spin drift  : default
  wind[0]              : 8.00 mph from 270.00 deg
```

Every component changed. The name did not. It still says `175 gr match at 1,000 yards` while describing a 140 gr ELD-M at 1,200 yards.

A notebook built on `replace_*` will mislabel every run

The name lives on the `Experiment`, and each `replace_<component>` rebuilds the experiment from the current one, carrying `current.name` forward unchanged (`session.lat:285`). Nothing in the partial-replacement path can rename anything.

This is not cosmetic. The name is printed on every `:solve` header, on every DOPE table title, and in every `Provenance` block — and it is written into every artifact you save. A range notebook full of identically-mislabelled tables is worse than no notebook.

The fix is a whole-experiment replacement, reusing the components you just set:

Listing 4.7: Input: renaming by rebuilding

```
let cur = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("140 gr ELD-M at 1,200 yd", cur.projectile,
    cur.rifle, cur.atmosphere, cur.winds, cur.shot, cur.solver))
```

Listing 4.8: `:show` head after the rename

```
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none

Experiment: 140 gr ELD-M at 1,200 yd
  projectile           : 140 gr ELD-M
```

Those eight lines — six replacements and a rename — are the setup used by every transcript in the rest of this chapter and the next. When a later listing says "after the setup," this is the setup.

Name first, then edit

The habit that avoids the trap entirely: name the experiment *before* you tune it, and rename it whenever the load changes materially. A rename is one `replace_experiment` call, and you can keep a one-liner for it in your notes.

4.5 Every write is revalidated

The Lab does not trust the values you hand it, even though you built them with its own constructors. Every `replace_*` runs its argument back through the domain constructor before storing it, and then rebuilds the entire `Experiment` so that relational rules re-run.

That second half matters. Some rules in this domain are not about a single field:

- `zero_distance` and `muzzle_angle` are mutually exclusive on a shot — you either tell the engine where the rifle is zeroed or you tell it the launch angle, never both.
- Coriolis requires a latitude on the atmosphere.
- Magnus and enhanced spin drift both require a projectile length, and cannot both be enabled.
- A wind array is either exactly one constant wind or a fully segmented list with strictly increasing boundaries. Mixing the two forms is rejected.
- The ground threshold must sit below the muzzle height.

None of these can be checked by looking at the value you passed. They are checked because the whole experiment is reassembled on every partial edit.

4.6 A failed edit changes nothing

This is the invariant that makes the Lab usable as a laboratory: **a mutation either fully succeeds or leaves the session exactly as it was.** There is no half-applied edit.

The mechanism is construct-then-publish. Each mutator validates, builds a complete next state, and publishes it with a single `Ref.set`. If any step throws, the `set` never happens.

Watch it work. Starting from the setup, we attempt five illegal edits and read the session back after each one:

Listing 4.9: Input: five rejected edits

```
sessions.experiment_of(lab).atmosphere.latitude.display_value
sessions.replace_atmosphere(lab, atmosphere(m(1200), celsius(12), hpa(880), 0.35))
sessions.experiment_of(lab).atmosphere.latitude.display_value
sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), true, true, true))
sessions.experiment_of(lab).solver.sampling_interval.display_value
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100), deg(1)))
sessions.experiment_of(lab).shot.zero_distance.display_value
sessions.replace_winds(lab, [wind(mph(8), deg(270), nil, yd(500)), wind(mph(12), deg(90))])
len(sessions.experiment_of(lab).winds)
sessions.replace_winds(lab, [wind(mph(8), deg(270), nil, yd(500)), wind(mph(12), deg(90), nil,
     yd(1300))])
len(sessions.experiment_of(lab).winds)
sessions.experiment_of(lab).winds[1].speed.display_value
```

Listing 4.10: Output: every rejection leaves prior state intact

```
44
Error
  code: evaluation_error
  message: assertion failed: experiment.atmosphere.latitude: required when Coriolis is enabled
44
Error
  code: evaluation_error
  message: assertion failed: effects: magnus and enhanced_spin_drift cannot both be enabled
25
Error
  code: evaluation_error
  message: assertion failed: shot: zero_distance and muzzle_angle cannot both be supplied
100
Error
  code: evaluation_error
  message: assertion failed: experiment.winds: cannot mix constant and segmented wind
1
2
12
```

Read it line by line:

- Latitude is 44°. We try to replace the atmosphere with one that has no latitude while Coriolis is on. Rejected — and latitude is *still* 44°. The old atmosphere was never removed.

- We try a solver with Magnus *and* enhanced spin drift. Rejected; the sampling interval is still 25 yd, so the old solver survived whole.
- We try a shot with both a zero distance and a muzzle angle. Rejected; the zero is still 100 yd.
- We try a wind array that mixes one segmented wind with one constant wind. Rejected; the array still has exactly the one wind we set during setup.
- The same array with a boundary on *both* winds is accepted: two winds, the second at 12 mph.

Errors are values, printed as data

Every rejection above printed a three-line `Error` block rather than killing the REPL. That is not exception handling bolted on at the top; it is the Lab's error model. Validation inside a module throws; the module seams — backend, command dispatch, session — catch and normalize to a `LaboratoryError` value with a code, a message, and often a JSON path. The REPL renders it and returns to the prompt. You can run a hundred bad commands in a row and the process still exits zero.

Shape errors are caught the same way. Handing a session slot the wrong type never corrupts it:

Listing 4.11: Output: type rejections

```
Error
  code: evaluation_error
  message: assertion failed: session.experiment.shot: expected Shot, got Int
Error
  code: evaluation_error
  message: assertion failed: session.experiment: expected Experiment, got String
Error
  code: evaluation_error
  message: assertion failed: display.distance_unit: unsupported value 'furlongs'; expected one
     of [m, km, yd, ft, in]
Error
  code: evaluation_error
  message: assertion failed: session.history_limit: must be non-negative
```

4.7 Display preferences

The `display` slot holds five fields: a distance unit, a velocity unit, an energy unit, an angle unit, and a decimal precision. It is the only part of the session that exists purely for your eyes. Nothing in it reaches the engine — the request is canonical SI, always.

The defaults are imperial:

Field	Default	Accepted
`distance_unit`	`yd`	`m`, `km`, `yd`, `ft`, `in`
`velocity_unit`	`ft/s`	`m/s`, `ft/s`
`energy_unit`	`ft-lb`	`J`, `ft-lb`
`angle_unit`	`mil`	`rad`, `deg`, `mil`, `moa`
`precision`	`2`	integer, 0–12

The setter is `sessions.set_display_preferences`. It takes a value built by one of two constructors, as follows:

Listing 4.12: Input: three profiles over one run

```
sessions.set_display_preferences(lab, sessions.display_preferences("m", "m/s", "J", "mil", 3))
:solve
sessions.set_display_preferences(lab, sessions.display_preferences("yd", "ft/s", "ft-lb", "moa",
    1))
:at 600 yd
sessions.set_display_preferences(lab, sessions.metric_display_preferences(2))
:at 600 yd
sessions.set_display_preferences(lab, sessions.display_preferences())
:at 600 yd
```

The solve, in metric at three decimals:

Listing 4.13: `:solve` rendered metric, precision 3 (notices elided)

```
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1097.280 m
  maximum height   : +0.047 m
  time of flight   : 1.851 s
  terminal velocity: 425.546 m/s
  terminal energy  : 821.408 J
  stability factor : 2.062
  spin drift       : not reported
```

Then the same observation — one solve, three renderings:

Listing 4.14: One trajectory, three display profiles

```
Observation
  distance: 600.0 yd
  time    : 0.8 s
  velocity: 1997.9 ft/s
  energy  : 1240.6 ft-lb
  drop    : +81.1 in
  windage : +16.3 in
  Mach    : 1.8
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 548.64 m
  time    : 0.77 s
  velocity: 608.96 m/s
  energy  : 1682.07 J
  drop    : +2.06 m
  windage : +0.41 m
  Mach    : 1.80
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 600.00 yd
  time    : 0.77 s
  velocity: 1997.90 ft/s
  energy  : 1240.63 ft-lb
  drop    : +81.08 in
  windage : +16.27 in
  Mach    : 1.80
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Three things to take from that:

Offsets get their own unit. `drop` and `windage` are not rendered in the distance unit. When the distance family is imperial they render in inches; when it is metric they render in metres. A drop expressed in yards would be useless, so the renderer derives an *offset unit* from the profile: `in` for `yd`/`ft`/`in`, `m` for `m`/`km`.

The query is converted, not preserved. We asked for `:at 600 yd` in every case. In the metric profile the answer came back as `distance: 548.64 m`. The command's unit says where to look; the display profile says how to print it.

Changing the profile does not re-solve. The second and third observations are the same numbers rendered differently — no child process ran. Display is downstream of everything.

Precision 1 is a lie detector

Dropping to `precision 1` prints `Mach : 1.8` and `time : 0.8 s`. Nothing was recomputed and nothing was lost; the stored values are still full doubles. If a number *changes* when you raise precision, you learned something about the number. If it changes when you change units, you learned something about the conversion.

4.8 History, current, and previous

Three of the seven slots track runs. They are updated by exactly one operation — a successful `:solve` — and cleared by exactly one — a `:reset`.

`current_run` and `previous_run` are a two-deep shift register: on each successful solve, the old current becomes previous and the new record becomes current. `history` is a bounded list, oldest first, trimmed from the front when it exceeds `history_limit`.

Listing 4.15: Input: a bounded history

```
sessions.set_history_limit(lab, 2)
:solve
:solve
:solve
len(sessions.history(lab))
sessions.history_limit_of(lab)
let h = sessions.history(lab)
struct_name(h[0])
phase_of(h[0])
h[0].backend_name
h[0].engine_version
h[0].termination
h[0].schema_version
len(h[0].result.observations)
typeof(h[0].resolved_request)
```

Listing 4.16: Output (three `:solve` blocks elided)

```
2
2
RunRecord
unphased
process-json-v1
0.32.0
max_range
1
49
Map
```

Three solves, a limit of two, two records retained. Each record is a `RunRecord` carrying nine fields: the backend's name, the experiment snapshot that was submitted, the full `Trajectory` result, the engine's resolved request, the schema version, the engine version, the assumptions, the warnings, and the termination reason.

`len(h[0].result.observations)` is 49 — the 1,200 yd trajectory sampled every 25 yd, endpoints included. Every one of those observations is still in memory, which is why the history bound exists.

Why `history_limit` is worth setting

A run record holds a whole trajectory. At the protocol's 10,000-sample ceiling, fifty retained runs is half a million observations. The default limit of 50 is generous for interactive work and wasteful for a scripted sweep. If you are solving in a loop, set the limit to what you will actually read.

Setting the limit to zero is a documented special case: history is disabled, but current and previous run tracking continues.

Listing 4.17: Input: `history_limit` of zero

```
sessions.set_history_limit(lab, 0)
len(sessions.history(lab))
:solve
len(sessions.history(lab))
sessions.current_run(lab).engine_version
```

Listing 4.18: Output: no history, but :compare still works

```
0
0
0.32.0
```

That matters because :compare reads current_run and previous_run, not history. You can compare two loads with the history bound at zero.

4.8.1 Reading a run from LATTICE

Everything :show summarizes is reachable as data. The accessors are plain functions:

Listing 4.19: Session accessors

```
sessions.experiment_of(lab)      // the active Experiment
sessions.backend_of(lab)         // the EngineBackend
sessions.display_of(lab)         // the DisplayPreferences
sessions.history_limit_of(lab)   // Int
sessions.history(lab)            // Array of RunRecord, oldest first
sessions.current_run(lab)        // RunRecord or nil
sessions.previous_run(lab)       // RunRecord or nil
```

They return values, not handles. Holding onto `let r = sessions.current_run(lab)` and then solving again does not change r; you are looking at the run you captured.

4.9 The backend slot

The backend slot holds an EngineBackend: a name and one closure that takes an experiment and returns a trajectory or an error. That is the whole interface.

Listing 4.20: Input: inspecting and replacing the backend

```
sessions.backend_of(lab).name
sessions.set_backend(lab, backends.engine_backend("offline-stub", |experiment| {
    return laboratory_error("offline", "no engine configured for this notebook")
}))
sessions.backend_of(lab).name
:solve
```

Listing 4.21: Output

```
process-json-v1
offline-stub
Error
  code: offline
  message: no engine configured for this notebook
```

The stub's error travelled all the way out to the prompt unchanged. That is the seam contract: a backend may return a `Trajectory` or a `LaboratoryError`, and the session passes either through untouched.

Return anything else and the seam says so:

Listing 4.22: Output: a backend that breaks its contract

```
Error
  code: backend_contract_error
  message: backend 'bad-shape' returned Int; expected Trajectory or LaboratoryError
```

Two uses for a fake backend

Swapping the backend is not a party trick. It is how the Lab's own test suite runs without an engine, and it is how you can draft a notebook on a laptop with no `ballistics-engine` installed — write the setup and the analysis against a stub, then swap in `process.process_backend("ballistics")` when you are at a machine that has one. The chapter on testing builds several such doubles.

Swapping the backend does *not* clear the run slots. A trajectory solved by the real engine remains `current_run` after you install a stub, which is worth remembering before you read a table.

4.10 :reset — exactly what it clears

`:reset` is the one lifecycle command that touches state, and its scope is narrower than the name suggests. The response string is precise: `Session run state reset.` — run state, not session.

Two solves, then `:show`, then `:reset`, then `:show` again:

Listing 4.23: Before `:reset`

```
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 2/50
  current run      : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
  previous run     : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
```

Listing 4.24: After `:reset`

```
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none
```

The full picture, verified slot by slot:

Slot	Cleared by `:reset`?	After
`history`	yes	empty
`current_run`	yes	`nil`
`previous_run`	yes	`nil`
`experiment`	**no**	unchanged, including the name
`backend`	**no**	unchanged
`display`	**no**	unchanged
`history_limit`	**no**	unchanged

`:reset` does not restore the shipped reference experiment, and it does not undo your edits. If you mutated the muzzle velocity to 3100 ft/s and then reset, the session still holds 3100 ft/s — with no run to go with it. To get back to a known experiment, save one and `:load` it, or restart the Lab.

4.11 The stale-run trap

Here is the sharpest edge in the session model, and it is worth a page.

`:load` replaces the experiment. It does not touch the run slots. So after a load, `:show` describes one rifle while `:at`, `:dope`, `:wind-card` and `:compare` answer from another.

Watch it happen. Solve at 2710 ft/s, save that experiment, mutate the muzzle velocity to 3300 ft/s, solve again, then load the saved file back. (The transcript used an absolute path in a scratch directory; it is shortened here. `:save` echoes back the full path it wrote.)

Listing 4.25: Input: save, mutate, solve, load

```
:solve
:save "/tmp/eld-m.json"
sessions.replace_shot(lab, shot(fps(3300), yd(1200), yd(100)))
:solve
:load "/tmp/eld-m.json"
:show
:at 1000 yd
```

`:show` now reports the loaded experiment, and it is correct:

Listing 4.26: `:show` after the load (excerpt)

```
  history          : 2/50
  current run      : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
  previous run     : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
Experiment: 140 gr ELD-M at 1,200 yd
  muzzle velocity      : 2710.00 ft/s
```

And `:at` answers from the 3300 ft/s run that is still sitting in `current_run`:

Listing 4.27: `:at 1000 yd` after the load — the wrong trajectory

```
Observation
  distance: 1000.00 yd
  time    : 1.16 s
  velocity: 2038.59 ft/s
  energy  : 1291.68 ft-lb
  drop    : +188.91 in
  windage : +36.88 in
  Mach    : 1.83
  flags   : none
```

The true answer for the loaded 2710 ft/s experiment is `1586.32 ft/s` at 1000 yd. We got 2038.59 ft/s — the faster load's trajectory, under the slower load's name.

`:load` does not clear run state

After a `:load`, every analysis command reports the *previous* experiment's trajectory until you solve again. There is no warning, because nothing failed: the session correctly holds a new experiment and an old run, and each command reads the slot it is documented to read.
The safe idiom is three commands, in this order:

`:load "..."` → `:reset` → `:solve`

The same caution applies to every `replace_*` call: a session that has been edited but not re-solved will answer analysis queries from the stale run.

Re-solving fixes it, and you can see the fix in the same session: after `:reset` and `:solve`, `:at 1000 yd` returns `velocity: 1586.32 ft/s`.

A one-line habit

Make `:solve` the last thing you type before you read any table. It costs about 40 milliseconds and it removes an entire class of silent error.

4.12 Phases: why this works at all

This section is for the LATTICE reader. If you came for the ballistics, the practical summary is one sentence: the experiment and every solved trajectory are immutable, and the only mutable thing in the entire Lab is a single reference cell.

Phase, in one paragraph

LATTICE values carry a **phase**. A **fluid** value is ordinary and mutable. A **crystal** value has been `freeze`d: it is deeply immutable and lives in a shared region, so passing it around costs nothing. **unphased** means neither has been applied. A struct field declared `fix` cannot be reassigned even when the struct itself is unphased — what the Lab's sources call an *alloy*: immutability without the cost of crystallizing an entire graph.

Ask the running Lab what phase everything is in:

Listing 4.28: Input: phase inspection

```
typeof(lab)
struct_name(lab)
typeof(lab.state)
phase_of(lab)
let st = lab.state.get()
struct_name(st)
phase_of(st)
phase_of(sessions.experiment_of(lab))
phase_of(sessions.display_of(lab))
typeof(sessions.backend_of(lab))
struct_name(sessions.backend_of(lab))
sessions.backend_of(lab).name
typeof(sessions.backend_of(lab).solve_fn)
```

Listing 4.29: Output

```
Struct
LabSession
Ref
unphased
LabSessionState
unphased
crystal
crystal
Struct
EngineBackend
process-json-v1
Closure
```

And after a solve, five more questions:

Listing 4.30: Input: phases of a completed run

```
phase_of(sessions.current_run(lab).result)
phase_of(sessions.current_run(lab).result.observations)
phase_of(sessions.current_run(lab).resolved_request)
phase_of(sessions.current_run(lab))
sessions.current_run(lab).result.experiment == sessions.experiment_of(lab)
```

Listing 4.31: Output

```
crystal
crystal
crystal
unphased
true
```

Put together, the picture is:

- The `LabSession` wrapper is **unphased** and holds a `Ref`. It *must* be: freezing the wrapper would freeze the `Ref` and disable `Ref.set()`, which is why the field is declared `flux` rather than `fix`.
- The state record inside is **unphased** too — it is replaced wholesale on every commit, never edited in place.
- Everything meaningful inside it is **crystal**: the experiment, the display preferences, the solved trajectory, its observations, the resolved request.
- The `RunRecord` and `EngineBackend` wrappers are deliberately *not* frozen. Their fields are `fix`, which gives immutability without copying an already-frozen 10,000-sample graph into another crystal region. The source says so:

> *RunRecord is an immutable alloy wrapper. Its fields are `fix`, and every composite payload placed in it is already frozen by the domain/backend boundary. Avoid freezing the outer wrapper again: a trajectory may contain 10,000 samples, and alloy fields give us immutability without copying that already-frozen graph into another crystal region.*

The thesis statement for the whole design sits a few lines below it:

> *Lattice values are passed by value. Keep the complete mutable state behind one Ref so module operations update every alias and can publish a prepared transition with one set(). The state struct is private; exported accessors return its current values. The public wrapper field must remain `flux`: a `fix` field freezes the Ref and disables Ref.set().*

That first sentence is the constraint everything else answers. LATTICE passes values by value; if the session were an ordinary struct, then `sessions.replace_shot(lab, ...)` would mutate a copy and your `lab` would never change. One `Ref` gives every alias of the session a view of the same cell, and one `set` per operation makes the transition atomic.

Why the identity check passes

`current_run(lab).result.experiment == sessions.experiment_of(lab)` returned `true` in the transcript above. That is not a coincidence — it is enforced. Before building a `RunRecord`, the session verifies that the trajectory the backend returned still carries the exact experiment snapshot that was submitted. A backend that quietly solved something else produces a `backend_contract_error` instead of a run. The provenance printed on every table is therefore load-bearing, not decorative.

4.13 What the session does not do

Three non-goals, stated plainly, because each one saves a reader from looking for a feature that is not there.

No autosave. The session lives in memory. Quit the Lab and the experiment, the history, and every solved trajectory are gone. `:save` writes the *experiment* — not the runs, not the history, not your display preferences. The chapter on persistence covers exactly what a saved document contains and what it deliberately omits.

No undo. Construct-then-publish guarantees that a *failed* edit changes nothing. It offers no protection against a successful edit you did not mean. The retained history holds past *runs*, not past experiments; there is no way to walk backwards through your edits. If a load matters, save it.

No engine state. The engine is invoked once per solve, with the entire problem on standard input, and it exits. There is no warm cache, no session token, no incremental refinement. Two identical solves cost two child processes and return identical answers — which is precisely why the Lab's output is reproducible across backends and across machines.

A session checklist

Before you trust a table, ask four questions of `:show`: is the *name* right; is the *experiment* the one you meant; is there a *current run*; and was that run solved *after* your last edit? The first three are printed. The fourth is the one the Lab cannot tell you — which is why Section 4.11 exists.

With the session model in hand, we can turn to the thing it exists to hold: a solved trajectory, and the two commands that produce and interrogate one.

Chapter 5

Solving and Sampling

Two commands do the work in the Ballistics Lab. `:solve` computes a trajectory. `:at` asks that trajectory a question.

Everything else in the Lab — DOPE tables, wind cards, comparisons, saved artifacts — is built on those two operations, so it is worth understanding them precisely: what a solve actually computes, what each line of its summary means, and exactly what number `:at` hands back when you ask for a distance.

This chapter also settles the question that governs every vertical number the Lab prints: what plane is drop measured from, and where does the zero put the bullet. That is Section 5.6.

5.1 What a solve is

A solve is one child process.

The Lab takes the session's current experiment, converts it to a canonical SI JSON request, spawns the `ballistics-engine` executable with the fixed argument list `["solve-json"]`, writes the request to the child's standard input, and reads exactly one JSON envelope back from its standard output. The child then exits. No shell is involved, no string is interpolated into a command line, and nothing is cached between solves.

Listing 5.1: What the Lab does, in shell terms

```
ballistics solve-json < request.json > response.json
```

The whole round trip is cheap. Twenty consecutive `:solve` calls on the 1,200 yd load from Chapter 4 added 0.41 s of wall clock over an otherwise identical session — about 20 ms each, including process spawn, request serialization, integration, and the decode and validation of 49 sample records.

Process-per-solve is a feature

Spawning a process per solve sounds wasteful until you consider what it buys: the engine cannot accumulate state, so two identical experiments cannot produce different answers; a crashed or hung solve cannot corrupt the session; and the Lab needs no cleanup path, no connection pool, and no cache-invalidation rule. The 20 ms is the price of a system with no invisible memory.

5.1.1 The solve transaction

A solve is also a transaction against the session. In order:

1. Take an immutable snapshot of the current experiment.
2. Invoke the backend exactly once.
3. If it failed, return the error and *touch nothing*.
4. Verify that the returned trajectory still carries the submitted experiment, and that it carries a resolved-request map.
5. Build a `RunRecord`.
6. Trim history to its bound.
7. Publish the whole next state with a single write.

The practical consequence: a failed solve leaves your previous run exactly where it was. You can experiment freely; a rejected or broken solve never costs you the last good answer.

5.2 Reading the run summary

Here is a complete `:solve`, using the 140 gr ELD-M setup from Section 4.4: 2710 ft/s, 100 yd zero, 1,200 yd maximum range, 8 mph wind from 270°, Coriolis on, 25 yd sampling.

Listing 5.2: `:solve`

```
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1200.00 yd
  maximum height   : +1.85 in
  time of flight   : 1.85 s
  terminal velocity: 1396.15 ft/s
  terminal energy  : 605.84 ft-lb
  stability factor : 2.06
  spin drift       : not reported
Assumptions
  - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
  - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
  - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
  - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
  - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
  - default_applied: Constant vertical wind speed defaulted to 0 m/s. [$.wind.vertical_speed_mps]
  - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
  - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
  - default_applied: Enhanced spin drift defaulted to disabled. [$.effects.enhanced_spin_drift]
Warnings
  none
```

Twelve summary fields, then two notice blocks. Take them in turn.

5.2.1 Provenance: backend, engine, schema, verified

`backend : process-json-v1` names the seam the answer came through. If you swapped in a stub (Section 4.9) this line changes, and that is the point — every number in the Lab is stamped with where it came from.

`engine : 0.32.0` is the version the child process reported, not a version the Lab assumed. If you pin `BALLISTICS_ENGINE_VERSION` in the environment and the child disagrees, the solve fails with `incompatible_engine_version` rather than quietly producing numbers from a different solver.

`schema : 1` is the protocol version, and it is checked exactly: an envelope claiming any other schema is rejected.

`verified : no` is the normal state for a fresh run and does not mean anything is wrong. "Verified" in the Lab's vocabulary is a specific, deliberate act: promoting a run to a portable, fully detached

`VerifiedRun` artifact with cross-checked provenance (Chapter 24). Nothing that comes straight off the wire is verified.

5.2.2 Termination: did you get the range you asked for?

`termination` is the most important field in the block, and the one most often skipped. It tells you *why the integration stopped*, and it takes one of four values:

Value	Meaning
`max_range`	The bullet reached the range you asked for.
`ground_threshold`	It hit the ground first.
`time_limit`	The integrator ran out of time budget.
`velocity_floor`	The bullet slowed below the solver's floor.

Read it together with `actual range`. When they agree with your requested maximum range, you have the trajectory you asked for. When they do not, the engine did not fail — it answered a shorter question.

Two verified examples. First, asking the same load for 3,000 yd:

Listing 5.3: Input

```
sessions.replace_shot(lab, shot(fps(2710), yd(3000), yd(100)))
:solve
```

Listing 5.4: The bullet lands before 3,000 yd (notices elided)

```
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : ground_threshold
  actual range     : 2411.26 yd
  maximum height   : +1.85 in
  time of flight   : 5.39 s
  terminal velocity: 880.37 ft/s
  terminal energy  : 240.89 ft-lb
  stability factor : 2.06
  spin drift       : not reported
```

No error, no warning: `termination : ground_threshold` and `actual range : 2411.26 yd`. With a flat 100 yd zero the bullet reaches the ground threshold at 2,411 yd, and the solve stops there. Ask that run for a distance past the end and it says so plainly:

Listing 5.5: `:at 2500 yd` on a run that terminated at 2411.26 yd

```
Error
  code: command_execution_error
  message: at: assertion failed: trajectory_at.distance: 2286 m is outside computed trajectory
     [0, 2204.86] m
```

Second, an absurdly slow shot:

Listing 5.6: Input

```
sessions.replace_shot(lab, shot(fps(150), yd(1200), yd(100)))
:solve
```

Listing 5.7: 150 ft/s, 100 yd zero (notices elided)

```
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : ground_threshold
  actual range     : 270.15 yd
  maximum height   : +208.44 in
  time of flight   : 5.71 s
  terminal velocity: 200.48 ft/s
  terminal energy  : 12.49 ft-lb
  stability factor : 0.79
  spin drift       : not reported
```

Two things worth noticing. The zero solution had to elevate the bore enormously to put a 150 ft/s projectile on target at 100 yd, so the apex is 208 inches — 17 feet — and the terminal velocity is *higher* than the muzzle velocity, because the bullet spent five seconds accelerating downward. And `stability factor : 0.79` is below 1.0: this projectile is not gyroscopically stable at that speed. The engine still solves it; interpreting the result is your job.

`termination` is the field to check first

`ground_threshold` is not an error, and nothing else in the output flags it. If you build a DOPE card without reading `termination`, and your load ran out of trajectory at 900 yd, the table simply stops — and it stops at a distance that looks like a deliberate choice.

5.2.3 The geometric fields

`actual range` is the downrange distance of the last computed sample, rendered in the display distance unit. It equals your requested maximum range only when `termination` is `max_range`.

`maximum height` is the apex of the trajectory measured from the horizontal plane through the *muzzle* — not the height above the ground, and not the height above the line of sight. For the reference 1,200 yd shot it reads `+1.85 in`, and the rifle's sight height is 1.8 inches. That is not a coincidence: with a 100 yd zero the bullet peaks a few hundredths of an inch above the line of sight, and the line of sight is itself 1.8 inches above the muzzle.

Move the zero to 300 yd and the same load reports `maximum height : +7.13 in`. Raise the scope to a 3.0 inch mount and keep the 100 yd zero and it reports `+3.03 in`. The number is a property of the zero and the mount, not of the range.

We can verify the definition exactly. Solving the shipped reference experiment — a 1.5 inch sight height, zeroed at 100 yd — with a 0.5 m sampling interval gives 1,830 samples, and:

Listing 5.8: Checking the definition of `maximum_height_m`

```
sight_height_m           = 0.038099999999999995
maximum_height_m         = 0.041117543694811437
min drop_m               = -0.0030150878780781862
sight_height - min_drop  = 0.041115087878078181
difference               = 2.4558167332558445e-06
```

The apex equals the sight height minus the smallest drop, to within 2.5 micrometres — the residual being the gap between the true minimum and the nearest 0.5 m sample. That identity is exactly what "height above the muzzle plane" means once you know that drop is measured downward from the sight plane, which sits one sight height higher. Note also that the smallest drop is *negative*: somewhere in the mid-range this bullet is 3 mm above the line of sight. Section 5.6 takes the datum apart.

`time of flight` is seconds from muzzle to the terminal sample. It is always printed in seconds regardless of the display profile, and it is the quantity that drives every wind and lead calculation — drift is not a function of distance, it is a function of *lag time*.

5.2.4 Terminal velocity and terminal energy

Both are properties of the last sample, and both are exactly reproduced there: the decoder refuses an envelope whose terminal sample disagrees with its own summary, using a tolerance of 10^{-10} absolute plus 10^{-9} relative.

Kinetic energy, and why it is quoted

Energy is $\frac{1}{2}mv^2$ — 605.84 ft-lb at 1,200 yd for our 140 gr bullet at 1396 ft/s. Energy is the usual proxy for terminal effectiveness, and many jurisdictions and organizations set minimum-energy thresholds at the target. Note that it falls much faster than velocity does: between the muzzle (2710 ft/s, 2282.62 ft-lb) and 1,200 yd the bullet keeps 52% of its speed and 27% of its energy.

5.2.5 Stability factor and spin drift

`stability factor` is the gyroscopic stability factor S_g — the Miller estimate, computed from bullet mass, diameter, length, twist rate and muzzle conditions. The conventional reading: $S_g < 1.0$ is unstable, $1.0 < S_g < 1.4$ is marginal, $S_g \geq 1.5$ is comfortable. Our load reports `2.06`, which is a fast twist for a 140 gr bullet and deliberately so.

`spin drift : not reported` is the normal state. The engine only reports a spin-drift figure when you have opted into the enhanced spin drift model, which is both optional and marked experimental in the protocol. Turn it on — it requires a projectile length, which our load has — and the field appears, together with a warning:

Listing 5.9: Input

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, true, true))
sessions.replace_shot(lab, shot(fps(2710), yd(2000), yd(100)))
:solve
```

Listing 5.10: Enhanced spin drift enabled (assumptions elided)

```
Run: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 2000.00 yd
  maximum height   : +1.85 in
  time of flight   : 4.03 s
  terminal velocity: 960.20 ft/s
  terminal energy  : 286.56 ft-lb
  stability factor : 2.06
  spin drift       : +52.35 in
Warnings
  - experimental_effect: The enhanced spin drift effect is an opt-in experimental v1 model.
    [$.effects.enhanced_spin_drift]
```

Spin drift is already inside `windage`

`spin drift : +52.35 in` is a *diagnostic*. It is not a correction you add to the windage column — the lateral offset it describes is already folded into every sample's `windage`. Adding it a second time doubles a real effect. The number is there so you can see how much of your lateral hold is wind and how much is the bullet's own rotation.

5.3 Assumptions and warnings

Every solve returns two lists of notices. Both are printed under the summary, both are stored on the run record, and both are reproduced in the `Provenance` block of every table the Lab renders.

Assumptions record choices the engine made because you did not. Our fully-specified experiment still produced nine of them, and that is normal: the solve-json request omits every field the Lab does not set, and the engine reports each default it applied, with the exact JSON path it applied it to.

Read down that list and it is a precise statement of what was *not* modelled: no aim azimuth, no shot azimuth, level ground, no cant, a ground threshold 100 m below the muzzle, no vertical wind, a 0.001 s solver time step, no Magnus, no enhanced spin drift.

There is a tenth line the engine emits when it applies its own default to `$.shot.target_height_m`, and its absence here is deliberate: the Lab always sends that field. Section 5.6 explains what it is and why it governs every drop number in the chapter.

The assumptions list is a checklist

If a solve does not match observed impacts, read the assumption list before you reach for the BC. Two entries on it — `shooting_angle_rad` and `cant_angle_rad` — describe things people routinely get wrong at the rifle, and both default to zero.

Warnings record things the engine did but is not fully confident about — the experimental spin-drift model above being the canonical example. An empty warnings list prints `none`, and that is what you normally want to see.

5.4 From the wire to the summary

Every line of the summary comes from a named field in the response envelope. Here is the whole map, taken from the protocol decoder.

Summary line	Wire field	SI unit
`actual range`	`actual_range_m`	metres
`maximum height`	`maximum_height_m`	metres
`time of flight`	`time_of_flight_s`	seconds
`terminal velocity`	`terminal_speed_mps`	m/s
`terminal energy`	`terminal_energy_j`	joules
`termination`	`termination`	—
`stability factor`	`stability_factor`	dimensionless, optional
`spin drift`	`spin_drift_m`	metres, optional

And each sampled observation is exactly eight fields:

Listing 5.11: One sample from a real response envelope

```
{
  "distance_m": 914.4,
  "time_s": 1.7064080077984438,
  "speed_mps": 349.2875788381682,
  "energy_j": 691.7386422597805,
  "drop_m": 9.97984245540283,
  "windage_m": -2.591700671089823,
  "mach": 1.0250616961776007,
  "flags": ["transonic", "terminal"]
}
```

That is a real terminal sample from the shipped reference experiment — a 175 gr G7 match load at 1,000 yd, wind from 90°. Two details already visible: `drop_m` is positive, and `windage_m` is negative for a wind from the shooter's right. Both are about to matter.

Only SI crosses the wire

There are no display units anywhere in that JSON. The request carries `muzzle_velocity_mps`, `max_range_m`, `temperature_k`, `pressure_pa`, `direction_from_rad`; the response carries metres, seconds, m/s and joules. Yards, inches, mils and foot-pounds exist only in the Lab, on the way in and on the way out. The chapter on units and quantities covers the types that make that separation enforceable.

5.5 `:at` — asking the trajectory a question

`:at` takes two arguments, a number and a unit, and reports the state of the bullet at that downrange distance.

Listing 5.12: Grammar

```
:at <distance> <unit>
```

The unit must be one of `m`, `km`, `yd`, `ft`, `in`. The distance must be non-negative. There is no default unit — you always say which one you mean.

Listing 5.13: Input

```
:solve
:at 100 yd
:at 600 yd
:at 1000 yd
:at 500 m
```

Listing 5.14: Four queries against one solve

```
Observation
  distance: 100.00 yd
  time    : 0.11 s
  velocity: 2582.80 ft/s
  energy  : 2073.37 ft-lb
  drop    : +0.00 in
  windage : +0.40 in
  Mach    : 2.32
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 600.00 yd
  time    : 0.77 s
  velocity: 1997.90 ft/s
  energy  : 1240.63 ft-lb
  drop    : +81.08 in
  windage : +16.27 in
  Mach    : 1.80
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +50.43 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 546.81 yd
  time    : 0.70 s
  velocity: 2056.04 ft/s
  energy  : 1313.91 ft-lb
  drop    : +63.91 in
  windage : +13.33 in
  Mach    : 1.85
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

5.5.1 The unit on the command is a query unit, not a display unit

Look at the last block. We asked `:at 500 m` and the Lab answered `distance: 546.81 yd`.

That is correct and worth internalizing. The unit on the command says *where to look*; the session's display profile says *how to print the answer*. 500 m is 546.81 yd, and the active profile prints distances in yards. If you want the echo in metres, switch the profile (Section 4.7); the query itself is unaffected either way, because internally it is 500 m in both cases.

5.5.2 The observation fields

`distance`
: The queried distance, echoed in the display unit.

`time`
: Time of flight to that distance, in seconds.

`velocity`
: Remaining speed. Note this is speed, not the downrange component — the engine's `speed_mps`.

`energy`
: Remaining kinetic energy.

`drop`
: Vertical offset from the line of sight, positive *downward* — the bullet is that far *below* where you are looking. Section 5.6.

`windage`
: Horizontal offset, positive to the shooter's right.

`Mach`
: Speed as a fraction of the local speed of sound, which the engine computes from the moist-air conditions you supplied — not from a fixed 1,125 ft/s.

`flags`
: Regime markers on the sample. Section 5.8.

Why Mach, and not just velocity

Drag is not a smooth function of speed. It rises steeply through the transonic region — roughly Mach 1.2 down to Mach 0.8 — where the shockwave structure around the bullet reorganizes, and a projectile that is marginally stable can become unstable there. The Mach column tells you where in that story your bullet is at a given range, which velocity alone cannot, because the speed of sound depends on the air you are shooting through.

5.6 Drop: what the number actually means

This is the section to read twice.

Every `:at` block ends with a legend:

```
Signs: drop + below/- above; windage + right/- left
```

Two claims: drop is positive *below* the line of sight, windage is positive to the shooter's *right*. Both are measurable, and so is the harder question underneath them — which plane, exactly, is "the line of sight", and where does the zero put the bullet relative to it. Let us measure all of it.

5.6.1 Evidence 1: drop is positive downward

Keep the load and move the zero from 100 yd to 300 yd:

Listing 5.15: Input

```
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(300)))
:solve
:at 100 yd
:at 300 yd
:at 500 yd
```

Listing 5.16: A 300 yd zero, sampled short, at, and beyond the zero

```
Observation
  distance: 100.00 yd
  drop    : -4.19 in
Observation
  distance: 300.00 yd
  drop    : +0.00 in
Observation
  distance: 500.00 yd
  drop    : +29.72 in
```

At 100 yd with a 300 yd zero the bullet is unambiguously *above* the line of sight — that is what a mid-range rise is — and the Lab prints `-4.19`, which the legend calls "above". Drop is positive *downward*. Read it as depth, not as height.

5.6.2 Evidence 2: at the zero, drop is zero; at the muzzle, it is the sight height

The 300 yd row is the one that pins the datum down: `+0.00 in` at the zero distance. The bullet is on the line of sight there, which is what a zero means and what a shooter expects.

The muzzle is the other end of the same statement. Solve the load with a 100 yd zero and query the muzzle and the zero:

Listing 5.17: `:at 0 yd` and `:at 100 yd`, 100 yd zero, drop lines only

```
  distance: 0.00 yd
  drop    : +1.80 in
  distance: 100.00 yd
  drop    : +0.00 in
```

`+1.80 in` at the muzzle is exactly the rifle's sight height. The bore sits 1.8 inches under the scope, so a bullet still in it is 1.8 inches below the line of sight. Nothing has been added to the number; that is where the bullet is.

Now change only the sight height, from 1.8 in to 3.0 in, keep the 100 yd zero, and re-solve:

Listing 5.18: Input

```
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in, 3.0 in mount", inches(3.0), m(0),
     inches(8), "right"))
:solve
:at 0 yd
:at 100 yd
:at 600 yd
```

Listing 5.19: A taller mount moves the muzzle drop, and flattens what follows

```
  distance: 0.00 yd
  drop    : +3.00 in
  distance: 100.00 yd
  drop    : +0.00 in
  distance: 600.00 yd
  drop    : +75.06 in
```

The muzzle drop tracks the sight height exactly; the zero-distance drop stays at zero, because the zero is still a zero. Downrange the taller mount reports *less* drop, not more: 600 yd moved from `+81.08` to `+75.06`, a difference of 6.02 inches. That is geometry, not an offset. A bullet that has to climb 3.0 inches to reach the line of sight in the first 100 yards is rising more steeply when it crosses than one that only had to climb 1.8, and the extra 1.2 inches of sight height buys about 1.2 inches of additional height for every hundred yards past the zero — five of them by 600 yd, hence 6.0.

The field that sets the datum

A zero is a geometric solution: the engine tilts the bore until the trajectory passes through a specified point at the zero distance. The v1 request names that point with `target_height_m`, documented as metres above the local ground, and the engine defaults it to 0.0 — a target sitting on the dirt — when the field is absent.

The Lab always sends it, defaulted to *muzzle height plus sight height* — the height of your line of sight above the ground. So the zero is solved to the line of sight, and `drop` at the zero distance is 0.00.

You can read the resolved value back off any run. For the reference load, `sight_height_m` and `target_height_m` both come back as 0.045719999999999997 — 1.8 inches — because `muzzle_height_m` is zero. Re-solve the same load with the muzzle 1.2 m off the ground and `target_height_m` becomes 1.2457199999999999, which is $1.2 + 0.04572$; the drop at the muzzle is still `+1.80 in` and the drop at 100 yd is still `+0.00 in`, because raising the whole rifle does not change where the sight line is *relative to the bore*.

5.6.3 Evidence 3: cross-checking against the engine's own table

The claim is strong enough to deserve an independent check, so we asked the same engine the same question through a completely different code path: its `range-table` subcommand, which reports drop relative to the line of sight in the conventional way.

Listing 5.20: The same load, straight to the engine CLI

```
ballistics range-table -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
  --sight-height 1.8 --zero-distance 100 --start 100 --end 1000 --step 100 \
  --wind-speed 0 --temperature 53.6 --pressure 25.99 --humidity 35 --altitude 3937
```

Listing 5.21: Engine `range-table` output

```
Range Table (zero: 100 yd, 0 mph crosswind)
| Range | Drop    | Drop    | Wind  | Wind  | Vel   | Energy | ToF  |
| (yd)  | (in)    | (MIL)   | (in)  | (MIL) | (fps) | (ft-lb)| (s)  |
|   100 |    -0.0 |  -0.000 |   0.0 |  0.00 |  2583 |   2073 | 0.11 |
|   200 |     3.4 |   0.473 |   0.0 |  0.00 |  2459 |   1879 | 0.23 |
|   300 |    12.6 |   1.164 |   0.0 |  0.00 |  2338 |   1699 | 0.36 |
|   400 |    28.1 |   1.951 |   0.0 |  0.00 |  2221 |   1534 | 0.49 |
|   500 |    50.7 |   2.815 |   0.0 |  0.00 |  2108 |   1381 | 0.63 |
|   600 |    81.1 |   3.754 |   0.0 |  0.00 |  1998 |   1241 | 0.77 |
|   700 |   120.2 |   4.770 |   0.0 |  0.00 |  1891 |   1111 | 0.93 |
|   800 |   169.1 |   5.870 |   0.0 |  0.00 |  1787 |    992 | 1.09 |
|   900 |   228.8 |   7.062 |   0.0 |  0.00 |  1685 |    883 | 1.26 |
|  1000 |   300.8 |   8.357 |   0.0 |  0.00 |  1586 |    782 | 1.45 |
```

Two caveats on the comparison

The table's borders are box-drawing characters in the terminal; they are drawn here with ASCII pipes so the listing sets cleanly. Nothing else was changed. And the CLI run has no wind and no Coriolis, while the Lab run has 8 mph from 270° and Coriolis on — which is why we compare only the vertical columns. The velocity columns match the Lab's to the digit (2582.80, 2458.89, 2338.37, 2221.35, 2107.93, 1997.90, 1890.96, 1786.81, 1685.30, 1586.32 ft/s), so we know it is the same trajectory.

Now put the two side by side. The Lab's `:dope 100 1000 100 yd` output for this load, against the engine's own table:

Range (yd)	Lab drop (in)	Engine drop (in)	Lab come-up (mil)	Engine drop (MIL)
100	0.00	−0.0	0.00	−0.000
200	3.41	3.4	0.47	0.473
300	12.58	12.6	1.16	1.164
400	28.10	28.1	1.95	1.951
500	50.68	50.7	2.82	2.815
600	81.08	81.1	3.75	3.754
700	120.20	120.2	4.77	4.770
800	169.05	169.1	5.87	5.870
900	228.81	228.8	7.06	7.062
1000	300.82	300.8	8.36	8.357

Ten out of ten in inches, and ten out of ten in mils, to the last digit either table prints. The two figures are the same trajectory read through two entirely separate code paths — solve-json plus the Lab's own trigonometry on one side, the engine's `range-table` formatter on the other.

What this means for you

The zero the Lab asks for is a *line-of-sight* zero. At the stated zero distance the bullet is on the point of aim, the reported drop is 0.00, and the come-up is 0.00.

So the drop column is the distance below your point of aim, directly, with nothing to subtract. The elevation correction in milliradians is $1000 \times \text{drop}/\text{distance}$ with both lengths in the same unit, and that is what the `Come-up` column already contains: at 1,000 yd, $1000 \times 300.82/36000 = 8.356$, printed as `+8.36`; at 600 yd, $1000 \times 81.08/21600 = 3.754$, printed as `+3.75`.

The one thing to keep straight is that the Lab computes the angle with `atan2` rather than the small-angle ratio, so the two agree to the printed precision at rifle ranges and diverge in the far decimals. Do not be surprised by a last-digit difference if you check it by hand.

Mil and MOA

A **milliradian** (mil) is exactly one thousandth of a radian — 3.6 inches at 100 yards, 10 cm at 100 m. The Lab treats it as exact: its conversion constant is `RADIANS_PER_MIL = 0.001`, with no "1/6400 of a circle" approximation anywhere.

A **minute of angle** (MOA) is $1/60$ of a degree, or $\pi/10800$ radians — 1.047 inches at 100 yards. The Lab uses true MOA, not the 1-inch-per-100-yard "shooter's MOA" rounding, and it will not silently convert between them.

The `Come-up` column, and its legend

The DOPE table's `Come-up` column is the angle subtended by the drop at that distance, and it carries its own legend:

```
Signs: drop + below/- above; come-up + correction up/- correction down
```

Two columns, two conventions, both stated: drop is positive below the line of sight, and a positive come-up is elevation you dial *up*. For this load at 1,000 yd the table prints `+300.82` and `+8.36`, and the engine's own table independently prints 300.8 in and 8.357 MIL. Chapter 6 treats the come-up column in full.

5.7 Windage

Windage runs the other way from drop, and deliberately: positive is to the shooter's right, negative to the left. It is an offset in the direction you would call it, not a depth.

Our 8 mph wind blows *from* 270° — from the shooter's left — and pushes the bullet right, so every windage value in this chapter is positive: `+0.40 in` at 100 yd, `+16.27 in` at 600 yd, `+50.43 in` at 1,000 yd.

Flip the wind to 90° — from the right — and the sign flips with it. The terminal sample of the shipped reference experiment, which uses a 90° wind, reads `"windage_m": -2.591700671089823` on the wire: 2.59 metres left at 1,000 yd.

Why the wind hold is negated and the come-up is not

A wind card's `Wind hold` column is the sign-flip of windage: the bullet goes left, so you hold right. A dope table's `Come-up` column is *not* a sign-flip of drop: the bullet is below, and you dial up, and both of those are positive under their own conventions. The two columns look symmetrical and are not, because the two underlying quantities are not measured in the same direction. That asymmetry is worth holding on to; it is the single most common way to misread a Lab table.

Wind-from, and why drift is not linear in range

Wind direction throughout the Lab and the engine is the bearing the wind blows *from*, relative to the shot: 0° headwind, 90° from the right, 180° tailwind, 270° from the left.
Crosswind drift is not the wind speed times the flight time. It is the wind speed times the *lag time* — the difference between the actual time of flight and the time a drag-free bullet would have taken. That is why drift grows faster than linearly with range: our load drifts 0.40 in at 100 yd but 50.43 in at 1,000 yd, a factor of 126 over a factor of 10 in range.

5.8 Flags

Each sample can carry up to four markers, and `:at` prints them:

Flag	Meaning
`transonic`	The bullet has entered the transonic region.
`subsonic`	It is below Mach 1.
`terminal`	This is the last sample of the trajectory.
`ground_threshold`	The sample is at the ground threshold.

Push the same load to 2,000 yd and the regime markers appear:

Listing 5.22: Flags at long range

```
Observation
  distance: 1600.00 yd
  time    : 2.84 s
  velocity: 1070.81 ft/s
  energy  : 356.38 ft-lb
  drop    : +1101.23 in
  windage : +185.14 in
  Mach    : 0.96
  flags   : transonic, subsonic
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 2000.00 yd
  time    : 4.03 s
  velocity: 960.20 ft/s
  energy  : 286.56 ft-lb
  drop    : +2186.77 in
  windage : +320.79 in
  Mach    : 0.86
  flags   : transonic, subsonic, terminal
Signs: drop + below/- above; windage + right/- left
```

`transonic` stays on

A sample at Mach 0.86 carries *both* `transonic` and `subsonic`. The flags describe the trajectory's history, not just the instant: once the bullet has entered the transonic region it is marked as having done so, and `subsonic` is added on top when it drops below Mach 1. That is useful — a bullet that has been through the transonic transition may be carrying yaw it did not have before, whatever its current Mach number.

There is one important gap. `:at` preserves the flags only when your query lands exactly on an engine sample. Between samples the observation is interpolated, and the flags come back empty. Section 5.9 explains why, and how to tell the two cases apart.

5.9 Exact samples and interpolation

`:at` does not re-run the engine. It reads the sample array that the solve already produced and does one of three things:

1. If the query matches a sample distance, return that sample — exactly, with its flags, re-labelled in your requested display unit.

2. If it falls between two samples, linearly interpolate every scalar between the neighbours and return an observation with *no* flags.

3. If it falls outside the computed trajectory, refuse.

Which case you are in depends on the solver's sampling interval. Our load samples every 25 yd, so:

Listing 5.23: Input

```
:at 25 yd
:at 30 yd
:at 1200 yd
:at 1300 yd
```

Listing 5.24: Exact hit, interpolated, terminal, out of range

```
Observation
  distance: 25.00 yd
  time    : 0.03 s
  velocity: 2677.90 ft/s
  energy  : 2228.86 ft-lb
  drop    : +0.89 in
  windage : +0.03 in
  Mach    : 2.41
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 30.00 yd
  time    : 0.03 s
  velocity: 2671.52 ft/s
  energy  : 2218.30 ft-lb
  drop    : +0.77 in
  windage : +0.04 in
  Mach    : 2.40
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1200.00 yd
  time    : 1.85 s
  velocity: 1396.15 ft/s
  energy  : 605.84 ft-lb
  drop    : +488.20 in
  windage : +77.26 in
  Mach    : 1.26
  flags   : terminal
Signs: drop + below/- above; windage + right/- left
Error
  code: command_execution_error
  message: at: assertion failed: trajectory_at.distance: 1188.72 m is outside computed
     trajectory [0, 1097.28] m
```

The 1,200 yd query is an exact sample and carries `flags : terminal`. The 1,300 yd query fails, and the error names the exact interval it had — `[0, 1097.28] m`, the whole computed trajectory in SI.

Sample finely if you interpolate finely

Linear interpolation between 25 yd samples is very good for velocity and time and slightly worse for drop, which is curving. If you intend to read values off the grid, tighten the sampling interval when you build the solver configuration:

`solver_config("rk45", nil, yd(5), nil, true, nil)`

The engine's ceiling is 10,000 samples per solve, which is 10,000 rows to decode and validate — so tighten it deliberately, not reflexively.

The ceiling is enforced, and it is enforced by the engine with a precise error:

Listing 5.25: Input

```
sessions.replace_solver(lab, solver_config("rk45", nil, m(0.1), nil, true, nil))
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100)))
:solve
```

Listing 5.26: A 0.1 m sampling interval over 1,097 m

```
Error
  code: resource_limit
  message: trajectory observation limit of 10000 exceeded (would produce 10974)
  path: $.sampling.interval_m
  exit code: 3
```

Note what that error carries: a code, a message with both the limit and the count that would have been produced, the JSON path of the offending request field, and the child process's exit status. The Lab checks that the exit status agrees with the error code — a resource error *must* exit 3 — and treats a disagreement as a corrupt response rather than a warning.

5.10 Doing it from Lattice

Both commands are thin wrappers over functions you can call directly, which is how you script a sweep instead of typing one `:at` per distance.

Listing 5.27: The two analysis entry points

```
analysis.trajectory_at(trajectory, distance)          // -> Observation
analysis.sample(trajectory, start, stop, interval)     // -> Array of Observation
```

Reaching the trajectory means reaching through the current run:

Listing 5.28: Input

```
:solve
let tr = sessions.current_run(lab).result
let a = analysis.trajectory_at(tr, yd(1000))
json_stringify(a.time_s)
json_stringify(a.distance.si_value)
let s = analysis.sample(tr, yd(100), yd(1000), yd(100))
len(s)
json_stringify(s[len(s)-1].distance.si_value)
json_stringify(s[len(s)-1].time_s)
len(tr.observations)
```

Listing 5.29: Output

```
1.4480186510587736
914.39999999999998
10
914.39999999999998
1.4480186510587736
49
```

`json_stringify` is the Lab's idiom for printing a float without the display formatter's rounding — `to_string` uses shorter display precision. Here it shows what `:at 1000 yd` rounds to `1.45 s`: the underlying value is 1.4480186510587736 seconds, and both the direct query and the tenth element of a 100 yd sampling grid return exactly the same double.

`analysis.sample` is worth knowing for one specific behaviour: if your requested stop is beyond the trajectory's actual range, it clips to the true terminal observation rather than failing, and appends that terminal sample exactly once even when it falls between grid points. That is what lets a DOPE table over a range the bullet did not reach still end at a real, computed row.

A sweep in four lines

```
:solve
let tr = sessions.current_run(lab).result
for o in analysis.sample(tr, yd(200), yd(1000), yd(200)) {
    print(to_string(units.distance_as(o.distance, "yd")) + " yd  " +
          to_string(units.velocity_as(o.velocity, "ft/s")) + " ft/s")
}
```

Because the observation carries typed quantities rather than bare numbers, the conversion functions are the only place a unit can be named — and naming the wrong one is a hard error, not a silent factor.

5.11 When things go wrong

Every failure mode of these two commands is a value, printed and survivable. Four you will actually meet:

Listing 5.30: `:at` before any `:solve`

```
Error
  code: command_state_error
  message: at: no current run is available
  path: $.session.current_run
```

Listing 5.31: An unsupported unit, and a negative distance

```
Error
  code: command_argument
  message: unsupported distance unit 'furlongs'; expected one of [m, km, yd, ft, in]
  path: arguments[1]
Error
  code: command_argument
  message: :at distance must be non-negative
  path: arguments[0]
```

Listing 5.32: Beyond the computed trajectory

```
Error
  code: command_execution_error
  message: at: assertion failed: trajectory_at.distance: 1188.72 m is outside computed
     trajectory [0, 1097.28] m
```

Notice that the first three carry a `path` pointing at exactly which argument or session slot was at fault, and that none of them ended the session. The Lab's error taxonomy separates "your command was malformed" (`command_argument`, `command_arity`) from "the session was not ready" (`command_state_error`) from "executing it failed" (`command_execution_error`), and the code is stable enough to branch on from a script.

The same numbers on both virtual machines

The Lab runs on either of LATTICE's two production backends — the default stack VM or the register VM, selected with `--regvm` on the launcher. A `:solve` plus `:at` plus `:dope` plus `:wind-card` workload produces byte-identical standard output on both, with identical SHA-256 digests, as does the heavier two-solve `:compare` workload.

That is a real guarantee about the Lab's own arithmetic — request construction, unit conversion, interpolation, table assembly and float formatting — and it is not a guarantee about the integration, which happens in the engine, in Rust, outside both VMs.

5.12 A worked reading

Pull it together on one line of output. Our 140 gr ELD-M at 1,000 yd:

```
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +50.43 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

- The bullet takes **1.45 s** to get there. That is the number the wind acts on.

- It arrives at **1586 ft/s**, Mach **1.43** — comfortably supersonic, well clear of the transonic region, which is why the flags are empty.
- It carries **782 ft-lb**, about a third of its muzzle energy.
- It has drifted **50.43 in right** in an 8 mph wind from the left — call it 1.4 mil of hold.
- It is **300.82 in below the line of sight**, so the elevation correction is **8.36 mil up** — which is what the `Come-up` column of a DOPE table over this run prints, and what the engine's own table prints as 8.357 MIL.

Every one of those is a number you can act on without adjusting it first. Chapter 6 builds the tables that carry them across a whole distance band, with the provenance attached.

Chapter 6

DOPE and Wind Cards

A solved trajectory is a few hundred numbers. What a shooter carries to the line is a dozen. The two commands in this chapter, `:dope` and `:wind-card`, are how the BALLISTICS LAB turns the first into the second: they resample a computed trajectory onto a grid you choose, convert every value into the units you asked for, and staple the run's provenance to the bottom so the card can never drift away from the assumptions that produced it.

How transcripts in this chapter were captured

Every session below was produced by piping a script of commands into `scripts/ballistics-lab.sh`. When standard input is a pipe, LATTICE's `input()` suppresses the `ballistics>` prompt, so the transcripts contain replies only. The commands that produced each reply are named in the caption or shown in a listing immediately above it. The three-line startup banner is elided throughout; it is shown in full in the opening session chapter. Nothing else has been edited.

6.1 What a DOPE card is

DOPE

Data On Previous Engagements — a table of the sight corrections a particular rifle, load, and set of conditions require at a set of distances. Traditionally it is what you wrote in the margin of your data book after a day of shooting. Computed dope is the prediction; recorded dope is the truth. The truing chapter is about closing the gap between them.

A dope card answers one question at each distance: *how far off the aiming point does the bullet land, and how much correction removes that*. Everything else on the card — velocity, energy, time of flight, Mach — is context that tells you whether the answer is trustworthy and whether the shot is worth taking.

Come-up, mil, MOA

The *come-up* is the angular elevation correction you dial (or hold) to put the bullet on the aiming point at a given distance.
A *mil* is a milliradian: one part in a thousand, so 1 mil subtends 0.1 m at 100 m, or about 3.6 inches at 100 yards. The Ballistics Lab's mil is *exactly* one milliradian, not an artillery mil and not a rounded approximation (`units.lat:64`).
A *MOA* (minute of angle) is 1/60 of a degree, so $\pi/10800$ radians. One MOA subtends about 1.047 inches at 100 yards. The Ballistics Lab uses the exact value; several tools in the field — including the `ballistics-engine`'s own printed tables — use the round 3438 mil-to-MOA constant instead. Chapter 8 works through what that costs.

6.2 The `:dope` command

The grammar is four arguments, all required:

Listing 6.1: The `:dope` grammar, from `:help`

```
:dope <start> <stop> <interval> <unit>
```

`start`, `stop`, and `interval` are plain numbers; `unit` is one of `m`, `km`, `yd`, `ft`, `in`. The parser enforces `start > 0`, `stop > start`, and `interval > 0` before anything is evaluated (`command_parser.lat:360`), and the command layer checks the same three constraints again on the parsed value (`commands.lat:142`).

`:dope` reads the *current run*. It does not solve. If you have not run `:solve` since the session started or since the last `:reset`, you get an error rather than a stale card:

Listing 6.2: `:dope` with no current run

```
Error
  code: command_state_error
  message: at: no current run is available
  path: $.session.current_run
```

Here is a complete card for the load we have been carrying since the session chapter — a 140 gr ELD-M at 2710 ft/s, 100 yard zero, 8 mph wind from 270°, 1200 m altitude, 12 °C, 880 hPa, Coriolis on at 44° of latitude — solved to 1200 yards and then carded from 100 to 1000 yards in 100-yard steps.

Listing 6.3: `:solve` then `:dope 100 1000 100 yd`; provenance block elided until Section 6.8

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       300.00 |    +12.58 |         +1.16 |         2338.37 |        1699.50 |     0.36 | 2.10
       400.00 |    +28.10 |         +1.95 |         2221.35 |        1533.66 |     0.49 | 2.00
       500.00 |    +50.68 |         +2.82 |         2107.93 |        1381.04 |     0.63 | 1.90
       600.00 |    +81.08 |         +3.75 |         1997.90 |        1240.63 |     0.77 | 1.80
       700.00 |   +120.20 |         +4.77 |         1890.96 |        1111.37 |     0.93 | 1.70
       800.00 |   +169.05 |         +5.87 |         1786.81 |         992.32 |     1.09 | 1.61
       900.00 |   +228.81 |         +7.06 |         1685.30 |         882.78 |     1.26 | 1.52
      1000.00 |   +300.82 |         +8.36 |         1586.32 |         782.12 |     1.45 | 1.43
Signs: drop + below/- above; come-up + correction up/- correction down
```

Read it straight down. At 100 yards — the zero — the bullet is on the line of sight and there is nothing to dial: `+0.00` inches, `+0.00` mil. Past the zero the drop grows and the come-up grows with it, row after row, which is what a turret does. At 1000 yards this rifle wants 8.36 mil up.

The three signs, and the legend that states them

Every table the BALLISTICS LAB prints ends with a legend, and that legend is the contract. There are only three conventions in the whole book.

- **Drop is positive *below*** the line of sight. A bullet above the sight line reads negative.
- **Windage is positive *right*** of the aiming point, seen from behind the rifle.
- **Come-up is positive when the sight must be *raised***, and wind hold is positive when it must be moved *right*. Both are corrections, not positions.

Drop and come-up therefore carry the same sign: a bullet 28 inches low needs the sight raised, and both columns print positive. Windage and wind hold carry opposite signs: a bullet pushed left needs a hold to the right. That asymmetry is the single thing most worth internalising about BALLISTICS LAB tables, and Section 6.5 shows the two lines of LATTICE that implement it.

6.3 The seven columns

The column set is fixed. `analysis.lat:471-478` declares it once, and `render.lat:946` asserts that the table it is handed carries exactly those seven keys *and* exactly the units the display profile implies. A mismatch is a hard failure, not a silently mislabelled card.

`Distance`
: The query distance, expressed in the session's display distance unit. Note that this is the *display* unit, not necessarily the unit you typed — Section 6.7 shows the two coming apart.

`Drop`
: The bullet's vertical offset from the line of sight, in linear units (inches for an imperial profile, metres for a metric one). Positive means the bullet is *below* the line of sight, negative means above it. Section 6.4 establishes that empirically rather than on trust.

`Come-up`
: The angular version of the same offset, in the session's angle unit (mil by default): the correction that puts the bullet back on the aiming point. Positive means the sight goes up. This is the column you dial, and Section 6.5 checks it against the `ballistics-engine`'s own elevation table at ten distances.

`Velocity`
: Remaining speed. This is the number that tells you whether you are still supersonic. Divide by roughly 1125 ft/s at sea level to sanity-check the Mach column.

`Energy`
: Kinetic energy at that distance, in ft-lb or joules. Hunters use it for terminal-performance thresholds; target shooters mostly ignore it.

`Time`
: Time of flight in seconds, always in seconds regardless of profile — the column's unit is the literal string `"s"` (`analysis.lat:477`). Time of flight is what a wind call is really made of: drift is (roughly) crosswind speed times lag time.

`Mach`
: Speed as a multiple of the local speed of sound, computed by the engine using the moist-air speed of sound for the atmosphere you gave it. Below about Mach 1.2 the bullet enters the transonic region, where drag models are least reliable and groups often open up. The card above stays above Mach 1.4 to 1000 yards; that is a comfortable card.

Read the Mach column before you trust the Drop column

Every drag model in the engine is a table of C_d against Mach. The tables are best determined in the supersonic regime and worst through transonic. If a row shows Mach between 1.2 and 0.8, treat its drop as an estimate that wants confirming on paper.
The `ballistics-engine` flags the band for you. Push the same load out to 1800 yards and query a sampled distance where it has slowed down:

Listing 6.4: `:at 1600 yd` on a run extended to 1800 yards

```
Observation
  distance: 1600.00 yd
  time    : 2.84 s
  velocity: 1070.81 ft/s
  energy  : 356.38 ft-lb
  drop    : +1101.23 in
  windage : +157.59 in
  Mach    : 0.96
  flags   : transonic, subsonic
Signs: drop + below/- above; windage + right/- left
```

Both flags at once, because the engine's transonic band is Mach 1.2 down to 0.8 inclusive and its subsonic flag is Mach below 1.0. The flags survive only on exact engine samples: an interpolated query between two samples comes back with `flags : none` (`analysis.lat:319`), so query a distance that lands on the sampling grid when you care about them.

6.4 What the Drop column is measured from

This is the single most important thing to get right about a computed card, and it is worth doing the arithmetic rather than taking anyone's word for it. The short answer is that Drop is measured from the *line of sight*, positive downward. Four measurements pin it down.

One: at the zero it is zero. The 100-yard row of the card above reads `+0.00` inches with a 100-yard zero. A zeroed rifle puts the bullet on the aiming point, and the aiming point is on the line of sight.

Two: at the muzzle it is the sight height. Query the run at zero distance, where the bullet has not yet gone anywhere:

Listing 6.5: `:at 0 yd` — the bore sits its sight height below the sight line

```
Observation
  distance: 0.00 yd
  time    : 0.00 s
  velocity: 2710.00 ft/s
  energy  : 2282.62 ft-lb
  drop    : +1.80 in
  windage : +0.00 in
  Mach    : 2.44
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

`+1.80` inches, and the rifle's sight height is 1.8 inches. That is not a coincidence and it is not a bias: the scope really does sit 1.8 inches above the bore, so a bullet still in the muzzle really is 1.8 inches below the line of sight. Positive-below, exactly as the legend says.

Three: it goes negative where the bullet is high. Card the near end and you can watch the bullet cross the sight line, climb above it, and come back:

Listing 6.6: `:dope 25 150 25 yd` on the 100-yard zero

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
        25.00 |     +0.89 |         +0.99 |         2677.90 |        2228.86 |     0.03 | 2.41
        50.00 |     +0.28 |         +0.16 |         2645.99 |        2176.07 |     0.06 | 2.38
        75.00 |     -0.02 |         -0.01 |         2614.30 |        2124.24 |     0.08 | 2.35
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       125.00 |     +0.34 |         +0.08 |         2551.51 |        2023.43 |     0.14 | 2.30
       150.00 |     +1.02 |         +0.19 |         2520.43 |        1974.44 |     0.17 | 2.27
Signs: drop + below/- above; come-up + correction up/- correction down
```

The bullet starts 1.8 inches low, rises through the sight line somewhere just under 75 yards (`-0.02`, i.e. two hundredths of an inch *above*), peaks, and settles back onto the line at the 100-yard zero. The 25-yard row's `+0.99` mil is not an anomaly either — it is 0.89 inches subtended at only 25 yards, and near-muzzle corrections are always large in angle.

Four: move the zero and the crossing moves with it. Re-zero at 300 yards and re-card the near end:

Listing 6.7: The same load with `shot(fps(2710), yd(1200), yd(300))`, then `:dope 100 500 100 yd`

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     -4.19 |         -1.16 |         2582.80 |        2073.36 |     0.11 | 2.32
       200.00 |     -4.97 |         -0.69 |         2458.88 |        1879.19 |     0.23 | 2.21
       300.00 |     +0.00 |         +0.00 |         2338.37 |        1699.49 |     0.36 | 2.10
       400.00 |    +11.33 |         +0.79 |         2221.34 |        1533.65 |     0.49 | 2.00
       500.00 |    +29.72 |         +1.65 |         2107.92 |        1381.03 |     0.63 | 1.90
Signs: drop + below/- above; come-up + correction up/- correction down
```

At 100 and 200 yards — where a 300-yard-zeroed bullet is unambiguously *above* the line of sight — Drop is negative and the come-up is negative with it: the sight wants to come *down* for a 100-yard shot on a 300-yard zero, which is what a shooter would do. At 300 yards, the new zero, Drop is `+0.00` again.

6.4.1 Where the datum comes from

None of that is automatic. A zero is solved to a *target*, and the engine needs to be told how high above the ground that target sits. The `solve-json` request carries a field for it, `shot.target_height_m`, documented as the height above ground the zero is solved to; left out, the engine defaults it to `0.0` — a target at ground level — and says so in the assumptions.

A conventional zero is not a shot at the dirt. The target sits on the line of sight, so the datum is the height of the sight line above the ground: muzzle height plus sight height. The Ballistics Lab fills that in for you when the experiment's `Shot` does not name a target height (`protocol_v1.lat:74-92`), and Section 6.5 shows the engine echoing the number back.

Overriding the datum

`shot()` takes `target_height` as an optional argument. Give it one and the Ballistics Lab sends yours instead — which is what you want if you really are zeroing on a target at a known height above the ground that is not your sight line, for instance a plate on the deck at a known-distance range. Leave it `nil` and you get the line-of-sight zero the tables in this chapter assume.

You can confirm the whole datum story against the `ballistics-engine`'s own elevation table, which is referenced to the line of sight and computed by a completely different code path — flags rather than JSON. Same load, same atmosphere, same 100-yard zero:

Listing 6.8: The engine's own come-up table, CSV output

```
$ ballistics come-ups --zero-distance 100 -v 2710 -b 0.315 -m 140 -d 0.264 \
    --drag-model g7 --start 100 --end 1000 --step 100 --adjustment-unit mil \
    --sight-height 1.8 --altitude 3937.0079 --temperature 53.6 \
    --pressure 25.9871 --humidity 35 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,-0.000,-0.000,2583,2073,0.113
200,0.473,0.474,2459,1879,0.232
300,1.164,0.691,2338,1699,0.358
400,1.951,0.787,2221,1534,0.489
500,2.815,0.864,2108,1381,0.628
600,3.754,0.938,1998,1241,0.774
700,4.770,1.016,1891,1111,0.928
800,5.870,1.100,1787,992,1.092
900,7.062,1.192,1685,883,1.265
1000,8.356,1.294,1586,782,1.448
```

Velocity, energy and time match the Ballistics Lab's card to the precision each side prints (2583 against 2582.80, 782 against 782.12, 1.448 against 1.45). So does the elevation. The engine's `drop_mil` is the total dial from the zero; convert it to inches at 0.036 inches per mil per yard and you get the Ballistics Lab's Drop column back:

$$\text{drop (in)} \;=\; \text{drop\_mil} \times R\,\text{(yd)} \times 0.036$$

At 200 yards, $0.473 \times 200 \times 0.036 = 3.4056$ — the card says `+3.41`. At 500 yards, $2.815 \times 500 \times 0.036 = 50.67$ against the card's `+50.68`. At 1000 yards, $8.356 \times 1000 \times 0.036 = 300.82$ against the card's `+300.82`. The odd hundredth is the engine's three-decimal mil being coarser than the Ballistics Lab's inch, not a disagreement.

One reference, two printings

The engine's `solve-json` samples report `drop_m` already referenced to the line of sight, and the Ballistics Lab passes that value through untouched — `process_backend.lat:403` constructs the observation's drop with `m(object.get("drop_m"))` and applies no sign change and no datum change. The `ballistics-engine`'s printed `come-ups` table is the same quantity in mils. If you cross-check the Ballistics Lab against a third solver and the elevations disagree by a term proportional to R, the first thing to check is what height each tool zeroed to.

6.5 The two sign rules, in two lines of LATTICE

The come-up and the wind hold are computed four dozen lines apart in the same file, and putting them side by side is the whole of the sign story. The come-up, at `analysis.lat:459`:

Listing 6.9: `analysis.lat:457-460` — the come-up calculation

```
row.set(
"come_up",
_angle_value(atan2(item.drop.si_value, item.distance.si_value), correction_unit)
)
```

and the wind hold, at `analysis.lat:549-555`:

Listing 6.10: `analysis.lat:549-555` — the wind hold calculation

```
row.set(
"wind_hold",
_angle_value(
atan2(0.0 - item.windage.si_value, item.distance.si_value),
correction_unit
)
)
```

One negates its input and the other does not, and that is correct. The two inputs point opposite ways.

`item.drop.si_value` is positive-*down* — Section 6.4 measured it four ways. A correction that fixes a low impact is *up*, and the BALLISTICS LAB prints come-up positive for up, so the come-up is the angle subtended by the drop itself: no negation. `item.windage.si_value` is positive-*right*. A correction that fixes a right impact is *left*, and the BALLISTICS LAB prints wind hold positive for right, so the hold is the negation. Same geometry, opposite input conventions, opposite code.

Why `atan2` and not division

Both corrections could be written as an offset divided by a distance, and at flat-fire angles the two agree to more decimals than anyone dials. `atan2` is used because it is the exact angle rather than its small-angle approximation, and because it stays correct in every quadrant — it carries the sign of the offset through without a special case for a bullet above the sight line. `_angle_value` then converts the radians into the session's angle unit.

Zero distance is excluded by assertion rather than by arithmetic: `analysis.dope` rejects a query at the muzzle with `dope.distances[0]: must be greater than zero`, which is why a dope band cannot start where `:at 0 yd` happily reports.

6.5.1 Checking the column against the engine

Reasoning about signs is how signs get inverted. The way to be sure is to put a Ballistics Lab card and an engine table for the identical load side by side, and the `ballistics-engine`'s `come-ups` subcommand is a good independent witness: it takes flags rather than `solve-json`, so it exercises a different path into the same physics. This is a fresh session, deliberately calm-air and Coriolis-off so that nothing but geometry can differ:

Listing 6.11: A verification load: 140 gr 6.5 mm, G7 BC 0.326, 1.75-inch sight, 100-yard zero, 4000 ft, 50 °F, 35% humidity

```
let p = projectile("140 gr 6.5mm", gr(140), inches(0.264), 0.326, "G7")
let r = rifle("verification rifle", inches(1.75))
let a = atmosphere(ft(4000), fahrenheit(50), nil, 0.35)
let s = shot(fps(2710), yd(1200), yd(100))
let sc = solver_config("rk45", nil, yd(10), nil, false, nil)
sessions.replace_experiment(lab,
    experiment("come-up verification", p, r, a, calm_wind(), s, sc))
```

Listing 6.12: `:solve` then `:dope 100 1000 100 yd`; provenance elided

```
DOPE: come-up verification
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2587.03 |        2080.17 |     0.11 | 2.34
       200.00 |     +3.44 |         +0.48 |         2467.13 |        1891.81 |     0.23 | 2.23
       300.00 |    +12.61 |         +1.17 |         2350.38 |        1717.00 |     0.36 | 2.12
       400.00 |    +28.07 |         +1.95 |         2236.87 |        1555.16 |     0.49 | 2.02
       500.00 |    +50.47 |         +2.80 |         2126.72 |        1405.77 |     0.63 | 1.92
       600.00 |    +80.57 |         +3.73 |         2019.81 |        1267.99 |     0.77 | 1.82
       700.00 |   +119.19 |         +4.73 |         1915.86 |        1140.83 |     0.92 | 1.73
       800.00 |   +167.30 |         +5.81 |         1814.57 |        1023.39 |     1.08 | 1.64
       900.00 |   +225.98 |         +6.97 |         1715.80 |         915.02 |     1.25 | 1.55
      1000.00 |   +296.48 |         +8.24 |         1619.41 |         815.10 |     1.43 | 1.46
Signs: drop + below/- above; come-up + correction up/- correction down
```

Listing 6.13: The same load through the engine's own come-up table

```
$ ballistics come-ups -b 0.326 --drag-model g7 -v 2710 -m 140 -d 0.264 \
    --sight-height 1.75 --zero-distance 100 --altitude 4000 --temperature 50 \
    --humidity 35 --start 100 --end 1000 --step 100 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,0.000,0.000,2587,2080,0.113
200,0.478,0.478,2467,1892,0.232
300,1.168,0.689,2350,1717,0.357
400,1.949,0.781,2237,1555,0.487
500,2.804,0.855,2127,1406,0.625
600,3.730,0.926,2020,1268,0.770
700,4.730,1.000,1916,1141,0.922
800,5.809,1.079,1815,1023,1.083
900,6.975,1.166,1716,915,1.253
1000,8.235,1.261,1619,815,1.433
```

`--humidity` is a percentage

The Ballistics Lab's `atmosphere()` takes relative humidity as a fraction (`0.35`); the engine CLI takes it as a percentage (`--humidity 35`). Passing `--humidity 0.35` is accepted and means 0.35%, which is a different atmosphere. The velocity and energy columns are sensitive enough to catch the mistake — they stop agreeing in the last printed digit.

Velocity, energy and time of flight agree at every one of the ten rows, to the precision each side prints: 2587.03 against 2587, 1555.16 against 1555, 1.43 against 1.433, and so on down the table. That establishes the baseline. It is one trajectory reported twice, so anything left over in the elevation columns would be a disagreement about geometry rather than about physics.

Now compare the Ballistics Lab's Come-up against the engine's `drop_mil`, which is the total dial from the zero. It is *not* the engine's column headed `come_up_mil`, which is the incremental step from the previous row — 0.478 plus 0.689 is the 1.168 at 300 yards, and so on down the table (Section 13.5 works through the naming):

Range (yd)	Ballistics Lab Come-up (mil)	Engine `drop_mil` (mil)	Ballistics Lab Drop (in)
100	+0.00	0.000	+0.00
200	+0.48	0.478	+3.44
300	+1.17	1.168	+12.61
400	+1.95	1.949	+28.07
500	+2.80	2.804	+50.47
600	+3.73	3.730	+80.57
700	+4.73	4.730	+119.19
800	+5.81	5.809	+167.30
900	+6.97	6.975	+225.98
1000	+8.24	8.235	+296.48

Same sign, same magnitude, at all ten distances. The largest disagreement anywhere in the table is 0.005 mil, at 900 and 1000 yards, which is half of the last digit the Ballistics Lab prints — the two tools round a shared number to different precisions and that is all. There is no offset, and nothing that grows with range.

The Drop column reconciles the same way. Converting the engine's dial to inches at 0.036 inches per mil per yard gives 3.4416 at 200 yards against the card's `+3.44`, 80.568 at 600 against `+80.57`, and 296.46 at 1000 against `+296.48`. The worst of those is two hundredths of an inch at 1000 yards, which is what a half-thousandth of a mil is worth at that distance.

6.5.2 The datum, echoed back by the engine

You do not have to take the line-of-sight zero on faith either. Every run keeps the engine's *resolved request* — the engine's own statement of what it actually solved, defaults filled in — and you can read it at the prompt:

Listing 6.14: Reading the resolved request out of the current run

```
let run = sessions.current_run(lab)
json_stringify(run.resolved_request.get("shot"))
json_stringify(run.resolved_request.get("rifle"))
```

Listing 6.15: The engine's echo, for the verification load above (each reply is one long line; wrapped here to fit the page)

```
{"cant_angle_rad":0,"muzzle_angle_rad":0.0011642456054687502,"max_range_m":1097.28,"zero_distance_m
     ":91.439999999999998,"aim_azimuth_rad":0,"shooting_angle_rad":0,"ground_threshold_m":-100,"
     shot_azimuth_rad":0,"target_height_m":0.044450000000000003}
{"twist_direction":"right","muzzle_velocity_mps":826.00800000000004,"muzzle_height_m":0,"sight_height_m
     ":0.044450000000000003,"twist_rate_m_per_turn":0.30480000000000002}
```

`target_height_m` and `sight_height_m` are the same number, 0.04445 m, which is 1.75 inches — the rifle's sight height, and with `muzzle_height_m` at zero, the height of the line of sight above the ground. The zero was solved to the line of sight, and the BALLISTICS LAB said so on the wire rather than letting the engine assume anything.

The assumption that is not there

The engine announces every field it had to invent. If `target_height_m` were missing from the request, the `Assumptions` block of every `:solve` would carry `default_applied: Target height defaulted to 0 m above the local ground datum`. Read the provenance in Section 6.8 and you will not find that line, because the field is sent. Absence, here, is evidence.

The wind card's hold column is checked the same way in Section 6.10, and it comes out with the opposite sign, for the reason the two code listings above make plain.

6.6 Choosing the band and the interval

Three numbers control the shape of the card, and each has a consequence.

6.6.1 The start

`start` must be greater than zero (the parser rejects `:dope 0 1000 100 yd` outright) and must lie inside the computed trajectory. Ask for a band that begins past where the bullet stopped and the resampler — not the parser — objects:

Listing 6.16: `:dope 0 1000 100 yd` and `:dope 1500 2000 100 yd` against a 1200-yard run

```
Error
  code: command_argument
  message: :dope start must be greater than zero
  path: arguments[0]
Error
  code: command_execution_error
  message: dope: assertion failed: sample.start: must be inside the computed trajectory
```

The first is caught before any work happens; the second requires knowing the trajectory, so it surfaces as an execution error from `analysis.lat:354`.

6.6.2 The stop, and what happens past the end

A `stop` beyond the computed range is *clipped*, not rejected. `analysis.lat:356-359` pulls the effective stop back to the terminal observation. Asking for 2000 yards of a 1200-yard run gives you the 1200-yard run:

Listing 6.17: `:dope 200 1200 200 yd` and `:dope 200 2000 200 yd` produce byte-identical tables

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       400.00 |    +28.10 |         +1.95 |         2221.35 |        1533.66 |     0.49 | 2.00
       600.00 |    +81.08 |         +3.75 |         1997.90 |        1240.63 |     0.77 | 1.80
       800.00 |   +169.05 |         +5.87 |         1786.81 |         992.32 |     1.09 | 1.61
      1000.00 |   +300.82 |         +8.36 |         1586.32 |         782.12 |     1.45 | 1.43
      1200.00 |   +488.20 |        +11.30 |         1396.15 |         605.84 |     1.85 | 1.26
Signs: drop + below/- above; come-up + correction up/- correction down
```

That clipping matters more than it looks. If the engine terminated early — the bullet hit the ground, or dropped below the velocity floor — the card stops where the bullet stopped, and the last row is the real terminal sample. It is not an extrapolation and it is not padded with zeros.

When the terminal distance does not land on your grid, `analysis.lat:385-397` appends it anyway, exactly once:

Listing 6.18: `:dope 100 1300 250 yd`: the terminal row is appended off-grid

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       350.00 |    +19.50 |         +1.55 |         2279.41 |        1614.88 |     0.42 | 2.05
       600.00 |    +81.08 |         +3.75 |         1997.90 |        1240.63 |     0.77 | 1.80
       850.00 |   +197.49 |         +6.45 |         1735.74 |         936.41 |     1.18 | 1.56
      1100.00 |   +386.67 |         +9.76 |         1489.88 |         689.92 |     1.64 | 1.34
      1200.00 |   +488.20 |        +11.30 |         1396.15 |         605.84 |     1.85 | 1.26
Signs: drop + below/- above; come-up + correction up/- correction down
```

The grid ran 100, 350, 600, 850, 1100 and then stopped, because the next point would have been 1350; the 1200-yard terminal row was appended after it.

The `ballistics-engine`'s own card subcommands handle the same situation, but not in the same way — they stop at the last requested row the flight reaches and say so in a notice rather than appending a terminal row. Section 6.11 sets the two side by side.

6.6.3 The interval, and the grid underneath

Here is the part that catches people. The Ballistics Lab has *two* grids: the one the engine sampled on (set by the experiment's `solver.sampling_interval`), and the one you asked for. Your `:dope` interval does not change the solve; it selects points on the already-computed trajectory, interpolating linearly between engine samples when your point falls between two of them (`analysis.lat:292-323`).

Our running experiment samples at 25 yards, so every 100-yard row above lands exactly on an engine sample and no interpolation happens at all. Ask for 33-yard steps and most rows would be interpolated. Linear interpolation across a 25-yard gap on a smooth trajectory is harmless. Across a 300-yard gap it is not — Section 7.8 in the next chapter measures the damage, because that is where it bites hardest.

Match the engine's sampling to the card you want

If you know you will card at 100-yard intervals, set the experiment's sampling interval to 100 yards (or a divisor of it) before you solve: `sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), nil, true, nil))`. A 25-yard sampling interval over 1200 yards is 49 samples — cheap, and every common card interval falls on it exactly.

Both grids are bounded. `analysis.lat:12` caps an analysis result at 10 000 rows and the resampler projects the row count *before* building anything (`analysis.lat:360-368`), so a fat-fingered `:dope 1 1000 0.001 m` fails immediately with a resource message rather than allocating for a minute first. The

engine has the matching bound: `sampling.interval_m` may not produce more than 10 000 samples, and it errors rather than thinning.

6.7 Query units and display units

The unit argument to `:dope` chooses the *grid*. The session's display preferences choose the *rendering*. They are independent, and watching them come apart is the fastest way to understand the BALLISTICS LAB's unit model (Chapter 8 does it properly).

Switch the session to metric and ask for the same 100-to-1000 band in yards:

Listing 6.19: Metric display, yard grid: `sessions.set_display_preferences(lab, sessions.metric_display_preferences(2))` then `:dope 100 1000 100 yd`

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (m) | Drop (m) | Come-up (mil) | Velocity (m/s) | Energy (J) | Time (s) | Mach
-------------+----------+---------------+----------------+------------+----------+-----
       91.44 |    +0.00 |         +0.00 |         787.24 |    2811.11 |     0.11 | 2.32
      182.88 |    +0.09 |         +0.47 |         749.47 |    2547.85 |     0.23 | 2.21
      274.32 |    +0.32 |         +1.16 |         712.74 |    2304.22 |     0.36 | 2.10
      365.76 |    +0.71 |         +1.95 |         677.07 |    2079.36 |     0.49 | 2.00
      457.20 |    +1.29 |         +2.82 |         642.50 |    1872.44 |     0.63 | 1.90
      548.64 |    +2.06 |         +3.75 |         608.96 |    1682.07 |     0.77 | 1.80
      640.08 |    +3.05 |         +4.77 |         576.36 |    1506.82 |     0.93 | 1.70
      731.52 |    +4.29 |         +5.87 |         544.62 |    1345.40 |     1.09 | 1.61
      822.96 |    +5.81 |         +7.06 |         513.68 |    1196.88 |     1.26 | 1.52
      914.40 |    +7.64 |         +8.36 |         483.51 |    1060.42 |     1.45 | 1.43
Signs: drop + below/- above; come-up + correction up/- correction down
```

The rows are still at 100-yard spacing — they are just labelled in metres. Note that the Come-up column is identical to the imperial card ($+0.00, +0.47, \ldots, +8.36$): an angle does not care what unit the distance was measured in.

Ask for a metric grid instead and you get round metres:

Listing 6.20: Metric display, metric grid: `:dope 100 1000 100 m`

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (m) | Drop (m) | Come-up (mil) | Velocity (m/s) | Energy (J) | Time (s) | Mach
-------------+----------+---------------+----------------+------------+----------+-----
      100.00 |    +0.00 |         +0.03 |         783.67 |    2785.76 |     0.12 | 2.31
      200.00 |    +0.12 |         +0.60 |         742.52 |    2500.88 |     0.26 | 2.19
```

(rows 300–1000 elided; the full table appears in Appendix A.)

The linear offset unit is chosen by policy, not by you: `render.lat:224` returns `"m"` when the display distance unit is metric (`m` or `km`) and `"in"` otherwise. The command layer then asks `analysis.lat` to *calculate* in exactly that unit, so the renderer's assertion about column units can never fail on a correctly built table. Choose `km` as your display unit and you get kilometres in the distance column and metres in the drop column:

Listing 6.21: `sessions.set_display_preferences(lab, sessions.display_preferences("km", "m/s", "J", "mil", 3))` then `:dope 200 1000 200 m`

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (km) | Drop (m) | Come-up (mil) | Velocity (m/s) | Energy (J) | Time (s) | Mach
--------------+----------+---------------+----------------+------------+----------+------
        0.200 |   +0.119 |        +0.597 |        742.518 |   2500.879 |    0.255 | 2.192
        0.400 |   +0.908 |        +2.270 |        664.003 |   1999.975 |    0.540 | 1.960
        0.600 |   +2.590 |        +4.317 |        590.550 |   1581.954 |    0.860 | 1.743
        0.800 |   +5.403 |        +6.753 |        521.376 |   1233.015 |    1.220 | 1.539
        1.000 |   +9.673 |        +9.673 |        455.974 |    943.119 |    1.630 | 1.346
Signs: drop + below/- above; come-up + correction up/- correction down
```

The precision argument (`3`, here) is the number of decimals in every numeric cell, and it is a session-wide setting, not a per-table one.

6.8 Provenance: why every table drags its history along

Every table the Ballistics Lab prints ends with a provenance block. Here is the one that belongs to the first card in this chapter, printed in full:

Listing 6.22: The provenance block appended to every `:dope` table

```
Provenance
  [0] 140 gr ELD-M at 1,200 yd
    engine: 0.32.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
    Assumptions
      - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
      - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
      - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
      - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
      - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
      - default_applied: Constant vertical wind speed defaulted to 0 m/s.
     [$.wind.vertical_speed_mps]
      - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
      - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
      - default_applied: Enhanced spin drift defaulted to disabled.
     [$.effects.enhanced_spin_drift]
    Warnings
      none
```

Read it as the card's fine print.

`engine` / `schema`
: Which engine build produced these numbers and which wire schema it spoke. Two cards from different engine versions are not interchangeable, and this line is how you find out.

`solver`
: The integration method the engine actually used, taken from the engine's *resolved request* rather than from what you asked for (`analysis.lat:151-155`). If you asked for `rk4` and the engine substituted something, the provenance would say so.

`verified`
: Whether this trajectory has been promoted to a verified artifact (the verified-artifacts chapter). A fresh solve is always `no`.

`termination`
: Why the run stopped: `max_range` (you got the range you asked for), `ground_threshold` (the bullet hit the dirt first), `time_limit`, or `velocity_floor`. This is the field that tells you whether the last row of your card is the distance you wanted or the distance you got.

`actual range`
: How far the solve actually reached.

`Assumptions`
: Every value the engine filled in for you, each with the JSON path of the field it filled. Nine `default_applied` notices on a fully specified experiment is normal, not a warning: they are the engine saying out loud that you did not specify a cant angle, so it used zero. Note which path is *not* in the list — `$.shot.target_height_m`. The BALLISTICS LAB sends that one (Section 6.4.1), so the engine has nothing to assume about the height the zero was solved to.

`Warnings`
: Things the engine did that you might not have wanted: a wind deck that ran out before the target, an explicit time step handed to an adaptive integrator, an experimental effect enabled. `none` is the common case and the one you want.

The engine stamps its own version

The `engine:` line reports whatever the binary reports about itself. The sessions in this book were captured against a `ballistics` executable that identifies as `0.32.0`; yours may differ, and if it does, your numbers may differ in the last digits. That is precisely why the provenance block exists — and why the verified-artifacts chapter makes the engine version part of the verified artifact rather than a footnote.

6.9 The wind card

A wind card answers the other half of the shooting problem: how far the wind pushes the bullet sideways, and how much you must hold or dial back.

Wind-FROM, windage, wind hold

The BALLISTICS LAB and the `ballistics-engine` both use the *wind-FROM* convention: 0° is a headwind, 90° is wind from the shooter's right, 180° is a tailwind, 270° is wind from the left.
Windage is where the bullet actually lands relative to the aim, positive to the shooter's right. A wind *from* the right pushes the bullet *left*, so a 90° wind produces negative windage.
Wind hold is the angular correction that cancels it — the opposite sign. A left impact needs a right hold.

The colon command mirrors `:dope` exactly:

Listing 6.23: The `:wind-card` grammar

```
:wind-card <start> <stop> <interval> <unit>
```

Listing 6.24: `:wind-card 200 1000 200 yd` on the running experiment

```
Wind card
Series | Trajectory               | Wind (mph) | From (deg) | Distance (yd) | Windage (in) | Wind hold (mil)
-------+--------------------------+------------+------------+---------------+--------------+----------------
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        200.00 |        +1.64 |           -0.23
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        400.00 |        +6.88 |           -0.48
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        600.00 |       +16.27 |           -0.75
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        800.00 |       +30.49 |           -1.06
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |       1000.00 |       +50.43 |           -1.40
Signs: windage + right/- left; wind hold + correction right/- correction left
```

A 270° wind comes from the left and pushes the bullet right, so windage is positive and the hold is left — negative. This is the pair whose signs oppose each other, and it is the reason Section 6.5 spends two code listings on the difference: a lateral *position* and a lateral *correction* cannot share a sign, whereas drop and come-up can and do.

The `Series` column renders as a float

`series_index` is an integer in the row data — the renderer's validator insists on it (`render.lat:776`) — but the table renderer formats every numeric cell to the profile's precision, so an index of 0 prints as `0.00`. It is cosmetic. The same applies to the comparison table in Chapter 7.

6.10 Reading a wind card

One row per distance for a single wind speed is not really a card; it is a column. What a shooter wants is a matrix: several wind speeds against several distances, so the call at the line is an interpolation between two printed numbers rather than a calculation.

`:wind-card` always builds a single series, because it reads the current run and a run has one wind. The analysis layer underneath has no such restriction — `analysis.lat:499` takes an *array* of trajectories — so you build the real card in LATTICE at the prompt.

Solve three times, keeping each run in a local binding, then hand all three to `analysis.wind_card`:

Listing 6.25: A three-speed wind card, typed at the Ballistics Lab prompt

```
sessions.replace_winds(lab, [wind(mph(5), deg(90))])
let a = sessions.solve_session(lab)
sessions.replace_winds(lab, [wind(mph(10), deg(90))])
let b = sessions.solve_session(lab)
sessions.replace_winds(lab, [wind(mph(15), deg(90))])
let c = sessions.solve_session(lab)
let card = analysis.wind_card([a.result, b.result, c.result],
    [yd(200), yd(400), yd(600), yd(800), yd(1000)], "in", "mil", "mph", "deg")
print(render.render_wind_card(card, sessions.display_of(lab)))
```

Listing 6.26: The three-speed card; provenance block (three entries) elided

```
Wind card
Series | Trajectory               | Wind (mph) | From (deg) | Distance (yd) | Windage (in) | Wind hold (mil)
-------+--------------------------+------------+------------+---------------+--------------+----------------
  0.00 | 140 gr ELD-M at 1,200 yd |       5.00 |      90.00 |        200.00 |        -0.89 |          +0.12
  1.00 | 140 gr ELD-M at 1,200 yd |      10.00 |      90.00 |        200.00 |        -1.86 |          +0.26
  2.00 | 140 gr ELD-M at 1,200 yd |      15.00 |      90.00 |        200.00 |        -2.83 |          +0.39
  0.00 | 140 gr ELD-M at 1,200 yd |       5.00 |      90.00 |        400.00 |        -3.74 |          +0.26
  1.00 | 140 gr ELD-M at 1,200 yd |      10.00 |      90.00 |        400.00 |        -7.82 |          +0.54
  2.00 | 140 gr ELD-M at 1,200 yd |      15.00 |      90.00 |        400.00 |       -11.90 |          +0.83
  0.00 | 140 gr ELD-M at 1,200 yd |       5.00 |      90.00 |        600.00 |        -8.86 |          +0.41
  1.00 | 140 gr ELD-M at 1,200 yd |      10.00 |      90.00 |        600.00 |       -18.52 |          +0.86
  2.00 | 140 gr ELD-M at 1,200 yd |      15.00 |      90.00 |        600.00 |       -28.19 |          +1.30
  0.00 | 140 gr ELD-M at 1,200 yd |       5.00 |      90.00 |        800.00 |       -16.64 |          +0.58
  1.00 | 140 gr ELD-M at 1,200 yd |      10.00 |      90.00 |        800.00 |       -34.77 |          +1.21
  2.00 | 140 gr ELD-M at 1,200 yd |      15.00 |      90.00 |        800.00 |       -52.90 |          +1.84
  0.00 | 140 gr ELD-M at 1,200 yd |       5.00 |      90.00 |       1000.00 |       -27.59 |          +0.77
  1.00 | 140 gr ELD-M at 1,200 yd |      10.00 |      90.00 |       1000.00 |       -57.60 |          +1.60
  2.00 | 140 gr ELD-M at 1,200 yd |      15.00 |      90.00 |       1000.00 |       -87.61 |          +2.43
Signs: windage + right/- left; wind hold + correction right/- correction left
```

Rows are grouped by distance, series within distance — the loop in `analysis.lat:533-559` iterates queries on the outside and trajectories on the inside. Now the card reads the way a shooter uses it: at 600 yards, a 5 mph left-to-right wind is 0.41 mil of hold and a 15 mph is 1.30, so an 8 mph call is somewhere around 0.65.

6.10.1 The card is linear in wind speed — and the intercept is not zero

Look down the 1000-yard block: -27.59, -57.60, -87.61 inches for 5, 10 and 15 mph. The differences are -30.01 and -30.01 — exactly linear, at -6.002 inches per mph. Extrapolate to zero wind and you do not get zero. You get $+2.42$ inches.

That residual is real, and you can see it directly by solving in still air:

Listing 6.27: `sessions.replace_winds(lab, [])` then `:solve` and `:at 1000 yd`

```
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.31 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +2.42 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Two and a half inches of right drift at 1000 yards in dead calm air is the Coriolis term. Our experiment enables it (`solver_config("rk45", nil, yd(25), nil, true, nil)` — the fifth argument) and supplies a 44° latitude, and the engine deflects a northern-hemisphere shot to the right. It is a small effect and it is not wind, but it lands in the same column, and reading a wind card without noticing it means calling 0.4 mil of wind that is not there. the chapter on stability and the small effects takes the effect apart properly.

Check the earlier single-series card against this: it used 8 mph from 270°, which pushes right, so we expect $2.42 + 8 \times 6.002 = 50.44$ inches at 1000 yards. The card said `+50.43`.

Zero-wind row as a diagnostic

When you build a multi-speed card, include a calm series. The zero-wind row is where every non-wind lateral effect — Coriolis, spin drift, a sight offset — shows up naked, and it turns your card from a wind table into a *lateral-budget* table.

6.10.2 Where the wind card refuses

The wind card's whole premise is that a series can be labelled with one wind speed and one direction. A segmented wind deck — different winds over different stretches of the flight — has no such label, so `analysis.lat:490` refuses:

Listing 6.28: A two-segment wind deck: 8 mph from 270° out to 600 yards, then 12 mph from 250°

```
sessions.replace_winds(lab, [wind(mph(8), deg(270), nil, yd(600)),
    wind(mph(12), deg(250), nil, yd(1200))])
```

Listing 6.29: `:solve` on that deck, then `:wind-card 200 1000 200 yd`

```
Error
  code: command_execution_error
  message: wind-card: assertion failed: wind_card.trajectories[0]: wind cards require calm or
     one constant wind per trajectory
```

`:dope` has no such restriction and works fine on the same run: the elevation card does not claim anything about the wind, so a segmented deck is simply part of the conditions.

Listing 6.30: `:dope 200 1000 400 yd` on a run with a two-segment wind deck

```
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       600.00 |    +81.08 |         +3.75 |         1997.91 |        1240.64 |     0.77 | 1.80
      1000.00 |   +300.68 |         +8.35 |         1588.24 |         784.01 |     1.45 | 1.43
Signs: drop + below/- above; come-up + correction up/- correction down
```

Notice how little the wind deck changed elevation: 300.68 against 300.82 inches at 1000 yards, a hundredth of a mil on the dial. Crosswind is nearly free of vertical consequence at flat-fire distances — *nearly*. the wind chapter deals with aerodynamic jump, which is the part that is not free.

6.11 The engine's own cards

The `ballistics-engine` has `come-ups` and `wind-card` subcommands of its own, and they are worth knowing about because they are shaped differently.

Listing 6.31: The engine's wind card is a matrix, one column per wind speed

```
$ ballistics wind-card --zero-distance 100 -v 2710 -b 0.315 -m 140 -d 0.264 \
    --drag-model g7 --sight-height 1.8 --altitude 3937.0079 --temperature 53.6 \
    --pressure 25.9871 --humidity 35 --wind-speeds 5,10,15 --wind-angle 90 \
    --adjustment-unit mil -o csv
# wind_angle=90
range_yd,wind_5_mph,wind_10_mph,wind_15_mph
100,-0.1,-0.1,-0.2
200,-0.1,-0.3,-0.4
300,-0.2,-0.4,-0.6
400,-0.3,-0.6,-0.9
500,-0.4,-0.7,-1.1
600,-0.4,-0.9,-1.3
700,-0.5,-1.1,-1.6
800,-0.6,-1.3,-1.9
900,-0.7,-1.5,-2.2
1000,-0.8,-1.7,-2.5
```

Three differences matter:

1. **Shape.** The engine emits wide (one column per wind speed); the Ballistics Lab emits long (one row per series-distance pair). Wide is easier to read on paper; long is easier to filter, join, and plot, which is why `analysis_export.lat` can hand it straight to `pandas` (the rendering and analysis chapter).

2. **Polarity.** The engine prints the *drift* — where the bullet goes. The Ballistics Lab prints drift *and* the hold that cancels it. At 1000 yards and 10 mph the engine says -1.7 mil of drift; the Ballistics Lab says -57.60 inches of windage and $+1.60$ mil of hold. Subtract the Ballistics Lab's 2.42 inch Coriolis term the engine is not modelling here and -60.02 inches over 36 000 is -1.667 mil, which rounds to the engine's -1.7.

3. **Provenance.** The engine's card is numbers. The Ballistics Lab's card carries the engine version, the solver, the termination reason, and every assumption the engine applied. If you are printing a card and stapling it inside a rifle case, that difference is the whole argument for the Ballistics Lab.

4. **What happens at the end of the flight.** Both refuse to invent a row past the point where the bullet stopped, and they stop in slightly different places. The Ballistics Lab clips your requested stop back to the trajectory and appends the true terminal row off-grid (Section 6.6), so its last line is where the bullet actually finished. From engine 0.37.0 an engine card stops at the last *requested* row the flight covers, prints everything up to it, and reports what it left out — a warning on stderr naming the last row, the requested end and the load's own reach, plus a

`truncated` object in `-o json`. Asking a rimfire for a 1200-yard card gets you the ten rows it can fly and a sentence about the two it cannot, where before 0.37.0 the engine either fabricated the missing rows or refused the whole card. Section 13.7 works the contract through on all five engine card surfaces.

Because the two stop differently, a Ballistics Lab card and a CLI card of the same short-falling load can disagree about their last printed distance by up to one row spacing without either being wrong. Reconcile them against the engine's `reach` figure, which is a property of the load, rather than against the two final labels.

6.12 Errors and limits, collected

Condition	Code	Raised by
No current run	`command_state_error`	`commands.lat:187`
Wrong argument count	`command_arity`	`command_parser.lat:286`
`start` ≤ 0	`command_argument`	`command_parser.lat:360`
`stop` ≤ `start`	`command_argument`	`command_parser.lat:360`
`interval` ≤ 0	`command_argument`	`command_parser.lat:360`
Unknown unit	`command_argument`	`command_parser.lat:321`
`start` past the run	`command_execution_error`	`analysis.lat:354`
Segmented wind + wind card	`command_execution_error`	`analysis.lat:490`
Over 10 000 rows	`command_execution_error`	`analysis.lat:183`
Over 16 arguments	`resource_limit`	`command_parser.lat:250`
Line over 4096 bytes	`resource_limit`	`command_parser.lat:121`

Every one of them is a value, not a crash. The Ballistics Lab prints the error and returns you to the prompt with the session exactly as it was; a rejected `:dope` cannot damage a run.

A working card in four lines

1. Set the experiment's sampling interval to a divisor of your card interval.
2. `:solve` — check `termination` and the warnings.
3. `:dope 100 1000 100 yd` — read the Come-up column as printed; positive is up. Sanity-check the zero row, which should read `+0.00` in both the Drop and Come-up columns for a rifle zeroed at that distance.
4. `:wind-card 200 1000 200 yd`, or build the multi-speed version in Lattice — and read the hold column as printed too, remembering that its sign runs opposite to the Windage beside it.

Chapter 7

Comparing Loads

Every load decision is a comparison. Is the heavier bullet worth the sixty feet per second you give up to seat it? Does the higher-BC bullet actually buy you anything inside 600 yards, or only past it? Does a five-mph error in the wind call cost more than a fifty-fps error in the chronograph?

Those are all the same question — *two trajectories, one set of conditions, where do they differ* — and the Ballistics Lab answers it with a single argument-free command: `:compare`.

7.1 What `:compare` actually compares

`:compare` takes no arguments at all. It compares the session's *previous run* against its *current run*.

That is a deliberate design choice and it takes a moment to get used to. There is no `:compare <fileA> <fileB>`, no load registry, no naming of things to compare. The session keeps exactly two solved runs on hand (`session.lat:40` — `current_run` and `previous_run`), and `:compare` reads them.

Current run and previous run

A *run* is one complete `:solve`: an experiment snapshot, the trajectory the engine returned for it, the engine's resolved request, and the notices that came with it. Solving advances the pair: the old current becomes previous, the new result becomes current (`session.lat:601`, `let next_previous = state.current_run`).
So the recipe for a comparison is always the same three steps: solve, change something, solve again.

Listing 7.1: `:compare` with only one solve behind it

```
Error
  code: command_state_error
  message: compare: no previous run is available
  path: $.session.previous_run
```

Because the pair advances on *every* solve, and because `:reset` clears both (the persistence chapter has the exact semantics), the comparison you get is always the last two things you asked for. That is a feature at the bench — you are almost always comparing "what I had" against "what I just changed" — and a trap if you solve three times and expect the first run to still be there. It is not; it is in the history array, and `:compare` does not read history.

7.2 A real comparison

Two 6.5 mm loads out of the same rifle, at the same conditions. The baseline is the 140 gr ELD-M we have been carrying: BC 0.315 G7, 2710 ft/s. The challenger is a 147 gr ELD-M: BC 0.351 G7, but 2650 ft/s because it is heavier and you do not get the case capacity for free.

Everything else — the rifle, the 1.8 inch sight height, the 1200 m altitude, 12 °C, 880 hPa, 35% humidity, the 8 mph wind from 270°, the 100 yard zero, the 1200 yard maximum range, RK45, Coriolis at 44° — is untouched between the two solves.

Listing 7.2: Setting up the comparison at the Ballistics Lab prompt

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(200), nil, true, nil))
:solve
sessions.replace_projectile(lab, projectile("147 gr ELD-M", gr(147), inches(0.264),
    0.351, "G7", inches(1.39)))
sessions.replace_shot(lab, shot(fps(2650), yd(1200), yd(100)))
let e0 = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("147 gr ELD-M at 1,200 yd", e0.projectile,
    e0.rifle, e0.atmosphere, e0.winds, e0.shot, e0.solver))
:solve
:compare
```

The first line sets the engine's sampling interval to 200 yards. That is unusually coarse, and Section 7.9 explains why we did it. The `replace_experiment` line is a rename, and Section 7.4 explains why that one is not optional.

Listing 7.3: The `:compare` output. The real table is one 245-character line per row; this is its left half, `cut -c1-123`

```
Trajectory comparison
Series | Trajectory               | Distance (yd) | Time (s) | Velocity (ft/s) | Energy (ft-lb) | Drop (in) | Windage (in)
-------+--------------------------+---------------+----------+-----------------+----------------+-----------+-------------
  0.00 | 140 gr ELD-M at 1,200 yd |          0.00 |     0.00 |         2710.00 |        2282.62 |     +1.80 |        +0.00
  1.00 | 147 gr ELD-M at 1,200 yd |          0.00 |     0.00 |         2650.00 |        2291.79 |     +1.80 |        +0.00
  0.00 | 140 gr ELD-M at 1,200 yd |        200.00 |     0.23 |         2458.89 |        1879.20 |     +3.41 |        +1.64
  1.00 | 147 gr ELD-M at 1,200 yd |        200.00 |     0.24 |         2426.81 |        1922.01 |     +3.60 |        +1.52
  0.00 | 140 gr ELD-M at 1,200 yd |        400.00 |     0.49 |         2221.35 |        1533.66 |    +28.10 |        +6.88
  1.00 | 147 gr ELD-M at 1,200 yd |        400.00 |     0.50 |         2214.65 |        1600.63 |    +29.07 |        +6.35
  0.00 | 140 gr ELD-M at 1,200 yd |        600.00 |     0.77 |         1997.90 |        1240.63 |    +81.08 |       +16.27
  1.00 | 147 gr ELD-M at 1,200 yd |        600.00 |     0.78 |         2013.87 |        1323.57 |    +82.98 |       +14.94
  0.00 | 140 gr ELD-M at 1,200 yd |        800.00 |     1.09 |         1786.81 |         992.32 |   +169.05 |       +30.49
  1.00 | 147 gr ELD-M at 1,200 yd |        800.00 |     1.09 |         1823.18 |        1084.78 |   +171.30 |       +27.84
  0.00 | 140 gr ELD-M at 1,200 yd |       1000.00 |     1.45 |         1586.32 |         782.12 |   +300.82 |       +50.43
  1.00 | 147 gr ELD-M at 1,200 yd |       1000.00 |     1.44 |         1641.19 |         879.02 |   +301.63 |       +45.73
  0.00 | 140 gr ELD-M at 1,200 yd |       1200.00 |     1.85 |         1396.15 |         605.84 |   +488.20 |       +77.26
  1.00 | 147 gr ELD-M at 1,200 yd |       1200.00 |     1.83 |         1467.39 |         702.70 |   +483.85 |       +69.51
Signs: drop + below/- above; windage + right/- left; deltas are series minus baseline [0]
```

The six delta columns live in the other half of the same lines:

Listing 7.4: The same output, right half, `cut -c124-`

```
| Mach | Delta time (s) | Delta velocity (ft/s) | Delta energy (ft-lb) | Delta drop (in) | Delta windage (in) | Delta Mach
+------+----------------+-----------------------+----------------------+-----------------+--------------------+-----------
| 2.44 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.38 |         +0.00 |               -60.00 |                +9.18 |           +0.00 |              +0.00 |      -0.05
| 2.21 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.18 |         +0.00 |               -32.08 |               +42.81 |           +0.19 |              -0.12 |      -0.03
| 2.00 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.99 |         +0.01 |                -6.70 |               +66.98 |           +0.97 |              -0.53 |      -0.01
| 1.80 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.81 |         +0.01 |               +15.97 |               +82.94 |           +1.91 |              -1.33 |      +0.01
| 1.61 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.64 |         +0.00 |               +36.37 |               +92.46 |           +2.25 |              -2.65 |      +0.03
| 1.43 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.48 |         -0.01 |               +54.87 |               +96.90 |           +0.81 |              -4.70 |      +0.05
| 1.26 |         +0.00 |                +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.32 |         -0.02 |               +71.24 |               +96.87 |           -4.35 |              -7.75 |      +0.06
```

Before reading the deltas, notice the muzzle row. Both series report `+1.80` inches of drop at 0.00 yards — two different bullets, at two different velocities, agreeing exactly on the rifle's 1.8-inch sight height. That is the sign convention stated in one number: drop is measured downward from the line of sight, and at the muzzle the bore sits precisely its sight height below it (Section 6.4).

Read the delta columns down the series-1 rows and the whole load decision is there in six lines of arithmetic.

- **Muzzle**: the 147 starts 60 ft/s slower but with 9 ft-lb more energy. The mass wins the energy trade immediately.

- **400 yards**: velocity delta is down to −6.70 ft/s. The higher BC has almost erased the muzzle deficit.
- **600 yards**: the delta goes positive, +15.97 ft/s. This is the crossover. Past here the 147 is the faster bullet.
- **1000 yards**: +54.87 ft/s, +96.90 ft-lb, and −4.70 inches of wind — about a tenth of a mil less wind hold.
- **1200 yards**: +71.24 ft/s, −7.75 inches of wind, and Mach 1.32 against 1.26. The 147 has 0.06 Mach more margin above the transonic region, which at this distance is the number that decides whether your groups stay round.
- **Drop**: essentially a wash. +2.25 inches at 800 (the 147 shoots flatter is *not* what this says — it drops slightly more there), then −4.35 inches at 1200. Four inches at 1200 yards is a tenth of a mil. Nobody picks a load on this.

The honest summary of the table: the 147 costs nothing inside 400 yards, buys about 4 ft-lb per 100 yards past that, and — the part that matters — takes 10% off your wind hold at distance. That is what you were actually shopping for.

7.3 Reading the deltas

Six of the fifteen columns are deltas, and every one of them is *series minus baseline*, where the baseline is series 0. The dispatcher puts *previous* at index 0 on purpose:

Listing 7.5: `commands.lat:318-320` — why series 0 is the previous run

```
// Previous is deliberately series zero: every delta is current - previous.
let table = analysis.compare_trajectories(
[previous.result, current.result],
```

So a delta reads *current minus previous*: how the thing you just did differs from the thing you had. Positive delta velocity means the new load is faster. Positive delta drop means the new load drops more. Positive delta windage means the new load lands further right — which, with a 270° wind pushing right, means *more* drift.

The baseline row is always all zeros — and that is not a bug

Every series-0 row shows `+0.00` in all six delta columns, because series 0 is the baseline and $x - x = 0$. Half the numbers in a `:compare` table are structurally zero. If you scan the table quickly and read a zero delta as "no difference between the loads," you have read the baseline row.
Read the *odd* rows — the series-1 rows. Those carry the answer.

The non-delta columns on both rows are the plain values, so you can always recover the absolute numbers without doing the subtraction yourself. At 1000 yards the 140 is doing 1586.32 ft/s and the 147 is doing 1641.19; the delta column simply saves you the arithmetic.

7.4 The naming trap

Look again at the `Trajectory` column: it reads `140 gr ELD-M at 1,200 yd` on the baseline rows and `147 gr ELD-M at 1,200 yd` on the current rows. That happened only because we explicitly renamed the experiment between solves.

The `sessions.replace_*` family — `replace_projectile`, `replace_shot`, `replace_atmosphere`, and the rest — rebuilds the experiment through `session.lat:285`:

Listing 7.6: `session.lat:285-303` — the name is always carried forward

```
fn _experiment_with(
current: any,
projectile: any,
rifle: any,
atmosphere: any,
winds: any,
shot: any,
solver: any
) -> any {
    return domain.experiment(
    current.name,
    projectile,
    rifle,
    ...
```

`current.name`. Always. You can replace the bullet, the barrel, the atmosphere, the wind, the muzzle velocity, and the solver, and the experiment will still be called whatever it was called when the session started.

Rename before the second solve, or the table lies to you

Here is the same comparison run without the rename — the 8 mph wind changed to 15 mph between solves, everything else identical. Both series carry the same label:

```
  0.00 | 140 gr ELD-M at 1,200 yd |      1200.00 |    1.85 |        1396.15 |        605.84 |   +488.20 |      +77.26
  1.00 | 140 gr ELD-M at 1,200 yd |      1200.00 |    1.85 |        1396.17 |        605.85 |   +488.20 |     +141.68
```

and the delta half of the same two lines:

```
| 1.26 |         +0.00 |               +0.00 |              +0.00 |          +0.00 |            +0.00 |      +0.00
| 1.26 |         +0.00 |               +0.02 |              +0.01 |          +0.00 |           +64.42 |      +0.00
```

The table is correct — 8 mph gives 77.26 inches of drift at 1200 yards and 15 mph gives 141.68, a difference of 64.42, and elevation is untouched — but the `Trajectory` column tells you nothing. Six months later, that saved table is unreadable.
The only way to change an experiment's name is a whole-experiment replacement:

```
let e0 = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("147 gr ELD-M at 1,200 yd",
    e0.projectile, e0.rifle, e0.atmosphere, e0.winds, e0.shot, e0.solver))
```

7.5 What is held constant

The phrase "at identical conditions" is doing real work here, and it is worth being precise about what guarantees it.

The Ballistics Lab does not re-solve anything during `:compare`. It reads two runs that were each solved earlier, each against its own frozen experiment snapshot. So "identical conditions" is not something the comparison enforces — it is something *you* arranged by changing one thing between two solves.

What the comparison *does* enforce is provenance. Each series carries its own complete provenance block, and if the two runs were solved under different atmospheres, or by different engine versions, the block says so:

Listing 7.7: The two-entry provenance block, with the nine `default_applied` lines under each `Assumptions` heading removed

```
Provenance
  [0] 140 gr ELD-M at 1,200 yd
    engine: 0.32.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
    Assumptions
    Warnings
      none
  [1] 147 gr ELD-M at 1,200 yd
    engine: 0.32.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
    Assumptions
    Warnings
      none
```

Check the termination lines before you read the deltas

If one series says `max_range` and the other says `ground_threshold`, the two runs did not cover the same ground and the comparison window has silently shrunk to the shorter one (Section 7.7). The provenance block is where you notice.

7.6 Comparing a load against itself

The most useful comparisons are often not between two loads at all. They are between one load and *the same load under different conditions* — and this is the thing the `ballistics-engine`'s own `compare` subcommand cannot express, because it solves every load at one shared set of conditions by construction.

Here is the 140 gr ELD-M solved twice: once in the thin air we have been using (1200 m altitude, 880 hPa) and once at sea level (0 m, 1013.25 hPa), everything else identical — same bullet, same muzzle velocity, same 12 °C, same 35% humidity, same 8 mph wind, same zero.

Listing 7.8: A density sensitivity study

```
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(12), hpa(1013.25), 0.35,
    deg(44)))
let e1 = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("140 gr ELD-M at sea level",
    e1.projectile, e1.rifle, e1.atmosphere, e1.winds, e1.shot, e1.solver))
:solve
:compare
```

Listing 7.9: Thin air against sea level, left half, `cut -c1-124`

```
Trajectory comparison
Series | Trajectory                | Distance (yd) | Time (s) | Velocity (ft/s) | Energy (ft-lb) | Drop (in) | Windage (in)
-------+---------------------------+---------------+----------+-----------------+----------------+-----------+-------------
  0.00 | 140 gr ELD-M at 1,200 yd  |          0.00 |     0.00 |         2710.00 |        2282.62 |     +1.80 |        +0.00
  1.00 | 140 gr ELD-M at sea level |          0.00 |     0.00 |         2710.00 |        2282.62 |     +1.80 |        +0.00
  0.00 | 140 gr ELD-M at 1,200 yd  |        200.00 |     0.23 |         2458.89 |        1879.20 |     +3.41 |        +1.64
  1.00 | 140 gr ELD-M at sea level |        200.00 |     0.23 |         2421.95 |        1823.16 |     +3.49 |        +1.89
  0.00 | 140 gr ELD-M at 1,200 yd  |        400.00 |     0.49 |         2221.35 |        1533.66 |    +28.10 |        +6.88
  1.00 | 140 gr ELD-M at sea level |        400.00 |     0.50 |         2152.07 |        1439.49 |    +29.01 |        +7.99
  0.00 | 140 gr ELD-M at 1,200 yd  |        600.00 |     0.77 |         1997.90 |        1240.63 |    +81.08 |       +16.27
  1.00 | 140 gr ELD-M at sea level |        600.00 |     0.79 |         1900.40 |        1122.50 |    +84.71 |       +19.06
  0.00 | 140 gr ELD-M at 1,200 yd  |        800.00 |     1.09 |         1786.81 |         992.32 |   +169.05 |       +30.49
  1.00 | 140 gr ELD-M at sea level |        800.00 |     1.13 |         1663.89 |         860.48 |   +179.16 |       +36.10
  0.00 | 140 gr ELD-M at 1,200 yd  |       1000.00 |     1.45 |         1586.32 |         782.12 |   +300.82 |       +50.43
  1.00 | 140 gr ELD-M at sea level |       1000.00 |     1.52 |         1440.98 |         645.37 |   +324.26 |       +60.46
  0.00 | 140 gr ELD-M at 1,200 yd  |       1200.00 |     1.85 |         1396.15 |         605.84 |   +488.20 |       +77.26
  1.00 | 140 gr ELD-M at sea level |       1200.00 |     1.97 |         1233.57 |         472.96 |   +536.97 |       +93.97
Signs: drop + below/- above; windage + right/- left; deltas are series minus baseline [0]
```

Listing 7.10: The same output, right half, `cut -c125-`

```
| Mach | Delta time (s) | Delta velocity (ft/s) | Delta energy (ft-lb) | Delta drop (in) | Delta windage (in) | Delta Mach
+------+----------------+-----------------------+----------------------+-----------------+--------------------+-----------
| 2.44 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.44 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.21 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.18 |         +0.00 |                -36.94 |               -56.04 |           +0.08 |              +0.25 |      -0.03
| 2.00 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.94 |         +0.01 |                -69.28 |               -94.17 |           +0.91 |              +1.11 |      -0.06
| 1.80 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.71 |         +0.02 |                -97.50 |              -118.14 |           +3.63 |              +2.79 |      -0.09
| 1.61 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.50 |         +0.04 |               -122.92 |              -131.84 |          +10.11 |              +5.61 |      -0.11
| 1.43 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.30 |         +0.07 |               -145.34 |              -136.75 |          +23.44 |             +10.03 |      -0.13
| 1.26 |         +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.11 |         +0.12 |               -162.58 |              -132.88 |          +48.76 |             +16.72 |      -0.15
```

Everything gets worse in dense air, and the deltas tell you by how much: −36.94 ft/s at 200 yards growing to −162.58 at 1200; +48.76 inches more drop at 1200; +16.72 inches more wind drift.

The Mach column is the one to stare at. In the thin air the bullet arrives at 1200 yards doing Mach 1.26 — above the transonic band. At sea level it arrives at Mach 1.11, inside it. Same rifle, same load, same day of the week: one of those trajectories has a well-behaved terminal phase and one does not. A comparison table found that in two solves.

Sensitivity studies are just comparisons

Change one input, solve, compare. The recipe works for every input the experiment has:

- **Muzzle velocity** — how much does a 30 ft/s chronograph error cost at distance? (This is the question the truing chapter answers properly.)
- **BC** — how much does the manufacturer's optimism cost you?
- **Atmosphere** — the study above.
- **Wind** — the study in Section 7.4: 8 against 15 mph, drop unchanged, drift nearly doubled.
- **Solver** — `"rk45"` against `"euler"`, which tells you how much of your answer is physics and how much is integration.

The trap in all of them is the same one: rename the experiment, or the table will not say which row is which.

7.7 The comparison grid

Two engines runs do not necessarily sample at the same distances, so `analysis.lat:582` builds a grid rather than assuming one.

It works in three steps.

One: the common window. For each trajectory, take the first and last sampled distance. The window starts at the *largest* of the firsts and ends at the *smallest* of the lasts (`analysis.lat:612-617`). If the two windows do not overlap at all, the comparison refuses rather than inventing an overlap.

Two: the union grid. Collect every sampled distance from every series that falls inside the window, sort it, and drop near-duplicates (`analysis.lat:625-639`). Two runs sampled at the same interval collapse onto one shared grid; two runs sampled differently produce the union of both.

Three: independent interpolation. For each grid point, every series is evaluated at that exact distance via `trajectory_at` — exactly, if the series happens to have a sample there; linearly interpolated between its two nearest samples if not.

Here is all three at once. The same load solved twice: once with a 300-yard sampling interval out to 1200 yards, once with a 400-yard interval out to 800 yards.

Listing 7.11: Union grid: two runs, 300-yard and 400-yard sampling, 1200-yard and 800-yard range. Left half of the output, `cut -c1-123`

```
Trajectory comparison
Series | Trajectory               | Distance (yd) | Time (s) | Velocity (ft/s) | Energy (ft-lb) | Drop (in) | Windage (in)
-------+--------------------------+---------------+----------+-----------------+----------------+-----------+-------------
  0.00 | 140 gr ELD-M at 1,200 yd |          0.00 |     0.00 |         2710.00 |        2282.62 |     +1.80 |        +0.00
  1.00 | 140 gr ELD-M at 1,200 yd |          0.00 |     0.00 |         2710.00 |        2282.62 |     +1.80 |        +0.00
  0.00 | 140 gr ELD-M at 1,200 yd |        300.00 |     0.36 |         2338.37 |        1699.50 |    +12.58 |        +3.77
  1.00 | 140 gr ELD-M at 1,200 yd |        300.00 |     0.37 |         2343.51 |        1720.90 |    +21.52 |        +5.16
  0.00 | 140 gr ELD-M at 1,200 yd |        400.00 |     0.50 |         2224.88 |        1546.55 |    +35.41 |        +7.94
  1.00 | 140 gr ELD-M at 1,200 yd |        400.00 |     0.49 |         2221.35 |        1533.66 |    +28.10 |        +6.88
  0.00 | 140 gr ELD-M at 1,200 yd |        600.00 |     0.77 |         1997.90 |        1240.63 |    +81.08 |       +16.27
  1.00 | 140 gr ELD-M at 1,200 yd |        600.00 |     0.79 |         2004.08 |        1262.99 |    +98.57 |       +18.68
  0.00 | 140 gr ELD-M at 1,200 yd |        800.00 |     1.10 |         1789.50 |        1002.06 |   +179.56 |       +31.88
  1.00 | 140 gr ELD-M at 1,200 yd |        800.00 |     1.09 |         1786.81 |         992.32 |   +169.05 |       +30.49
Signs: drop + below/- above; windage + right/- left; deltas are series minus baseline [0]
```

The grid is 0, 300, 400, 600, 800 — the union of $\{0, 300, 600, 900, 1200\}$ and $\{0, 400, 800\}$, clipped to the window $[0, 800]$ because series 1 stopped at 800. The 900 and 1200 points from series 0 are outside the window and simply do not appear.

7.8 Interpolation artifacts

Now look at the delta columns of that same table.

Listing 7.12: The union-grid output, right half, `cut -c124-`

```
| Mach | Delta time (s) | Delta velocity (ft/s) | Delta energy (ft-lb) | Delta drop (in) | Delta windage (in) | Delta Mach
+------+----------------+-----------------------+----------------------+-----------------+--------------------+-----------
| 2.44 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.44 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.10 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.11 |          +0.01 |                 +5.14 |               +21.39 |           +8.95 |              +1.38 |      +0.00
| 2.00 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 2.00 |          -0.01 |                 -3.53 |               -12.89 |           -7.31 |              -1.06 |      +0.00
| 1.80 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.80 |          +0.02 |                 +6.18 |               +22.36 |          +17.50 |              +2.42 |      +0.01
| 1.61 |          +0.00 |                 +0.00 |                +0.00 |           +0.00 |              +0.00 |      +0.00
| 1.61 |          -0.01 |                 -2.69 |                -9.74 |          -10.51 |              -1.39 |      +0.00
```

Every nonzero delta in that table is a lie

Both series in the union-grid example are the *same rifle firing the same load into the same air*. The only difference between the two solves is the sampling interval the engine was asked to report on. The physics is identical. Every delta should be zero.
Instead the table claims the two loads differ by 17.50 inches of drop at 600 yards and 6.18 ft/s of velocity. Those numbers are pure linear-interpolation error. At 600 yards, series 1 (sampled every 400 yards) has no sample; it is interpolated on the straight line between its 400- and 800-yard samples, and a trajectory is not a straight line over 400 yards. At 400 yards it is series 0's turn to be interpolated, between 300 and 600, and the error flips sign.
The comparison table is only as good as the coarser of the two sampling grids. If you are going to compare, set the sampling interval fine enough that linear interpolation is honest — 25 yards over 1200 is 49 samples and costs nothing — and set it *before* both solves so the union grid is a single shared grid with no interpolation at all.

You can confirm that the artifact vanishes when the grids agree: in the 140 against 147 comparison at the top of this chapter, both runs sampled at 200 yards, both grids coincided, and the deltas at 0 and 200 yards are the small, believable numbers you would expect from a genuine 60 ft/s muzzle difference.

7.9 Controlling the size of the table

`:compare` has no start, no stop, and no interval. It emits the *entire* union grid, two rows per point.

With our experiment's normal 25-yard sampling over 1200 yards, that is 49 grid points, 98 table rows, 102 lines of table, and 198 lines of session output for a single `:compare`. Every one of those lines is 245 characters wide. That is not a table you read; it is a table you pipe somewhere.

The sampling interval is the compare table's row control

There is exactly one knob, and it lives on the experiment, not the command:

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(200), nil, true, nil))
```

The third argument is `sampling_interval`. At 200 yards over 1200 you get 7 grid points and 14 rows — the table printed at the start of this chapter, and one that fits on a page. Set it before the first of the two solves so both runs share the grid.
Coarse sampling costs you accuracy at unsampled distances (Section 7.8), so it is a genuine trade: 25 yards for numbers you will act on, 200 yards for a table you will read. If you want both, solve at 25 yards and use `:dope` for the reading.

The row count is bounded, not unbounded: `analysis.lat:640` rejects a comparison whose grid times series count would exceed 10 000 rows, and it checks before building anything.

7.10 Cross-checking against the engine

The `ballistics-engine` has its own `compare` subcommand, which solves two to eight loads from scratch at shared conditions. It is a different tool for the same question, and running both is the best confidence check available.

Listing 7.13: The engine's `compare`, same two loads, same atmosphere

```
$ ballistics compare \
    --load "140 gr ELD-M:g7:0.315:140:2710:0.264" \
    --load "147 gr ELD-M:g7:0.351:147:2650:0.264" \
    --zero-distance 100 --start 200 --end 1200 --step 200 \
    --wind-speed 8 --wind-direction 270 --sight-height 1.8 \
    --altitude 3937.0079 --temperature 53.6 --pressure 25.9871 --humidity 35 \
    --adjustment-unit mil -o csv
range_yd,140 gr ELD-M_drop_in,140 gr ELD-M_drop_MIL,140 gr ELD-M_drift_in,140 gr
     ELD-M_drift_MIL,140 gr ELD-M_velocity_fps,140 gr ELD-M_energy_ft-lb,140 gr ELD-M_tof_s,147
     gr ELD-M_drop_in,147 gr ELD-M_drop_MIL,147 gr ELD-M_drift_in,147 gr ELD-M_drift_MIL,147 gr
     ELD-M_velocity_fps,147 gr ELD-M_energy_ft-lb,147 gr ELD-M_tof_s
200,3.41,0.47,1.55,0.22,2459,1879,0.232,3.60,0.50,1.44,0.20,2427,1922,0.237
400,28.10,1.95,6.53,0.45,2221,1534,0.489,29.07,2.02,6.00,0.42,2215,1601,0.495
600,81.08,3.75,15.46,0.72,1998,1241,0.774,82.98,3.84,14.13,0.65,2014,1324,0.780
800,169.05,5.87,29.00,1.01,1787,992,1.092,171.30,5.95,26.34,0.91,1823,1085,1.093
1000,300.82,8.36,48.02,1.33,1586,782,1.448,301.64,8.38,43.30,1.20,1641,879,1.440
1200,488.22,11.30,73.63,1.70,1396,606,1.851,483.86,11.20,65.88,1.52,1467,703,1.826
```

Take the 1000-yard row for the 140 and set the two tools side by side.

Quantity	Ballistics Lab	Engine CLI	Reconciliation
Velocity (ft/s)	1586.32	1586	identical
Energy (ft-lb)	782.12	782	identical
Time of flight (s)	1.45	1.448	identical
Drop (in)	300.82	300.82	identical
Windage (in)	50.43	48.02	+2.41 = Coriolis

Four of the five agree outright, and the drop agreement is exact to the hundredth of an inch: two tools, two code paths — `solve-json` against command-line flags — one number. That is worth more as a confidence check than any amount of reasoning about reference frames, and it holds down the

whole column: 3.41, 28.10, 81.08, 169.05 and 300.82 inches on both sides at 200 through 1000 yards. Only at 1200 do they part, 488.20 against 488.22, by two hundredths of an inch.

The one real discrepancy is explained and worth internalising.

- **Windage differs by the Coriolis term.** Our experiment enables Coriolis at 44° latitude; the CLI invocation above does not (it takes neither `--enable-coriolis` nor `--latitude`). In still air our experiment shows $+2.42$ inches of right drift at 1000 yards, and $50.43 - 2.42 = 48.01$ against the engine's 48.02.

What to do when the elevations *do* disagree

If you cross-check a Ballistics Lab card against a third solver and the two elevation columns differ by an amount proportional to range, the first thing to check is what height each tool solved its zero to. The Ballistics Lab zeroes to the line of sight and says so on the wire (Section 6.4.1); a tool that zeroes to the ground instead will differ by the sight height scaled by R/R_{zero} — 18 inches at 1000 yards for a 1.8-inch sight on a 100-yard zero. It is a datum disagreement, not a physics one, and it will not show up in the velocity or time-of-flight columns.

Two comparison tools, two shapes

The engine's table is *wide*: one block of columns per load, one row per distance. The Ballistics Lab's is *long*: one row per load-distance pair, with an explicit `series_index`. Wide is more compact for two loads and unreadable for six; long is constant-shape for any number of series and is what a dataframe wants. `analysis_export.lat` exists to hand the long form straight to `pandas.read_csv` with the provenance repeated on every row.
The engine's compare also re-solves from scratch, so it can compare eight loads in one invocation. The Ballistics Lab's reads two already-solved runs, so it can compare a load against *itself under different conditions* — a different atmosphere, a different wind, a different solver — which the engine's compare cannot express at all.

7.11 More than two series

The analysis layer is not limited to two. `analysis.lat:591` requires *at least* two trajectories and imposes no upper bound, so this is a perfectly good three-way comparison. The session below is back on the default 25-yard sampling interval, so its grid has 49 points:

Listing 7.14: A three-series comparison built at the prompt

```
let a = sessions.previous_run(lab)
let b = sessions.current_run(lab)
let t = analysis.compare_trajectories([a.result, b.result, b.result],
    "yd", "in", "in", "ft/s", "ft-lb")
len(t.rows)
```

```
147
```

Forty-nine grid points times three series — the table is built and it is correct. But the terminal renderer refuses it:

Listing 7.15: `render.render_comparison` insists on exactly two runs

```
Error
  code: evaluation_error
  message: assertion failed: render.comparison.provenance: exactly two runs are required
```

`render.lat:976` enforces the two-series rule, because the comparison's sign legend (`deltas are series minus baseline [0]`) and its column widths are written for a pairwise reading. A three-way table would need a different legend and a different story about what the deltas mean.

Three or more loads: export instead of render

An `AnalysisTable` with any number of series can be turned into JSON, CSV, or plain text by `analysis_export.lat`, which validates the table structurally and has no two-series restriction:

```
import "examples/ballistics_lab/analysis_export" as tables
print(tables.table_to_csv(t))
```

The alias matters: `export` is a LATTICE keyword, and `as export` is a parse error — the BALLISTICS LAB reports `'export' must precede fn, struct, enum, trait, or variable binding`. Any non-reserved name works.
The CSV carries fourteen provenance columns repeated on every row, so `pandas.read_csv` gives you the full audit trail without a sidecar file. The rendering and analysis chapter covers the export module properly.

7.12 Errors

Condition	Code	Raised by
No current run	`command_state_error`	`commands.lat:187`
No previous run	`command_state_error`	`commands.lat:187`
Any argument at all	`command_arity`	`command_parser.lat:286`
Fewer than two series	`command_execution_error`	`analysis.lat:591`
No shared range	`command_execution_error`	`analysis.lat:622`
Over 10 000 rows	`command_execution_error`	`analysis.lat:640`
Not exactly two series (render)	`command_execution_error`	`render.lat:976`

`:compare` is read-only. It does not solve, does not touch the experiment, and does not advance the run pair. You can run it as many times as you like; the answer will not change until you solve again.

A comparison worth saving

1. Set the sampling interval once, before either solve, fine enough to avoid interpolation error.
2. Solve the baseline. Check its `termination` line.
3. Change exactly one thing.
4. Rename the experiment so the `Trajectory` column means something.
5. Solve again, then `:compare`.
6. Read the series-1 rows. Ignore the series-0 zeros.
7. If the answer matters, promote both runs to verified artifacts — the verified-artifacts chapter shows how — so the comparison survives the session.

Chapter 8

Units and Quantities

A ballistics program that gets units wrong does not crash. It produces a plausible number that is wrong by a factor of 0.9144, and you find out on paper at 800 yards.

The Ballistics Lab treats this as the primary hazard it exists to eliminate. There are no bare numbers in the domain model. A muzzle velocity is not `2710`; it is `fps(2710)`, a `Velocity` that knows it means 826.008 m/s and remembers that you said it in feet per second. This chapter is about that machinery: what it buys, what it costs to type, and where it hands off to the `ballistics-engine`.

8.1 Why a quantity is a type

Quantity

A *quantity* in the Ballistics Lab is an immutable struct carrying three things: a canonical SI value, the value you actually typed, and the unit you typed it in. `units.lat` defines eight of them, one per physical dimension, and every domain value in the Ballistics Lab is built from them.

The header comment on `units.lat` states the intent in four lines:

Listing 8.1: `units.lat:1-12` — the module's thesis and its first struct

```
// Typed quantities for the source-level ballistics laboratory.
//
// Each value stores a canonical SI value and the exact value/unit requested
// for display.  Every struct field is an alloy `fix` field, so callers can
// inspect quantities but cannot silently change either their dimension or
// their display metadata.

export struct Distance {
    si_value: fix Number,
    display_value: fix Number,
    display_unit: fix String
}
```

`fix` fields (for readers coming from the shooting side)

Lattice values carry a *phase*. A **flux** value may be mutated; a **fix** value may not. Marking every field of a struct `fix` makes the struct immutable without freezing it into a separate memory region — the Ballistics Lab calls this an *alloy* value. The practical consequence is that once you have built `fps(2710)`, nothing anywhere can quietly change its `si_value` to something that disagrees with its `2710 ft/s` label. The architecture chapters go into the phase system properly.

The payoff is that dimensions cannot be crossed by accident. Wind speed and muzzle velocity have the same physical dimension — both are lengths per time — and the Ballistics Lab still gives them separate types, with the reason written into the source:

Listing 8.2: `units.lat:415-417`

```
// Wind speed remains a distinct type even though it has the same dimension as
// Velocity. This prevents a muzzle velocity from being accepted as wind.
```

Try it and the Ballistics Lab stops you before the engine ever sees the request:

Listing 8.3: `wind(fps(10), deg(90))` — a velocity where a wind speed belongs

```
Error
  code: evaluation_error
  message: assertion failed: wind.speed: expected WindSpeed, got Velocity
```

8.2 The eight quantities

Type	Canonical SI	Constructors	Converter units
`Distance`	metre	`m km yd ft inches`	`m km yd ft in`
`Velocity`	metre/second	`mps fps`	`m/s ft/s`
`Mass`	kilogram	`kg grams gr`	`kg g gr`
`Angle`	radian	`rad deg mil moa`	`rad deg mil moa`
`Temperature`	kelvin	`kelvin celsius fahrenheit`	`K C F`
`Pressure`	pascal	`pa hpa inhg`	`Pa hPa inHg`
`Energy`	joule	`joules foot_pounds`	`J ft-lb`
`WindSpeed`	metre/second	`wind_mps mph kph`	`m/s mph km/h`

Two naming details catch everyone once. The distance constructor for inches is `inches`, not `in` — `in` is a LATTICE keyword (as in `for x in xs`) and cannot be a function name. But the *converter* unit string *is* `"in"`. Similarly the kilometres-per-hour constructor is `kph` while its unit string is `"km/h"`.

Constructors take a number; converters take a quantity

`yd(100)` builds a `Distance` from a number. `units.distance_as(d, "m")` takes a `Distance` and returns a plain number. The two never appear on the same side of an expression, and the converter is the only way back to a bare number.

8.3 The three fields

Type a constructor at the BALLISTICS LAB prompt and the REPL shows you all three fields:

Listing 8.4: `yd(100)` at the prompt, then `yd(100).si_value`

```
Distance { si_value: 91.44, display_value: 100, display_unit: yd }
91.44
```

`si_value`
: The canonical value, always in the SI unit for that dimension. This is the only field that ever crosses the process boundary to the engine (Section 8.10).

`display_value`
: Exactly the number you typed. Not rounded, not re-derived from `si_value`. If you said `yd(100)` this stays `100` forever, even though $100 \times 0.9144 \div 0.9144$ is not guaranteed to be exactly 100 in binary floating point.

`display_unit`
: The unit string you typed it in. This is what gets persisted to disk (the persistence chapter) and what a renderer falls back to when nobody has said otherwise.

That separation is the whole design. The engine gets a number it can trust; you get your number back unchanged; and the pair can be cross-checked against each other, which the persistence layer does on every save:

Listing 8.5: `persistence.lat:107-121` — the round-trip check on save

```
    // Reject non-finite canonical state even though canonical SI is intentionally
    // not persisted. This prevents a forged quantity from hiding invalid state.
    ...
    let reconstructed = _quantity_from_map_for_type(out, expected, path)
    if reconstructed.si_value != value.si_value {
        _fail(path + ": canonical SI value disagrees with display value/unit")
    }
```

8.4 Conversion, and the constants that do it

Every converter is the same shape: divide the canonical value by a constant, or fall through to a diagnostic. Here is the whole of distance conversion:

Listing 8.6: `units.lat:142-160` — `distance_as`

```
export fn distance_as(value: Distance, unit: String) -> Number {
    let si_value = _distance_si(value)
    if unit == "m" {
        return si_value
    }
    if unit == "km" {
        return si_value / METERS_PER_KILOMETER
    }
    if unit == "yd" {
        return si_value / METERS_PER_YARD
    }
    if unit == "ft" {
        return si_value / METERS_PER_FOOT
    }
    if unit == "in" {
        return si_value / METERS_PER_INCH
    }
    _unsupported("distance_as", unit, "m, km, yd, ft, in")
}
```

The constants are declared once, at `units.lat:56-70`, and every one of them is an exact definition rather than a measured approximation:

Constant	Value	Note
`METERS_PER_KILOMETER`	1000.0	
`METERS_PER_YARD`	0.9144	exact by international agreement
`METERS_PER_FOOT`	0.3048	exact
`METERS_PER_INCH`	0.0254	exact
`KILOGRAMS_PER_GRAM`	0.001	
`KILOGRAMS_PER_GRAIN`	0.00006479891	exact
`RADIANS_PER_DEGREE`	$\pi/180$	
`RADIANS_PER_MIL`	0.001	a mil is a milliradian
`RADIANS_PER_MOA`	$\pi/10800$	the exact minute of angle
`PASCALS_PER_HECTOPASCAL`	100.0	
`PASCALS_PER_INHG`	3386.389	
`JOULES_PER_FOOT_POUND`	1.3558179483314004	
`METERS_PER_SECOND_PER_MPH`	0.44704	exact
`METERS_PER_SECOND_PER_KPH`	$1/3.6$	

You can read any of them back at the prompt. `to_string` shortens to six significant figures, so wrap the call in `json_stringify` when you want every digit the runtime has:

Listing 8.7: Eight conversions typed at the Ballistics Lab prompt

```
json_stringify(units.angle_as(rad(1), "moa"))
json_stringify(units.angle_as(mil(1), "moa"))
json_stringify(units.pressure_as(inhg(29.92), "hPa"))
json_stringify(units.pressure_as(hpa(1013.25), "inHg"))
json_stringify(units.mass_as(gr(140), "kg"))
json_stringify(units.velocity_as(fps(2710), "m/s"))
json_stringify(units.energy_as(foot_pounds(1), "J"))
json_stringify(units.distance_as(inches(1), "m"))
```

Listing 8.8: The eight replies, in order

```
3437.7467707849396
3.4377467707849396
1013.2075888000001
29.921252401894762
0.0090718473999999993
826.00800000000004
1.3558179483314003
0.025399999999999999
```

29.92 inHg is not 1013.25 hPa

The third and fourth lines above are worth a moment. The imperial standard atmosphere is quoted as 29.92 inHg and the metric one as 1013.25 hPa, and people use them interchangeably. They are not the same pressure: 29.92 inHg is 1013.2076 hPa, and 1013.25 hPa is 29.9213 inHg. The gap is 4 Pa — utterly negligible for a firing solution, and exactly the sort of thing that makes two solvers disagree in the sixth digit when you are trying to establish that they agree at all.

8.5 The mil and the MOA

Angles get their own section because the field is full of near-synonyms.

Listing 8.9: `units.lat:238-239` — the comment above the angle constructors

```
// Angle: canonical radians. Signed angles are valid. Ballistic mil is exactly
// one milliradian, not an artillery mil or an angular approximation.
```

`mil`
: Exactly 0.001 rad. There are about 6283.19 of them in a circle. Some artillery systems divide the circle into 6400 "mils" and some into 6000; neither is what a rifle scope means, and neither is what the Ballistics Lab means.

`moa`
: Exactly $\pi/10800$ rad — one sixtieth of one degree. One radian is `3437.7467707849396` MOA, and one mil is `3.4377467707849396` MOA.

The Ballistics Lab's MOA and the engine's printed MOA are not the same number

`units.lat` uses the exact $\pi/10800$. The `ballistics-engine`'s *printed tables* deliberately use the round constant 3438 mil-to-MOA instead — it is locked in on purpose, because dope cards have been printed with 3438 for decades and changing it would silently move everyone's data. The discrepancy is $3438/3437.7467 = 1.0000737$, or 7 parts per hundred thousand. On a 30 MOA come-up that is 0.002 MOA: invisible. It matters in exactly one situation — when you are comparing the Ballistics Lab's angular output to the engine's printed table digit by digit and wondering why the last place disagrees. Now you know.

The engine's `solve-json` interface, which is what the Ballistics Lab actually speaks, reports *no* angles in MOA at all. It reports metres and radians, and the Ballistics Lab does every angular conversion itself. So the 3438 constant never enters a Ballistics Lab number.

8.6 What may be negative

Not every dimension accepts a negative value, and the choices are deliberate. `units.lat` has two private validators: `_scaled_finite`, which accepts any finite number, and `_scaled_nonnegative`, which does not.

Type	Signed?	Why
`Distance`	yes	drop and windage are offsets, not lengths
`Velocity`	yes	velocity components have direction
`Angle`	yes	a hold can be left or right, up or down
`Mass`	no	
`Pressure`	no	absolute magnitude
`Energy`	no	
`WindSpeed`	no	direction lives in `direction_from`
`Temperature`	special	see Section 8.7

The source says so directly:

Listing 8.10: `units.lat:105-107`

```
// Distance: canonical meters. Generic distances may be signed so the same
// quantity can represent left/right windage and above/below trajectory offsets.
// Domain constructors enforce positivity for ranges and physical dimensions.
```

That last sentence matters: `m(-5)` is a perfectly good `Distance`, but `shot(fps(2710), m(-5), ...)` is not a perfectly good shot. The unit layer permits the sign; the domain layer (`domain.lat`) rejects it where it makes no sense. Two layers, two jobs.

Listing 8.11: `m(-5)`, then `mph(-5)`, then `gr(-1)`

```
Distance { si_value: -5, display_value: -5, display_unit: m }
Error
  code: evaluation_error
  message: assertion failed: mph: value must be non-negative
Error
  code: evaluation_error
  message: assertion failed: gr: value must be non-negative
```

8.7 Temperature is special

Temperature is the one dimension where the conversion is affine rather than a scaling, and `units.lat` handles it separately as a result. The constructor converts *first* and checks the absolute-zero bound *after*:

Listing 8.12: `units.lat:289-320` — conversion before validation

```
fn _temperature(
kelvin_value: Number,
display_value: Number,
unit: String,
context: String
) -> Temperature {
    let canonical = _finite_number(kelvin_value, context + " canonical SI value")
    if canonical < 0 {
        assert(false, context + ": temperature must not be below absolute zero")
    }
    ...
}

export fn celsius(value: any) -> Temperature {
    let display = _finite_number(value, "celsius")
    return _temperature(display + 273.15, display, "C", "celsius")
}

export fn fahrenheit(value: any) -> Temperature {
    let display = _finite_number(value, "fahrenheit")
    let canonical =(display - 32.0) *(5.0 / 9.0) + 273.15
    return _temperature(canonical, display, "F", "fahrenheit")
}
```

Ordering it this way is what lets $-40\ ^\circ\mathrm{C}$ be legal (it is 233.15 K) while $-300\ ^\circ\mathrm{C}$ is not:

Listing 8.13: `celsius(-300)`

```
Error
  code: evaluation_error
  message: assertion failed: celsius: temperature must not be below absolute zero
```

Note also that `_temperature` takes the kelvin value and the display value as *separate arguments*, which no other dimension needs. For a scaling conversion the display value can always be recovered by dividing; for an affine one it cannot be recovered without knowing which scale you started in.

8.8 Two layers of type checking

The Ballistics Lab gets its type discipline from two different mechanisms, and their error messages look different enough to tell apart at a glance.

Lattice's own parameter annotations. The converter's signature is

```
export fn velocity_as(value: Velocity, unit: String) -> Number
```

and the runtime enforces the annotation before the body runs:

Listing 8.14: `units.velocity_as(mph(10), "m/s")`

```
Error
  code: evaluation_error
  message: function 'velocity_as' parameter 'value' expects type Velocity, got WindSpeed
```

Structural checks in the domain layer. `domain.lat` does not import `units.lat` at all. It checks quantities by asking for their struct name:

Listing 8.15: `domain.lat:178` — structural typing without an import

```
fn _require_quantity(value: any, expected: String, field: String) {
    if typeof(value) != "Struct" || struct_name(value) != expected {
```

Listing 8.16: `projectile("x", inches(0.264), gr(140), 0.315, "G7")` — mass and diameter swapped

```
Error
  code: evaluation_error
  message: assertion failed: projectile.mass: expected Mass, got Distance
```

The second style is what catches argument-order mistakes, which are the ones you will actually make. `projectile` takes mass then diameter; a bullet diameter typed where the mass goes is caught by name, immediately, before the value can reach the wire.

Finally, the unit *strings* are validated by the converters themselves, and the diagnostic always lists what would have been acceptable:

Listing 8.17: `units.distance_as(m(1), "furlong")`

```
Error
  code: evaluation_error
  message: assertion failed: distance_as: unsupported unit 'furlong'; supported: m, km, yd, ft,
     in
```

Converters used purely as validators

`analysis.lat` exploits that diagnostic deliberately. Before building a DOPE table it runs four throwaway conversions of zero:

```
units.distance_as(units.m(0), drop_unit)
units.angle_as(units.rad(0), correction_unit)
units.velocity_as(units.mps(0), velocity_unit)
units.energy_as(units.joules(0), energy_unit)
```

The return values are discarded. The point is the *failure*: if a caller asked for a table in furlongs, the table-builder fails immediately with the canonical unit diagnostic rather than half-way through row 47.

8.9 Display preferences

A quantity remembers the unit it was built with. A *session* has a separate opinion about how to show things, and that opinion is what every renderer consults.

Listing 8.18: `session.lat:162-172` — the five knobs and their permitted values

```
export fn display_preferences(
distance_unit: any = "yd",
velocity_unit: any = "ft/s",
energy_unit: any = "ft-lb",
angle_unit: any = "mil",
precision: any = 2
) -> DisplayPreferences {
    let checked_distance = _choice(distance_unit, ["m", "km", "yd", "ft", "in"],
    "display.distance_unit")
    let checked_velocity = _choice(velocity_unit, ["m/s", "ft/s"],
    "display.velocity_unit")
    let checked_energy = _choice(energy_unit, ["J", "ft-lb"], "display.energy_unit")
    let checked_angle = _choice(angle_unit, ["rad", "deg", "mil", "moa"],
    "display.angle_unit")
```

Five knobs, each validated against a closed list, plus a precision between 0 and 12. There is no `:set` command; you change them by calling the function:

```
sessions.set_display_preferences(lab, sessions.metric_display_preferences(2))
sessions.set_display_preferences(lab, sessions.display_preferences("km", "m/s", "J",
    "mil", 3))
```

Ask for something outside the list and you are told what the list is:

Listing 8.19: An unsupported velocity unit and an out-of-range precision

```
Error
  code: evaluation_error
  message: assertion failed: display.velocity_unit: unsupported value 'mph'; expected one of
     [m/s, ft/s]
Error
  code: evaluation_error
  message: assertion failed: display.precision: must be between 0 and 12
```

Note that `mph` is rejected as a *velocity* unit. The wind speed unit is not one of the five knobs — it is derived. `render.lat` keeps three small policy functions that turn the five knobs into the units the analysis layer must calculate in:

Policy	Metric family (`m`, `km`)	Otherwise
`offset_unit_for` (drop, windage)	`m`	`in`
`wind_speed_unit_for`	`m/s`	`mph`
`direction_unit_for`	`deg`	`deg`
`_mass_unit` (private)	`g`	`gr`
`_temperature_unit` (private)	`C`	`F`
`_pressure_unit` (private)	`hPa`	`inHg`

So choosing `"m"` or `"km"` as the distance unit flips the whole session into metric, including quantities you never named. Here is the same experiment under both profiles — the values in the `Experiment` struct are identical in both cases; only the rendering changed.

Listing 8.20: `:show` with the default imperial profile; the block continues past `zero distance` and is cut here

```
Experiment: 140 gr ELD-M at 1,200 yd
  projectile           : 140 gr ELD-M
  mass                 : 140.00 gr
  diameter             : 0.26 in
  length               : 1.34 in
  drag model           : G7
  ballistic coefficient: 0.32
  rifle                : Tikka T3x CTR 24 in
  sight height         : 1.80 in
  muzzle height        : 0.00 in
  twist rate           : 8.00 in
  twist direction      : right
  altitude             : 1312.34 yd
  temperature          : 53.60 F
  pressure             : 25.99 inHg
  humidity             : 35.00 %
  latitude             : +44.00 deg
  muzzle velocity      : 2710.00 ft/s
  maximum range        : 1200.00 yd
  zero distance        : 100.00 yd
```

Listing 8.21: The same experiment after `sessions.set_display_preferences(lab, sessions.metric_display_preferences(2))`, cut at the same line

```
Experiment: 140 gr ELD-M at 1,200 yd
  projectile           : 140 gr ELD-M
  mass                 : 9.07 g
  diameter             : 0.01 m
  length               : 0.03 m
  drag model           : G7
  ballistic coefficient: 0.32
  rifle                : Tikka T3x CTR 24 in
  sight height         : 0.05 m
  muzzle height        : 0.00 m
  twist rate           : 0.20 m
  twist direction      : right
  altitude             : 1200.00 m
  temperature          : 12.00 C
  pressure             : 880.00 hPa
  humidity             : 35.00 %
  latitude             : +44.00 deg
  muzzle velocity      : 826.01 m/s
  maximum range        : 1097.28 m
  zero distance        : 91.44 m
```

Precision is global, and small lengths suffer

The precision knob applies to *every* numeric field. At two decimals, a 6.7 mm bullet diameter renders as `0.01 m` in the metric profile and `0.26 in` in the imperial one — neither of which is the 0.264 inch value you entered. The length, at 34 mm, renders as `0.03 m`. Nothing is lost internally; the `si_value` is still 0.0067056, and the request that goes to the engine is exact (Section 8.10). It is a display artefact — but it is a display artefact in the one panel you would use to check that you typed the bullet in correctly.

Raise the precision when you are proof-reading an experiment: `sessions.display_preferences("m", "m/s", "J", "mil", 5)`.

Altitude renders in the distance unit

The imperial profile reports `altitude : 1312.34 yd`. That is correct — 1200 m is 1312.34 yards — and it is an odd way to state a station altitude, which the shooting world quotes in feet. The renderer has one distance unit and applies it everywhere, and altitude is a distance. If you set your experiment's altitude with `ft(3937)` the underlying quantity would remember feet, but the `:show` panel would still print yards, because the panel asks the profile, not the quantity.

8.10 Display in, SI out: the wire

Everything above is about what *you* see. The engine sees none of it.

`protocol_v1.lat` builds the solve request by reading `si_value` and nothing else. Display values and unit strings are not serialized — there is no code path that could put them on the wire. Every field name on the wire carries its SI unit as a suffix, so the contract is self-describing: `_m`, `_mps`, `_kg`, `_rad`, `_k`, `_pa`, `_j`, `_s`.

Here is the engine's *resolved request* for the experiment shown above — the engine's own echo of what it decided to solve, retrieved from the current run record with `let r = sessions.current_run(lab)` and written out with `write_file`:

Listing 8.22: `resolved_request`, written to a file and re-read through `python3 -m json.tool --sort-keys`. The wire form is one compact line

```
{
    "atmosphere": {
        "altitude_m": 1200,
        "latitude_rad": 0.767944870877505,
        "pressure_pa": 88000,
        "relative_humidity": 0.35,
        "temperature_k": 285.15
    },
    "effects": {
        "coriolis": true,
        "enhanced_spin_drift": false,
        "magnus": false
    },
    "projectile": {
        "ballistic_coefficient": 0.315,
        "diameter_m": 0.0067056,
        "drag_model": "G7",
        "length_m": 0.034036000000000004,
        "mass_kg": 0.0090718474
    },
    "rifle": {
        "muzzle_height_m": 0,
        "muzzle_velocity_mps": 826.008,
        "sight_height_m": 0.04572,
        "twist_direction": "right",
        "twist_rate_m_per_turn": 0.2032
    },
    "sampling": {
        "interval_m": 22.86
    },
    "schema_version": 1,
    "shot": {
        "aim_azimuth_rad": 0,
        "cant_angle_rad": 0,
        "ground_threshold_m": -100,
        "max_range_m": 1097.28,
        "muzzle_angle_rad": 0.0011795043945312502,
        "shooting_angle_rad": 0,
        "shot_azimuth_rad": 0,
        "target_height_m": 0.04572,
        "zero_distance_m": 91.44
    },
    "solver": {
        "method": "rk45",
        "time_step_s": 0.001
    },
    "wind": {
        "direction_from_rad": 4.71238898038469,
        "speed_mps": 3.57632,
        "vertical_speed_mps": 0
    }
}
```

Every number we typed in imperial is there in SI: 2710 ft/s became `826.008` m/s, the 1.8 inch sight height became `0.04572` m, 8 mph became `3.57632` m/s, the 270° wind became `4.71238898038469` radians, and 880 hPa became `88000` pascals exactly. The 1200 yard maximum range became `1097.28` m and the 100 yard zero `91.44` m — and note `muzzle_angle_rad`, which we never supplied: the engine solved the zero and wrote back the elevation it used.

There is a second field we never typed, and it is the only number on the whole wire that the BALLISTICS LAB *calculates* rather than copies. `target_height_m` reads `0.04572` m — the same value as `sight_height_m` in the `rifle` object above. It is the height above the ground that the zero is solved *to*, and the engine documents its default for it as ground level, `0.0`. A conventional zero puts the target on the line of sight instead, so when an experiment leaves `Shot.target_height` unset — ours does; `:show` still prints `target height : default` — `protocol_v1.lat:74-91` adds the rifle's muzzle height to its sight height and sends the sum. Here that is $0 + 0.04572$ m. Every other value in the request is some quantity's `si_value` copied verbatim; this one is arithmetic, and it is arithmetic done in SI, on `si_value` fields, for the same reason everything else on this page is.

Note `relative_humidity: 0.35`. The wire wants a *fraction* between 0 and 1, and `domain.lat` takes it as a fraction too — our experiment was built with `0.35`, even though the `:show` panel prints `humidity : 35.00 %`. Humidity is the one input in the whole model that is a bare number rather than a quantity, and it is the one place where a factor of 100 can bite you.

Humidity is a fraction on input, a percentage on display

`atmosphere(m(1200), celsius(12), hpa(880), 0.35, deg(44))` — that fourth argument is `0.35`, not `35`. It is the one input in the whole model that is a bare number rather than a quantity, so there is no unit label to remind you which convention you are in.
`domain.lat:313-315` does check the range, which turns a silent hundred-fold error into a loud one:

```
Error
  code: evaluation_error
  message: assertion failed: atmosphere.relative_humidity: must be between 0 and 1
```

The genuinely dangerous case is the one the check cannot catch: typing `0.5` when you meant 50% is correct, and typing `0.05` when you meant 50% is a legal 5% humidity. Confirm on the `:show` panel, which prints the percentage: `humidity : 35.00 %`.

8.10.1 `-units` does not reach `solve-json`

The `ballistics-engine` has a global `-u/--units` flag, defaulting to `imperial`, that changes how every other subcommand interprets its flags and formats its tables. It has no effect whatsoever on the interface the BALLISTICS LAB uses. That is verifiable in one command:

Listing 8.23: The same request, solved with and without `-u metric`

```
$ ballistics solve-json < minreq.json > resp_default.json
$ ballistics -u metric solve-json < minreq.json > resp_metric.json
$ cmp -s resp_default.json resp_metric.json && echo BYTE-IDENTICAL
BYTE-IDENTICAL
```

`solve-json` is SI in, SI out, always. The unit system is a property of the Ballistics Lab's display layer and of the engine's *human* interfaces, and the two never meet.

8.11 Where the engine's unit convention does matter

If you drop out of the Ballistics Lab and drive the `ballistics-engine` directly — to cross-check a card, or to use a subcommand the Ballistics Lab does not wrap — you are back in a world of flags whose units depend on a global switch. Three facts from the binary's own help text are worth carrying:

Listing 8.24: Three flags, three different unit conventions, straight from `-help`

```
$ ballistics trajectory --help | grep -A1 -- '--bore-height <'
      --bore-height <BORE_HEIGHT>
          Bore height above ground (inches for imperial, mm for metric; MBA-1339 unified this
     with --sight-height and the WASM --muzzle-height flag). Default: 60in/1500mm (= 5ft/1.5m).
     Also accepts --muzzle-height

$ ballistics trajectory --help | grep -A1 -- '--sample-interval <'
      --sample-interval <SAMPLE_INTERVAL>
          Sampling interval in meters (default: 10) Trajectory sampling interval in meters (used
     with --sample-trajectory; always metric, not unit-system dependent)

$ ballistics mpbr --help | grep -A1 -- '--vital-zone <'
      --vital-zone <VITAL_ZONE>
          Vital zone diameter (inches for imperial, cm for metric)
```

- `--bore-height` is **inches** in the imperial system, and its default is 60 — five feet, not five inches. Enter `5` thinking in feet and you have modelled a rifle fired from five inches off the ground.
- `--sample-interval` is **always metres**, in both systems.
- Metric small lengths are **mm on some flags and cm on others**: `--diameter` and `--sight-height` are mm, `--vital-zone` is cm.

None of this reaches the Ballistics Lab, which is the point. The Ballistics Lab's constructors name their unit in the call — `inches(1.8)`, `m(1200)`, `hpa(880)` — so there is no global mode to be in the wrong one of.

Unit habits that pay off

1. Type each value in the unit you measured it in. `inches(1.8)` for a scope height you measured with calipers, `m(1200)` for an altitude off a map. The Ballistics Lab converts; you do not.
2. Set the display profile to match how you *think*, not how you typed. They are independent, and the session's profile is what the tables use.
3. Raise the precision to 4 or 5 while proof-reading an experiment, then drop it back to 2 for cards.
4. Remember humidity is a fraction.
5. If a number disagrees with another solver in the third digit, check the sight-height reference and the MOA constant before you suspect the physics.

Chapter 9

Saving, Loading, and Resetting

A session is a working surface. You set a rifle up, solve it, card it, tweak something, solve again — and at the end of the evening every one of those values lives in a process that is about to exit.

Three commands manage what survives: `:save` writes the experiment to a file, `:load` reads one back, and `:reset` clears the run state. All three are narrower than they look, and one of them — the combination of `:load` with a stale run — can hand you a number that belongs to a rifle you no longer have loaded. This chapter is about getting those boundaries exactly right.

9.1 What is worth saving

The Ballistics Lab's persistence layer stores one thing: the *experiment*. Not the run, not the trajectory, not the session, not the backend, not your display preferences. The module says so in its first paragraph:

Listing 9.1: `persistence.lat:1-6` — the module's scope, stated up front

```
// Strict, versioned persistence for source-level ballistics experiments.
//
// Persistence stores only the immutable domain experiment. Backend closures,
// process state, session Refs, and run records are deliberately outside this
// document. Quantities retain their requested display value/unit; canonical SI
// values are reconstructed by the public units constructors while loading.
```

That is a real design decision and it is worth defending before we look at bytes.

An *experiment* is a question: this bullet, this rifle, this air, this wind, this shot. It is small, it is entirely made of validated domain values, and it is reproducible — given the same experiment and the same engine you get the same trajectory, every time. So a saved experiment is a durable, portable statement of intent.

A *run* is an answer, and answers are cheap to regenerate and expensive to store honestly. A trajectory can carry ten thousand samples. Worse, a stored answer without the engine version, the resolved request, and the notices that produced it is a number with no provenance — exactly the thing this whole program exists to avoid. The Ballistics Lab does have a format for storing answers: the *verified run* artifact, which carries all of that plus a cross-check that every field agrees with its embedded trajectory. That is a separate module and a separate chapter.

Two documents, two purposes

`lattice-ballistics/experiment`
: What you want solved. Written by `:save`, read by `:load`. Small, editable, diff-able. This chapter.

`lattice-ballistics/verified-run`
: What the engine answered, with the request it actually resolved, the engine version, and the notices. Written by `verified_artifacts.lat`. The artifacts chapter.

9.2 The document

`:save` takes one argument, a path, quoted:

Listing 9.2: `:save` with an absolute path; the middle of the path is abbreviated here

```
Saved experiment to /private/tmp/.../p2b-tables/eld-m.json
```

The result is 1281 bytes of compact JSON on one line. Pretty-printed, it looks like this — and this is the whole file, nothing omitted:

Listing 9.3: `python3 -m json.tool < eld-m.json`

```
{
    "experiment": {
        "rifle": {
            "twist_direction": "right",
            "name": "Tikka T3x CTR 24 in",
            "sight_height": {
                "unit": "in",
                "value": 1.8
            },
            "muzzle_height": {
                "unit": "m",
                "value": 0
            },
            "twist_rate": {
                "unit": "in",
                "value": 8
            }
        },
        "atmosphere": {
            "pressure": {
                "unit": "hPa",
                "value": 880
            },
            "relative_humidity": 0.35,
            "temperature": {
                "unit": "C",
                "value": 12
            },
            "latitude": {
                "unit": "deg",
                "value": 44
            },
            "altitude": {
                "unit": "m",
                "value": 1200
            }
        },
        "name": "140 gr ELD-M at 1,200 yd",
        "shot": {
            "cant_angle": null,
            "ground_threshold": null,
            "target_height": null,
            "max_range": {
                "unit": "yd",
                "value": 1200
            },
            "muzzle_angle": null,
            "shot_azimuth": null,
            "shooting_angle": null,
            "aim_azimuth": null,
            "muzzle_velocity": {
                "unit": "ft/s",
                "value": 2710
            },
            "zero_distance": {
                "unit": "yd",
                "value": 100
            }
        },
        "solver": {
            "method": "rk45",
```

Four things about that document are load-bearing.

9.2.1 Display values, not SI

Every quantity is stored as `{"unit": ..., "value": ...}` in the unit you typed it in. The sight height is `1.8` inches, not `0.04572` metres. The wind is `8` mph, not `3.57632` m/s.

(One caveat about the listing above: it is `python3 -m json.tool`'s rendering, and Python prints the shortest decimal that round-trips. The bytes on disk spell the humidity `0.34999999999999998`. No information differs; the two spellings name the same double.)

This is the exact opposite of the choice `protocol_v1.lat` makes for the wire, where only `si_value` crosses the boundary and display metadata is deliberately unrepresentable. The two choices are consistent because they serve different readers. The engine wants an unambiguous number; a person opening this file six months from now wants to recognise their own rifle.

The saved file is meant to be read and edited

Because it stores display units, an experiment document is a perfectly reasonable thing to edit in a text editor, put in version control, or diff against another load. Changing `"value": 2710` to `"value": 2745` and reloading is a legitimate way to true a velocity. The loader re-validates everything, so a bad edit is caught with a JSON path naming the field (Section 9.8).

9.2.2 Optional fields are explicit nulls

`"cant_angle": null`, `"magnus": null`, `"time_step_s": null`. Nothing you did not set is omitted — it is present and null.

That is deliberate, and again it is the opposite of the wire format, where an unset optional field is simply absent from the request Map. The loader uses the strictness: `persistence.lat:89-103` requires the key set to match *exactly*, rejecting both missing keys and unknown ones. A document written by an older version of the BALLISTICS LAB that lacked a field would be rejected rather than silently defaulted.

9.2.3 Key order is not part of the format

Read the document again and notice that `rifle` comes before `atmosphere` before `name` before `shot`; that `kind` and `schema_version` are at the *end*; that inside `shot`, `cant_angle` precedes `max_range`. None of that is meaningful. It is whatever order the runtime's Map implementation produced.

If you need canonical, order-independent JSON — for a checksum, a golden fixture, or a byte-comparison in a test — that is what `analysis_export.lat` does for analysis tables, by assembling the JSON text by hand rather than serializing a Map. Persistence does not need it, because a document is validated by parsing, not by hashing.

9.2.4 Three top-level keys, checked

`kind` must be exactly `"lattice-ballistics/experiment"`, `schema_version` must be exactly the integer `1`, and `experiment` must be the object. Nothing else is permitted at the top level (`persistence.lat:733`).

9.3 What actually round-trips

The strongest statement you can make about a serializer is that it is a fixed point. Save, load, save again, and compare bytes:

Listing 9.4: Save, load, save, compare

```
$ cmp -s eld-m.json eld-m-2.json && echo BYTE-IDENTICAL
BYTE-IDENTICAL
$ shasum -a 256 eld-m.json eld-m-2.json
4e8c9a24cc44e7612403c4df1f3b50f4a8e1ecb17ae2d5f6ad6aeddc2124999d  eld-m.json
4e8c9a24cc44e7612403c4df1f3b50f4a8e1ecb17ae2d5f6ad6aeddc2124999d  eld-m-2.json
```

The second file was produced by loading the first into a fresh session and saving it again. Every display value, every unit string, every explicit null, and even the Map ordering came back identical.

Why the round trip is exact

It is exact because the display value is stored, not recomputed. If the format stored SI and reconstructed the display value by dividing, a 100-yard zero would come back as something like 99.99999999999999 on some inputs, the file would differ on the second save, and every diff of a saved load would be noise. Storing what you typed makes the round trip a copy rather than a calculation.

The canonical value is still checked. `persistence.lat:118-121` reconstructs the quantity from the `{value, unit}` pair it is about to write and asserts that the reconstruction's `si_value` equals the original's — so a quantity whose canonical and display fields disagree can never be written to disk.

What does *not* round-trip is everything outside the experiment:

Session state	Saved?	Where it lives instead
Experiment (all seven parts)	yes	the document
Experiment name	yes	`experiment.name`
Backend	no	`ballistics-repl.lat` startup / `set_backend`
Display preferences	no	`set_display_preferences`
History limit	no	`set_history_limit`
Run history	no	regenerate with `:solve`
Current / previous run	no	regenerate with `:solve`
Trajectory samples	no	the verified-run artifact
Engine version	no	the verified-run artifact

9.4 :save — validate, then write

The write path has an ordering property worth naming, because it is what makes a failed save harmless:

Listing 9.5: `persistence.lat:765-772` — serialization precedes the filesystem

```
export fn save_experiment(path: any, value: any) {
    let checked_path = _path(path, "save_experiment")
    // Complete validation and serialization precede the filesystem mutation.
    let source = experiment_to_json(value)
    if !write_file(checked_path, source) {
        _fail("save_experiment: unable to write '" + checked_path + "'")
    }
}
```

The entire document is built and validated in memory before `write_file` is called even once. There is no path on which a partially written file is left behind because validation failed half-way through.

And `experiment_to_json` does something stranger than it looks. It encodes the experiment to Maps, and then immediately *decodes* the result again before publishing bytes:

Listing 9.6: `persistence.lat:718-729` — encode, then re-decode, then write

```
export fn experiment_to_map(value: any) -> Map {
    let encoded_experiment = _experiment_value_to_map(value, "$.experiment")
    // Struct names alone are not a trust boundary: callers can construct exported
    // struct literals without going through the domain constructors. Rebuild once
    // before publishing bytes so save/load enforce identical relationship checks.
    _experiment_value_from_map(encoded_experiment, "$.experiment")
    flux out = Map::new()
    out.set("kind", EXPERIMENT_KIND)
    out.set("schema_version", EXPERIMENT_SCHEMA_VERSION)
    out.set("experiment", encoded_experiment)
    return out
}
```

The decoded value is thrown away. The point is that the decode path runs all of `domain.lat`'s relational checks — Coriolis requires a latitude, Magnus requires a projectile length, zero distance and muzzle angle are mutually exclusive — so a document can never be written that its own loader would refuse. Save and load enforce identical rules by construction.

A write failure is reported, not swallowed:

Listing 9.7: `:save "/no/such/dir/x.json"`

```
Error
  code: command_execution_error
  message: save: assertion failed: save_experiment: unable to write '/no/such/dir/x.json'
```

Quoting paths

The colon-command grammar treats a quoted argument as one token, backslashes and all, so paths with spaces work and Windows paths survive unmangled: `:save "/tmp/dope cards/6.5 CM 140.json"`. Only the matching quote and the backslash are escapes; every other backslash is preserved byte for byte (`command_parser.lat:197`). The argument never reaches a shell — there is no shell anywhere in the Ballistics Lab.

9.5 :load — reconstruct, then mutate

The load path mirrors the save path. `commands.lat:334-341` does the file read, the full reconstruction, *and* the rendering before it touches the session:

Listing 9.8: `commands.lat:334-341`

```
    if name == "load" {
        let path = args.get("path")
        // Complete I/O, reconstruction, and rendering before the sole mutation.
        let loaded = persistence.load_experiment(path)
        let output = render.render_experiment_summary(loaded, profile)
        sessions.replace_experiment(lab, loaded)
        return _response(name, output, loaded)
    }
```

One mutation, at the end, after everything that could fail has succeeded. The consequence is that a failed load leaves the session exactly as it was. Here is that verified: an experiment at 2710 ft/s, then a load of a deliberately corrupted document, then `:show`:

Listing 9.9: A rejected `:load` does not disturb the session

```
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.shot.max_range.unit: unsupported unit 'furlong'
```

Listing 9.10: and `:show` immediately afterwards, three excerpted lines

```
  active experiment: 140 gr ELD-M at 1,200 yd
  muzzle velocity      : 2710.00 ft/s
  maximum range        : 1200.00 yd
```

9.6 :reset — exactly what it clears

`:reset` answers `Session run state reset.` and that sentence is precisely accurate. It is not "session reset". It is not "new session". It clears the *run* state and nothing else.

Here is the whole implementation:

Listing 9.11: `session.lat:617-630` — the complete `reset_session`

```
// Reset run state while retaining the current experiment, backend configuration,
    display
// preferences, and configured history bound. The external engine remains stateless.
export fn reset_session(session: any) {
    let state = _session_state(session)
    _commit(session, _state(
    state.experiment,
    state.backend,
    state.display,
    state.history_limit,
    [],
    nil,
    nil
    ))
}
```

Four fields are carried forward unchanged. Three are replaced: the history array becomes empty, and `current_run` and `previous_run` become `nil`.

That is what the source says. Here is the same claim measured, in a single session. We set a non-default display profile and a non-default history limit, solve twice, change the muzzle velocity and the zero, and then reset:

Listing 9.12: The probe

```
sessions.set_display_preferences(lab, sessions.display_preferences("m", "m/s", "J",
    "moa", 3))
sessions.set_history_limit(lab, 5)
:solve
:solve
:show
sessions.replace_shot(lab, shot(fps(3100), yd(1200), yd(200)))
:reset
:show
```

Listing 9.13: `:show` before the reset

```
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : m, m/s, J, moa (3 decimals)
  history          : 2/5
  current run      : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
  previous run     : 140 gr ELD-M at 1,200 yd (0.32.0, max_range)
```

Listing 9.14: `:reset`, then `:show`

```
Session run state reset.
Ballistics laboratory session
  active experiment: 140 gr ELD-M at 1,200 yd
  backend          : process-json-v1
  display          : m, m/s, J, moa (3 decimals)
  history          : 0/5
  current run      : none
  previous run     : none
```

And further down the same two `:show` outputs, the experiment itself:

Listing 9.15: From the first `:show`, taken before the shot was replaced

```
  muzzle velocity      : 826.008 m/s
  maximum range        : 1097.280 m
  zero distance        : 91.440 m
```

Listing 9.16: From the second `:show`, after `replace_shot` and after `:reset`

```
  muzzle velocity      : 944.880 m/s
  maximum range        : 1097.280 m
  zero distance        : 182.880 m
```

The order matters: the shot was replaced *between* the two `:show` calls, and `:reset` happened after that. So the second block is the state on the far side of a reset — and it still carries the edit. 944.880 m/s is 3100 ft/s; 182.880 m is 200 yards.

:reset does not reset the experiment

Verified in the session above, field by field:

State	Cleared?	Evidence
Run history	**yes**	`2/5` becomes `0/5`
Current run	**yes**	becomes none
Previous run	**yes**	becomes none
History *limit*	no	stays 5 (the `/5`)
Display preferences	no	stays `m, m/s, J, moa (3 decimals)`
Backend	no	stays `process-json-v1`
Experiment name	no	stays `140 gr ELD-M at 1,200 yd`
Experiment values	**no**	3100 ft/s and the 200-yard zero survive

If you want a clean rifle, `:reset` is not it. Load a saved experiment, or rebuild the experiment with the `sessions.replace_*` functions. `:reset` is for clearing answers, not questions.

Why the engine needs no reset

There is no state on the other side of the process boundary to clear. The process backend spawns one child per solve, hands it a request on standard input, reads one envelope from standard output, and lets the child exit. The comment at `session.lat:3-4` makes the guarantee explicit: a session "never retains a child-process handle: `process_backend`'s closure remains process-per-solve." A reset therefore has nothing to tell the engine.

9.7 The trap: `:load` does not clear the run

Now put the two halves together, and watch a real session tell you something false.

`:load` calls exactly one session function — `sessions.replace_experiment` — and that function replaces the experiment while carrying `current_run`, `previous_run`, and the history forward untouched. That is correct behaviour for `replace_experiment`; it is how every other `replace_*` behaves, and it is what lets you compare a run before an edit against a run after it.

But `:at`, `:dope`, `:wind-card`, and `:compare` all read the *current run*, not the experiment. So after a load, they answer questions about the trajectory that was solved *before* the load.

Here is the sequence, run exactly as written:

Listing 9.17: The trap, in seven lines

```
:solve
:save "/tmp/trap.json"
sessions.replace_shot(lab, shot(fps(3300), yd(1200), yd(100)))
:solve
:load "/tmp/trap.json"
:at 1000 yd
```

The load succeeds and reports the experiment it read — the 2710 ft/s one:

Listing 9.18: `:load` reply, excerpted

```
Experiment: 140 gr ELD-M at 1,200 yd
  projectile           : 140 gr ELD-M
  muzzle velocity      : 2710.00 ft/s
```

And then:

Listing 9.19: `:at 1000 yd` immediately after the load

```
Observation
  distance: 1000.00 yd
  time    : 1.16 s
  velocity: 2038.59 ft/s
  energy  : 1291.68 ft-lb
  drop    : +188.91 in
  windage : +36.88 in
  Mach    : 1.83
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

After a :load, always :reset and :solve

The session says the rifle is doing 2710 ft/s. :at 1000 yd says the bullet arrives at 2038.59 ft/s with 188.91 inches of drop. Those numbers belong to the 3300 ft/s trajectory that was still sitting in current_run. The true 2710 ft/s answer at 1000 yards is 1586.32 ft/s and 300.82 inches — a difference of 111.91 inches of drop, about 3.1 mil at that distance.
Nothing failed. No error was printed. :show and :at simply described two different rifles.
The safe idiom is three commands, in this order:

```
:load "/tmp/trap.json"
:reset
:solve
```

:reset forces the issue — with no current run, the query commands refuse rather than answer:

```
Session run state reset.
Error
  code: command_state_error
  message: at: no current run is available
  path: $.session.current_run
```

and after the :solve, the same query gives the right answer:

```
Observation
  distance: 1000.00 yd
  time    : 1.45 s
  velocity: 1586.32 ft/s
  energy  : 782.12 ft-lb
  drop    : +300.82 in
  windage : +50.43 in
  Mach    : 1.43
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

How to notice without remembering

Every table the BALLISTICS LAB prints carries a provenance block naming the series it came from, and `:show` prints the current and previous run with their experiment names. If you have renamed your experiments as the comparison chapter recommends, a stale run announces itself: `:show` will name the loaded experiment while the DOPE table's provenance names the old one. If every experiment in your session is called the same thing, nothing will warn you.

9.8 What a bad document gets you

The loader is a trust boundary, and it behaves like one. Every rejection below came from feeding `:load` a copy of the good document with one field changed.

Listing 9.20: Eight tampered documents, in order: an added field, a bad unit, a removed key, a wrong `kind`, a wrong `schema_version`, Coriolis without a latitude, a string BC, and a file that does not exist

```
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.projectile: unknown field 'bc_g1'
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.shot.max_range.unit: unsupported unit 'furlong'
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.rifle: missing required field 'twist_rate'
Error
  code: command_execution_error
  message: load: assertion failed: $.kind: unsupported document kind 'lattice-ballistics/verified-run'
Error
  code: command_execution_error
  message: load: assertion failed: $.schema_version: unsupported value 2
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.atmosphere.latitude: required when Coriolis is enabled
Error
  code: command_execution_error
  message: load: assertion failed: $.experiment.projectile.ballistic_coefficient: expected Number, got String
Error
  code: command_execution_error
  message: load: assertion failed: experiment JSON: expected String, got Nil
```

Seven of the eight are exactly what you want: a JSON path naming the offending field and a sentence saying what was wrong with it. Three of them are worth pointing out specifically.

Unknown fields are fatal, not ignored. Adding `"bc_g1": 0.62` to the projectile — a plausible thing for a well-meaning script to do — is rejected. A document is a closed schema. This is what makes it safe to trust the fields that *are* present.

Cross-component checks run on the plain Maps. The Coriolis rejection is raised at line 671 of `persistence.lat`, before any frozen domain value has been constructed. That is what lets the error name a JSON path directly instead of reporting a failure from somewhere inside a constructor:

Listing 9.21: `persistence.lat:665-675`

```
    // Check cross-component requirements while the document is still plain
    // Maps. Besides producing a direct path, this avoids constructing a partial
    // graph of frozen domain values only to reject the final Experiment.
    let encoded_projectile = _map(object.get("projectile"), path + ".projectile")
    let encoded_atmosphere = _map(object.get("atmosphere"), path + ".atmosphere")
    let encoded_solver = _map(object.get("solver"), path + ".solver")
    if encoded_solver.has("coriolis") && encoded_solver.get("coriolis") == true {
        if encoded_atmosphere.has("latitude") && encoded_atmosphere.get("latitude") ==
    nil {
            _fail(path + ".atmosphere.latitude: required when Coriolis is enabled")
        }
    }
```

The eighth message is a rough edge. A file that does not exist produces `experiment JSON: expected String, got Nil`. That is an internal type assertion leaking out where "no such file" belongs: `read_file` returned `nil` and the string check downstream reported what it saw. The behaviour is safe — the load is refused and the session is untouched — but the message will not tell a newcomer that they mistyped a path.

Why every load error is `command_execution_error`

`:load` is the one command whose failures all arrive with the same code. The dispatcher runs commands inside a two-stage `try` (`commands.lat:360-383`): a validation stage that produces `command_validation_error`, then an execution stage that produces `command_execution_error`. Argument problems — a missing path, too many arguments — are caught by the parser and get precise codes. Everything that goes wrong *inside* the persistence module is a thrown assertion, and the dispatcher normalizes thrown assertions into `command_execution_error` with the assertion's message appended. The message is precise; only the code is generic.

9.9 A working routine

Session hygiene that survives contact with a real bench

1. Build the experiment once, give it a real name with `replace_experiment`, and `:save` it immediately. The name is the only label your cards will carry.
2. Keep one file per load, named for the load, in a directory you back up. The documents are about 1.3 KB and diff cleanly, so version control works.
3. When you come back: `:load`, then `:reset`, then `:solve`. Every time. The `:reset` costs nothing and closes the trap in Section 9.7.
4. Set your display preferences after loading, not before saving — they are not in the file, and they never will be.
5. When a run matters — a load you have trued, a card you will print — promote it to a verified-run artifact. That is the format that stores the answer, the resolved request, and the engine version together, and it is the subject of the artifacts chapter.
6. Edit the JSON directly when it is convenient. Changing a velocity and reloading is a legitimate workflow, and the loader will catch you if you break something.

9.10 Command summary

Command	Reads	Writes	Leaves alone
`:save "<path>"`	experiment	the file	everything in the session
`:load "<path>"`	the file	experiment	runs, history, display, backend
`:reset`	—	runs, history	experiment, display, backend, limit

All three are transactional. `:save` serializes completely before it opens a file; `:load` reconstructs and renders completely before its single mutation; `:reset` builds an entire next state and publishes it with one write to the session's `Ref`. There is no partial state to recover from, on any of the three.

Part III

The Ballistics

Chapter 10

What the Engine Solves

Everything you have typed at the Ballistics Lab prompt so far has been bookkeeping: naming a bullet, choosing units, deciding which numbers count as an experiment. This part of the book is about the other side of the pipe. When the Lab spawns `ballistics solve-json` and hands it a request, some very specific mathematics happens, and the quality of every DOPE card you print afterwards depends on what that mathematics does and does not include.

This chapter answers four questions, in order:

- What differential equation is being integrated?
- How is it integrated, and how much does the method matter?
- Where does the drag number come from — drag models, drag tables, ballistic coefficients, BC segments?
- What, exactly, does a completed solve hand back?

Physics first. The commands come after the ideas they implement.

Which engine produced these numbers

Every number in this part came from an engine run performed while writing it. The binary on `PATH` on the writing machine reports version `0.32.0`; the binary in the engine checkout reports `0.30.1`. Because the book is nominally pinned to `0.30.1`, every solve-json case shown in this chapter was run through *both* binaries and the `summary` and `samples` objects compared. Across eight cases — baseline, Euler, RK4, a G1 load, a constant crosswind, a segmented wind, a hot thin atmosphere, and a 2000 m station altitude — the two versions produced byte-identical output. Section 10.9 shows the comparison. Where a number comes from the CLI rather than `solve-json`, it is labelled as such, because the two surfaces have different defaults.

10.1 A trajectory is an initial-value problem

A bullet in flight is a point with a position and a velocity. Give it a starting position and a starting velocity, tell it what accelerations act on it at every instant, and its whole future is determined. That is an *initial-value problem*, and solving one numerically is what the engine spends its time doing.

Point-mass trajectory model

The engine integrates a **six-state point-mass** model: three position components and three velocity components. The bullet has mass and a drag coefficient but — for the purposes of the flight path — no extent, no orientation, and no angular momentum. Spin, yaw, precession and nutation are modelled, but as diagnostics computed *alongside* the flight path or as closed-form offsets applied to it, not as torques inside the integration.

The state is a six-vector (x, y, z, v_x, v_y, v_z). The frame is McCoy's: **x downrange, y up, z right**, stated in the engine source at `src/wind.rs:9-20`.

The JSON output relabels the axes

The internal frame is x-downrange / y-up / z-right. The engine's *native* `-o json` output uses the opposite convention and says so in its own legend: there, `x` is lateral offset (positive right), `y` is height above ground, and `z` is downrange. Read the source and then the JSON and you will transpose x and z. The `solve-json` v1 samples avoid the problem entirely by naming their fields `distance_m`, `drop_m` and `windage_m` instead of using axis letters.

The derivative of position is velocity; that half is free. The interesting half is the acceleration, assembled in `src/derivatives.rs` in `compute_derivatives` (line 146). Three terms matter for an ordinary shot:

Gravity. Constant, downward, rotated into the shot frame by the inclination of the shot:

$$\mathbf{a}_g = \big(-g\sin\theta,\ -g\cos\theta,\ 0\big), \qquad g = 9.80665\ \mathrm{m/s^2}$$

with θ the uphill/downhill line-of-sight angle (`src/derivatives.rs:165–170`; the constant is `G_ACCEL_MPS2` in `src/constants.rs:4`). For a level shot $\theta = 0$ and gravity is straight down in the shot frame, as you would expect.

Drag. This is the whole game. Drag does not act along the bullet's velocity; it acts along the bullet's velocity *relative to the air*:

Listing 10.1: The wind-relative velocity, `src/derivatives.rs:174–176`

```
let wind_vector = level_vector_to_shot_frame(wind_vector, theta);
let velocity_adjusted = vel - wind_vector;
let speed_air = velocity_adjusted.norm();
```

Subtracting the wind vector from the bullet's velocity is the single line that makes Chapter 12 necessary. Everything a crosswind does to a bullet happens because of that subtraction.

The magnitude of the drag acceleration is then

$$a_{\text{drag}} \;=\; \frac{C_d(M)\ v_{\text{rel}}^2\ \big(\rho/\rho_0\big)\ k}{\mathrm{BC}}$$

which is literally how the code writes it (`src/derivatives.rs`, in the block ending `accel_drag = -a_drag_m_s2 * (velocity_adjusted / speed_air)`), with v_{rel} in feet per second, $\rho_0 = 1.225\ \mathrm{kg/m^3}$, and

Listing 10.2: The retardation constant, `src/constants.rs:54–61`

```
/// Exact imperial retardation form for density normalized to ICAO sea-level air
/// (1.225 kg/m^3 / ICAO_DENSITY_LB_FT3):
///
/// `a_ft/s^2 = Cd * v_fps^2 * (rho / 1.225) * CD_TO_RETARD / BC`
///
/// The older `0.000683 * 0.30` value is the Army Standard Metro constant and is
/// only consistent with an ASM_DENSITY_LB_FT3 density reference.
pub const CD_TO_RETARD: f64 = 2.08551e-4;
```

Read that formula slowly, because four whole chapters of this book are hiding in it:

- $C_d(M)$ — the drag coefficient, a function of Mach number only. That is the *drag model*, and it is Section 10.4.
- v_{rel}^2 — drag grows with the *square* of air speed. A bullet leaving at 2700 ft/s sheds far more velocity in its first hundred yards than in its last.
- ρ/ρ_0 — air density, normalised to sea-level standard. That is the whole of Chapter 11, and it enters *linearly*.
- BC — the ballistic coefficient, sitting in the denominator. It is not a fudge factor; it is the divisor of the entire retardation. Section 10.5 takes it apart.

Everything else. Magnus force, Coriolis acceleration, and the optional advanced effects add further terms, all of them small at the ranges most shooters work at. The chapter on stability and small effects takes them up. For a level, sub-1000-yard shot with the default `effects` block, gravity and drag are the model.

10.2 What the solver is actually given

The initial conditions are less obvious than they look. The bullet does not start at the origin travelling along the line of sight; it starts at the muzzle, travelling along the *bore*, and the bore is tipped up relative to the sight line by whatever angle makes the trajectory cross the sight line again at the zero distance.

Zero distance and muzzle angle

The `shot` object accepts *either* `zero_distance_m` — "find the launch angle that puts the bullet on the sight line at this range" — or `muzzle_angle_rad` — "use this launch angle, I already know it." The Lab's `shot()` constructor rejects both together (`domain.lat:391–393`); the engine accepts both but refuses to run the elevation search and emits a `zero_distance_elevation_not_resolved` warning. Supplying a zero distance costs an extra trajectory solve: the engine has to search for the angle before it can fly the shot you asked for.

Three more geometric quantities set the starting point and the reference plane: `sight_height_m` (how far the optic sits above the bore), `muzzle_height_m` (how far the bore sits above the ground datum), and `ground_threshold_m` (how far below that datum the run is allowed to go before the engine calls it terminated). A fourth, `target_height_m`, does more work than its name suggests.

`target_height_m` is the height the zero is solved to

`shot.target_height_m` is the height *above the local ground datum* of the point the elevation search aims at. The engine documents it as "meters above ground for zeroing" and defaults it to `0.0` — "target at ground level by default" (`src/cli_api.rs:181` and `:502`). It therefore picks the bore angle, and through the bore angle it sets every drop the run reports.
Leave it out and the engine solves a zero that puts the bullet on the *ground* at the zero distance. Set it to the height of the line of sight — muzzle height plus sight height — and the bullet arrives at the aim point instead, which is what "zeroed at 100 yards" means to a shooter. The Ballistics Lab sends the second of those: `protocol_v1.lat`'s `shot_to_v1_map` fills the field in from the rifle whenever the experiment does not name one explicitly. Chapter 13 takes the consequences apart in detail.

Raise the muzzle, raise the target

Because `target_height_m` is an *absolute* world height above the local ground datum — not a height relative to the muzzle — a raised muzzle needs a raised target with it. Hand the engine `muzzle_height_m: 1.5` while leaving `target_height_m` unset and you have asked it to zero a rifle held 1.5 m up onto a target lying in the grass. It fails outright, with `solve_failed` and the message "Zero angle did not converge: residual height error too large (target not reachable / not bracketed)".
Driving the engine from the Ballistics Lab does not reach that failure, because the target height is derived from the rifle. The same geometry entered as `rifle("1.5 m above the ground", inches(1.5), m(1.5), inches(8), "right")` comes back with `target_height_m: 1.5381` in the resolved request — 1.5 m of muzzle height plus 0.0381 m of sight height — and solves.

10.3 Integration: RK45, RK4, Euler

The differential equation has no closed-form solution — C_d is a lookup table, and ρ varies with height — so the engine marches the state forward in small steps. It offers three ways to do it, exposed on `solve-json` as `solver.method` and on the CLI as flags.

Method	CLI flag	Behaviour
RK45 (Dormand–Prince 5(4))	*default*	Adaptive step. Ignores `--time-step`. The reference.
RK4 fixed step	`--use-rk4-fixed`	Fourth order, constant step. Honours `--time-step` (default 0.001 s).
Euler	`--use-euler`	First order, constant step. Honours `--time-step`.

RK45 is the same Dormand–Prince tableau `scipy.integrate.solve_ivp` uses (`src/trajectory_integration.rs:92–230`), with the control constants set in the same file:

Listing 10.3: RK45 control constants, `src/trajectory_integration.rs:17–21`

```
const RK45_MIN_STEP: f64 = 1e-6;
const RK45_DEFAULT_TOLERANCE: f64 = 1e-6;
const RK45_SAFETY_FACTOR: f64 = 0.9;
const RK45_MIN_SCALE: f64 = 0.1;
const RK45_MAX_SCALE: f64 = 2.0;
```

The method takes a step, estimates its own error by comparing a fifth-order and a fourth-order result, and shrinks or grows the next step accordingly — never by more than a factor of two up or ten down. Near the muzzle, where velocity is changing fastest, the steps are tiny; out at the far end they stretch.

10.3.1 How much does the choice matter?

Enough theory. Here is the same 175 gr G7 load, 2700 ft/s, 100 yd zero, 1000 yd of range, solved six ways through `solve-json`. All numbers are raw SI from the response envelope; `drop` is the terminal sample's `drop_m` at 914.4 m.

Method	Time of flight (s)	Terminal speed (m/s)	Drop (m)
RK45 (default)	1.7064011690199543	349.28132093557986	9.979643353612667
RK4, $dt = 10^{-3}$	1.706396383133215	349.27988129219466	9.98091865758223
RK4, $dt = 10^{-4}$	1.7063963360560142	349.27986695289786	9.980917486146152
Euler, $dt = 10^{-3}$	1.7074575573620636	349.0426284091595	9.995548717477943
Euler, $dt = 10^{-4}$	1.7065023909462858	349.25613940734394	9.98293697930828
Euler, $dt = 10^{-5}$	1.706406940667394	349.27749420052146	9.981119416372783

Three things to take from that table.

Euler at its default step is wrong by about six tenths of an inch at 1000 yards. $9.995549 - 9.979643 = 0.015905$ m, which is 0.626 in, or 0.0174 mil. That is smaller than any group you will ever shoot. Euler is not *dangerous* here; it is merely unnecessary.

Euler converges, slowly. Measured against the converged fixed-step answer of 9.980917 m, Euler is 14.6 mm out at $dt = 10^{-3}$, 2.0 mm out at 10^{-4} and 0.20 mm out at 10^{-5} — the last step of ten buying exactly a factor of ten, which is what first order means. They do converge; they just make you pay for it.

RK4 is already converged at its default step. Going from $dt = 10^{-3}$ to $dt = 10^{-4}$ moves the 1000-yard drop by 1.2 micrometres. And the converged fixed-step answer sits 1.27 mm from RK45's, which is the adaptive method's own 10^{-6} tolerance showing up in the third decimal of a millimetre.

10.3.2 The same experiment from the Lab

The Lab exposes the integrator through `solver_config`, whose first parameter is the method name and whose second is the fixed time step (`domain.lat:409-445`). Only `"rk45"`, `"rk4"` and `"euler"` are accepted; anything else is rejected before a process is spawned.

Listing 10.4: Switching integrators from the Lab prompt

```
sessions.replace_solver(lab, solver_config("euler", 0.001, yd(100), nil, false, nil))
sessions.replace_solver(lab, solver_config("rk4",   0.001, yd(100), nil, false, nil))
```

Solving between the two, and asking `:at 1000 yd` each time, gives the imperial view of the same table:

Listing 10.5: RK45, then Euler, then RK4 — same load, same range

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.94 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.15 ft/s
  energy  : 509.48 ft-lb
  drop    : +393.53 in
  windage : +0.00 in
  Mach    : 1.02
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.93 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.95 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
```

392.90, 393.53, 392.95 inches. The Lab's two-decimal display is exactly the right resolution to see the Euler error and nothing finer.

Reading the `drop` sign

The legend under every observation says what it means: `Signs: drop + below/- above`. In `solve-json` v1 `drop_m` is a *depth*, not a height — positive means the bullet is below the sight line. The Lab passes `drop_m` straight through (`process_backend.lat:403`), so the number is the engine's. Read +392.90 in as "392.90 inches *below* the sight line", which is what a 1000-yard .308 does. Negative values are real and appear between the near and far zero crossings, where the bullet is above the line of sight.

10.3.3 What it costs

Twenty consecutive `solve-json` runs of the baseline request, including process spawn, took 0.0765 s of wall time — about 3.83 ms each. Twenty runs of the same request with Euler at $dt = 10^{-5}$ took 2.2377 s, about 112 ms each: twenty-nine times the cost for a slightly worse answer.

Use RK45 unless you are benchmarking

The adaptive method is both the most accurate option and, by a wide margin, the fastest. The fixed-step methods exist so that results can be compared against other software that uses them, and so that a regression suite can pin a deterministic step count. They are not an accuracy/speed trade you need to make.

RK45 keeps its own step

Send `solver.time_step_s` together with `method: "rk45"` and the engine accepts it, echoes it in `resolved_request`, ignores it during integration, and tells you so with a `rk45_time_step_ignored` warning: "Adaptive RK45 owns its time step; the explicitly supplied fixed time step is retained in resolved_request but ignored by integration." Nothing is silently dropped.

10.4 Drag models and drag tables

$C_d(M)$ is not a formula. It is a table of measured drag coefficients against Mach number for a *standard projectile shape*, and every "drag model" is one such table.

Drag model

A **drag model** is a reference projectile whose drag has been measured across the Mach range, tabulated, and published. G1 is a flat-based, blunt-nosed 1881 artillery shell. G7 is a boat-tailed, long-ogive form much closer to a modern match bullet. A ballistic coefficient is meaningful only *relative to* the model it was measured against: "BC 0.243" means nothing until you say "G7".

The engine ships nine families, each with its own real Mach-indexed table and no silent fallbacks between them: G1, G2, G5, G6, G7, G8, GI, GS and RA4. You can print any of them:

Listing 10.6: Dumping a reference drag curve

```
$ ballistics drag-curve --drag-model g7 -o csv | head -3
mach,cd
0,0.1198
0.05,0.1197
$ ballistics drag-curve --drag-model g7 -o csv | wc -l
      85
$ ballistics drag-curve --drag-model g1 -o csv | wc -l
      80
```

Eighty-five lines is a header plus 84 rows for G7; 80 is a header plus 79 rows for G1. Both run from Mach 0 to Mach 5.

10.4.1 The transonic wall

Plot either table and the shape is the same: a low, slowly-rising subsonic plateau, a violent rise through Mach 1, a rounded peak, then a slow decline. Here is G7 through the band that matters, straight from `drag-curve --drag-model g7 -o csv`:

Mach	C_d	**Mach**	C_d	**Mach**	C_d
0.75	0.1215	0.925	0.1660	1.075	0.4034
0.775	0.1226	0.95	0.2054	1.1	0.4014
0.8	0.1242	0.975	0.2993	1.15	0.3955
0.825	0.1266	1.0	0.3803	1.2	0.3884
0.85	0.1306	1.025	0.4015	1.25	0.3810
0.875	0.1368	1.05	0.4043	1.3	0.3732
0.9	0.1464				

From Mach 0.8 to the G7 peak at Mach 1.05, C_d goes from 0.1242 to 0.4043: it **more than triples**. That is the whole reason transonic flight is treated as a distinct regime and why the engine flags samples inside it. G1 peaks later and higher, at $C_d = 0.6625$ near Mach 1.4.

Two corrections are worth knowing about. A transonic drag-rise correction is applied automatically, exactly once, after the base table lookup (`src/derivatives.rs`, with a long comment explaining that it used to be applied twice and made near-Mach-1 drag roughly three times too high). A Reynolds number correction exists in `src/reynolds.rs` but is *not* applied by default on any solver path — deliberately, so the three solver families in the engine cannot disagree subsonically.

10.4.2 Which models the Lab can ask for

Four: `G1`, `G6`, `G7`, `G8`. Ask for anything else and the Lab refuses before a process is ever spawned:

Listing 10.7: The Lab refuses G5 without asking the engine

```
Error
  code: evaluation_error
  message: assertion failed: projectile.drag_model: unsupported value 'G5'; expected one of [G1,
     G6, G7, G8]
```

It is worth being exact about *whose* restriction that is, because it used to be the engine's and is now the Lab's alone.

Through engine 0.32.0 the v1 protocol really did accept only those four and reject the other five outright, with an `invalid_value` naming the permitted list. That changed in 0.33.2. The reason had nothing to do with the Lab: `monte-carlo --wez` attributes hit-probability variance to individual inputs by re-solving through the v1 request shape, so any drag family v1 could not express was a family whose variance the sweep had to report as `n/a`. Widening v1 to the full built-in set was the fix. All nine families now solve, and each produces its own trajectory rather than quietly aliasing to G1 — here is the book's reference request from Chapter C, with only the drag model substituted:

Listing 10.8: Nine families, nine answers, engine 0.37.0

```
$ for m in G1 G2 G5 G6 G7 G8 GI GS RA4; do
>     printf '%-4s ' $m
>     sed "s/\"G7\"/\"$m\"/" req.json | ballistics solve-json \
>       | jq -c '[.status, .summary.terminal_speed_mps]'
> done
G1   ["ok",233.15265486200218]
G2   ["ok",319.5801094982818]
G5   ["ok",296.4592771254353]
G6   ["ok",296.1004654279713]
G7   ["ok",323.89758274449713]
G8   ["ok",304.05956055730854]
GI   ["ok",234.4301633277849]
GS   ["ok",120.35181796842697]
RA4  ["ok",251.56933733700956]
```

Spread out like that the families stop being interchangeable labels: the same bullet, the same declared BC of 0.243 and the same thousand metres arrive anywhere between 120 and 324 m/s depending on which reference shape you claim to be scaling from. That is the argument for the Lab keeping its own list rather than inheriting the engine's — a BC is meaningful only against the family it was measured on, and four families the Lab can reason about beats nine it cannot.

The Lab enforces the four in three separate places, and it is not belt-and-braces duplication — each guards a different boundary. The `projectile()` constructor checks it at construction (`domain.lat:245`), so a bad model never becomes a session object. The process backend checks it again on the way out (`process_backend.lat:153`), and the resolved-request validator checks the model the engine *echoed back* (`resolved_request_v1.lat:112`), so an engine that widened its accepted set — exactly what happened here — cannot silently widen the Lab's by answering a request the Lab would not have made.

The CLI rejects typos, and always did

The `trajectory` subcommand has taken all nine families since long before v1 did, and it refuses a name it does not recognise at the argument parser rather than falling back to anything. Verified on 0.32.0 and 0.37.0 alike, byte for byte:

```
$ ballistics trajectory ... --drag-model g77
error: invalid value 'g77' for '--drag-model <DRAG_MODEL>'
  [possible values: g1, g2, g5, g6, g7, g8, gi, gs, ra4]
```

The two surfaces now agree on coverage as well as on strictness. A load that needs G2, G5, GI, GS or RA4 can be run from the engine either way — from the CLI or through `solve-json` — but not from the BALLISTICS LAB, and the error you get when you try names a Lab field path rather than a JSON one.

10.5 The ballistic coefficient

Look again at the drag law: BC is the divisor of the whole retardation. Double the BC and you halve the deceleration. It is not a tuning knob bolted onto the side of the model; it *is* the model's scaling between "how the standard shape behaves" and "how your bullet behaves".

Ballistic coefficient

The **ballistic coefficient** is the ratio of your bullet's sectional density to that of the reference projectile, or equivalently the factor by which your bullet resists deceleration better than the reference shape. It is dimensionless and always attached to a specific drag model. A G7 BC and a G1 BC for the same bullet are different numbers describing the same physics.

10.5.1 Feeding a G7 BC to a G1 curve

The most common way to be catastrophically wrong in exterior ballistics is to type a BC from one table under a different table's name. Here is the 175 gr load with its true G7 BC of 0.243, then the

same 0.243 declared as a G1 BC, then the G1 BC that actually matches — all from the Lab, all at 1000 yards.

Listing 10.9: G7 0.243, then G1 0.243, then G1 0.4775

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.94 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
Observation
  distance: 1000.00 yd
  time    : 2.42 s
  velocity: 805.15 ft/s
  energy  : 251.86 ft-lb
  drop    : +780.81 in
  windage : +0.00 in
  Mach    : 0.72
  flags   : subsonic, terminal
Observation
  distance: 1000.00 yd
  time    : 1.70 s
  velocity: 1204.21 ft/s
  energy  : 563.39 ft-lb
  drop    : +392.86 in
  windage : +0.00 in
  Mach    : 1.08
  flags   : transonic, terminal
```

The middle block is the mistake: 780.81 inches of drop instead of 392.90, a bullet arriving subsonic at 805 ft/s instead of transonic at 1146, and a time of flight nearly a second longer. Thirty-two feet of extra elevation on a 1000-yard target. The mistake was a single missing conversion.

10.5.2 Fitting a G1 BC to a G7 answer

The third block is more subtle. A G1 BC of 0.4775 reproduces the G7 load's 1000-yard drop to within four hundredths of an inch — and is wrong everywhere else. Here are the two DOPE tables the Lab printed for those two loads, 200 to 1000 yards:

Listing 10.10: G7 BC 0.243 — `:dope 200 1000 200 yd`

```
DOPE: 175 gr SMK at 1,000 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       200.00 |     +4.01 |         +0.56 |         2334.07 |        2116.56 |     0.24 | 2.09
       400.00 |    +32.26 |         +2.24 |         1998.50 |        1551.71 |     0.52 | 1.79
       600.00 |    +95.69 |         +4.43 |         1689.82 |        1109.39 |     0.84 | 1.51
       800.00 |   +208.45 |         +7.24 |         1403.91 |         765.74 |     1.23 | 1.26
      1000.00 |   +392.90 |        +10.91 |         1145.94 |         510.18 |     1.71 | 1.03
Signs: drop + below/- above; come-up + correction up/- correction down
```

Listing 10.11: G1 BC 0.4775 — the same command, the same rifle

```
DOPE: 175 gr SMK at 1,000 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       200.00 |     +4.01 |         +0.56 |         2330.14 |        2109.44 |     0.24 | 2.08
       400.00 |    +32.36 |         +2.25 |         1990.39 |        1539.14 |     0.52 | 1.78
       600.00 |    +96.17 |         +4.45 |         1683.04 |        1100.51 |     0.85 | 1.51
       800.00 |   +209.58 |         +7.28 |         1416.34 |         779.36 |     1.24 | 1.27
      1000.00 |   +392.86 |        +10.91 |         1204.21 |         563.39 |     1.70 | 1.08
Signs: drop + below/- above; come-up + correction up/- correction down
```

The elevation columns never come more than four hundredths of a mil apart, and at 200 and 1000 yards they are identical. The *velocity* columns are a different story: at 1000 yards the G1-fitted load is arriving at 1204.21 ft/s against the G7 load's 1145.94 — 58 ft/s and 53 ft-lb optimistic — and it is still comfortably supersonic at Mach 1.08 where the real bullet is at 1.03 and about to enter trouble. Fit a G1 BC to drop and you have bought a correct elevation curve and a wrong terminal ballistics story.

Use the model your BC was measured against

If the bullet maker publishes a G7 BC, declare `"G7"`. The reason G7 is preferred for long boat-tail bullets is not fashion: the reference shape is close enough to the real bullet that a *single* BC number stays honest across the whole velocity range, which is exactly what the two tables above show G1 failing to do.

The come-up column is the total dial, and it checks out

Both tables read the way a turret works: positive, growing with distance, and +10.91 mil at 1000 yards means dial 10.91 mil *up* from the 100-yard zero. That is a *total* from the zero, not an increment from the row above.

The engine will tell you the same thing by a completely different route. Its `come-ups` subcommand takes flags rather than v1 JSON, and its `drop_mil` column is the total dial:

```
$ ballistics come-ups -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --zero-distance 100 --sight-height 1.5 --start 200 --end 1000 \
      --step 200 --adjustment-unit mil -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
200,0.557,0.557,2334,2117,0.239
400,2.240,1.684,1999,1552,0.517
600,4.430,2.190,1690,1109,0.843
800,7.238,2.808,1404,766,1.233
1000,10.913,3.676,1146,510,1.706
```

0.557, 2.240, 4.430, 7.238, 10.913 against the Lab's +0.56, +2.24, +4.43, +7.24, +10.91 — agreement at every row, to the Lab's two printed decimals. Note that the subcommand's own `come_up_mil` column is the *increment* from the previous row and has no counterpart in the Lab's table; Section 13.5 takes that pair of column names apart.

10.5.3 Which standard atmosphere your BC is referenced to

A BC is a ratio of drags, and drag depends on air density, so a BC implicitly carries a reference density. This engine's retardation constant is calibrated to the ICAO figure. Several manufacturers — Sierra, Hornady and Barnes among them — publish BCs referenced to the older, denser Army Standard Metro atmosphere instead. Feeding an ASM-referenced BC in raw under-predicts drag by about 1.8%.

The CLI has a `--bc-reference` flag taking `icao` (the default) or `army-standard-metro`, converting once at solver construction. The effect is real. The reference load of this chapter, at 1000 yards:

```
--bc-reference icao                {"v":1146.1674756649934,"tof":1.7062218910001727}
--bc-reference army-standard-metro {"v":1125.4031242178455,"tof":1.7227251081958828}
```

Twenty-one feet per second of impact velocity, from a flag that changes no physics — only which reference density your BC number was measured against.

`solve-json` v1 has no BC-reference field

The protocol the Lab speaks has `ballistic_coefficient` and nothing else: v1 assumes ICAO. If your BC came off an ASM-referenced data sheet, convert it before you type it into `projectile()`, because neither the Lab nor the wire will do it for you. This is one of several places where the CLI is a superset of the protocol.

10.6 BC segments: one bullet, several coefficients

A single BC pretends your bullet's drag curve is a scaled copy of the reference curve at every Mach number. It never quite is. The standard patch is to publish several BCs, each valid over a velocity band — what everyone calls *banded* or *segmented* BCs.

The engine will generate a plausible set for you. Ask it for a 175 gr, 0.308-inch match bullet with a base G7 BC of 0.243:

Listing 10.12: `generate-bc-segments` for a 175 gr 0.308 match bullet

```
$ ballistics generate-bc-segments -b 0.243 -m 175 -d 0.308 --drag-model g7 -o json
{
  "base_bc": 0.243,
  "caliber_inches": 0.308,
  "drag_model": "g7",
  "mass_grains": 175.0,
  "model": "",
  "segments": [
    {
      "bc": 0.243,
      "velocity_max_fps": 5000.0,
      "velocity_min_fps": 2800.0
    },
    {
      "bc": 0.24115584384,
      "velocity_max_fps": 2800.0,
      "velocity_min_fps": 2200.0
    },
    {
      "bc": 0.23746753152,
      "velocity_max_fps": 2200.0,
      "velocity_min_fps": 1600.0
    },
    {
      "bc": 0.2337792192,
      "velocity_max_fps": 1600.0,
      "velocity_min_fps": 0.0
    }
  ]
}
```

Four bands, with breaks at 2800, 2200 and 1600 fps. The top band is the base BC unchanged; the bottom is 0.2337792192, about 3.8% lower. Note also that the `"model"` field comes back empty — the generator had no named bullet to key off, only the numbers we gave it.

`-model` changes the band layout, not just the labels

If you pass a bullet-family name, you get a different — and finer — set of bands. The same call with `--model "SMK"` appended returns *five* bands, broken at 2800, 2400, 2000 and 1600 fps, running 0.243, 0.24023376576, 0.23654545344, 0.23285714112 and 0.2291688288. That is not a relabelling of the four bands above; it is a different partition of the velocity axis with different coefficients on it. Section 14.11 shows the named-model form for the 6.5 mm bullet we true in Chapter 14. When you compare banded sets — ours against a friend's, or this year's against last year's — check that the breakpoints match before you conclude anything from the BC values.

Two things about the four bands are worth pausing on before we use them. The first is that the top band never applies to this load at all: the reference shot leaves the muzzle at 2700 fps, below the 2800 floor, so the bullet starts its life in the *second* band and only ever moves down. The second is that the bands are not visited evenly. Walking the segmented solve out in 100-yard steps, this bullet crosses 2200 fps between 270 and 280 yards, and 1600 fps at almost exactly 650 — so it spends roughly the last 350 yards, and near enough half its flight time, down in the bottom band. That is where the segmentation does its work.

Feed the bands back in and the answer moves:

Listing 10.13: Flat BC versus four velocity bands, 1000 yd

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact -o json | jq '.time_of_flight, .impact_velocity'
1.7062218910001727
1146.1674756649934

$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact \
    --bc-segment 2800:5000:0.243 --bc-segment 2200:2800:0.24115584384 \
    --bc-segment 1600:2200:0.23746753152 \
    --bc-segment 0:1600:0.2337792192 -o json | jq '.time_of_flight, .impact_velocity'
1.72399054917383
1117.9804861405337
```

Twenty-eight feet per second of impact velocity, and 0.018 s of extra flight time. For comparison, running the whole shot flat at the *lowest* band's BC of 0.2337792192 gives 1.7428309383361942 s and 1102.7357702173306 ft/s — so the segmented answer sits, as it must, between the two flat cases: slower and lower-energy than the optimistic single 0.243, but not as pessimistic as pretending the bullet flew the whole way at its subsonic coefficient.

Segments do not cross the wire

`--bc-segment` is a CLI-only feature. So are custom drag tables (`--drag-table`), 5D BC tables, powder-temperature curves, atmosphere zones, wind shear, pitch damping and precession. `docs/SOLVE_JSON_V1.md` lists them explicitly as *not* in v1: the protocol is a deliberately narrow, deterministic, transport-free core. The Lab therefore cannot ask for banded BCs today. If you need them, you need the CLI — and you lose the Lab's provenance, verification and artifact machinery in exchange.

10.7 Custom drag tables

When a bullet maker publishes radar-measured drag — Hornady's CDM files, Lapua's `.drg` decks — you can skip the reference-model abstraction entirely. `--drag-table <FILE>` accepts a CSV of `mach,cd` pairs or a vendor `.drg`, and it replaces both the G-model *and* the BC: the retardation denominator becomes the bullet's sectional density instead, because the tabulated C_d is already the projectile's actual drag coefficient rather than a reference shape's.

Two details worth remembering. Out-of-range Mach numbers *hold the nearest tabulated* C_d; there is no extrapolation off either end of your deck, and the source says why (`src/drag.rs:136–137`): *"A table has no information beyond its measured Mach domain. Hold the nearest endpoint rather than extending the local edge slope indefinitely (which can drive Cd to 0.01)."* And the transonic correction and any name-derived form-factor multiplier are *not* applied to a custom table, on the grounds that the measured curve already encodes the real projectile's transonic behaviour and correcting it again would double-count.

10.8 What "solving" produces

A solve is not a number. It is a document, and the shape of that document is the whole reason the Lab can make the provenance claims it makes. One request on stdin, one compact JSON envelope on stdout, exit code 0.

The success envelope has seven top-level members: `schema_version`, `engine_version`, `status`, `resolved_request`, `assumptions`, `warnings`, `summary` and `samples`.

10.8.1 The summary

Listing 10.14: `summary` for the reference load at 1000 yd

```
{
  "actual_range_m": 914.4,
  "maximum_height_m": 0.04111759127084448,
  "time_of_flight_s": 1.7064011690199543,
  "terminal_speed_mps": 349.28132093557986,
  "terminal_energy_j": 691.7138558400849,
  "stability_factor": 3.7979628593634613,
  "termination": "max_range"
}
```

`actual_range_m` is the range the bullet *reached*, which is not necessarily the range you asked for. `termination` tells you which:

`termination`	**Meaning**
`max_range`	The run reached the requested range.
`ground_threshold`	The bullet fell below the ground threshold first.
`time_limit`	An internal flight-time limit was hit.
`velocity_floor`	The bullet slowed below the solver's floor.

The field is populated from explicit engine metadata, never inferred by the consumer. `maximum_height_m` is the greatest *world-vertical* height above the same datum as `rifle.muzzle_height_m` — not the height above the line of sight, and not the shot-frame y. For this load it is 0.0411 m, 1.62 in, which is the 1.5-inch sight height plus the tenth of an inch or so the bullet actually climbs above the sight line before falling back through it.

`termination` is the field that tells you whether you got what you asked for

Ask this load for 60,000 yards (54 864 m) and the run does not fail; it terminates on the ground and says so: `termination: ground_threshold`, `actual_range_m: 1914.797910944317` — 2093.99 yards. A trajectory that "worked" but stopped short is indistinguishable from a full-range one unless you read `termination`. The Lab surfaces it on the first screen of `:solve`.

10.8.2 The samples

Listing 10.15: `samples` at a 457.2 m interval

```
[
 { "distance_m": 0.0,   "time_s": 0.0,
   "speed_mps": 822.9599999999999, "energy_j": 3840.017532297961,
   "drop_m": 0.038099999999999995, "windage_m": 0.0,
   "mach": 2.415158238069403, "flags": [] },
 { "distance_m": 457.2, "time_s": 0.6732866256269442,
   "speed_mps": 561.1954982086846, "energy_j": 1785.6819580395331,
   "drop_m": 1.4943272816046234,  "windage_m": 0.0,
   "mach": 1.6469523800259647, "flags": [] },
 { "distance_m": 914.4, "time_s": 1.7064011690199543,
   "speed_mps": 349.28132093557986, "energy_j": 691.7138558400849,
   "drop_m": 9.979643353612667,  "windage_m": 0.0,
   "mach": 1.025043330977604, "flags": ["transonic","terminal"] }
]
```

Note the first sample: `drop_m` is 0.0381, which is exactly the sight height of 1.5 inches. At the muzzle the bullet is one sight-height *below* the optic's line, which is the clearest possible statement that `drop_m` measures depth below the sight line. It is also the one drop value in the whole response that does not depend on how the zero was solved: the bore starts where it starts.

`windage_m` is positive to the shooter's right. The terminal sample always appears exactly once, even when it falls off the interval grid — ask for a 300 m interval on a 914.4 m shot and you get samples at 0, 300, 600, 900 and then 914.4 flagged `terminal`, not a truncated table and not a duplicated last row.

10.8.3 Flags

The `flags` array marks samples the shooter should look at twice. Extending the same load to 1600 m with a 100 m sampling interval shows the boundaries precisely:

Listing 10.16: Flag boundaries, measured

```
 700.0 m  mach=1.298375  flags=[]
 800.0 m  mach=1.165806  flags=['transonic']
 900.0 m  mach=1.041893  flags=['transonic']
1000.0 m  mach=0.950497  flags=['transonic', 'subsonic']
1100.0 m  mach=0.906387  flags=['transonic', 'subsonic']
1200.0 m  mach=0.871440  flags=['transonic', 'subsonic']
1300.0 m  mach=0.840260  flags=['transonic', 'subsonic']
1400.0 m  mach=0.811485  flags=['transonic', 'subsonic']
1500.0 m  mach=0.784432  flags=['subsonic']
1600.0 m  mach=0.758780  flags=['subsonic', 'terminal']
```

`transonic` covers Mach 0.8 through 1.2 inclusive; `subsonic` is Mach below 1.0. Both appear together from 0.8 up to just under 1.0, which is correct and not a bug. `terminal` marks the last sample; `ground_threshold` marks a sample that ended the run in the dirt.

10.8.4 Assumptions and warnings

Everything the engine filled in for you is reported. The reference request in this chapter — which specifies mass, diameter, length, drag model, BC, muzzle velocity, sight height, muzzle height, twist, altitude, temperature, pressure, humidity, range, zero and target height — still draws **ten** `default_applied` assumptions, because it says nothing about aim azimuth, shot azimuth, shooting angle, cant, ground threshold, wind, time step, Magnus, Coriolis or enhanced spin drift.

The list is worth reading for what is *not* on it. There is no `Target height defaulted to 0 m above the local ground datum` line, because the request carries `target_height_m` explicitly. Seeing that assumption in a Ballistics Lab run would mean something had gone wrong upstream of the engine.

That is the design working as intended. An assumption is not a complaint; it is the engine writing down, in machine-readable form, every value it invented on your behalf. The Lab prints them under every `:solve` and stores them in every table's provenance block — which is what makes a Lab DOPE card auditable six months later.

10.8.5 The resolved request

`resolved_request` is not a replay of your input. It is a distinct type with every documented default materialised as a concrete value, so a run is reproducible from the response alone. `temperature_k`, `pressure_pa` and the effective `shot.muzzle_angle_rad` are *always* present, even when you omitted them. Values that are semantically inapplicable — projectile length when nothing needs it, latitude when Coriolis is off — stay absent rather than being invented.

10.9 Two binaries, one answer

The machine this book was written on carries two engine builds: `0.30.1` in the source checkout and `0.32.0` on `PATH`. They differ in subcommands — `0.32.0` adds six online reverse solvers that `0.30.1` does not have — but the question that matters for a physics chapter is whether they differ in *answers*.

Eight representative requests were run through both and their `summary` and `samples` objects compared as sorted JSON:

Listing 10.17: Cross-version comparison, solve-json path

```
baseline     0.32.0 vs 0.30.1: identical=True
euler        0.32.0 vs 0.30.1: identical=True
rk4          0.32.0 vs 0.30.1: identical=True
G1           0.32.0 vs 0.30.1: identical=True
crosswind    0.32.0 vs 0.30.1: identical=True
segwind      0.32.0 vs 0.30.1: identical=True
hotthin      0.32.0 vs 0.30.1: identical=True
altitude     0.32.0 vs 0.30.1: identical=True
ALL IDENTICAL: True
```

Byte-identical, in every case. That is not an accident: v1's compatibility rules say that removing a field, changing a unit or a sign, renaming an enum value or changing a field's meaning all require a v2, and that a new response field must be *omitted* whenever its feature is inactive so an unexercising response stays byte-identical to one from before the field existed.

Pin the protocol, not the binary

The Lab can require an exact engine version through the `BALLISTICS_ENGINE_VERSION` environment variable, and a `VerifiedRun` artifact records whatever version produced it. Use the version pin when you are reproducing a specific historical result. For ordinary work, the schema version is the contract, and it has held.

10.10 The Lab and the engine agree

One last cross-check, because it is the foundation of everything in Parts II and III. The `solve-json` request built by hand for this chapter and the request the Lab builds from `projectile()`, `rifle()`, `atmosphere()`, `shot()` and `solver_config()` describe the same experiment. Do they produce the same answer?

The hand-built request's terminal sample reads `drop_m: 9.979643353612667` and `terminal_speed_mps: 349.28132093557986`. Convert: $9.979643\ \text{m} = 392.90\ \text{in}$, and $349.2813\ \text{m/s} = 1145.94\ \text{ft/s}$. The Lab, asked `:at 1000 yd` on the same load:

Listing 10.18: The Lab reproducing the hand-built solve

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.94 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
```

The agreement is not automatic, and it is worth saying why. The hand-built request in this chapter carries `target_height_m: 0.0381` — the same value the BALLISTICS LAB derives from a 1.5-inch sight height on a rifle at ground level. Omit that one field from the hand-built request and the engine solves a different zero, and the two surfaces part company by sight height $\times\ d/d_{\text{zero}}$: fifteen inches at 1000 yards for this rifle. Two requests that differ in nothing else are two different questions.

That caveat aside, the point of the whole architecture stands: the Lab adds units, validation, provenance and a notebook, and adds *nothing* to the physics.

10.11 What the model does not include

Being clear about the boundary is part of trusting the answers.

- **No aerodynamic bullet model.** The projectile is a point with a drag coefficient. Yaw, precession and pitch damping are computed and reported, but they do not push the point mass around.
- **No barrel or launch dynamics.** Muzzle velocity is an input. Barrel time, muzzle blast, and tip-off are not modelled except through the optional `tipoff_yaw` library field.
- **No shot-to-shot variation.** A solve is deterministic. Dispersion lives in the `monte-carlo` subcommand and in the chapter on uncertainty.
- **No terrain.** The ground is a plane at a fixed datum, and the only thing it does is terminate the run.

- **No time-varying weather.** Atmosphere and wind vary with *distance* and *height*, never with the clock. A gust that arrives halfway through a string is outside the model entirely.

With the equation, the integrator and the drag law understood, the two biggest practical levers on a real firing solution are what the air is made of and what it is doing. Those are the next two chapters.

Chapter 11

Atmosphere and Environment

In the drag law from Chapter 10, air density appears as a plain linear multiplier:

$$a_{\text{drag}} = \frac{C_d(M)\; v_{\text{rel}}^2\; (\rho/\rho_0)\; k}{\text{BC}}$$

Halve the density and you halve the drag. Nothing else in the model is that simple, and nothing else in a shooter's day-to-day environment moves as far. A rifle carried from a cold coastal range to a hot mountain range can lose a quarter of its drag without a single thing changing about the rifle, the load, or the shooter.

This chapter is about the four numbers that set ρ — altitude, pressure, temperature and humidity — how they interact, why one derived number (*density altitude*) is worth more than the four of them separately, and exactly where that convenient summary starts to lie.

The experiment behind every number here

Unless labelled otherwise, every figure is a `solve-json` run of one load: a 175 gr, 0.308-inch, 1.24-inch-long G7 match bullet with a BC of 0.243, leaving at 2700 ft/s (822.96 m/s), zeroed at 100 yd (91.44 m), sight height 1.5 in, solved to 1000 yd (914.4 m) with RK45 in still air. "Drop" is the terminal sample's `drop_m`: metres *below* the line of sight. Only one atmospheric quantity changes between rows of any table. Where the CLI is used instead, it is said so, because the CLI's defaults are not the protocol's.

11.1 The standard atmosphere

Before you can talk about a *non*-standard day, you need a standard one. The engine carries the full ICAO Standard Atmosphere layer stack out to 84 km (`ICAO_LAYERS`, `src/atmosphere.rs:37`), of which the troposphere is all anyone shooting a rifle will ever touch:

Quantity	Sea-level standard
Temperature	288.15 K (15 °C, 59 °F)
Pressure	101 325 Pa (1013.25 hPa, 29.92 inHg)
Lapse rate	-0.0065 K/m to 11 km
Gravity	9.80665 m/s^2
Gas constant R_{air}	287.0531
Ratio of specific heats γ	1.4

The tropopause sits at 11 km and is isothermal at 216.65 K. The model is valid from -5000 m to 84 000 m. Geopotential height uses an Earth radius of 6 356 766 m.

You can watch the standard atmosphere materialise. Send a request whose `atmosphere` object contains *only* `altitude_m` and read what comes back in `resolved_request`:

Altitude (m)	T (K)	P (Pa)	Drop (m)	Terminal (m/s)
0	288.150000	101325.0000	9.9796433536126674	349.281
500	284.900256	95461.2901	9.5801567903516425	367.612
1000	281.651022	89876.2862	9.2208760171833895	386.051
1500	278.402300	84559.6782	8.8967195253705889	404.341
2000	275.154089	79501.4265	8.6029773142392756	422.359
3000	268.659198	70121.1647	8.0929316840314645	457.248

Two thousand metres of station altitude removes 1.3767 m of 1000-yard drop — 54.2 inches, or 1.51 mil. That is a hit or a miss on any target smaller than a car door, and it happened without touching the rifle.

The engine writes down what it did, too. The last row's assumptions include:

Listing 11.1: The engine reporting a standard-atmosphere substitution

```
- icao_standard_temperature: Station temperature was omitted; ICAO standard temperature at
   3000.000000 m is 268.659198451641 K. [$.atmosphere.temperature_k]
- icao_standard_pressure: Station pressure was omitted; ICAO standard pressure at 3000.000000
   m is 70121.164749995369 Pa. [$.atmosphere.pressure_pa]
```

Those two codes — `icao_standard_temperature` and `icao_standard_pressure` — are distinct from the general `default_applied`, precisely so that a reader of a saved artifact can tell "I did not supply a temperature and the engine derived one from altitude" from "I did not supply a cant angle and the engine used zero".

11.2 Density: what the engine actually computes

Air density is not P/RT. The engine uses the CIPM-2007 moist-air formula (`calculate_air_density_cimp`, `src/atmosphere.rs:530`), which adds a compressibility factor and a water-vapour mole fraction to the ideal-gas skeleton:

$$\rho \;=\; \frac{p\,M_a}{Z\,R\,T}\Big(1 - x_v\big(1 - M_v/M_a\big)\Big)$$

with $R = 8.314472$, $M_a = 28.96546 \times 10^{-3}$ kg/mol, $M_v = 18.01528 \times 10^{-3}$ kg/mol, Z the compressibility factor, and x_v the mole fraction of water vapour derived from an IAPWS saturation-vapour-pressure formula and a CIPM enhancement factor.

Why humid air is lighter

$M_v < M_a$: a water molecule weighs about 62% of the average air molecule it displaces. So the $(1 - x_v(1 - M_v/M_a))$ term is always ≤ 1, and adding moisture always makes air *less* dense. The intuition that humid air is "heavy" is backwards for ballistics. The engine's own source comment records the reference value: at 15 °C, 1013.25 hPa and 50% relative humidity the CIPM-2007 moist density is about 1.2211 kg/m^3, against 1.2254 for dry air.

Density is not the only thing the atmosphere sets. The engine also computes a **moist speed of sound** (`moist_speed_of_sound`, `src/atmosphere.rs:497`), and that number decides the Mach at which the bullet is flying, and therefore which part of the C_d curve it is riding. Hold that thought; it is the whole of Section 11.5.3.

11.3 The four dials

Here is each atmospheric input moved alone, everything else held at the sea-level standard, 1000-yard drop reported in metres.

11.3.1 Temperature

Constant 1013.25 hPa station pressure, 50% relative humidity, zero altitude:

T	Drop (m)	Terminal (m/s)	ToF (s)
−20 °C	10.94756258851759	312.620	1.809417
−10 °C	10.638242755143857	323.097	1.776803
0 °C	10.356689219407684	333.660	1.746996
15 °C	9.9796433536126674	349.281	1.706401
25 °C	9.7496440952879446	359.601	1.681286
35 °C	9.5283924372000488	370.045	1.657042
45 °C	9.3119182784745682	380.891	1.632970

A 65 °C swing — a hard winter morning to a hot afternoon — moves the 1000-yard impact by 1.635644 m: **64.40 inches**, or **1.79 mil**. Hot air is thin air; a warm day flattens the trajectory.

11.3.2 Pressure

Constant 288.15 K, 50% humidity, zero altitude:

P	Drop (m)	Terminal (m/s)	ToF (s)
85 000 Pa	8.763849874832503 8	412.037	1.570337
90 000 Pa	9.10981986899634 32	392.056	1.609652
95 000 Pa	9.476437143986155 2	372.683	1.651007
101 325 Pa	9.9796433536126674	349.281	1.706401
105 000 Pa	10.292078064512259	336.964	1.740140

850 to 1050 hPa spans 1.528228 m — 60.17 inches, 1.67 mil. Note the 850 hPa row: that is roughly the pressure at 1500 m, and its drop of 8.764 m is *lower* than the 1500 m ICAO row's 8.897 m from Section 11.1, because here the air is thin *and* still 15 °C warm, whereas the standard atmosphere at 1500 m is 5 °C.

11.3.3 Humidity

Constant 101 325 Pa and 35 °C — chosen deliberately, because humidity only matters when the air is warm enough to hold water:

RH	Drop (m)	Terminal (m/s)
0%	9.591390212179440 8	367.015
25%	9.559769821596724 2	368.528
50%	9.5283924372000488	370.045
75%	9.4970520291582723	371.566
100%	9.46591683627779 55	373.093

Bone dry to saturated, at 35 °C, is 0.125473 m: **4.94 inches**, **0.137 mil**. And in the correct direction — wetter air is thinner, so the bullet drops *less*. At 15 °C the same swing would be a fraction of that, because cold air holds almost no vapour.

11.3.4 Ranked

Dial, over a realistic span	Δdrop (m)	inches	mil
Temperature, −20 to +45 °C	1.635644	64.40	1.789
Pressure, 850 to 1050 hPa	1.528228	60.17	1.671
Station altitude, 0 to 2000 m (ICAO)	1.376666	54.20	1.506
Humidity, 0 to 100% at 35 °C	0.125473	4.94	0.137

Humidity is a rounding error; the other three are not

Humidity is an order of magnitude smaller than the rest, and that is with the temperature stacked in its favour. Get the pressure and temperature right and take 50% for humidity if you have nothing better — the error you buy is under a fifth of a mil at 1000 yards. Get the *pressure* wrong by 50 hPa and you have bought yourself half a mil.

11.4 The redundancy trap: altitude versus pressure

Altitude and pressure are not independent inputs. Station pressure *is* what altitude does to the air. Give the model both and you have over-determined it, and the two surfaces of this engine resolve that over-determination in **opposite ways**. This is the single most important thing in this chapter.

11.4.1 What the CLI does

The CLI treats a temperature or pressure left at its sea-level default, at a non-zero altitude, as *omitted*. The rule is in the source, with tolerances:

Listing 11.2: The omission sentinel, `src/atmosphere.rs:152–159`

```
pub fn resolve_station_pressure(pressure_hpa: f64, altitude_m: f64) -> Option<f64> {
    const SEA_LEVEL_HPA: f64 = 1013.25;
    if (pressure_hpa - SEA_LEVEL_HPA).abs() < 0.5 && altitude_m.abs() > 1.0 {
        None // pressure left at default + real altitude -> derive station pressure from altitude
    } else {
        Some(pressure_hpa) // explicit station pressure is authoritative
    }
}
```

A matching rule for temperature uses a 0.1 °C tolerance around 15 °C. The rationale is sound: without it, `--altitude` with the default pressure produced sea-level density and altitude had no effect on drag at all.

The consequence is a band of pressures you cannot enter. Here is `--altitude 5000` with a sweep of `--pressure` values, time of flight to 1000 yards:

`--pressure`	ToF (s)	`--pressure`	ToF (s)
24.0	1.559623739	29.915	1.583553689
26.0	1.614033078	29.92	1.583553689
28.0	1.672436670	29.925	1.583553689
29.0	1.703231677	29.93	1.583553689
29.5	1.719037667	29.935	1.583553689
29.900	1.731870954	29.939	1.733133710
29.905	1.732032789	29.94	1.733166101
29.91	1.583553689	31.0	1.767944743

`--altitude 5000` with no `--pressure` at all gives 1.583553689 s. Everything from about 29.91 to about 29.935 inHg collapses onto that value; 29.905 and 29.939 do not. That band is ±0.5 hPa around 1013.25 hPa, exactly as the source says (29.92 inHg is 1013.21 hPa, which is why the band is not centred on 29.92).

A real station pressure of 29.92 inHg at altitude is unreachable

If you are standing at 5000 feet on a day when your barometer genuinely reads 29.92 inHg as *station* pressure — unusual, but not impossible — the CLI will discard it and use the standard-atmosphere pressure for 5000 ft instead, which is about 24.9 inHg. Nine tenths of a second of flight time versus one and a half. There is no warning. Enter it as a QNH with `--pressure-type qnh` (which bypasses the sentinel unconditionally), or nudge the altitude, or use `solve-json`.

11.4.2 What `solve-json` does

The protocol takes the opposite position, deliberately. An explicit `temperature_k` or `pressure_pa` is **authoritative** — even a value of exactly 288.15 K at a non-zero altitude, and even 101 325 Pa at 2000 m. The protocol has explicit presence to work with; it does not need to infer omission from a magic number.

That is a better contract, and it has a sharp edge of its own. Watch what happens in the Lab when you set an altitude but leave the sea-level temperature and pressure in place:

Listing 11.3: Two ways to say "I am at 1524 metres"

```
sessions.replace_atmosphere(lab, atmosphere(m(1524), celsius(15), hpa(1013.25), 0.5))
sessions.replace_atmosphere(lab, atmosphere(m(1524), nil, nil, 0.5))
```

Listing 11.4: First form: explicit sea-level T and P at 1524 m

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.94 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
```

Listing 11.5: Second form: temperature and pressure omitted

```
Observation
  distance: 1000.00 yd
  time    : 1.58 s
  velocity: 1329.44 ft/s
  energy  : 686.66 ft-lb
  drop    : +349.68 in
  windage : +0.00 in
  Mach    : 1.21
  flags   : terminal
```

The first block is *byte-for-byte the sea-level answer* — 392.90 inches, 1145.94 ft/s, identical to the run at `m(0)`. Altitude did nothing, because you told the engine what the air was and it believed you. The second block is the mile-high answer: 349.68 inches, 43.22 inches less, 1.20 mil less, and the bullet arrives at Mach 1.21 — clear of the transonic band that the sea-level run enters.

Pass `nil` for what you did not measure

The Lab's `atmosphere()` takes five optional arguments and `nil` means "I have no reading". That is a real value with a real meaning: it becomes an *omitted* field on the wire, which asks the engine for the standard atmosphere at your altitude. Typing `celsius(15)` because it is the number you remember from the manual is not the same statement at all — it is a measurement claim, and the engine will act on it.

11.4.3 QNH versus station pressure

Almost every pressure a shooter can read off a phone is a **QNH**: a sea-level-corrected altimeter setting, the kind of value a METAR or a weather app reports. Station pressure — what the air at your feet actually weighs — is what the density formula needs. At 5000 feet those differ by about 5 inHg.

The CLI has `--pressure-type` with values `absolute` (the default) and `qnh`. QNH values are reduced to station pressure through the ICAO inverse-barometric relation before use:

$$P_{\text{station}} = \text{QNH} \times \left(1 - \frac{0.0065\,h}{288.15}\right)^{5.25588}$$

A QNH of exactly 1013.25 hPa reduces to precisely the ICAO station pressure at any altitude, which is why `--pressure 29.92 --pressure-type qnh` at 5000 feet (1.583525613 s) lands within a fraction of a millisecond of the plain `--altitude 5000` answer (1.583553689 s). The residual is the inHg-to-hPa conversion: 29.92 inHg is 1013.21 hPa, not 1013.25.

Move off the standard value and QNH mode behaves as an ordinary explicit input: `--pressure 29.93 --pressure-type qnh` gives 1.583751404 s, honoured rather than swallowed by the sentinel.

`solve-json` v1 has no pressure reference

The v1 `atmosphere` object accepts `altitude_m`, `temperature_k`, `pressure_pa`, `relative_humidity` and `latitude_rad`. A `pressure_reference` field is documented in `docs/SOLVE_JSON_V1.md` but is **not implemented** in either the 0.30.1 or the 0.32.0 binary:

```
{"schema_version":1,"status":"error","error":{"code":"unknown_field",
 "message":"unknown field `pressure_reference`",
 "path":"$.atmosphere.pressure_reference","line":null,"column":null}}
```

Exit code 2. If you feed the Lab a QNH thinking it is a station pressure, nothing will catch it. Reduce it yourself, or read your barometer in station-pressure mode.

11.5 Density altitude

Four dials is three too many for a range card. The shooting world's answer is **density altitude**: the altitude in the standard atmosphere at which the air would have the density you are actually standing in.

Density altitude

The altitude in the ICAO standard atmosphere whose air density equals the air density at your position. A hot day at sea level has a *positive* density altitude; a bitter day at sea level has a negative one. The engine's own `--density-altitude` help names the company it keeps: Shooter, Nosler, AB Analytics, Ballistic AE and TRASOL all accept it as a single-value atmosphere input.

The engine reports and consumes density altitude using the published NWS/FAA model, quoted verbatim from the source (`src/atmosphere.rs:336–348`):

Listing 11.6: The density-altitude model, and its inverse

```
pressure_alt_ft = 145366.45 * (1 - (station_hpa / 1013.25)^0.190284)
isa_temp_f      = 59.0 - pressure_alt_ft / 1000.0 * 3.57
density_alt_ft  = pressure_alt_ft + (120*5/9) * (station_temp_f - isa_temp_f)

pressure_alt_ft = (density_alt_ft - K*(station_temp_f - 59.0)) / (1 + K*3.57/1000.0)
station_hpa     = 1013.25 * (1 - pressure_alt_ft/145366.45)^(1/0.190284)
```

`--density-altitude` back-solves an ISA-equivalent station altitude and pressure from the number you give it. It is important that this is a back-solve and not a direct-density bypass: Mach number, lapse rate and downrange atmosphere behaviour are all preserved, because the engine ends up with a genuine (altitude, T, P) triple rather than a naked ρ.

11.5.1 Verifying the definition

If density altitude means what it says, then declaring a density altitude of 5000 ft and standing at a standard-atmosphere station altitude of 5000 ft should give the same trajectory. They do:

Listing 11.7: `--density-altitude 5000` against `--altitude 5000`

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact --density-altitude 5000 -o json \
  | jq '.time_of_flight, .impact_velocity'
1.5834963049525075
1329.7193566385083

$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact --altitude 5000 -o json \
  | jq '.time_of_flight, .impact_velocity'
1.583553689
1329.6218724843743
```

0.1 ft/s and 58 microseconds apart. The definition holds.

11.5.2 The scale

Sweeping density altitude with everything else standard, again from the CLI:

DA (ft)	ToF (s)	Impact (ft/s)	Zero angle (°)
0	1.7062599289308455	1146.118256010433	0.06373386118322298
2000	1.653180594662773	1219.5600446944816	0.06364643476184682
5000	1.5834963049525075	1329.7193566385083	0.06347158190190945
8000	1.523952795812753	1436.911088390571	0.06329672907634216
12000	1.4572786722370215	1572.7238885377017	0.06312187623358984

Twelve thousand feet of density altitude turns a bullet that arrives transonic at 1146 ft/s into one that arrives comfortably supersonic at 1573 ft/s. Note also the zero angle: it changes by six ten-thousandths of a degree across the whole sweep. **Density altitude barely touches a 100-yard zero and completely rewrites a 1000-yard one** — which is why you re-true your DOPE for a new elevation and do not re-zero your rifle.

11.5.3 Does density altitude actually summarise the atmosphere?

Mostly. Not entirely. Here is the honest version.

Take the NWS formula above, compute density altitude for a spread of genuinely different conditions — all at 0% humidity so nothing else varies — and put the computed DA next to the trajectory each one produces:

T	P (hPa)	Alt (m)	DA (ft)	Drop (m)
−5.000 °C	1013.25	0	−2400.0	10.500023396047045
15.000 °C	1013.25	0	0.0	10.00117998476626
0.000 °C	950.00	0	393.8	9.8102222370944734
35.000 °C	1013.25	0	2400.0	9.59139021217944O8
5.094 °C	843.11	1524	4997.3	8.891318908516723O
30.000 °C	900.00	0	5813.3	8.88602667194708 22
15.000 °C	850.00	0	5916.6	8.781244667847772 7

Drop falls monotonically with density altitude across a set of conditions that have almost nothing else in common: sea level and a mile up, freezing and hot, 1013 hPa and 843 hPa. That is a strong endorsement of the idea.

But look at the last three rows. A standard mile-high station (DA 4997) and a hot low-pressure sea-level day (DA 5813) — 816 ft apart in density altitude — give drops within half a centimetre of each other. Meanwhile the DA 5813 and DA 5917 rows, only 103 ft apart, differ by 10.5 cm. Density altitude ranks these atmospheres correctly but does not scale them perfectly.

11.5.4 The residual is the speed of sound

Here is the cleanest way to see what density altitude leaves on the table. Hold the *dry-air density* exactly constant — vary temperature and scale pressure by the same ratio, so P/T never moves — and set humidity to zero so density is genuinely fixed:

T	P (Pa)	ρ_{dry}	Drop (m)	Terminal (m/s)
−20 °C	89 017.61	1.224999	9.7573157014291585	359.473
−10 °C	92 534.01	1.224999	9.829003210174992 2	356.051
0 °C	96 050.40	1.224999	9.8979617160962921	352.813
15 °C	101 325.00	1.224999	10.00117998476626	348.353
30 °C	106 599.60	1.224999	10.100893905725004	344.978
45 °C	111 874.19	1.224999	10.199767751502929	344.437

Identical density, and the 1000-yard drop still moves 0.442452 m across the sweep — **17.42 inches**, **0.48 mil**. The same −20 to +45 °C swing at constant *pressure* and the same zero humidity spans 1.537406 m (10.949581723979742 down to 9.4121754082279736). So density explains about **71%** of the temperature effect and the speed of sound explains the other **29%**.

The mechanism is worth spelling out because it is not obvious. Warmer air has a higher speed of sound, so the same bullet flies at a *lower* Mach number. Look back at the G7 table in Section 10.4: through the supersonic region, C_d *rises* as Mach falls — 0.2424 at Mach 3, 0.2980 at Mach 2, 0.3440 at Mach 1.5. So dropping the Mach number moves the bullet onto a higher-drag part of the curve.

Hot air is thin, which helps, and it is also acoustically slow, which hurts, and density altitude only accounts for the first.

Density altitude, used honestly

Density altitude is the right single number for organising DOPE, for comparing range days, and for deciding whether today's card is close enough to yesterday's. It is not a complete description of the air. If you are truing at 1000 yards and the temperature has moved 30 °C since you last shot, expect a few inches that density altitude will not account for. Give the engine the real temperature as well — `--density-altitude` plus an explicit `--temperature` re-solves the pressure so your DA is still reproduced exactly.

Verified: holding `--density-altitude 5000` and sweeping the explicit temperature moves the answer exactly as much as the residual predicts.

`--temperature`	ToF (s)	Impact (ft/s)
20 °F	1.5775771640289027	1340.3270964273 03
50 °F	1.5860665581221165	1325.1824105280 58
80 °F	1.5946536895792325	1310.2887207899 76
110 °F	1.6015779648417 91	1298.6065282253 808

Same density altitude, 90 °F of temperature, 41.7 ft/s of impact velocity.

11.6 Atmosphere segments: the feature with no wire

Wind can vary downrange, and Chapter 12 is largely about the machinery that lets it. Air *density* can vary downrange too — a valley shot that starts in cold air over water and ends in hot air over rock is a real situation — and the engine has a model for it:

Listing 11.8: `src/atmosphere.rs:856-872`

```
/// A single downrange-referenced atmosphere zone:
/// `(temp_c, pressure_hpa, humidity_percent, until_distance_m)`.
pub type AtmoSegment = (f64, f64, f64, f64);

/// Downrange-segmented atmosphere handler (MBA-1137), the density analogue of
/// `crate::wind::WindSock`.
pub struct AtmoSock {
    /// Zones sorted ascending by `until_distance_m` (segment slot 3).
    segments: Vec<AtmoSegment>,
}
```

The shape deliberately mirrors a wind segment, and the composition rule is spelled out in the source: the zone sets the base station-referenced temperature, pressure and humidity for that stretch of range, and the vertical altitude lapse then multiplies on top of it, so downrange variation and vertical variation compose without double-counting. There is one behavioural difference from wind worth noting: past the final boundary, wind goes to zero, but the *last* atmosphere zone continues — there is no such thing as "no atmosphere".

Now the honest part. `AtmoSock` is reachable only from the Rust library API, through `TrajectorySolver::set_atmo_segments` (`src/cli_api.rs:1947`). Grep the whole engine for callers and you find exactly one, inside a test. There is **no CLI flag** and **no `solve-json` field**. The protocol rejects the obvious guesses:

Listing 11.9: Neither field exists in v1

```
{"schema_version":1,"status":"error","error":{"code":"unknown_field",
 "message":"unknown field `density_altitude_m`",
 "path":"$.atmosphere.density_altitude_m","line":null,"column":null}}
```

A capability with no transport

Segmented atmosphere is one of the capabilities the Lab's design notes single out as impossible to express through a flat C struct: it is a variable-length array of four-tuples, and `set_atmo_segments` takes a `Vec`. The design document those notes come from proposed a native Rust extension linked into LATTICE precisely to reach features like this. **That extension was never built.** What shipped is the subprocess bridge, and the subprocess speaks v1, and v1 does not carry atmosphere zones. The Lab's wind support (which *is* variable-length, and *is* in v1) shows the pattern that segmented atmosphere would follow if it ever crossed the wire.

11.7 Working the atmosphere in the Lab

The Lab's atmosphere is one immutable struct built by one constructor:

Listing 11.10: `atmosphere()`, from `domain.lat:288`

```
atmosphere(altitude = nil, temperature = nil, pressure = nil,
           relative_humidity = nil, latitude = nil)
```

Every argument is optional and every one is a typed quantity, not a bare number: `m(1524)` or `ft(5000)` for altitude, `celsius(15)` or `fahrenheit(59)` or `kelvin(288.15)` for temperature, `hpa(1013.25)` or

`inhg(29.92)` or `pa(101325)` for pressure. Relative humidity is a bare fraction from 0 to 1 — the one exception, because it is dimensionless. Latitude is an `Angle` and exists only to feed Coriolis.

A worked session. Baseline, then a hot day, then a cold one:

Listing 11.11: Three atmospheres

```
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(15), hpa(1013.25), 0.5))
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(35), hpa(1013.25), 0.5))
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(-20), hpa(1013.25), 0.5))
```

Listing 11.12: 15 °C, then 35 °C, then −20 °C at 1000 yd

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.94 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : +0.00 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.66 s
  velocity: 1214.06 ft/s
  energy  : 572.64 ft-lb
  drop    : +375.13 in
  windage : +0.00 in
  Mach    : 1.05
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.81 s
  velocity: 1025.66 ft/s
  energy  : 408.70 ft-lb
  drop    : +431.01 in
  windage : +0.00 in
  Mach    : 0.98
  flags   : transonic, subsonic, terminal
Signs: drop + below/- above; windage + right/- left
```

375.13 to 431.01 inches: 55.88 inches, 1.55 mil, between a hot afternoon and a hard winter morning. And read the `flags` row on the cold run: the bullet has gone **subsonic** before 1000 yards. That is the atmosphere changing not just where the bullet lands but what kind of shot it is.

11.7.1 Relational validation bites here

The Lab re-validates the *whole* experiment on every partial edit. The atmosphere carries latitude, and Coriolis needs latitude, so replacing the atmosphere on an experiment with Coriolis enabled fails if the new atmosphere has none:

Listing 11.13: A partial edit rejected by a cross-object rule

```
Error
  code: evaluation_error
  message: assertion failed: experiment.atmosphere.latitude: required when Coriolis is enabled
```

The Lab's default experiment has Coriolis on and latitude 45 ° (`reference_experiment.lat`), so this catches almost every reader the first time they replace an atmosphere. The fix is ordering: turn Coriolis off (or supply a latitude) before you drop the old atmosphere.

Listing 11.14: Solver first, atmosphere second

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, false, nil))
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(15), hpa(1013.25), 0.5))
```

Nothing was left half-changed

That error is not a rollback — it is a refusal. `session.lat` builds a complete candidate state and publishes it with a single `Ref.set`, so a validation failure leaves the session exactly as it was. Run `:show` after the error and the old atmosphere is still there, intact, with the old latitude.

11.8 Zero-day versus shot-day atmosphere

One asymmetry the Lab cannot express. A rifle zeroed at 500 ft in October and fired at 6000 ft in July was zeroed in different air from the air it is shooting in, and that changes the launch angle the zero implies.

The CLI models this with a parallel set of zero-day flags: `--zero-temperature`, `--zero-pressure`, `--zero-pressure-type`, `--zero-humidity`, `--zero-altitude`, `--zero-density-altitude`, `--zero-velocity`

and `--zero-powder-temp`, all used with `--auto-zero`. Omitting them all reuses the shot-day values, which is the behaviour you get everywhere else.

`solve-json` v1 has one atmosphere and one `zero_distance_m`; the zero trial runs in the same air as the shot. For most work that is right — if you confirm your zero at the range you shoot from, the two atmospheres are the same atmosphere — and when it is not, that is a case for the CLI.

11.9 Summary

- Density multiplies drag linearly, and it is the environmental quantity with the widest practical range.
- Altitude, pressure and temperature each move 1000-yard drop by roughly a mil and a half over a realistic span. Humidity moves it by about a seventh of a mil, and only when it is hot.
- Altitude and pressure are redundant. The CLI resolves the redundancy with a ± 0.5 hPa omission sentinel; `solve-json` resolves it by believing whatever you explicitly send. Know which surface you are on.
- In the Lab, `nil` means "derive this from altitude". It is not the same as typing the standard value.
- Density altitude is an excellent organising number and an imperfect physical one: at fixed density, a 65 °C temperature swing still moves 1000-yard drop by 17.4 inches through the speed of sound.
- Segmented atmosphere exists in the engine's Rust API and nowhere else.

The air's density is half the environment. The other half is what the air is doing, and that is the next chapter.

Chapter 12

Wind

Elevation is arithmetic. Wind is judgement. Once you have trued a load, the elevation number for a known range is repeatable to a tenth of a mil and stays repeatable all afternoon; the wind number is a guess about a fluid you cannot see, made at one end of a corridor whose far end may be doing something else entirely. Every serious long-range miss is a wind miss.

This chapter is about what the engine actually does with wind — which is surprisingly little, and surprisingly precise — and about the one place where the Lab's architecture visibly earns its keep: **wind segments**, a variable-length array that the flat-struct C interface the Lab was originally designed around could not have carried at all.

The load behind the numbers

Same experiment as the previous two chapters: 175 gr, 0.308-inch, G7, BC 0.243, 2700 ft/s, 100 yd zero, 1.5-inch sight height, sea-level standard atmosphere, solved to 1000 yd (914.4 m) with RK45. `windage_m` is metres to the shooter's *right*; negative is left. Every `solve-json` request in this chapter carries `target_height_m: 0.0381` — the height of the line of sight, which is what the Ballistics Lab sends — so the drop figures here and the Ballistics Lab's agree row for row.
Where CLI numbers appear they are the native `-o json` lateral offset `x` in yards, and they are not directly comparable with the `solve-json` figures because the CLI's default bore height is 60 inches while the protocol's default `muzzle_height_m` is zero.

12.1 Wind is not a force on the bullet

There is no "wind force" term in the engine's derivative function. Go back and look at the three lines from Section 10.1:

Listing 12.1: src/derivatives.rs:174–176

```
let wind_vector = level_vector_to_shot_frame(wind_vector, theta);
let velocity_adjusted = vel - wind_vector;
let speed_air = velocity_adjusted.norm();
```

That is the entire wind model. The wind vector is subtracted from the bullet's velocity, and everything downstream — the Mach number, the drag coefficient lookup, the magnitude of the drag acceleration and, crucially, its *direction* — uses the result.

Why a crosswind moves a bullet sideways

A bullet in a left-to-right crosswind is, from the air's point of view, flying slightly sideways. Drag opposes motion *through the air*, so the drag vector tips away from the downrange axis and acquires a lateral component. That lateral component accelerates the bullet toward the downwind side until the bullet is moving with the air laterally. The bullet is not blown; it is *dragged into agreement with the air*. This is why a high-BC bullet drifts less: it comes into lateral agreement more slowly, and there is less time for the deflection to accumulate.

The wind vector itself is built in one function:

Listing 12.2: src/wind.rs:9–20

```
/// THE wind-vector builder (McCoy frame: x downrange, y up, z right).
/// Horizontal wind uses the wind-FROM convention (0 = headwind, PI/2 = from
/// the right); `vertical_mps` is positive-updraft and lands on y UNSCALED --
/// boundary-layer shear models scale horizontal flow only (MBA-728 decision).
pub fn wind_vector(speed_mps: f64, direction_rad: f64, vertical_mps: f64) -> Vector3<f64> {
    Vector3::new(
        -speed_mps * direction_rad.cos(),
        vertical_mps,
        -speed_mps * direction_rad.sin(),
    )
}
```

Three components, two minus signs, and a comment that settles the convention question before you can ask it.

12.2 The wind-FROM convention

Everywhere in this engine, wind direction is the direction the wind is coming *from*, measured clockwise from the shooter's line of fire.

Degrees	Clock	Meaning
0	12 o'clock	Headwind, blowing into your face
90	3 o'clock	From your right; pushes the bullet *left*
180	6 o'clock	Tailwind, blowing from behind you
270	9 o'clock	From your left; pushes the bullet *right*

The minus signs in `wind_vector` are what implement "from": a 90-degree wind gives $z = -s\sin(\pi/2) = -s$, a wind vector pointing left, and a bullet dragged into agreement with it goes left too.

The CLI accepts clock positions directly and treats them as exactly equivalent:

Listing 12.3: Clock and degrees are the same input

```
--wind-direction 90    ->  x = -2.9094616712136196 yd at 1000 yd
--wind-direction 3oc   ->  x = -2.9094616712136196 yd at 1000 yd
```

Bit-identical. `3oc`, `10h30` and `10:30` all parse; `12oc` is a headwind. `solve-json` takes radians only — `direction_from_rad` — and the Lab's `wind()` constructor takes an `Angle` quantity, so `deg(90)`, `rad(1.5708)` and `mil(1570.8)` are all acceptable ways to say the same thing.

Every ballistics program picks a convention, and they differ

Some solvers use wind-*to*; some measure from north rather than from the bore; some report drift positive-left. If you are cross-checking this engine against another program and the magnitudes agree but the signs do not, check the convention before you check the physics. In this engine: direction is **from**, measured from the bore, clockwise, and `windage_m` is positive to the **shooter's right**.

12.3 Crosswind: the dominant term

A 10 mph wind, swept through a full circle of directions, at 1000 yards:

From	Windage (m)	Drop (m)	ToF (s)
0°	0.0	10.090858356580389	1.716753
30°	−1.344275805457878	10.075577183646238	1.715353
45°	−1.8975115833798566	10.057926277929777	1.713696
60°	−2.3182740972435516	10.03478240776848	1.711546
90°	−2.661213207510891	9.979709297807709	1.706408
120°	−2.2912055486401646	9.925205412315755	1.701321
135°	−1.8662551500263598	9.902960108292861	1.699235
150°	−1.3172040506189286	9.885948604873 09	1.697640
180°	-3.2×10^{-16}	9.871646151974591	1.696299
270°	+2.661213207510891	9.979709297807709	1.706408

The 90 and 270 rows are exact mirror images to the last digit. The 180-degree row is zero to floating-point noise. The model is behaving.

2.661213207510891 m is **104.77 inches** — 2.91 mil — of drift from a 10 mph full-value wind at 1000 yards. Compare that with the 4.4-inch swing the *entire* humidity range produced in Chapter 11. This is why wind gets a chapter.

12.3.1 Drift is linear in wind speed

Holding the direction at 90 degrees and sweeping the speed:

Speed	Windage (m)	Per mph (m)
5 mph	−1.330594751649324	−0.266118950
10 mph	−2.661213207510891	−0.266121321
15 mph	−3.9918790707983076	−0.266125271
20 mph	−5.322616042727467	−0.266130802
30 mph	−7.984398101452454	−0.266146603

From 5 to 30 mph — a factor of six — the drift per mph changes in the sixth decimal place: 0.266119 to 0.266147, about one part in ten thousand. For all practical purposes crosswind drift is exactly proportional to wind speed.

This is why a wind card is one column wide

Because drift scales linearly with speed, a card only needs the drift for *one* wind speed at each range; every other speed is multiplication. The engine's `wind-card` subcommand prints several columns anyway, and the Lab's `:wind-card` prints the trajectory's actual wind, but the underlying fact is that you can carry a single 10 mph column and scale it in your head.

12.3.2 The clock-face rule, and its error

The traditional field method takes the full-value drift and multiplies by the sine of the clock angle: a 3 o'clock wind is full value, a 1 or 5 o'clock wind is half, a 2 or 4 o'clock wind is sin 60° ≈ 0.87. How good is it?

From	Sine rule (m)	Actual (m)	Error
30°	−1.330607	−1.344276	+1.02%
45°	−1.881762	−1.897512	+0.83%
60°	−2.304678	−2.318274	+0.59%
120°	−2.304678	−2.291206	−0.59%
135°	−1.881762	−1.866255	−0.83%
150°	−1.330607	−1.317204	−1.02%

The sine rule is good to about 1% — which at 1000 yards is a single inch, far inside anyone's ability to call the wind. But note the *pattern*: it under-predicts for winds with a head component and over-predicts for winds with a tail component, and the two errors are almost exactly symmetric.

The reason is in the previous table's third column. A quartering headwind increases the bullet's speed through the air, which increases drag, which increases time of flight, which gives the lateral drag more time to work. A quartering tailwind does the opposite. The sine rule assumes crosswind and head/tail components act independently; they do not, because time of flight couples them.

Round toward the headwind

If you are doing the clock arithmetic in your head and the wind has any head component, your answer is about one percent light. At 1000 yards on a 10 mph quartering wind that is an inch. Worth knowing; not worth changing your process over.

12.4 Head and tail wind

Head and tail winds produce no drift at all — the 0- and 180-degree rows have exactly zero windage — and they are routinely dismissed as harmless. They are not. They change the *elevation*.

From the Lab, the same rifle in a 10 mph crosswind, a 10 mph headwind, and a 10 mph tailwind:

Listing 12.4: 90°, then 0°, then 180° at 1000 yd

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.96 ft/s
  energy  : 510.20 ft-lb
  drop    : +392.90 in
  windage : -104.77 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.72 s
  velocity: 1130.46 ft/s
  energy  : 496.49 ft-lb
  drop    : +397.28 in
  windage : +0.00 in
  Mach    : 1.01
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.70 s
  velocity: 1161.27 ft/s
  energy  : 523.93 ft-lb
  drop    : +388.65 in
  windage : +0.00 in
  Mach    : 1.04
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
```

Calm drop for this load is 392.90 inches. A 10 mph headwind makes it 397.28 — **4.38 inches more**, 0.12 mil. A 10 mph tailwind makes it 388.65 — 4.25 inches less. The full head-to-tail span is 8.63 inches, or about a quarter of a mil, at 1000 yards.

Impact velocity moves too: 1130.46 ft/s into the wind against 1161.27 with it, a 31 ft/s spread, and the headwind run is a hair closer to the subsonic transition.

A 20 mph headwind is a real elevation correction

A quarter of a mil is inside a 10-inch plate at 1000 yards, so most of the time you can ignore this. Double the wind and it stops being ignorable, and if you are shooting the same course of fire into and then away from a steady wind, the two elevations genuinely differ. The engine models it correctly whether or not you think about it — as long as you tell it the wind direction rather than entering every wind as a pure crosswind.

12.5 Where the wind is matters

Here is the result that motivates everything else in this chapter. A 10 mph 90-degree wind blowing over the *first half* of the range, versus the same wind over the *second half*:

Wind present over	Windage (m)	Share of full
The whole 1000 yd (constant)	−2.661213207510891	100.0%
First half only (0–500 yd)	−1.6850488958361078	63.3%
Second half only (500–1000 yd)	−0.9762422671121747	36.7%
First third (0–333 yd)	−1.1512773667175347	43.3%
Middle third (333–667 yd)	−1.0049964508602518	37.8%
Last third (667–1000 yd)	−0.5045857584818023	19.0%

The first third of the range does more than twice as much to your wind call as the last third. A flag at the muzzle is worth two flags at the target.

This is not intuition; it falls straight out of the model. A lateral acceleration applied early has the whole remaining flight time to turn into displacement; the same acceleration applied in the last 300 yards has a third of a second to work with. Displacement goes as the square of the remaining time.

12.5.1 And it superposes

Add the two halves: $-1.6850488958361078 + -0.9762422671121747 = -2.6612911629482827$, against the constant wind's -2.661213207510891. They differ by 7.8×10^{-5} m — **three thousandths of an inch**. Add the three thirds and you land within 3.6×10^{-4} m of the constant answer.

Crosswind drift adds

To an accuracy far beyond anything you can shoot, the drift from a segmented wind is the sum of the drifts each segment would produce alone. That is what makes mental wind arithmetic — "half value out to the mid-flag, then full value" — legitimate rather than merely convenient.

12.6 Wind segments

A constant wind is a fiction. Real ranges have terrain, tree lines, gullies and mirage boils, and the honest description of a 1000-yard shot is a *list* of wind conditions with the distances they apply over. That list is what `solve-json` v1 calls `wind.segments`.

12.6.1 The wire shape

The `wind` object takes exactly one of two shapes and never both:

Listing 12.5: Constant wind, and segmented wind

```
"wind": { "speed_mps": 4.4704, "direction_from_rad": 1.5707963267948966,
          "vertical_speed_mps": 0.0 }

"wind": { "segments": [
  { "until_distance_m": 400.0, "speed_mps": 4.4704,
    "direction_from_rad": 1.5707963267948966, "vertical_speed_mps": 0.0 },
  { "until_distance_m": 914.4, "speed_mps": 2.2352,
    "direction_from_rad": 4.71238898038469,  "vertical_speed_mps": 0.0 } ] }
```

An empty `wind` object is still air. Each segment applies from the previous boundary out to its own `until_distance_m`, boundaries must *strictly* increase, and `segments` conflicts with all three constant fields.

The second block above is a real `resolved_request.wind` echo: 10 mph from 3 o'clock out to 400 m, then 5 mph from 9 o'clock to the target. Its terminal windage is -0.9040692164451878 m — barely a third of what the constant 10 mph wind produced, because the reversal near the end partially cancels an already-established deflection.

12.6.2 The variable-length problem

The Lab's design notes worked through binding the engine as a native LATTICE extension over its C ABI. The conclusion was blunt: the C boundary is a lossy projection of the engine, and the *hard blockers are everything needing a variable-length array*. Wind segments were the headline example — `TrajectorySolver::set_wind_segments` takes a `Vec<(speed_kmh, angle_deg, until_m)>`, and the flat scalar C struct the FFI exposes has no vocabulary for a list. Atmosphere zones, BC segments, custom drag tables and powder-temperature curves were all in the same boat. Binding the C ABI would have meant adding half a dozen new `extern "C"` wrappers to the engine in the first week.

The dylib extension was never built

That design document proposed a Rust cdylib exporting LATTICE's `lat_ext_*` symbols and statically linking the engine, so the extension could call `set_wind_segments` directly. It is a considered alternative, not the shipped architecture. **What shipped is the subprocess bridge**: the Lab spawns `ballistics solve-json` through `exec_argv()` and speaks solve-json v1. JSON carries arrays natively, so the variable-length problem that sank the C-ABI plan simply does not arise. A wind segment list is an array in the request and an array in the resolved echo, and the Lab needed no engine changes at all to use it.

12.6.3 Segments from the Lab

The Lab's `wind()` constructor has a fourth parameter, `until_distance`, and supplying it is what turns a wind into a segment (`domain.lat:337`):

Listing 12.6: `wind()` and its segment form

```
wind(speed, direction_from, vertical_velocity = nil, until_distance = nil)

// Constant: one wind, no boundary
sessions.replace_winds(lab, [wind(mph(10), deg(90))])

// Segmented: wind over the first 500 yd, calm after
sessions.replace_winds(lab, [wind(mph(10), deg(90), nil, yd(500)),
                             wind(mph(0),  deg(90), nil, yd(1000))])
```

Listing 12.7: First-half wind, then second-half wind

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.93 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : -66.34 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.93 ft/s
  energy  : 510.18 ft-lb
  drop    : +392.90 in
  windage : -38.43 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
```

-66.34 and -38.43 inches, summing to -104.77 against the constant wind's -104.77. The same superposition result, in the units you would actually call the wind in.

Note that the drop and velocity columns are identical across all three cases to the last displayed digit: a pure crosswind changes windage and nothing else worth printing.

12.6.4 Two rules the Lab enforces before spawning anything

`domain.lat`'s `_validate_winds` (lines 446–471) allows exactly two shapes: *one* constant wind, or *all* segmented winds with strictly increasing boundaries. Break either and the error arrives without a process being started:

Listing 12.8: Both wind-list rules, refused locally

```
Error
  code: evaluation_error
  message: assertion failed: experiment.winds: cannot mix constant and segmented wind
Error
  code: evaluation_error
  message: assertion failed: experiment.winds: segment boundaries must increase strictly
```

The first came from `[wind(mph(10), deg(90)), wind(mph(5), deg(270), nil, yd(1000))]` — a bare wind followed by a segmented one. The second came from boundaries listed as `yd(1000)` then `yd(500)`.

12.6.5 Short decks and `partial_wind_coverage`

Wind is zero beyond the last segment boundary — there is no "last wind continues" rule, unlike atmosphere zones. If your deck ends short of the target the engine solves anyway and warns:

Listing 12.9: A 600 m wind deck on a 914.4 m shot

```
[
 {
  "code": "partial_wind_coverage",
  "message": "Segmented wind ends at 600.000000000000 m; wind is calm from that boundary through
     the required 914.400000000000 m solve/zero range.",
  "path": "$.wind.segments"
 }
]
```

That run produced −2.1289360533496913 m of drift instead of the −2.661213207510891 a full-coverage wind would have given: an 80% answer delivered with a warning rather than an error. Coverage is checked through the *farther* of `max_range_m` and `zero_distance_m`, because the zero trial flies through the same wind.

Always close your deck at the target

Make the last segment's `until_distance` equal to (or greater than) your maximum range, even if that last segment is calm. An explicit `wind(mph(0), deg(90), nil, yd(1000))` says "I believe the last stretch is still" — a short deck says the same thing but cannot be distinguished from an oversight, which is exactly why the engine warns about it.

12.6.6 The same thing from the CLI

Listing 12.10: `--wind-segment SPEED:ANGLE:UNTIL[:VERTICAL]`

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact --wind-speed 10 --wind-direction 90 --full -o json \
  | jq '.trajectory[-1].x'
-2.9094616712136196

$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --auto-zero 100 --max-range 1000 --sight-height 1.5 --twist-rate 8 \
    --ignore-ground-impact --wind-segment 10:90:500 --wind-segment 5:270:1000 \
    --full -o json | jq '.trajectory[-1].x'
-1.3086804818479345
```

`--wind-segment` is repeatable and overrides `--wind-speed`/`--wind-direction`/`--wind-vertical` entirely. Two warts to know about: it is **not compatible with `--enable-wind-shear`**, and the optional fourth field `VERTICAL` is **always metres per second** regardless of `--units`, even though `SPEED` in the same colon-separated token follows `--units`. A per-token unit split inside one flag is exactly the sort of thing the Lab's typed quantities exist to prevent.

12.7 Vertical wind

Both the protocol and the CLI accept a vertical wind component: `vertical_speed_mps` on the wire, `--wind-vertical` on the CLI, and the third argument of the Lab's `wind()`. Positive is an updraft. It lands on the y axis unscaled — boundary-layer shear models touch only the horizontal flow.

Most shooters never enter one. They should probably be more nervous about that than they are:

Vertical	Drop (m)	vs. calm
−2.0 m/s	11.089376185521713	+1.110 m
0.0 m/s	9.979643353612667	—
+1.0 m/s	9.424800587509484	−0.555 m
+2.0 m/s	8.869970540852897	−1.110 m

A 2 m/s updraft — about 4.5 mph, a modest thermal over warm rock — removes 1.11 m of 1000-yard drop. That is **43.7 inches**, or **1.21 mil**: five times the entire head-to-tail elevation effect from a 10 mph horizontal wind. The response is essentially perfectly linear at 0.5549 m per m/s.

Do not type a vertical wind you did not measure

Vertical wind is the most powerful and least observable input in this chapter. A guessed updraft will move your solution by more than a badly-called crosswind. The Lab defaults it to `nil` and the engine defaults it to zero; leave it there unless you have an instrument telling you otherwise.

12.8 Wind shear

Real wind is slower near the ground. `--enable-wind-shear` turns on an altitude-dependent horizontal wind profile, referenced to an assumed muzzle height of 1.5 m (`APPROX_MUZZLE_HEIGHT_AGL_M`, `src/wind_shear.rs:12`), with the model selected by `--wind-shear-model`. This is CLI-only: it is on the `docs/SOLVE_JSON_V1.md` list of things explicitly *not* in v1, so the Lab cannot ask for it.

At flat-fire ranges it does nothing measurable. The same 1000-yard, 10 mph 90-degree shot gives $x = -2.909462$ yd with and without `--enable-wind-shear` — identical to six decimal places, because the bullet never climbs far enough above the reference height to leave the boundary layer's top.

Push the launch angle to 25 degrees and the range to 4000 yards and it bites hard:

Setting	Lateral x (yd)
no `--enable-wind-shear`	-60.442884
`--enable-wind-shear` (default power law)	-101.787399
`--wind-shear-model logarithmic`	-97.905574
`--wind-shear-model ekman_spiral`	-60.442884
`--wind-shear-model none`	-101.787399

`--wind-shear-model none` does not disable shear

Look at the last row. `none` produces *exactly* the power-law answer. The match arms in `src/cli_api.rs` cover `"logarithmic"`, `"power_law"` and its aliases, `"ekman_spiral"` and `"custom_layers"`, and then fall through to a catch-all that returns the power law. The string `"none"` lands in the catch-all. To disable shear, omit `--enable-wind-shear`; do not pass `--wind-shear-model none` and assume it turned anything off.

Note also that `ekman_spiral` reproduced the no-shear answer to the last digit at this geometry. Treat the shear models as an area to validate against your own data before trusting them.

12.9 Shooter-relative versus compass wind

Everything so far has measured wind direction from the bore. If you are reading bearings off a weather station instead, you want earth-fixed angles. `wind_reference: "compass"` on the wire, `--wind-ref`

`compass` on the CLI, re-references every wind angle the run consumes — constant, CSV, and every segment — as a bearing, normalised as `bearing - shot_azimuth`.

Listing 12.11: A 90° bearing on a 45° shot

```
$ ballistics trajectory ... --wind-speed 10 --wind-direction 90 \
    --wind-ref compass --shot-direction 45 --full -o json | jq '.trajectory[-1].x'
-2.0745136879224457
```

A wind from due east while shooting northeast is a 45-degree quartering wind, and the same run with a plain `--wind-direction 45` and no compass reference returns `-2.0745136879224457` — bit-identical. The conversion is doing exactly what it says and nothing else.

Two guard rails. Compass mode **requires** a shot azimuth, and the protocol refuses without one rather than silently treating bearings as shooter-relative:

Listing 12.12: The engine refusing an ambiguous request

```
{"code":"conflicting_fields",
 "message":"wind_reference \"compass\" requires shot.shot_azimuth_rad (earth-fixed bearings are
     re-referenced against the shot azimuth; omitting it would silently treat bearings as
     shooter-relative)",
 "path":"$.wind.wind_reference"}
```

And compass mode rejects clock positions, which are shooter-relative by definition.

The Lab has no compass mode

`wind_reference` is accepted by the 0.30.1 and 0.32.0 binaries, but the Lab's `wind()` constructor has no parameter for it and `protocol_v1.lat`'s `winds_to_v1_map` never emits it. In the Lab, wind direction is always measured from the bore. Convert bearings yourself: `deg(bearing - shot_azimuth)`, normalised into $[0, 360)$.

12.10 Wind cards

A wind card is the field artifact this whole chapter exists to produce: drift, or the hold that cancels it, tabulated against range for a reference wind.

12.10.1 From the Lab

`:wind-card <start> <stop> <interval> <unit>` builds one from the current run's actual wind:

Listing 12.13: `:wind-card 200 1000 200 yd` on a 10 mph 3 o'clock wind

```
Wind card
Series | Trajectory             | Wind (mph) | From (deg) | Distance (yd) | Windage (in) | Wind
     hold (mil)
-------+------------------------+------------+------------+---------------+--------------+----------------
  0.00 | 175 gr SMK at 1,000 yd |      10.00 |      90.00 |        200.00 |        -2.96 |
         +0.41
  0.00 | 175 gr SMK at 1,000 yd |      10.00 |      90.00 |        400.00 |       -12.75 |
         +0.89
  0.00 | 175 gr SMK at 1,000 yd |      10.00 |      90.00 |        600.00 |       -31.10 |
         +1.44
  0.00 | 175 gr SMK at 1,000 yd |      10.00 |      90.00 |        800.00 |       -60.55 |
         +2.10
  0.00 | 175 gr SMK at 1,000 yd |      10.00 |      90.00 |       1000.00 |      -104.77 |
         +2.91
Signs: windage + right/- left; wind hold + correction right/- correction left
```

Two columns, two different things. **Windage** is where the bullet goes: negative, left, because the wind is from the right. **Wind hold** is what you do about it: positive, right, because the correction opposes the drift. `analysis.lat:552` produces the hold by negating the windage, which is right for *this* column and only this one: windage is positive-right, so reversing it turns a deflection into a correction.

That is worth stating explicitly, because the elevation half of the Lab carries the *opposite* convention. Drop is positive-*below* the sight line, so a come-up is the drop's own angle with no negation at all: the same load reads $+392.90$ inches of drop and $+10.91$ mil of come-up at 1000 yards, both positive, because "low" and "dial up" are already the same direction. Windage needs the sign flip; elevation does not. Each table's legend names the convention it is using, and reading the legend is how you keep the two apart.

The `Series` column rendering as `0.00` rather than `0` is cosmetic: it is an array index carried through the table machinery as a number.

12.10.2 From the engine, and a trap

The engine has its own `wind-card` subcommand. Its numbers agree with the Lab's exactly — when you call it correctly:

Listing 12.14: `wind-card`, `-start 100 -step 100`: correct

```
$ ballistics wind-card -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --zero-distance 100 --wind-speeds 10 --start 100 --end 1000 --step 100 \
    -o json | jq -r '.data[] | "\(.range) \(.wind_10)"'
100 -0.19849211501450464
200 -0.4109938240164462
300 -0.639436286668842
400 -0.8854525070572374
500 -1.1510955629151618
600 -1.4397215665372955
700 -1.755217478705041
800 -2.1021361048822844
900 -2.4855457226998983
1000 -2.9101297681913385
```

Convert the 1000-yard row: -2.9101297681913385 mil at 1000 yd is -104.7647 inches — the Lab's -104.77, and the protocol's -2.661213207510891 m. Three independent paths, one answer.

Now change nothing but the step:

Invocation	Rows	First	Last
`--start 100 --step 50`	19	$100 : -0.1985$	$1000 : -2.9101$
`--start 100 --step 100`	10	$100 : -0.1985$	$1000 : -2.9101$
`--start 100 --step 125`	8	$100 : -0.3127$	$975 : -2.9847$
`--start 100 --step 150`	7	$100 : -0.4543$	$1000 : -3.2949$
`--start 100 --step 200`	5	$100 : 0.0000$	$900 : -3.2335$
`--start 100 --step 300`	4	$100 : 0.0000$	$1000 : -2.2370$

The 1000-yard cell reads -2.9101, -3.2949 or -2.2370 mil depending on the step size: 13% high or 23% low against the correct value, on the number you would dial.

`wind-card`: make every requested range a multiple of `--step`

The mechanism is an index misalignment. The card samples the trajectory at multiples of `--step` starting from zero, then looks each requested range up at the *nearest* sample — but labels the row, and does the inches-to-mil division, with the range you asked for. Verified against the correct table: with `--start 100 --step 200`, the row labelled 900 yd reports −3.2335 mil, which is −104.7647 linear inches — the drift at **1000** yards — divided by 900 yards. With `--step 300`, the row labelled 400 yd reports the drift at 300, the row labelled 700 reports the drift at 600, and the row labelled 1000 reports the drift at 900. The first row often reads exactly `0.0000` because it snaps to the muzzle.

The safe rule: choose a `--start` that is an exact multiple of `--step` (100/100, 200/200, 100/50). Then every requested range is a real sample and the card is right. Better still, build the card in the Lab, which interpolates the solved trajectory properly and cannot land off-grid.

The Lab's `:wind-card` does not have this problem because `analysis.lat` interpolates linearly between adjacent samples of the actual trajectory and asserts that requested distances lie inside the solved interval.

12.11 Hold corridors

A wind card answers "what if the wind is this?". The question a competitor actually faces is "the wind is doing *something* in that corridor — what one hold survives all of it?" The engine's `hold-corridor` subcommand answers that directly, and it does it over **named segmented-wind scenarios**, which is the richest use of wind segments anywhere in the toolchain.

You describe the scenarios in a small versioned JSON file:

Listing 12.15: A four-scenario corridor

```
{"version":1,"nominal":"steady","scenarios":[
  {"name":"steady",       "segments":["10:90:1000"]},
  {"name":"front-loaded", "segments":["14:90:400","6:90:1000"]},
  {"name":"back-loaded",  "segments":["6:90:400","14:90:1000"]},
  {"name":"switching",    "segments":["10:90:500","8:270:1000"]}
]}
```

Each scenario is the same average wind arranged differently: steady 10 mph, front-loaded, back-loaded, and one that reverses at the midpoint.

Listing 12.16: `hold-corridor` over those scenarios

```
$ ballistics hold-corridor -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --zero-distance 100 --scenarios scen.json --ranges 400,600,800,1000 \
    --target rect:20x20

Robust Hold Corridor
====================

Scenarios (4): back-loaded, front-loaded, steady, switching
Nominal:        steady
Target:         rectangle 20.0 x 20.0 in (per-axis / L-inf metric)

Holds are milliradians: elevation positive = hold UP, windage positive = hold RIGHT.
The corridor is the span of the scenarios you supplied. It is NOT a probability interval.
```

Listing 12.17: The 600-yard and 1000-yard blocks

```
--- 600 yd ---
  Scenario              Elev(mil)  Wind(mil)
  back-loaded               4.314     -1.035
  front-loaded              4.314     -1.844
  steady                    4.314     -1.440
  switching                 4.314     -1.335
  Corridor                  4.314     -1.844   (min)
                            4.314     -1.035   (max)
  Minimax hold              4.314     -1.440
  Worst case from it        0.405 mil (8.74 in), scenario 'back-loaded'
  Holding the nominal       0.405 mil (8.74 in)
  Fits target           yes — one hold covers every scenario

--- 1000 yd ---
  Scenario              Elev(mil)  Wind(mil)
  back-loaded              10.788     -2.876
  front-loaded             10.788     -2.944
  steady                   10.788     -2.910
  switching                10.788     -0.988
  Corridor                 10.788     -2.944   (min)
                           10.788     -0.988   (max)
  Minimax hold             10.788     -1.966
  Worst case from it        0.978 mil (35.22 in), scenario 'front-loaded'
  Holding the nominal       1.922 mil (69.20 in)
  Fits target           NO — no single hold covers every scenario here
```

There is a great deal packed into that output.

The elevation column never moves. 4.314 mil at 600, 10.788 at 1000, identical across all four scenarios. Rearranging a crosswind does not change your elevation, which is the same result Section 12.5 showed from the other direction.

Front-loaded and back-loaded bracket the steady case at 600 yards: −1.844 against −1.035, with steady at −1.440 in between. The same total wind energy, distributed differently, produces an 0.8-mil spread on the hold — and the front-loaded scenario needs *more* hold, exactly as the thirds table predicted.

The switching scenario is the outlier, and it is the one that breaks the corridor at 1000 yards: −0.988 mil against everyone else's −2.9. A wind that reverses does not average out, and no single hold covers both it and the non-reversing scenarios inside a 20-inch box.

The minimax hold is not the nominal hold. At 1000 yards the nominal (steady) hold of −2.910 leaves you 1.922 mil (69.20 in) wrong in the worst case, while the minimax hold of −1.966 — the midpoint of the corridor — caps the worst case at 0.978 mil (35.22 in). Half the exposure, at the cost of being wrong by a little in every scenario instead of being right in one.

Read the last line first

`Fits target` is the whole report compressed to a word. "yes" means one hold covers every wind you thought plausible and you can commit. "NO" means the corridor is wider than the target and you must either narrow your scenario set — i.e., read the wind better — or accept that this shot is a gamble.

A corridor is not a probability

The tool says so itself: *"The corridor is the span of the scenarios you supplied. It is NOT a probability interval."* It contains exactly the outcomes you thought to write into the JSON file. If every scenario in your file is a 90-degree wind, the corridor will not tell you what a 45-degree wind does. Garbage scenarios in, confident garbage out.

`hold-corridor` is CLI-only

The Lab has no equivalent. You could build one — solve each scenario as a separate experiment, keep the resulting trajectories, and take the min and max of `analysis.wind_card`'s `wind_hold` column across them — and that is a good exercise for the extension chapter. What the Lab gives you for free that `hold-corridor` does not is the provenance block: every scenario's engine version, solver, assumptions and warnings, attached to the table.

12.12 What the wind model does not do

- **No time.** Wind varies with downrange distance and (with shear) with height. It never varies with the clock. A gust that arrives during your shot is outside the model.
- **No turbulence.** Each segment is a steady, uniform flow.
- **No terrain.** The engine does not know there is a gully at 600 yards; you know, and you write a segment for it.
- **No mirage.** Reading mirage is how you *produce* the segment list, not something the engine does for you.
- **No coupling to the atmosphere zones** — which, as Section 11.6 noted, are not reachable from the CLI or the protocol anyway.

The engine's job is to turn a wind description into a firing solution exactly. Yours is to produce the description. The segmented interface exists so that when you can see the wind is doing three different things between you and the target, you have somewhere to say so — and so the wind call you made is recorded, with the run, for the next time the same range does the same thing.

12.13 Summary

- Wind enters the model in exactly one place: as a vector subtracted from the bullet's velocity before drag is computed.
- Direction is measured **from**, clockwise from the bore. Windage is positive to the shooter's right, and the wind hold that cancels it is therefore the sign flip. Drop runs the other way — positive **below** the sight line — so a come-up needs no flip at all.
- A 10 mph full-value crosswind is 104.77 inches (2.91 mil) at 1000 yards for this load, and drift is linear in wind speed to one part in ten thousand.
- The clock-face sine rule is good to about 1%, erring light into a headwind because head components lengthen the flight.
- A 10 mph head or tail wind changes elevation by about ± 4.3 inches at 1000 yards.
- Wind in the first third of the range does 43% of the work; the last third does 19%. Drifts from separate stretches add.
- Wind segments are a variable-length array — the exact feature that killed the C-ABI extension design and that the shipped subprocess bridge carries for free.
- A 2 m/s updraft is worth 1.21 mil at 1000 yards. Do not guess one.

- The engine's `wind-card` subcommand is only correct when every requested range is a multiple of `--step`. The Lab's `:wind-card` interpolates properly and has no such constraint.

Chapter 13

Zeroing and DOPE

Everything in the previous three chapters was about the air the bullet flies through. This chapter is about the rifle it leaves. A zero is the one number that ties a computed trajectory to a physical scope, and DOPE is the table you carry to the line. Get either of them wrong and every atmospheric refinement in the atmosphere chapter is decoration on a miss.

We are going to be unusually careful about two things: *where drop is measured from*, and *what height the zero was solved to*. Those are different questions, they are easy to confuse, and between them they decide every number on a dope card. The engine reports three different vertical references depending on which surface you ask, and it will happily zero your rifle onto the dirt if you forget to tell it where the target is. This chapter settles both.

13.1 What a zero actually is

Sight height, bore line, line of sight

The **bore line** is the axis of the barrel, extended forward forever. The **line of sight** (LOS) is the line from your eye through the scope to the aim point. **Sight height** is the perpendicular distance between the optic's optical axis and the bore axis at the muzzle — typically 1.5 to 2.0 inches on a scoped bolt rifle, and considerably more on an AR-pattern rifle with a raised mount.

A bullet leaving the muzzle starts below the line of sight by exactly the sight height, and begins falling immediately. To make it arrive at the aim point at some chosen distance, the barrel must be tilted *up* relative to the line of sight. That tilt is the **zero angle**.

Two consequences follow, and both matter in the field.

First, because the bullet starts below the line of sight, rises above it, and then falls back through it, a given zero angle produces *two* crossings — a near zero on the ascending branch and a far zero on the descending branch. The familiar 25/300 battle zero is not two settings; it is one bore angle with two crossings.

Second, the zero angle is a property of the *rifle plus the conditions it was zeroed in*, not of the trajectory you are about to shoot. It is the portable quantity. Hornady and Kestrel's 4DOF both treat it that way, and so does this engine: capture the angle once, and you can recover what ranges it implies later under any conditions you like.

mil and MOA

A **milliradian** (mil, mrad) is an angle of 10^{-3} radians. At 1000 yards — 36,000 inches — one mil subtends exactly 36 inches. A **minute of angle** (MOA) is $1/60$ of a degree, which subtends 1.047 inches at 100 yards. The `ballistics-engine`'s printed dial tables deliberately convert with the round constant 3438 rather than the exact 3437.7467, so a mil value multiplied by 3.438 reproduces the engine's MOA column exactly.

13.2 The `zero` subcommand

The `ballistics-engine` solves the zero angle with a dedicated subcommand. Here is our working rifle for this chapter — a 140 gr 6.5 mm match bullet, G7 BC 0.315, 2710 ft/s, in a 1:8 twist with the scope 1.8 inches over the bore.

Listing 13.1: Solving a 100-yard zero

```
$ ballistics zero -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --target-distance 100 --sight-height 1.8 -o json
{
  "max_ordinate": 0.05142794927511805,
  "point_blank_range": 173.77136676522406,
  "sight_adjustment_moa": 4.065328593991589,
  "units": "imperial",
  "zero_angle_degrees": 0.06775547656652649,
  "zero_angle_moa": 4.065328593991589,
  "zero_angle_mrad": 1.18255615234375
}
```

The bore is tilted 0.0678 degrees — 1.18 mil, 4.07 MOA — above the line of sight. That is the number the scope's internal erector has to absorb when you "zero the rifle": your turrets are not moving the barrel, they are moving the reticle down until it points where the barrel already shoots.

The default output is a drawn box

Without `-o json` the command prints the same numbers inside a Unicode-boxed table. Every engine table in this book is shown in its `json`, `csv`, or plain-text form instead, because those are the forms that survive being typeset. The numbers are identical; the JSON carries more digits.

Two of the fields deserve names. `sight_adjustment_moa` is the same quantity as `zero_angle_moa`; it is repeated under a shooter-facing name. `point_blank_range` is a convenience figure that grows with sight height — and, oddly, comes out identical to seven figures for a 1.5-inch and a 1.8-inch scope in the sweep below. Use the `mpbr` subcommand (Section 13.9) when you need point-blank range against a stated vital zone; it is the command built for the question.

13.2.1 Sight height changes the zero angle

Raising the optic raises the whole geometry, so the barrel needs more tilt to put the bullet on the aim point at the same distance. Sweeping the sight height with everything else fixed:

Listing 13.2: Zero angle versus sight height, 100 yd zero — two fields lifted from five JSON runs

```
$ for sh in 0.9 1.5 1.8 2.6 3.5; do
>   ballistics zero -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
>       --target-distance 100 --sight-height $sh -o json
> done

sight=0.9   zero_angle_degrees 0.053417543460835715  point_blank_range 165.54906970489765
sight=1.5   zero_angle_degrees 0.06294702339083752   point_blank_range 173.77137937440145
sight=1.8   zero_angle_degrees 0.06775547656652649   point_blank_range 173.77136676522406
sight=2.6   zero_angle_degrees 0.08043230766607017   point_blank_range 190.1017772731181
sight=3.5   zero_angle_degrees 0.09477024077176092   point_blank_range 206.2817280246083
```

An AR-height mount at 2.6 inches needs 19% more bore tilt than a bolt-gun mount at 1.8 inches for the same 100-yard zero, and its point-blank range is 16 yards longer. Sight height is not a cosmetic input.

`zero` echoes sight height in yards

The table form of `zero` prints `Sight Height: 0.050 yd` for the 1.8-inch scope we entered. The flag takes inches (imperial) or millimetres (metric); the echo is rendered in the command's distance unit, which for a vertical offset of a couple of inches reads as an unhelpful `0.050`. Check what you typed, not what it echoes.

13.3 One angle, two zeros

Given a bore angle, `zero -from-angle` reports both crossings it finds. Feeding back the exact angle solved above:

Listing 13.3: Recovering both zeros from a stored bore angle

```
$ ballistics zero -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --from-angle 0.06775547656652649 --sight-height 1.8

    ZERO RANGE FROM STORED ANGLE
 Input Zero Angle:    0.0678 deg
 Input Zero Angle:      4.07 MOA
 Input Zero Angle:      1.18 mrad
 Near Zero (ascending):     71.4 yd
 Far Zero (descending):    100.0 yd
 Target Height:         0.00 yd
 Sight Height:         0.050 yd
 Max Ordinate:         0.051 yd
```

(The engine draws that report inside a box; the field names and values above are its contents verbatim, with the box rules and the degree sign removed so the page can hold them.)

The far zero comes back as 100.0 yards, which is a useful self-check on the forward solve. The near zero is 71.4 yards — with a 100-yard zero the bullet is only above the line of sight between 71 and 100 yards, and by less than a tenth of an inch.

Tilt the barrel further and the two crossings spread apart dramatically:

Listing 13.4: A larger bore angle gives a classic near/far pair

```
$ ballistics zero -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --from-angle 0.0921 --sight-height 1.8

 Input Zero Angle:    0.0921 deg
 Input Zero Angle:      5.53 MOA
 Input Zero Angle:      1.61 mrad
 Near Zero (ascending):     36.7 yd
 Far Zero (descending):    190.6 yd
 Max Ordinate:         0.094 yd
```

That is the battle-zero relationship in miniature: one angle, a near zero at 37 yards and a far zero at 191, with a maximum ordinate of 0.094 yd on the `zero` command's own datum — which, as Section 13.4 will show, sits one sight height above the line of sight.

Store the angle, not the distance

`-from-angle` is not a single-valued inverse of `-target-distance`. The forward solve picks whichever root reaches the distance you asked for; the reverse solve reports both. If you record one number about your rifle's zero, record `zero_angle_degrees`. It reproduces under any atmosphere; "100 yards" does not, because the angle that zeroes at 100 yards at sea level is not the angle that zeroes there at 6000 feet.

13.4 Where drop is measured from, and what the zero was solved to

Set the rifle up in the Ballistics Lab, zero it at 100 yards, solve, and ask what the bullet is doing at 100 and 300 yards.

Listing 13.5: A 100-yard zero, queried at 100 and 300 yards

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140), inches(0.264), 0.315, "G7",
    inches(1.34)))
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in", inches(1.8), m(0), inches(8), "right"))
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(15), hpa(1013.25), 0.5, deg(44)))
sessions.replace_winds(lab, [])
sessions.replace_shot(lab, shot(fps(2710), yd(600), yd(100)))
sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), nil, false, nil))
:solve
:at 100 yd
:at 300 yd

Observation
  distance: 100.00 yd
  time    : 0.11 s
  velocity: 2565.10 ft/s
  energy  : 2045.04 ft-lb
  drop    : +0.00 in
  windage : +0.00 in
  Mach    : 2.29
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 300.00 yd
  time    : 0.36 s
  velocity: 2288.38 ft/s
  energy  : 1627.62 ft-lb
  drop    : +12.89 in
  windage : +0.00 in
  Mach    : 2.05
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

At the zero distance the bullet is on the line of sight and the drop is `+0.00 in`, which is what a zero means. Now move the zero to 300 yards and ask the same two questions:

Listing 13.6: The same rifle zeroed at 300 yards (other observation fields elided)

```
sessions.replace_shot(lab, shot(fps(2710), yd(600), yd(300)))
:solve
:at 100 yd
:at 300 yd

Observation
  distance: 100.00 yd
  drop    : -4.30 in
Observation
  distance: 300.00 yd
  drop    : +0.00 in
```

The pattern is the right one. Drop reads `+0.00 in` at whatever distance you zeroed, wherever you put the zero. And with the zero pushed out to 300 yards, the 100-yard row goes *negative* — `-4.30 in` — because between the near and far crossings the bullet is above the line of sight, and drop is positive-below.

The Ballistics Lab's three sign conventions

Every table the Ballistics Lab prints ends with a legend naming the conventions its columns use. There are only three in the whole tool.

- **Drop is positive below** the line of sight. Above reads negative.
- **Windage is positive right** of the aim point, from behind the rifle.
- **Come-up is positive when the sight must be raised.**

Drop and come-up therefore share a sign: a bullet that is low needs the sight brought up. Windage runs the other way, so a wind hold is the negative of the windage that produced it. That asymmetry is not a wart; it is what makes each column read the way a shooter would say it out loud.

13.4.1 Why the zero lands on the line of sight

None of this is automatic. It depends on one field in the wire request that has nothing to do with drop and everything to do with the bore angle: `shot.target_height_m`.

target_height_m

`shot.target_height_m` is the height *above the local ground datum* of the point the engine's elevation search aims at. The engine's own documentation calls it "meters above ground for zeroing" and defaults it to `0.0` — "target at ground level by default" (`src/cli_api.rs:181` and `:502`).

Left at that default, the engine finds the bore angle that puts the bullet on the *ground* at the zero distance. Set to the height of the line of sight — muzzle height plus sight height — it finds the angle that puts the bullet on the *aim point*, which is the zero a shooter actually shot.

The Ballistics Lab sends the second of those. `protocol_v1.lat`'s `shot_to_v1_map` uses `shot.target_height` when the experiment names one, and otherwise fills the field in from the rifle as muzzle height plus sight height. The rifle above sits on the ground with a 1.8-inch scope, so the resolved request comes back with:

Listing 13.7: `json_stringify(run.resolved_request.get("shot"))`, reflowed to fit

```
{"cant_angle_rad":0,"muzzle_angle_rad":0.00118255615234375,
 "max_range_m":548.6399999999999,"zero_distance_m":91.439999999999998,
 "aim_azimuth_rad":0,"shooting_angle_rad":0,"ground_threshold_m":-100,
 "shot_azimuth_rad":0,"target_height_m":0.045719999999999997}
```

`target_height_m` is 0.04572 m, which is 1.8 inches, which is exactly `sight_height_m` for this rifle — the muzzle height being zero. And `muzzle_angle_rad` is 0.00118255615234375: the same 1.18255615234375 mrad the `zero` subcommand reported for this rifle back in Section 13.2, to every digit. Two independent code paths in the engine, asked the same geometric question, return the same bore angle.

Driving `solve-json` by hand? Send the field

If you build v1 requests yourself rather than through the Ballistics Lab, omitting `target_height_m` is a silent 1.8-inch-per-100-yards error, not a crash. The engine tells you it happened — the response carries `default_applied: Target height defaulted to 0 m above the local ground datum` in its `assumptions` array — but it tells you in the one place people skim. Reading the assumption list is the check.

Here is the field's effect, isolated. The same 91.44 m (100 yd) zero and 0.04572 m (1.8 in) sight height, run three ways against the engine directly:

Listing 13.8: Isolating the zero datum — distance and drop lifted from three solve-json responses

```
# target_height_m omitted -> engine default 0.0; zeroed to the ground
#   resolved muzzle_angle_rad = 0.0006835937500000001
  distance_m     drop_m
        0.00  +0.045720000
       91.44  +0.045625602
      182.88  +0.179773652
      274.32  +0.464194403

# target_height_m = 0.04572 -> zeroed to the line of sight (what the Lab sends)
#   resolved muzzle_angle_rad = 0.00118255615234375
        0.00  +0.045720000
       91.44  +0.000000497
      182.88  +0.088523528
      274.32  +0.327319229

# sight_height_m = 0.0 and target_height_m = 0.0 -> optic on the bore axis
#   resolved muzzle_angle_rad = 0.0006835937500000001
        0.00  +0.000000000
       91.44  -0.000094398
      182.88  +0.134053652
      274.32  +0.418474403
```

Read the three blocks as one argument. The first and third share a bore angle, because both zero to the ground; subtract them row by row and the difference is 0.04572 m at every single distance — the sight height exactly, and nothing else. Moving the optic moves the datum; it does not move the bullet.

The second block is the one with a *different* bore angle: 0.00118 rad against 0.00068, because the target was raised to the optic's own height. That is the change that matters. Only there do the optic and the aim point sit at the same height, so only there is the line of sight horizontal and the drop column measured from the line the shooter is actually looking along. The third block is the degenerate case that proves the point — drop the optic onto the bore axis at ground level and the two datums coincide again, which is why it reads zero both at the muzzle and at the zero distance.

At 274.32 m the gap between the first and second blocks is $0.464194 - 0.327319 = 0.136875$ m. That is sight height $\times\, d/d_{\text{zero}}$: $0.04572 \times 3 = 0.13716$ m, to within the millimetre that the two slightly different flight paths account for. The error a missing target height produces is not a constant offset in inches; it is a constant *angle*, which is why it would show up on a dope card as the same number of mils at every range.

The three vertical references, and which surface uses which

- **`solve-json` `drop_m`** — metres below the *horizontal plane through the optic*. This is what the Ballistics Lab shows in `:at` and `:dope`. For a level shot at a target set at the height of the line of sight — which is what the Ballistics Lab asks for — that plane *is* the line of sight, so the column reads 0 at the zero.
- **The CLI `range-table` / `come-ups` drop columns** — inches (or mils) below the *line of sight*. These also read 0.0 at the zero distance.
- **The native `-o json` `y` axis** — height above the *ground* in the world frame, in yards or metres.

Because the Ballistics Lab and the CLI table now answer the same question, they can be checked against each other row by row — which is the strongest thing you can say about two independent surfaces on the same engine:

Listing 13.9: The CLI range table

```
$ ballistics range-table -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --sight-height 1.8 --zero-distance 100 --start 100 --end 600 --step 100 -o csv
range_yd,drop_in,drop_mil,wind_in,wind_mil,velocity_fps,energy_ft-lb,tof_s
100,0.0,0.000,-0.5,-0.15,2565,2045,0.11
200,3.5,0.484,-2.2,-0.31,2425,1827,0.23
300,12.9,1.193,-5.2,-0.48,2288,1628,0.36
400,28.9,2.010,-9.4,-0.66,2157,1446,0.50
500,52.5,2.915,-15.2,-0.84,2030,1281,0.64
600,84.4,3.909,-22.6,-1.04,1907,1131,0.79
```

Listing 13.10: `:dope 100 600 100` yd on the same rifle in the Ballistics Lab

```
DOPE: 140 gr ELD-M at 600 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2565.10 |        2045.04 |     0.11 | 2.29
       200.00 |     +3.49 |         +0.48 |         2424.51 |        1827.01 |     0.23 | 2.17
       300.00 |    +12.89 |         +1.19 |         2288.38 |        1627.62 |     0.36 | 2.05
       400.00 |    +28.94 |         +2.01 |         2156.91 |        1445.97 |     0.50 | 1.93
       500.00 |    +52.47 |         +2.91 |         2029.99 |        1280.80 |     0.64 | 1.82
       600.00 |    +84.44 |         +3.91 |         1907.15 |        1130.49 |     0.79 | 1.71
Signs: drop + below/- above; come-up + correction up/- correction down
```

Column against column: 0.0/3.5/12.9/28.9/52.5/84.4 inches from the CLI against +0.00/ + 3.49/ + 12.89/ + 28.94/ + 52.47/ + 84.44 from the Ballistics Lab, and 0.000/0.484/1.193/2.010/2.915/3.909 mil against +0.00/ + 0.48/ + 1.19/ + 2.01/ + 2.91/ + 3.91. The CLI rounds to one decimal and one more mil digit than the Ballistics Lab prints; underneath, they are the same numbers. The `wind_in` and `wind_mil` columns have no counterpart in the comparison: `range-table` accepts no wind or twist arguments at all — its whole usage line is velocity, BC, mass, diameter, drag model, sight height, zero distance, start, end and step — while the Ballistics Lab's `windage` for this calm run is `+0.00 in`. Compare the elevation columns and leave those two alone.

The one place the two *do* still differ is the *maximum ordinate*. For the same load and the same 265.19-yard zero:

Listing 13.11: Max ordinate differs by exactly the sight height — fields lifted from four JSON runs

```
sight=1.8  optimal_zero=265.185546875
           mpbr max ordinate = 3.994165672488917 in   (above the line of sight)
           zero max ordinate = 5.794197431600744 in
           difference = 1.800031759111827

sight=3.0  optimal_zero=278.515625
           mpbr max ordinate = 4.009836086540627 in
           zero max ordinate = 7.010062302003432 in
           difference = 3.0002262154628045
```

`mpbr` reports the ordinate a shooter cares about — how high above the crosshair the bullet flies. `zero` reports the bullet's greatest height above the *muzzle* datum, which sits one sight height lower, so its figure is larger by exactly that. The same quantity comes back from `solve-json` as `summary.maximum_height_m`: fed the 265.19-yard zero geometry with `target_height_m` at the sight height, the protocol returns 0.14717228594091725 m, which is 5.794184 inches against `zero`'s 5.794197. Three surfaces, one bullet, two datums — and no mystery once you know which is which.

13.5 Come-ups: the dial table

DOPE and come-up

DOPE — "data on previous engagements" — is the table of turret settings a particular rifle and load need at a series of distances. A **come-up** is an elevation correction dialled *up* from the zero. The engine's `come-ups` subcommand prints them.

Listing 13.12: A mil come-up table, 100 yd zero

```
$ ballistics come-ups -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --start 100 --end 1000 --step 100 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,0.000,0.000,2565,2045,0.114
200,0.484,0.484,2425,1827,0.234
300,1.193,0.709,2288,1628,0.361
400,2.010,0.816,2157,1446,0.496
500,2.915,0.905,2030,1281,0.640
600,3.909,0.994,1907,1131,0.792
700,4.999,1.090,1788,994,0.955
800,6.195,1.196,1672,869,1.128
900,7.510,1.314,1560,756,1.314
1000,8.959,1.449,1451,654,1.514
```

The two columns mean different things

`drop_mil` is the *total* dial from the zero: at 1000 yards you come up 8.959 mil from your 100-yard setting. `come_up_mil` is the *incremental* step from the previous row: 1.449 mil more than the 900-yard setting. The first row has no predecessor, so the boxed table form leaves it as an em dash and the CSV form repeats the total.

Dial the `drop` column. Read the `come_up` column when you are thinking about how fast the curve is steepening — notice that it grows from 0.484 to 1.449 mil per hundred yards across this band, which is why long-range misses grow faster than the distance does.

13.5.1 Choosing the dial unit

`--adjustment-unit` is orthogonal to `--units`; it selects what the elevation column is expressed in.

Listing 13.13: The same solve in MOA and in clicks

```
$ ballistics come-ups ... --adjustment-unit moa -o csv
range_yd,drop_moa,come_up_moa,velocity_fps,energy_ft-lb,time_s
200,1.664,1.664,2425,1827,0.234
400,6.909,5.245,2157,1446,0.496
600,13.440,6.530,1907,1131,0.792
800,21.299,7.859,1672,869,1.128

$ ballistics come-ups ... --adjustment-unit clicks --elevation-click-value 0.1mil -o csv
range_yd,drop_clicks,come_up_clicks,velocity_fps,energy_ft-lb,time_s
200,5,5,2425,1827,0.234
400,20,15,2157,1446,0.496
600,39,19,1907,1131,0.792
800,62,23,1672,869,1.128
```

The MOA column is the mil column times 3.438 — $6.195 \times 3.438 = 21.30$ at 800 yards — and the clicks column is the mil column divided by the graduation you declared. If your turret is 0.1 mil per click, dialling 62 clicks at 800 yards puts you within 0.005 mil of the solution.

13.5.2 Scope tracking correction

Real turrets do not always move what they claim. A tall-target test measures the error, and the `ballistics-engine` turns it into a correction factor:

Listing 13.14: Deriving and applying a tracking correction factor

```
$ ballistics tall-target --dialed 10 --measured 36.2 --range 100 --unit mil

Tall-Target Test Result
  Dialed travel:   10.00 MIL
  Actual travel:   10.06 MIL (36.20 in at 100 yd)
  Correction factor (actual / dialed): 1.0056
  Enter this as --elevation-cf / profile elevation_cf (or the windage equivalents); dial
     solutions are divided by it.

$ ballistics come-ups ... --elevation-cf 1.0056 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
200,0.481,0.481,2425,1827,0.234
400,1.998,1.517,2157,1446,0.496
600,3.887,1.889,1907,1131,0.792
800,6.161,2.273,1672,869,1.128
```

$6.195 / 1.0056 = 6.161$: the scope over-travels by 0.56%, so the engine asks for 0.56% less dial. Note that `tall-target` runs no trajectory at all — it is measured travel divided by dialled travel, and nothing else. The factor is bounded to $(0.5, 1.5)$; a value outside that band means the test was misread, not that the scope is exotic.

13.6 Where a card's rows come from

A card subcommand does not integrate a separate trajectory per row. It solves the flight once, samples it on a grid of the engine's own choosing, and then reads each requested range off that grid. Two things about that grid are worth knowing, because between them they are the whole answer to "can I trust this row?"

The spacing is a constant of the engine — `hold_curve::CARD_SAMPLE_INTERVAL_M`, 0.9144 m, which is exactly one yard — and it has nothing whatever to do with `--step`. It is anchored at the muzzle, it is the same grid the reticle-hold machinery reads its holds off, and refining it costs one interpolation pass over knots the solver has already produced rather than another integration, which is why the engine can afford to make it fine and fixed instead of letting your request set it.

A requested range is then read *at* that range, by linear interpolation between the two samples bracketing it. Nothing is snapped to a neighbour, and the angular columns divide by the distance the row is labelled with rather than by some nearby sample's. The consequence is the property you actually want out of a card: **a given range produces the same row on every card that contains it**, whatever `--start`, `--end` and `--step` you reached it through.

That is cheap to check, and worth checking once so you stop wondering:

Listing 13.15: 700 yards, asked for three different ways

```
$ ballistics range-table ... --start 700 --end 1200 --step 500 -o csv
range_yd,drop_in,drop_mil,wind_in,wind_mil,velocity_fps,energy_ft-lb,tof_s
700,126.0,4.999,-31.7,-1.26,1788,994,0.95
1200,533.3,12.345,-111.2,-2.57,1244,481,1.96

$ ballistics range-table ... --start 700 --end 700 --step 100 -o csv
range_yd,drop_in,drop_mil,wind_in,wind_mil,velocity_fps,energy_ft-lb,tof_s
700,126.0,4.999,-31.7,-1.26,1788,994,0.95

$ ballistics come-ups ... --start 700 --end 1200 --step 250 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
700,4.999,4.999,1788,994,0.955
950,8.216,3.217,1505,704,1.412
1200,12.345,4.129,1244,481,1.960
```

700 yards is 4.999 mil on all three cards and 1200 yards is 12.345 on both of the two that reach it, even though no two of those commands share a `--step` and only one has a `--start` that is a multiple of its own step. (The wind columns are not zero because `range-table` applies a default 10 mph wind from 90° unless told otherwise; `come-ups` has no wind column at all.)

True from engine 0.37.0 — and not true before it

Through 0.36.x the sampling grid's spacing *was* `--step`, anchored at zero, and each row was filled from whichever sample lay nearest its label, with anything inside one and a half whole steps accepted. A card whose `--start` was not a multiple of its `--step` therefore had every one of its rows sitting between two grid points, and quietly got a neighbouring range's numbers printed under its own label.

On the load above, the 0.32.0 this book was first verified against answered `--start 700 --end 1200 --step 500` with the 500- and 1000-yard rows carrying 700- and 1200-yard labels: 8.959 mil against the 1200-yard line, where the truth for this load is 12.345. A shooter who dialled it would be 3.386 mil low — 146 inches, better than twelve feet.

If you are on an older binary — `ballistics --version` settles it — the workaround is the one the first edition of this book taught: make `--start` a whole multiple of `--step`, and spot-check a row against a single-range card (`--start R --end R`), which was computed at the range you asked for even then. From 0.37.0 there is nothing left to work around, and no reason to shape a card's band around the engine rather than around the ranges you shoot.

13.7 When the card outruns the load

The other half of "read the row at the range asked for" is what happens when the range asked for is past where the bullet ever got. `range-table` defaults `--end` to 1200 yards, and plenty of loads — a .22 LR at 1050 ft/s, a 9 mm, a .45 ACP — never see it. The engine has nothing honest to put on those lines: it cannot interpolate past the end of the flight, and filling them from the last range the bullet *did* reach is exactly the substitution Section 13.6 describes the engine having stopped doing.

Since 0.37.0 it prints the rows it has and tells you about the ones it does not:

Listing 13.16: A rimfire asked for 1200 yards of card

```
$ ballistics come-ups -v 1050 -b 0.125 -m 40 -d 0.224 --drag-model g1 \
      --zero-distance 50 --sight-height 1.5 --start 100 --end 1200 --step 100 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,2.177,2.177,900,72,0.311
200,7.930,5.753,800,57,0.665
300,14.887,6.957,720,46,1.061
400,23.046,8.159,650,38,1.500
500,32.538,9.492,589,31,1.987
600,43.558,11.020,533,25,2.525
700,56.364,12.805,483,21,3.122
800,71.284,14.921,438,17,3.785
900,88.740,17.455,397,14,4.524
1000,109.264,20.525,362,12,5.350
warning: card truncated at 1000 yd: 1200 yd was requested, but this load's solved flight reaches
     only 1000 yd; the rows past that are left out rather than filled in from a range the bullet
     does reach
```

Ten real rows, two absent ones, and a sentence saying which is which. The rows that survive a truncation are byte-identical to the same card asked for with `--end 1000` in the first place — truncating changes what is printed, never what a printed row says.

13.7.1 Where the notice goes

The warning is on *stderr*, which is why it appears after the CSV above and why redirecting stdout to a file leaves the file clean. That is deliberate: `-o csv` and `-o json` are machine surfaces, and a comment line injected into a CSV breaks the reader it was meant to warn.

A program therefore reads the truncation structurally instead. `-o json` grows a `truncated` object, which is simply absent from an untruncated card:

Listing 13.17: The same truncation, in JSON

```
  "truncated": {
    "last_row": 1000.0,
    "message": "card truncated at 1000 yd: 1200 yd was requested, but this load's solved flight
     reaches only 1000 yd; the rows past that are left out rather than filled in from a range
     the bullet does reach",
    "reach": 1000.4173004455556,
    "requested_end": 1200.0
  },
```

`last_row` is the furthest line actually printed; `reach` is the solved flight's own terminal distance, a property of the load rather than of your request, so it does not move when you change `--end`. The two differ here by four tenths of a yard — 1000 is the last requested row the flight covers, 1000.42 is where the flight ended — and the distinction is not pedantry. A reach read off the sampling grid rather than off the flight would move whenever you moved `--end`, since the span sampled is derived from it; a number offered as a fact about the load must not depend on the question you asked. Ask this load for `--end 1100`, 1200 or 2000 and the reach comes back 1000.4173004455556 every time. Only `requested_end` changes, which is the field whose job that is.

13.7.2 All five card surfaces, and the one case still refused

The contract is the same on `come-ups`, `range-table`, `wind-card`, `compare` and `adaptive-card`. It reaches paper as well: `adaptive-card -o pdf` prints a shortened form of the same notice into the document's own footer, beside the engine and table provenance, on every page —

Listing 13.18: The footer line of a truncated printed card

```
TRUNCATED at 1000 yd: 1200 yd requested, this load reaches only 1000 yd
```

— which matters more than any of the machine-readable forms, because the printed card is the one most likely to be read at a range with no screen beside it.

`wind-card` and `compare` put several flights on one range axis, so they solve all of them and trim the axis to the shortest column. `compare` names the load that set the limit, which is the only thing you want to know at that moment:

Listing 13.19: `compare` names the load that shortened the card

```
$ ballistics compare --load "match:g7:0.315:140:2710:0.264" \
      --load "rimfire:g1:0.125:40:1050:0.224" \
      --zero-distance 100 --start 100 --end 1200 --step 200 -o csv
...
900,245.73,7.58,-55.93,-1.73,1560,756,1.314,2804.34,86.55,-343.57,-10.60,397,14,4.523
warning: load 'rimfire': card truncated at 900 yd: 1200 yd was requested, but this load's solved
     flight reaches only 1007 yd; the rows past that are left out rather than filled in from a
     range the bullet does reach
```

The reach quoted there is 1007 yards rather than the 1000 of the card above because it is not the same flight: this rimfire is zeroed at 100 yards here instead of 50, and the extra bore elevation buys it about six and a half yards more. Reach is a property of the flight, and the zero is part of the flight — which is worth remembering before concluding that two cards of "the same load" disagree.

A truncated card still exits 0: it is a complete answer to a slightly smaller question. What is still an error is a card with *no* reachable row at all, because then there is no card to print:

Listing 13.20: Nothing to truncate to

```
$ ballistics come-ups ... --start 1100 --end 1200 --step 100 -o csv
Error: "no trajectory sample at 1100 yd: this load's solved flight reaches only 1000 yd"
$ echo $?
1
```

The Ballistics Lab and the `ballistics-engine` stop short differently

Both refuse to invent a row past the end of the flight, but they say so in different places and they do not stop in the same place. `:dope` clips your requested stop back to the trajectory and *appends the true terminal row*, off-grid, so its last line is where the bullet actually finished (Section 6.6 in the DOPE chapter). An engine card stops at the last *requested* row the flight covers and leaves the terminal distance in the notice instead — 1000 yards printed, 1000.42 reported, in the transcript above.

So a Lab card and a CLI card of the same short-falling load will disagree about their final line by up to one row spacing, and neither is wrong. If you are reconciling the two, compare `reach` against the Lab's terminal row, not the two last printed distances.

13.8 DOPE inside the Ballistics Lab

The Ballistics Lab's `:dope` command builds the same kind of table from the current run, and adds something the engine's CLI does not: a provenance block naming the engine version, the solver, the verification state, and every assumption the engine applied.

Listing 13.21: :dope over a realistic band

```
:solve
:dope 100 1000 100 yd

DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       300.00 |    +12.58 |         +1.16 |         2338.37 |        1699.50 |     0.36 | 2.10
       400.00 |    +28.10 |         +1.95 |         2221.35 |        1533.66 |     0.49 | 2.00
       500.00 |    +50.68 |         +2.82 |         2107.93 |        1381.04 |     0.63 | 1.90
       600.00 |    +81.08 |         +3.75 |         1997.90 |        1240.63 |     0.77 | 1.80
       700.00 |   +120.20 |         +4.77 |         1890.96 |        1111.37 |     0.93 | 1.70
       800.00 |   +169.05 |         +5.87 |         1786.81 |         992.32 |     1.09 | 1.61
       900.00 |   +228.81 |         +7.06 |         1685.30 |         882.78 |     1.26 | 1.52
      1000.00 |   +300.82 |         +8.36 |         1586.32 |         782.12 |     1.45 | 1.43
Signs: drop + below/- above; come-up + correction up/- correction down
Provenance
  [0] 140 gr ELD-M at 1,200 yd
    engine: 0.32.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
```

(That table is from the reference session of Chapter 6: a 140 gr ELD-M at 1200 m altitude with Coriolis enabled, which is why its numbers differ from the sea-level engine tables earlier in this chapter. Never cross-quote two tables computed under different conditions.)

Read it the way you would use it. The zero row is `+0.00` in both elevation columns, because at 100 yards this rifle is already on. Past the zero the drop grows and the come-up grows with it, and at 1000 yards the card says dial 8.36 mil up — a total from the zero, not an increment from the 900-yard row. Between the two, the Drop column is what the bullet did and the Come-up column is what you do about it; they carry the same sign because low and up are the same direction.

Two things the Come-up column is not

It is **not an increment**. The engine's `come-ups` subcommand prints both a total (`drop_mil`) and an increment (`come_up_mil`); the Ballistics Lab prints only the total, so compare a Ballistics Lab Come-up against the engine's `drop_mil` column, never against its `come_up_mil` column. Section 13.5 shows the two side by side.

It is also **not a hold**. It is the correction, expressed as an angle. Dial it or hold it as you prefer, but if you hold, you hold the reticle mark that many mils *above* the target, which is the same correction applied at the other end of the optic.

The provenance block is the reason to build DOPE in the Ballistics Lab rather than from a bare CLI invocation. A printed dope card with no record of which engine version, which solver, and which defaults produced it is a card you cannot audit six months later. The Ballistics Lab's tables carry that record in the same artifact as the numbers — and Chapter 24 shows how to freeze one into a portable document.

13.9 Maximum point-blank range

MPBR

Listing 13.22: MPBR for an 8-inch vital zone

```
$ ballistics mpbr -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --sight-height 1.8 --vital-zone 8 -o json
{
  "far_zero": 265.16755702333853,
  "impact_energy": 1603.0402197729,
  "impact_energy_unit": "ft-lb",
  "impact_velocity": 2271.0417060801997,
  "impact_velocity_unit": "fps",
  "max_ordinate": 3.994165672488917,
  "max_ordinate_distance": 149.0,
  "max_ordinate_unit": "in",
  "mpbr": 312.21559615196384,
  "mpbr_unit": "yd",
  "near_zero": 25.78136209186501,
  "optimal_zero": 265.185546875,
  "vital_zone": 8.0,
  "vital_zone_unit": "in"
}
```

Read it as a set of instructions: zero at 265 yards, and from the muzzle out to 312 yards you can hold on the middle of an 8-inch vital zone and hit it. The maximum ordinate is 3.994 inches at 149 yards — half the vital zone, which is the constraint that determines the answer. The near zero at 25.8 yards is where the rising bullet first crosses the line of sight.

Halve the vital zone and the whole solution shrinks:

Listing 13.23: A 4-inch vital zone

```
  "far_zero": 206.16884708486253,
  "max_ordinate": 2.0072598487621556,
  "max_ordinate_distance": 124.0,
  "mpbr": 240.3406866203004,
  "near_zero": 33.854966162414215,
  "optimal_zero": 206.15234375000003,
  "vital_zone": 4.0
```

MPBR drops from 312 to 240 yards. Precision costs reach, and it costs it faster than linearly.

13.9.1 Checking MPBR against `zero`

The two subcommands are independent code paths, so agreeing is meaningful. Take MPBR's optimal zero, solve the bore angle for it, then ask what crossings that angle implies:

Listing 13.24: An MPBR cross-check

```
$ ballistics mpbr ... -o json | grep optimal_zero
  "optimal_zero": 265.185546875,

$ ballistics zero ... --target-distance 265.185546875 -o json
  "zero_angle_degrees": 0.12117302002736223,
  "max_ordinate": 0.16094992865557622,

$ ballistics zero ... --from-angle 0.12117302002736223
 Near Zero (ascending):     25.8 yd
 Far Zero (descending):    265.2 yd
 Max Ordinate:         0.161 yd
```

Near 25.8 versus MPBR's 25.78; far 265.2 versus MPBR's 265.17. The two solvers agree to the printed precision. The max ordinates differ by exactly the sight height, for the reason Section 13.4 gave.

13.10 Zeroing for a stage, not a distance

MPBR asks for one hold across a continuous band. A practical rifle match asks something different: several targets at known, discrete ranges, and a preference for as little dialling as possible. `optimal-zero` solves the minimax version — the single zero that minimises the largest hold anyone has to take.

Listing 13.25: Four steel targets, one zero

```
$ ballistics optimal-zero -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --sight-height 1.8 --target 200 --target 400 --target 600 --target 800 --vital 10

Optimal Zero (min-max hold)
===========================

Optimal zero:  543.8 yd
Largest hold:  2.856 mil

Range(yd)  Target(in)  Hold(mil)  Center-hold miss(in)  Fits
---------  ----------  ---------  --------------------  ----
      200        10.0     -2.855                -20.56  NO
      400        10.0     -1.330                -19.15  NO
      600        10.0      0.570                 12.31  NO
      800        10.0      2.856                 82.24  NO

At least one target needs a hold: no single zero keeps them all centered.
Center-hold miss is signed: positive = the bullet lands LOW of the aim point.
```

The honest answer here is "you cannot hold centre on all four". A 543.8-yard zero balances the problem — 2.855 mil of hold-under at 200 and 2.856 mil of hold-over at 800 — but no ten-inch plate in that spread is reachable with a centre hold. That is a genuinely useful thing for a solver to tell you before you walk to the line, rather than after.

13.11 A zeroing checklist

1. **Measure the sight height.** Not the mount's advertised height — the distance from the bore axis to the optical axis. A quarter-inch error moves the 100-yard zero angle by roughly 4%.
2. **Record the bore angle**, not only the zero distance. It is the quantity that survives a change of altitude, temperature or load.
3. **Run a tall-target test** and carry the correction factor. A 0.5% tracking error is 0.045 mil at 1000 yards — small; a 3% error is 0.27 mil, which is a miss on a torso plate.
4. **Know your grid.** Keep `--start` a multiple of `--step` in every printed table.
5. **Know your datum.** Ballistics Lab drop is measured down from the horizontal plane through the optic, and reads `+0.00` at the zero because the Ballistics Lab zeroes to a target at the height of the line of sight. CLI table drop is measured from the line of sight and reads zero there too. The two agree; a run where they do not is a run to investigate.

6. **Send a target height if you drive the engine yourself.** The BALLISTICS LAB fills `target_height_m` in for you as muzzle height plus sight height. A hand-built `solve-json` request that omits it is zeroed to the ground and carries a constant $h_{\text{sight}}/R_{\text{zero}}$ of error — 0.5 mil for a 1.8-inch scope on a 100-yard zero — with only an entry in the `assumptions` array to say so.
7. **Read each table's legend.** Drop is positive *below* the sight line, windage positive *right*, come-up positive when the sight must be raised. Elevation and windage therefore behave oppositely: a wind hold is the negative of the windage, a come-up is not the negative of the drop.

With a zero you trust and a dope card you can audit, the next question is whether the model that produced them agrees with the rifle. That is truing, and it is the subject of Chapter 14.

Chapter 14

Truing

A ballistic solver is a model. A rifle is not. Truing is the disciplined process of measuring where the two disagree and correcting the model — and it is the single most misunderstood operation in practical long-range shooting.

This chapter covers the `ballistics-engine`'s six truing and calibration subcommands: `true-velocity`, `estimate-bc`, `generate-bc-segments`, `true-wind`, `dsf`, and `plan-truing`. Along the way we will build a synthetic rifle whose "true" parameters we know, feed the engine observations from it, and check whether truing recovers what we hid. That is the only way to demonstrate a fitting procedure honestly: give it a problem whose answer you already know.

14.1 What truing is

Truing

Truing is adjusting one or more *model* parameters — muzzle velocity, ballistic coefficient, a transonic drop scale factor — so that the solver reproduces impacts you actually measured. The adjusted parameter is an *effective* value: the number that makes the model behave like your rifle, not necessarily the number a chronograph or a manufacturer would report.

The need is real, and it comes from four places at once.

- **Your chronograph is not the muzzle.** Most units read 10–15 feet downrange, and every one of them has an error budget. A 1% velocity error is 27 ft/s on a 2700 ft/s load, which is roughly 0.2 mil at 1000 yards.

- **The published BC is not your bullet's BC.** Manufacturers measure under conditions you did not shoot in, sometimes against the older Army Standard Metro reference atmosphere, and always with a different barrel.

- **Your scope does not track perfectly.** A 2% tracking error looks exactly like a 2% drop error to a fitting routine.

- **The drag model is an approximation.** A G7 curve is one standard projectile's shape. Yours differs, and the difference grows through the transonic region.

Truing is not curve-fitting away a bad BC

Every parameter in a ballistic model has a *distinct signature*. Muzzle velocity changes drop nearly in proportion to time of flight. Ballistic coefficient changes drop faster than that, because it changes the velocity decay that generates the time of flight in the first place. Those two signatures are similar over a narrow band of ranges and diverge outside it.

That similarity is the trap. Adjust muzzle velocity until the model matches one observed drop and you can *always* succeed — at that range. You will have absorbed a BC error, a scope-tracking error, an atmospheric error, and a rangefinder error into a number labelled "muzzle velocity", and the model will then be wrong everywhere else, in a direction you have no way to predict.

A trued muzzle velocity that differs from your chronograph by more than about 2% is not a trued velocity. It is a symptom.

14.2 The instruments

Command	Solves for	From
`estimate-bc`	BC (G1, G7, or both)	a drop curve and/or a velocity curve
`true-velocity`	effective MV; jointly MV+BC	one or more observed drops
`generate-bc-segments`	velocity-banded BC	a scalar BC and a bullet model name
`true-wind`	effective crosswind	an observed horizontal miss
`dsf`	a Mach-keyed drop scale factor	one transonic/subsonic observed drop
`plan-truing`	*which ranges to shoot*	a nominal load and a reading resolution
`tall-target`	a scope tracking factor	dialled versus measured travel

Table 14.1: The calibration surface of the engine. Only `tall-target` runs no trajectory.

`true-velocity` calls a network service by default

This is the one place in the `ballistics-engine` that reaches off the machine, and nothing in its normal output says so beyond a single line.

```
$ ballistics true-velocity ... --api-url http://127.0.0.1:1 -o json
Calculating effective velocity via API...
API Error: Network error: Connection refused
Hint: If the service rejected your credentials, run `ballistics login` (create a token at
     https://ballisticsinsight.com/account)
Hint: Use --offline for local calculation or --offline-fallback for automatic fallback
```

The credentials hint is printed on *every* API failure, including this one, where the service was never reached and no credential was ever offered. That is deliberate rather than sloppy. The endpoint sits behind authentication, and the HTTP client the engine uses raises a 401 as a status-code error, which is classified as `NetworkError("http status: 401")` by the time the hint would be chosen — so a hint shown only on a recognised authentication failure would stay silent in precisely the case it exists for. Printing it unconditionally costs a false lead on a genuine outage; withholding it costs a user who reads "network error", gives up on the online path permanently, and never discovers that an expired token was the whole problem.

Read the two lines in that light. If the machine has a route to the service, the first hint is the one to act on; if it does not — a laptop at a range, a firewalled build host — it is noise, and the second line is your answer.

Pass `--offline` to keep the solve local. The two paths do not agree to the last digit — see Section 14.4 — so pick one and stay on it. Nothing in the BALLISTICS LAB ever calls this command; the BALLISTICS LAB speaks only `solve-json`, which is offline by construction.

14.3 A rifle whose answer we know

Throughout this chapter the "rifle" is a 140 gr 6.5 mm match bullet at 2710 ft/s nominal, G7, zeroed at 100 yards with a 1.8-inch sight height, in the ICAO standard atmosphere. We will use the engine itself to generate the impacts a *particular* truth would produce, then hide that truth and try to recover it.

Truth A: the chronograph reads 2710 ft/s but the rifle actually delivers 2660 ft/s. The BC of 0.315 is correct.

Listing 14.1: Truth A — what a 2660 ft/s rifle actually does

```
$ ballistics come-ups -v 2660 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --start 400 --end 1000 --step 200 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
400,2.104,2.104,2113,1388,0.506
600,4.085,1.980,1866,1082,0.809
800,6.471,2.386,1633,829,1.152
1000,9.362,2.891,1414,622,1.547
```

Truth B — the interesting one — is the opposite error: the chronograph is right at 2710 ft/s, but the bullet's real BC is 0.295 rather than the published 0.315.

Listing 14.2: Truth B — a correct velocity and an over-stated BC

```
$ ballistics come-ups -v 2710 -b 0.295 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --start 400 --end 1200 --step 200 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
400,2.042,2.042,2122,1400,0.501
600,3.999,1.956,1858,1073,0.803
800,6.386,2.387,1611,806,1.150
1000,9.321,2.935,1379,591,1.552
1200,12.989,3.669,1166,423,2.026
```

Note how close Truth A and Truth B look at 800 yards: 6.471 versus 6.386 mil. Less than a tenth of a mil apart — two and a half inches on the target. Two completely different physical faults, nearly the same 800-yard dope. That is the identifiability problem in one line, and everything else in this chapter follows from it.

14.4 Velocity truing

Give `true-velocity` one observed drop and it searches for the muzzle velocity that reproduces it, holding everything else fixed. Feed it Truth A's 800-yard drop while believing the correct BC:

Listing 14.3: Recovering a velocity error, local solver

```
$ ballistics true-velocity --measured-drop 6.471 --range 800 \
      -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --chrono-velocity 2710 --offline -o json
Calculating effective velocity locally...
{
  "adjustment_percent": -1.7851994926199262,
  "calculated_drop_mil": 6.464061739800945,
  "confidence": "medium",
  "effective_velocity": 2661.62109375,
  "final_error_mil": -0.00693826019905508,
  "iterations": 11,
  "measured_drop_mil": 6.471,
  "used_bc_table": false,
  "velocity_adjustment": -48.37890625,
  "velocity_unit": "fps"
}
```

The hidden truth was 2660 ft/s; the fit returns 2661.62. That is 1.6 ft/s of error, which corresponds to the 0.0069 mil residual the solver reports — it stopped when the drop matched to seven thousandths of a mil, which is a quarter of an inch at 800 yards.

Run the same command without `--offline` and a different program answers:

Listing 14.4: The same observation through the remote solver

```
$ ballistics true-velocity --measured-drop 6.471 --range 800 ... -o json
Calculating effective velocity via API...
{
  "adjustment_percent": -1.95,
  "calculated_drop_mil": 6.48,
  "confidence": "high",
  "effective_velocity": 2657.0,
  "final_error_mil": 0.007,
  "iterations": 9,
  "measured_drop_mil": 6.471,
  "used_bc_table": false,
  "velocity_adjustment": -53.0,
  "velocity_unit": "fps"
}
```

2657.0 versus 2661.62 — 4.6 ft/s apart, with the remote path reporting to one decimal and calling its answer "high" confidence while the local path reports full precision and calls the same quality of fit

"medium". Neither is wrong; they are different implementations with different stopping rules and different confidence heuristics. Do not mix them inside one truing session.

Read the residual, not the confidence label

`final_error_mil` tells you how well the fitted velocity reproduces the observation you gave it. `iterations` tells you how hard the search worked. The word `high` or `medium` in `confidence` is a heuristic about the search, not a statement about your rifle.

14.4.1 What a field reading does to the answer

Nobody measures 6.471 mil. Round the observation to what a shooter would actually call from a target — 6.5 mil — and the fit moves:

Listing 14.5: A realistic reading resolution

```
$ ballistics true-velocity --measured-drop 6.5 --range 800 ... -o json
{
  "adjustment_percent": -2.13,
  "calculated_drop_mil": 6.51,
  "confidence": "high",
  "effective_velocity": 2652.0,
  "final_error_mil": 0.006,
  "iterations": 7,
  "measured_drop_mil": 6.5,
  "used_bc_table": false,
  "velocity_adjustment": -58.0,
  "velocity_unit": "fps"
}
```

A 0.029 mil difference in the reading — eight tenths of an inch at 800 yards — moved the fitted velocity by 5 ft/s. That sensitivity is not a defect of the solver; it is a property of the problem, and it is exactly what `plan-truing` exists to quantify.

14.5 The failure mode, demonstrated

Now take Truth B — correct velocity, over-stated BC — and do what most shooters do: true the velocity.

Listing 14.6: Truing MV against an 800-yard drop while believing a wrong BC

```
$ ballistics true-velocity --measured-drop 6.386 --range 800 \
      -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --chrono-velocity 2710 --offline -o json
{
  "adjustment_percent": -1.0622369407287824,
  "calculated_drop_mil": 6.42195426105298,
  "confidence": "medium",
  "effective_velocity": 2681.21337890625,
  "final_error_mil": 0.0359542610529795,
  "iterations": 13,
  "measured_drop_mil": 6.386,
  "used_bc_table": false,
  "velocity_adjustment": -28.78662109375,
  "velocity_unit": "fps"
}
```

It converged. It reports a 1.06% adjustment, comfortably inside the range a reasonable person would accept, and it produces a model that matches the observed 800-yard drop to within four hundredths of a mil. Everything looks fine.

Now push that model out to 1200 yards and compare it against what the rifle actually does:

Listing 14.7: The trued model versus the truth

```
# TRUTH:  MV 2710, BC 0.295
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
800,6.386,6.386,1611,806,1.150
1000,9.321,2.935,1379,591,1.552
1200,12.989,3.669,1166,423,2.026

# MV-ONLY TRUING:  MV 2681.21337890625, BC 0.315 (wrong)
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
800,6.351,6.351,1650,846,1.142
1000,9.187,2.836,1430,635,1.533
1200,12.668,3.481,1225,466,1.986
```

At 800 yards the trued model is 0.035 mil off — one inch, invisible. At 1000 yards it is 0.134 mil off. At 1200 yards it is 0.321 mil off, which is 13.9 inches: a clean miss on a torso plate, from a model that "matched the data."

Look at the velocity column too. The trued model predicts 1225 ft/s at 1200 yards where the rifle delivers 1166. It has the bullet still comfortably supersonic where the real one is entering the transonic region at Mach 1.04. Every downstream judgement — when the bullet destabilises, how much wind it takes, whether it will still cycle steel — is now wrong in the same direction.

The tell is the range dependence

A genuine velocity error produces a residual that is nearly proportional to time of flight. A BC error produces a residual that grows faster than that. If you true at one range you cannot see the difference. **If you cannot observe at two well-separated ranges, do not true.** Shoot your chronograph number and accept the error you know rather than manufacturing one you do not.

14.6 Fitting BC first

The right first move against Truth B is to fit the drag, not the velocity. `estimate-bc` takes a drop curve — distance and drop *in inches* — and searches for the BC that reproduces it.

Listing 14.8: Recovering the ballistic coefficient from three drops

```
$ ballistics estimate-bc -v 2710 -m 140 -d 0.264 \
      --data "400,29.4;600,86.4;800,183.9" --zero-range 100 --sight-height 1.8

BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G1     drop (3 pts)                0.586     0.04 in
  G7     drop (3 pts)                0.295     0.02 in
```

0.295 — the hidden truth, exactly, with a 0.02-inch residual across three stations. The default `--drag-model both` fits G1 and G7 side by side, and the RMS column is the discriminator: the G7 curve fits this boat-tail bullet twice as well as the G1 curve, which is what you would expect and a useful sanity check that your data is not garbage.

`--zero-range` changes what "drop" means

Supply `--zero-range` and the numbers in `--data` are read as drop below the line of sight from a rifle zeroed there — what a dope card records. Omit it and they are read as drop below the *extended bore axis*, which is a very different curve. There is no way for the command to tell which you meant.

The same command will fit against a velocity curve instead, which is what you have if you own a Doppler radar or shot over chronographs at several distances:

Listing 14.9: Fitting BC from velocity retention

```
$ ballistics estimate-bc -v 2710 -m 140 -d 0.264 --drag-model g7 \
      --velocity-data "200,2406;400,2122;600,1858;800,1611"

BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G7     velocity (4 pts)            0.295     0.19 fps
```

Same answer, 0.19 ft/s residual. Velocity data is the better instrument when you have it, because it measures drag directly instead of measuring drag's effect on a trajectory that also depends on your zero, your sight height and your scope tracking.

The fitted BC is always ICAO-referenced

Whatever atmosphere flags you pass, the BC `estimate-bc` reports is searched under the engine's own ICAO-calibrated retardation math. Do not feed it back with `--bc-reference army-standard-metro` — that would convert a number that is already converted, and cost you about 1.8% of drag.

14.6.1 Then true the velocity

With the drag fixed, the residual velocity error is small and honest:

Listing 14.10: Velocity truing on top of a fitted BC

```
$ ballistics true-velocity --measured-drop 6.386 --range 800 \
      -b 0.295 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --chrono-velocity 2710 --offline -o json
{
  "adjustment_percent": 0.32287822878228783,
  "calculated_drop_mil": 6.383748953601605,
  "confidence": "high",
  "effective_velocity": 2718.75,
  "final_error_mil": -0.002251046398394685,
  "iterations": 5,
  "measured_drop_mil": 6.386,
  "used_bc_table": false,
  "velocity_adjustment": 8.75,
  "velocity_unit": "fps"
}
```

Five iterations instead of thirteen, a 0.32% adjustment instead of 1.06%, and a residual one-sixteenth the size. Compare the resulting model against the truth at long range:

Listing 14.11: BC-then-MV, checked at 1200 yards

```
# TRUTH:                MV 2710,      BC 0.295 -> 1200 yd: 12.989 mil
# BC fitted, MV trued:  MV 2718.75,   BC 0.295 -> 1200 yd: 12.887 mil
# MV trued only:        MV 2681.213,  BC 0.315 -> 1200 yd: 12.668 mil
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
800,6.338,6.338,1618,813,1.145
1000,9.249,2.912,1385,596,1.546
1200,12.887,3.638,1172,427,2.018
```

The honest workflow lands 0.102 mil from truth at 1200 yards — 4.4 inches — where the shortcut lands 0.321 mil away, 13.9 inches. Three times better, from the same field data, by fitting the parameter that was actually wrong.

14.7 Fitting both at once

With more than one observation, `true-velocity` stops being a one-dimensional search and becomes a joint maximum-a-posteriori fit of muzzle velocity *and* ballistic coefficient, with a local Gaussian uncertainty estimate. Give it three of Truth B's drops and a stated reading error:

Listing 14.12: Uncertainty-aware joint MV+BC truing

```
$ ballistics true-velocity -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 \
      --measured-drop 2.042 --range 400 \
      --observed 800:6.386 --observed 1000:9.321 \
      --observation-sigma 0.05 --drop-unit mil \
      --predict-range 1200 --prediction-sigma 0.1 --offline

Fitting 3 weighted observations (uncertainty-aware joint MV+BC MAP)...

=== UNCERTAINTY-AWARE MV + BC TRUING ===
  Method: weighted joint MAP + local Gaussian approximation
  Observation errors: absolute known 1-sigma (mil)

  MAP muzzle velocity:    2710.18 fps
  MAP ballistic coefficient:  0.29490
  MV 95% interval: [2643.95, 2776.40] fps (SD 33.79)
  BC 95% interval: [0.26586, 0.32394] (SD 0.01482)
  MV/BC correlation: -0.98896

  Range (yd)  Observed (mil)      Sigma  Predicted (mil)   Std resid
       400.0        2.0420     0.0500        2.0422      +0.005
       800.0        6.3860     0.0500        6.3858      -0.005
      1000.0        9.3210     0.0500        9.3211      +0.002

  Chi-square: 0.000; effective DOF: 1.000; reduced chi-square: 0.000

  Predictive drop bands (95%):
      1200.0 yd: mean 12.9908 mil; model [12.6882, 13.2934]; future [12.6303, 13.3513]

  Warnings:
    - objective_mesh_convergence: LM's broad-stencil gradient test stalled (final norm
     2.372e-3); a direct-objective pattern search found no penalized-chi-square improvement
     above 1.0e-8 at scaled radius 1.0e-7 (largest observed 2.587e-10)
    - low_effective_degrees_of_freedom: one or fewer effective residual degrees of freedom:
     residual fit quality is weakly validated
    - prediction_outside_observed_domain: one or more predictive ranges lie outside the observed
     range domain; local linear uncertainty may understate nonlinear extrapolation risk
```

The point estimates are nearly perfect: 2710.18 ft/s against a truth of 2710, and 0.29490 against a truth of 0.295. The predicted 1200-yard drop is 12.9908 mil against a truth of 12.989 — two thousandths of a mil.

And then look at the intervals.

A correlation of -0.989 is the whole story

The 95% interval on muzzle velocity is 2644 to 2776 ft/s — 132 ft/s wide. The 95% interval on BC runs from 0.266 to 0.324, about $\pm 10\%$. Those are enormous intervals around a pair of point estimates that happen to be exactly right.

They are enormous because the fit found a *ridge*, not a peak. The MV/BC correlation of -0.98896 says that raising the velocity and lowering the BC moves you along a direction the data barely constrains. Almost every pair on that ridge reproduces these three drops. The fit landed on the truth because the data was noise-free; with a real 0.05 mil reading error it could have landed anywhere along the ridge and reported the same chi-square.

The predictive band is the number to trust. It says the 1200-yard drop lies in [12.688, 13.293] mil with 95% confidence — ± 0.3 mil, ± 13 inches — and that is an honest statement even though the point estimates are not individually identified.

The three warnings at the bottom are not noise, and the engine is right to print them:

- `low_effective_degrees_of_freedom` — three observations fitting two parameters leaves almost nothing to validate the fit *with*.
- `prediction_outside_observed_domain` — the 1200-yard band is an extrapolation beyond every station you shot.
- `objective_mesh_convergence` — the optimiser stopped on a stalled gradient test rather than a clean minimum, which for a ridge-shaped objective is the expected outcome.

All three are true. Read them.

14.8 Designing the experiment before you shoot it

If MV and BC are nearly degenerate over a narrow range band, the fix is to choose stations that break the degeneracy. `plan-truing` does exactly that: it takes a nominal load and a stated reading resolution, evaluates every combination of candidate ranges, and reports the set that makes the experiment identifiable.

Listing 14.13: Planning a three-station truing experiment

```
$ ballistics plan-truing -v 2710 -b 0.295 -m 140 -d 0.264 --drag-model g7 \
      --measurement-resolution 0.1 --range-grid 200:1200:100 \
      --observation-count 3 --minimum-separation 100

Evaluating 11 candidate ranges for a 3-station truing plan...

=== TRUING EXPERIMENT PLAN ===
  Recommendation: joint MV + BC
  Measurement assumption: independent 1-sigma reading error = 0.1000 mil
  Selected ranges: 700.0, 800.0, 1200.0 yd
  Constraints: 3 stations, minimum separation 100.0 yd

  BC sensitivity ratio: 0.30477
  Condition number: 98.56
  Singular values: min 21.53861, max 386.07318; weak-axis fractional 1-sigma: 0.0464

  Range (yd)    Drop (mil)  MV sensitivity  BC sensitivity   Info (nats)
       700.0        5.0855     -118.73864      -21.52713       0.33310
       800.0        6.3378     -148.27116      -30.87829       0.36211
      1200.0       12.9391     -317.37339     -106.25658       2.78749

  Search: exhaustive (165 of 165 candidate sets evaluated)
  Notes:
    - measurement model assumes independent 1-sigma 0.100000 mil drop uncertainty at every
     station
    - information-gain values use an identity reference information matrix in fractional MV/BC
     coordinates; they are experiment-design scores, not posterior intervals or an undeclared
     fit prior
    - local weak-axis fractional 1-sigma is 0.0464; it scales linearly with measurement
     resolution (0.0232 at 0.5x sigma, 0.0929 at 2x sigma)
```

There is a lot here, and all of it is useful.

Selected ranges: 700, 800, 1200. Not evenly spaced — the planner wants two nearby stations to pin the near-field curvature and one far station to break the MV/BC degeneracy. The `Info (nats)` column says why: the 1200-yard station carries 2.79 nats of information against 0.33 and 0.36 for the near pair. Almost all the discriminating power lives at the far station.

Condition number 98.56. This is the ratio of the largest to smallest singular value of the sensitivity matrix. A condition number near 1 would mean MV and BC are independently determined; 98.56 means the problem is still ill-conditioned even with the best three stations available. That is the honest state of MV/BC truing, and it is why the joint fit's intervals were so wide.

Weak-axis fractional 1-sigma: 0.0464. Along the poorly determined direction of the MV/BC plane, a 0.1 mil reading error leaves about 4.6% uncertainty. Halve your reading error and it halves; double it and it doubles. That is the return on buying a better spotting scope, quantified before you spend the money.

Run the plan before the range trip

`plan-truing` costs nothing and takes a second. Running it first turns "I will shoot some groups and true it" into a specific list of distances with a stated expected precision — and occasionally tells you that the ranges you have access to cannot identify what you want to know, which is worth learning at the desk rather than at the firing line.

14.9 Wind-call truing

Elevation is not the only thing you can true. `true-wind` back-solves the crosswind that would have produced an observed horizontal miss — and, more usefully, tells you what your own wind calls are worth.

Its sign conventions are stated in its help text and they are worth quoting exactly, because they catch everyone:

Listing 14.14: The `true-wind` sign conventions

```
--miss             positive = the group landed RIGHT of the aim point.
solved wind        positive = wind FROM the shooter's LEFT (9 o'clock), which
                   pushes impacts RIGHT; negative = FROM the right, pushing left.
--twist-right      a right-hand twist drifts RIGHT, left-hand drifts LEFT.
--shot-direction   compass bearing fired ALONG, 0 = North, 90 = East.
```

Suppose you called eight miles an hour from three o'clock at 800 yards, held nothing for it, and the group landed 44 inches left:

Listing 14.15: Back-solving the effective crosswind

```
$ ballistics true-wind --miss 800:-44 -v 2710 -b 0.295 -m 140 -d 0.264 \
      --drag-model g7 --twist-rate 8 --zero-distance 100 --sight-height 1.8 \
      --called-wind -8

Back-solving effective wind from 1 observed miss...

=== EFFECTIVE WIND TRUING (from observed horizontal miss) ===

  Range (yd)     Miss (in)   Spin/Cor (in)        Wind (mph)  Resid (in)
  ------------------------------------------------------------------------
       800.0        -44.00           +4.92            -10.53      +0.000
  ------------------------------------------------------------------------

  Effective crosswind:   -10.53 mph
  Called wind:            -8.00 mph   ->  wind-call correction factor 1.32
    (multiply your wind calls by 1.32 to match what actually hit)
  Effects subtracted:  spin drift
  NOT subtracted (absorbed into the solved wind): Coriolis (supply --latitude and
     --shot-direction to subtract it)
```

The wind was 10.53 mph, not the 8 you called: you under-call by a factor of 1.32. That number is far more valuable than the individual solve, because it is a correction you can apply to every future call.

Note the `Spin/Cor` column: +4.92 inches of the 44-inch miss was gyroscopic spin drift, not wind. That is why `--twist-rate` is a *required* argument on this command — without it, the fit would read your barrel's spin drift as a permanent left-hand wind. Add the latitude and shot direction and Coriolis comes out too:

Listing 14.16: Subtracting Coriolis as well

```
$ ballistics true-wind --miss 800:-44 ... --latitude 44 --shot-direction 90

  Range (yd)     Miss (in)   Spin/Cor (in)        Wind (mph)  Resid (in)
       800.0        -44.00           +6.45            -10.86      +0.000

  Effective crosswind:   -10.86 mph
  Effects subtracted:  spin drift, Coriolis
```

Coriolis accounts for another 1.53 inches shooting east at 44 degrees north, and the solved wind moves from 10.53 to 10.86 mph. The design principle here is worth naming: **anything the model does not**

know about gets absorbed into the fitted parameter, and the report tells you exactly what was and was not subtracted.

`--miss` is always in linear inches

`RANGE` follows `--units`, but `RIGHT` and `SIGMA` are inches in both unit systems, because they are tape measurements off a target. Scope tracking factors deliberately do not apply to them, and this command has no `--windage-cf` flag for that reason.

14.10 Transonic corrections: DSF

Beyond about Mach 1.2 a scalar BC stops describing the bullet. The Applied Ballistics answer is a **drop scale factor**: a multiplicative correction to predicted drop, keyed to Mach number, measured one range at a time and stored on a profile. The `ballistics-engine` implements it as `dsf`.

`dsf` is deliberately narrow. It reads everything from a saved profile — no CLI overrides at all — solves that profile's own trajectory, predicts the drop at the range you observed, and sets $\text{dsf} = \text{observed}/\text{predicted}$ at that range's Mach. It accepts at most six points, supersedes any existing point within 0.05 Mach, and *rejects* a supersonic observation outright, on the grounds that a supersonic mismatch is `true`-velocity's business.

Listing 14.17: Recording a transonic DSF point

```
$ ballistics profile save TRUEDEMO --velocity 2719 --bc 0.295 --mass 140 \
      --diameter 0.264 --drag-model g7 --twist-rate 8 --sight-height 1.8 \
      --zero-distance 1000
Profile 'TRUEDEMO' saved to "/Users/alexjokela/.ballistics/profiles/TRUEDEMO.json"

$ ballistics dsf --saved-profile TRUEDEMO --range 1050 --observed-drop 1.0mil
Added DSF point: Mach 1.19 -> DSF 1.2021
note: no subsonic window inside the solved range
Profile 'TRUEDEMO' saved to "/Users/alexjokela/.ballistics/profiles/TRUEDEMO.json"
DSF table for 'TRUEDEMO' (1 points, Mach 1.19-1.19):
  Mach 1.19  DSF 1.2021
```

The point lands in the profile file as ordinary JSON:

Listing 14.18: The stored DSF table

```
{
  "name": "TRUEDEMO",
  "velocity": 2719.0,
  "bc": 0.295,
  "mass": 140.0,
  "diameter": 0.264,
  "drag_model": "G7",
  "twist_rate": 8.0,
  "sight_height": 1.8,
  "zero_distance": 1000.0,
  "units": "imperial",
  "dsf_points": [
    {
      "mach": 1.1898038647069342,
      "dsf": 1.2020762106133436
    }
  ]
}
```

`dsf` cannot reach past the profile's ground impact

The profile solve `dsf` runs honours ground impact, and the default bore height is 60 inches. On a 100-yard-zeroed profile the trajectory terminates around 530 yards, and every transonic range you might want to calibrate is outside it:

```
$ ballistics dsf --saved-profile TRUEDEMO --range 1400 --observed-drop 17.9mil
error: saved profile 'TRUEDEMO' trajectory does not reach 1400 yd (solved to 530 yd)
```

Meanwhile `come-ups --profile TRUEDEMO` happily prints a 1400-yard row for the same profile. The two commands do not agree about where the trajectory ends.

The workaround is to save the profile with a far zero, which raises the bore angle enough to carry the flight out — but not so far that the predicted drop at your observation approaches zero, because DSF is a ratio and the engine rejects anything outside $(0.5, 2)$:

```
$ ballistics dsf --saved-profile TRUEDEMO --range 1400 --observed-drop 0.6mil
warning: observation at 1400 yd is beyond 90% of the solved range; solution
  reliability degrades past this point
error: DSF value 22661.093563869817 is out of range: must be finite and satisfy 0.5 < dsf
    < 2
```

Between those two walls there is a usable window, and the 1050-yard example above sits in it. Expect to hunt for it.

14.11 Velocity-banded BC

A middle course between a scalar BC and a full custom drag deck is a set of velocity-banded BCs. `generate-bc-segments` produces a starting set from a scalar BC and a bullet-family name:

Listing 14.19: Generated BC segments for a match bullet

```
$ ballistics generate-bc-segments -b 0.295 -m 140 -d 0.264 --drag-model G7 \
      --model "SMK" -o json
{
  "base_bc": 0.295,
  "caliber_inches": 0.264,
  "drag_model": "G7",
  "mass_grains": 140.0,
  "model": "SMK",
  "segments": [
    { "bc": 0.295,          "velocity_max_fps": 5000.0, "velocity_min_fps": 2800.0 },
    { "bc": 0.291915952,    "velocity_max_fps": 2800.0, "velocity_min_fps": 2400.0 },
    { "bc": 0.28780388799999995, "velocity_max_fps": 2400.0, "velocity_min_fps": 2000.0 },
    { "bc": 0.283691824,    "velocity_max_fps": 2000.0, "velocity_min_fps": 1600.0 },
    { "bc": 0.27957976,     "velocity_max_fps": 1600.0, "velocity_min_fps": 0.0 }
  ]
}
```

The BC falls about 5.2% from the top band to the bottom — a modest, monotone taper. Understand what this is: a *parametric estimate* from a bullet-family model, not a measurement of your bullet. It is a better starting point than a single scalar, and it is not a substitute for `estimate-bc` on your own data. Feed the segments to `trajectory` with repeated `--bc-segment VMIN:VMAX:BC` flags.

None of this reaches `solve-json` v1

BC segments, custom drag decks, powder-temperature curves and DSF tables are all explicitly outside the `solve-json` v1 contract, which is a deliberately narrow deterministic core. The Ballistics Lab therefore cannot consume them. What the Ballistics Lab *can* consume is the scalar output of a truing session: type the fitted BC into `projectile(...)` and the fitted velocity into `shot(...)`, and the Ballistics Lab will carry them — with full provenance — through every table it builds. Section 14.13 does exactly that and checks the result against the engine.

14.12 Scope tracking is not truing

One last distinction, because it is the most common way a truing session goes wrong. If your scope dials 10.06 mil when you ask for 10.00, then every dialled observation you feed a truing routine is 0.6% too small, and the routine will absorb that into velocity or BC.

Measure it separately (Section 13.5 showed the test), and hand the factor to the fit:

Listing 14.20: Keeping tracking error out of the fit

```
$ ballistics true-velocity ... --elevation-cf 1.0056
```

Dialled observations — mil and MOA drops — are multiplied by the factor before the fit, converting scope units to true angles, and dial-unit report values are shown back divided by it. Linear `in` drops are tape measurements and are never scaled. That asymmetry is correct and it is the reason the flag exists.

14.13 Carrying the result into the Ballistics Lab

A truing session ends in two scalars. Section 14.6 produced them for our synthetic rifle: a ballistic coefficient of 0.295 and a muzzle velocity of 2718.75 ft/s. Neither number should end its life in a shell history. Type them into an experiment, where they will be carried with the atmosphere they were trued in and the engine version that did the arithmetic:

Listing 14.21: The trued load as a Ballistics Lab experiment

```
let trued_bullet = projectile("140 gr 6.5 mm, BC trued", gr(140), inches(0.264),
    0.295, "G7")
let trued_rifle = rifle("6.5 mm truing demo", inches(1.8), m(0), inches(8), "right")
let icao = atmosphere(ft(0), fahrenheit(59), nil, 0.5, nil)
let calm = [wind(mph(0), deg(90))]
let trued_firing = shot(fps(2718.75), yd(1200), yd(100))
let numerics = solver_config("rk45", nil, yd(10), nil, nil, nil)
let trued_load = experiment("140 gr 6.5 mm, trued", trued_bullet, trued_rifle, icao,
    calm, trued_firing, numerics)
sessions.replace_experiment(lab, trued_load)
```

Listing 14.22: `:solve` then `:dope 400 1200 200 yd`; run header and provenance block elided

```
DOPE: 140 gr 6.5 mm, trued
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       400.00 |    +29.18 |         +2.03 |         2129.67 |        1409.68 |     0.50 | 1.91
       600.00 |    +85.72 |         +3.97 |         1865.35 |        1081.47 |     0.80 | 1.67
       800.00 |   +182.53 |         +6.34 |         1617.49 |         813.16 |     1.15 | 1.45
      1000.00 |   +332.99 |         +9.25 |         1384.99 |         596.20 |     1.55 | 1.24
      1200.00 |   +556.75 |        +12.89 |         1171.66 |         426.67 |     2.02 | 1.05
Signs: drop + below/- above; come-up + correction up/- correction down
```

Drop is positive *below* the line of sight, and the come-up is positive when the sight has to be raised — so both columns grow with distance, and the come-up column is the dial. Set it beside what the same trued model produces through the engine's own flag interface, on the same band:

Listing 14.23: The trued model through `come-ups`

```
$ ballistics come-ups -v 2718.75 -b 0.295 -m 140 -d 0.264 --drag-model g7 \
      --zero-distance 100 --sight-height 1.8 --start 400 --end 1200 --step 200 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
400,2.026,2.026,2130,1410,0.499
600,3.968,1.942,1865,1082,0.800
800,6.338,2.369,1618,813,1.145
1000,9.249,2.912,1385,596,1.546
1200,12.887,3.638,1172,427,2.018
```

Match the columns carefully, because the two programs use the word "come-up" for different quantities. The Ballistics Lab's `Come-up (mil)` is the **total** dial from the zero, so its counterpart is the engine's `drop_mil`; the engine's `come_up_mil` is the increment from the previous row and has no counterpart in the Ballistics Lab at all (Section B.5).

Read that way, they agree at every station: 2.026 against +2.03, 3.968 against +3.97, 6.338 against +6.34, 9.249 against +9.25, 12.887 against +12.89. The velocity columns agree too — 2130 against 2129.67, 1618 against 1617.49, 1172 against 1171.66. Two independent paths into the same engine, one over flags and one over `solve-json`, reporting the same trajectory.

Why the two land on the same zero

`come-ups` takes `--sight-height` and solves the zero to the line of sight. Over `solve-json` the same datum is an explicit field: `shot.target_height_m`, the height above ground the zero is solved to, which the engine defaults to 0.0 — the ground. The BALLISTICS LAB therefore states it. When an experiment sets no target height of its own, the wire adapter fills in muzzle height plus sight height, which is the height of the line of sight (`protocol_v1.lat:74–92`). A conventional zero puts the target on the line of sight, and that is what the request now says.

14.14 The honest workflow

1. **Fix the instrument first.** Run a tall-target test. Carry `--elevation-cf` into every subsequent command. A scope error masquerades perfectly as a ballistic one.

2. **Measure the atmosphere you shot in**, not the atmosphere you remember. Station pressure, not a sea-level altimeter setting, unless you pass `--pressure-type qnh`.

3. **Plan the experiment.** `plan-truing` with your real reading resolution and your actually-available ranges. If the condition number is bad even at the best stations, know that going in.

4. **Fit drag before velocity.** `estimate-bc` against a drop curve or, better, a velocity curve. A three-station drop curve recovered our hidden BC to the third decimal.

5. **Then true velocity**, with the fitted BC in hand. Expect a small adjustment. A large one means step 4 did not work.

6. **Or fit both jointly** with `--observed` and `--observation-sigma`, and *read the intervals, not the point estimates*. A correlation past -0.95 means you have a ridge.

7. **Validate outside the fitted band.** Predict at a range you did not use, then shoot it. This is the only step that can detect the failure in Section 14.5, and it is the one everybody skips.

8. **Only then reach for DSF**, and only below Mach 1.2.

9. **Record what you did.** Which observations, which ranges, which atmosphere, which engine version. A trued number without its provenance is a rumour.

Step 9 is what the Ballistics Lab is for

The Ballistics Lab cannot fit a BC. What it can do is hold the *result* — a named experiment carrying the trued projectile, the atmosphere it was trued in, and the engine version that did the arithmetic — and emit tables that carry all of it in the same artifact. Section 14.13 shows the handoff, and shows the resulting dope matching the engine's own come-up table station for station. A truing session that ends in a saved experiment document — the subject of the persistence chapter — is a session you can still defend a year later.

Chapter 15

Stability and the Small Effects

A point-mass trajectory — gravity, drag, wind — explains most of where a bullet goes. This chapter is about the rest: the handful of effects that come from the fact that the bullet is a *spinning rigid body* in a rotating frame, not a point.

Each of them has a magnitude, and the magnitudes are wildly different. Spin drift will move a 1000-yard impact eight inches. Aerodynamic jump will move it four inches for every ten miles an hour of crosswind. Coriolis will move it two and a half. Magnus, on its own, will move it four hundred-thousandths of an inch. Two of the effects the engine offers change the trajectory not at all — they are pure diagnostics.

Knowing which is which is the point of the chapter. We will measure all of them against the same load, at the same range, on the same day.

15.1 The reference measurement

Every number in this chapter comes from one load: a 140 gr 6.5 mm match bullet, 1.34 inches long, G7 BC 0.315, 2710 ft/s, 1:8 right-hand twist, 1.8-inch sight height, auto-zeroed at 100 yards, ICAO standard atmosphere, fired to exactly 1000 yards with `--ignore-ground-impact` so that every run terminates at the same place.

Listing 15.1: The baseline, and each effect switched on alone — four fields lifted from eleven JSON runs

```
$ ballistics trajectory -v 2710 -b 0.315 -m 140 -d 0.264 --drag-model g7 \
      --sight-height 1.8 --auto-zero 100 --max-range 1000 \
      --twist-rate 8 --bullet-length 1.34 --ignore-ground-impact \
      --sample-trajectory --sample-interval 914.4 --full -o json  [+ one effect flag]

                                x (yd)      max_height (yd)  tof (s)     impact (fps)
baseline                      +0.000000     1.7180948793    1.5134792    1450.871444
--enable-spin-drift           +0.223127     1.7180948793    1.5134792    1450.871444
--enable-magnus               +0.000000     1.7180937843    1.5134792    1450.871447
--enable-coriolis lat45 east  +0.069958     1.7185626303    1.5134777    1450.871179
--enable-coriolis lat45 west  +0.069958     1.7176702633    1.5134807    1450.871694
--enable-coriolis lat45 north +0.070349     1.7180948809    1.5134792    1450.871440
--enable-coriolis lat45 south +0.069568     1.7180948777    1.5134792    1450.871441
--enable-coriolis lat-45 east -0.069958     1.7185626303    1.5134777    1450.871179
--enable-pitch-damping        +0.000000     1.7180948793    1.5134792    1450.871444
--enable-precession           +0.000000     1.7180948793    1.5134792    1450.871444
--enable-aerodynamic-jump     +0.000000     1.7180948793    1.5134792    1450.871444
```

The `-o json` axis names are the opposite of the internal frame

The engine's internal integration frame is McCoy's: x downrange, y up, z right. The native JSON output relabels them — there, `x` is *lateral offset, positive right*, `y` is height above ground, and `z` is downrange distance. The table above uses the JSON convention, because that is what the command printed. If you read `src/wind.rs` and then read a JSON response, you will transpose x and z.

Three of the rows are byte-identical to the baseline in every column. That is not a rounding coincidence; `--enable-pitch-damping`, `--enable-precession` and (in calm air) `--enable-aerodynamic-jump` do not perturb the point mass at all.

15.2 Gyroscopic stability

Gyroscopic stability factor

The **gyroscopic stability factor** S_g compares the bullet's spin-derived gyroscopic rigidity against the overturning aerodynamic moment. Above 1.0 the bullet holds its nose forward; below 1.0 it tumbles. Above about 1.4 it is reliably stable; above about 1.5 it is stable with margin. It is dimensionless.

The engine uses the Miller formulation, which is an empirical fit in imperial units:

$$S_g = \frac{30\,m}{t^2\,d^3\,l\left(1+l^2\right)}$$

with m in grains, d in inches, t the twist rate in calibers per turn, and l the bullet length in calibers. Notice what is *not* in it: the ballistic coefficient. Stability is geometry and spin, not drag.

Listing 15.2: Stability for the reference bullet in a 1:8 twist

```
$ ballistics stability -l 1.34 -t 8 -m 140 -d 0.264 -v 2710

        Stability Analysis
 Stability Factor (Sg):          1.81
 Status:                Fully Stable
 Twist Rate:                 1:8" RH
 Min Twist (Sg=1.5):          1:8.8"
 Min Twist (Sg=1.0):         1:10.8"
 Bullet Length:             1.340   "
 Muzzle Velocity:           2710 fps
```

(The engine draws that report inside a Unicode box; the labels and values above are its contents, with the rules removed.)

Slow the twist and watch the margin evaporate:

Listing 15.3: Stability versus twist rate — two fields lifted from four JSON runs

```
$ for t in 8 9 10 12; do ballistics stability -l 1.34 -t $t -m 140 -d 0.264 -v 2710 -o json; done

twist   stability_factor   status
1:8               1.81     Fully Stable
1:9               1.43     Marginally Stable
1:10              1.16     Marginally Stable
1:12              0.80     UNSTABLE
```

The $S_g \propto 1/t^2$ dependence is unforgiving. Going from 1:8 to 1:10 — a 25% slower twist — costs 36% of the stability factor. The bullet that is comfortably stable in a match barrel keyholes in a hunting barrel.

15.2.1 The velocity correction

Miller's velocity correction $(v/2800)^{1/3}$ rides on top, and the `stability` subcommand applies it. For our bullet the base formula gives $S_g = 1.8298884$, and the command's answers track the correction exactly:

Listing 15.4: The Miller velocity correction — engine values from four runs, formula values computed alongside

```
$ for v in 2200 2710 2800 3200; do ballistics stability -l 1.34 -t 8 -m 140 -d 0.264 -v $v -o
     json; done
v=2200   1.69          base 1.8298884 * (2200/2800)^(1/3) = 1.6885457
v=2710   1.81          base 1.8298884 * (2710/2800)^(1/3) = 1.8100686
v=2800   1.83          base 1.8298884 * (2800/2800)^(1/3) = 1.8298884
v=3200   1.91          base 1.8298884 * (3200/2800)^(1/3) = 1.9131775
```

The velocity dependence is weak — a cube root — but it is real: the same bullet in the same barrel is 7% less stable from a short barrel at 2200 ft/s than at 2800.

15.2.2 Air density moves S_g too

The overturning moment is proportional to air density, so thinner air stabilises a bullet and colder, denser air destabilises it. Cold weather at sea level is the worst case:

Listing 15.5: Stability at 0 degrees F, then across three station pressures (one field per JSON run)

```
$ ballistics stability -l 1.34 -t 9 -m 140 -d 0.264 -v 2710 --temperature 0
 Stability Factor (Sg):          1.27
 Min Twist (Sg=1.5):          1:8.3"
 Min Twist (Sg=1.0):         1:10.1"

$ for p in 29.92 24.0 20.0; do ballistics stability ... --pressure $p -o json; done
pressure (inHg)   stability_factor
29.92                       1.43
24.00                       1.78
20.00                       2.14
```

Twenty inches of mercury — station pressure somewhere around ten thousand feet — lifts the same bullet from a marginal 1.43 to a comfortable 2.14. Dropping the temperature to 0 °F at sea level pushes it the other way, from 1.43 down to 1.27.

`stability -altitude` does nothing

The flag is accepted and ignored. Measured across a 15,000-foot sweep the stability factor never moves:

```
alt=0      Sg 1.43   min-twist(Sg=1.5) 8.78
alt=3000   Sg 1.43   min-twist(Sg=1.5) 8.78
alt=6000   Sg 1.43   min-twist(Sg=1.5) 8.78
alt=10000  Sg 1.43   min-twist(Sg=1.5) 8.78
alt=15000  Sg 1.43   min-twist(Sg=1.5) 8.78
```

`--temperature` and `--pressure` both work. If you want the stability factor at altitude, enter the *station pressure* there, not the altitude.

15.2.3 Where the BALLISTICS LAB gets its stability factor

You do not have to run `stability` separately — every solve reports it. The BALLISTICS LAB surfaces the engine's `summary.stability_factor` in the run header:

Listing 15.6: `: solve` reports the muzzle stability factor

```
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1000.00 yd
  maximum height   : +1.85 in
  time of flight   : 1.51 s
  terminal velocity: 1450.66 ft/s
  terminal energy  : 654.07 ft-lb
  stability factor : 1.81
  spin drift       : not reported
```

That is the *muzzle* S_g, computed from the twist rate and the projectile length you supplied. If you omit the length, `solve-json` estimates it from mass and diameter and says so, with an assumption coded `estimated_projectile_length` — and every geometry-dependent effect below inherits that estimate.

Measure the bullet

`projectile(name, mass, diameter, bc, drag_model, length)` takes an optional sixth argument. Supply it. Length enters S_g as $l(1 + l^2)$, so a 5% length error is a 15% stability error, and the same length drives the aerodynamic-jump coefficient in Section 15.7.

15.3 Spin drift

Spin drift

A gyroscopically stabilised bullet does not point exactly along its velocity vector. As the trajectory arcs over, the nose lags slightly above and to one side; the resulting small angle of attack — the **yaw of repose** — produces a steady sideways force. A right-hand twist drifts the bullet right, a left-hand twist left, in still air, always.

This is the largest of the small effects, and the only one that is present on every shot from every rifled barrel.

Listing 15.7: Spin drift at 1000 yards

```
$ ballistics trajectory ... --enable-spin-drift

 Spin Drift:            0.22 yd
 Direction:            Right
```

The JSON carries the exact figure: `spin_drift` is `0.22312675995993433` yards — 8.033 inches, or 0.223 mil, to the right at 1000 yards.

The engine implements the Litz empirical total-drift fit (`src/spin_drift.rs`):

$$\text{drift [in]} \;=\; 1.25\,\left(S_g + 1.2\right)\,t^{1.83}$$

with t the time of flight in seconds and S_g the *muzzle* stability factor. We can check that directly against the run above: $S_g = 1.8101438$, $t = 1.5134792$ s, so

$$1.25 \times (1.8101438 + 1.2) \times 1.5134792^{1.83} = 8.0325634 \text{ in}$$

and $8.0325634/36 = 0.22312676$ yards — the engine's number to fifteen significant figures. The formula *is* the model.

Spin drift is inside windage, not alongside it

In a `solve-json` response with `effects.enhanced_spin_drift` enabled, `summary.spin_drift_m` and the terminal sample's `windage_m` are equal in calm air. The drift has been folded into windage. Do not add them.
The BALLISTICS LAB reflects this faithfully:

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, nil, true))
:solve
:at 1000 yd

  stability factor : 1.81
  spin drift       : +8.03 in
Warnings
  - experimental_effect: The enhanced spin drift effect is an opt-in experimental v1
     model. [$.effects.enhanced_spin_drift]
Observation
  distance: 1000.00 yd
  drop    : +322.52 in
  windage : +8.03 in
```

`spin drift +8.03 in` and `windage +8.03 in` are the same eight inches reported twice.

Wind and spin drift superpose linearly. Adding a 10 mph wind from the right:

Listing 15.8: Spin drift on top of a crosswind — terminal lateral offsets from two runs

```
wind 10mph@90 only          x = -1.987195 yd
wind 10mph@90 + spin drift  x = -1.764067 yd
                   difference = +0.223128 yd  (8.033 in)
```

Exactly the calm-air drift, added to the wind deflection.

`--enable-spin-drift` works on its own

`CLI_USAGE.md` states that spin drift "Requires `--enable-magnus` or `--enable-coriolis` plus `--enable-spin-drift`". That is not true of the shipped binary: the run above carries neither companion flag and produces full drift. In `solve-json` the corresponding field is `effects.enhanced_spin_drift`, and it likewise stands alone — indeed it *conflicts* with `effects.magnus`.

15.3.1 When it matters

Time of flight enters as $t^{1.83}$, so spin drift is negligible up close and grows faster than range. Measured on the same load at 600 yards:

```
$ ballistics trajectory ... --max-range 600 --enable-spin-drift -o json
spin_drift  0.06826154414349489 yd  =  2.457 in  =  0.114 mil   (tof 0.792 s)
```

Two and a half inches at 600 yards against eight at 1000 — and 0.114 mil against 0.223. Below 600 yards it is inside your group. Past 800 it is a hold.

15.4 Coriolis

Coriolis acceleration

The Earth rotates at $\Omega = 7.2921159 \times 10^{-5}$ rad/s. A bullet flying over its surface is observed from a rotating frame, so it appears to accelerate sideways: $\vec{a}_{\text{cor}} = -2\,\vec{\omega} \times \vec{v}$. The horizontal component deflects right in the northern hemisphere and left in the southern, independently of which way you are shooting. The vertical component — the **Eötvös effect** — lifts eastward shots and drops westward ones.

Listing 15.9: Coriolis at 45 degrees latitude, 1000 yards — terminal sample and summary fields from six runs

```
                                x (yd)     terminal y (yd)   height vs baseline
baseline (no coriolis)         +0.000000       -7.2412947            ---
--latitude 45 --shot-direction  90 (east)  +0.069958  -7.1713227    +2.519 in
--latitude 45 --shot-direction 270 (west)  +0.069958  -7.3112671    -2.519 in
--latitude 45 --shot-direction   0 (north) +0.070349  -7.2412947     0.000 in
--latitude 45 --shot-direction 180 (south) +0.069568  -7.2412947     0.000 in
--latitude -45 --shot-direction 90 (east)  -0.069958  -7.1713227    +2.519 in
```

Read the two columns separately, because they are two different physical mechanisms.

The horizontal deflection is 0.070 yards — 2.52 inches — to the right, at every azimuth. North, south, east, west: 0.070349, 0.069568, 0.069958, 0.069958. The variation across the compass is under two hundredths of an inch. This is the classic result: in the northern hemisphere the horizontal Coriolis deflection depends on latitude and time of flight, and essentially not at all on the direction you face. Flip to the southern hemisphere and it reverses exactly: -0.069958.

The vertical deflection depends entirely on azimuth. Shooting east the bullet lands 2.519 inches high; shooting west, 2.519 inches low; shooting north or south, dead level. The full east-to-west spread is 5.038 inches at 1000 yards — twice the horizontal effect, and in the axis where a shooter is least likely to have allowed for it.

Coriolis needs a latitude, and the Eötvös split needs an azimuth

`--enable-coriolis` requires `--latitude`. Supplying the latitude but leaving `--shot-direction` at its default of 0 (north) gives you the horizontal deflection and *no* vertical component — which looks like a working Coriolis model right up until you shoot east.

In the Ballistics Lab the same requirement is enforced by the domain layer before any process is spawned:

```
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(15), hpa(1013.25), 0.5, nil))
Error
  code: evaluation_error
  message: assertion failed: experiment.atmosphere.latitude: required when Coriolis is
     enabled
```

The reference experiment ships with Coriolis enabled and latitude 45, so removing the latitude is what trips the check.

Confirming the same magnitude through the Ballistics Lab, with Coriolis on, latitude 45, and the default north azimuth:

Listing 15.10: Coriolis in the Ballistics Lab: horizontal only, because the azimuth is north

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, true, nil))
:solve
:at 1000 yd

Observation
  distance: 1000.00 yd
  drop    : +322.52 in
  windage : +2.53 in
```

`+2.53 in` of windage against the CLI's 0.069958 yd = 2.518 in, and the drop is unchanged at `+322.52 in` — identical to the calm baseline, because a northward shot has no Eötvös component.

`solve-json` has no shot-azimuth default worth trusting

`shot.shot_azimuth_rad` defaults to 0, which the response describes as "defaulted to 0 rad (north)". If you enable Coriolis and care about elevation, set it. The Ballistics Lab exposes it as the sixth argument to `shot(...)`.

15.4.1 When it matters

2.5 inches at 1000 yards is inside almost any practical group. At 1500 yards the time of flight roughly doubles and Coriolis grows faster than linearly with it. The honest summary: for a competition rifle inside 1000 yards it is a rounding error you should nonetheless enable because it costs nothing; past 1200 yards, and especially on east–west shots, it is a real correction.

15.5 Magnus

Magnus force

A spinning body moving through air with a small angle of attack experiences a force perpendicular to both the spin axis and the airflow. On a bullet, the Magnus force acts through a point ahead of or behind the centre of gravity and therefore also produces a *moment*, which is what actually matters for stability.

Switched on alone, Magnus is the smallest thing in this chapter:

Listing 15.11: Magnus, on its own, at 1000 yards — summary fields from two runs

```
baseline          max_height 1.7180948793 yd   impact 1450.871444 fps   x +0.000000
--enable-magnus   max_height 1.7180937843 yd   impact 1450.871447 fps   x +0.000000
                  difference 0.0000010950 yd = 0.0000394 in
```

Four hundred-thousandths of an inch of maximum ordinate, and no lateral displacement at all.

That is not a bug. The Magnus *force* on a well-stabilised bullet at a fraction of a degree of yaw genuinely is negligible. The Magnus *moment* is not negligible — it is one of the terms that determines the yaw of repose, and therefore spin drift. The engine's own architecture reflects that: Magnus and enhanced spin drift are **mutually exclusive**, because the enhanced spin-drift model is a calibrated empirical fit that already contains the Magnus contribution.

Listing 15.12: The Ballistics Lab rejects the combination before spawning a process

```
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), true, nil, true))
Error
  code: evaluation_error
  message: assertion failed: effects: magnus and enhanced_spin_drift cannot both be enabled
```

The legacy CLI solver silently suppresses Magnus when enhanced spin drift is set. `solve-json` v1 instead returns a hard `conflicting_fields` error, so the resolved request can never misstate which physics actually ran — and `domain.lat` catches it one layer earlier still, at construction time.

Magnus and enhanced spin drift both need a projectile length

Both effects are geometry-dependent, and the Ballistics Lab enforces the requirement:

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140), inches(0.264), 0.315,
    "G7"))
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, nil, true))
Error
  code: evaluation_error
  message: assertion failed: experiment.projectile.length: required for the selected effect
```

15.6 Pitch damping and precession

Two of the engine's flags change nothing about where the bullet goes. They are worth knowing about anyway, because what they *report* is not available any other way.

15.6.1 Pitch damping

Listing 15.13: `--enable-pitch-damping` adds a diagnostic block

```
$ ballistics trajectory ... --enable-pitch-damping

         TRAJECTORY RESULTS
 Max Range:          1000.00 yd
 Max Height:            1.72 yd
 Zero Angle:          0.0678 deg
 Time of Flight:       1.513 s
 Impact Velocity:    1450.87 fps
 Impact Energy:       654.26 ft-lb
 Stability (SG):        1.81
 Status:            STABLE
 Min Pitch Damping:   -5.000
 Transonic Status:  STABLE
```

Every trajectory number there is bit-identical to the baseline. What is new is the last two lines. The pitch-damping coefficient $C_{m_q} + C_{m_{\dot{\alpha}}}$ is regime-dependent, and the engine's defaults are worth memorising because they explain why transonic flight is dangerous:

Regime	Coefficient
Subsonic ($M < 0.8$)	-8.0
Transonic low ($0.8 \le M < 1.0$)	-3.0
Transonic high ($1.0 \le M < 1.2$)	$+2.0$
Supersonic ($M \ge 1.2$)	-5.0

Table 15.1: Default pitch-damping coefficients. A *positive* value is destabilising.

The sign flip at Mach 1.0 is the whole story. Between Mach 1.0 and 1.2 the damping term is positive: yaw excursions grow rather than decay. Our reference run stays above Mach 1.2 to the target — `Min Pitch Damping: -5.000` is exactly the supersonic constant — and the engine reports `Transonic Status: STABLE`. A load that crosses into that band with a marginal S_g is where groups fall apart, and this flag is how you see it coming without shooting the load.

15.6.2 Precession and nutation

Precession and nutation

A spinning bullet disturbed from perfect alignment responds with two superimposed angular motions: a slow **precession** — the nose sweeping a cone around the velocity vector — and a faster, smaller **nutation** riding on top of it. Together they are the "corkscrew" the nose traces.

Listing 15.14: `--enable-precession` adds an angular-motion block

```
$ ballistics trajectory ... --enable-precession

      Angular Motion Analysis
 Final Pitch Angle:   0.0010 rad
 Final Yaw Angle:    -0.0001 rad
 Max Yaw Angle:       0.0019 rad
                  (    0.11 deg)
 Max Precession:    284.4079 rad
 Nutation Phase:    1230.48 rad
```

Again, the trajectory summary is unchanged. The maximum yaw angle over the whole 1000-yard flight is 0.0019 radians — 0.11 degrees — which tells you directly why the Magnus force was four hundred-thousandths of an inch: there is almost no angle of attack to work with.

Diagnostic, not dynamic

Neither pitch damping nor precession feeds a force back into the point-mass integration. They are computed alongside it. Read them as answers to "is this bullet flying cleanly?", not as corrections to the impact point. Neither is part of `solve-json` v1, so neither is reachable from the Ballistics Lab.

15.7 Aerodynamic jump

Aerodynamic jump

A crosswind acting on a spinning bullet in the first instants after it leaves the muzzle produces a *vertical* impulse, not a horizontal one. The bullet is launched at a slightly different angle than the bore points. A right-twist bullet jumps up in a wind from the right and down in a wind from the left.

This is the effect most often left out of a mental model, and it is bigger than Coriolis.

Listing 15.15: Aerodynamic jump with a 10 mph full-value crosswind — terminal samples from five runs

```
                                x (yd)     terminal y (yd)   vertical shift
wind 10mph@90 (from right)     -1.987195      -7.2413363          ---
  + --enable-aerodynamic-jump  -1.987184      -7.1310190      +3.971 in
wind 10mph@270 (from left)     +1.987195      -7.2413363          ---
  + --enable-aerodynamic-jump  +1.987205      -7.3516539      -3.971 in
calm + --enable-aerodynamic-jump                (bit-identical to calm baseline)
```

Nearly four inches of vertical shift at 1000 yards from a ten-mile-an-hour crosswind — and the sign flips with the wind's side. Note also that the lateral displacement barely changes (1.987195 to 1.987184 yards): jump is vertical, wind deflection is lateral, and they do not interfere.

The solver uses the validated Litz crosswind estimator rather than the older heuristic:

$$Y \;=\; 0.01\,S_g \;-\; 0.0024\,L \;+\; 0.032 \qquad \text{[MOA per mph of crosswind]}$$

where L is the bullet length in calibers. For our bullet, $S_g = 1.8101438$ and $L = 1.34/0.264 = 5.0758$ calibers, giving

$$Y = 0.018101 - 0.012182 + 0.032 = 0.0379196 \text{ MOA/mph}.$$

Ten miles an hour is 0.379196 MOA, and one MOA subtends 10.471976 inches at 1000 yards, so the predicted jump is **3.9709 inches**. The measured shift was **3.9714 inches**. Five ten-thousandths of an inch apart — the formula and the implementation are the same thing.

The jump model is a regression, and it has a domain

The engine's own doc comment warns that the Litz fit is "valid mainly near $S_g \approx 1.75$; accuracy degrades for bullets well outside the fitted data set". Our S_g of 1.81 sits right in the fitted band, which is part of why the agreement above is so clean. A bullet at $S_g = 1.1$ or $S_g = 2.6$ is extrapolation.

The flag's help text also marks it **EXPERIMENTAL**. Take that at face value: use it to understand the *magnitude and sign* of an effect that is otherwise invisible, not as a precision correction you dial.

Which way does yours jump?

Right-hand twist, wind from the right: **up**. Right-hand twist, wind from the left: **down**. Left-hand twist reverses both. Because the effect scales with crosswind speed, a full-value 20 mph wind on this load is about eight inches of vertical at 1000 yards — comparable to the spin drift you would never dream of ignoring.

15.8 Putting the magnitudes side by side

Effect	Lateral	Vertical	When it matters
Spin drift	+8.03 in right	—	Always, past about 600 yd
Aero jump (10 mph)	—	±3.97 in	Any crosswind, past about 600 yd
Coriolis, horizontal	+2.52 in right	—	Past 1000 yd
Coriolis, Eötvös	—	±2.52 in	East/west shots past 1000 yd
Magnus (alone)	0	0.00004 in	Never, as a standalone force
Pitch damping	0	0	Diagnostic: transonic stability
Precession	0	0	Diagnostic: angular motion

Table 15.2: Every figure measured at 1000 yards on the reference load of Section 15.1. Coriolis figures are at 45 degrees latitude.

Two conclusions follow, and they are not the ones people expect.

First, **aerodynamic jump outranks Coriolis** for anyone who shoots in wind — which is everyone. A ten-mile-an-hour crosswind produces more vertical error than the Earth's rotation produces in any axis, and it does so at every latitude.

Second, **the two effects with the most impressive physics — Magnus and precession — move the impact point by nothing at all**. Their value is explanatory. If you want to know whether a load will survive the transonic transition, run `--enable-pitch-damping` and read the transonic status. If you want to know where the bullet lands, `--enable-spin-drift` and a correct twist rate will buy you more than every other flag in this chapter combined.

15.9 Turning them on in the Ballistics Lab

The Ballistics Lab exposes exactly the three effects that `solve-json` v1 carries, through the sixth, fourth and fifth arguments of `solver_config`:

Listing 15.16: The solver-config effect switches

```
solver_config(method, time_step_s, sampling_interval, magnus, coriolis,
    enhanced_spin_drift)

// Coriolis only (requires atmosphere latitude):
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, true, nil))

// Enhanced spin drift only (requires projectile length):
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, nil, true))

// Both:
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, true, true))
```

Wind shear, pitch damping, precession, aerodynamic jump and powder-temperature curves are all outside v1 and therefore outside the Ballistics Lab. If you need them, drive the `trajectory` subcommand directly — and accept that you have left the schema-versioned, provenance-carrying path behind to do it.

Why the Ballistics Lab validates before the engine does

Every conflict in this chapter is caught twice: once by `domain.lat` when you construct the value, and once by the engine when it decodes the request. The duplication is deliberate. The Ballistics Lab's copy gives you a Lattice-level error with a field path at the moment you make the mistake, before a process is spawned; the engine's copy guarantees the invariant even for a caller that is not the Ballistics Lab. Chapter 21 works through the whole relational-validation layer.

Everything so far has assumed the inputs are known. They are not. The next chapter is about what happens when you admit that.

Chapter 16

Uncertainty

Every trajectory in this book so far has been a single curve computed from numbers stated to fifteen decimal places. Not one of those numbers is known to fifteen decimal places. Your muzzle velocity varies shot to shot. Your ballistic coefficient came from a fit. Your wind call is an estimate of a quantity that changes while the bullet is in the air. Your rifle does not point in exactly the same direction twice.

A deterministic solver answers "where does the bullet go?" The honest question is "where does the *next* bullet go, and how sure am I?" This chapter is about the two tools available for answering it: the `ballistics-engine`'s Monte Carlo simulator, and deterministic perturbation studies run inside the Ballistics Lab.

16.1 What a single trajectory leaves out

Start with a deterministic sensitivity study, because it costs nothing and it tells you which uncertainties are worth simulating. Here is our reference load — 140 gr, G7 BC 0.315, 2710 ft/s, 1:8 twist, 100-yard zero, sea level, an 8 mph wind from the right — solved five times in the Ballistics Lab with one input perturbed at a time, and queried at 1000 yards.

Listing 16.1: A five-run perturbation study — three fields lifted from five `:at 1000 yd` blocks

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140), inches(0.264), 0.315, "G7",
    inches(1.34)))
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in", inches(1.8), m(0), inches(8), "right"))
sessions.replace_atmosphere(lab, atmosphere(m(0), celsius(15), hpa(1013.25), 0.5, nil))
sessions.replace_winds(lab, [wind(mph(8), deg(90))])
sessions.replace_shot(lab, shot(fps(2710), yd(1000), yd(100)))
sessions.replace_solver(lab, solver_config("rk45", nil, yd(100), nil, nil, nil))
:solve
:at 1000 yd

                        drop (in)    windage (in)   velocity (ft/s)
baseline  2710 / 0.315    +322.52          -57.25          1450.67
MV 2690   2690 / 0.315    +328.20          -57.94          1436.08
MV 2730   2730 / 0.315    +316.99          -56.56          1465.27
BC 0.309  2710 / 0.309    +326.13          -58.77          1429.90
BC 0.321  2710 / 0.321    +319.12          -55.80          1470.79
```

Read the deltas rather than the values.

- ± 20 ft/s of muzzle velocity — a realistic extreme spread for good handloads — moves the 1000-yard impact by about ± 5.6 inches, which is 0.155 mil.
- ± 0.006 of ballistic coefficient — about $\pm 2\%$, a realistic uncertainty on a published number — moves it by about ± 3.5 inches, 0.098 mil.
- The 8 mph wind alone is worth 57.25 inches of drift, so **one mile an hour of wind-call error is 7.16 inches** at 1000 yards, 0.199 mil.

One mile an hour of wind is worth more than twenty-five feet per second of muzzle velocity. That single comparison should reorder most shooters' priorities, and it took five solves to establish.

Do this before you simulate

A perturbation study is deterministic, reproducible, and interpretable. It tells you *which* inputs matter. Monte Carlo tells you how the errors combine into a distribution — a harder question you should only ask after you know which terms are worth including.

16.2 What `monte-carlo` actually perturbs

The `ballistics-engine`'s simulator draws random values for exactly five quantities and resolves a full trajectory for each trial.

Flag	Default	Perturbs
`--velocity-std`	1.0	muzzle velocity (fps / m s^{-1})
`--angle-std`	0.1	launch angle (degrees)
`--bc-std`	0.01	ballistic coefficient (dimensionless)
`--wind-std`	1.0	wind speed (mph / m s^{-1})
`--wind-direction-std`	0.0	wind direction (degrees)

Table 16.1: Every stochastic input `monte-carlo` offers. `-n/--num-sims` defaults to 1000.

The defaults are placeholders, not a shooter's error budget

An `--angle-std` of 0.1 degrees is 1.75 milliradians — a 6 MOA rifle. A `--bc-std` of 0.01 on a 0.3 BC is a $\pm 3.3\%$ drag uncertainty. Running `monte-carlo` without setting all five explicitly produces a distribution that describes no rifle in particular.

What it does *not* perturb matters as much: nothing atmospheric, nothing about your rangefinder, nothing about your zero, no group dispersion beyond what `--angle-std` represents, and — notably — `--angle-std` perturbs the launch angle in the vertical plane only. There is no horizontal aiming-error input at all. Every lateral spread the simulator produces comes from wind.

`monte-carlo` has no `-drag-model` flag

It always runs G1 (or a custom deck via `--drag-table`). Feeding it a G7 BC therefore models a very different, far draggier bullet than you have.
Measured directly, with the same launch angle of 0.5810360 degrees and the same BC of 0.315:

```
trajectory --drag-model g1  ->  838.14 yd = 766.40 m,  impact 312.78 m/s
trajectory --drag-model g7  -> 1097.72 yd = 1003.75 m, impact 410.88 m/s
monte-carlo (no dispersion) ->              765.43 m,  impact 313.01 m/s
```

The simulator lands on the G1 answer, 238 metres short of the G7 one.
The fix is to convert your BC to its G1 equivalent first. For our bullet, `estimate-bc --drag-model both` against the same drop curve reports G7 0.315 and G1 0.627 — so `monte-carlo -b 0.627` models the rifle we have been shooting all along.

16.3 Getting a run to reach the target

`monte-carlo` solves no zero. It fires at the bore angle you give it with `-a/--angle`, which defaults to 0 — horizontal. A horizontal shot from the default 60-inch bore height hits the ground in a few hundred metres, so the first run almost everyone makes reports a 100% shortfall:

Listing 16.2: The default launch angle is horizontal

```
$ ballistics monte-carlo -v 2710 -b 0.627 -m 140 -d 0.264 \
      --target-distance 1000 --target-radius 0.5 \
      --velocity-std 0 --angle-std 0 --bc-std 0 --wind-std 0 -n 5 -o full
{
  "mean_range": 418.00983459413635,
  "std_range": 0.0,
  "mean_impact_velocity": 635.2913667611188,
  "std_impact_velocity": 0.0,
  "cep": null,
  "target_shortfall_fraction": 1.0,
  "hit_probability": 0.0
}
```

Four hundred and eighteen metres — 457 yards — from a 1000-yard problem. And note `cep` comes back `null` rather than 0: not one trial reached the target plane, so there is no impact dispersion to report a circular error probable *about*. A null CEP is the shape of a run that never arrived.

Solve the angle first with `zero`, then hand it over:

Listing 16.3: A sanity run with every dispersion set to zero

```
$ ballistics zero -v 2710 -b 0.627 -m 140 -d 0.264 --drag-model g1 \
      --target-distance 1000 --sight-height 1.8 -o json | grep zero_angle_degrees
  "zero_angle_degrees": 0.5801835888575633,

$ ballistics monte-carlo -v 2710 -b 0.627 -m 140 -d 0.264 -a 0.5801835888575633 \
      --target-distance 1000 --target-radius 0.5 \
      --velocity-std 0 --angle-std 0 --bc-std 0 --wind-std 0 -n 5 -o full
{
  "mean_range": 1003.4216560528779,
  "std_range": 0.0,
  "mean_impact_velocity": 420.72590094790314,
  "std_impact_velocity": 0.0,
  "cep": 0.0,
  "target_shortfall_fraction": 0.0,
  "hit_probability": 1.0
}
```

Always run the zero-dispersion case first

With every standard deviation at zero, `std_range` must be exactly 0, `cep` exactly 0, and `hit_probability` exactly 1. If it is not, your launch angle, your BC, or your target distance is wrong, and every dispersed run after it will be wrong in the same way while looking plausible.

The summary reports metres regardless of `--units`

`--target-distance 1000` was interpreted in yards (imperial is the default), but `mean_range` came back as 1003.42 — metres. That is 1097.4 yards: the *ground-impact* range, well past the 1000-yard target, which is exactly right for a 1000-yard zero fired from a 60-inch bore height. `mean_range` is where the trial *landed*, not where the target was.

16.4 Reading a dispersed run

Now switch the dispersions on: 10 ft/s of velocity spread, 0.02 degrees of launch angle (0.35 mrad, about 1.2 MOA), 0.006 of BC, an 8 mph beam wind gusting ±2 mph, and a half-metre hit radius.

Listing 16.4: 5000 trials, default summary output

```
$ ballistics monte-carlo -v 2710 -b 0.627 -m 140 -d 0.264 -a 0.5801835888575633 \
      --target-distance 1000 --target-radius 0.5 \
      --velocity-std 10 --angle-std 0.02 --bc-std 0.006 --wind-std 2 \
      --wind-speed 8 --wind-direction 90 -n 5000

      MONTE CARLO RESULTS
      5000 simulations
 Mean Range:          1003.30 m
 Std Dev Range:         18.42 m
 Mean Impact Vel:      420.84 m/s
 Std Dev Velocity:       5.55 m/s
 CEP (arrivals):         0.42 m
 Target Shortfall:        0.0 %
 Hit Probability:        55.6 %
```

The same command run again, asking for JSON:

Listing 16.5: The identical command, one run later

```
{
  "mean_range": 1003.2638379718727,
  "std_range": 18.74181232626075,
  "mean_impact_velocity": 420.81362308263385,
  "std_impact_velocity": 5.590220527885996,
  "cep": 0.42889521814632,
  "target_shortfall_fraction": 0.0,
  "hit_probability": 0.5438
}
```

Two runs of one command, 5000 trials each: the hit probability moved from 55.6% to 54.38%, and the CEP from 0.42 to 0.4289 m.

There is no seed

`monte-carlo --help` offers no `--seed` flag. Every invocation draws a fresh stream, and identical commands give different answers. Three consecutive 5000-trial runs of one calm-air command:

```
cep 0.27979926720520676   hit_probability 0.7996   std_range 18.50033014796257
cep 0.28539588993164080   hit_probability 0.8076   std_range 18.51177639430830
cep 0.28598769780900990   hit_probability 0.7884   std_range 18.84016354149580
```

A 2-percentage-point spread on hit probability at $n = 5000$. Quote the run you made, note n, and do not read more than two significant figures out of any of it. Section 16.7 shows how to put an interval on that number.

16.4.1 The field meanings

Where trials struck the ground, in metres. Useful as a sanity check and as a measure of how much the trajectory shape wobbles; not a statement about the target.

The fraction of trials that never reached the target distance. Anything above zero invalidates the rest of the summary.

Circular error probable at the target plane, in metres — the radius containing half the impacts.

The fraction of trials landing within `--target-radius` of the centre of the impact pattern.

Hit probability measures precision, not accuracy

The hit test is taken about the *centroid of the simulated impacts*, not about your aim point. A steady crosswind you have not held for shifts the whole pattern and does not reduce the reported probability at all:

```
# 8 mph beam wind, zero dispersion, five-centimetre target radius
{"mean_range":1003.4197686293619, "cep":0.0, "hit_probability":1.0}

# calm, zero dispersion, five-centimetre target radius
{"mean_range":1003.4216560528606, "cep":0.0, "hit_probability":1.0}
```

That 8 mph wind is worth 1.441 metres of lateral displacement at 1000 yards, and the simulator still reports a certain hit inside a 5 cm circle. In `--target-distance` mode, `monte-carlo` answers "how tight is the group?", never "is the group where I aimed?"

16.5 Which uncertainty dominates

Because the simulator takes each dispersion independently, you can attribute the result by switching them on one at a time. Eight thousand trials each, same geometry, half-metre hit radius:

Listing 16.6: CEP attribution by input, 1000 yards — three fields from nine 8000-trial runs

	CEP (m)	P(hit r=0.5m)	std_range (m)
no dispersion	0.0000	1.0000	0.000
velocity-std 10	0.0503	1.0000	4.857
angle-std 0.02	0.2797	0.8040	17.895
bc-std 0.006	0.0300	1.0000	3.240
wind-std 2 (calm base)	0.0057	1.0000	0.661
wind 8@90, wind-std 0	0.0000	1.0000	0.000
wind 8@90, wind-std 2	0.2461	0.7884	0.001
wind 8@90, wind-dir-std 10	0.0105	1.0000	0.458
all four, calm	0.2837	0.8013	18.523

Four things fall out of that table, and each of them changes how you use the tool.

Launch-angle dispersion dominates everything. 0.02 degrees — a 1.2 MOA rifle — produces a 0.28 m CEP on its own, five times the velocity contribution and nine times the BC contribution. Rifle precision, not ballistic uncertainty, is what determines the group at 1000 yards for a well-characterised load.

Wind-speed dispersion does nothing without a base wind. With `--wind-speed` at its default of 0, and `--wind-direction` at its default of 0 — a headwind — perturbing the wind *speed* produces 0.0057 m of CEP. It is varying the strength of a wind that is blowing straight down the range. Set a base wind at an angle and the same `--wind-std 2` jumps to 0.2461 m.

`--wind-direction-std` is a weak instrument at a beam angle. Ten degrees of directional uncertainty on an 8 mph beam wind is 0.0105 m of CEP, because cos is flat near 90 degrees. At an oblique angle it would matter much more.

`std_range` tracks vertical dispersion. Angle-std produces 17.9 m of ground-impact scatter; wind-std at a beam angle produces one millimetre. That column is a free read on which axis your error lives in.

Build the budget one term at a time

Run each input alone, note its CEP, then run them together. If the combined CEP is far from the quadrature sum of the parts, one of your inputs is interacting with the geometry in a way worth understanding before you trust the answer.

16.6 The WEZ sweep

WEZ

A **weapon employment zone** sweep reports hit probability against a target of stated size as a function of range, and attributes the misses to their causes. It is invoked with `--wez` plus `--target-size`, and it replaces the single-distance summary.

Listing 16.7: A WEZ sweep against an 18 by 30 inch target

```
$ ballistics monte-carlo -v 2710 -b 0.627 -m 140 -d 0.264 -a 0.5801835888575633 \
      --velocity-std 10 --angle-std 0.02 --bc-std 0.006 --wind-std 2 \
      -n 2000 --wez --target-size 18x30 --wez-start 600 --wez-end 1000 --wez-step 200 \
      -o statistics
range,p_hit,dominant_error_source,wind_call_share,mv_sd_share,other_share
600.00,0.0000,other,0.0000,0.0097,0.9903
800.00,0.0000,other,0.0000,0.0215,0.9785
1000.00,0.6180,other,0.0000,0.0413,0.9587
```

At first reading that is alarming: zero hit probability at 600 and 800 yards, 62% at 1000. The explanation is the single most important fact about this mode.

The WEZ sweep fires every range at the same bore angle

It does not re-zero, and it does not dial. The angle you pass with `-a` is used for every row in the sweep. With a 1000-yard bore angle the bullet passes many feet above an 18-inch-wide target at 600 yards, so the hit probability is genuinely zero.
Swap in the 600-yard bore angle and the pattern inverts exactly:

```
$ ballistics zero ... --target-distance 600 -o json   ->  0.29165454171088057 deg
$ ballistics monte-carlo ... -a 0.29165454171088057 --wez ... -o statistics
range,p_hit,dominant_error_source,wind_call_share,mv_sd_share,other_share
600.00,0.9350,other,0.0000,0.0097,0.9903
800.00,0.0000,n/a,0.0000,0.0000,0.0000
1000.00,0.0000,n/a,0.0000,0.0000,0.0000
```

93.5% at 600 yards; `n/a` beyond, because the trajectory never reaches those ranges at that angle. So the question `--wez` answers is *"if I fire at this one bore angle*, what is my hit probability at each range?" — which is the right question for a fixed-sight or battle-zero weapon, and the wrong one for a rifle whose shooter dials. To get the dialled answer, run one sweep per range with that range's own zero angle.

The wind and mean-wind behaviour is the same trap as before, one step sharper. In WEZ mode the target is fixed in space, so an unheld mean wind *is* a miss:

Listing 16.8: WEZ hit probability at 1000 yards — the `p_hit` column of seven 4000-trial sweeps

```
                                P(hit) on 18x30 in
calm, no dispersion                      100.0%
calm, wind-std 2                         100.0%
calm, velocity-std 10                    100.0%
calm, angle-std 0.02                      65.6%
calm, bc-std 0.006                       100.0%
calm, all four                            64.7%
8 mph beam wind, no dispersion             0.0%
```

A perfectly precise rifle firing into a steady 8 mph crosswind, with no dispersion of any kind, misses an 18-inch-wide target 100% of the time — the 56.7-inch drift takes it off the plate. That is a truthful answer to the question the mode is asking, and it is a startling one if you assumed the shooter had held for the wind.

16.6.1 Wind-call error

`--wind-call-error` is the one input that models the *shooter* rather than the atmosphere, and it is available only with `--wez`. It composes with `--wind-std` in quadrature:

$$\sigma_{\text{eff}} = \sqrt{\sigma_{\text{wind}}^2 + \sigma_{\text{call}}^2}$$

Listing 16.9: A 4 mph wind-call error changes the attribution

```
$ ballistics monte-carlo ... --wez --target-size 18x30 --wind-call-error 4
WEZ sweep: 2000 sims/step, wind call 4.00 mph + wind std 2.00 mph (quadrature) = 4.47 mph
     effective
 Range(yd)   P(hit)   Dominant     Wind call   MV SD    Other/grp
     200.0     0.0%   other             8.9%    0.1%       91.0%
     400.0     0.0%   other            28.8%    0.2%       71.0%
     600.0     0.0%   other            46.8%    0.4%       52.8%
     800.0     0.0%   wind_call        58.9%    0.6%       40.5%
    1000.0     4.3%   wind_call        66.2%    0.7%       33.1%
```

Ignore the `P(hit)` column here — the whole sweep is at the 1000-yard bore angle again — and read the attribution. The wind-call share rises from 8.9% at 200 yards to 66.2% at 1000. Past about 800 yards, for this load and this shooter, *the wind call is the dominant error source*, and no amount of load development changes that. Nothing else in the engine states that conclusion so plainly.

The JSON form carries the same rows with SI units and full precision:

Listing 16.10: A WEZ sweep as structured output


```
{
  "target_size": { "width_m": 0.4572, "height_m": 0.762 },
  "wind_speed_std_mps": 0.89408,
  "wind_call_error_mps": 0.0,
  "combined_wind_speed_std_mps": 0.89408,
  "num_sims_per_step": 2000,
  "rows": [
    { "range_m": 548.64, "p_hit": 0.0,    "dominant_error_source": "other",
      "wind_call_share": 0.0, "mv_sd_share": 0.007668541672156486,
      "other_share": 0.9923314583278435, "attribution_unavailable": false },
    { "range_m": 731.52, "p_hit": 0.0,    "dominant_error_source": "other",
      "wind_call_share": 0.0, "mv_sd_share": 0.013973152199096138,
      "other_share": 0.9860268478009038, "attribution_unavailable": false },
    { "range_m": 914.4,  "p_hit": 0.0005, "dominant_error_source": "other",
      "wind_call_share": 0.0, "mv_sd_share": 0.02125111354623254,
      "other_share": 0.9787488864537676, "attribution_unavailable": false }
  ]
}
```


16.7 Welford, Wilson, and what the CLI does not print

The engine ships a careful streaming-statistics module, `src/mc_stats.rs`, and none of it is reachable from the `monte-carlo` subcommand in the version this book covers.

It contains three things worth knowing about.

Welford's algorithm accumulates a running mean and variance one trial at a time in constant memory. Its module comment states the reason plainly: the naive "sum of squares minus square of sum" variance "loses precision catastrophically once the values share a large common offset" — which is exactly the shape of a set of impact velocities all near 420 m/s. The `std_range` and `std_impact_velocity` you read are Bessel-corrected sample standard deviations built this way.

The Wilson score interval puts a confidence interval on a hit probability. Unlike the naive Wald interval $p \pm z\sqrt{p(1-p)/n}$, it stays inside $[0, 1]$ and keeps its coverage when n is small or p is near an endpoint — precisely the regime a hit-probability run starts in. It is computed in centre-plus-spread form:

$$\text{centre} = \frac{p + z^2/2n}{1 + z^2/n}, \qquad \text{spread} = \frac{z}{1 + z^2/n}\sqrt{\frac{p(1-p)}{n} + \frac{z^2}{4n^2}}$$

with three pinned critical values: $z = 1.6448536269514722$ for 90%, 1.959963984540054 for 95%, and 2.5758293035489004 for 99%.

A Bernoulli confidence sequence — an anytime-valid interval that lets an adaptive driver peek after every trial and stop on a data-dependent rule without inflating its error rate.

The adaptive surface is not in this build

`--adaptive`, `--confidence` and `--target-ci-half-width` exist in the source tree but not in the shipped binary:

```
$ ballistics monte-carlo ... --adaptive
error: unexpected argument '--adaptive' found
```

Neither is a `--seed` flag. Until those land, the intervals are yours to compute.

16.7.1 Computing the interval in the Ballistics Lab

The Wilson interval is six lines of Lattice, and the Ballistics Lab's REPL evaluates ordinary Lattice at the prompt, so you can compute it right beside the run it describes.

Listing 16.11: A Wilson score interval at the Ballistics Lab prompt

```
fn wilson(hits: Number, trials: Number, z: Number) {
    let p = (hits * 1.0) / (trials * 1.0)
    let denom = 1.0 + z * z / trials
    let center = (p + z * z / (2.0 * trials)) / denom
    let spread = (z / denom) * sqrt(p * (1.0 - p) / trials + z * z / (4.0 * trials * trials))
    return [center - spread, center + spread]
}
wilson(544, 1000, 1.959963984540054)
wilson(2176, 4000, 1.959963984540054)
wilson(0, 200, 1.959963984540054)
wilson(17, 20, 1.959963984540054)

[0.513021, 0.574642]
[0.52853, 0.559385]
[0, 0.0188453]
[0.639581, 0.947631]
```

The first two lines are the same 54.4% hit rate measured at $n = 1000$ and $n = 4000$: the interval narrows from ± 3.1 points to ± 1.5. The third is zero hits in 200 trials, where the Wald interval would collapse to the single point 0 and Wilson honestly reports "somewhere below 1.9%". The fourth is 17 of 20, where Wilson reports 64% to 95% — a reminder of how little a twenty-shot sample tells you.

Integer division will silently ruin this

Writing `let p = hits / trials` with two integer arguments truncates to zero, and the function returns a plausible-looking interval near the bottom of the range instead of an error. The first version of the listing above did exactly that and returned `[2.1684e-19, 0.00382676]` for 544 of 1000. Multiplying by `1.0` forces the floating-point path. This is a general Lattice hazard, not a ballistics one.

How many trials?

The Wilson half-width at $p \approx 0.5$ shrinks as $1/\sqrt{n}$: about ± 3.1 points at $n = 1000$, ± 1.5 at $n = 4000$, ± 1.0 at $n = 10{,}000$. Since `monte-carlo` is unseeded, run-to-run scatter is the same size as that interval — which is the honest reason not to report a hit probability to more than two significant figures.

16.8 What the BALLISTICS LAB can and cannot do here

`solve-json` v1 is deterministic by design: one request, one trajectory, no randomness anywhere. There is no Monte Carlo in the protocol and therefore none in the BALLISTICS LAB.

What the BALLISTICS LAB has instead is cheap, reproducible, fully-provenanced *re-solving*. The five-run study in Section 16.1 took a fraction of a second and produced numbers you can put in a report, because each one carries its engine version, its solver, its assumptions and its warnings. Extending it is a matter of writing a loop in Lattice at the prompt.

Deterministic sensitivity versus Monte Carlo

A perturbation study gives you $\partial(\text{impact})/\partial(\text{input})$ — exact, repeatable, and directly comparable across inputs. Monte Carlo gives you the shape of the combined distribution, including the parts where the response is non-linear. Use the first to build the error budget and to decide what to measure better; use the second when you need a probability rather than a sensitivity, and when you can state every input distribution honestly.

16.9 An uncertainty checklist

`mean_range` `std_range` `cep` `hit_probability` handing it to the simulator. Running `estimate-bc --drag-model both` gives you the pair.

2. **Solve the launch angle** with `zero` and pass it with `-a`. The default of 0 degrees is horizontal.
3. **Run the zero-dispersion case** and confirm `std_range` is 0, `cep` is 0 and `hit_probability` is 1.

4. **Set all five standard deviations explicitly.** The defaults describe a 6 MOA rifle.
5. **Give `--wind-std` something to vary.** With no base wind and the default headwind direction it contributes nothing lateral.
6. **Check `target_shortfall_fraction` first.** Anything above zero invalidates the rest.
7. **Remember what the hit test measures.** In `--target-distance` mode it is precision about the pattern centre. In `--wez` mode it is accuracy against a fixed target at a fixed bore angle.
8. **Attribute before you conclude.** One dispersion at a time, then all together.
9. **Put an interval on the probability**, and report n alongside it.
10. **Do not quote a Monte Carlo number twice.** It will not be the same number.

Uncertainty is where the shooter's half of this book ends. What follows is the machinery underneath: the engine's own command line, the `solve-json` contract the Ballistics Lab speaks, and then — in Part V — the fourteen thousand lines of Lattice that hold it all together.

Part IV

The Engine Underneath

Chapter 17

The Engine at the Command Line

Everything the Ballistics Lab has shown you so far arrived through a single door: one subprocess, one JSON request in, one JSON envelope out. That door is deliberately narrow. The `ballistics-engine` binary itself is not narrow at all — it carries thirty subcommands, a couple of hundred flags on its flagship solver, and a set of purpose-built tools for truing, reticles, and match paperwork that the Ballistics Lab never touches.

This chapter is the tour. We are going to drive the engine directly, by hand, grouped by the task you actually have at the bench: solve one shot, build a card, true a load, analyse dispersion, fit a reticle, keep house. Everything printed here came from a real run on a real binary, in the order shown.

17.1 Two binaries wearing the same name

Before a single number, a housekeeping matter that will otherwise bite you.

This book is pinned to `ballistics-engine` **v0.30.1**. On the machine that produced these transcripts, two different `ballistics` executables exist, and they are not the same program.

Listing 17.1: Two binaries, two versions

```
$ which ballistics
/Users/alexjokela/.local/bin/ballistics
$ ballistics --version
ballistics 0.32.0
$ /Users/alexjokela/projects/ballistics-engine/target/release/ballistics --version
ballistics 0.30.1
```

The 0.32.0 binary on PATH has six subcommands the 0.30.1 binary does not (`login`, `logout`, `recommend-powder`, `recommend-twist`, `recommend-col`, `calibrate-bc`), all of which reach a network service. Everything in Part IV of this book was captured with the **0.30.1** binary placed first on PATH, so the command you see typed is literally the command that ran. When you follow along, check `ballistics --version` first and expect small differences if your number is higher.

The engine's documentation ran ahead of the engine, and the engine caught up

`CLI_USAGE.md` and `docs/SOLVE_JSON_V1.md` in the engine repository describe the *development tree*, not a shipped binary. When these transcripts were captured that was a live hazard: those documents described five subcommands (`explain`, `error-budget`, `tolerance`, `dial-plan`, `adaptive-card`) and a request field (`atmosphere.pressure_reference`) that v0.30.1 and v0.32.0 both rejected outright.

Every one of those six shipped in v0.33.0 and is present in the current engine, v0.37.0. The five subcommands run there with the flags the manual documents; `atmosphere.pressure_reference` is accepted and performs its QNH-to-station-pressure reduction. So the specific gap this box was written to warn you about is closed, and Section B.1.1 records what closing it added.

The posture is what to keep. A manual tracks a working tree and a binary is a frozen artefact, so they will part company again at the next release — the only thing that changes is which of the two is ahead. Treat the documents as a roadmap and the binary as the contract, which, as we will see in Chapter 18, is exactly the posture the BALLISTICS LAB itself takes: it validates against what the engine actually answers, not against what the engine is documented to answer.

How transcripts are typeset

The engine draws its tables with Unicode box-drawing glyphs (`U+2550` and friends), plus a degree sign in the zero-angle line. This book's typesetting cannot place those characters inside a code listing, so every box glyph is rendered as its nearest ASCII equivalent (`+`, `-`, `|`) and the degree sign as `deg`. Every digit and every word is exactly what the engine printed; nothing is rounded.

Two further conventions apply throughout Part IV. Long invocations are wrapped with a trailing backslash, and long JSON strings are wrapped to the page width — so a JSON listing here is readable rather than parseable. Where rows or blocks have been dropped from a long output, the caption or the text says so explicitly.

17.2 The shape of every command

Three conventions run through the whole CLI, and knowing them saves a lot of squinting at help text.

One global unit switch. Every subcommand accepts `-u/--units <metric|imperial>`, defaulting to `imperial`. It selects how every dimensional *flag* is read and how tables are rendered. Internally the engine is SI throughout; the switch is a boundary translation only.

One output switch, mostly. Most subcommands take `-o/--output` with values `table` (the default), `json`, `csv`, and `pdf` — the last requiring `--output-file`. A few carry their own vocabulary: `monte-carlo` offers `summary`, `full`, and `statistics` instead.

Required arguments are enforced by the parser, not the solver. Ask for something incomplete, or misname a flag, and you get a usage error and exit status 2 before any physics runs. Pass `--zero-distance` to `trajectory` and it answers `error: unexpected argument '--zero-distance' found`, with a helpfully wrong suggestion (`--zero-set`) and the full usage line.

That is worth internalising: `trajectory` spells its zero `--auto-zero <DISTANCE>`, while `range-table`, `come-ups`, `wind-card`, `compare` and `hold-corridor` all spell it `--zero-distance`. The flag names are not uniform across subcommands, and neither are the sets of physics toggles each one exposes. We will flag each mismatch as we hit it.

17.3 Solving one shot: `trajectory`

`trajectory` is the flagship. Every physics toggle in the engine lives here and nowhere else. Let us fire the load this book keeps coming back to — a 175 gr match bullet, G7, BC 0.243, 0.308 inch, 2700 ft/s, zeroed at 100 yd with a 1.5 inch sight height and a 1:8 twist — and ask for 1000 yards.

Listing 17.2: A first trajectory — and a surprise

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --max-range 1000 --auto-zero 100 \
    --sight-height 1.5 --twist-rate 8
+========================================+
|         TRAJECTORY RESULTS             |
+========================================+
| Max Range:           508.38 yd         |
| Max Height:            1.71 yd         |
| Zero Angle:          0.0637deg           |
| Time of Flight:       0.687 s          |
| Impact Velocity:    1828.44 fps        |
| Impact Energy:      1298.87 ft-lb      |
+========================================+
| Stability (SG):        3.96            |
| Status:             STABLE             |
+========================================+

Bullet struck ground at 508 yd
```

We asked for 1000 yards and got 508. Nothing is broken. Ground-impact detection is on by default, and the default bore height is **60 inches** — five feet of muzzle above the ground datum. Launched a sixteenth of a degree nose-up, this bullet takes 0.687 seconds to rise, arc over, and reach the ground five feet below the muzzle — and 0.687 seconds of flight for this load is 508 yards. The engine told us so on the last line.

`--bore-height` is inches, not feet

The engine's own `CLI_USAGE.md` unit table says bore height is in "feet / meters." The flag's help text says inches and millimetres, with defaults of 60 in / 1500 mm. **The flag help is right.** A reader who believes the unit table and types `--bore-height 5` is modelling a rifle fired from five *inches* off the ground, and will see the trajectory truncate at a couple of hundred yards.

There are two honest ways out. Either raise the muzzle to where it really is, or tell the engine you do not care about the ground:

Listing 17.3: `--ignore-ground-impact` takes the run to the requested range (the Sg block is elided)

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --max-range 1000 --auto-zero 100 \
    --sight-height 1.5 --twist-rate 8 --ignore-ground-impact
| Max Range:          1000.00 yd        |
| Max Height:            1.71 yd        |
| Zero Angle:          0.0637deg          |
| Time of Flight:       1.706 s          |
| Impact Velocity:    1146.17 fps       |
| Impact Energy:       510.39 ft-lb     |

Bullet struck ground at 1000 yd
```

Note that the trailer still says the bullet struck ground, now at the range we asked for. It is a cosmetic leftover: the run terminated at `--max-range`, not at the ground.

17.3.1 Getting the numbers out

The boxed summary is for humans. For anything else, ask for a different surface. `-o json` gives you the summary plus a legend that tells you what the axes mean:

Listing 17.4: `trajectory -o json` — summary and axis legend


```
{
  "units": "imperial",
  "max_range": 1000.0,
  "max_height": 1.7116337445739929,
  "time_of_flight": 1.7062218910001727,
  "impact_velocity": 1146.1674756649934,
  "impact_energy": 510.387903967369,
  "stability_coefficient": 3.960863045940478,
  "spin_drift": null,
  "zero_angle_degrees": 0.06373386118322298,
  "trajectory": [],
  "legend": {
    "units": { "distance": "yd", "velocity": "fps", "energy": "ft-lb" },
    "axes": { "x": "lateral offset ... positive means to the shooter's right",
              "y": "height above the ground in the world frame ... This is NOT
                    height above the line of sight",
              "z": "downrange distance from the muzzle" }
  }
}
```


The JSON axis names are the transpose of the source

The engine's internal frame is McCoy's: **x downrange, y up, z right**. The `-o json` legend above uses the *opposite* convention: **x lateral, y up, z downrange**. Both agree on signs — a crosswind from the right pushes the bullet left, a right-hand twist drifts it right — but if you read the Rust source and then the JSON, you will transpose two axes. The `solve-json` interface avoids the whole question by naming its fields `distance_m`, `drop_m`, and `windage_m`.

The `trajectory` element is empty above because we did not ask for points. `--full` turns them on, and `--sample-trajectory --sample-interval <METRES>` replaces the integrator's own step points with an even downrange grid. Note the interval is *always* metres, in both unit systems. Here is 500 yards sampled every 91.44 m (100 yd), as CSV:

Listing 17.5: Sampled points as CSV — drop is measured below the line of sight

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --auto-zero 100 --sight-height 1.5 --twist-rate 8 \
    --max-range 500 --ignore-ground-impact \
    --full --sample-trajectory --sample-interval 91.44 -o csv
distance_yd,drop_in,drift_in,velocity_fps,energy_ft-lb,time_s
0.00,1.50,0.00,2700.00,2832.25,0.0000
100.00,-0.00,0.00,2513.40,2454.30,0.1152
200.00,4.01,0.00,2334.13,2116.68,0.2390
300.00,14.41,0.00,2162.56,1816.94,0.3726
400.00,32.26,0.00,1998.62,1551.91,0.5169
500.00,58.83,0.00,1841.34,1317.27,0.6733
```

Read that first row carefully, because it settles a sign question that recurs throughout this book. At the muzzle, `drop_in` is `+1.50` — exactly the sight height. At the 100 yd zero it is `-0.00`. Beyond the zero it grows positive. So **drop is positive downward**: it is the depth of the bullet below the line of sight, not its height above anything. The `solve-json` field `drop_m` carries the identical convention, which is why the BALLISTICS LAB's observation blocks read the way they do.

17.3.2 Looking at it

For a quick shape check without leaving the terminal, `--plot` renders four stacked panels — drop, drift, velocity, energy — against range. Bare `--plot` uses a braille-dot canvas; `--plot ascii` falls back to one asterisk per cell for terminals without braille coverage. The canvas is fixed at 72 by 12 cells and monochrome, and it applies only to `-o table`; the machine-readable formats are unchanged by it. The drop panel for the run above spans $y \in [-9.20, 1.71]$ yards over $x \in [0, 1000]$, which is the whole story of a 100-yard zero in one picture.

17.4 Where does it cross? `zero`

`zero` answers the question `trajectory --auto-zero` answers silently, but it answers it in both directions.

Forward — "what bore angle puts me on at 100 yards?":

Listing 17.6: Solving a zero angle

```
$ ballistics zero -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --target-distance 100 --sight-height 1.5
+========================================+
|          ZERO CALCULATION              |
+========================================+
| Target Distance:      100.0 yd         |
| Target Height:         0.00 yd         |
| Sight Height:         0.042 yd         |
+========================================+
| Zero Angle:          0.0637deg          |
| Zero Angle (MOA):      3.82 MOA        |
| Zero Angle (mrad):     1.11 mrad       |
| Max Ordinate:         0.045 yd         |
+========================================+
```

Backward — "I have a stored bore angle; what does it zero at?" A bore angle generally implies *two* crossings, and the command reports both. This is the classic battle-zero relationship: one angle, a near zero and a far zero.

Listing 17.7: Recovering both zero ranges from a stored angle

```
$ ballistics zero -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --from-angle 0.06373386118322298 --sight-height 1.5
+========================================+
|     ZERO RANGE FROM STORED ANGLE       |
+========================================+
| Input Zero Angle:    0.0637deg          |
+========================================+
| Near Zero (ascending):     58.6 yd     |
| Far Zero (descending):    100.0 yd     |
| Max Ordinate:         0.045 yd         |
+========================================+
```

(The MOA and mrad restatements of the input angle, and the target and sight height lines, are elided.) The round trip closes: the far zero comes back as 100.0 yd, and we learn as a bonus that this load also crosses the sight line on the way up at 58.6 yd. Hornady and Kestrel's 4DOF treat the bore angle as the portable quantity for exactly this reason — capture it once, recover the ranges it implies whenever you like.

`zero` defaults to G1, and it will not tell you

Every solver in the engine defaults `--drag-model` to `g1`. Feed a G7 BC to `zero` without saying `--drag-model g7` and it runs the G7 number against the G1 reference curve. Same command as above, minus the drag model:

```
| Zero Angle:          0.0657deg          |
| Zero Angle (MOA):      3.94 MOA         |
```

0.0657 degrees against the correct 0.0637: a 0.12 MOA error at the zero, which compounds all the way out. There is no warning. Say the drag model every time.

17.5 Building cards

Four subcommands exist purely to turn one load into a piece of paper you can tape to a stock.

17.5.1 `range-table` — everything at once

Listing 17.8: A full range table with a 10 mph crosswind from the right

```
$ ballistics range-table -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 --zero-distance 100 \
    --start 100 --end 1000 --step 100 --wind-speed 10 --wind-direction 90

Range Table (zero: 100 yd, 10 mph crosswind)
+-------+---------+---------+---------+---------+---------+---------+-------+
|Range  |Drop     |Drop     |Wind     |Wind     |Vel      |Energy   |ToF    |
|(yd)   |(in)     |(MIL)    |(in)     |(MIL)    |(fps)    |(ft-lb)  |(s)    |
+-------+---------+---------+---------+---------+---------+---------+-------+
|   100 |     0.0 |   0.000 |    -0.7 |   -0.20 |    2513 |    2454 |  0.12 |
|   200 |     4.0 |   0.557 |    -3.0 |   -0.41 |    2334 |    2117 |  0.24 |
|   300 |    14.4 |   1.334 |    -6.9 |   -0.64 |    2162 |    1817 |  0.37 |
|   400 |    32.3 |   2.240 |   -12.8 |   -0.89 |    1999 |    1552 |  0.52 |
|   500 |    58.8 |   3.268 |   -20.7 |   -1.15 |    1841 |    1317 |  0.67 |
|   600 |    95.7 |   4.430 |   -31.1 |   -1.44 |    1690 |    1109 |  0.84 |
|   700 |   144.8 |   5.744 |   -44.2 |   -1.76 |    1544 |     926 |  1.03 |
|   800 |   208.4 |   7.238 |   -60.5 |   -2.10 |    1404 |     766 |  1.23 |
|   900 |   289.9 |   8.946 |   -80.5 |   -2.49 |    1271 |     627 |  1.46 |
|  1000 |   392.9 |  10.913 |  -104.8 |   -2.91 |    1146 |     510 |  1.71 |
+-------+---------+---------+---------+---------+---------+---------+-------+
```

Two things to read off it. The wind column is **negative**: a wind *from* 90 degrees — from the shooter's right — pushes the bullet left. That is the engine's wind-FROM convention throughout, and it agrees with `windage_m` on the wire. And the mil column is a plain angular conversion of the inch column: 392.9 in at 1000 yd is $392.9/36000 \times 1000 = 10.91$ mil.

MOA in printed tables is 3438, on purpose

Turret units are a separate axis from `--units`. The `--adjustment-unit` flag takes `mil`, `moa`, `smoa`, `iphy`, or `clicks`, and the MOA conversion the printed tables use is the locked constant **3438**, not the exact 3437.7467. That is a deliberate, documented decision so that a printed card matches every other printed card in the sport. `smoa` and `iphy` are the same arithmetic ($\times 3600$) under two headers.

17.5.2 `come-ups` — elevation only

Listing 17.9: A come-up table

```
$ ballistics come-ups -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 --zero-distance 100 \
    --start 100 --end 1000 --step 100

Come-Up Table (zero: 100 yd, MIL)
+----------+----------+----------+----------+----------+----------+
|Range (yd)|Drop (MIL)|Come-Up   | Vel (fps)|Energy    | Time (s) |
+----------+----------+----------+----------+----------+----------+
|      100 |    0.000 |    -     |     2513 |     2454 |    0.115 |
|      200 |    0.557 |    0.557 |     2334 |     2117 |    0.239 |
|      300 |    1.334 |    0.777 |     2162 |     1817 |    0.373 |
|      400 |    2.240 |    0.906 |     1999 |     1552 |    0.517 |
|      500 |    3.268 |    1.028 |     1841 |     1317 |    0.673 |
|      600 |    4.430 |    1.162 |     1690 |     1109 |    0.843 |
|      700 |    5.744 |    1.314 |     1544 |      926 |    1.029 |
|      800 |    7.238 |    1.494 |     1404 |      766 |    1.233 |
|      900 |    8.946 |    1.708 |     1271 |      627 |    1.458 |
|     1000 |   10.913 |    1.967 |     1146 |      510 |    1.706 |
+----------+----------+----------+----------+----------+----------+
```

The Come-Up column is incremental, not cumulative

The two numeric columns say different things. **Drop** is the total dial from the zero; **Come-Up** is the *change from the row above*. Check it: $1.334 - 0.557 = 0.777$, and $10.913 - 8.946 = 1.967$. If you are dialing from zero to 1000 yd you want **10.913** mil, not 1.967. If you are walking a target board from 900 to 1000, you want 1.967. Read the header.

Switch `--adjustment-unit clicks` and both columns become integers — provided you also supply a graduation, either `--elevation-click-value` or a saved profile's stored click size. With a 0.1 mil turret the same table reads 6, 13, 22, 33, 44 clicks of drop at 200 through 600, with incremental come-ups of 6, 7, 9, 11, 11.

17.5.3 `wind-card` — one range column per wind speed

Listing 17.10: Drift in mils across four wind speeds

```
$ ballistics wind-card -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 --zero-distance 100 \
    --wind-speeds 5,10,15,20 --wind-angle 90

Wind Card (zero: 100 yd, MIL) - wind angle 90deg (wind-FROM: 0=head, 90=right, 180=tail,
     270=left)
+----------+----------+----------+----------+----------+
|Range (yd)|       5 mph |      10 mph |      15 mph |      20 mph |
+----------+----------+----------+----------+----------+
|      100 |     -0.1 |     -0.2 |     -0.3 |     -0.4 |
|      200 |     -0.2 |     -0.4 |     -0.6 |     -0.8 |
|      300 |     -0.3 |     -0.6 |     -1.0 |     -1.3 |
|      400 |     -0.4 |     -0.9 |     -1.3 |     -1.8 |
|      500 |     -0.6 |     -1.2 |     -1.7 |     -2.3 |
|      600 |     -0.7 |     -1.4 |     -2.2 |     -2.9 |
|      700 |     -0.9 |     -1.8 |     -2.6 |     -3.5 |
|      800 |     -1.1 |     -2.1 |     -3.2 |     -4.2 |
|      900 |     -1.2 |     -2.5 |     -3.7 |     -5.0 |
|     1000 |     -1.5 |     -2.9 |     -4.4 |     -5.8 |
+----------+----------+----------+----------+----------+
```

The card header restates the convention every time, which is a kindness. Wind direction also accepts clock positions — `3oc`, `10h30`, `10:30`, with `12oc` meaning a headwind — and those are identical to the numeric equivalents.

17.5.4 `lead` — moving targets

Listing 17.11: Lead on a 4 mph crossing target (rows at 100, 300, 600 yd)

```
$ ballistics lead -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 \
    --target-speed 4 --start 100 --end 600 --step 100

Moving-Target Lead Table (target speed: 4.0 mph, angle: 90deg, MIL)
Positive lead = hold in the direction of target travel.

+----------+----------+----------+----------+----------+
|Range (yd)|TOF (s)   |Lead ( yd)|Lead (MIL)|Intercept |
+----------+----------+----------+----------+----------+
|      100 |    0.115 |     0.23 |    2.252 |    100.0 |
|      300 |    0.373 |     0.73 |    2.429 |    300.0 |
|      600 |    0.843 |     1.65 |    2.749 |    600.0 |
+----------+----------+----------+----------+----------+
```

(The 200, 400 and 500-yard rows are elided above; nothing else is changed.) Note `lead` has no `--zero-distance` at all — lead is a time-of-flight problem, and where the rifle is zeroed does not enter it. The mil lead barely moves with range, 2.25 to 2.75 across 500 yards, because both the lead and the subtension scale with range; the linear lead more than doubles.

17.5.5 `compare` — loads side by side

Listing 17.12: Two loads at identical conditions

```
$ ballistics compare \
    --load "175 G7:g7:0.243:175:2700:0.308" \
    --load "168 G7:g7:0.223:168:2750:0.308" \
    --zero-distance 100 --start 200 --end 800 --step 200
Load Comparison
Zero: 100 yd | Wind: 10 mph @ 90deg | Sight: 1.5 in | Alt: 0 ft

  #1 175 G7: G7 BC 0.243, 175 @ 2700 fps (dia 0.308)
  #2 168 G7: G7 BC 0.223, 168 @ 2750 fps (dia 0.308)

   Range |           175 G7           |           168 G7
    (yd) |     Drop    Drift      Vel |     Drop    Drift      Vel
-------------------------------------------------------------------
     200 |     0.56    -0.41     2334 |     0.54    -0.44     2349
     400 |     2.24    -0.89     1999 |     2.20    -0.95     1983
     600 |     4.43    -1.44     1690 |     4.42    -1.56     1649
     800 |     7.24    -2.10     1404 |     7.33    -2.30     1342
-------------------------------------------------------------------
Drop/Drift in MIL (- is down/left), Vel in fps. Use -o json or -o csv for linear
drop/drift (in), energy (ft-lb), time of flight, and deltas vs load #1.
```

This is the shape of a real load decision. The faster, lower-BC 168 is flatter to about 600 yd and then loses — and it is already 0.20 mil wider in the wind at 800. The drift column is where these two loads actually differ; the elevation difference at 800 yd is 0.09 mil, which is under one click on a 0.1 mil turret.

The engine's `--load` syntax is `NAME:DRAG:BC:MASS:VELOCITY[:DIAMETER]`, repeatable two to eight times, and `--profile NAME` can be mixed in to pull a saved load (Chapter 19).

17.6 Truing: making the model match the target

Truing is the discipline of adjusting model inputs until predicted impacts match observed ones. The engine gives it five subcommands, and they differ in what they are willing to move.

17.6.1 `true-velocity` — one observation, or several

The single-observation form back-solves an effective muzzle velocity from one measured drop:

Listing 17.13: Truing muzzle velocity from a 1000 yd drop

```
$ ballistics true-velocity -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --measured-drop 11.4 --range 1000 --drop-unit mil \
    --zero-distance 100 --sight-height 1.5
Calculating effective velocity via API...

+============================================================+
|           VELOCITY TRUING RESULTS                          |
+============================================================+
|  Effective Muzzle Velocity:   2652.0  fps                  |
+============================================================+
|  Confidence:                     high                       |
|  Iterations:                        7                       |
|  Final Error:                  0.0130 MIL                  |
|  Calculated Drop:               11.41 MIL                  |
|  Measured Drop:                 11.40 MIL                  |
+============================================================+
```

(The "via API" banner is a leftover string; the solve is local. No network traffic leaves the machine.)

Give it one or more extra `--observed RANGE:DROP` pairs and it stops being a root-find and becomes a joint muzzle-velocity-and-BC fit with an uncertainty estimate:

Listing 17.14: A three-station joint MV + BC fit

```
$ ballistics true-velocity -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --measured-drop 11.4 --range 1000 --drop-unit mil \
    --zero-distance 100 --sight-height 1.5 \
    --observed 600:4.6 --observed 800:7.5 \
    --observation-sigma 0.1 --predict-range 1200
Fitting 3 weighted observations (uncertainty-aware joint MV+BC MAP)...

=== UNCERTAINTY-AWARE MV + BC TRUING ===
  Method: weighted joint MAP + local Gaussian approximation
  Observation errors: absolute known 1-sigma (mil)

  MAP muzzle velocity:    2670.28 fps
  MAP ballistic coefficient:  0.23854
  MV 95% interval: [2578.46, 2762.09] fps (SD 46.85)
  BC 95% interval: [0.21308, 0.26401] (SD 0.01299)
  MV/BC correlation: -0.98662

  Range (yd)  Observed (mil)      Sigma  Predicted (mil)   Std resid
      1000.0       11.4000     0.1000       11.3945      -0.055
       600.0        4.6000     0.1000        4.5846      -0.154
       800.0        7.5000     0.1000        7.5182      +0.182

  Chi-square: 0.060; effective DOF: 1.000; reduced chi-square: 0.060
```

It also reports a 95% predictive band at the range we asked it to extrapolate to: 1200 yards, mean 16.5599 mil, model interval [16.0277, 17.0920].

Truing is not curve-fitting

Look at that correlation: **-0.98662**. Muzzle velocity and BC are almost perfectly anti-correlated over this range band — lowering one and raising the other produces nearly the same drop curve. That is why the 95% interval on velocity is 184 fps wide even though every residual is under a fifth of a sigma. The fit is excellent and the parameters are barely identified. Chasing a "trued" BC to five decimal places from three targets at similar ranges is arithmetic, not measurement.

The engine says as much itself. The run above also emitted two warnings we elided from the transcript: an `objective_mesh_convergence` notice that the optimiser's gradient test stalled with a final norm of `1.700e-2`, and a `prediction_outside_observed_domain` notice that the 1200-yard prediction band lies outside the observed range domain, "so local linear uncertainty may understate nonlinear extrapolation risk." Both are the fit telling you not to over-read it.

17.6.2 `plan-truing` — choosing the targets first

If velocity and BC are anti-correlated, the fix is to pick ranges that separate them. That is exactly what `plan-truing` does: it scores candidate range sets by how identifiable they make the two parameters.

Listing 17.15: Designing a three-station truing experiment

```
$ ballistics plan-truing -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --measurement-resolution 0.1 --range-grid 400:1200:100 \
    --observation-count 3 --drop-unit mil
Evaluating 9 candidate ranges for a 3-station truing plan...

=== TRUING EXPERIMENT PLAN ===
  Recommendation: joint MV + BC
  Measurement assumption: independent 1-sigma reading error = 0.1000 mil
  Selected ranges: 700.0, 800.0, 1200.0 yd
  Constraints: 3 stations, minimum separation 0.0 yd

  BC sensitivity ratio: 0.38708
  Condition number: 110.35

  Range (yd)    Drop (mil)  MV sensitivity  BC sensitivity   Info (nats)
       700.0        5.6251     -133.98718      -30.43204       0.32991
       800.0        7.1163     -171.74107      -45.34512       0.36644
      1200.0       15.6782     -390.69002     -164.30731       2.81210

  Search: exhaustive (84 of 84 candidate sets evaluated)
```

The 1200-yard station carries almost eight times the information of either short station, because that is where the BC sensitivity finally pulls away from the velocity sensitivity. The condition number of 110 is the quantitative version of the warning box above.

17.6.3 `estimate-bc` — fit the BC instead

Turn the problem around and hold velocity fixed. Here we feed `estimate-bc` three drop observations taken straight off the range table we printed earlier:

Listing 17.16: Recovering a BC from drop data

```
$ ballistics estimate-bc -v 2700 -m 175 -d 0.308 \
    --data "300,14.4;500,58.8;800,208.4" --drag-model g7 \
    --zero-range 100 --sight-height 1.5
BC Estimation
  Model  Fit basis            Estimated BC   Fit RMS
  ------ -------------------- ------------   ----------
  G7     drop (3 pts)                0.243     0.04 in
```

It recovers 0.243 — the BC we started with — to an RMS of 0.04 inches. That is a closed loop, and it is the best kind of confidence check: the fitter and the solver agree with each other on synthetic data before you trust either on real data.

17.6.4 `true-wind` — back-solving the wind call

You missed right. Was it wind, or spin drift, or the earth? `true-wind` takes the observed horizontal miss and subtracts what it can account for:

Listing 17.17: Back-solving the effective crosswind from two misses

```
$ ballistics true-wind -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --twist-rate 10 --velocity 2700 --miss 600:12.5 --miss 800:22.0
Back-solving effective wind from 2 observed misses...

=== EFFECTIVE WIND TRUING (from observed horizontal miss) ===

  Range (yd)     Miss (in)   Spin/Cor (in)        Wind (mph)  Resid (in)
  ------------------------------------------------------------------------
       600.0        +12.50           +3.42             +2.92      -0.000
       800.0        +22.00           +6.85             +2.50      +0.000
  ------------------------------------------------------------------------

  Effective crosswind:    +2.71 mph  (mean of 2 observations)
  Effects subtracted:  spin drift
  NOT subtracted (absorbed into the solved wind): Coriolis (supply --latitude
  and --shot-direction to subtract it)
```

The command closes with a four-line sign legend we have elided: `--miss` positive means the impact landed right of aim, and a positive solved wind means a wind *from* the shooter's left — the wind-FROM convention, as everywhere else in the engine.

Of the 12.5 inches of right-hand miss at 600 yards, 3.42 was spin drift — which is why `--twist-rate` is **required** on this command. Without a twist rate the engine would silently attribute all of that to wind. Note also the honest disclosure line: Coriolis was not subtracted because we did not supply a latitude, so it is folded into the answer.

`--miss` is always in inches

`RANGE` in the `--miss` triple follows `--units`, but `RIGHT` and `SIGMA` are **always linear inches**, in both unit systems. They are tape measurements off a target, and the engine treats them as such.

17.6.5 `tall-target` — is the scope honest?

Pure arithmetic, no trajectory. You dialed a known amount, you measured what the reticle actually travelled, and the ratio is your scope's tracking correction factor:

Listing 17.18: A tall-target test

```
$ ballistics tall-target --dialed 10 --measured 34.9 --range 100 --unit mil

Tall-Target Test Result
  Dialed travel:   10.00 MIL
  Actual travel:   9.69 MIL (34.90 in at 100 yd)
  Correction factor (actual / dialed): 0.9694
  Enter this as --elevation-cf / profile elevation_cf (or the windage
  equivalents); dial solutions are divided by it.
```

A scope that moves 3% less than it claims will put you 0.33 mil low at 1000 yards on an 11-mil dial — roughly a foot. Feed the 0.9694 back in through `--elevation-cf` and every dial output is divided by it. Raw linear inches are untouched; only angular dial values are corrected. The factor is bounded to $(0.5, 1.5)$, which is a sanity rail, not a physical claim.

17.6.6 `dsf` — a Mach-keyed drop correction

Past about Mach 1.2 the reference drag curves stop describing any particular bullet very well. The drop-scale-factor mechanism lets you pin the model to reality in the transonic band, keyed by Mach rather than by range so it travels across atmospheres. `dsf` derives one point and writes it into a saved profile:

Listing 17.19: Deriving and storing a DSF point

```
$ ballistics dsf --saved-profile SUBDEMO --range 200 --observed-drop 4.9mil
Added DSF point: Mach 0.88 -> DSF 1.0866
warning: no DSF point in the transonic band (Mach 1.2-0.9); transonic drops
remain uncorrected
note: DSF window: at or beyond 105.3 yd (90% of the Mach 0.9 distance)
Profile 'SUBDEMO' saved to "/Users/alexjokela/.ballistics/profiles/SUBDEMO.json"
DSF table for 'SUBDEMO' (1 points, Mach 0.88-0.88):
  Mach 0.88  DSF 1.0866
```

The command refuses to record a supersonic observation, and it says why:

Listing 17.20: `dsf` guards its own domain

```
$ ballistics dsf --saved-profile LABBOOK --range 400 --observed-drop 2.35mil
error: observation is supersonic (Mach 1.79); calibrate muzzle velocity first
(true-velocity), then collect DSF points at Mach <= 1.2
```

That refusal is the right order of operations stated as an error message: velocity first, then BC, then transonic shape. Note also `--observed-drop` takes a *unit-suffixed literal* with no separator — `4.9mil`, `17.4moa`, `42.0in`.

17.7 Analysis and design

17.7.1 `mpbr` — point-blank range

Listing 17.21: Maximum point-blank range for an 8-inch vital zone

```
$ ballistics mpbr -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 --vital-zone 8

MPBR Analysis (vital zone: 8.0 in)
  MPBR:               301 yd
  Optimal zero:       256 yd
  Near zero:           22 yd
  Far zero:           256 yd
  Max ordinate:       4.0 in at 139 yd
  Impact velocity:   2161 fps
  Impact energy:     1814 ft-lb
```

(The box rules around that block are dropped here for space; the values are verbatim.) The max ordinate lands exactly on half the vital zone, which is the definition doing its job: hold centre from the muzzle to 301 yards and the bullet stays inside a four-inch band above and below the line of sight.

17.7.2 `optimal-zero` — one zero for a whole stage

Listing 17.22: Finding the minimax zero for three targets

```
$ ballistics optimal-zero -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 \
    --target 200:10 --target 350:10 --target 500:12 --vital 10
Optimal Zero (min-max hold)
Optimal zero:  365.4 yd
Largest hold:  1.356 mil

Range(yd)  Target(in)  Hold(mil)  Center-hold miss(in)  Fits
---------  ----------  ---------  --------------------  ----
      200        10.0     -1.356                 -9.77  NO
      350        10.0     -0.141                 -1.78  yes
      500        12.0      1.355                 24.40  NO

At least one target needs a hold: no single zero keeps them all centered.
Center-hold miss is signed: positive = the bullet lands LOW of the aim point.
```

The answer is symmetric by construction — 1.356 mil high at 200 balanced against 1.355 mil low at 500 — and the command is candid that two of the three targets still need a hold.

17.7.3 `hold-corridor` — holds that survive every wind scenario

This is the most interesting of the analysis commands, and the least like anything else in the toolbox. You describe several named segmented-wind scenarios; it reports, per range, the hold that minimises the worst case across all of them.

Listing 17.23: A three-scenario wind file

```
{"version":1,"nominal":"steady","scenarios":[
 {"name":"steady","segments":["8:90:1000"]},
 {"name":"letup","segments":["8:90:400","3:90:1000"]},
 {"name":"switch","segments":["8:90:500","8:270:1000"]}]}
```

Listing 17.24: The corridor at 900 yards (the 300, 500 and 700-yard blocks are elided)

```
$ ballistics hold-corridor -v 2700 -b 0.243 -m 175 -d 0.308 \
    --drag-model g7 --sight-height 1.5 --zero-distance 100 \
    --scenarios scen.json --ranges 300,500,700,900 --target rect:10x20
Robust Hold Corridor
====================

Scenarios (3): letup, steady, switch
Nominal:        steady
Target:         rectangle 10.0 x 20.0 in (per-axis / L-inf metric)

Holds are milliradians: elevation positive = hold UP, windage positive = hold RIGHT.
The corridor is the span of the scenarios you supplied. It is NOT a probability interval.

--- 900 yd ---
  Scenario              Elev(mil)  Wind(mil)
  letup                     8.946     -1.472
  steady                    8.946     -1.988
  switch                    8.946     -0.836
  Corridor                  8.946     -1.988   (min)
                            8.946     -0.836   (max)
  Minimax hold              8.946     -1.412
  Worst case from it        0.576 mil (18.67 in), scenario 'switch'
  Holding the nominal       1.152 mil (37.33 in)
  Fits target           NO - no single hold covers every scenario here
```

At 300 yards, in the block we elided, all three scenarios agree to three decimals (`1.334` elevation, `-0.512` wind) because no segment boundary has been crossed yet, and the command reports `Fits target  yes`. At 900 they have fanned out over a mil and a sixth, and the minimax hold cuts the worst-case miss exactly in half: 0.576 mil instead of 1.152 for holding the nominal. The line the command prints about the corridor *not* being a probability interval deserves to be read twice. It is the span of the scenarios you chose to type in, and nothing more.

17.7.4 `monte-carlo` — dispersion and hit probability

Listing 17.25: 2000 simulations against an 800-yard target

```
$ ballistics monte-carlo -v 2700 -b 0.243 -m 175 -d 0.308 -n 2000 \
    -a 0.4453 --target-distance 800 --target-radius 5 \
    --velocity-std 10 --bc-std 0.005 --wind-std 2
+========================================+
|      MONTE CARLO RESULTS               |
|      2000 simulations                   |
+========================================+
| Mean Range:          614.76 m          |
| Std Dev Range:        48.64 m          |
| Mean Impact Vel:     306.33 m/s        |
| Std Dev Velocity:     14.95 m/s        |
| CEP (arrivals):        0.79 m          |
| Target Shortfall:      99.5 %          |
| Hit Probability:        0.5 %          |
+========================================+
```

Three things about this command differ from every other solver in the engine, and all three are visible above.

First, `monte-carlo` has **no** `--drag-model` flag — it runs the G1 default, always. Second, it has no `--auto-zero` and no `--sight-height`; you must supply a launch angle with `-a` directly. Third, **it reports in metres and metres per second regardless of `--units`**: we asked in imperial and got 614.76 m. The `-o statistics` surface is honest about that in its own way — it emits a header row (`range_min,range_max,range_mean,range_std,vel_min,...`) and one row of bare numbers (`480.29,748.54,612.88,46.99,...`) with no unit names attached at all.

The 99.5% shortfall is the ground-impact rule again: with a five-foot bore height and no correction, most of these bullets are on the dirt before 800 yards. Read `Target Shortfall` before you read `Hit Probability`.

The `--wez` mode sweeps hit probability against range instead of reporting one target, and attributes each miss to its dominant cause. Its header line is the most instructive thing it prints. Given `--wind-call-error 3` on top of the default 1 mph wind variability, it reports: "wind call 3.00 mph + wind std 1.00 mph (quadrature) = 3.16 mph effective." Errors add in quadrature, not linearly — which is why halving your largest error source is worth far more than eliminating your smallest.

17.7.5 `stability` — Miller, standalone

Listing 17.26: Gyroscopic stability and minimum twist

```
$ ballistics stability -l 1.24 -t 8 -m 175 -d 0.308 -v 2700

+========================================+
|         Stability Analysis             |
+========================================+
| Stability Factor (Sg):          3.80   |
| Status:                Fully Stable    |
| Twist Rate:                 1:8" RH    |
| Min Twist (Sg=1.5):         1:12.7"    |
| Min Twist (Sg=1.0):         1:15.6"    |
| Bullet Length:            1.240   "    |
| Muzzle Velocity:           2700 fps    |
+========================================+
```

Gyroscopic stability factor (Sg)

The Miller formula, $S_g = 30m/(t^2 d^3 l(1+l^2))$ with imperial inches and grains, compares the bullet's spin-derived gyroscopic stiffness to the overturning aerodynamic moment. Below 1.0 the bullet tumbles; below about 1.4 it is marginal and its BC degrades; 1.5 is the conventional design floor. The value the engine reports is a *muzzle* Sg — it rises downrange, because velocity decays faster than spin does.

That 3.80 is worth comparing with the 3.96 the `trajectory` run printed for the same load. They differ because `trajectory` was not given a bullet length and estimated one from mass and diameter, while `stability` was told 1.24 inches. Length is the term Sg is most sensitive to.

17.8 Standalone calculators

Four commands run no trajectory at all.

The linear model is the familiar one: `powder -v 2750 --temperature 20 --powder-temp 70 --powder-temp-sensitivity 1.2 -m 175` reports "Resolved velocity: 2690.0 fps (-60.0)" — 50 degrees below the reference temperature at 1.2 ft/s per degree. A measured curve is better:

Listing 17.27: Powder temperature, measured curve with a sweep

```
$ ballistics powder -v 2700 --powder-temp-curve "40:2620,70:2700,100:2760" \
    --sweep "40:100:20"
Powder Temperature Velocity
===========================
  Model:              measured curve, 3 points (40.0-100.0 degF)
  Nominal velocity:   2700.0 fps

   Temp (degF)  Velocity (fps)   Shift (fps)
        40.0          2620.0         -80.0
        60.0          2673.3         -26.7
        80.0          2720.0          20.0
       100.0          2760.0          60.0
```

A measured curve always beats the linear model when both are supplied, and it is **clamped at its endpoints** — no extrapolation past the coldest or hottest knot you measured. That is the conservative choice and the right one; a powder's temperature response is not linear and is least linear at the extremes.

Listing 17.28: Free recoil from a SAAMI momentum balance

```
$ ballistics recoil -b 175 -c 43 -v 2700 -f 11 --firearm-type rifle
Free Recoil (SAAMI Momentum Balance)
====================================
  Firearm weight:      11.00 lb
  Bullet weight:       175.0 gr
  Charge weight:       43.0 gr
  Muzzle velocity:     2700.0 fps
  Gas velocity model:  saami-rifle  (4725.0 fps)

  Recoil velocity:     8.78 fps
  Recoil energy:       13.16 ft-lb
  Recoil impulse:      3.000 lb-s
```

Note the gas term: the propellant leaves at a SAAMI-convention 1.75 times muzzle velocity for a rifle — 4725 ft/s here — and 43 grains of it carries real momentum. Omit the charge weight and you understate recoil by roughly a fifth.

The third, `power-factor`, is one line of arithmetic with a rulebook attached: 175 gr at 2700 ft/s scores 472, comfortably over both the USPSA Minor (125) and Major (165) floors. The fourth, `drag-curve`, prints the reference drag tables themselves; it belongs with the rest of the bundled data, so it is covered in Chapter 19.

17.9 Reticles and BDC

Four subcommands treat the reticle as a first-class object with its own file format. The workflow is a genuine round trip, and running it end to end is the best way to see the pieces fit.

Step one: solve the drops. We already have them — the come-up table in Section 17.5 gives 0.557, 1.334, 2.240, 3.268 and 4.430 mil at 200 through 600 yards.

Step two: build a reticle description from them. The `bdc` generator deliberately runs no solve of its own, so the ladder's provenance stays with whoever produced the drops:

Listing 17.29: Generating a BDC ladder

```
$ ballistics reticle generate bdc --name "175gr-BDC" \
    --drop 200:0.557 --drop 300:1.334 --drop 400:2.240 \
    --drop 500:3.268 --drop 600:4.430
Reticle: 175gr-BDC
Focal plane: FFP  Marks: 6

   # Kind    Down(mil)  Right(mil)  Label
---- ------- ---------- ----------- --------------------
   0 center       0.000       0.000  -
   1 hash         0.557       0.000  200 yd
   5 hash         4.430       0.000  600 yd
```

(Marks 2 through 4 are elided.) Add `-o json` and you get the interchange format that `reticle hold`, `mark-to-range`, and `profile save --reticle-json` all consume:

Listing 17.30: The reticle description schema (abridged)

```
{
  "name": "175gr-BDC",
  "focal_plane": "ffp",
  "reference_magnification": 1.0,
  "marks": [
    { "down_mil": 0.0,   "right_mil": 0.0, "kind": "center" },
    { "down_mil": 0.557, "right_mil": 0.0, "kind": "hash", "label": "200 yd" },
    { "down_mil": 1.334, "right_mil": 0.0, "kind": "hash", "label": "300 yd" },
    { "down_mil": 2.24,  "right_mil": 0.0, "kind": "hash", "label": "400 yd" },
    { "down_mil": 3.268, "right_mil": 0.0, "kind": "hash", "label": "500 yd" },
    { "down_mil": 4.43,  "right_mil": 0.0, "kind": "hash", "label": "600 yd" }
  ]
}
```

Step three: place a firing solution in it. `reticle hold` takes a drop and wind hold and names the nearest mark:

Listing 17.31: Reading a solution off the reticle

```
$ ballistics reticle hold --reticle-json 175gr-bdc.json \
    --drop-mil 3.9 --wind-mil -0.9 --mag 10
Reticle Hold Point
==================

Reticle:          175gr-BDC
Focal plane:      FFP
Magnification:    10.00x (FFP: subtensions are magnification-independent)

Hold down:     3.900 mil
Hold right:   -0.900 mil

Nearest mark:     #5 hash (600 yd)
  at (down 4.430, right 0.000) mil true
  distance from hold: 1.044 mil

WARNING: the hold falls outside the marked area of this reticle (dial instead,
or use a reticle with more holdover).
```

The warning is doing real work: the wind hold of 0.9 mil left has nothing to reference, because this ladder has no windage marks. The nearest mark is 1.044 mil away in the combined metric.

Step four: close the loop. `mark-to-range` takes the reticle and the load and reports where each mark actually lands:

Listing 17.32: Mapping marks back to ranges

```
$ ballistics mark-to-range --reticle-json 175gr-bdc.json \
    -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --zero-distance 100 --sight-height 1.5 --max-range 1200
Mark-to-Range
=============

Reticle:      175gr-BDC
Focal plane:  FFP
Zero:         100 yd

Note: 1 mark(s) excluded - the optical center and off-axis windage marks do not
map to a range.

Mark(mil)  True(mil)  Range(yd)  Time(s)  Vel(fps)  Energy(ft-lb)
---------  ---------  ----------  -------  ----------  ------------
     0.56      0.557         200    0.239        2334          2116
     1.33      1.334         300    0.373        2162          1817
     2.24      2.240         400    0.517        1999          1552
     3.27      3.268         500    0.673        1841          1317
     4.43      4.430         600    0.843        1690          1109
```

Every mark comes back to the range it was built from. That is the round trip closing, and it is the sanity check to run whenever you suspect a BDC reticle, an atmosphere change, or a load change.

17.9.1 `bdc-match` — the magnification that makes an SFP reticle fit

A second-focal-plane reticle's subtensions scale with magnification, which means a BDC ladder etched for one load can be made to fit another by turning the power ring. `bdc-match` solves for that magnification:

Listing 17.33: Fitting an SFP BDC ladder to this load

```
$ ballistics bdc-match -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
    --zero-distance 100 --sight-height 1.5 --reference-mag 10 \
    --mark-range 1:300 --mark-range 2:400 --mark-range 3:500
BDC Magnification Match
=======================

Reference magnification: 10.00x
Fitted magnification:    8.963x
Subtension scale at fit: 1.1157x
Zero:                    100 yd

Mark(mil)  Range(yd)  Apparent(mil)  Drop(mil)  Residual(mil)
---------  ----------  -------------  ---------  -------------
     1.00         300          1.116      1.334         -0.218
     2.00         400          2.231      2.240         -0.009
     3.00         500          3.347      3.268          0.079

Worst residual: 0.218 mil   RMS: 0.134 mil

WARNING: the worst residual exceeds 0.200 mil - this reticle does not fit this
load well at ANY magnification. Treat the fitted value as the least-bad
compromise, not a BDC calibration.
```

At 300 yards, 0.218 mil is 2.4 inches — and the command says so rather than quietly handing back a number. Ask it for a first-focal-plane reticle and it refuses on principle:

Listing 17.34: An FFP reticle has nothing to fit

```
$ ballistics bdc-match ... --focal-plane ffp
Error: "bdc-match applies to SECOND focal plane reticles only: an FFP reticle's
subtensions are the same at every magnification, so no magnification makes it
match a load better than any other. Use `mark-to-range` to see where its marks
land instead."
```

That is an unusually good error message: it explains the physics, and it names the command you actually wanted.

17.10 Housekeeping

17.10.1 `profile`

Profiles are the engine's persistence layer — one JSON file per named load under `~/.ballistics/profiles/`. They get a chapter of their own (Chapter 19); here it is enough to know that `profile save`, `list`, `show`, `delete`, `import`, and `zero-set` exist, and that saving one looks like this:

Listing 17.35: Saving a profile

```
$ ballistics profile save LABBOOK --velocity 2700 --bc 0.243 --mass 175 \
    --diameter 0.308 --drag-model g7 --twist-rate 8 --sight-height 1.5 \
    --zero-distance 100 --bullet-length 1.24 \
    --elevation-click 0.1mil --windage-click 0.1mil
Profile 'LABBOOK' saved to "/Users/alexjokela/.ballistics/profiles/LABBOOK.json"
```

17.10.2 `completions`

Shell completions for `bash`, `elvish`, `fish`, `powershell`, and `zsh`, written to stdout. Redirect `ballistics completions zsh` into your fpath and be done.

17.10.3 `mcp` — the engine as a tool server

`ballistics mcp` runs a Model Context Protocol server over stdio, exposing exactly two tools: `solve` (which takes a solve-json v1 request object as its arguments) and `engine_info`. Driving it by hand with a JSON-RPC handshake and one tool call:

Listing 17.36: The MCP handshake and `engine_info`

```
--> {"jsonrpc":"2.0","id":1,"method":"initialize","params":{
      "protocolVersion":"2024-11-05","capabilities":{},
      "clientInfo":{"name":"book","version":"1"}}}
<-- {"id":1,"jsonrpc":"2.0","result":{"capabilities":{"tools":{}},
      "protocolVersion":"2024-11-05",
      "serverInfo":{"name":"ballistics-engine","version":"0.30.1"}}}

--> {"jsonrpc":"2.0","id":2,"method":"tools/call",
     "params":{"name":"engine_info","arguments":{}}}
<-- {"id":2,"jsonrpc":"2.0","result":{"content":[{"text":
      "{\"drag_models\":[\"G1\",\"G6\",\"G7\",\"G8\"],
        \"engine_version\":\"0.30.1\",
        \"features\":[\"online\",\"pdf\",\"profile-import\"]}",
      "type":"text"}],"isError":false}}
```

Note what `engine_info` reports: the drag models available over the machine interface are **G1, G6, G7, G8 only**, even though the CLI's `--drag-model` accepts nine. The narrow set is the `solve-json` contract, and the next chapter is entirely about it.

17.10.4 `generate-bc-segments`

One last utility: it turns a single BC into a velocity-banded ladder you can paste back in as `--bc-segment` arguments. For our 175 gr G7 at BC 0.243 it emits five bands — 0.2430 above 2800 ft/s, then 0.2402, 0.2365, 0.2329, and 0.2292 below 1600 ft/s. That is a 5.7% decline from muzzle to subsonic, about what a modern match bullet really does. Velocity-banded BCs are **not** part of `solve-json` v1, so the Ballistics Lab cannot use them.

17.11 Scripting the engine

If you are going to drive the engine from a program — as the Ballistics Lab does, and as you might from a shell script — three facts matter more than any flag.

Exit status is meaningful. For `solve-json` specifically, the mapping is exact and is verified in Chapter 18: 0 for a success envelope, 2 for a schema or validation failure, 3 for a resource limit or a solve failure, 1 for I/O or an internal fault. The human subcommands use 2 for a usage error.

Machine surfaces do not change with `--units`, except when they do. `solve-json` is always SI and ignores `--units` entirely. `monte-carlo` ignores it too, but reports metric anyway, which is not the same thing. `-o json` and `-o csv` on the other commands *do* follow it.

Some flags never follow `--units` at all. `--sample-interval` is always metres. The optional fourth field of `--wind-segment` is always metres per second even though the first field in the same token is mph under imperial. The `RIGHT` and `SIGMA` fields of `true-wind --miss` are always inches. These are not oversights — each is a measurement that has a natural unit — but they will surprise you exactly once.

If you are automating, prefer `solve-json`

Every human-facing subcommand in this chapter is a convenience wrapper with its own flag vocabulary, its own defaults, and its own idea of which physics toggles it exposes. The `solve-json` interface has one vocabulary, one set of defaults, and it echoes every default it applied back to you. If your program needs a trajectory rather than a card, use that door. That is the whole reason the Ballistics Lab does.

The next chapter opens that door and reads the contract on it, line by line.

Chapter 18

The solve-json v1 Wire Contract

Every number the Ballistics Lab has ever shown you crossed a process boundary as text. There is no shared library, no foreign function interface, no C ABI, no dynamic loading. One Lattice process writes a JSON document to a child process's standard input; the child writes a JSON document back and exits. That document pair is **solve-json v1**, and this chapter is its specification, its evidence, and its edges.

What was designed and what shipped

The original design note for the Ballistics Lab proposed a native Rust dynamic library loaded into the Lattice runtime as an extension, with the ballistics solver callable as an intrinsic. That is **not** what shipped. The Ballistics Lab drives the engine as an ordinary child process through `exec_argv()`, speaking solve-json v1 over pipes, one process per solve. The dylib remains an unbuilt alternative — and, as this chapter's determinism results suggest, the subprocess turned out to buy a great deal for what it costs.

18.1 Why the engine has two front doors

Chapter 17 walked thirty subcommands, and every one of them is a convenience wrapper: its own flag names, its own defaults, its own subset of the physics, its own rendering. `trajectory` spells a zero `--auto-zero`; `come-ups` spells it `--zero-distance`. `monte-carlo` has no drag-model flag at all. `zero` silently defaults to G1. Excellent tools for a person. Impossible ground to build a library on.

So the engine grew a second door with exactly the opposite properties:

- **One vocabulary.** Nine top-level members, all required, all named.
- **One unit system.** SI, marked in every field name.

- **No hidden defaults.** Every default the engine applies is echoed back to you as a concrete value with a reason attached.
- **No filesystem, no network, no profiles.** Nothing enters the solve that was not in the request.
- **Closed schema.** An unrecognised field is an error, not a shrug.

Listing 18.1: The whole interface

```
$ ballistics solve-json < request.json > response.json
```

One document in on stdin, **one compact JSON envelope plus a newline** out on stdout, and an exit status. That is the entire transport.

18.2 Transport and its limits

Three limits are enforced before any physics runs, and all three are verifiable from the command line.

Input size: 1 048 576 bytes, and the exact limit is accepted. We padded a valid minimal request with whitespace to exactly one mebibyte and then to one byte more:

Listing 18.2: The input cap, probed from both sides

```
$ for f in pad_exact.json pad_over.json; do
    ballistics solve-json < $f > $f.out
    echo "$f exit=$? bytes_in=$(wc -c < $f)"; head -c 100 $f.out; echo
  done
pad_exact.json exit=0 bytes_in= 1048576
{"schema_version":1,"engine_version":"0.30.1","status":"ok","resolved_request":
pad_over.json exit=3 bytes_in= 1048577
{"schema_version":1,"status":"error","error":{"code":"resource_limit","message":
```

Truncated at 100 bytes for the page; the over-limit envelope reads in full:

```
{"schema_version":1,"status":"error","error":{"code":"resource_limit",
 "message":"solve-json input exceeds the 1048576-byte limit",
 "path":null,"line":null,"column":null}}
```

Sample count: 10 000 observations. The limit is evaluated against the range the bullet *actually reaches*, and the service errors rather than thinning the grid:

Listing 18.3: Asking for a 2 cm sampling interval over 1000 m

```
{"schema_version":1,"status":"error","error":{
  "code":"resource_limit",
  "message":"trajectory observation limit of 10000 exceeded (would produce 50001)",
  "path":"$.sampling.interval_m","line":null,"column":null}}
```

Exit status is part of the contract. Every code we could produce is listed in Table 18.1, and every row was captured by running the binary.

Table 18.1: solve-json exit status, v0.30.1, all rows verified

Status	Meaning	Error codes that produce it
0	Success envelope	—
1	I/O or internal fault	`io_error`, `internal_error`
2	Schema, shape, or semantic failure	`invalid_json`, `unsupported_schema_version`, `unknown_field`, `missing_field`, `invalid_value`, `conflicting_fields`
3	Resource limit or solve failure	`resource_limit`, `solve_failed`

The engine's own v1 document gets one of these rows wrong

`docs/SOLVE_JSON_V1.md` in the engine tree places `solve_failed` at exit 2. Running it says otherwise:

```
$ ballistics solve-json < zerofail.json > /dev/null 2>&1 ; echo $?
3
```

The Ballistics Lab agrees with the binary, not the document: `process_backend.lat` groups `resource_limit` and `solve_failed` together at exit 3. When the docs and the binary disagree, run the binary.

18.3 The request

18.3.1 Nine members, all required

A v1 request is a JSON object with exactly nine top-level members: `schema_version` plus eight sections — `projectile`, `rifle`, `shot`, `atmosphere`, `wind`, `solver`, `effects`, `sampling`. Sections may be empty objects, but they may not be absent. Omitting one is a `missing_field` error naming the exact JSON path.

Discovering that by probing is instructive: send `{}` and the engine asks for `$.schema_version`; supply it and it asks for `$.projectile`; supply an empty projectile and it asks for `$.rifle`. It walks the schema in declaration order and stops at the first gap.

Here is the smallest request that solves:

Listing 18.4: A minimal valid v1 request

```
{
  "schema_version": 1,
  "projectile": { "mass_kg": 0.01134, "diameter_m": 0.00782,
                  "drag_model": "G7", "ballistic_coefficient": 0.243 },
  "rifle":      { "muzzle_velocity_mps": 823.0 },
  "shot":       { "max_range_m": 300.0 },
  "atmosphere": {},
  "wind":       {},
  "solver":     {},
  "effects":    {},
  "sampling":   { "interval_m": 100.0 }
}
```

Four fields in `projectile`, one in `rifle`, one in `shot`. Everything else has a documented default, and every default that gets applied will come back to you in the response.

18.3.2 SI is in the name

Every dimensional field carries its unit as a suffix: `_kg`, `_m`, `_mps`, `_j`, `_s`, `_pa`, `_k`, `_rad`. There is no unit negotiation and `--units` has no effect on this command. `relative_humidity` is a bare fraction from 0 to 1 — not a percent — and that is one of the two places a careless caller silently loses.

The three hard rules on values

Numbers must be finite. NaN and infinity are rejected. **Explicit `null` is invalid everywhere**, and reports as `invalid_value` with a message of "expected a number" — omission and null are not the same thing. **Every object rejects unknown fields.** A typo does not fall through to a default; it stops the solve and names itself.

18.3.3 Field reference

Table 18.2 and Table 18.3 give the fields for v0.30.1. Anything not listed is rejected.

Wind is the one section with two shapes, and never both. **Constant** wind takes `speed_mps` and `direction_from_rad` — which must be supplied *together* — and an optional `vertical_speed_mps`.

Table 18.2: `projectile` and `rifle`

Field	Default	Notes
projectile		
`mass_kg`	required	
`diameter_m`	required	
`drag_model`	required	**G1, G6, G7, G8 only**. G2/G5/GI/GS are rejected, not aliased
`ballistic_coefficient`	required	referenced to the named model
`length_m`	estimated	from mass and diameter; emits `estimated_projectile_length`
rifle		
`muzzle_velocity_mps`	required	lives on `rifle`, not `shot`
`sight_height_m`	0.05	above bore
`muzzle_height_m`	0	bore above the ground datum
`twist_rate_m_per_turn`	0.3048	= 1:12 inch
`twist_direction`	`"right"`	`"left"` or `"right"`
`sight_offset_lateral_m`	0	positive = sight right of bore; $\lvert v \rvert < 0.5$ m

Table 18.3: `shot`, `atmosphere`, `solver`, `effects`, `sampling`

Field	Default	Notes
shot		
`max_range_m`	required	requested termination distance
`zero_distance_m`	absent	solve muzzle elevation for this zero
`muzzle_angle_rad`	absent	supply elevation directly
`aim_azimuth_rad`	0	horizontal aim offset in the sight frame
`shot_azimuth_rad`	0	compass bearing; 0 = north
`shooting_angle_rad`	0	uphill / downhill line-of-sight angle
`cant_angle_rad`	0	clockwise positive from the shooter
`target_height_m`	0	**absolute** world height above the ground datum, not relative to the muzzle
`ground_threshold_m`	-100	stop below this height
`zero_poi_up_m`	0	Kestrel "zero height"; $\lvert v\rvert < 1$ m
`zero_poi_right_m`	0	Kestrel "zero offset"
`drops_reference`	`"los"`	`"los"` or `"target"`; output mode only
atmosphere		
`altitude_m`	0	station altitude
`temperature_k`	ICAO at altitude	authoritative when present
`pressure_pa`	ICAO at altitude	authoritative station pressure
`relative_humidity`	0.5	**fraction 0–1**
`latitude_rad`	absent	*required* when Coriolis is enabled
solver / effects / sampling		
`solver.method`	`"rk45"`	`rk45`, `rk4`, `euler`
`solver.time_step_s`	0.001	ignored by rk45, with a warning
`effects.magnus`	false	warns `experimental_effect`
`effects.coriolis`	false	requires `latitude_rad`
`effects.enhanced_spin_drift`	false	conflicts with `magnus`
`sampling.interval_m`	10	max 10 000 samples

Segmented wind takes a `segments` array whose `until_distance_m` boundaries must strictly increase. An empty `wind` object is still air.

Raise the muzzle, raise the target

`shot.target_height_m` is an **absolute** height above the same ground datum as `rifle.muzzle_height_m`, not an offset from the bore. Set `muzzle_height_m: 1.5` with a 100 m zero and leave `target_height_m` at its default 0 and the zero search is being asked to put the bullet on the ground at 100 m from a muzzle 1.5 m up. It fails, honestly and unhelpfully:

```
{"code":"solve_failed",
 "message":"Zero angle did not converge: residual height error too large
           (target not reachable / not bracketed)",
 "path":null,"line":null,"column":null}
```

Set `target_height_m: 1.5` and it solves immediately. The same trap bites inclined shots: a 10-degree uphill zero at 91.44 m needs `target_height_m` of 15.878, not 0.

18.4 How the BALLISTICS LAB builds a request

On the LATTICE side, exactly one module knows the wire format: `protocol_v1.lat`, 230 lines, importing nothing. Its whole job is to turn a validated `Experiment` into the request `Map`, and its header comment states the rule it exists to enforce:

> *Every dimensional value is read from a quantity's canonical SI field. Display values and units never enter the wire request.*

Every one of the module's ten adapter functions is the same three-line shape:

Listing 18.5: `protocol_v1.lat` — the projectile adapter

```
export fn projectile_to_v1_map(projectile: any) -> Map {
    _require_struct(projectile, "Projectile", "experiment.projectile")
    flux out = Map::new()
    out.set("mass_kg", projectile.mass.si_value)
    out.set("diameter_m", projectile.diameter.si_value)
    if projectile.length != nil {
        out.set("length_m", projectile.length.si_value)
    }
    out.set("drag_model", projectile.drag_model)
    out.set("ballistic_coefficient", projectile.ballistic_coefficient)
    return freeze(out)
}
```

Three details in eleven lines are worth naming, because each is a protocol decision, not a coding style.

Only `si_value` is ever read. A `Distance` in the BALLISTICS LAB carries `si_value`, `display_value` and `display_unit`. Two of those three exist purely for the human, and the adapter cannot reach for them by accident because it never writes a display field's name.

Absent means omitted, not null. `length_m` is set only when the projectile has one. Contrast `persistence.lat`, the BALLISTICS LAB's save-file module, which does the opposite — it serializes absent optionals as explicit `null`. Two modules, two correct answers, because they are writing to two different contracts. The wire says "omit and I will default"; the save file says "say what you meant so a reader can tell."

One `freeze` at the end. The Map is built **flux** and crystallized once when it is complete.

The one shape mismatch between the BALLISTICS LAB's domain and the wire is muzzle velocity: the BALLISTICS LAB keeps it on the `Shot`, because that is where a shooter thinks it lives, while the wire keeps it on `rifle`. The adapter resolves it by taking both:

Listing 18.6: The rifle adapter takes the shot too

```
export fn rifle_to_v1_map(rifle: any, shot: any) -> Map {
    _require_struct(rifle, "Rifle", "experiment.rifle")
    _require_struct(shot, "Shot", "experiment.shot")
    flux out = Map::new()
    out.set("muzzle_velocity_mps", shot.muzzle_velocity.si_value)
    ...
```

Wind gets the module's only branch, because it is the only section with two shapes. The rule is enforced structurally: if the first `Wind` has no `until_distance` it must be the only one, and if it has one then every wind must, with strictly increasing boundaries. Mixing them is an assertion, not a wire error.

Listing 18.7: Segment boundaries must increase

```
    for wind in winds {
        _require_struct(wind, "Wind", "experiment.winds[]")
        if wind.until_distance == nil {
            assert(false, "experiment.winds: cannot mix constant and segmented wind")
        }
        if wind.until_distance.si_value <= last_boundary {
            assert(false, "experiment.winds: segment boundaries must increase
    strictly")
        }
        last_boundary = wind.until_distance.si_value
        segments.push(_wind_segment_to_v1_map(wind))
    }
```

Finally, `experiment_to_v1_request` assembles the nine members in a fixed order and `v1_request_json` serializes. That is the entire LATTICE-to-wire path: no I/O, no process, no string templating.

18.5 The success envelope

A successful response has exactly eight top-level members — and a ninth, `reticle_hold`, only if the request carried a `reticle` block.

Listing 18.8: The envelope, pretty-printed (the wire form is one compact line)

```
{
  "schema_version": 1,
  "engine_version": "0.30.1",
  "status": "ok",
  "resolved_request": { ... nine members, every default made concrete ... },
  "assumptions": [ { "code": ..., "message": ..., "path": ... }, ... ],
  "warnings":    [ ... ],
  "summary":     { ... },
  "samples":     [ { "distance_m": ..., "flags": [...] }, ... ]
}
```

18.5.1 `resolved_request` — the reproducibility record

This is the member that makes the protocol worth building. It is **not** an echo of your input. It is a distinct type in which every documented default has been materialised as a concrete value. Send the minimal request from Section 18.3 and the resolved request that comes back has a sight height of 0.05, a twist rate of 0.3048, a temperature of 288.15 K, a pressure of 101325 Pa, still air, rk45 at a 0.001 s step, and all three effects off — none of which you typed.

Values that are semantically inapplicable stay absent rather than being invented. The minimal request's resolved form has no `latitude_rad` (Coriolis is off) and no `length_m` under `projectile` in some configurations. Absence is information here, not an omission.

Archive the resolved request, not your input

Your input is what you meant. The resolved request is what ran. If you keep one of them next to a set of impacts, keep the second one. The BALLISTICS LAB's `VerifiedRun` artifact does exactly that — it embeds the resolved request, validated a second time on the way in (Chapter 24).

18.5.2 Assumptions and warnings

Both are arrays of `{code, message, path}` objects emitted in deterministic request-field order. The distinction is real:

An assumption says "you did not tell me, so I chose." Four codes exist: `default_applied`, `icao_standard_temperature`, `icao_standard_pressure`, and `estimated_projectile_length`.

A warning says "you told me, and something about it deserves your attention." Three are reachable in v0.30.1: `partial_wind_coverage`, `experimental_effect`, and `rk45_time_step_ignored` — and those are exactly the three the BALLISTICS LAB's decoder allows.

A fourth, `zero_distance_elevation_not_resolved`, appears in the engine's documentation for the case where a request supplies both a zero distance and an explicit muzzle angle. It is unreachable here, because v0.30.1 treats that combination as an error rather than a warning:

Listing 18.9: Zero distance and muzzle angle together, v0.30.1

```
{"schema_version":1,"status":"error","error":{"code":"conflicting_fields",
 "message":"zero_distance_m and muzzle_angle_rad cannot both be supplied",
 "path":"$.shot","line":null,"column":null}}
```

The BALLISTICS LAB's own `shot()` constructor enforces the same exclusion one layer earlier, so the request never leaves the process.

Here is a real response's warning block — a segmented wind deck that stops short of the solve range, an explicit time step under the adaptive integrator, and an experimental effect enabled:

Listing 18.10: Three warnings from one solve, captured live on v0.30.1

```
"warnings": [
  { "code": "partial_wind_coverage",
    "message": "Segmented wind ends at 800.000000000000 m; wind is calm from
               that boundary through the required 1000.000000000000 m
               solve/zero range.",
    "path": "$.wind.segments" },
  { "code": "rk45_time_step_ignored",
    "message": "Adaptive RK45 owns its time step; the explicitly supplied fixed
               time step is retained in resolved_request but ignored by
               integration.",
    "path": "$.solver.time_step_s" },
  { "code": "experimental_effect",
    "message": "The enhanced spin drift effect is an opt-in experimental v1
               model.",
    "path": "$.effects.enhanced_spin_drift" }
]
```

Read the second one carefully. The engine did not discard your time step — it kept it in the resolved request, so the record of what you asked for survives — and it did not silently honour it either. It told you which of those two things happened. That is the tone of the whole protocol.

18.5.3 `summary` and `samples`

The summary always carries `actual_range_m`, `maximum_height_m`, `time_of_flight_s`, `terminal_speed_mps`, `terminal_energy_j`, and `termination`. Three more appear conditionally: `stability_factor` when computable, `spin_drift_m` *only* when enhanced spin drift is on, and `equivalent_horizontal_range_m` on an inclined shot.

`spin_drift_m` is already inside `windage_m`

Do not add them. In calm air the terminal `windage_m` equals `summary.spin_drift_m` exactly, because spin drift is folded into the lateral solution rather than reported alongside it.

`maximum_height_m` deserves one sentence of its own: it is the greatest *world-vertical* height above the same datum as `rifle.muzzle_height_m`. It is not the shot-frame Y, and it is not height above the line of sight. On a flat shot from a zero-height muzzle it is essentially the max ordinate; on an inclined shot it is dominated by the inclination.

Each sample is `{distance_m, time_s, speed_mps, energy_j, drop_m, windage_m, mach, flags}`. `drop_m` is **positive below the line of sight**; `windage_m` is **positive to the shooter's right**. Flags come from a closed set: `transonic` (Mach 0.8 to 1.2 inclusive), `subsonic` (below Mach 1.0 — so both appear together from 0.8 to just under 1.0), `terminal`, and `ground_threshold`.

The terminal sample always appears exactly once, and last, even when it falls off the interval grid. That guarantee is what lets a consumer treat the last element as the impact without searching. The Ballistics Lab does more than trust it — it checks it, as we will see in Section 18.9.

18.6 The error envelope

An error response is always exactly this shape, and nothing else:

Listing 18.11: The error envelope

```
{"schema_version":1,"status":"error",
 "error":{"code":"...","message":"...",
          "path":"..."|null,"line":N|null,"column":N|null}}
```

`path` and `(line, column)` are mutually exclusive

Structurally valid JSON that fails validation reports a JSON `path` with null line and column. Malformed JSON reports 1-based `line` and `column` with a null path — and never a zero column; serde's column-0 end-of-file case is normalised to 1. Errors with no location at all use null for all three. The Ballistics Lab's `laboratory_error` constructor encodes the same rule as a domain invariant: supply `line` and `column` together, and never alongside a `path`.

Every code below was produced by feeding a probe to the v0.30.1 binary; the envelopes are verbatim.

Listing 18.12: Malformed input — `invalid_json`, exit 2

```
$ printf 'hello' | ballistics solve-json
{"schema_version":1,"status":"error","error":{"code":"invalid_json",
 "message":"expected value at line 1 column 1","path":null,"line":1,"column":1}}

$ printf '{"schema_version":1,' | ballistics solve-json
{"schema_version":1,"status":"error","error":{"code":"invalid_json",
 "message":"EOF while parsing a value at line 1 column 20",
 "path":null,"line":1,"column":20}}
```

Listing 18.13: Schema failures — all exit 2

```
missing_field
{"code":"missing_field","message":"missing required field `schema_version`",
 "path":"$.schema_version","line":null,"column":null}

unsupported_schema_version
{"code":"unsupported_schema_version",
 "message":"unsupported schema_version 2; expected 1",
 "path":"$.schema_version","line":null,"column":null}

unknown_field   (we typed "balistic_coefficient")
{"code":"unknown_field","message":"unknown field `balistic_coefficient`",
 "path":"$.projectile.balistic_coefficient","line":null,"column":null}

invalid_value   (enum)
{"code":"invalid_value","message":"invalid value `G5`; expected one of G1, G6, G7, G8",
 "path":"$.projectile.drag_model","line":null,"column":null}

invalid_value   (explicit null)
{"code":"invalid_value","message":"expected a number",
 "path":"$.projectile.mass_kg","line":null,"column":null}

invalid_value   (cross-field)
{"code":"invalid_value",
 "message":"latitude_rad is required when the Coriolis effect is enabled",
 "path":"$.atmosphere.latitude_rad","line":null,"column":null}
```

`conflicting_fields` is the code for two inputs that are each legal alone. All three we could reach:

Listing 18.14: `conflicting_fields`, exit 2

```
{"code":"conflicting_fields",
 "message":"magnus and enhanced_spin_drift cannot both be enabled",
 "path":"$.effects","line":null,"column":null}

{"code":"conflicting_fields",
 "message":"segments cannot be combined with constant-wind fields",
 "path":"$.wind","line":null,"column":null}

{"code":"conflicting_fields",
 "message":"speed_mps and direction_from_rad must be supplied together",
 "path":"$.wind","line":null,"column":null}
```

That third one is a small design lesson. Supplying a wind speed without a direction is not an underspecified request the engine could default its way out of; it is a request that means nothing. Rejecting it beats guessing zero.

The documented-but-unshipped field

`docs/SOLVE_JSON_V1.md` describes `atmosphere.pressure_reference` in detail — the QNH-versus-station pressure distinction that the CLI already has as `--pressure-type`. Send it to v0.30.1 and the closed schema does its job:

```
{"code":"unknown_field","message":"unknown field `pressure_reference`",
 "path":"$.atmosphere.pressure_reference","line":null,"column":null}
```

The field exists in the engine's working tree, landed after the 0.32.0 tag. It is not in v1 yet, and the document that describes it is describing the future.

18.7 Versioning, and an experiment

The v1 invariants are absolute. Removing a field, changing a unit, changing a sign, renaming an enum value, or changing what a field *means* all require v2. Inside v1, the schema grows only by addition, under two rules:

1. A new **request** field must be optional and must default to the exact pre-existing behaviour.
2. A new **response** field must be **omitted** whenever its feature is inactive, so a response that does not exercise the feature is byte-identical to one from before the field existed.

Explicitly not part of the contract: member order, whitespace, the human-readable `message` strings, `engine_version` itself, and the exact decimal spelling of floating-point numbers. Regression comparisons use $|a - e| \le 10^{-10} + 10^{-9}\max(|a|, |e|)$, and the Ballistics Lab uses precisely that tolerance in `process_backend.lat`.

That is the claim. Here is the test.

18.7.1 Four engine versions, one set of fixtures

The Ballistics Lab ships six conformance fixture pairs in `examples/ballistics_lab/fixtures/conformance/` — a request and the response it produced — covering calm air, a crosswind, a zeroed shot, segmented wind, early termination, and the representative case. They were captured against engines **0.24.1** and **0.26.0**, months before v0.30.1 existed.

We replayed all six requests through the pinned v0.30.1 binary, and then through the 0.32.0 binary as well, and compared each response to its checked fixture with `engine_version` removed:

Listing 18.15: Replaying six checked fixtures against two newer engines

```
$ python3 replay.py     # v0.30.1 on PATH
calm-air          exit=0 live_ev=0.30.1 fixture_ev=0.26.0 bytewise_equal=True tol_diffs=0
crosswind         exit=0 live_ev=0.30.1 fixture_ev=0.26.0 bytewise_equal=True tol_diffs=0
early-termination exit=0 live_ev=0.30.1 fixture_ev=0.24.1 bytewise_equal=True tol_diffs=0
representative    exit=0 live_ev=0.30.1 fixture_ev=0.24.1 bytewise_equal=True tol_diffs=0
segmented-wind    exit=0 live_ev=0.30.1 fixture_ev=0.26.0 bytewise_equal=True tol_diffs=0
zeroed            exit=0 live_ev=0.30.1 fixture_ev=0.26.0 bytewise_equal=True tol_diffs=0

$ python3 replay.py --engine ~/.local/bin/ballistics   # v0.32.0
calm-air             exit=0 live_ev=0.32.0 identical_modulo_version=True
crosswind            exit=0 live_ev=0.32.0 identical_modulo_version=True
early-termination    exit=0 live_ev=0.32.0 identical_modulo_version=True
representative       exit=0 live_ev=0.32.0 identical_modulo_version=True
segmented-wind       exit=0 live_ev=0.32.0 identical_modulo_version=True
zeroed               exit=0 live_ev=0.32.0 identical_modulo_version=True
```

Not close — identical. Not within a tolerance: the same JSON data model, member for member, float for float, including every message string, every assumption, every warning, and every one of the sample arrays. Across four engine versions spanning eight minor releases.

To make sure that is not an artefact of the comparison, here is the representative fixture — captured against 0.24.1 — summarised next to the live 0.30.1 run:

Listing 18.16: The same summary from two engines, four minor versions apart

```
engine_version live: 0.30.1  fixture: 0.24.1
summary live: {"actual_range_m": 50.0, "maximum_height_m": 1.2314367365919499,
  "time_of_flight_s": 0.06191400316677915, "terminal_speed_mps": 792.4430484357767,
  "terminal_energy_j": 3560.5671350304387, "stability_factor": 4.095309067995591,
  "termination": "max_range"}
summary fixt: {"actual_range_m": 50.0, "maximum_height_m": 1.2314367365919499,
  "time_of_flight_s": 0.06191400316677915, "terminal_speed_mps": 792.4430484357767,
  "terminal_energy_j": 3560.5671350304387, "stability_factor": 4.095309067995591,
  "termination": "max_range"}
samples identical: True
resolved_request identical: True
```

What this buys a laboratory

A number in a notebook is only useful if you can get it again. This result means a solve you recorded eighteen months and eight releases ago reproduces bit for bit today from the archived request alone. That is a stronger guarantee than most scientific software offers, and it is available precisely *because* the interface is narrow: no profiles, no files, no network, no ambient state, no locale, no unit switch.

One thing that is not a difference, and one that is

Not a difference: member order and whitespace. Compare envelopes as JSON data, never as strings. **Is a difference:** an explicit value that equals the documented default. Sending `"drops_reference": "los"` or `"wind_reference": "shooter"` solves identically but adds an echo field to `resolved_request`, so the bytes differ. Omission and explicit-default are the same physics and different documents.

18.8 The bounded-request discipline

The engine polices its own inputs. The Ballistics Lab polices them again, before the process is spawned, because a laboratory should not learn about its own runaway request from a child process's exit code.

`process_backend.lat` opens with the numbers:

Listing 18.17: `process_backend.lat` — the bounds, stated once

```
let SCHEMA_VERSION = 1
let MAX_REQUEST_BYTES = 1048576
let MAX_SAMPLES = 10000
let DEFAULT_TIMEOUT_MS = 30000
let DEFAULT_MAX_STDOUT_BYTES = 8388608
let DEFAULT_MAX_STDERR_BYTES = 262144
let SUMMARY_ABSOLUTE_TOLERANCE = 0.0000000001
let SUMMARY_RELATIVE_TOLERANCE = 0.000000001
```

The first two mirror the engine's own caps exactly. The next three are the Ballistics Lab's, and they exist because `exec_argv` is the one place a pure program touches an unbounded outside world: a 30-second wall clock, 8 MiB of stdout, 256 KiB of stderr. All three are overridable per backend, and all three are validated as positive integers when you do.

The request check happens before the spawn:

Listing 18.18: Serialize, measure, then spawn

```
fn _solve_process(executable: String, argv: Array, limits: Map,
                  expected_engine_version: any, experiment: any) -> any {
    let request = try {
        v1_request_json(experiment)
    } catch error {
        return laboratory_error("request_serialization_error",
            "failed to encode solve-json v1 request: " + error.message)
    }
    if len(request) > MAX_REQUEST_BYTES {
        return laboratory_error("resource_limit",
            "solve-json request exceeds the 1048576-byte v1 input limit")
    }
    let result = try {
        exec_argv(executable, argv, request, limits)
    } catch error {
        return laboratory_error("process_error", error.message)
    }
    return decode_v1_process_result(experiment, result, expected_engine_version)
}
```

`exec_argv`, not a shell

The Ballistics Lab never assembles a command *string*. `exec_argv` takes an executable path and an **array of arguments**, which for the ordinary constructor is the fixed literal `["solve-json"]`. There is no shell to quote for, no word splitting, no glob expansion, no environment interpolation, no `$(...)`. An experiment name containing a semicolon is a string in a JSON document on stdin and nothing else. The argv array is defensively copied element by element, with every element type-checked, before the backend closure captures it.

And one process per solve. Nothing is retained between solves — no daemon, no warm pool, no cached solver state. A hung engine costs one 30-second timeout and a `process_error`, not a wedged laboratory.

18.9 Fail closed

Here is the Ballistics Lab's governing decision about the response: **anything the protocol did not promise is an error.** Not a warning, not a best-effort parse — a `LaboratoryError` with the code `invalid_engine_response`.

The enforcement mechanism is one function, used everywhere:

Listing 18.19: The closed-schema enforcer

```
fn _require_exact_keys(object: Map, required: Array, optional: Array, path: String) {
    for key in required {
        if !object.has(key) {
            _fail(path + ": missing required field '" + key + "'")
        }
    }
    for key in object.keys() {
        if !required.contains(key) && !optional.contains(key) {
            _fail(path + ": unknown field '" + key + "'")
        }
    }
}
```

It runs on the envelope, on the error object, on the resolved request and each of its nine sections, on every notice, on the summary, and on every single sample. Alongside it sit an allowlist per notice kind, an allowlist for sample flags, an allowlist for termination reasons, and a check that every number is finite.

Three cross-checks go further than schema validation, and they are the interesting ones:

1. **Exactly one terminal sample, and it must be last.** Not zero, not two, not in the middle.

2. **The terminal sample must agree with the summary**, to $10^{-10} + 10^{-9} \max(|a|, |b|)$, on range, time, speed, and energy. A response whose summary contradicts its own last row is rejected even though every field is individually well-formed.

3. **The error code must match the exit status.** An error envelope claiming `invalid_value` that arrives with exit 3 is not an `invalid_value`; it is an `invalid_engine_response`.

18.9.1 Watching it work

`process_backend.lat` exports its decoder as a pure function, `decode_v1_process_result(experiment, process_result)`, taking a Map of `{stdout, stderr, exit_code}`. That makes the whole policy testable without a process. We fed it six hand-built results:

Listing 18.20: The fail-closed decoder, exercised directly

```
$ clat probe_decode.lat
good response, exit 0        -> Trajectory  engine=0.30.1  samples=6
                                termination=max_range
good response, exit 1        -> LaboratoryError [invalid_engine_response]
   solve-json v1 response: status 'ok' requires exit 0, got 1
good response, stderr banner -> LaboratoryError [invalid_engine_response]
   solve-json v1 response: status 'ok' must not write diagnostics to stderr
error envelope, exit 2       -> LaboratoryError [invalid_value]
   invalid value `G5`; expected one of G1, G6, G7, G8
error envelope, wrong exit   -> LaboratoryError [invalid_engine_response]
   solve-json v1 response: $.error.code 'invalid_value' requires exit 2, got 3
resource_limit, exit 3       -> LaboratoryError [resource_limit]
   trajectory observation limit of 10000 exceeded (would produce 50001)
```

Line three is the one people find surprising, so it is worth dwelling on. A perfectly good success envelope, exit 0, all numbers valid — but the engine also printed something to stderr. The Ballistics Lab rejects it. The reasoning is that a success on this protocol is defined as "one envelope on stdout and nothing else"; a banner, a progress line, or a deprecation notice means the process you spawned is not the process the contract describes. Better to know.

Line four shows the other half: a well-formed *error* envelope is not a failure of the protocol. It decodes cleanly into a typed `LaboratoryError` carrying the engine's own code, message, and path. Errors are values here, all the way down.

18.10 Where fail-closed meets forward compatibility

Now the honest part.

The engine's compatibility rules say, in as many words, that **consumers must tolerate unknown response fields**, and that "a consumer pinning a DTO with `deny_unknown_fields` is pinning an engine version, not the contract." The Ballistics Lab pins a DTO with the moral equivalent of `deny_unknown_fields`, on every object, deliberately.

These two positions cannot both be fully satisfied, and on v0.30.1 they collide in two reachable places.

18.10.1 Collision one: `reticle_hold`

Send a request with a `reticle` block and the success envelope grows a tenth top-level member. The engine considers this rule-compliant, because the field is omitted unless you opt in. Hand that response to the Ballistics Lab's decoder:

Listing 18.21: A reticle response, decoded by the Lab

```
$ clat probe_reticle.lat
typeof   : Struct
struct   : LaboratoryError
code     : invalid_engine_response
message  : assertion failed: solve-json v1 response: $: unknown field 'reticle_hold'
```

This one is arguably fine. The Ballistics Lab's domain has no reticle concept and its `Experiment` cannot produce a `reticle` block, so the response is unreachable from inside the Ballistics Lab. You have to go out of your way.

18.10.2 Collision two: inclined shots

This one you do not have to go out of your way for. The engine's compatibility document names exactly one exception to the omit-unless-active rule: `summary.equivalent_horizontal_range_m` appears on *any* inclined shot, with no opt-in. And the Ballistics Lab's `Shot` constructor has taken a `shooting_angle` since the beginning.

So: a 10-degree uphill shot, set up entirely through the Ballistics Lab's own API, with the target height raised to match as Section 18.3 requires.

Listing 18.22: An inclined shot, end to end, in the Lab

```
sessions.replace_shot(lab, shot(fps(2700), yd(500), yd(100), nil, nil, nil,
    deg(10), nil, m(15.878389365864109), m(-500)))
:solve

Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response:
           $.summary: unknown field 'equivalent_horizontal_range_m'
  exit code: 0
```

The engine solved it correctly. Running the identical request through `solve-json` by hand returns exit 0 and a complete summary:

Listing 18.23: What the engine actually returned

```
{ "actual_range_m": 500.0,
  "maximum_height_m": 84.8169390540018,
  "time_of_flight_s": 0.7512254369002528,
  "terminal_speed_mps": 539.3741894743315,
  "terminal_energy_j": 1649.5420072570919,
  "stability_factor": 1.7547882257792367,
  "equivalent_horizontal_range_m": 494.51708190917975,
  "termination": "max_range" }
```

Inclined shots do not work in the Lab against engine 0.30.1

This is a real, reproducible limitation of the shipped Ballistics Lab, not a misconfiguration. Any `shooting_angle` other than zero produces a summary field the decoder's optional list does not contain (`_decode_summary` allows only `stability_factor` and `spin_drift_m` as optional), and every solve fails with `invalid_engine_response`. The fix is one string in one array in `process_backend.lat`. Until it lands, keep `shooting_angle` at `nil` and use the engine's `trajectory --shooting-angle` directly for inclined work.

18.10.3 Which policy is right?

Both, in their own scope, and the disagreement is worth understanding rather than resolving by fiat.

The engine is a *platform*. Platforms must be able to add. If every consumer hard-fails on a new field, no field can ever be added, and the version number becomes the only way to ship anything.

The Ballistics Lab is an *instrument*. Instruments must be able to say "I do not know what this reading means." A laboratory that silently ignores an unfamiliar field in a response has, by construction, no way to notice when that field is the one that changes the answer.

The reconcilable position — and the one a future v1.1 of the Ballistics Lab should take — is to distinguish **unknown fields inside objects whose values the Ballistics Lab consumes** (still fatal, because the reading may be wrong) from **unknown sibling fields the Ballistics Lab does not read** (record them in the provenance, warn once, continue). That is more code than a policy sentence, which is presumably why the shipped version chose the blunt instrument. It is the right blunt instrument: an instrument that refuses to read is annoying, and an instrument that reads wrong is dangerous.

18.11 The other v1 validator

One more module belongs to this chapter, because it validates the same schema for a completely different reason.

`resolved_request_v1.lat` is 335 lines, imports nothing, and exports a single function:

Listing 18.24: `resolved_request_v1.lat` — the whole public surface

```
// Validate a solve-json v1 resolved request and return a detached immutable
// snapshot. The caller's Map remains independently mutable.
export fn validate_resolved_request_v1(value: any) -> Map {
    let object = _map(value, "$.resolved_request")
    _require_exact_keys(object,
        ["schema_version", "projectile", "rifle", "shot", "atmosphere",
         "wind", "solver", "effects", "sampling"], [], "$.resolved_request")
    ...
    if shot.get("ground_threshold_m") >= rifle.get("muzzle_height_m") {
        _fail("$.resolved_request.shot.ground_threshold_m: must be below the resolved
    muzzle height")
    }
    if effects.get("coriolis") && !atmosphere.has("latitude_rad") {
        _fail("$.resolved_request.atmosphere.latitude_rad: required when coriolis is
    enabled")
    }

    return freeze(clone(object))
}
```

Its nine section validators are near-verbatim twins of the ones inside `process_backend.lat`. That duplication is deliberate and it is documented as such: `process_backend` validates **responses in flight**, and `resolved_request_v1` validates **snapshots at rest**. An artifact you load from disk a year from now must be validated by code that has no idea processes exist, because there is no process — and the artifact path must not depend on the transport path continuing to exist in its current form.

The last line is the one to remember. `freeze(clone(object))`, in that order: clone first to detach from the caller's Map, then crystallize. The caller keeps a mutable Map; the artifact keeps an immutable one; neither can reach the other. Chapter 24 shows what that detachment is for.

18.12 Reading the contract like a lawyer

A working summary, in the order it will matter to you:

- **Nine members, always.** Empty objects are fine; absent ones are not.
- **SI, in the field names.** `relative_humidity` is a fraction, not a percent.
- **Omission is not null.** Leave it out to get the default; sending `null` is an error.
- **Unknown fields are errors** in both directions — the engine rejects unknown request fields, the Ballistics Lab rejects unknown response fields.
- **The resolved request is the record.** Archive it, not your input.
- **Exit status is data.** 0 success, 2 schema, 3 resource or solve, 1 internal.
- **Compare envelopes as JSON, never as bytes** — but expect them to be byte-identical anyway, because on the evidence of four engine versions they are.
- **Do not add `spin_drift_m` to `windage_m`.**
- **Inclined shots break the shipped Ballistics Lab** against v0.30.1. Use the CLI for those until the decoder learns the field.

The next chapter follows the other half of the engine's world — the parts of it that read files: profiles, reference drag curves, and the surprising number of places a working directory can change an answer.

Chapter 19

Profiles, Reference Data, and Where Files Live

Chapter 18 described an interface with no filesystem at all — one document in, one document out, nothing else consulted. That purity is exactly why solve-json is reproducible eight releases later, and it is also why the Ballistics Lab cannot use half the engine's conveniences.

This chapter is about the other half: the files. Where a saved load lives, what is actually in it, which commands read it and which quietly do not, where the reference drag curves come from, and one working-directory behaviour that can change your answer without changing anything you typed.

19.1 Three storage systems, not one

It helps to name them before touching any of them, because they never talk to each other.

1. **The engine's profile store** — `~/.ballistics/`. One JSON file per named load, plus credentials and caches. Written and read only by the engine's human-facing subcommands.

2. **The engine's bundled reference data** — the drag tables, some compiled into the binary and some loaded from disk at runtime. Read-only, and not something you normally think about until it surprises you.

3. **The Ballistics Lab's own documents** — experiment save files and verified run artifacts, written by `persistence.lat` and `verified_artifacts.lat`, in whatever directory you name.

The Lab cannot see engine profiles, and never will

There is no bridge between store one and store three. A profile you saved with `ballistics profile save` is invisible to the BALLISTICS LAB, because `solve-json` does not read the filesystem. This is not an oversight — it is the whole point. If the wire request could name a profile, an archived request would no longer be sufficient to reproduce a run.

19.2 Inside `~/.ballistics/`

The directory is resolved as `home_dir().join(".ballistics")` and created on demand. There is **no** environment-variable override and no per-project store: one location, per user, per machine.

Listing 19.1: The engine's home directory (the `total` and parent-directory lines are elided)

```
$ ls -la ~/.ballistics/
drwxr-xr-x    6 alexjokela  staff   192 Aug  4 20:51 .
-rw-------    1 alexjokela  staff    59 Aug  4 20:51 credentials.toml
drwxr-xr-x    2 alexjokela  staff    64 Jan 26  2026 era5_cache
drwxr-xr-x    2 alexjokela  staff    64 Aug  5 01:31 profiles
-rw-r--r--    1 alexjokela  staff   125 Jan 26  2026 tos.json
```

- `profiles/` — one `<NAME>.json` per saved load.
- `credentials.toml` — the API token for the online reverse solvers, mode `0600`. Only the v0.32.0 binary's `login` writes it.
- `tos.json` — terms acceptance state.
- `era5_cache/` — cached reanalysis weather for the online 3D weather feature.

Nothing in `solve-json`'s path touches any of these. You can delete the whole directory and every number in Chapter 18 still reproduces.

19.3 What a saved profile actually contains

Listing 19.2: Saving a load

```
$ ballistics profile save LABBOOK --velocity 2700 --bc 0.243 --mass 175 \
    --diameter 0.308 --drag-model g7 --twist-rate 8 --sight-height 1.5 \
    --zero-distance 100 --bullet-length 1.24 \
    --elevation-click 0.1mil --windage-click 0.1mil
Profile 'LABBOOK' saved to "/Users/alexjokela/.ballistics/profiles/LABBOOK.json"
```

Listing 19.3: `LABBOOK.json`, complete

```
{
  "name": "LABBOOK",
  "velocity": 2700.0,
  "bc": 0.243,
  "mass": 175.0,
  "diameter": 0.308,
  "drag_model": "G7",
  "twist_rate": 8.0,
  "sight_height": 1.5,
  "zero_distance": 100.0,
  "units": "imperial",
  "temperature": 59.0,
  "pressure": 29.92,
  "humidity": 50.0,
  "altitude": 0.0,
  "bullet_name": null,
  "created": "1785913103",
  "bullet_length": 1.24,
  "elevation_click": "0.1mil",
  "windage_click": "0.1mil"
}
```

19.3.1 Values are stored in the units you typed

That is the single most important line in the file, and it is the `"units"` discriminator. The engine does **not** normalise to SI on the way in. Save the same rifle metrically and you get metric numbers with a different tag:

Listing 19.4: The same rifle, saved with `-u metric`

```
{ "name":"METBOOK", "velocity":823.0, "bc":0.243, "mass":11.34,
  "diameter":7.82, "drag_model":"G7", "twist_rate":203.0,
  "sight_height":38.0, "zero_distance":100.0, "units":"metric",
  "temperature":15.0, "pressure":1013.25, "humidity":50.0,
  "altitude":0.0, "bullet_name":null, "created":"1785914637" }
```

The conversion on the way *out* is correct. A metric-saved profile consumed by an imperial command produces the imperial answer:

Listing 19.5: A metric profile driving an imperial command

```
$ ballistics come-ups --profile METBOOK --zero-distance 100 \
    --start 100 --end 300 --step 100

Come-Up Table (zero: 100 yd, MIL)
+----------+----------+----------+----------+----------+----------+
|Range (yd)|Drop (MIL)|Come-Up   | Vel (fps)|Energy    | Time (s) |
+----------+----------+----------+----------+----------+----------+
|      100 |    0.000 |    -     |     2513 |     2455 |    0.115 |
|      200 |    0.558 |    0.557 |     2334 |     2117 |    0.239 |
|      300 |    1.335 |    0.777 |     2163 |     1817 |    0.373 |
+----------+----------+----------+----------+----------+----------+
```

Compare with the imperial-saved profile's 0.557 and 1.334 at the same ranges. The residual thousandth is not a conversion bug — it is the 2700 ft/s versus 823 m/s rounding in our two *saves*. 2700 ft/s is 823.0 (and a bit) metres per second, and we rounded when we typed it.

`profile show` and `profile list` ignore `--units`

`ballistics -u metric profile show LABBOOK` prints byte-identical output to the imperial call — 2700.0 fps, 175.0 gr, 0.308 in, 59.0 degrees F, 29.92 inHg. The display always follows the profile's *stored* units, never your requested ones.

`profile list` is worse, because it puts profiles in different systems in the same table with no units column at all:

```
+--------------------+--------+-------+--------+----------+----------+
| Name               | Vel    | BC    | Mass   | Diameter | Drag     |
+--------------------+--------+-------+--------+----------+----------+
| LABBOOK            |   2700 | 0.243 |  175.0 |    0.308 | G7       |
| METBOOK            |    823 | 0.243 |   11.3 |    7.820 | G7       |
| SUBDEMO            |   1050 | 0.270 |  220.0 |    0.308 | G7       |
+--------------------+--------+-------+--------+----------+----------+
```

The first and second rows are the same rifle. Nothing on screen says so.

19.3.2 Optional blocks

Beyond the fixed fields, a profile grows optional keys as you use it: `elevation_click`, `windage_click`, `elevation_cf`, `windage_cf`, `bc_reference`, `pressure_type`, `density_altitude`, `bullet_length`, `use_bc_segments`, `twist_right`, `wind_speed`, `wind_direction`, `shooting_angle`, `auto_zero` — plus three structured blocks:

Listing 19.6: A DSF point and a zero set, as stored

```
"dsf_points": [
  { "mach": 0.8753624451510928, "dsf": 1.086566991504561 }
],
"zero_sets": [
  { "name": "100yd-hot", "zero_distance": 100.0,
    "poi_up_mil": -0.25, "notes": "summer load" }
]
```

and `"reticle"`, the full reticle description from Section 17.9, attached with `profile save --reticle-json FILE` and validated at save time.

Re-saving carries the structured blocks forward

Running `profile save` again on an existing name does **not** wipe the DSF table, the zero sets, or the attached reticle. We re-saved `LABBOOK.json` with a reticle attached and confirmed the `zero_sets` array survived unchanged. To clear them deliberately, pass `--clear-dsf` or `--clear-reticle`.

Zero sets are the mechanism for "same rifle, second dope card." A set carries an optional zero distance and a constant angular dial correction, selected at solve time with `--zero-set NAME`:

Listing 19.7: Adding and listing a zero set

```
$ ballistics profile zero-set add LABBOOK --name "100yd-hot" \
    --zero-distance 100 --poi-up -0.25 --notes "summer load"
Added zero set '100yd-hot' to profile 'LABBOOK'.

$ ballistics profile zero-set list LABBOOK
Zero sets for 'LABBOOK':
  100yd-hot            zero 100 yd, up -0.25 mil - summer load
```

The zero-set sign convention is inverted from intuition

`--poi-up` is the *correction*, not the observation. A load that impacts 0.25 mil **high** at the zero distance needs `--poi-up -0.25`. Get this backwards and you double the error instead of removing it.

19.4 Reusing a profile — and the two flags named `--profile`

This is the single biggest usability hazard in the CLI, and it is worth stating flatly: **there are two different `--profile` flags with different argument types.**

Table 19.1: Which flag means which kind of profile

Flag	Argument	Where
`--profile <NAME>`	saved-profile name	`come-ups`, `wind-card`, `range-table`, `lead`, `mpbr`, `stability`, `compare`, `hold-corridor`, `mark-to-range`, `bdc-match`, `optimal-zero`, `reticle hold`, `plan-truing`
`--profile <FILE>` + `--profile-row <NAME>`	a CSV file	**`trajectory` only**
`--saved-profile <NAME>`	saved-profile name	`trajectory`, `dsf`

The failure is loud, at least:

Listing 19.8: `trajectory --profile` wants a file

```
$ ballistics trajectory --profile LABBOOK --max-range 500
error: the following required arguments were not provided:
  --profile-row <NAME>

Usage: ballistics trajectory --profile-row <NAME> --profile <FILE>
       --max-range <MAX_RANGE>
```

Listing 19.9: `--saved-profile` is what you meant

```
$ ballistics trajectory --saved-profile LABBOOK --max-range 500 \
    --ignore-ground-impact
Loaded saved profile 'LABBOOK'
Effective values: velocity=2700.0 (trued=2700.0), BC=0.243 (trued=0.2430)
+========================================+
|         TRAJECTORY RESULTS             |
+========================================+
| Max Range:           500.00 yd         |
| Max Height:            1.71 yd         |
| Zero Angle:          0.0637deg           |
| Time of Flight:       0.673 s          |
| Impact Velocity:    1841.34 fps        |
| Impact Energy:      1317.27 ft-lb      |
+========================================+
| Stability (SG):        3.80            |
| Status:             STABLE             |
+========================================+
```

Note the `Effective values:` echo. Any command that loads a profile prints what it resolved, and explicit flags always override stored values. Note also the stability factor: 3.80 here versus 3.96 for the same load driven by bare flags in Section 17.3, because the profile stores `bullet_length: 1.24` and the flags-only run had to estimate a length from mass and diameter.

19.4.1 What a profile does not feed

A saved profile is not a complete solve description, and the gaps are specific.

The atmosphere is stored but not always read. A profile always carries `temperature`, `pressure`, `humidity` and `altitude`, and `come-ups`, `range-table` and the other card commands do consume them. `trajectory --saved-profile` does **not**. On that command, shot-day atmosphere comes only from explicit flags or a `--location` CSV.

Required flags stay required. `come-ups --profile LABBOOK` still demands `--zero-distance`, even though the profile stores one.

And one command reads the profile but not its drag model. This one is worth a box.

`reticle hold --profile` ignores the stored drag model

Our `LABBOOK.json` stores `"drag_model": "G7"`. `come-ups --profile LABBOOK` honours it and reports **3.268 mil** at 500 yards with 1841 ft/s remaining — identical to the explicit-flag G7 run. `reticle hold --profile LABBOOK --range 500` does not:

```
$ ballistics reticle hold --profile LABBOOK --range 500 --mag 10
Solved at 500 yd - 1220.0 fps remaining, 0.841 s time of flight
...
Hold down:     4.853 mil

$ ballistics reticle hold --profile LABBOOK --range 500 --mag 10 --drag-model g7
Solved at 500 yd - 1841.2 fps remaining, 0.673 s time of flight
...
Hold down:     3.268 mil
```

4.853 mil against a true 3.268 — a 1.585 mil error, 48% high, at 500 yards. The 1220.0 ft/s figure is exactly what an explicit `--drag-model g1` run produces, which identifies the cause precisely. **Always pass `--drag-model` explicitly to `reticle hold`**, even when the profile already has one. While we are here: `mark-to-range --profile LABBOOK` does not pick up the profile's *attached reticle* either — it answers `"at least one --mark <MIL> is required (or --reticle-json <FILE>)"`. Pass the reticle file.

19.4.2 Importing from elsewhere

`profile import <FILE>` reads ArcherBC2's `.a7p` format. Three flags matter. `--dry-run` parses and prints the full mapping report without writing anything. `--strict` makes a checksum mismatch fatal instead of a warning. And `--zero-click <CLICK>` exists because of a genuine gap in the source format: `.a7p` stores its zeroing state as raw device *click counts* with no record of the device's click size. Without being told the graduation, the importer cannot convert them and reports those fields as unmapped rather than guessing.

The mapping report names every field imported, every field that could not be mapped, and every warning. Imported profiles are stored in metric units.

19.5 CSV profiles and locations

`trajectory` alone can read a two-file CSV setup: a gun profile file selected by `--profile FILE --profile-row NAME`, and a location file selected by `--location FILE --site NAME`. This is the batch-processing path, and it has sharp edges.

Listing 19.10: A minimal pair of CSV files

```
$ cat gun_profiles.csv
RIFLE_NAME,VELOCITY,BC,BC_TYPE,BULLET_WEIGHT,CALIBER,ZERO_RANGE,SIGHT_HEIGHT,TWIST_RATE
LABBOOK308,2700,0.243,G7,175,0.308,100,1.5,8

$ cat locations.csv
LOCATION_NAME,ALTITUDE,PRESSURE,TARGET_TEMP,HUMIDITY
DULUTH,1400,28.44,41,60
```

Listing 19.11: Both files, loaded

```
$ ballistics trajectory --profile gun_profiles.csv --profile-row LABBOOK308 \
    --location locations.csv --site DULUTH -m 175 -d 0.308 \
    --max-range 500 --ignore-ground-impact
Loaded profile 'LABBOOK308' from "gun_profiles.csv"
Loaded location 'DULUTH' from "locations.csv"
Effective values: velocity=2700.0 (trued=2700.0), BC=0.243 (trued=0.2430)
                  altitude=1400, temp=41.0, pressure=28.44
+========================================+
| Max Range:           500.00 yd        |
| Zero Angle:          0.0736deg          |
| Impact Velocity:    1235.34 fps       |
+========================================+
note: twist rate not supplied; assumed 1:11.3" for the stability/spin-drift
estimate - Sg shown is not an evaluated stability verdict.
```

(Rows elided from the boxed summary for length.) Since v0.30.1 both file flags require their row selectors at the parser level. Previously a `--profile` without a `--profile-row` was silently ignored *entirely*, producing what the engine's own changelog calls "a clean trajectory computed against air or a gun the user never chose." A bare `--profile-row` or `--site` is still legal, because those also label the PDF dope card.

Now the edges. All three were found by running the thing.

Three CSV traps, all verified

1. The documented header form does not work. The engine's `CLI_USAGE.md` shows the header line beginning with a hash: `#RIFLE_NAME,VELOCITY,BC,...`. With v0.30.1 that line is skipped as a comment, the first *data* row becomes the header, and you get nonsense:

```
Warning: Column 'G7' not recognized - did you mean 'MV'? Using MV.
Warning: Column '8' not recognized - did you mean 'MV'? Using MV.
Error: "failed to load profile row 'LABBOOK308' from \"gun_profiles.csv\":
        Row 'LABBOOK308' not found in CSV"
```

Drop the hash and it works.

2. `BULLET_WEIGHT` and `CALIBER` do not satisfy `-m` and `-d`. Omit those flags and the run stops with `Error: --mass is required (or use --saved-profile)` even though both columns are present and allowlisted.

3. `BC_TYPE` and `DRAG_MODEL` are not consumed at all, and nothing warns you. Both column names are in the engine's known-column allowlist, so neither triggers the "Column not recognized" warning — and neither reaches the solver. Same file, same row, with and without an explicit drag model:

```
CSV says BC_TYPE=G7, no --drag-model flag:
  Zero Angle 0.0657 deg    Impact Velocity 1220.15 fps
same CSV row, plus --drag-model g7:
  Zero Angle 0.0637 deg    Impact Velocity 1841.34 fps
```

621 ft/s of terminal velocity at 500 yards, silently, because a column that looks consumed is not. **Always pass `--drag-model` with a CSV profile.**

The lesson generalises past this CSV reader. An allowlist that suppresses warnings for columns nothing reads is worse than no allowlist: it converts a typo warning into silence and a wrong answer.

19.6 The reference drag curves

What a G-model actually is

A drag model is not a formula. It is a **table** of drag coefficient C_d against Mach number, measured for one specific *standard projectile* of a named shape. G1 is a flat-based bullet with a 2-caliber ogive; G7 is a boat-tail with a 7-caliber secant ogive. A ballistic coefficient is then the ratio that says "my bullet decelerates like that standard one, scaled by this much," and the solver's drag term is C_d/BC.
A BC is therefore meaningless without naming its model, and a G7 BC fed to a G1 curve is not slightly wrong — it is describing a different bullet shape.

The `drag-curve` subcommand prints these tables directly, which is a rare and useful thing for a ballistics program to expose:

Listing 19.12: The G7 reference curve through the transonic rise

```
$ ballistics drag-curve
G7 reference drag curve
84 points, Mach 0.00 to 5.00

    Mach        Cd
--------  --------
   0.000    0.1198
   0.500    0.1194
   0.700    0.1202
   0.800    0.1242
   0.850    0.1306
   0.900    0.1464
   0.925    0.1660
   0.950    0.2054
```

(Intermediate rows elided; the table is 84 points.) That is the transonic drag rise in eight numbers: essentially flat to Mach 0.7, then C_d nearly doubling between Mach 0.85 and 0.95. It is the reason a bullet's dope curve bends where it does, and the reason DSF corrections exist.

Note the default: `drag-curve` defaults to **G7**, not G1 — the only command in the engine that does.

19.6.1 Nine models, nine real tables

Each family has its own measured table. There are no silent fallbacks between them. Counting rows with `-o csv`:

Table 19.2: Reference drag table sizes, v0.30.1

Model	Rows	Model	Rows	Model	Rows
G1	79	G6	79	GI	81
G2	85	G7	84	GS	81
G5	76	G8	78	RA4	87

Only four — **G1, G6, G7, G8** — are available over `solve-json` v1, and therefore only four are available to the Ballistics Lab. Ask for G5 on the wire and you are told so rather than being aliased to G1: `invalid value `G5`; expected one of G1, G6, G7, G8`. On the CLI, by contrast, an unrecognised `--drag-model` string warns on stderr and falls back to G1 — another argument for the machine interface.

19.6.2 Provenance

The tables ship in `data/` in the engine repository with a README that states their lineage, and it is worth repeating because most software does not tell you:

> Source: US Army Ballistic Research Laboratory, Aberdeen Proving Ground. Original tabulation by E. D. Lowry, Winchester-Western Division, "Exterior Ballistic Tables Based on Numerical Integration," 4 May 1965; refined by Robert L. McCoy and the BRL team. **Public domain** (US Government work).

The README is also candid about accuracy: excellent below Mach 0.8, good with interpolation through Mach 0.8–1.2, very good above 1.2, and limited above Mach 3.0 where the data thin out and extrapolation begins. That transonic band is exactly where a real bullet departs most from a standard one, which is why Doppler-derived custom curves exist.

19.6.3 Where the tables actually live — and one that moves

Here the story gets interesting, and this section exists because we tried to break it and succeeded.

Seven of the nine tables are compiled into the binary with Rust's `include_str!`, so they cannot be affected by anything on disk: G1, G2, G5, G7, GI, GS, RA4. The source comment says why in as many words — an earlier version fell back to a coarse in-source table when the file was missing, "flattening the transonic drag rise," and one fallback value was wrong by more than 10% at Mach 0.9.

G6 and G8 are different. They still go through a runtime loader that searches for a directory named `drag_tables` relative to the *current working directory*:

Listing 19.13: The search path, from `src/drag.rs` (Rust)

```rust
fn find_drag_tables_dir() -> Option<std::path::PathBuf> {
    let candidates = [
        "../drag_tables", "../../drag_tables",
        "../../../drag_tables", "drag_tables",
    ];
    for candidate in &candidates { ... }
}
```

If such a directory exists and contains `g6.npy` or `g6.csv`, the engine loads it in preference to its built-in table. Let us find out whether that is theoretical. In an empty scratch directory we created `drag_tables/` containing a deliberately absurd three-point `g6.csv` — and a three-point `g1.csv` as a control:

Listing 19.14: A working directory that redefines G6

```
$ printf '0.0,0.9999\n1.0,0.8888\n5.0,0.7777\n' > drag_tables/g6.csv
$ printf '0.0,0.1111\n1.0,0.2222\n5.0,0.3333\n' > drag_tables/g1.csv

$ ballistics drag-curve --drag-model g6 -o csv | head -4
mach,cd
0,0.9999
1,0.8888
5,0.7777

$ ballistics drag-curve --drag-model g1 -o csv | head -4
mach,cd
0,0.2629
0.05,0.2558
0.1,0.2487
```

G1 is immune. G6 is not — it now reports our fabricated curve. And this is not confined to the diagnostic command. The identical solve-json request, the same binary, the same version, run from two different directories:

Listing 19.15: The same G6 request, two working directories, two answers

```
$ cd clean_dir  &&  ballistics solve-json < g6req.json
"summary": {"actual_range_m": 500.0, "time_of_flight_s": 0.7801763461180865,
  "terminal_speed_mps": 495.12512258291804,
  "terminal_energy_j": 1389.9941893622902, ...}

$ cd poisoned_dir  &&  ballistics solve-json < g6req.json
"summary": {"actual_range_m": 500.0, "time_of_flight_s": 1.192497503005923,
  "terminal_speed_mps": 238.7913107033844,
  "terminal_energy_j": 323.3107146823863, ...}
```

G6 and G8 solves are working-directory dependent

495.1 m/s versus 238.8 m/s of terminal velocity, from byte-identical requests to the same binary. The response envelope reports `status: "ok"`, the same `engine_version`, the same `resolved_request`, and no warning of any kind. Nothing in the protocol can tell you which curve ran.

This is a synthetic attack — we created the directory — but it needs no privileges, no configuration, and no flag. Any directory tree containing a folder called `drag_tables` within three levels above the working directory changes G6 and G8 answers silently.

Practical guidance. For reproducible work, prefer G1 or G7, which are compiled in. If you must use G6 or G8, run the engine from a known-clean working directory — and note that the Ballistics Lab launches the engine from wherever the Ballistics Lab itself was launched, so the same care applies there. The Section 18.7 reproducibility result was obtained with G7 fixtures, and that is not an accident of which fixtures happened to exist.

19.7 The Ballistics Lab's own data

The Ballistics Lab ships two experiments and thirteen fixture documents, and every one of them has a job.

19.7.1 Two reference experiments, deliberately different

`reference_experiment.lat` exports `representative_experiment()`: the *documentation* load. It is what the REPL starts with and what most of this book's examples use. A 175 gr G7 match bullet at BC 0.243, 0.308 inch, 1.24 inch long; a 24-inch match rifle with a 1.5 inch sight height and a 1:8 right twist; 250 m altitude, 15 degrees C, 1013.25 hPa, 50% humidity, latitude 45 degrees; one 10 mph wind from 90 degrees; 2700 ft/s to 1000 yards with a 100 yard zero; rk45 with 10-yard sampling and **Coriolis enabled**. It is stated in imperial units throughout, because a reader is meant to recognise it.

`fixture_experiment.lat` exports `representative_fixture_experiment()`: the *test* load. Everything is raw SI — 0.01134 kg, 0.00782 m, 823 m/s — there is no latitude and Coriolis is off, the muzzle sits 1.2 m up, there is an explicit 0.001 rad muzzle angle instead of a zero distance, and the maximum range is **50 metres**. Fifty. That tiny range is the whole point: it produces six samples, which makes `fixtures/representative.success.json` 3691 bytes and hand-auditable.

The two exist because they are optimised for opposite things. The first must be recognisable; the second must be checkable.

19.7.2 The fixture library

Listing 19.16: `examples/ballistics_lab/fixtures/`, sizes in bytes (columns rearranged for the page)

```
   207  internal.error.json
   176  invalid-json.error.json
  1032  representative.request.json
  3691  representative.success.json
   256  resource.error.json
   228  validation.error.json

conformance/
  1049  calm-air.request.json          3684  calm-air.success.json
  1067  crosswind.request.json         3803  crosswind.success.json
  1036  early-termination.request.json 2328  early-termination.success.json
  1032  representative.request.json    3691  representative.success.json
  1482  segmented-wind.request.json    4262  segmented-wind.success.json
  1048  zeroed.request.json            3734  zeroed.success.json
```

The four `*.error.json` documents are the negative fixtures — one per error class the decoder must handle. They are small enough to read in full:

Listing 19.17: `fixtures/validation.error.json`

```
{
  "schema_version": 1,
  "status": "error",
  "error": {
    "code": "invalid_value",
    "message": "value must be finite and greater than zero",
    "path": "$.projectile.mass_kg",
    "line": null, "column": null
  }
}
```

The six conformance pairs are the ones we replayed across four engine versions in Section 18.7. `representative.request.json` doubles as a *golden request*: the Ballistics Lab's own test suite compares `protocol.v1_request_json(fixture.representative_fixture_experiment())` against those exact bytes, so a change to any adapter in `protocol_v1.lat` fails a byte comparison before it can reach an engine.

19.7.3 The Ballistics Lab's save-file format

The Ballistics Lab's `:save` writes a versioned document that is deliberately *not* a solve-json request:

Listing 19.18: A saved experiment (abridged; 1257 bytes complete)

```
{
  "kind": "lattice-ballistics/experiment",
  "schema_version": 1,
  "experiment": {
    "name": "175 gr match at 1,000 yards",
    "projectile": {
      "name": "175 gr match",
      "mass":     { "unit": "gr", "value": 175 },
      "diameter": { "unit": "in", "value": 0.308 },
      "length":   { "unit": "in", "value": 1.24 },
      "drag_model": "G7",
      "ballistic_coefficient": 0.243
    },
    "shot": {
      "muzzle_velocity": { "unit": "ft/s", "value": 2700 },
      "max_range":       { "unit": "yd",   "value": 1000 },
      "zero_distance":   { "unit": "yd",   "value": 100 },
      "muzzle_angle": null, "aim_azimuth": null, "shot_azimuth": null,
      "shooting_angle": null, "cant_angle": null,
      "target_height": null, "ground_threshold": null
    },
    ...
  }
}
```

Three differences from the wire format are all deliberate, and together they say what this document is for.

It stores display units, not SI. Every quantity is a `{unit, value}` pair in the unit you typed. The wire stores canonical SI and no unit at all. A save file is for a human to reopen; a request is for a machine to run.

Absent optionals are explicit `null`. The wire omits them. Here the null is information: it says the field exists in the schema and you did not set it, so a reader can tell "unset" from "this version had no such field."

It contains no run, no history, and no result. The 1257 bytes above are the experiment and nothing else. A `:save` after two `:solve` calls writes exactly the same document as a `:save` before them.

`"target_height": null` is not a missing zero datum

`target_height` is the height above ground that the *zero* is solved to. The engine documents its wire counterpart, `shot.target_height_m`, as "meters above ground for zeroing" and defaults it to 0.0 — the ground. The save file above genuinely does not set it, and `:show` reports it as `default` for the same reason: the domain field really is unset.

The datum is supplied one layer down, at encoding time. `protocol_v1.lat` fills `target_height_m` with muzzle height plus sight height — the height of the line of sight — whenever the experiment leaves the domain field alone, because a conventional zero puts the target on the line of sight rather than on the dirt. You can read it back off any run, from the engine's own echo of what it resolved:

Listing 19.19: `json_stringify(run.resolved_request.get("shot"))` after a `:solve`

```
{"cant_angle_rad":0,"muzzle_angle_rad":0.0011642456054687502,"max_range_m":1097.28,"
    zero_distance_m":91.439999999999998,"aim_azimuth_rad":0,"shooting_angle_rad":0,"
    ground_threshold_m":-100,"shot_azimuth_rad":0,"target_height_m":0.044450000000000003}
```

That is a rifle with a 1.75-inch sight height and no muzzle height, so `target_height_m` is 0.04445 m — and the same request's `rifle.sight_height_m` is 0.04445 m too. Identical, as they should be.

This is the save-file/wire split above at its sharpest. The document is what you typed; the request is what gets solved; the adapter between them is where a convention like "zero to the line of sight" belongs. Put the convention in the document instead and every save file ever written would be claiming a decision its author never made.

The document is round-tripped before it is written

`persistence.lat` does not trust its own encoder. `experiment_to_map` encodes the experiment and then immediately *re-decodes* the result through the loading path before publishing any bytes — because, as the source comment puts it, "callers can construct exported struct literals without going through the domain constructors." A struct name is not a trust boundary. Every quantity adapter does the same thing one level down: it reconstructs the quantity from its `{value, unit}` form and asserts the reconstructed `si_value` matches, "to prevent a forged quantity from hiding invalid state."

19.8 A map of every file

Table 19.3: Where everything lives

Path	What it is
Engine, per user	
`~/.ballistics/profiles/<N>.json`	one saved load, in the units it was entered
`~/.ballistics/credentials.toml`	online API token, mode 0600 (v0.32.0 only)
`~/.ballistics/tos.json`	terms acceptance
`~/.ballistics/era5_cache/`	cached reanalysis weather
Engine, bundled	
`data/g1.csv` … `ra4.csv`	seven reference drag tables, compiled in with `include_str!`
`data/g1.npy`, `g7.npy`	the same data as NumPy binaries, for external tooling
`data/README.md`, `LICENSE`	BRL/Lowry/McCoy provenance; public domain
`data/bc_tables/`, `bc_corrections_transonic.bin`	5D BC correction data (not in solve-json v1)
`<cwd>/drag_tables/g6.{npy,csv}`	**runtime-loaded** G6/G8 override — see Section 19.6
`fonts/LiberationSans-*.ttf`	compiled in, for PDF dope cards
Engine, caller-supplied	
`gun_profiles.csv`	`trajectory --profile FILE`
`locations.csv`	`trajectory --location FILE`
`*.a7p`	ArcherBC2 import source
`*.json` (reticle)	`reticle generate -o json` output
`*.json` (scenarios)	`hold-corridor --scenarios`
The Lab	
`examples/ballistics_lab/*.lat`	36 source files, 14 038 lines
`.../reference_experiment.lat`	the documentation load
`.../fixture_experiment.lat`	the 50-metre test load
`.../fixtures/*.json`	golden request, success, four error classes
`.../fixtures/conformance/`	six request/response pairs
`scripts/ballistics-lab.sh`	the launcher
anywhere you name	`lattice-ballistics/experiment` save files and `/verified-run` artifacts

19.9 Working habits

A short list, all of it earned by something that went wrong above.

- **Say `--drag-model` every time.** Every solver defaults to G1. A CSV's `BC_TYPE` column does not set it. A saved profile does set it — except on `reticle hold`.
- **Use `--saved-profile` on `trajectory`** and `--profile` everywhere else. The names are backwards from what you expect exactly once.
- **Store the bullet length.** It is the term the stability factor is most sensitive to, and without it the engine estimates.
- **Read the `Effective values:` echo.** Every profile-loading command prints what it resolved. That line is the cheapest bug detector in the toolbox.
- **Prefer G1 or G7 for anything you intend to reproduce.** They are compiled into the binary; G6 and G8 are not.
- **Archive the resolved request, not the profile.** A profile is a convenience that can change under you. A resolved request is the run.

That last one closes Part IV. The engine is a large, capable, occasionally inconsistent piece of software with thirty ways in — and one narrow door through which everything is stated explicitly, defaulted visibly, and reproduced exactly. Part V turns to the other side of that door, and to how fourteen thousand lines of Lattice decide what to do with what comes through it.

Part V

How the Lab Is Built

Chapter 20

Architecture

Part V is for the programmer. If you have been reading this book to learn what your rifle does at 900 yards, you can stop here with a clear conscience — the ballistics is behind you. What follows is a case study: a working, tested, 14,062-line application written in ordinary Lattice, examined the way you would examine any codebase you were about to inherit.

The subject is worth the attention for one specific reason. The Ballistics Lab is not a toy that shows off language features. It is a scientific instrument with a hard correctness requirement, a hostile input boundary, an external process it does not trust, and a user interface that has to stay usable after sixteen consecutive errors. Every design decision in it was forced by one of those constraints. That makes it a much better teacher than a tutorial.

This chapter answers three questions: what the modules are, how they depend on each other, and what actually happens — function by named function — when a user types `:solve` and presses Enter.

Which engine produced the numbers in Part V

Every transcript in these seven chapters was captured against the `ballistics` binary on this machine's PATH, which reports **0.32.0** — the version the copyright page names. Every `:solve` in this part self-reports `engine : 0.32.0`, and the raw envelopes carry `"engine_version": "0.32.0"`. Part IV notes where the older 0.30.1 build differs. The lab's behaviour is identical either way — `engine_version` is opaque to it and `schema_version` is what matters — but do not expect a 0.30.1 binary to reproduce these trajectory numbers to the last decimal.

Parts I–IV defined a Lattice term on first use every time one was needed by a reader who came for the ballistics. Part V owes the same courtesy in the other direction. If you arrived here as a

programmer, this is the whole domain vocabulary the case study leans on; each entry names the chapter that takes it apart properly.

Domain vocabulary for Part V

Drag model — a published reference drag curve, indexed by Mach number, that stands in for a whole family of bullet shapes; its name (`G1`, `G7`, and so on) is what the wire field `drag_model` carries (Section 10.4).
Ballistic coefficient (BC) — one dimensionless number saying how much less this bullet decelerates than the reference bullet of its drag model, which is exactly why a BC is meaningless without naming that model (Section 10.5).
Zero — the distance at which the sight line and the trajectory are made to agree; every drop in this book is stated relative to some zero, and `zero_distance_m` is the field that carries it (Chapter 13). Two wire fields together decide what "agree" means: `zero_distance_m` says *where*, and `target_height_m` says *at what height above the ground*. The Ballistics Lab fills the second in with the height of the line of sight, so a zero lands on the sight line (Section 20.4).
Drop — how far the bullet is below the line of sight at a given range, printed by the Ballistics Lab as a positive number *downward*: positive is below the sight line, negative is above it. At the muzzle the drop equals the sight height, because that is how far under the sight line the bore sits (Chapter 5).
Come-up — the angular elevation correction you dial or hold to put that bullet back on the aiming point, positive when the sight must be *raised*, which is why dope is quoted as an angle and not as inches (Chapter 6).
Mil — exactly one milliradian, 0.001 radian, subtending 3.6 inches at 100 yards; **MOA** — a minute of angle, 1/60 of a degree, about 1.047 inches at 100 yards. They are different angles, and the Ballistics Lab keeps both exact (Chapter 8).
Truing — adjusting model parameters, usually muzzle velocity or BC, until the solver reproduces impacts you actually measured, which makes the adjusted number an *effective* value rather than a measured one (Chapter 14).
Mach, and transonic — Mach number is speed divided by the local speed of sound, and the transonic band from roughly Mach 1.2 down to 0.8 is where drag rises steeply and groups tend to open up (Chapter 5).
Spin drift — the steady sideways drift a spin-stabilised bullet develops because its nose rides slightly off the velocity vector: right-hand twist drifts it right, in still air, on every shot (Section 15.3).

20.1 Fourteen Thousand Lines of Ordinary Lattice

Start with the negative space. The Ballistics Lab adds nothing to the Lattice runtime. There is no ballistics intrinsic, no native extension, no `dlopen`, no C ABI, no vendored Rust. The README

states the thesis in its first paragraph, and the thesis is falsifiable: if the lab needed a language change, the claim would be dead.

Listing 20.1: The lab, by the numbers

```
$ cd ~/projects/lattice/examples/ballistics_lab
$ wc -l *.lat | tail -1
   14062 total
$ ls *.lat | wc -l
      36
```

Thirty-six top-level `.lat` files: eighteen library and application modules, eighteen test modules. Alongside them sit `examples/` (six runnable scripts, 621 lines), `fixtures/` (a fake engine written in LATTICE, six root JSON fixtures, twelve conformance fixtures), and a 27 KB README that is the user-facing manual.

The one thing the lab does *not* do is compute ballistics. Every trajectory number in this book came out of a separate Rust program, `ballistics-engine`, invoked as a child process. The lab owns everything else: units, domain validation, protocol encoding, response decoding, interpolation, table construction, rendering, persistence, artifacts, session state, and a REPL.

A subprocess, not a library

An earlier design document in the LATTICE tree proposes binding the engine's Rust API from a native extension — a `ballistics.dylib` loaded into the interpreter. That is not what shipped. Section 20.7 explains the proposal and why the process boundary won. Whenever this book says "the engine," it means a child process the lab spawns, hands one JSON request on stdin, and reads one JSON envelope back from.

20.2 The Module Map

Eighteen modules, in dependency order — leaves first. That ordering is not cosmetic; it is the order in which you can read the codebase without ever following a forward reference to a module you have not met.

Module	Lines	Responsibility
`units.lat`	455	Eight nominally distinct SI-backed quantity types with constructors and converters.
`domain.lat`	728	Validated, immutable, relationally checked values: the vocabulary of a shot.

Module	Lines	Responsibility
`protocol_v1.lat`	230	Experiment → solve-json v1 request `Map`. Reads only canonical SI.
`analysis_export.lat`	524	`AnalysisTable` → Map, canonical JSON, CSV, or plain text. No I/O.
`resolved_request_v1.lat`	335	Strict validation of a detached solve-json request snapshot.
`repl_support.lat`	6	The EOF-compatibility predicate, factored out so it can be tested.
`backend.lat`	85	The transport-independent engine seam: one struct, one closure.
`command_parser.lat`	466	The bounded colon-command grammar, parsed into inert data.
`analysis.lat`	706	Checked interpolation, bounded resampling, three table builders.
`session.lat`	630	All mutable state behind one `Ref`; transactional mutation.
`process_backend.lat`	860	`EngineBackend` over `exec_argv`: the subprocess implementation.
`persistence.lat`	786	Strict versioned experiment JSON through explicit per-type adapters.
`render.lat`	1031	Deterministic terminal strings. Never prints, never serializes.
`verified_artifacts.lat`	724	The deliberate promotion of a run record to a portable document.
`commands.lat`	392	Routing a parsed command to session, analysis, persistence, render.
`reference_experiment.lat`	79	The documentation load: a 175 gr G7 match rifle.
`fixture_experiment.lat`	63	The test load: raw SI, 50 m, six samples, hand-auditable.
`ballistics-repl.lat`	195	The interactive application. The only module with a `main()` loop.

Two observations before we go further.

The renderer is the biggest file. `render.lat` is 1,031 lines — larger than the module that talks to the engine, larger than the domain model. That is normal and it is honest. Formatting numbers correctly for a shooter (fixed decimals, explicit signs, right-aligned columns, negative-zero normalization) is genuinely more code than parsing a JSON envelope.

The smallest module is six lines. `repl_support.lat` exists only so that one predicate — "is this value the runtime's end-of-input marker?" — can be unit-tested without a terminal. It is the smallest possible demonstration of what "make it testable" means in practice.

Listing 20.2: `repl_support.lat`, in full

```
// Pure compatibility helpers shared by the self-hosted REPL and transcripts.

export fn is_eof_value(value: any) -> Bool {
    let kind = typeof(value)
    return kind == "Nil" || kind == "Unit"
}
```

20.3 The Dependency Graph

Every `import` statement in the tree, tabulated. This was produced by grepping the sources, not by reading the documentation.

Listing 20.3: Extracting the true import graph

```
$ for f in units domain protocol_v1 analysis_export resolved_request_v1 \
    repl_support backend command_parser analysis session process_backend \
    persistence render verified_artifacts commands ballistics-repl; do
    printf "%-22s " "$f"
    grep -o 'ballistics_lab/[a-z_0-9]*"' $f.lat \
      | sed 's|.*ballistics_lab/||;s|"||' | sort -u | tr '\n' ' '
    echo
  done
units
domain
protocol_v1
analysis_export
resolved_request_v1
repl_support
backend                domain
command_parser         domain
analysis               domain units
session                backend domain
process_backend        backend domain protocol_v1 units
persistence            domain session units
render                 session units
verified_artifacts     domain persistence resolved_request_v1
commands               analysis command_parser domain persistence render session units
ballistics-repl        analysis backend command_parser commands domain persistence
                       process_backend reference_experiment render repl_support
                       resolved_request_v1 session units verified_artifacts
```

The graph is a directed acyclic graph with **six roots** and **one sink**. There are no cycles anywhere, and there is no module that imports something merely for convenience.

20.3.1 Six modules import nothing

Six modules have no imports whatsoever:

- `units.lat` — a genuine leaf; it defines its own quantities.
- `repl_support.lat` — a genuine leaf; six lines around a single `typeof` check.
- `domain.lat` — consumes quantities without importing them.
- `protocol_v1.lat` — consumes the whole domain graph.
- `analysis_export.lat` — consumes analysis tables.
- `resolved_request_v1.lat` — consumes protocol maps.

The last four are the interesting ones. Each validates struct types it did not define and cannot name, using a single trick.

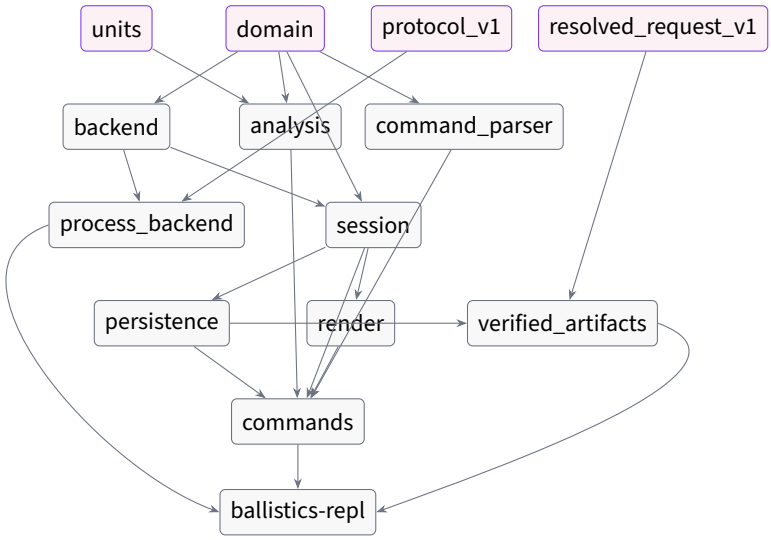


Figure 20.1: The spine of the dependency graph. Purple nodes import nothing at all. Edges into `commands` and `ballistics-repl` are shown selectively; those two modules import seven and fourteen modules respectively.

Listing 20.4: `protocol_v1.lat:8–16` — structural typing instead of an import

```
fn _require_struct(value: any, expected: String, field: String) {
    if typeof(value) != "Struct" {
        assert(false, field + ": expected " + expected + ", got " + typeof(value))
    }
    let actual = struct_name(value)
    if actual != expected {
        assert(false, field + ": expected " + expected + ", got " + actual)
    }
}
```

Structural typing via `struct_name`

`struct_name(value)` returns the declared name of a struct value as a `String`. Comparing that name to an expected literal lets a module validate a value it did not define and cannot name. The cost is that the check is a runtime check on a name, not a compile-time check on a type: a struct literal declared elsewhere with the same name will pass. `persistence.lat` takes that seriously — see Section 20.8.

This is the lab's main decoupling technique, and it buys something concrete. `analysis_export.lat` is 524 lines that turn an `AnalysisTable` into JSON or CSV. It imports nothing — not even the module that defines `AnalysisTable`. As a result nothing in the library depends on it, and you can point it at any table-shaped value from any source. Conversely, if you delete `analysis_export.lat`, the REPL still builds. The dependency report proves it: `analysis_export` has zero importers among the library modules.

20.3.2 The two import styles, and when each is used

LATTICE has two import forms, and the lab uses both on purpose.

Listing 20.5: Selective import — a seam module naming exactly what it needs

```
import {
    laboratory_error
}
from "examples/ballistics_lab/domain"
```

Listing 20.6: Aliased import — an orchestration module keeping every call qualified

```
import "examples/ballistics_lab/units" as units
import "examples/ballistics_lab/domain" as domain
import "examples/ballistics_lab/session" as sessions
import "examples/ballistics_lab/analysis" as analysis
```

The pattern is consistent enough to be a rule. *Seam* modules import selectively. `backend.lat` takes one name. So does `command_parser.lat`. The process backend takes three names from units, six from domain, and one each from backend and protocol. In each case the import list doubles as a statement of exactly how much of its neighbour the seam touches.

Orchestration modules import under an alias. In `session.lat`, `render.lat`, and `commands.lat` every call site reads `sessions.solve_session(lab)` or `render.render_dope(table, profile)`, so you can tell which layer a line belongs to without scrolling back to the header.

`ballistics-repl.lat` does both simultaneously, and says why:

Listing 20.7: `ballistics-repl.lat:9–11`

```
// Constructors are intentionally imported into the top-level environment so
// lat_eval users can write concise experiments while retaining module aliases
// for discoverable, explicit access.
```

Twenty-two unit constructors and twelve domain constructors land unqualified in the top-level environment, because a user typing at the prompt should be able to write `projectile(...)` and not `domain.projectile(...)`. Fourteen module aliases land alongside them, because the same user needs `sessions.replace_shot(lab, ...)` to change anything. Both surfaces exist because the REPL's evaluator runs against this exact environment.

Import paths are repository-root-relative

Every import in the lab reads `"examples/ballistics_lab/units"`. That is a path from the repository root, not from the importing file. It is why every script header says "run from the repository root" and why the launcher script changes directory before invoking `clat`. If you copy a module elsewhere, you must rewrite its imports.

20.4 Five Rules That Explain the Layering

Reading the lab module by module, five rules recur. They are not written down in one place in the source; they are visible in the shape of every file.

20.4.1 Rule 1: SI at the wire, display metadata at the edge

Every dimensional value in the lab is a struct with three fields: `si_value`, `display_value`, and `display_unit`. Chapter 21 covers the type in detail. Architecturally what matters is what crosses each boundary:

- `protocol_v1.lat` reads `si_value` and nothing else. Display metadata cannot reach the engine.
- `persistence.lat` writes `display_value` and `display_unit` and *not* `si_value`, then reconstructs and cross-checks the SI value on the way back in.
- `render.lat` converts `si_value` to whatever the display profile asks for at the last possible moment.

The claim is checkable. Here is the exact request the lab put on the engine's stdin for the reference experiment — a load defined entirely in grains, inches, feet per second, and yards:

Listing 20.8: The reference experiment on the wire (819 bytes, reformatted for width)

```
{"sampling":{"interval_m":9.1440000000000001},
 "rifle":{"twist_direction":"right","muzzle_velocity_mps":822.96000000000004,
          "muzzle_height_m":0,"sight_height_m":0.038099999999999995,
          "twist_rate_m_per_turn":0.20319999999999999},
 "schema_version":1,
 "shot":{"max_range_m":914.39999999999998,"zero_distance_m":91.439999999999998,
         "target_height_m":0.038099999999999995},
 "atmosphere":{"altitude_m":250,"latitude_rad":0.78539816339744828,
               "relative_humidity":0.5,"pressure_pa":101325,
               "temperature_k":288.14999999999998},
 "wind":{"direction_from_rad":1.5707963267948966,"speed_mps":4.4703999999999997},
 "solver":{"method":"rk45"},
 "projectile":{"drag_model":"G7","diameter_m":0.0078231999999999989,
               "length_m":0.031495999999999996,
               "ballistic_coefficient":0.24299999999999999,
               "mass_kg":0.011339809249999999},
 "effects":{"coriolis":true}}
```

Listing 20.9: Grepping the wire for display units

```
$ grep -c -E '"(gr|in|fps|yd|mph|inHg|ft-lb|mil)"' ref_request.json
0
```

Not one display unit made it across.

Typed as	Wire field	Wire value
`gr(175)`	`mass_kg`	0.011339809249999999
`fps(2700)`	`muzzle_velocity_mps`	822.96000000000004
`yd(1000)`	`max_range_m`	914.39999999999998
`inches(0.308)`	`diameter_m`	0.0078231999999999989
`deg(45)`	`latitude_rad`	0.78539816339744828

One field the adapter supplies rather than copies

Almost every key in that request is a domain value converted to SI. One is not. `target_height_m` appears above as `0.038099999999999995` — the same number as `sight_height_m` — even though the experiment never set a target height and `:show` still reports `target height : default`.

The field is the height above the ground that the zero is solved to. The engine documents it that way and defaults it to `0.0`, target at ground level, if it is absent. A rifle is not zeroed to the dirt, so `protocol_v1.lat:74–92` supplies the height of the line of sight instead — muzzle height plus sight height — whenever the domain `Shot` leaves it `nil`. Sending nothing was a real defect in earlier builds of the Ballistics Lab, and Section 23.6 takes it apart as a worked example. The point here is architectural: the adapter is the only layer that knows what a v1 request means, so it is the only layer entitled to fill in a datum the domain model does not carry.

20.4.2 Rule 2: failure is a value at module seams

Internal validation throws. `units.lat` and `domain.lat` call `assert(false, message)` freely, and they do it deep inside private helpers. But at three specific boundaries — the backend, the command dispatcher, and the response decoder — a thrown error is caught and converted into a `LaboratoryError` struct.

Listing 20.10: `backend.lat:59–67` — the conversion point

```
let solve_fn = backend.solve_fn
let result = try {
    solve_fn(experiment)
} catch error {
    return laboratory_error(
    "backend_contract_error",
    "backend '" + backend.name + "' failed: " + error.message
    )
}
```

The consequence a user sees is that the REPL never dies. Sixteen consecutive malformed commands leave the process running and exiting 0.

20.4.3 Rule 3: construct then publish

Session mutation builds a complete next state and installs it with exactly one `Ref.set`. There is no partial update anywhere in `session.lat`: every mutator ends in the same two-line function.

Listing 20.11: session.lat:130–132 — the only write in the module

```
fn _commit(session: any, state: any) {
    session.state.set(state)
}
```

A failed solve, a failed load, or a failed validation therefore leaves prior state untouched — not by cleanup, but because nothing was written.

20.4.4 Rule 4: freeze the graph once, at the top

Children stay mutable (**fluid**) or merely field-immutable (**alloy**) during construction; a single `freeze()` at the end owns the finished value. `domain.lat` names that point:

Listing 20.12: domain.lat:223–225

```
fn _frozen(value: any) -> any {
    return freeze(value)
}
```

Every one of the fourteen domain constructors ends `return _frozen(Struct ... )`. Deliberate exceptions exist and are documented in comments where they occur — `backend.lat:36–38` for a closure-bearing struct, `session.lat:19–22` for a record that may wrap ten thousand samples. Chapter 21 takes this apart.

20.4.5 Rule 5: fail closed

Anything the protocol did not promise becomes an error. Not a warning, not a best-effort parse: an `invalid_engine_response` value. Chapter 22 tabulates the forty-plus response shapes that were driven through the decoder to check this.

20.5 One Solve, End to End

Now the walk-through. A user has launched the lab, changed nothing, and typed `:solve`. Sixty-odd named functions run. Here they are in order.

The starting state was established at module load. `ballistics-repl.lat` builds the session at top level, on purpose:

Listing 20.13: ballistics-repl.lat:82–89

```
// Top-level on purpose: lat_eval executes against this environment, so users
// can inspect and mutate the same Ref-backed LabSession used by colon commands.
let engine_executable = _engine_executable()
let engine_version = _engine_version()
let lab = sessions.new_session(
reference.representative_experiment(),
_engine_backend(engine_executable, engine_version)
)
```

20.5.1 Phase A — terminal to command value

1. `main()` calls `input(prompt)`, which returns the string `":solve"`.
2. `repl_support.is_eof_value(line)` returns false.
3. The empty-line guard is skipped.
4. `_starts_command(line)` trims leading spaces and checks `substring(0, 1) == ":"`. True. Because the multiline buffer is empty, the command branch is taken — a colon *inside* a multiline block would be source, not a command.
5. `_run_command(line)` opens a `try`.
6. `commands.execute_command(lab, ":solve")`.
7. `parser.parse_command(":solve")` runs the lexer: the 4,096-byte line cap, the NUL check, the control-character scan, the leading colon, then one token `CommandToken { text: "solve", column: 2, quoted: false }`. `"solve"` is in the zero-arity list, so `_arity` passes and `_command` returns `freeze(ColonCommand { name: "solve", args: {} })`.

The important structural fact is what *did not* happen: the command text was never turned into LATTICE source and never handed to `lat_eval`. `commands.lat` says so in its header — "This module never synthesizes source and never invokes `lat_eval`, a shell, an executable, or a process."

20.5.2 Phase B — dispatch and validation

`dispatch_command` is a two-stage `try`, and the two stages produce different error codes on purpose.

Listing 20.14: commands.lat:360–383 — validation, then execution

```
export fn dispatch_command(lab: any, command: any) -> any {
    flux checked = nil
    flux profile = nil
    let validation_error = try {
        _require_struct(lab, "LabSession", "dispatch_command.lab")
        checked = _validate_command(command)
        profile = render.render_profile_from_display(sessions.display_of(lab))
        nil
    } catch error {
        error
    }
    if validation_error != nil {
        return _error("command_validation_error", validation_error.message)
    }

    return try {
        _dispatch_checked(lab, checked, profile)
    } catch error {
        _error(
        "command_execution_error",
        checked.name + ": " + error.message
        )
    }
}
```

A caller can therefore distinguish “your command was malformed” from “executing it failed” without string matching. Note also that `_validate_command` re-checks a command that the parser already validated — because `dispatch_command` is public and accepts hand-built `ColonCommand` values from code that never went through the parser.

Reading the display profile pulls one value through the `Ref`. The accessor `display_of` calls the private `_session_state`, which performs exactly one `Ref.get` and then asserts that what came back is a `LabSessionState`.

20.5.3 Phase C — the session transaction

Listing 20.15: session.lat:574–590 — the transactional core

```
export fn solve_session(session: any) -> any {
    let state = _session_state(session)
    // Experiment is already a frozen domain value, so this is an immutable O(1)
    // snapshot on shared-crystal runtimes and an independent immutable value on all
    VMs.
    let snapshot = state.experiment

    // Exactly one backend invocation. solve_with converts thrown/wrong-shape backends
    into
    // LaboratoryError values; every error path below returns before touching session
    state.
    let result = backends.solve_with(state.backend, snapshot)
    if backends.solve_failed(result) {
        return result
    }

    let record = _record_or_error(state.backend, snapshot, result)
    if _is_struct(record, "LaboratoryError") {
        return record
    }
```

session.lat imports domain and backend and nothing else. It has no idea that processes exist. That is the single most important structural fact in the whole application: the module that owns state cannot see the module that owns the engine.

20.5.4 Phase D — the process backend

solve_with invokes the backend's closure, which was created at process_backend.lat:837 and captured a validated executable path, a validated argv array, and a validated limits Map. That closure calls _solve_process, which does four things in order:

1. Serialize the request, inside a try. A throw here becomes a request_serialization_error value.
2. Check the request length against MAX_REQUEST_BYTES, which is 1,048,576.
3. Call exec_argv, inside a try. A throw here becomes a process_error value.
4. Decode the result.

We can watch step 3 from the child's side. Replace the engine with a shell script that records what it was handed:

Listing 20.16: A recorder standing in for the engine

```
$ cat record-engine.sh
#!/bin/sh
{
  echo "argc=$#"
  i=1
  for a in "$@"; do echo "argv[$i]=$a"; i=$((i+1)); done
  echo "--- stdin ---"
  cat
} > "$RECORD_TO"
echo 'not json'
```

Listing 20.17: What the child actually received

```
$ RECORD_TO=record.txt ./clat argv_probe.lat ./record-engine.sh
invalid_engine_response: engine stdout is not exactly one JSON envelope:
json_parse error at position 0: unexpected character
$ head -3 record.txt
argc=1
argv[1]=solve-json
--- stdin ---
$ tail -1 record.txt | wc -c
     819
```

One argument. The literal string `solve-json`. Eight hundred and nineteen bytes of request on stdin. No experiment value appears in argv at all. Chapter 22 makes the security consequence of that explicit.

20.5.5 Phase E — decoding the response

`decode_v1_process_result` is exported separately from the transport so it can be tested against a hand-written `{stdout, stderr, exit_code}` `Map` with no child process at all. It checks the shape of that map first, then parses the JSON, then hands the envelope to `_decode_envelope`, which checks `schema_version` and `status` and branches to `_decode_success` or `_decode_error`.

For a successful solve the ordering inside `_decode_success` is load-bearing: the envelope's key set is checked, then exit code and stderr, then engine version, then the resolved request, then notices, then the summary, then samples, then the terminal-sample cross-check — and only then is a `Trajectory` constructed. Nothing partial is ever built.

Here is what the real engine sent back for the reference experiment, viewed directly:

Listing 20.18: The same request, run against the engine by hand

```
$ ballistics solve-json < ref_request.json > ref_resp.json
$ python3 - <<'PY'
import json; d=json.load(open('ref_resp.json'))
print(list(d.keys()))
print(d['status'], d['engine_version'], d['schema_version'], len(d['samples']))
print(json.dumps(d['summary'], indent=1))
PY
['schema_version', 'engine_version', 'status', 'resolved_request',
 'assumptions', 'warnings', 'summary', 'samples']
ok 0.32.0 1 101
{
 "actual_range_m": 914.4,
 "maximum_height_m": 0.04111754369481144,
 "time_of_flight_s": 1.7064080077984438,
 "terminal_speed_mps": 349.2875788381682,
 "terminal_energy_j": 691.7386422597805,
 "stability_factor": 3.7979628593634613,
 "termination": "max_range"
}
```

20.5.6 Phase F — committing the run

Back in `solve_session`, `_record_or_error` performs three provenance checks before building a `RunRecord`: the returned trajectory must retain the exact experiment snapshot that was submitted (`result.experiment != snapshot` is a hard failure), it must carry a resolved-request `Map`, and it must be a `Trajectory` at all. Only then does history get cloned, appended, and trimmed — and only then does the single `_commit` run.

20.5.7 Phase G — rendering

`render.render_run_summary(record, profile)` performs a deep structural audit of the record before formatting one character of it, then builds twelve labelled rows and joins them. It never prints; it returns a `String` that `_run_command` hands to `print`.

Two mechanical notes about every transcript in this part. The launcher writes one line to *stderr* — `ballistics-lab: backend=stack-vm, engine=/Users/alexjokela/.local/bin/ballistics` — which never appears in a stdout capture. And because `input()` uses readline, the `ballistics>` prompt is suppressed when stdin is a pipe, so piped transcripts contain no prompts at all. The three-line startup banner is elided below because its first line contains an em dash that this book's listing environment cannot typeset verbatim.

Listing 20.19: `:solve` on the shipped default experiment (banner elided)

```
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1000.00 yd
  maximum height   : +1.62 in
  time of flight   : 1.71 s
  terminal velocity: 1145.96 ft/s
  terminal energy  : 510.20 ft-lb
  stability factor : 3.80
  spin drift       : not reported
```

Every one of those numbers is the SI value from the envelope above, converted at render time.

Envelope field (SI)	Value	Rendered
`actual_range_m`	914.4	`1000.00 yd`
`time_of_flight_s`	1.7064080077984438	`1.71 s`
`terminal_speed_mps`	349.2875788381682	`1145.96 ft/s`
`terminal_energy_j`	691.7386422597805	`510.20 ft-lb`
`stability_factor`	3.797962859363461 3	`3.80`

$914.4/0.9144 = 1000$ exactly. $349.2875788381682/0.3048 = 1145.957$, rendered at two decimals as `1145.96`. $691.7386422597805/1.3558179483314004 = 510.200$, rendered as `510.20`. The conversion constants are the ones in `units.lat:56–70`, and nothing else touches the numbers.

20.5.8 The chain, compressed

For reference, the full call chain of one `:solve`, in order:

Listing 20.20: The named functions of a single solve

```
input -> is_eof_value -> _starts_command -> _run_command -> execute_command
  -> parse_command -> _lex -> _first_forbidden_control -> _arity
  -> _empty_command -> _command
  -> dispatch_command -> _validate_command -> _require_exact_args
  -> render_profile_from_display -> _profile -> display_preferences
  -> _dispatch_checked
  -> solve_session -> _session_state -> solve_with -> (closure)
  -> _solve_process -> v1_request_json -> experiment_to_v1_request
  -> projectile_to_v1_map ... sampling_to_v1_map -> json_stringify
  -> exec_argv
  -> decode_v1_process_result -> json_parse -> _decode_envelope
  -> _decode_success -> _validate_resolved_request -> _decode_notices
  -> _decode_summary -> _decode_samples -> _decode_checked_sample
  -> observation -> _validate_terminal_sample -> _numbers_match -> trajectory
  -> _record_or_error -> _commit -> Ref.set
  -> render_run_summary -> _validate_run -> _fields -> _fixed
  -> _response -> print
```

20.5.9 What it costs

Listing 20.21: Wall clock, measured three times each

```
$ printf ':quit\n' | /usr/bin/time -p sh scripts/ballistics-lab.sh > /dev/null
real 0.02
$ printf ':solve\n:quit\n' | /usr/bin/time -p sh scripts/ballistics-lab.sh > /dev/null
real 0.04
$ /usr/bin/time -p sh -c 'i=0; while [ $i -lt 50 ]; do
    ballistics solve-json < ref_request.json > /dev/null; i=$((i+1)); done'
real 0.17
```

Launching the lab and quitting takes about 20 ms. Adding one solve takes it to about 40 ms. Fifty engine solves — fifty process spawns included — take 170 ms, so a solve costs roughly 3.4 ms of engine time. The process boundary is not the bottleneck at human interaction rates, and that observation is most of the argument for the architecture that shipped.

20.6 What the Layering Buys

20.6.1 The backend is a closure, so tests need no process

backend.lat is 85 lines and defines a struct with two fields: a name and a closure.

Listing 20.22: backend.lat:12–15 and :31–43

```
export struct EngineBackend {
    name: fix String,
    solve_fn: fix Fn
}

export fn engine_backend(name: any, solve_fn: any) -> EngineBackend {
    let checked_name = _nonempty(name, "backend.name")
    if typeof(solve_fn) != "Closure" {
        assert(false, "backend.solve_fn: expected Closure, got " + typeof(solve_fn))
    }
    // The fields themselves are alloy `fix` fields. Avoid crystallizing a
    // closure-bearing struct: closures may capture independently frozen config,
    // and nesting those shared regions in another crystal is unnecessary.
    return EngineBackend {
        name: checked_name,
        solve_fn: solve_fn
    }
}
```

That is the whole interface. session.lat calls backends.solve_with(backend, experiment) and receives a Trajectory or a LaboratoryError, and it does not care whether a process, a fixture, or a counter-incrementing test double produced it. The session tests build backends out of nothing but a lambda and a Ref.

20.6.2 The decoder is separable from the transport

decode_v1_process_result takes a plain {stdout, stderr, exit_code} Map. You can drive every hostile-response case through it with a string literal. The lab does something stronger — it writes a fake engine in LATTICE and runs it as a real child process, so the transport is exercised too — but the seam means you have both options.

20.6.3 Commands cannot reach the engine

The command dispatcher imports seven modules: units, domain, session, analysis, persistence, the command parser, and the renderer. It does *not* import `backend.lat` or `process_backend.lat`. Every path to the engine goes through `sessions.solve_session`. That is what makes the command layer testable against injected backends, and it is also why "a colon command started a process I did not expect" is not a failure mode that exists.

20.7 The Road Not Taken: A Native Extension

The LATTICE tree contains a design document, `design/ballistics-extension.md`, that proposes a different architecture: a Rust `cdylib` installed at `~/.lattice/ext/ballistics.dylib`, statically linking the engine crate and binding its Rust API through LATTICE's extension ABI. Its status line reads *"Proposed. Architecture decided; pre-implementation."*

This is a design document, not the shipped system

No `extensions/ballistics/` directory exists. The BALLISTICS LAB you have been using in Parts I–IV drives the engine as a subprocess through `exec_argv`, speaking solve-json v1. Read the extension document as an argument that was made and set aside, not as documentation of anything you can run.

The proposal is not silly. Its central argument — that the engine's C FFI is a lossy projection of the engine, exposing eight symbols and no vocabulary at all for variable-length arrays like wind segments, BC segments, or custom drag tables — is correct, and it is why the document rejects the generic FFI route. Binding the Rust API from a Rust extension really would reach capability that the C boundary cannot.

What the subprocess architecture bought instead is worth stating plainly, because the trade is a good example of a decision that looks like a compromise and is not.

- **A versioned wire contract instead of an ABI.** solve-json v1 has a `schema_version` field that the lab checks on both sides. An extension has a shared-object ABI, and the document's own "§11 risk" is that the extension resolves `lat_ext_*` symbols from the host `clat` binary at load time.
- **A trust boundary that is a process boundary.** The engine cannot corrupt the interpreter's heap, cannot hold a reference to a frozen LATTICE value, and cannot outlive the solve. The extension design spends a long paragraph on precisely this problem — it has to force-copy every value crossing into the extension because a shared crystal-region handle escaping into a `dlopen`'d library would be an over-release.

- **Distribution.** The extension document notes that native extensions ship as manually installed per-platform shared libraries, because the LATTICE package registry serves source packages only. A subprocess ships as whatever the engine already ships as.

- **A seam a test can impersonate.** "One argv token and a JSON document on stdin" is something you can write a fake engine against, in LATTICE, with no compiler and no linker in the loop. `fixtures/fake_solve_json_engine.lat` is 295 lines of exactly that (Chapter 25), and Section 26.11 argues it is the point an extender should weigh first.

- **Cost.** 3.4 ms per solve, measured above. For an interactive tool that is free.

Neither architecture reaches a browser today, and the book should say so plainly: compiled for emscripten, `require_ext` fails with "native extensions not available in WASM" and `exec_argv` fails with "exec_argv: not available in browser." Both are closed doors. The difference is that one of them is a data protocol, and a data protocol can be spoken by something other than a child process.

The one thing the process boundary genuinely gives up is the engine's non-solve capability. The engine binary on the machine these transcripts came from lists thirty-six subcommands plus `help`, and `solve-json` is one of them. Truing, Monte-Carlo dispersion, MPBR, and reticle work are all reachable from the command line, as Part IV showed, but not through the lab's backend seam. Section 26.11 picks the same question up from the other end — what reaching one of those verbs actually costs, and why an extension is not the cheaper way to do it.

This section is the book's one full treatment of the unbuilt design. Everything after it in Part V describes code that exists.

20.8 Where the Layering Leaks

No architecture survives contact with a real feature list intact. Four places in the lab will surprise a reader who has internalized the rules above. All four are deliberate; three of them are documented in the source.

20.8.1 `render` imports `session`

A reader who expects "presentation depends on nothing" will trip on `render.lat:9`. The renderer imports the session module — not for state, but to reuse `display_preferences()` as its own profile validator. Rather than duplicate five range checks, `render.lat` runs its input back through the session module's constructor. It is a defensible call and a real coupling: you cannot use `render.lat` without pulling in `session.lat`, `domain.lat`, and `backend.lat`.

20.8.2 `resolved_request_v1` duplicates `process_backend`

`resolved_request_v1.lat` is a near-verbatim twin of `process_backend.lat:141–329`. Same nine section validators, same relational checks, same JSON paths. A reviewer's instinct is to file that as a defect and factor it out.

Do not. The two validators run at different times against different threat models. `process_backend` validates a response *in flight*, from a child process, on the path that constructs domain values. `resolved_request_v1` validates a snapshot *at rest*, out of a file someone may have edited, on the path that builds a portable artifact. The artifact path must not depend on the process backend — and it does not: `verified_artifacts.lat` imports `resolved_request_v1` and never touches `process_backend`. The duplication is what makes that independence real. The endings differ too, which is the tell: one returns `freeze(object)`, the other `freeze(clone(object))`.

20.8.3 `analysis_export` is unreachable from the REPL

There is no colon command that emits CSV. `:dope` renders through `render.lat`; JSON and CSV export is a source-API-only capability exercised by `examples/export.lat`. If you want a table in a spreadsheet, you write four lines of LATTICE, not a command. The README documents this under "Python and Jupyter interoperability."

20.8.4 Wire key order is not declaration order

`protocol_v1.lat:212–220` sets the nine top-level request keys in a readable order: `schema_version`, `projectile`, `rifle`, `shot`, `atmosphere`, `wind`, `solver`, `effects`, `sampling`. The bytes that reach the engine are in a different order — look again at the request listing above, which starts with `sampling` and puts `schema_version` third. `Map` iteration order is a property of the runtime's map implementation, and `json_stringify` follows it.

This does not matter for the engine, which parses JSON objects as unordered. It matters for anyone who assumes the two orders agree. The lab's own golden request test is careful about it: `test_backend.lat:56–64` compares `json_parse` of the generated request against `json_parse` of the fixture, not the raw strings.

Where determinism *is* guaranteed

`analysis_export.lat` faces the same problem for its JSON output and solves it the other way: it assembles canonical JSON by hand rather than serializing a `Map`, with the comment "Produce canonical JSON without serializing a Map, so object member order is independent of the runtime's Map implementation." `json_stringify` is used there only for scalars. Two modules, two conclusions, both correct for their purpose — and the difference is exactly the difference between "bytes a machine will parse" and "bytes a human will diff."

20.9 Reading Order

If you are going to read the source, read it in dependency order:

Listing 20.23: The spine, in reading order

```
units.lat            455
domain.lat           728
protocol_v1.lat      230
backend.lat           85
process_backend.lat  860
analysis.lat         706
session.lat          630
render.lat          1031
command_parser.lat   466
commands.lat         392
ballistics-repl.lat  195
                    ----
                    5778
```

That is 5,778 lines and it is the whole spine. Persistence, resolved-request validation, verified artifacts, and table export are branches you can take when you need them.

The next two chapters take the two most instructive stops on that path in detail: the domain model, where LATTICE's phase system does structural work that a type system would do in another language; and the process backend, where 860 lines exist for the sole purpose of not trusting 23 KB of JSON.

Chapter 21

The Domain Model

`domain.lat` is 728 lines and it defines thirteen structs and fifteen exported constructor functions. It imports nothing. It performs no I/O, computes no physics, and formats no strings. What it does is refuse to build a value that does not make sense.

That is a bigger job than it sounds. A shot is not a bag of numbers. A zero distance and a muzzle angle are alternative ways of saying the same thing, so supplying both is a contradiction, not an over-specification. Coriolis deflection needs a latitude. Magnus needs a bullet length. Wind is either one constant vector or an ordered list of downrange segments, never a mixture. A ground threshold above the muzzle describes a bullet that starts underground. Every one of those is a relationship between fields, and every one of them is checked in this file before any value exists.

This chapter is about how LATTICE's phase system, structural typing, and default parameters combine to make that enforcement cheap enough that nobody was tempted to skip it.

21.1 Thirteen Structs

Struct	What it is
`Projectile`	Name, mass, diameter, length, drag model, BC.
`Rifle`	Name, sight height, muzzle height, twist rate, twist direction.
`Atmosphere`	Altitude, temperature, pressure, relative humidity, latitude.
`Wind`	Speed, direction-from, vertical velocity, until-distance.
`Shot`	Muzzle velocity, max range, and eight optional geometry fields.
`SolverConfig`	Method, time step, sampling interval, and three effect toggles.
`Experiment`	Name plus the six above. The unit of work.

Struct	What it is
`Observation`	One trajectory sample: distance, time, velocity, energy, drop, windage, Mach, flags.
`Assumption`	An engine notice: code, message, optional JSON path.
`Warning`	The same shape, a different vocabulary of codes.
`TrajectorySummary`	Actual range, max height, ToF, terminal velocity and energy, stability, spin drift, termination.
`Trajectory`	The whole answer: schema and engine version, the submitted experiment, the resolved request, samples, notices, summary, verified flag.
`LaboratoryError`	A failure, as data: code, message, path or line/column, stderr, exit code.

Notice the shape of that list. Six of the thirteen describe an experiment *before* it is solved. Five describe an answer *after*. One — the `Trajectory` — joins the two by carrying the exact experiment it came from. And one is an error.

Listing 21.1: `domain.lat:105–115` — the join

```
export struct Trajectory {
    schema_version: fix Int,
    engine_version: fix String,
    experiment: fix any,
    resolved_request: fix any,
    observations: fix Array,
    assumptions: fix Array,
    warnings: fix Array,
    summary: fix any,
    verified: fix Bool
}
```

Carrying the experiment inside the answer is what makes provenance work. A DOPE table rendered from that trajectory can print the engine version, the solver method, and every assumption the engine applied, without anyone threading that information through five function calls by hand. And the session layer checks it: `session.lat:549` rejects a trajectory whose `experiment` field is not the exact value that was submitted.

21.2 Quantities

Every dimensional field in those structs holds a quantity value from `units.lat`. There are eight quantity types and they all have the same three fields.

Listing 21.2: `units.lat:1–12` — the header comment and the first type

```
// Typed quantities for the source-level ballistics laboratory.
//
// Each value stores a canonical SI value and the exact value/unit requested
// for display.  Every struct field is an alloy `fix` field, so callers can
// inspect quantities but cannot silently change either their dimension or
// their display metadata.

export struct Distance {
    si_value: fix Number,
    display_value: fix Number,
    display_unit: fix String
}
```

`Velocity`, `Mass`, `Angle`, `Temperature`, `Pressure`, `Energy`, and `WindSpeed` are byte-for-byte identical declarations under different names. That repetition is the point.

Nominal typing

Two types are *nominally* distinct when they differ by name even though they have identical structure. `Velocity` and `WindSpeed` both store metres per second in `si_value`, and there is no structural difference between them at all. They are different types solely because they have different names, and the domain constructors compare names.

The payoff is a class of bug that cannot be written:

Listing 21.3: Handing a muzzle-velocity type to the wind constructor

```
$ ./clat probe.lat
assertion failed: wind.speed: expected WindSpeed, got Velocity
```

`mps(4.0)` and `wind_mps(4.0)` produce values with identical `si_value`, and the second is the only one `wind()` will accept. That is deliberate: a wind speed and a projectile speed are physically different

roles even when they share a dimension, and confusing them silently produces a plausible wrong answer rather than a crash.

Each dimension has a small family of constructors and exactly one converter.

Dimension	Constructors	Converter units
Distance (m)	`m km yd ft inches`	`m km yd ft in`
Velocity (m/s)	`mps fps`	`m/s ft/s`
Mass (kg)	`kg grams gr`	`kg g gr`
Angle (rad)	`rad deg mil moa`	`rad deg mil moa`
Temperature (K)	`kelvin celsius fahrenheit`	`K C F`
Pressure (Pa)	`pa hpa inhg`	`Pa hPa inHg`
Energy (J)	`joules foot_pounds`	`J ft-lb`
Wind speed (m/s)	`wind_mps mph kph`	`m/s mph km/h`

The conversion constants are exact where exactness exists.

Listing 21.4: `units.lat:56–70`

```
let PI = 3.141592653589793
let METERS_PER_KILOMETER = 1000.0
let METERS_PER_YARD = 0.9144
let METERS_PER_FOOT = 0.3048
let METERS_PER_INCH = 0.0254
let KILOGRAMS_PER_GRAM = 0.001
let KILOGRAMS_PER_GRAIN = 0.00006479891
let RADIANS_PER_DEGREE = PI / 180.0
let RADIANS_PER_MIL = 0.001
let RADIANS_PER_MOA = PI / 10800.0
let PASCALS_PER_HECTOPASCAL = 100.0
let PASCALS_PER_INHG = 3386.389
let JOULES_PER_FOOT_POUND = 1.3558179483314004
let METERS_PER_SECOND_PER_MPH = 0.44704
let METERS_PER_SECOND_PER_KPH = 1.0 / 3.6
```

`RADIANS_PER_MIL = 0.001` is a decision, not an approximation. A ballistic mil in this lab is exactly one milliradian — not the NATO 6400, not the Warsaw Pact 6000, not the 1/6400-of-a-circle a reticle vendor may have used. If your scope's clicks disagree, the scope is what needs the correction factor.

Listing 21.5: Two angle constants, verified

```
mil is exactly a milliradian   : 0.001
moa in radians                 : 0.000290888
```

Converters double as unit validators

Every `*_as` function ends with a call to `_unsupported`, which throws a message listing every unit it accepts. `analysis.lat` exploits this: before building a DOPE table it calls `units.distance_as(units.m(0), drop_unit)` purely to make the check run, discarding the result. One canonical diagnostic, reused instead of duplicated.

```
assertion failed: distance_as: unsupported unit 'furlong';
  supported: m, km, yd, ft, in
assertion failed: wind_speed_as: unsupported unit 'ft/s';
  supported: m/s, mph, km/h
```

Note the second one. A wind speed cannot be displayed in feet per second, on purpose — the wind vocabulary is metres per second, miles per hour, and kilometres per hour, and nothing else.

21.3 Immutability Without a Region

Now the part that is specific to Lattice.

Phases

A Lattice value has a *phase*. **fluid** values are freely mutable. **crystal** values are deeply immutable and live in a shared, refcounted region — `freeze(x)` produces one. `phase_of(value)` reports the phase as a string, and a value that has never been frozen and was never declared `flux` reports `"unphased"`.
Separately, a struct *field* may be declared `fix`, which means that field can never be assigned after construction. The lab's comments call a struct whose fields are all `fix` an *alloy* value: immutable in practice, but not living in a crystal region.

That distinction is not academic here, so let us watch it happen.

Listing 21.6: Phases through one domain constructor

```
$ ./clat domain_probe.lat
-- phases --
phase_of(units.gr(175))     : unphased
phase_of(projectile)        : crystal
phase_of(projectile.mass)   : crystal
phase_of(the caller's gr()) : unphased
struct_name(projectile.mass): Mass
mass si / display / unit    : 0.0113398 / 175 / gr
-- immutability --
assign to an unphased quantity: cannot assign to frozen field 'si_value'
assign to a frozen projectile : cannot assign to frozen field 'name'
```

Four facts in six lines, and each one is load-bearing.

A freshly constructed quantity is `unphased`. `units.gr(175)` does not call `freeze`. It builds a `Mass` struct with three `fix` fields and returns it. There is no crystal region, no refcount, no sharing machinery.

It is still immutable. Assigning to that field fails, and the runtime says so: *cannot assign to frozen field*. The `fix` declaration alone was enough. This is why `units.lat` can be 455 lines of allocation-light constructors — it gets immutability from the field declarations and never pays for a region.

Freezing the parent freezes the children. After `domain.projectile(...)`, both the projectile and its `mass` field report `crystal`. One `freeze` at the top of the graph crystallized everything below it.

The caller's binding is untouched. The last line is the subtle one. `raw_mass` — the same `Mass` the constructor received — still reports `unphased` after the projectile was built and frozen. The constructor froze its own finished value; it did not reach back and mutate the phase of the argument the caller still holds.

Do not read "unphased" as "mutable"

This is the most common misreading of `phase_of` output. A value can be completely unmodifiable and still report `unphased`, because `phase_of` tells you which *region model* the value is in, not whether you can write to it. In the lab, the answer to "can I mutate this?" is almost always no regardless of the phase, because almost every struct in the lab declares every field `fix`.

21.4 One Freeze at the Top

`domain.lat` names its crystallization point, once, and every constructor routes through it.

Listing 21.7: domain.lat:223–225

```
fn _frozen(value: any) -> any {
    return freeze(value)
}
```

A three-line wrapper around a builtin looks like ceremony until you grep for it. Thirteen constructors call `_frozen` — one for each struct in the module — and `freeze` appears anywhere else in the file exactly twice: inside `calm_wind()`, which returns `freeze([])`, and inside `observation()`, where the flags array is frozen separately before the enclosing struct is. Of the fifteen exported functions, the two that do not call `_frozen` are `calm_wind()` and `standard_atmosphere()`, which delegates to `atmosphere()`.

Listing 21.8: domain.lat:542–560 — nested freeze in `observation`

```
    flux checked_flags = []
    for item in flags {
        checked_flags.push(_require_choice(
        item,
        ["transonic", "subsonic", "terminal", "ground_threshold"],
        "observation.flags[]"
        ))
    }
    return _frozen(Observation {
        distance: distance,
        time_s: checked_time,
        velocity: velocity,
        energy: energy,
        drop: drop,
        windage: windage,
        mach: checked_mach,
        flags: freeze(checked_flags)
    }
    )
```

The array is built `flux`, filled one validated element at a time, frozen, and then placed into a struct that is itself frozen. Both freezes are necessary: the inner one because the array was constructed locally and would otherwise be a mutable value sitting inside an immutable struct, and the outer one because the struct is the thing callers will hold.

Listing 21.9: The result, verified

```
flags        : [transonic, terminal]
flags phase  : crystal
calm_wind()  : [] phase=crystal
```

The single-freeze discipline has a cost that the lab pays knowingly. A `Trajectory` for a 1000-yard solve carries 101 observations, each with five quantity structs inside it. Freezing the trajectory crystallizes all of them. That is fine here, because the trajectory is built once and read many times — but it is exactly why `session.lat` declines to freeze the record that *wraps* a trajectory:

Listing 21.10: `session.lat:19–22` — a documented non-freeze

```
// RunRecord is an immutable alloy wrapper. Its fields are `fix`, and every composite
// payload placed in it is already frozen by the domain/backend boundary. Avoid
    freezing
// the outer wrapper again: a trajectory may contain 10,000 samples, and alloy fields
    give
// us immutability without copying that already-frozen graph into another crystal
    region.
```

The rule, stated plainly

Freeze a value when you built it and callers will hold it. Do not freeze a wrapper around a value someone else already froze. `fix` fields give you the immutability; `freeze` gives you the region, and you only want the region once.

21.5 The Runtime Dependency That Is Not an Import

`domain.lat` validates that a projectile's mass is a `Mass` and its diameter is a `Distance`. Those types live in `units.lat`. And `domain.lat` does not import `units.lat`.

Listing 21.11: domain.lat:178–192 — the whole mechanism

```
fn _require_quantity(value: any, expected: String, field: String) {
    if typeof(value) != "Struct" {
        assert(false, field + ": expected " + expected + ", got " + typeof(value))
    }
    let actual = struct_name(value)
    if actual != expected {
        assert(false, field + ": expected " + expected + ", got " + actual)
    }
}

fn _require_optional_quantity(value: any, expected: String, field: String) {
    if value != nil {
        _require_quantity(value, expected, field)
    }
}
```

The expected type is a string literal at the call site:

Listing 21.12: domain.lat:235–243

```
    let checked_name = _require_nonempty(name, "projectile.name")
    _require_quantity(mass, "Mass", "projectile.mass")
    _require_quantity(diameter, "Distance", "projectile.diameter")
    _require_optional_quantity(length, "Distance", "projectile.length")
    _require_positive(mass.si_value, "projectile.mass")
    _require_positive(diameter.si_value, "projectile.diameter")
    if length != nil {
        _require_positive(length.si_value, "projectile.length")
    }
```

So `units` is a *runtime* dependency of `domain` without being a compile-time one. The graph in Chapter 20 shows `domain` as a root with no edges, and that is literally true of the imports — but you cannot use `domain.lat` without `units.lat`, because nothing else in the tree produces a struct named `Mass`.

Struct names are not a trust boundary

Nothing stops a caller from writing their own `struct Mass` with an `si_value` field and getting it past `_require_quantity`. The lab knows this, and `persistence.lat` says so out loud where it matters: when it encodes an experiment for the disk, it immediately re-decodes the result through the domain constructors before writing bytes, "because callers can construct exported struct literals without going through the domain constructors." Structural typing is a decoupling tool, not a security mechanism.

21.6 Four Layers of Validation

Every domain constructor runs the same four passes, in the same order.

21.6.1 Layer 1 — scalar predicates

Nine private helpers, all with the same signature shape, all throwing through `assert(false, message)`:

Listing 21.13: The scalar predicate set

```
_require_nonempty       _require_bool           _require_choice
_require_finite         _require_optional_bool  _require_optional_path
_require_positive       _require_nonnegative    _require_positive_int
```

Listing 21.14: domain.lat:137–154 — the finite/positive pair

```
fn _require_finite(value: any, field: String) -> Number {
    let kind = typeof(value)
    if kind != "Int" && kind != "Float" {
        assert(false, field + ": expected Number, got " + kind)
    }
    if is_nan(value) || is_inf(value) {
        assert(false, field + ": must be finite")
    }
    return value
}

fn _require_positive(value: any, field: String) -> Number {
    let checked = _require_finite(value, field)
    if checked <= 0 {
        assert(false, field + ": must be greater than zero")
    }
    return checked
}
```

Two details are worth stealing. The helpers *return* the checked value, so call sites read `let checked_bc = _require_positive(bc, "...")` rather than check-then-use. And every one of them takes a `field` string, which is why the errors you saw in Part I name the exact field that failed rather than saying "invalid argument."

21.6.2 Layer 2 — type checks on quantities

`_require_quantity` and `_require_struct`, described above. `_require_struct` is literally an alias:

Listing 21.15: domain.lat:194–196

```
fn _require_struct(value: any, expected: String, field: String) {
    _require_quantity(value, expected, field)
}
```

Same mechanism, different intent — one checks a unit, the other checks a domain value — and having both names makes the constructors read correctly.

21.6.3 Layer 3 — per-field range checks

Bounds that involve one field and one constant. Altitude between -5000 and 84000 metres. Relative humidity between 0 and 1 (a fraction, never a percentage). Latitude between $-\pi/2$ and $\pi/2$ radians. Sight height non-negative, twist rate positive, muzzle velocity positive.

Listing 21.16: `domain.lat:317–322` — latitude, in radians

```
    if latitude != nil {
        let half_pi = 3.141592653589793 / 2.0
        if latitude.si_value < 0.0 - half_pi || latitude.si_value > half_pi {
            assert(false, "atmosphere.latitude: must be between -pi/2 and pi/2
    radians")
        }
    }
```

Note `0.0 - half_pi` rather than a unary minus. That idiom recurs throughout the lab.

21.6.4 Layer 4 — relational checks

The rules that involve two or more fields. These are the ones that make the module worth its length, and they get their own section.

21.7 The Relational Rules

Eight rules, all verified by running them.

Listing 21.17: Every relational rule in `domain.lat`, triggered

```
$ ./clat relational_probe.lat
shot: zero + muzzle angle       : assertion failed: shot: zero_distance and
  muzzle_angle cannot both be supplied
solver: magnus + spin drift     : assertion failed: effects: magnus and
  enhanced_spin_drift cannot both be enabled
winds: constant mixed with segs : assertion failed: experiment.winds: cannot
  mix constant and segmented wind
winds: boundaries not increasing: assertion failed: experiment.winds: segment
  boundaries must increase strictly
winds: two constant winds       : assertion failed: experiment.winds: constant
  wind must contain exactly one Wind
coriolis without latitude       : assertion failed:
  experiment.atmosphere.latitude: required when Coriolis is enabled
magnus without length           : assertion failed:
  experiment.projectile.length: required for the selected effect
ground threshold above muzzle   : assertion failed:
  experiment.shot.ground_threshold: must be below the muzzle height
coriolis WITH latitude          : accepted
```

21.7.1 Mutual exclusion

Listing 21.18: `domain.lat:388–393`

```
    if zero_distance != nil {
        _require_positive(zero_distance.si_value, "shot.zero_distance")
    }
    if zero_distance != nil && muzzle_angle != nil {
        assert(false, "shot: zero_distance and muzzle_angle cannot both be supplied")
    }
```

A zero distance is a request: *find* the launch angle that puts the bullet on the line of sight at this range. A muzzle angle is an assertion: the barrel is pointed *here*. They cannot both be inputs, and a library that silently picked one would produce trajectories nobody could explain.

The same pattern guards the effects, where the physics reason is different: Magnus and enhanced spin drift are two models of the same lateral phenomenon, and enabling both double-counts it.

21.7.2 Wind is one shape or the other

Listing 21.19: `domain.lat:446–471` — the whole wind rule

```
fn _validate_winds(winds: Array) {
    if len(winds) == 0 {
        return
    }
    flux segmented = false
    flux last_boundary = 0.0
    for item in winds {
        _require_struct(item, "Wind", "experiment.winds[]")
        if item.until_distance != nil {
            segmented = true
            if item.until_distance.si_value <= last_boundary {
                assert(false, "experiment.winds: segment boundaries must increase
    strictly")
            }
            last_boundary = item.until_distance.si_value
        }
    }
    if segmented {
        for item in winds {
            if item.until_distance == nil {
                assert(false, "experiment.winds: cannot mix constant and segmented
    wind")
            }
        }
    } else if len(winds) != 1 {
        assert(false, "experiment.winds: constant wind must contain exactly one Wind")
    }
}
```

Read the control flow carefully, because it is a nice piece of work. One pass sets a `flux` flag and checks monotonicity as it goes. If the flag came out true, a second pass requires that *every* element is segmented. If it came out false, the array must have length exactly one. An empty array is valid — that is calm air — and is what `calm_wind()` returns.

Three distinct errors come out of twenty-six lines, and each names the actual problem. “Cannot mix constant and segmented wind” is a better message than “invalid wind array,” and it costs one extra loop.

21.7.3 Effects imply inputs

Listing 21.20: `domain.lat:492–506` — the cross-component checks

```
if solver_value.coriolis == true && atmosphere_value.latitude == nil {
    assert(false, "experiment.atmosphere.latitude: required when Coriolis is
enabled")
}
if (solver_value.magnus == true || solver_value.enhanced_spin_drift == true) &&
projectile_value.length == nil {
    assert(false, "experiment.projectile.length: required for the selected effect")
}
if shot_value.ground_threshold != nil {
    flux muzzle_height_si = 0.0
    if rifle_value.muzzle_height != nil {
        muzzle_height_si = rifle_value.muzzle_height.si_value
    }
    if shot_value.ground_threshold.si_value >= muzzle_height_si {
        assert(false, "experiment.shot.ground_threshold: must be below the muzzle
height")
    }
}
```

This is the only place in the module where fields of four different structs are compared, and it is the reason `experiment()` exists as a constructor rather than as a struct literal. You cannot assemble an `Experiment` without running these three checks, because the only exported path to one runs them.

The Coriolis rule has a nice property worth noticing: it is a rule about *information*, not about physics. Coriolis deflection depends on latitude; if you did not supply a latitude, the engine would have to invent one, and the lab would rather refuse than let you compare two solves that silently assumed different places on Earth. The shipped reference experiment is the positive case — it enables Coriolis *and* supplies 45 degrees.

21.8 Closed Vocabularies

Five string fields in the domain accept only an enumerated set. All five go through the same helper.

Listing 21.21: domain.lat:198–204

```
fn _require_choice(value: any, choices: Array, field: String) -> String {
    let checked = _require_nonempty(value, field)
    if !choices.contains(checked) {
        assert(false, field + ": unsupported value '" + checked + "'; expected one of
    " + repr(choices))
    }
    return checked
}
```

Field	Accepted values
projectile.drag_model	G1 G6 G7 G8
rifle.twist_direction	left right
solver.method	rk45 rk4 euler
observation.flags[]	transonic subsonic terminal ground_threshold
summary.termination	max_range ground_threshold time_limit velocity_floor

Listing 21.22: Three closed vocabularies, verified

```
assertion failed: projectile.drag_model: unsupported value 'G9';
  expected one of [G1, G6, G7, G8]
assertion failed: solver.method: unsupported value 'dormand-prince';
  expected one of [rk45, rk4, euler]
assertion failed: summary.termination: unsupported value 'ran_out_of_gas';
  expected one of [max_range, ground_threshold, time_limit, velocity_floor]
```

The drag-model list is the interesting one. The engine supports nine drag families on its command line — G1, G2, G5, G6, G7, G8, GI, GS, and RA4 — and solve-json v1 accepts four. The domain enforces the protocol's four, not the engine's nine, because the protocol is the contract. If you need a G5 solve, you need the CLI (Part IV), not the lab, and finding that out at construction time with a message listing the four legal values is better than finding it out from the child process.

repr() in error messages

_require_choice interpolates repr(choices) into the message, so the accepted set is always in the diagnostic and can never drift from the check. Adding a drag model is a one-line change and the error text updates itself.

21.9 The Engine's Output as Domain Values

Assumptions, warnings, observations, and summaries all arrive as JSON from a process the lab does not trust. They become domain values through the same constructors as everything else — which means the engine's output is validated by exactly the same code that validates your typing.

Listing 21.23: `domain.lat:563–579` — notices

```
export fn assumption(code: any, message: any, path: any = nil) -> Assumption {
    return _frozen(Assumption {
        code: _require_nonempty(code, "assumption.code"),
        message: _require_nonempty(message, "assumption.message"),
        path: _require_optional_path(path, "assumption.path")
    }
    )
}

export fn warning(code: any, message: any, path: any = nil) -> Warning {
    return _frozen(Warning {
        code: _require_nonempty(code, "warning.code"),
        message: _require_nonempty(message, "warning.message"),
        path: _require_optional_path(path, "warning.path")
    }
    )
}
```

Two identical structs and two identical constructors, distinguished only by name. Again, that is nominal typing doing structural work: a warning cannot be mistaken for an assumption anywhere downstream, and `render.lat` prints them under different headings without ever inspecting a discriminator field.

The vocabularies of *codes* are enforced one layer up, in `process_backend.lat`, because they are protocol facts rather than domain facts:

Listing 21.24: The notice code allowlists

```
assumptions: default_applied
             icao_standard_temperature
             icao_standard_pressure
             estimated_projectile_length

warnings:    partial_wind_coverage
             experimental_effect
             rk45_time_step_ignored
```

A fifth assumption code would be an unknown response, not an unknown domain value.

`observation()` is the constructor that runs most often — once per sample, 101 times for a 1000-yard solve — and it is worth noting what it checks and what it does not.

Listing 21.25: `domain.lat:529–541` — the sample checks

```
    _require_quantity(distance, "Distance", "observation.distance")
    _require_quantity(velocity, "Velocity", "observation.velocity")
    _require_quantity(energy, "Energy", "observation.energy")
    _require_quantity(drop, "Distance", "observation.drop")
    _require_quantity(windage, "Distance", "observation.windage")
    _require_nonnegative(distance.si_value, "observation.distance")
    let checked_time = _require_nonnegative(time_s, "observation.time_s")
    _require_nonnegative(velocity.si_value, "observation.velocity")
    _require_nonnegative(energy.si_value, "observation.energy")
    let checked_mach = _require_nonnegative(mach, "observation.mach")
    if typeof(flags) != "Array" {
        assert(false, "observation.flags: expected Array, got " + typeof(flags))
    }
```

Distance, time, speed, energy, and Mach must be non-negative. Drop and windage are typed as `Distance` and are *not* sign-constrained, because they are offsets: a bullet can be left of the line of sight or above it. That asymmetry is stated in `units.lat:104–106`:

```
// Distance: canonical meters. Generic distances may be signed so the same
// quantity can represent left/right windage and above/below trajectory offsets.
// Domain constructors enforce positivity for ranges and physical dimensions.
```

21.10 Failure as a Value

`LaboratoryError` is the thirteenth struct and the one that ties the architecture together. Everything that crosses a module seam and fails becomes one of these.

Listing 21.26: `domain.lat:117–125`

```
export struct LaboratoryError {
    code: fix String,
    message: fix String,
    path: fix any,
    line: fix any,
    column: fix any,
    stderr: fix any,
    exit_code: fix any
}
```

Five of those seven fields are optional, and the constructor encodes a non-obvious contract about two of them.

Listing 21.27: `domain.lat:702–717` — path versus location

```
    let checked_path = _require_optional_path(path, "laboratory_error.path")
    flux checked_line = nil
    flux checked_column = nil
    if line == nil && column != nil {
        assert(false, "laboratory_error.location: line and column must be supplied
    together")
    }
    if line != nil && column == nil {
        assert(false, "laboratory_error.location: line and column must be supplied
    together")
    }
    if line != nil {
        if checked_path != nil {
            assert(false, "laboratory_error.location: path conflicts with line and
    column")
        }
        checked_line = _require_positive_int(line, "laboratory_error.line")
        checked_column = _require_positive_int(column, "laboratory_error.column")
    }
```

An error points at *either* a request field or a source location, never both. That is the engine's contract expressed as a domain invariant: an `invalid_value` error has a JSON path like `$.projectile.mass_kg`; an `invalid_json` error has a line and a column in the bytes you sent. A response claiming both is malformed and gets rejected before a value is built.

Listing 21.28: The location rules, verified

```
path AND line/column   : assertion failed: laboratory_error.location:
  path conflicts with line and column
line without column    : assertion failed: laboratory_error.location:
  line and column must be supplied together
line zero              : assertion failed: laboratory_error.line:
  must be greater than zero
```

And this is what the two well-formed shapes look like once the renderer has them:

Listing 21.29: Two errors, rendered

```
Error
  code: invalid_value
  message: value must be finite and greater than zero
  path: $.projectile.mass_kg
  exit code: 2
Error
  code: invalid_json
  message: expected value
  location: 3:12
  exit code: 2
```

Why not exceptions all the way?

The lab throws internally and returns values at seams, and both halves of that are deliberate. Inside a constructor, a throw is the right tool: it aborts construction at the point of failure and no partially built value escapes. Across a seam, a value is the right tool: the caller is a REPL that must print something and get back to the prompt, or a session that must leave prior state untouched. `LaboratoryError` is the adapter between the two disciplines, and the three places that perform the conversion are `backend.lat`, `commands.lat`, and `process_backend.lat`.

21.11 Idioms Worth Stealing

21.11.1 Eight default parameters

Listing 21.30: `domain.lat:364–375` — the shot signature

```
export fn shot(
muzzle_velocity: any,
max_range: any,
zero_distance: any = nil,
muzzle_angle: any = nil,
aim_azimuth: any = nil,
shot_azimuth: any = nil,
shooting_angle: any = nil,
cant_angle: any = nil,
target_height: any = nil,
ground_threshold: any = nil
) -> Shot {
```

Two required parameters and eight defaulted ones. `shot(fps(2700), yd(1000))` is a valid call, and so is the ten-argument version. Without defaults this would be a builder or an options `Map`, and either would lose the per-parameter type checking that makes the constructor useful.

The cost shows up when you want the tenth parameter and not the third through ninth: you write seven `nil`s. That is real, and it is visible in the lab's own test code.

21.11.2 `flux` accumulators inside pure functions

Listing 21.31: `domain.lat:274–277` — a one-slot accumulator

```
    flux checked_direction = nil
    if twist_direction != nil {
        checked_direction = _require_choice(twist_direction, ["left", "right"],
    "rifle.twist_direction")
    }
```

`flux` declares a mutable binding. Used this way — one local, assigned at most once, immediately consumed by the struct literal — it is the idiomatic LATTICE answer to "optional value that needs validating only when present." The function is still pure; the mutability never leaves the frame.

The same shape appears at `domain.lat`:310 for humidity, :417 and :421 for solver method and time step, :600 for stability factor, and :691–704 for the error fields.

21.11.3 Optional means absent, not null

Every optional domain field defaults to `nil` and stays `nil`. The protocol layer then simply omits the key — look back at the wire request in Chapter 20: there is no `muzzle_angle_rad`, because the shot did not have one. `persistence.lat` does the opposite, writing explicit `null`s, because a saved document should record that a field was considered and left unset. Two layers, two conventions, each right for its consumer.

21.12 What the Domain Refuses to Do

Worth listing, because each absence is a decision.

It does not convert units. `units.distance_as` exists and `domain.lat` never calls it. A domain value stores whatever the caller gave it, in SI plus their chosen display metadata; conversion happens in `analysis.lat` and `render.lat` at the point of use.

It does not know about the engine. There is no `engine_version` validation, no schema-version vocabulary beyond "must be exactly 1," no notion of a process. `Trajectory` takes an `engine_version` string and checks only that it is non-empty — the README calls it "an opaque non-empty implementation identifier," and the domain honours that.

It does not format anything. No `to_string` of a quantity, no rounding, no unit suffixes. Error messages are the sole exception, and they are diagnostics rather than output.

It does not have an update method. There is no `with_muzzle_velocity`. Changing one field of an `Experiment` means calling `experiment()` again with six of the old components and one new one — which re-runs every relational check. `session.lat` does exactly that in `_experiment_with`, and it is why changing your zero distance can produce an error about your muzzle angle.

The name comes along for the ride

`_experiment_with` always passes `current.name` forward. Every `sessions.replace_*` call therefore keeps the old experiment name, no matter how completely you have replaced the load underneath it. A session that started as "175 gr match at 1,000 yards" will still say so after you have swapped the bullet, the rifle, the atmosphere, the wind, and the shot. Renaming requires a whole-experiment replacement through `sessions.replace_experiment`. This is a real trap for anyone building a lab notebook on top of the session API.

The next chapter takes the module that has to trust none of this: 860 lines whose entire job is to decide whether 23 KB of JSON from a child process deserves to become one of the values described here.

Chapter 22

The Process Backend

Eight hundred and sixty lines exist to answer one question: does this JSON deserve to become a `Trajectory`?

`process_backend.lat` is the largest non-rendering module in the Ballistics Lab, and the shape of it is the whole argument. Roughly forty lines serialize a request and spawn a child. Roughly six hundred validate what comes back. Roughly two hundred are scalar predicates and configuration. There is no parsing code beyond one call to `json_parse`, and there is no physics at all.

The module has two jobs and treats them as completely different problems. Getting the request out is a *security* problem: user values must never become executable. Getting the response in is a *trust* problem: a child process may be a different version, a wrapper script, a symlink to something else entirely, or simply broken, and none of those may be allowed to produce a plausible wrong answer.

This chapter takes both apart, with everything verified by running it.

22.1 The Seam

Before the process backend, the thing it implements. `backend.lat` is 85 lines and defines the entire engine interface.

Listing 22.1: `backend.lat:12–15`

```
export struct EngineBackend {
    name: fix String,
    solve_fn: fix Fn
}
```

A name and a closure. That is the interface. `solve_with` is the only way to invoke one, and it makes a hard guarantee: the caller sees a `Trajectory` or a `LaboratoryError`, never anything else and never a throw.

Listing 22.2: backend.lat:45–77 — the entire contract

```
export fn solve_with(backend: any, experiment: any) -> any {
    if !_is_struct(backend, "EngineBackend") {
        return laboratory_error(
        "backend_contract_error",
        "solve_with: expected EngineBackend, got " + typeof(backend)
        )
    }
    if !_is_struct(experiment, "Experiment") {
        return laboratory_error(
        "backend_contract_error",
        "solve_with: expected Experiment, got " + typeof(experiment)
        )
    }

    let solve_fn = backend.solve_fn
    let result = try {
        solve_fn(experiment)
    } catch error {
        return laboratory_error(
        "backend_contract_error",
        "backend '" + backend.name + "' failed: " + error.message
        )
    }

    if _is_struct(result, "Trajectory") || _is_struct(result, "LaboratoryError") {
        return result
    }
    return laboratory_error(
    "backend_contract_error",
    "backend '" + backend.name + "' returned " + typeof(result) +
    "; expected Trajectory or LaboratoryError"
    )
}
```

Four checks: the backend is a backend, the argument is an experiment, the closure did not throw, and the closure returned one of the two legal shapes. A backend that returns a `Map` produces an error naming the offending `typeof`, not a crash three layers up.

Closures as an interface

LATTICE has traits, and the lab does not use one here. A backend is a struct holding a lambda, checked at construction with `typeof(solve_fn) != "Closure"`. The benefit is that a test backend is five lines with no declaration ceremony, and the process backend — 860 lines of it — is still just a function that happens to return one of these structs.

`backend.lat` also contains one of the most instructive comments in the codebase, three lines that explain a *missing* `freeze`:

Listing 22.3: `backend.lat:36–38`

```
// The fields themselves are alloy `fix` fields. Avoid crystallizing a
// closure-bearing struct: closures may capture independently frozen config,
// and nesting those shared regions in another crystal is unnecessary.
```

The process backend's closure captures a validated executable path, a validated argv array, and a validated limits `Map`. Freezing the struct that holds the closure would drag all of that into a second crystal region for no benefit. The `fix` fields already make the struct unmodifiable.

22.2 No Shell String Is Ever Assembled

Here is the whole transport, in one statement.

Listing 22.4: `process_backend.lat:810–817`

```
let result = try {
    exec_argv(executable, argv, request, limits)
} catch error {
    return laboratory_error(
    "process_error",
    error.message
    )
}
```

exec_argv

`exec_argv(program, args, stdin, options)` runs a program directly, without a shell. `args` is an array of strings that becomes the child's argv from index 1 onward. `stdin` is written to the child's standard input, or `nil` to close it immediately. `options` is a `Map` of positive integers bounding the run. It returns a `Map` with `stdout`, `stderr`, and `exit_code`.

The LATTICE runtime implements this with `posix_spawnp` on POSIX systems. The argv array is built element by element from the LATTICE array, with the program name at index 0, and handed to the kernel directly:

Listing 22.5: `src/process_ops.c:860–861 and :944`

```
argv[0] = args[0].as.str_val;
for (size_t i = 0; i < supplied_argc; i++)
    argv[i + 1] = args[1].as.array.elems[i].as.str_val;
...
posix_spawnp(&child, argv[0], &actions,
             attributes_initialized ? &attributes : NULL, argv, environ);
```

No string is concatenated. No `/bin/sh` is involved. There is nothing for a metacharacter to mean.

LATTICE does have a shell builtin — the lab does not use it

`shell(cmd)` exists, takes a command string, and runs it through the system shell. Grepping the entire lab — eighteen library modules, eighteen test modules, six examples, and the fake engine — finds zero calls to it, and zero calls to `exec`. `exec_argv` appears in exactly one library module, at `process_backend.lat:811`. That is the whole process surface of the application.

22.2.1 Proving it

A claim about injection is worth exactly as much as the experiment that tries to break it. Here is the experiment.

Listing 22.6: A probe that tries three shell attacks

```
// 1. A metacharacter-laden ARGUMENT is passed through verbatim.
let r1 = exec_argv("/bin/echo",
    ["; touch MARKER ;", "$(touch MARKER)", "`touch MARKER`"], nil)
print("argv exit_code: " + to_string(r1.get("exit_code")))
print("argv stdout   : " + r1.get("stdout").trim())

// 2. A metacharacter-laden PROGRAM name is not a command line.
let r2 = try {
    exec_argv("/bin/echo hi; touch MARKER", [], nil)
} catch error {
    "THREW: " + error.message
}
print("program probe : " + to_string(r2))

// 3. Did anything create the marker?
print("marker exists : " + to_string(file_exists("MARKER")))
```

Listing 22.7: The result

```
$ ./clat shellprobe.lat
argv exit_code: 0
argv stdout   : ; touch MARKER ; $(touch MARKER) `touch MARKER`
program probe : THREW: exec_argv: failed to spawn
                '/bin/echo hi; touch MARKER': No such file or directory
marker exists : false
$ ls MARKER
ls: MARKER: No such file or directory
```

Three separate attacks, three separate outcomes, all correct:

1. The semicolons, the `$(...)`, and the backticks arrived at `/bin/echo` as literal characters and were echoed back unchanged. A shell would have run three `touch` commands.
2. Putting a command line into the *program* slot does not create a command line. `posix_spawnp` looked for an executable file whose name is literally `/bin/echo hi; touch MARKER`, did not find one, and the call threw.
3. No marker file exists.

22.2.2 And in the lab itself

The property holds at the user surface too. The colon-command grammar accepts quoted path arguments, and those arguments are inert data all the way to the filesystem call:

Listing 22.8: Two paths that look like attacks (directory elided)

```
:save "<dir>/inj/a;touch INJECTED.json"
:save "<dir>/inj/$(whoami).json"
:quit

Saved experiment to <dir>/inj/a;touch INJECTED.json
Saved experiment to <dir>/inj/$(whoami).json
Quit requested.
```

Listing 22.9: What was actually created

```
$ ls -1 inj
$(whoami).json
a;touch INJECTED.json
```

Two files, named exactly what the user typed. No `INJECTED`, no username substitution. The second filename contains a literal dollar sign and parentheses because that is what it was asked to contain.

exec_argv is not a sandbox

The README says this and it needs repeating. Injection is impossible; execution is not restricted. The configured executable runs with the LATTICE process's full operating-system permissions and inherited environment. If you point `process_backend` at a malicious binary, it will do whatever it likes. The guarantee is that *your experiment data* cannot choose which binary that is, or add arguments to it.

22.3 What the Child Actually Sees

We can settle the argv question empirically by replacing the engine with a recorder.

Listing 22.10: A stand-in engine that records its inputs

```
$ cat record-engine.sh
#!/bin/sh
{
  echo "argc=$#"
  i=1
  for a in "$@"; do echo "argv[$i]=$a"; i=$((i+1)); done
  echo "--- stdin ---"
  cat
} > "$RECORD_TO"
echo 'not json'
```

Listing 22.11: Driving the reference experiment through it

```
$ RECORD_TO=record.txt ./clat argv_probe.lat ./record-engine.sh
invalid_engine_response: engine stdout is not exactly one JSON envelope:
json_parse error at position 0: unexpected character
$ head -3 record.txt
argc=1
argv[1]=solve-json
--- stdin ---
$ tail -1 record.txt | wc -c
     780
```

One argument. The literal string `solve-json`. The request — 780 bytes describing a 175-grain G7 bullet, a 24-inch rifle, an atmosphere, a wind, a shot, and a solver — arrived on standard input. Nothing derived from the experiment appears in argv at all, which is why nothing derived from the experiment can be interpreted as an option.

That argv comes from a two-line function:

Listing 22.12: process_backend.lat:849–860 — the ordinary constructor

```
export fn process_backend(
executable: any,
expected_engine_version: any = nil,
options: any = nil
) -> any {
    return process_backend_with_argv(
    executable,
    ["solve-json"],
    expected_engine_version,
    options
    )
}
```

The array literal is the whole argv policy. process_backend_with_argv exists so tests can point the backend at ./clat with the fake engine's path as an argument, and so an adapter can wrap the engine in a launcher — but even that seam takes an array of strings, never a command line.

22.4 Bounding the Request

Before any child is spawned, _solve_process does two things that can fail.

Listing 22.13: process_backend.lat:796–809

```
    let request = try {
        v1_request_json(experiment)
    } catch error {
        return laboratory_error(
        "request_serialization_error",
        "failed to encode solve-json v1 request: " + error.message
        )
    }
    if len(request) > MAX_REQUEST_BYTES {
        return laboratory_error(
        "resource_limit",
        "solve-json request exceeds the 1048576-byte v1 input limit"
        )
    }
```

The order matters and is checkable. A request that exceeds one mebibyte must fail *without* spawning anything — otherwise a pathological experiment becomes a way to hand an unbounded write to a child process.

To test that, build experiments with an increasing number of wind segments and point the backend at an executable that does not exist. If the size check fires first, we get `resource_limit`; if the spawn is attempted, we get `process_error`. The transition tells us the order.

Listing 22.14: The size check versus the spawn, measured

```
$ ./clat bounds_probe.lat
12000 segments, 973251 request bytes => process_error: exec_argv: failed to
  spawn './no-such-engine': No such file or directory
13000 segments, 1055251 request bytes => resource_limit: solve-json request
  exceeds the 1048576-byte v1 input limit
14000 segments, 1137251 request bytes => resource_limit: solve-json request
  exceeds the 1048576-byte v1 input limit
```

At 973,251 bytes the backend tried to run the program. At 1,055,251 bytes it did not. The limit is enforced before the process boundary, exactly as written.

Making a million-byte request

Thirteen thousand wind segments is not a realistic load, and that is fine — the point of the probe is to find the boundary, not to model a shot. Each segment serializes to about 81 bytes of JSON, and the domain requires strictly increasing boundaries, so a `while` loop pushing `wind(wind_mps(4.0), rad(1.5707963267948966), nil, m(i * 10.0))` produces a valid experiment of any size you like.

22.5 Bounding the Child

Three limits are passed to `exec_argv` on every solve.

Listing 22.15: process_backend.lat:25–34 — the constants

```
let SCHEMA_VERSION = 1
let MAX_REQUEST_BYTES = 1048576
let MAX_SAMPLES = 10000
let DEFAULT_TIMEOUT_MS = 30000
let DEFAULT_MAX_STDOUT_BYTES = 8388608
let DEFAULT_MAX_STDERR_BYTES = 262144
// Match the solve-json fixture comparison convention: a small absolute floor
// plus a tight relative allowance for differently rounded finite JSON numbers.
let SUMMARY_ABSOLUTE_TOLERANCE = 0.0000000001
let SUMMARY_RELATIVE_TOLERANCE = 0.000000001
```

Thirty seconds of wall clock, 8 MiB of captured stdout, 256 KiB of captured stderr. `_limits` builds the `Map`, applies any caller overrides, and rejects anything it does not recognize.

Listing 22.16: process_backend.lat:765–787

```
fn _limits(options: any) -> Map {
    flux limits = Map::new()
    limits.set("timeout_ms", DEFAULT_TIMEOUT_MS)
    limits.set("max_stdout_bytes", DEFAULT_MAX_STDOUT_BYTES)
    limits.set("max_stderr_bytes", DEFAULT_MAX_STDERR_BYTES)
    if options == nil {
        return limits
    }
    if typeof(options) != "Map" {
        assert(false, "process_backend.options: expected Map or nil, got " +
    typeof(options))
    }
    for key in options.keys() {
        if !["timeout_ms", "max_stdout_bytes", "max_stderr_bytes"].contains(key) {
            assert(false, "process_backend.options: unknown option '" + key + "'")
        }
        let value = options.get(key)
        if typeof(value) != "Int" || value <= 0 {
            assert(false, "process_backend.options." + key + ": expected a positive
    Int")
        }
        limits.set(key, value)
    }
    return limits
}
```

A typo in an option name is an error at backend-construction time, not a silently ignored setting. That is the same fail-closed instinct that runs through the response decoder.

All three limits are real. Driving the lab's fake engine with deliberately tight bounds triggers each of them:

Listing 22.17: Every transport limit, triggered

```
$ ./clat transport_probe.lat
timeout (250 ms budget) => process_error: exec_argv: timed out after 250 ms
stdout cap (1024 bytes) => process_error: exec_argv: stdout exceeded
  max_stdout_bytes (1024 bytes)
stderr cap (1024 bytes) => process_error: exec_argv: stderr exceeded
  max_stderr_bytes (1024 bytes)
```

All three surface as `process_error`, because all three are transport failures rather than protocol failures. The distinction matters when you are debugging: `process_error` means the child misbehaved as a process, `invalid_engine_response` means it misbehaved as an engine.

22.6 Four Layers

The module is built in four bands, and reading it top to bottom is reading it in dependency order.

Lines	Band	Contents
36–139	Scalar predicates	`_fail`, `_nonempty_string`, `_integer`, `_number`, `_boolean`, `_map`, `_array`, `_required`, `_require_exact_keys`, `_require_choice`.
141–329	Request validators	Nine section validators plus `_validate_resolved_request`, which also re-checks two cross-object relationships.
331–504	Payload decoders	Notices, samples, summary, the float comparison, the terminal-sample audit.
506–860	Envelope and transport	Error decoding, success decoding, the exported decode seam, argv copying, limits, and the solve function.

Everything in bands 1–3 throws on failure. Only band 4 catches, and it catches in exactly three places. That separation is why the validators can be written as straight-line code with no error plumbing: they are allowed to be optimistic because someone above them is not.

22.7 Closed Schemas

The single most-used function in the module is eleven lines long.

Listing 22.18: process_backend.lat:120–131

```
fn _require_exact_keys(object: Map, required: Array, optional: Array, path: String) {
    for key in required {
        if !object.has(key) {
            _fail(path + ": missing required field '" + key + "'")
        }
    }
    for key in object.keys() {
        if !required.contains(key) && !optional.contains(key) {
            _fail(path + ": unknown field '" + key + "'")
        }
    }
}
```

Two loops. The first rejects missing fields; the second rejects unknown ones. An object must match its declared schema exactly — no extra keys, ever, at any depth.

That second loop is the interesting half, and it is the one most protocol implementations skip. Ignoring unknown fields is the friendly, forward- compatible choice, and it is the wrong one here. A future engine that adds `summary.coriolis_drift_m` would be sending information the lab does not understand, in a response the lab would otherwise accept and render as complete. Rejecting it forces the version conversation to happen.

Listing 22.19: An unknown envelope field and an unknown error field

```
unknown-field       => invalid_engine_response: assertion failed: solve-json
  v1 response: $: unknown field 'future_field'
unknown-error-field => invalid_engine_response: assertion failed: solve-json
  v1 response: $.error: unknown field 'future'
```

The path in each message — `$` for the envelope, `$.error` for the error object — comes from the `path` argument threaded through every validator. Sixty-odd call sites pass a JSON path literal, and the result is that every rejection message tells you exactly where in the document the problem is.

22.8 Validate Everything, Then Construct

`_decode_success` is the spine of the module, and its ordering is the design.

Listing 22.20: process_backend.lat:598–628 — ordering is the design

```
    _require_exact_keys(
    envelope,
    [
        "schema_version", "engine_version", "status", "resolved_request",
        "assumptions", "warnings", "summary", "samples"
    ],
    [],
    "$"
    )
    if exit_code != 0 {
        _fail("status 'ok' requires exit 0, got " + to_string(exit_code))
    }
    if len(stderr) != 0 {
        _fail("status 'ok' must not write diagnostics to stderr")
    }
    let engine_version = _nonempty_string(envelope.get("engine_version"),
    "$.engine_version")
    if expected_engine_version != nil && engine_version != expected_engine_version {
        return laboratory_error(
        "incompatible_engine_version",
        "expected engine version '" + expected_engine_version + "', got '" +
    engine_version + "'",
        nil,
        _diagnostic(stderr),
        exit_code
        )
    }
    let resolved = _validate_resolved_request(envelope.get("resolved_request"))
    let assumptions = _decode_notices(envelope.get("assumptions"), "assumptions")
    let warnings = _decode_notices(envelope.get("warnings"), "warnings")
    let summary = _decode_summary(envelope.get("summary"))
    let samples = _decode_samples(envelope.get("samples"))
    _validate_terminal_sample(samples, summary)
```

Eight checks before the constructor call on the following line. The cheap, whole-document checks run first: key set, exit code, stderr, version. Then the resolved request, then the notices, then the summary, then the samples, then the cross-check between the last sample and the summary. Only after all of that does `domain.trajectory(...)` run.

Why a successful solve must not write to stderr

`status 'ok' must not write diagnostics to stderr` looks harsh until you consider what stderr on a success would mean: the engine had something to say that the protocol has no field for. The protocol has `assumptions` and `warnings` for exactly that purpose. A banner, a deprecation notice, or a library's debug output on stderr is evidence that the child is not the program the lab thinks it is talking to.

22.8.1 The two-phase sample decode

Ten thousand samples are the protocol maximum, and decoding them has a subtlety the source comments on directly.

Listing 22.21: `process_backend.lat:410–428`

```
fn _decode_samples(value: any) -> Array {
    let input = _array(value, "$.samples")
    if len(input) == 0 {
        _fail("$.samples: must contain a terminal observation")
    }
    if len(input) > MAX_SAMPLES {
        _fail("$.samples: exceeds the 10000-sample v1 limit")
    }
    flux index = 0
    for item in input {
        _validate_sample(item, index)
        index = index + 1
    }
    let output = freeze(input.enumerate().map( |pair| {
        return _decode_checked_sample(pair[1])
    }
    ))
    return output
}
```

The length check comes first, before a single element is touched — an array of 10,001 items is rejected on its length, so a hostile response cannot cost 10,001 validations. Then a full validation pass. Then, and only then, a construction pass.

Listing 22.22: `process_backend.lat:393–397` — why the phases are separate

```
fn _decode_checked_sample(value: any) -> any {
    // _decode_samples validates every element before Array.map invokes this
    // construction-only path, so backend callback errors cannot leave a
    // partially built native map result.
    let object = value
```

The reason is mechanical: `Array.map` is a native operation, and a throw from inside the callback would abandon a partially built native result. Doing all the throwing in a plain `for` loop first means the map callback is guaranteed not to fail.

This is also the only place in the entire lab that uses functional-style iteration. Every other loop is `for`, `while`, or `loop`.

Listing 22.23: The sample ceiling, at the boundary

```
sample-limit-exact    => OK, engine 0.24.1, 10000 observations
sample-limit-exceeded => invalid_engine_response: assertion failed: solve-json
  v1 response: $.samples: exceeds the 10000-sample v1 limit
```

Ten thousand is accepted. Ten thousand and one is not.

22.9 Comparing Floats

The summary and the last sample describe the same instant of flight, and they must agree. They are also two independently serialized sets of JSON numbers, so “agree” cannot mean bitwise equality.

Listing 22.24: process_backend.lat:465–477

```
fn _numbers_match(left: Number, right: Number) -> Bool {
    flux scale = abs(left)
    if abs(right) > scale {
        scale = abs(right)
    }
    return abs(left - right) <= SUMMARY_ABSOLUTE_TOLERANCE +
    SUMMARY_RELATIVE_TOLERANCE * scale
}

fn _require_summary_match(sample_value: Number, summary_value: Number, field: String) {
    if !_numbers_match(sample_value, summary_value) {
        _fail("$.samples: terminal " + field + " does not match $.summary")
    }
}
```

An absolute floor of 10^{-10} plus a relative allowance of 10^{-9} scaled to the larger magnitude. The absolute term keeps the test meaningful near zero — a drop of exactly 0 and a drop of 10^{-15} should compare equal — and the relative term keeps it meaningful at 900 metres, where an absolute tolerance of 10^{-10} would be tighter than double precision can honour.

Work the numbers for the lab's own test fixture, whose terminal distance is 50 m:

$$10^{-10} + 10^{-9} \times 50 = 5.0000000001 \times 10^{-8}\ \text{m}$$

About fifty nanometres. The fake engine has two scenarios that probe exactly this: one perturbs `actual_range_m` from `50.0` to `50.00000001` — a difference of 10^{-8} m, well inside the window — and one perturbs it to `51.0`.

Listing 22.25: Inside and outside the tolerance

```
summary-within-tolerance   => OK (6 samples)
summary-distance-mismatch  => invalid_engine_response: assertion failed:
  solve-json v1 response: $.samples: terminal distance_m does not match $.summary
```

All four terminal quantities are checked — distance, time, speed, and energy — and all four have their own scenario:

Listing 22.26: The other three mismatches

```
summary-time-mismatch   => invalid_engine_response: ... $.samples: terminal
  time_s does not match $.summary
summary-speed-mismatch  => invalid_engine_response: ... $.samples: terminal
  speed_mps does not match $.summary
summary-energy-mismatch => invalid_engine_response: ... $.samples: terminal
  energy_j does not match $.summary
```

The terminal audit checks structure as well as numbers.

Listing 22.27: process_backend.lat:479–504

```
fn _validate_terminal_sample(samples: Array, summary: any) {
    flux terminal_count = 0
    flux terminal_index = -1
    flux index = 0
    for sample in samples {
        for flag in sample.flags {
            if flag == "terminal" {
                terminal_count = terminal_count + 1
                terminal_index = index
            }
        }
        index = index + 1
    }
    if terminal_count != 1 {
        _fail("$.samples: expected exactly one terminal flag, got " +
    to_string(terminal_count))
    }
    if terminal_index != len(samples) - 1 {
        _fail("$.samples: the terminal observation must be last")
    }

    let terminal = samples[len(samples) - 1]
    _require_summary_match(terminal.distance.si_value, summary.actual_range.si_value,
    "distance_m")
    _require_summary_match(terminal.time_s, summary.time_of_flight_s, "time_s")
    _require_summary_match(terminal.velocity.si_value,
    summary.terminal_velocity.si_value, "speed_mps")
    _require_summary_match(terminal.energy.si_value, summary.terminal_energy.si_value,
    "energy_j")
}
```

Exactly one terminal flag, on the last sample. Three fake-engine scenarios attack that invariant from three directions:

Listing 22.28: Terminal-flag attacks

```
missing-terminal   => invalid_engine_response: ... $.samples: expected exactly
  one terminal flag, got 0
early-terminal     => invalid_engine_response: ... $.samples: the terminal
  observation must be last
multiple-terminal  => invalid_engine_response: ... $.samples: expected exactly
  one terminal flag, got 2
empty-samples      => invalid_engine_response: ... $.samples: must contain a
  terminal observation
```

22.10 Exit Status Must Agree With the Error Code

When the engine reports a protocol error, it says so twice — once in the JSON envelope's `error.code`, once in its process exit status. The lab requires the two to agree.

Listing 22.29: `process_backend.lat:530–544` — the code-to-status table

```
fn _expected_error_exit(code: String) -> Int {
    if [
        "invalid_json", "unsupported_schema_version", "unknown_field", "missing_field",
        "invalid_value", "conflicting_fields"
    ].contains(code) {
        return 2
    }
    if ["resource_limit", "solve_failed"].contains(code) {
        return 3
    }
    if ["io_error", "internal_error"].contains(code) {
        return 1
    }
    _fail("$.error.code: unsupported value '" + code + "'")
}
```

Exit 2 for anything wrong with the request. Exit 3 for a resource limit or a solve that could not converge. Exit 1 for I/O and internal failures. A code the table does not know is itself a failure — the fall-through calls `_fail`.

A disagreement is a hard error, not a warning:

Listing 22.30: The engine claiming one thing and exiting another

```
exit-mismatch => invalid_engine_response: assertion failed: solve-json v1
  response: $.error.code 'invalid_value' requires exit 2, got 3
```

That check catches a whole class of problem no single-signal check would. A wrapper script that swallows the child's exit status, a shell that reports its own status instead, a version whose exit codes drifted — each shows up as a coupling violation rather than as a subtly wrong error report.

Well-formed error envelopes, by contrast, pass straight through with everything intact:

Listing 22.31: Four genuine engine errors, decoded

```
validation-error => invalid_value: value must be finite and greater than zero
located-error    => invalid_json: expected value
resource-error   => resource_limit: trajectory observation limit of 10000
  exceeded (would produce 50001)
internal-error   => internal_error: solve-json command failed unexpectedly
```

The engine's own `code` survives. So do its `path` or its `line`/`column`, its stderr text, and its exit code. The lab does not rewrite the engine's diagnosis; it only refuses to accept one that is internally inconsistent.

22.11 The Error Taxonomy

Everything above produces one of a small set of codes. The lab's fake engine ships forty-five named scenarios plus a parameterized `sample-count-N` family; driving them through `process_backend` and printing the resulting code and message gives the complete map.

Code	Means
engine's own code	A valid v1 error envelope. `invalid_value`, `invalid_json`, `resource_limit`, `internal_error`, and the rest of the protocol vocabulary pass through unchanged.
`invalid_engine_response`	The response was not a valid v1 document. The catch-all, and by far the most common.
`unsupported_schema_version`	The envelope parsed and declared a schema version other than 1.

Code	Means
`incompatible_engine_version`	A version pin was set and the engine reported a different one.
`process_error`	The child failed as a process: spawn failure, timeout, output-limit overflow.
`resource_limit`	The *request* exceeded 1 MiB, before spawning.
`request_serialization_error`	The experiment could not be encoded at all.
`backend_contract_error`	Not from this module: `backend.lat` caught a throw or a wrong return shape.

The interesting rows are the ones that collapse many distinct failures into `invalid_engine_response`. Here is that band, verbatim from the probe run, wrapped to page width.

Listing 22.32: Framing attacks — the response is not exactly one JSON document

```
contaminated           => invalid_engine_response: engine stdout is not exactly
  one JSON envelope: json_parse error at position 0: unexpected character
trailing               => invalid_engine_response: ... json_parse error at
  position 3692: unexpected trailing content
concatenated           => invalid_engine_response: ... json_parse error at
  position 3692: unexpected trailing content
truncated              => invalid_engine_response: ... json_parse error at
  position 34: expected ',' or '}' in object
duplicate-envelope-key => invalid_engine_response: ... json_parse error at
  position 88: duplicate object key
oversized-integer      => invalid_engine_response: ... json_parse error at
  position 41: integer out of signed 64-bit range
```

Note how much work Lattice's `json_parse` is doing here. A banner printed before the envelope fails at position 0. A second envelope appended after the first fails as trailing content. A duplicate key inside the object fails as a duplicate key — many JSON parsers accept those and silently keep one. An integer too large for 64 bits fails rather than being coerced to a float. The lab gets all of that for free by requiring that stdout parse as exactly one document.

Listing 22.33: Non-finite numbers, at three depths

```
nonfinite          => invalid_engine_response: assertion failed: solve-json v1
  response: $.summary.actual_range_m: must be finite
nonfinite-resolved => invalid_engine_response: ... $.resolved_request.
  projectile.mass_kg: must be finite
nonfinite-sample   => invalid_engine_response: ... $.samples[0].mach: must be
  finite
```

Listing 22.34: The resolved request, attacked five ways

```
invalid-resolved-altitude   => ... $.resolved_request.atmosphere.altitude_m:
  must be between -5000 and 84000
invalid-resolved-ground     => ... $.resolved_request.shot.ground_threshold_m:
  must be below the resolved muzzle height
invalid-resolved-effects    => ... $.resolved_request.effects: magnus and
  enhanced_spin_drift cannot both be enabled
invalid-resolved-coriolis   => ... $.resolved_request.atmosphere.latitude_rad:
  required when coriolis is enabled
invalid-resolved-zero-range => ... $.resolved_request.shot.zero_distance_m:
  too large for a finite zeroing range
```

That last band deserves a moment. The *resolved request* is the engine's statement of what it actually solved, after applying every default. The lab does not merely record it — it validates it against the same rules `domain.lat` applies to your input, including the relational ones. An engine that resolved Coriolis on without a latitude, or that filled in a ground threshold above the muzzle, is caught here.

Listing 22.35: `process_backend.lat:322–328` — the relational checks, on the way back

```
    if shot.get("ground_threshold_m") >= rifle.get("muzzle_height_m") {
        _fail("$.resolved_request.shot.ground_threshold_m: must be below the resolved
    muzzle height")
    }
    if effects.get("coriolis") && !atmosphere.has("latitude_rad") {
        _fail("$.resolved_request.atmosphere.latitude_rad: required when coriolis is
    enabled")
    }
    return freeze(object)
```

Finally, the protocol-level shape failures:

Listing 22.36: Envelope shape and version

```
unknown-schema  => unsupported_schema_version: unsupported response
  schema_version 2; expected 1
unknown-status  => invalid_engine_response: assertion failed: solve-json v1
  response: $.status: unsupported value 'future'
success-stderr  => invalid_engine_response: assertion failed: solve-json v1
  response: status 'ok' must not write diagnostics to stderr
malformed-error-location => invalid_engine_response: ... $.error: path
  conflicts with line and column
```

Two scenarios that pass

Not everything unusual is rejected. `notice-path-absent` and `notice-path-null` both succeed: an assumption may omit its `path` key entirely, or supply it as JSON `null`. Both mean "this notice is not tied to a request field," and `_decode_notice` treats them identically:

```
    flux field_path = nil
    if object.has("path") && object.get("path") != nil {
        field_path = _nonempty_string(object.get("path"), path + ".path")
    }
```

Strictness is about the shapes that could change an answer, not about punishing every degree of freedom the protocol allows.

22.12 Version Pinning

`engine_version` is an opaque string. The lab never parses it, never compares it for ordering, and never infers capability from it. It does exactly one thing: if you supplied a pin, the versions must match exactly.

Listing 22.37: Pinning, verified against a fake engine reporting 0.24.1

```
$ ./clat pin_probe.lat
no pin, engine 0.24.1     => OK, engine 0.24.1, 6 observations
pin 0.24.1                => OK, engine 0.24.1, 6 observations
pin 0.25.0                => incompatible_engine_version: expected engine
  version '0.25.0', got '0.24.1'
no pin, wrong-version     => OK, engine 9.9.9, 6 observations
pin 0.24.1, wrong-version => incompatible_engine_version: expected engine
  version '0.24.1', got '9.9.9'
```

Line four is the one to read twice. Without a pin, an engine reporting version `9.9.9` is accepted, because the compatibility boundary is `schema_version`, not `engine_version`. A pin is a *reproducibility* tool, not a safety one: it lets a saved artifact assert which engine produced it. Strict schema validation happens either way.

The REPL and the launcher read the pin from an environment variable:

Listing 22.38: Pinning a whole session, against a real 0.32.0 engine

```
$ BALLISTICS_ENGINE_VERSION=0.32.0 sh scripts/ballistics-lab.sh <<'IN'
:solve
:quit
IN
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.32.0
  schema           : 1
  verified         : no
  ...
$ BALLISTICS_ENGINE_VERSION=0.30.1 sh scripts/ballistics-lab.sh <<'IN'
:solve
:quit
IN
Error
  code: incompatible_engine_version
  message: expected engine version '0.30.1', got '0.32.0'
  exit code: 0
Quit requested.
```

That second run is worth keeping in mind if you are reproducing this book's numbers. The engine on the machine these transcripts came from reports `0.32.0`, and a pin for anything else stops the session at the first solve.

An empty value disables it. The lab's own end-to-end REPL test sets that variable explicitly, so child processes pin the fixture contract rather than inheriting whatever version a developer happens to have installed.

22.13 Three Catches

In 860 lines there are exactly three `try`/`catch` blocks in the response path, and their placement is the module's error-handling design.

Listing 22.39: `process_backend.lat:727–741` — catches two and three

```
let envelope = try {
    json_parse(stdout)
} catch error {
    return _invalid_response(
    "engine stdout is not exactly one JSON envelope: " + error.message,
    stderr,
    exit_code
    )
}
return try {
    _decode_envelope(envelope, experiment, expected_engine_version, stderr,
exit_code)
} catch error {
    _invalid_response(error.message, stderr, exit_code)
}
```

The first catch is around the `exec_argv` call itself, producing a `process_error`. The second is around `json_parse`. The third wraps the *entire* decode tree — every validator, every decoder, every domain constructor call — and turns any throw into one `invalid_engine_response` carrying the thrown message.

That is why the messages in the taxonomy tables read `assertion failed: solve-json v1 response: $.samples[0].mach: must be finite`. Three layers are visible in one string: the outer code (`invalid_engine_response`), the `_fail` prefix identifying the protocol, and the specific JSON path and rule. One catch, six hundred lines of validators, and no error plumbing in between.

Listing 22.40: process_backend.lat:36–38 — the prefix that makes it readable

```
fn _fail(message: String) {
    assert(false, "solve-json v1 response: " + message)
}
```

22.14 The Configuration Closure

The last piece is how a backend gets built.

Listing 22.41: process_backend.lat:821–847

```
export fn process_backend_with_argv(
executable: any,
argv: any,
expected_engine_version: any = nil,
options: any = nil
) -> any {
    if typeof(executable) != "String" || len(executable.trim()) == 0 {
        assert(false, "process_backend.executable: expected a non-empty String")
    }
    if expected_engine_version != nil {
        if typeof(expected_engine_version) != "String" ||
    len(expected_engine_version.trim()) == 0 {
            assert(false, "process_backend.expected_engine_version: expected a
    non-empty String or nil")
        }
    }
    let checked_argv = _copy_argv(argv)
    let checked_limits = _limits(options)
    return engine_backend("process-json-v1", |experiment| {
        return _solve_process(
        executable,
        checked_argv,
        checked_limits,
        expected_engine_version,
        experiment
        )
    }
    )
}
```

Everything is validated *before* the closure is created. By the time the lambda exists, `executable` is a non-empty string, `checked_argv` is an array of strings that the caller does not have a reference to, and `checked_limits` is a `Map` of positive integers with no unknown keys. The closure captures those bindings and can never see anything else.

Misconfiguration therefore fails at construction, loudly, in the caller's frame — not on the first solve, three commands later, inside a `try`.

Listing 22.42: `process_backend.lat:743–763` — the defensive copy

```
fn _copy_argv(argv: any) -> Array {
    if typeof(argv) != "Array" {
        assert(false, "process_backend.argv: expected Array, got " + typeof(argv))
    }
    if len(argv) == 0 {
        assert(false, "process_backend.argv: must not be empty")
    }
    flux copied = []
    flux index = 0
    for item in argv {
        if typeof(item) != "String" {
            assert(false, "process_backend.argv[" + to_string(index) + "]: expected
    String")
        }
        copied.push(item)
        index = index + 1
    }
    // This is a private defensive copy captured by the backend closure. Keeping
    // it fluid avoids nesting a shared crystal region inside a closure while no
    // caller retains an alias capable of mutation.
    return copied
}
```

The argv array is copied element by element rather than aliased, so a caller who mutates the array they passed in cannot change what the backend will execute. And it is deliberately *not* frozen, for the same reason `backend.lat` does not freeze the struct: the copy is private, nobody else holds a reference, and a crystal region inside a closure would buy nothing.

The backend is stateless between solves

The closure holds configuration, not a process. There is no persistent child, no connection, no handle. Every solve spawns a fresh engine, writes one request, reads one response, and lets it exit. `session.lat` states the invariant in its header: a session "never retains a child-process handle: process_backend's closure remains process-per-solve." The cost, measured in Chapter 20, is about 3.4 ms.

22.15 What This Buys

Step back from the code and count what the module actually guarantees.

- No experiment value can become an argument, an option, or a shell token. Verified by probe, and structurally true because argv is a literal array and the request travels on stdin.
- No response can produce a domain value unless every field of it, at every depth, matched a closed schema with the right types, finite numbers, and legal enumerated values.
- No response can be internally inconsistent: the terminal sample must agree with the summary to within fifty nanometres at fixture scale, the error code must agree with the exit status, and a successful solve must be silent on stderr.
- No child can run longer than the timeout, write more than the stdout cap, or produce more than 10,000 samples.
- No misconfiguration survives past backend construction.
- Every failure is a `LaboratoryError` value with a code, a message, and where possible a JSON path — printable, matchable, and incapable of taking down the REPL.

Eight hundred and sixty lines is a lot of code to write for a program you control, on a protocol you designed, talking to a binary you built. That is precisely the argument for writing it. The lab's decoder is strict about the engine's output because the engine will change — it is already two minor versions ahead of the one this book was planned against — and the day it changes in a way that matters, the lab will name the field and the JSON path rather than quietly rendering a table that looks right and is not.

Chapter 23

Rendering and Analysis

Everything you have read on screen so far in this book came out of three files. `analysis.lat` turns a solved trajectory into rows of numbers. `analysis_export.lat` turns those rows into JSON, CSV, or plain text. `render.lat` turns them into the aligned, signed, unit-labelled blocks the Ballistics Lab prints at the prompt.

Together they are 2,261 lines — `render.lat` alone is 1,031, the largest file in the Ballistics Lab. That is a lot of code for "print a table," and this chapter is about why. The short answer is that a ballistic table is a *measurement report*: it has to say what units it is in, what engine produced it, what the engine assumed, and where the numbers stop being trustworthy. None of that is decoration.

If you are here for the ballistics

You can read §23.2, §23.6, and §23.7 and skip the rest. Those three sections tell you how the Ballistics Lab decides which units to print, exactly what each sign legend means, and the one cosmetic wart that is still in the output. The remainder is for readers who want to see how a 1,000-line formatter is organized in Lattice.

23.1 Three modules, one rule

The rule is stated at the top of each file and it is absolute: **these modules perform no I/O.** They do not print, they do not open files, they do not spawn processes, they do not evaluate source. Every public function returns a value — an `AnalysisTable` or a `String` — and the caller decides what to do with it.

Listing 23.1: The contract, from the head of `render.lat`

```
// Deterministic, terminal-oriented rendering for the ballistics laboratory.
//
// Every public operation is pure and returns a String. This module deliberately
// performs no printing, file/process I/O, JSON/CSV serialization, source
// evaluation, or Map iteration. Analysis rows are read only through their
// declared column sequence.
```

That last sentence — *rows are read only through their declared column sequence* — is the single most important design decision in the rendering stack, and §23.10 shows what happens when you violate it.

The division of labour is:

Module	Input	Output
`analysis.lat`	`Trajectory`	`AnalysisTable` (numbers)
`render.lat`	`AnalysisTable` + profile	`String` (terminal)
`analysis_export.lat`	`AnalysisTable`	`String` (JSON/CSV/text)

`render.lat` and `analysis_export.lat` are siblings, not a chain. They consume the same table and never talk to each other. In fact `analysis_export.lat` imports *nothing at all* — not even `analysis.lat`, whose structs it validates. It checks `struct_name(value) == "AnalysisTable"` and reads the fields it needs.

Export is a source-level capability, not a colon command

There is no `:export` command. `analysis_export.lat` is unreachable from the colon surface; you get at it by writing LATTICE, as `examples/export.lat` does. That is deliberate: the terminal renderer and the machine-readable serializers have different audiences and different change rates, and coupling them through a command would make the colon grammar grow every time a new format appeared.

23.2 The display profile

Every renderer takes two arguments: the thing to render, and a *display profile*. The profile is a `DisplayPreferences` struct with five fields — `distance_unit`, `velocity_unit`, `energy_unit`, `angle_unit`, and `precision`. That is the whole of the BALLISTICS LAB's presentation configuration.

Listing 23.2: Profile constructors, `render.lat:206–216`

```
export fn render_profile_from_display(value: any) -> any {
    return _profile(value)
}

export fn metric_render_profile(precision: any = 2) -> any {
    return sessions.metric_display_preferences(precision)
}

export fn imperial_render_profile(precision: any = 2) -> any {
    return sessions.display_preferences("yd", "ft/s", "ft-lb", "mil", precision)
}
```

The private `_profile` helper does not merely check the struct. It *re-runs* the value through the session module's own `display_preferences`, which revalidates all five fields and returns a fresh frozen copy. A profile handed to a renderer can therefore never be a half-built or externally mutated value. It is also the reason `render.lat` imports `session.lat` — a dependency edge that surprises most readers, since you would expect presentation to depend on nothing.

23.2.1 Unit policy lives in the renderer

Three small functions are exported purely so that the command layer can ask the renderer what units to calculate in:

Listing 23.3: Unit policy, `render.lat:224–243`

```
export fn offset_unit_for(display: any) -> String {
    let profile = _profile(display)
    if _metric_distance_family(profile) {
        return "m"
    }
    return "in"
}

export fn wind_speed_unit_for(display: any) -> String {
    let profile = _profile(display)
    if profile.velocity_unit == "m/s" {
        return "m/s"
    }
    return "mph"
}

export fn direction_unit_for(display: any) -> String {
    _profile(display)
    return "deg"
}
```

An *offset* is a small distance measured across the line of sight — drop, windage, sight height, maximum ordinate. You do not want those in yards. So the profile's `distance_unit` governs ranges, and `offset_unit_for` derives a separate unit for offsets: metres when the range unit is metric, inches otherwise. `direction_unit_for` always returns `"deg"` and ignores the profile entirely; wind directions are compass bearings and nobody reads them in milliradians.

Watch the whole policy change one command at a time. This is a real session — the same solved 140 gr ELD-M load, rendered twice:

Listing 23.4: Imperial then metric, one session (verbatim, provenance blocks elided)

```
:dope 100 500 100 yd
DOPE: 140 gr ELD-M at 1,200 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2582.80 |        2073.37 |     0.11 | 2.32
       200.00 |     +3.41 |         +0.47 |         2458.89 |        1879.20 |     0.23 | 2.21
       300.00 |    +12.58 |         +1.16 |         2338.37 |        1699.50 |     0.36 | 2.10
       400.00 |    +28.10 |         +1.95 |         2221.35 |        1533.66 |     0.49 | 2.00
       500.00 |    +50.68 |         +2.82 |         2107.93 |        1381.04 |     0.63 | 1.90
Signs: drop + below/- above; come-up + correction up/- correction down

sessions.set_display_preferences(lab, sessions.metric_display_preferences(2))
:dope 100 500 100 m
DOPE: 140 gr ELD-M at 1,200 yd
Distance (m) | Drop (m) | Come-up (mil) | Velocity (m/s) | Energy (J) | Time (s) | Mach
-------------+----------+---------------+----------------+------------+----------+-----
      100.00 |    +0.00 |         +0.03 |         783.67 |    2785.76 |     0.12 | 2.31
      200.00 |    +0.12 |         +0.60 |         742.52 |    2500.88 |     0.26 | 2.19
      300.00 |    +0.41 |         +1.38 |         702.61 |    2239.26 |     0.39 | 2.07
      400.00 |    +0.91 |         +2.27 |         664.00 |    1999.98 |     0.54 | 1.96
      500.00 |    +1.62 |         +3.25 |         626.68 |    1781.42 |     0.70 | 1.85
Signs: drop + below/- above; come-up + correction up/- correction down
```

The two tables are not the same rows in different clothes: the first is banded in yards and the second in metres, so the metric table's last row is 547 yd out and its dope is correspondingly larger. What they share is the solve underneath and the convention: the rifle is zeroed at 100 yd, so the imperial table's first row reads `+0.00 in` of drop and `+0.00 mil` of dial, and both columns grow positive downrange.

Four of the seven column headings changed: distance, drop, velocity, and energy. `Come-up` stayed in mil because `metric_display_preferences` keeps `"mil"` as its angle unit, and `Time` is always seconds. `Mach` is dimensionless and carries no unit at all — its `TableColumn.unit` is `nil`, and `_heading()` omits the parenthesis when it sees that.

Altitude is printed in the distance unit

`render_experiment_summary` formats `atmosphere.altitude` with `profile.distance_unit`, not the offset unit. Under the default imperial profile a 250 m field elevation prints as `altitude : 273.40 yd`. It is arithmetically correct and reads strangely; nobody quotes elevation in yards. Convert mentally, or switch to a metric profile when you are checking atmosphere.

23.3 `_fixed`: formatting a number by hand

Lattice has no `printf`. Every fixed-point number in the Ballistics Lab — every `2582.80`, every `+1.80` — comes out of one 38-line function, and it is worth reading in full because it is a small masterclass in what "deterministic" costs.

Listing 23.5: render.lat:130–167

```
fn _fixed(value: any, precision: Int, explicit_sign: Bool = false) -> String {
    let checked = _require_finite(value, "render.number")
    if precision < 0 || precision > 12 {
        _fail("render.precision", "must be between 0 and 12")
    }
    let scale = pow(10, precision)
    let magnitude = abs(checked)
    let safe_magnitude = 9000000000000000000 / scale
    if magnitude > safe_magnitude {
        // Fixed scaling cannot represent this magnitude in an Int. The JSON
        // scalar spelling is deterministic and round-trip-safe on every
        // backend, and avoids making presentation fail for an otherwise valid
        // finite domain value.
        let fallback = json_stringify(checked)
        if explicit_sign && checked >= 0 {
            return "+" + fallback
        }
        return fallback
    }
    // Force floating multiplication before round(), avoiding Int wraparound
    // when a caller supplies an Int scalar.
    let scaled = round(magnitude * 1.0 * scale)
    let whole = scaled / scale
    let remainder = scaled % scale
    flux sign = ""
    // A value that rounds to zero is normalized before applying an explicit
    // sign, so negative zero is always rendered as +0.00 in signed fields.
    if scaled != 0 && checked < 0 {
        sign = "-"
    } else if explicit_sign {
        sign = "+"
    }
    if precision == 0 {
        return sign + to_string(whole)
    }
    let fraction = to_string(remainder)
    return sign + to_string(whole) + "." + "0".repeat(precision - len(fraction)) +
    fraction
}
```

Four details are load-bearing.

Scale, round, split. The value is multiplied by $10^{\text{precision}}$, rounded to an integer, then split into whole part and remainder by integer division and modulo. The remainder is zero-padded on the left. No floating-point formatting library is involved, so the output cannot vary with a runtime's `printf` implementation.

The overflow escape hatch. If the magnitude is too large to survive scaling into a 64-bit integer, `_fixed` gives up on fixed-point and returns `json_stringify(value)` instead. Presentation must not fail on a value the domain layer accepted. You can watch it happen:

Listing 23.6: An observation built at 10^{20}, rendered (verbatim)

```
let big = pow(10.0, 20)
let huge = observation(m(0), big, mps(big), joules(big), m(big), m(0.0 - big), big, ["terminal"])
print(render.render_observation(huge, render.metric_render_profile(2)))
Observation
  distance: 0.00 m
  time    : 1e+20 s
  velocity: 1e+20 m/s
  energy  : 1e+20 J
  drop    : +1e+20 m
  windage : -1e+20 m
  Mach    : 1e+20
  flags   : terminal
Signs: drop + below/- above; windage + right/- left
```

Negative zero is normalized. The sign test is `scaled != 0 && checked < 0`, not `checked < 0`. A small negative number that rounds away to zero prints `+0.00` in a signed field, never `-0.00`. Watch it fire on real data. The shipped 175 gr load is zeroed at 100 yd, so between 50 and 100 yd the bullet is a tenth of an inch *above* the sight line — a genuinely negative drop — and at precision 0 both of those rows print `+0`, not `-0`.

Listing 23.7: One DOPE band at three precisions (verbatim, provenance elided)

```
:dope 50 100 25 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
        50.00 |     +0.11 |         +0.06 |         2605.79 |        2638.05 |     0.06 | 2.33
        75.00 |     -0.11 |         -0.04 |         2559.35 |        2544.90 |     0.09 | 2.29
       100.00 |     +0.00 |         +0.00 |         2513.36 |        2454.23 |     0.12 | 2.25

sessions.set_display_preferences(lab, sessions.display_preferences("yd", "ft/s", "ft-lb", "mil",
    0))
:dope 50 100 25 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
           50 |        +0 |            +0 |            2606 |           2638 |        0 |    2
           75 |        +0 |            +0 |            2559 |           2545 |        0 |    2
          100 |        +0 |            +0 |            2513 |           2454 |        0 |    2

sessions.set_display_preferences(lab, sessions.display_preferences("yd", "ft/s", "ft-lb", "mil",
    5))
:dope 50 100 25 yd
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+--------
     50.00000 |  +0.11267 |      +0.06259 |      2605.78773 |     2638.04603 |  0.05656 | 2.33089
     75.00000 |  -0.10536 |      -0.03902 |      2559.35444 |     2544.90067 |  0.08561 | 2.28935
    100.00000 |  +0.00003 |      +0.00001 |      2513.36454 |     2454.23004 |  0.11517 | 2.24821
```

The 75 yd row is the one to watch: -0.10536 in and -0.03902 mil at precision 5, and `+0` in both columns at precision 0. The 100 yd row is the zero itself, three hundred-thousandths of an inch of interpolation residue away from exactly on the sight line.

Precision changes column widths. Compare the `Mach` rule across those blocks — `-----` at precision 0 and `--------` at precision 5. Widths are computed from the rendered strings, not declared in advance. §23.8 covers the algorithm.

Precision is a session preference, not a command flag

There is no `:precision` command. Set it the same way you set everything else — by evaluating LATTICE at the prompt: `sessions.set_display_preferences(lab, sessions.display_preferences("yd", "ft/s", "ft-lb", "mil", 3))`. Legal precisions are 0 through 12.

23.4 Field blocks

Four of the ten renderers are not tables at all — they are label/value blocks: `render_experiment_summary`, `render_observation`, `render_run_summary`, and `render_session_summary`. All four go through one helper (and so does the fifth we add in Chapter 26):

Listing 23.8: `render.lat:324–341`

```
fn _fields(title: String, fields: Array) -> String {
    flux width = 0
    for field in fields {
        if typeof(field) != "Array" || len(field) != 2 {
            _fail("render.fields", "expected [label, value] pairs")
        }
        let label = _require_string(field[0], "render.fields[].label")
        _require_string(field[1], "render.fields[].value", true)
        if len(label) > width {
            width = len(label)
        }
    }
    flux lines = [_plain(title)]
    for field in fields {
        lines.push("  " + _pad_right(_plain(field[0]), width) + ": " + field[1])
    }
    return lines.join("\n")
}
```

The input is an `Array` of two-element `Array`s used as tuples. Labels are padded to the widest label, which is why every `: show` block has its colons in a column. Lattice has no tuple type; a fixed-length array is the idiom.

The call sites are the best place in the Ballistics Lab to see `if` used as an *expression* inside an array literal:

Listing 23.9: render.lat:631–642

```
["stability factor", if summary.stability_factor == nil {
        "not reported"
    } else {
        _fixed(summary.stability_factor, profile.precision)
    }
],
["spin drift", if summary.spin_drift == nil {
        "not reported"
    } else {
        _distance_text(summary.spin_drift, offset_unit, profile, true)
    }
]
```

That is where `spin drift : not reported` in every `:solve` block comes from. `ballistics-engine` 0.32.0 does not populate `spin_drift_m` for the loads in this book, and rather than print a misleading zero the renderer says so.

23.4.1 Escaping, and why a renderer needs it

Notice strings, experiment names, and observation flags all come from outside the Ballistics Lab — the engine wrote them, or a user typed them. They land in a line-oriented, pipe-delimited display. So every string passes through `_plain`:

Listing 23.10: render.lat:107–114

```
fn _plain(value: String) -> String {
    return value
    .replace("\\", "\\\\")
    .replace("\r", "\\r")
    .replace("\n", "\\n")
    .replace("\t", "\\t")
    .replace("|", "\\|")
}
```

Order matters: backslash first, or the escapes you introduce get escaped again. The effect is visible in the render module's own snapshot dump, where a fixture notice deliberately contains a pipe and a newline:

Listing 23.11: From `./clat examples/ballistics_lab/test_render.lat dump` (verbatim)

```
  Assumptions
    - model_default: Used G7 \| model. [$.projectile.drag_model]
  Warnings
    - range_notice: Line one.\nLine two. [$.shot.max_range]
```

An engine that emitted a newline in a warning cannot break the table's row structure. The export module carries a byte-for-byte identical copy of this chain, named `_plain_text`, at its own lines 432–439 — deliberate duplication across a module boundary, because that module imports nothing.

23.5 From trajectory to numbers

Now the other half. `analysis.lat` never formats anything; it produces `Numbers` in requested units.

23.5.1 `trajectory_at`: the interpolator

A solved trajectory is a list of samples on the engine's grid. You almost never want a number *at* a grid point; you want it at 600 yd. `trajectory_at` handles three cases: an exact hit on a sample, a linear interpolation between two adjacent samples, or an out-of-range refusal.

Listing 23.12: The out-of-range guard, `analysis.lat:265–273`

```
if (query < first.distance.si_value && !_near(query, first.distance.si_value)) ||
(query > terminal.distance.si_value && !_near(query, terminal.distance.si_value)) {
    assert(
    false,
    "trajectory_at.distance: " + to_string(query) +
    " m is outside computed trajectory [" + to_string(first.distance.si_value) +
    ", " + to_string(terminal.distance.si_value) + "] m"
    )
}
```

The diagnostic names the actual interval, which is exactly what you need when a solve terminated early:

Listing 23.13: Querying past the end of a 914.4 m trajectory — one long line wrapped

```
out of range     : assertion failed: trajectory_at.distance: 2000 m is outside
                   computed trajectory [0, 914.4] m
```

Two subtleties. First, when the query lands exactly on a sample, the returned `Observation` carries the *caller's* distance value — so asking for 500 m gives you back a `Distance` labelled in metres, even though the underlying sample was recorded in whatever unit the engine used. Second, an interpolated observation has *empty flags*. Flags such as `transonic` and `terminal` belong to real samples; manufacturing one between two samples would be a lie.

23.5.2 `sample`: bounded grids that clip

`sample(trajectory, start, stop, interval)` walks a regular grid and returns an `Array` of observations. It does three careful things.

It **projects the row count before building anything**, so a pathological interval fails fast instead of allocating for ten seconds first. It **clips a requested stop to the real terminal sample**, so asking for 2000 m of a trajectory that ended at 914.4 m is not an error — you get the trajectory you actually have. And it **appends the true terminal exactly once**, even when the terminal does not fall on a grid point.

Listing 23.14: Sampling the reference load, engine 0.32.0 — verbatim, two messages wrapped

```
engine samples   : 101
first sample     : 0 m
terminal sample  : 914.4 m
summary range    : 914.4 m
sample 0..500/125: 0, 125, 250, 375, 500
sample 800..2000 : 800, 914.4
phase of sample  : crystal
bad interval     : assertion failed: sample.interval: must be greater than zero
grid too large   : assertion failed: sample: grid exceeds the 10000-observation
                   analysis limit
```

`sample 800..2000` returned two rows: the requested start, and the clipped terminal. That is the behaviour that makes `:dope 100 2000 100 yd` useful on a trajectory that hit the ground at 1500 yd.

The 10,000-row ceiling

`MAX_ANALYSIS_ROWS = 10000` appears in `analysis.lat`, `analysis_export.lat` (as `MAX_EXPORT_ROWS`), and `render.lat` (inline, at `_validate_table`). All three enforce it independently, because all three can be reached without the others. The same number bounds the engine's sample array in `process_backend.lat`. It is not a coincidence and it is not a constant shared by import — it is a stated protocol limit reasserted at every boundary.

23.5.3 The three table builders

`dope`, `wind_card`, and `compare_trajectories` all produce the same struct:

Listing 23.15: `analysis.lat:39–45`

```
export struct AnalysisTable {
    kind: fix String,
    title: fix String,
    columns: fix Array,
    rows: fix Array,
    provenance: fix Array
}
```

`columns` is an array of `TableColumn { key, label, unit }`. `rows` is an array of `Map`s keyed by those column keys. `provenance` is an array of `TrajectoryProvenance` — one per source trajectory, carrying engine version, schema version, solver method, verification status, assumptions, warnings, termination reason, and actual range.

That last field is the reason the BALLISTICS LAB's tables are longer than a spreadsheet's. Every table you print carries the complete story of where its numbers came from:

Listing 23.16: The provenance block under every table (verbatim, assumptions truncated)

```
Provenance
  [0] 140 gr ELD-M at 1,200 yd
    engine: 0.32.0 (schema 1)
    solver: rk45
    verified: no
    termination: max_range
    actual range: 1200.00 yd
    Assumptions
      - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
      - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
      - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
    Warnings
      none
```

Print that table, paste it into an email six months later, and the recipient can still tell which engine build produced it and what it silently assumed.

`compare_trajectories` is the most involved of the three. It computes the shared `[common_start, common_stop]` window across all supplied trajectories, builds the sorted union of every sample distance inside that window, de-duplicates near-identical distances, and then interpolates *each* series independently onto the merged grid. Two loads sampled on different intervals therefore compare cleanly without anybody re-solving. Deltas are always relative to series 0.

23.5.4 One freeze at the top

Columns, rows, notices, and provenance are built as ordinary mutable values and frozen exactly once, at the end, by the containing table. The source says so:

Listing 23.17: `analysis.lat:117–124 and 186–195`

```
fn _column(key: String, label: String, unit: any = nil) -> TableColumn {
    // Remain alloy until the containing AnalysisTable is frozen. Freezing each
    // child separately would create a graph of shared regions for a short-lived
    // calculation result.
    return TableColumn {
        key: key, label: label, unit: unit
    }
}

    // One freeze owns the complete calculation graph. Child columns, rows,
    // notices, and provenance deliberately remain alloy/fluid until here.
    return freeze(AnalysisTable {
        kind: kind, title: title, columns: columns,
        rows: rows, provenance: provenance
    })
```

Alloy versus crystal

A struct whose fields are declared `fix` is an *alloy*: the fields cannot be reassigned, but the value is not in a frozen region. A value that has been through `freeze()` is a *crystal*: deeply immutable, and on this runtime shared by reference rather than copied. `phase_of(table)` returns `"crystal"` for every `AnalysisTable` the Ballistics Lab hands you.

There is one more phase-aware line in the module, and it is the subtlest in the whole Ballistics Lab:

Listing 23.18: `analysis.lat:126–130`

```
fn _copy_string(value: String) -> String {
    // Materialize a source-independent string rather than retaining a handle
    // into a large solved-trajectory crystal region.
    return value.substring(0, len(value))
}
```

A trajectory with 10,000 samples is one large crystal region. If a table's provenance simply referenced the engine-version string *inside* that region, the whole trajectory would stay alive as long as the table did. `substring` forces a fresh allocation. Every string that crosses from a trajectory into a table goes through it.

23.6 The sign conventions

Three conventions govern every signed number the Ballistics Lab prints, and every table states the ones it uses in its own last line:

- **Drop is positive below.** A positive drop is distance below the line of sight; a negative drop is above it. At the muzzle the drop equals the sight height.
- **Windage is positive right.** A positive windage is impact to the right of the aim point, from the shooter's view.
- **Corrections are positive in the direction the sight moves.** Come-up is positive when the sight must be raised; wind hold is positive when it must go right.

Two derived columns exist purely to answer the shooter's actual question, which is never "how far did it drop" but "how much do I dial." They are the only places in the Ballistics Lab where a sign convention is converted rather than carried.

Listing 23.19: The two derived angles, `analysis.lat:457–460 and 549–555`

```
row.set(
"come_up",
_angle_value(atan2(item.drop.si_value, item.distance.si_value), correction_unit)
)

row.set(
"wind_hold",
_angle_value(
atan2(0.0 - item.windage.si_value, item.distance.si_value),
correction_unit
)
)
```

Read those two statements next to each other and the asymmetry is the whole lesson of this section: the come-up takes the angle subtended by the offset, and the wind hold takes the angle subtended by its *negation*. They differ by one minus sign because the two offsets they read are signed in opposite senses relative to the correction.

Start with the one that negates. `item.windage` is positive to the right, and a correction is positive to the right, so an impact that lands right needs a hold to the left — opposite signs, hence the negation. The Ballistics Lab's own tests pin it, and the wind card in a live session bears it out. A 270° wind (from the west, shooting north) pushes the bullet right, and the hold is left:

Listing 23.20: `:wind-card 200 1000 200 yd` (verbatim, provenance elided)

```
Wind card
Series | Trajectory               | Wind (mph) | From (deg) | Distance (yd) | Windage (in) |
     Wind hold (mil)
-------+--------------------------+------------+------------+---------------+--------------+-----------------
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        200.00 |        +1.64 |
          -0.23
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        400.00 |        +6.88 |
          -0.48
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        600.00 |       +16.27 |
          -0.75
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |        800.00 |       +30.49 |
          -1.06
  0.00 | 140 gr ELD-M at 1,200 yd |       8.00 |     270.00 |       1000.00 |       +50.43 |
          -1.40
Signs: windage + right/- left; wind hold + correction right/- correction left
```

23.6.1 Why one of them negates and the other must not

Now the elevation column. `item.drop` is built on the line immediately above `item.windage`, inside the same constructor call, and both are just as untouched by the decoder:

Listing 23.21: `process_backend.lat:403–404` — both offsets pass through unchanged

```
m(object.get("drop_m")),
m(object.get("windage_m")),
```

Identical treatment, opposite conventions. `solve-json`'s `windage_m` is positive to the right; its `drop_m` is positive *downward*, distance below the line of sight. And "down" is the direction the correction points: a bullet that is low needs the sight raised. So for elevation the offset and the correction already agree in sign, and the come-up is the angle subtended by the drop *itself*. Negating it would name the direction the impact sits, which is exactly backwards from the dial.

That is why `analysis.lat:430–434` says what it says, in the source, directly above `dope`:

Listing 23.22: `analysis.lat:430–434`

```
// Build a conventional come-up table. Positive come_up means the sight must be
// raised. The engine reports drop as positive-DOWNWARD (metres below the line
// of sight: at the muzzle it equals the sight height), so the come-up is the
// angle subtended by the drop itself, not its negation. Windage is the other
// convention -- positive-RIGHT -- which is why the wind hold below negates it.
```

23.6.2 Reading the convention off a real table

Two rows prove the drop convention without any argument, and both are free — you get them from any solved trajectory.

The load for the rest of this section is not the ELD-M of the earlier transcripts. It is a deliberately plain one — 140 gr 6.5 mm, BC 0.326 G7, 2710 fps, a 1.75-inch sight, a 100-yard zero, 4,000 ft and 50°F, with no bullet length supplied and Coriolis off — chosen because the engine's own `come-ups` subcommand can solve exactly that problem from command-line flags, which makes the cross-check at the end of this section an honest one.

The muzzle row. Query the trajectory at zero distance and the drop is the sight height, because that is how far below the sight line the bore sits before the bullet has gone anywhere:

Listing 23.23: `:at 0 yd` (verbatim)

```
Observation
  distance: 0.00 yd
  time    : 0.00 s
  velocity: 2710.00 ft/s
  energy  : 2282.62 ft-lb
  drop    : +1.75 in
  windage : +0.00 in
  Mach    : 2.45
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

The zero row. At the zero distance the bullet is on the line of sight and the dial is nothing. Here is the same load carried out to 1,000 yards:

Listing 23.24: `:dope 100 1000 100 yd` (verbatim, provenance elided)

```
DOPE: 140 gr 6.5 mm reference load
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2587.03 |        2080.17 |     0.11 | 2.34
       200.00 |     +3.44 |         +0.48 |         2467.13 |        1891.81 |     0.23 | 2.23
       300.00 |    +12.61 |         +1.17 |         2350.38 |        1717.00 |     0.36 | 2.12
       400.00 |    +28.07 |         +1.95 |         2236.87 |        1555.16 |     0.49 | 2.02
       500.00 |    +50.47 |         +2.80 |         2126.71 |        1405.77 |     0.63 | 1.92
       600.00 |    +80.57 |         +3.73 |         2019.81 |        1267.98 |     0.77 | 1.82
       700.00 |   +119.19 |         +4.73 |         1915.86 |        1140.83 |     0.92 | 1.73
       800.00 |   +167.30 |         +5.81 |         1814.57 |        1023.39 |     1.08 | 1.64
       900.00 |   +225.98 |         +6.97 |         1715.81 |         915.02 |     1.25 | 1.55
      1000.00 |   +296.48 |         +8.24 |         1619.42 |         815.10 |     1.43 | 1.46
Signs: drop + below/- above; come-up + correction up/- correction down
```

Now the cross-check. The `come-ups` subcommand takes flags rather than v1 JSON, so it reaches the same solver by a completely different path through the same binary — a second opinion from inside the same program:

Listing 23.25: The engine's own come-up table, run directly (verbatim except the box-drawing characters and one em dash, transliterated to ASCII)

```
$ ballistics come-ups --velocity 2710 --bc 0.326 --mass 140 --diameter 0.264 \
    --drag-model g7 --zero-distance 100 --sight-height 1.75 --altitude 4000 \
    --temperature 50 --humidity 35 --start 100 --end 1000 --step 100 \
    --adjustment-unit mil
Come-Up Table (zero: 100 yd, MIL)
+----------+----------+----------+----------+----------+----------+
|Range (yd)|Drop (MIL)|Come-Up   | Vel (fps)|Energy    | Time (s) |
+----------+----------+----------+----------+----------+----------+
|      100 |    0.000 |    -     |     2587 |     2080 |    0.113 |
|      200 |    0.478 |    0.478 |     2467 |     1892 |    0.232 |
|      300 |    1.168 |    0.689 |     2350 |     1717 |    0.357 |
|      400 |    1.949 |    0.781 |     2237 |     1555 |    0.487 |
|      500 |    2.804 |    0.855 |     2127 |     1406 |    0.625 |
|      600 |    3.730 |    0.926 |     2020 |     1268 |    0.770 |
|      700 |    4.730 |    1.000 |     1916 |     1141 |    0.922 |
|      800 |    5.809 |    1.079 |     1815 |     1023 |    1.083 |
|      900 |    6.975 |    1.166 |     1716 |      915 |    1.253 |
|     1000 |    8.235 |    1.261 |     1619 |      815 |    1.433 |
+----------+----------+----------+----------+----------+----------+
```

Which engine column to compare against

The engine's *total* dial is the column headed `Drop (MIL)`. Its column headed `Come-Up` is the *incremental* step from the previous row — $0.478 + 0.689 = 1.167$, and the next row reads 1.168. The BALLISTICS LAB's `Come-up (mil)` column is the total, so it lines up against `Drop (MIL)`, and the engine's `Come-Up` column has no counterpart in a BALLISTICS LAB dope table at all.

Set the two side by side and they agree at all ten distances, to the two decimals the BALLISTICS LAB prints:

Range (yd)	Lab come-up (mil)	Engine dial (mil)	Lab drop (in)
100	+0.00	0.000	+0.00
200	+0.48	0.478	+3.44
300	+1.17	1.168	+12.61
400	+1.95	1.949	+28.07
500	+2.80	2.804	+50.47
600	+3.73	3.730	+80.57
700	+4.73	4.730	+119.19
800	+5.81	5.809	+167.30
900	+6.97	6.975	+225.98
1000	+8.24	8.235	+296.48

The residual is the BALLISTICS LAB interpolating onto its 10-yard sampling grid where the engine solves at the exact range.

23.6.3 The defect this column used to carry

None of that was true while this book was being drafted, and the story is worth telling because it is the best argument in this book for the discipline the next section describes.

Until LATTICE commit `cd92f9e`, the come-up column told the shooter to dial the wrong way, by the wrong amount. Two independent faults were stacked in it.

The sign. `analysis.lat` computed `atan2(0.0 - item.drop.si_value, item.distance.si_value)` — the negation you see two pages up in the *wind* hold, copied onto a quantity signed the other way. Every come-up came out negative under a legend that reads `+ correction up/- correction down`. At 400 yards the card said -2.44 mil for a shot that needs 1.95 up.

The datum. The BALLISTICS LAB never sent `target_height_m`. The engine documents that field as the height above ground the zero is solved to and defaults it to `0.0` when it is absent, so every solve was zeroed so the bullet struck the *ground* at 100 yards rather than the line of sight. That biased every drop by the height of the sight line and every come-up by $h_{\text{sight}}/R_{\text{zero}}$ — a constant angle, identical

at 100 yards and at 1,000, which is what eventually pointed at the datum rather than at the solver. For a 1.75-inch sight over a 100-yard zero it is $0.04445/91.44 = 4.861 \times 10^{-4}$ rad, printing as 0.49 mil.

Both are fixed. The sign correction is the missing minus you have already read; the datum correction is one `else` branch in `protocol_v1.lat:74–92` that fills in `target_height_m` as muzzle height plus sight height whenever the domain `Shot` leaves it unset — which is what all six conformance fixtures had been doing by hand all along. The engine needed no change; the Ballistics Lab had been under-specifying the request.

	400 yd		600 yd	
	Come-up	**Drop**	**Come-up**	**Drop**
Before `cd92f9e`	−2.44 mil	+35.07 in	−4.22 mil	+91.08 in
After	+1.95 mil	+28.07 in	+3.73 mil	+80.57 in
Engine `come-ups`	1.949 mil	—	3.730 mil	—

If you are holding an older card

Numbers printed by a Ballistics Lab older than `cd92f9e` are wrong in both columns and cannot be repaired by flipping a sign — the trajectory itself was solved to the wrong datum. Re-solve. A come-up column you can trust reads `+0.00` at the zero distance and climbs from there, and the table's last line says `Signs: drop + below/- above`. If a card in your records says `drop + above/- below`, it came from the old build.

23.7 Rough edges you must know about

The Ballistics Lab is careful code, and there are still places where the output reads oddly. None of them is a wrong number any more, and this book is not going to inflate them into one.

Read the legend, not your memory

Every table and every observation block prints its own sign legend as its last line, and there are three distinct ones:

- `Signs: drop + below/- above; windage + right/- left` — `:at`.
- `Signs: drop + below/- above; come-up + correction up/- correction down` — `:dope`.
- `Signs: windage + right/- left; wind hold + correction right/- correction left` — `:wind-card`.

`:compare` prints the first of those with a third clause appended, `deltas are series minus baseline [0]`. Trust the legend on the card in front of you rather than a convention you carried in from somewhere else: drop positive-below is what `ballistics-engine` reports and what the BALLISTICS LAB prints, and nothing obliges another tool's card to agree.

The one genuine wart left is cosmetic. `series_index` is stored as an `Int` but rendered through `_fixed`, so the `Series` column of a wind card or comparison shows `0.00` and `1.00` rather than `0` and `1`. Nothing downstream depends on it; `analysis_export.lat` writes the integer.

The other one you have already met, in §23.2: altitude is printed in the profile's distance unit, so a 4,000-foot field elevation reads `altitude : 1333.33 yd` under the default imperial profile. Arithmetically correct, and nobody quotes elevation in yards.

23.7.1 Why this book checks every number

The come-up column is the reason this book runs everything it prints.

Consider what the defect survived. The column had a documented intent — the comment above `dope` said that positive `come_up` means the sight must be raised — and a printed legend that agreed with the comment. The renderer asserts at `render.lat:946` that the table carries exactly the seven expected keys in exactly the expected units, and that assertion passed. Every neighbouring column was right. The trajectory underneath was right to the last digit the engine printed, for the request it was actually sent. Even the expression was a correct formula — it was simply the wind hold's formula, applied to a quantity signed the other way.

And it was tested. `test_analysis.lat` carries five test functions, one of them dedicated to `dope`, and it reaches the come-up column directly. Today that assertion reads:

Listing 23.26: `test_analysis.lat:196–198`

```
// drop is positive-downward (distance below the line of sight), so a
// positive drop means the sight must be raised: come_up must be positive.
assert(table.rows[0].get("come_up") > 0, "positive fixture drop requires a positive
    come-up")
```

Before `cd92f9e` the same line read `assert(table.rows[0].get("come_up") < 0)`, with the failure message "positive fixture drop requires a *negative* come-up." That is not an oversight. It is the inversion written down as the specification — and it passed, every run, on both virtual machines. The very next test function, `test_wind_card_sign_convention`, pins the *wind* hold the other way (`test_analysis.lat:221`, `"left drift must require a positive/right hold"`), and that one was correct all along. A suite records the author's belief about each column, which is exactly what a suite is for and exactly why it cannot catch this class of error on its own. Section 25.5 follows that thread further: the fixtures feeding the render, command and artifact snapshots supplied drop values that grew more *negative* with distance, so every layer of the suite agreed with every other layer.

Nothing internal to the Ballistics Lab could have caught it. Reading the code is how the error was written; testing the code is how the belief was preserved. It was caught by one thing only: taking the printed number to a second implementation of the same physics and asking whether they agreed. They did not — the Ballistics Lab said −8.72 mil at 1,000 yards where the engine's own `come-ups` subcommand said 8.235 up. And the residual, once the sign was restored, was constant in *angle* across ten ranges rather than growing with distance, which is what pointed at the zero datum rather than at the solver and turned one defect into two.

A book of ballistics tables that has not done that is a book of assertions. The cost of the discipline is that every transcript here had to come from a real run, and that a fix in the Ballistics Lab means recapturing them. The return is that the numbers on these pages are the numbers the tool prints.

A note for anyone reading `render.lat` in an editor

From line 678 to the end of the file the indentation drifts progressively rightward: `render_dope` sits at four spaces, `render_comparison` at sixteen, and `render_error` deeper still. It is cosmetic — Lattice is not indentation-sensitive — but it makes the tail of the file harder to read than the head. Listings in this chapter have been re-indented; the behaviour is untouched.

23.8 Building the table

`_render_table` is 56 lines and does the whole job: measure, format, pad, join. The algorithm is two passes.

Pass one walks `table.columns`, computes each heading (`"Drop (in)"`), seeds the column width from the heading length, and asks `_column_numeric` whether the column holds numbers.

Pass two walks the rows, formats each cell with `_scalar_text`, and widens the column if the formatted cell is longer than anything seen so far. Only then are the header, the rule, and the rows padded and joined.

Listing 23.27: The assembly, `render.lat:916–941` (re-indented)

```
flux header_fields = []
flux rule_fields = []
flux index = 0
while index < len(headings) {
    header_fields.push(_pad_right(headings[index], widths[index]))
    rule_fields.push("-".repeat(widths[index]))
    index = index + 1
}
flux lines = [_plain(table.title), header_fields.join(" | "), rule_fields.join("-+-")]
for row in rendered_rows {
    flux output = []
    index = 0
    while index < len(row) {
        if numeric[index] {
            output.push(_pad_left(row[index], widths[index]))
        } else {
            output.push(_pad_right(row[index], widths[index]))
        }
        index = index + 1
    }
    lines.push(output.join(" | "))
}
lines.push(sign_legend)
lines.push(_render_provenance(table, profile))
return lines.join("\n")
```

Three details worth naming. The separator between cells is `" | "` but the rule joins with `"-+-"`, which is what makes the `+` land under the `|`. Numeric columns are right-aligned and text columns left-aligned — look back at the wind card and you will see `140 gr ELD-M at 1,200 yd` flush left while every number is flush right. And the module builds an `Array` of lines and joins once, rather than concatenating strings in a loop; the same discipline appears in `command_parser.lat`, where the source comment spells out why (repeated immutable concatenation would make a valid 4,096-byte token quadratic).

23.8.1 A column may not mix numbers and text

`_column_numeric` scans the whole column and refuses ambiguity:

Listing 23.28: render.lat:850–865 (re-indented)

```
fn _column_numeric(table: any, column: any) -> Bool {
    flux numeric = nil
    for row in table.rows {
        let value = row.get(column.key)
        if value != nil {
            let kind = typeof(value)
            let current = kind == "Int" || kind == "Float"
            if numeric == nil {
                numeric = current
            } else if numeric != current {
                _fail("render.table." + column.key, "column mixes numeric and text
    values")
            }
        }
    }
    return numeric == true
}
```

Hand-build a table whose `mach` column holds a number in one row and a string in the next, and the renderer stops:

Listing 23.29: Mixed column: the message from the thrown error

```
=== 4b mixed numeric/text column ===
assertion failed: render.table.mach: column mixes numeric and text values
```

23.9 The coupling check

Here is the sharpest idea in `render.lat`. The command layer asks the renderer which units to use, hands them to `analysis.lat`, gets a table back, and hands that table to the renderer. If those two conversations ever disagreed, you would get a table labelled `mil` containing MOA. So the renderer asserts the contract:

Listing 23.30: `render.lat:943–956` (re-indented)

```
export fn render_dope(value: any, display: any) -> String {
    let table = _validate_table(value, "dope")
    let profile = _profile(display)
    _expect_columns(table,
    ["distance", "drop", "come_up", "velocity", "energy", "time_s", "mach"],
    [profile.distance_unit, offset_unit_for(profile), profile.angle_unit,
        profile.velocity_unit, profile.energy_unit, "s", nil]
    )
    return _render_table(
    table,
    profile,
    "Signs: drop + below/- above; come-up + correction up/- correction down"
    )
}
```

The renderer checks the column *keys* and the column *units*, in order, against what the profile implies. Build a DOPE table in MOA and try to render it under a mil profile and you get a precise diagnostic rather than a mislabelled card:

Listing 23.31: Unit and kind mismatches: the messages from the thrown errors

```
=== 2 unit mismatch ===
assertion failed: render.table.columns[2].unit: expected 'mil', got 'moa'
=== 3 kind mismatch ===
assertion failed: render.table.kind: expected 'wind_card', got 'dope'
```

This is a hard coupling between two modules, and it is deliberate. The alternative — the renderer converting units itself — would mean the same conversion happening twice with two chances to disagree.

23.10 Never iterate a Map

Rows are `Map`s, and LATTICE's `Map` does not promise a stable iteration order. `render.lat` therefore never iterates one. It walks `table.columns` — an `Array`, and therefore ordered — and calls `row.get(column.key)`.

`test_render.lat` proves it. The test takes a real DOPE table, rebuilds every row *inserting the keys in reverse order*, renders it, and asserts the output is byte-identical to the original snapshot.

Why the paranoia is justified: here is what a `Map` actually gives you back when you ask for its keys. This is the top level of a table converted by `analysis_export.table_to_map`, whose insertion order was `kind, title, columns, rows, provenance`:

Listing 23.32: Insertion order is not iteration order (verbatim)

```
=== table_to_map keys ===
[provenance, kind, rows, title, columns]
```

Which is exactly why `analysis_export.lat` does not serialize `Map`s either.

23.11 Exporting: JSON, CSV, and text

`table_to_json` produces JSON *without ever serializing a Map*. It concatenates strings, calling `json_stringify` only on scalars:

Listing 23.33: analysis_export.lat:290–315

```
// Produce canonical JSON without serializing a Map, so object member order is
// independent of the runtime's Map implementation.
export fn table_to_json(table: any) -> String {
    let checked = _validate_table(table)

    flux columns = []
    for column in checked.columns {
        columns.push(
        "{" +
        "\"key\":" + json_stringify(column.key) + "," +
        "\"label\":" + json_stringify(column.label) + "," +
        "\"unit\":" + json_stringify(column.unit) +
        "}"
        )
    }

    flux rows = []
    for row in checked.rows {
        flux fields = []
        for column in checked.columns {
            fields.push(
            json_stringify(column.key) + ":" + json_stringify(row.get(column.key))
            )
        }
        rows.push("{" + fields.join(",") + "}")
    }
    ...
```

The output is a single line, in the declared column order, with full float precision:

Listing 23.34: table_to_json output — one line, wrapped and elided here

```
{"kind":"dope","title":"DOPE: 175 gr match at 1,000 yards","columns":[{"key":
"distance","label":"Distance","unit":"yd"},{"key":"drop","label":"Drop","unit":
"in"},{"key":"come_up","label":"Come-up","unit":"mil"},...],"rows":[{"distance":
100,"drop":2.5159832198556669e-05,"come_up":6.9888422773768524e-06,"velocity":
2513.3645411952612,"energy":2454.2300395683042,"time_s":0.11517142182612057,
"mach":2.2482122507182454},{"distance":200,"drop":4.0081278979629857,"come_up":
0.55668437276759963,...
```

Two formatters, two audiences

`render.lat` prints `2513.36`; `analysis_export.lat` writes `2513.3645411952612`. That is not an inconsistency, it is the point. The terminal renderer formats for a human at a bench and rounds to the session precision. The exporter formats for a machine and uses `json_stringify` specifically because it is round-trip-safe — the generic `to_string` uses a shorter display precision and would lose bits.

23.11.1 CSV that carries its own provenance

The CSV export appends *fourteen fixed provenance columns to every row*:

Listing 23.35: The CSV header — one line in the file, wrapped here

```
Distance (yd),Drop (in),Come-up (mil),Velocity (ft/s),Energy (ft-lb),Time (s),Mach,
Table kind,Table title,Provenance index,Series name,Schema version,Engine version,
Solver method,Verified,Termination,Actual range (m),Actual range value,
Actual range unit,Assumptions,Warnings
```

That is redundant by any normalization standard and completely right for the purpose. An analyst loads the file with an ordinary `pd.read_csv` and still has the engine version, the solver, and the full assumption list without a sidecar file, a comment header, or a custom parser. The assumptions and warnings columns hold JSON arrays, RFC-4180 quoted by hand in `_csv_field`.

For a multi-series table, `_row_provenance_index` reads the row's `series_index` to select the right provenance entry, so a comparison CSV attributes each row to its own run.

23.11.2 `render_table`: the export module's own text form

`analysis_export.lat` also ships a plain-text renderer — deliberately *not* the pretty one. No padding, no alignment, no rounding:

Listing 23.36: `analysis_export.render_table` — head, rows truncated at the right, later rows elided

```
title: DOPE: 175 gr match at 1,000 yards
kind: dope
rows:
  Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
  100 | 2.5159832198556669e-05 | 6.9888422773768524e-06 | 2513.3645411952612 | ...
provenance:
  [0] 175 gr match at 1,000 yards
    schema_version: 1
    engine_version: 0.32.0
    solver_method: rk45
    verified: false
    termination: max_range
    actual_range: 914.39999999999998 m (914.39999999999998 m SI)
```

It exists so that a diff-based test or a log can capture a table losslessly. Note `actual_range` printing both the display value and the SI value: at this level of the stack nothing is hidden.

23.12 What this chapter bought

Two thousand lines to print a table looks extravagant until you list what those lines guarantee:

- Numbers are formatted by one function whose output cannot vary with a runtime's `printf`.
- A table cannot be rendered under a profile it was not calculated for.
- Engine-supplied strings cannot break the table structure.
- `Map` ordering cannot affect any output, in any of four formats.
- Every table carries the engine version, solver, and assumption list that produced it.
- Two ten-thousand-row limits and one out-of-range assertion stand between a typo and an unbounded allocation.

And the payoff shows up in a place you would not expect. Feed one input file to the stack VM and to the register VM and the rendered output is *byte*-identical, not merely close — same SHA-256, both for the two dope tables at the top of this chapter (ten come-up values through `atan2`, hashing to `bb03d564...` on both backends) and for a 140-line `:compare`. The float-heavy integration happens in the Rust engine, so that does not prove the two VMs agree on ODE arithmetic; it proves they agree on request construction, unit conversion, interpolation, table assembly, and every line of the formatting code in this chapter. Which is the part written in Lattice.

Chapter 24 takes a table's provenance one step further, and asks what it would take to make a run into a document you could hand to somebody else.

Chapter 24

Verified Artifacts

Every table in the previous chapter said the same thing about itself:

Listing 24.1: The line that motivates this chapter

```
verified: no
```

Not *failed*. Not *suspect*. Just: nobody has promoted this run into a document yet. That promotion is what `verified_artifacts.lat` is for, and it is the one place in the Ballistics Lab where a working session value becomes something you can hand to another person, another machine, or your own future self.

This chapter answers three questions. What exactly is an artifact? Why does the Ballistics Lab bother having one, when it already has a perfectly good `:save`? And what does the 724-line module actually check?

24.1 Working data versus a document

Recall what you hold after a `:solve`. `session.lat` built a `RunRecord`:

Listing 24.2: `session.lat:19–33`

```
// RunRecord is an immutable alloy wrapper. Its fields are `fix`, and every composite
// payload placed in it is already frozen by the domain/backend boundary. Avoid
    freezing
// the outer wrapper again: a trajectory may contain 10,000 samples, and alloy fields
    give
// us immutability without copying that already-frozen graph into another crystal
    region.
export struct RunRecord {
    backend_name: fix String,
    experiment: fix any,
    result: fix any,
    resolved_request: fix Map,
    schema_version: fix Int,
    engine_version: fix String,
    assumptions: fix Array,
    warnings: fix Array,
    termination: fix String
}
```

That is excellent working data and a poor document. Three reasons.

It is enormous. `result` is a whole `Trajectory`. For the book's reference load that is 101 sampled observations; the protocol permits 10,000. Nobody wants a 10,000-row JSON file to record that they checked a come-up at 600 yards.

It is redundant, and the redundancy is unchecked. The record carries `engine_version`, `schema_version`, `assumptions`, `warnings`, `termination`, and `resolved_request` — and so does the `Trajectory` inside it. Nothing in `session.lat` forces those copies to agree once the record exists. A caller who hand-built a `RunRecord` could make them disagree.

It is entangled. On this runtime a frozen value is shared by reference. The record's strings and sub-structures live inside the trajectory's crystal region. Serialize the record naively and you have not produced an independent document; you have produced a view onto a live object graph.

Artifact

In the Ballistics Lab, an *artifact* is a `VerifiedRun`: a frozen, self-consistent, fully detached record of one completed solve, containing the experiment, the exact request the engine resolved, the summary, the notices, and a small explicit selection of observations — and containing no closure, no `Ref`, no process handle, no trajectory, and no session state.

24.2 The struct and the document contract

Listing 24.3: verified_artifacts.lat:12–30

```
let VERIFIED_RUN_KIND = "lattice-ballistics/verified-run"
let VERIFIED_RUN_SCHEMA_VERSION = 1
let SOLVE_SCHEMA_VERSION = 1

export struct VerifiedRun {
    kind: fix String,
    artifact_schema_version: fix Int,
    backend_name: fix String,
    solve_schema_version: fix Int,
    engine_version: fix String,
    experiment: fix any,
    resolved_request: fix Map,
    assumptions: fix Array,
    warnings: fix Array,
    summary: fix any,
    termination: fix String,
    selected_observations: fix Array,
    note: fix any
}
```

Thirteen fields, and the JSON document has exactly those thirteen keys — no more, no fewer. Note the *two* schema versions. `artifact_schema_version` describes this file format; `solve_schema_version` describes the wire protocol that produced the embedded `resolved_request`. They can evolve independently, and the loader checks both.

Two document kinds, two purposes

`:save` writes a `"lattice-ballistics/experiment"` document: just the experiment, about 1.3 KB, meant to be *reloaded and re-solved*. An artifact is a `"lattice-ballistics/verified-run"` document: the experiment *plus* what actually happened, meant to be *read*, not re-run. The experiment inside an artifact is written by the very same `persistence.lat` adapters, so the two formats never drift.

24.3 Making one

There is exactly one way in, and it takes a `RunRecord`:

Listing 24.4: verified_artifacts.lat:292–298

```
// Promote a completed session RunRecord into a detached, immutable artifact.
// A Trajectory is never accepted directly, regardless of its legacy `verified` flag.
export fn verify_run(
record: any,
observation_indices: any = nil,
note: any = nil
) -> VerifiedRun {
```

Here it is against the real engine. The session solves the reference 175 gr load, and the record is promoted with a note:

Listing 24.5: A live promotion, engine 0.32.0 (verbatim)

```
record type      : RunRecord
observations     : 101
record phase     : unphased
artifact type    : VerifiedRun
artifact phase   : crystal
kind             : lattice-ballistics/verified-run
artifact schema  : 1
solve schema     : 1
engine           : 0.32.0
backend          : process-json-v1
termination      : max_range
selected samples : 3
note             : range card check, 2026-08-05
selected ranges  : 0, 457.2, 914.4 m
document bytes   : 4947
```

One hundred and one observations went in; three came out. The document is 4,947 bytes. And notice the phase change: the `RunRecord` is `unphased` (an alloy — immutable fields, but not a frozen region), while the `VerifiedRun` is a `crystal`.

24.3.1 The cross-checks

Before it copies anything, `verify_run` makes the record prove it is internally consistent. Seven comparisons, each between a `RunRecord` field and the corresponding field of the `Trajectory` it wraps:

Listing 24.6: verified_artifacts.lat:310–333

```
_require_struct(record.result, "Trajectory", "verify_run.record.result")
let result = record.result
if result.experiment != record.experiment {
    _fail("verify_run.record.result.experiment", "does not match the RunRecord
    snapshot")
}
if result.resolved_request != record.resolved_request {
    _fail("verify_run.record.result.resolved_request", "does not match the RunRecord
    snapshot")
}
if result.schema_version != solve_schema {
    _fail("verify_run.record.result.schema_version", "does not match the RunRecord")
}
if result.engine_version != engine_version {
    _fail("verify_run.record.result.engine_version", "does not match the RunRecord")
}
if result.assumptions != record.assumptions {
    _fail("verify_run.record.result.assumptions", "does not match the RunRecord")
}
if result.warnings != record.warnings {
    _fail("verify_run.record.result.warnings", "does not match the RunRecord")
}
_require_struct(result.summary, "TrajectorySummary",
    "verify_run.record.result.summary")
if result.summary.termination != record.termination {
    _fail("verify_run.record.termination", "does not match the trajectory summary")
}
```

Hand-build a `RunRecord` with a plausible-looking but wrong engine version and it stops there:

Listing 24.7: A forged record, refused — the thrown error's message, wrapped

```
--- verify a hand-built RunRecord with a swapped engine version ---
assertion failed: verified artifact verify_run.record.result.engine_version:
does not match the RunRecord
```

And handing it the `Trajectory` directly — the shortcut a reader will try first — is refused by name:

Listing 24.8: A Trajectory is not a record — the thrown error's message, wrapped

```
--- verify a Trajectory instead of a RunRecord ---
assertion failed: verified artifact verify_run.record: expected RunRecord,
got Trajectory
```

These are value comparisons, not identity comparisons

`result.experiment != record.experiment` compares *structurally*. A record whose `experiment` field is a freshly constructed but equal `Experiment` passes the check — verified live. So the cross-checks prove the record is *self-consistent*, not that its parts came from one allocation. That is the correct guarantee for a document, and it is a weaker statement than "these are the same object."

24.3.2 Which observations survive

An artifact keeps a handful of samples, not the whole trajectory. If you do not say which, the module picks:

Listing 24.9: `verified_artifacts.lat:220–232`

```
fn _default_indices(count: Int) -> Array {
    if count <= 0 {
        _fail("verify_run.record.result.observations", "must not be empty")
    }
    if count == 1 {
        return [0]
    }
    if count == 2 {
        return [0, 1]
    }
    let middle = to_int(floor((count - 1) / 2.0))
    return [0, middle, count - 1]
}
```

First, middle, last. For the 101-sample reference trajectory that is indices 0, 50, and 100 — which is why the live run above reported `selected ranges : 0, 457.2, 914.4 m`.

Supply your own and they are checked for type, range, and duplication:

Listing 24.10: The index checker, condensed from :234–259

```
let selected = _integer(index, path)
if selected < 0 || selected >= count {
    _fail(path, "index " + to_string(selected) + " is outside the observation range")
}
if checked.contains(selected) {
    _fail(path, "duplicate index " + to_string(selected))
}
```

Choose your own samples when the run has a story

The default triple is a summary, not evidence. If the run's point is where the bullet went transonic, or the last range at which it carried 1,000 ft-lb, pass those indices explicitly: `artifacts.verify_run(record, [0, 37, 88, 100], "carried 1000 ft-lb to 700 yd")`. The note is stored verbatim in the document and must be a non-empty string if supplied.

24.4 Detachment

This is the part of the module a LATTICE programmer should read twice. Making the artifact *independent* of the session takes more work than making it correct.

24.4.1 Copying by revalidating

There is no `deep_copy()` call anywhere in the module. Instead:

Listing 24.11: verified_artifacts.lat:128–142

```
fn _copy_distance(value: any) -> any {
    return persistence.distance_from_map(persistence.distance_to_map(value))
}

fn _copy_velocity(value: any) -> any {
    return persistence.velocity_from_map(persistence.velocity_to_map(value))
}

fn _copy_energy(value: any) -> any {
    return persistence.energy_from_map(persistence.energy_to_map(value))
}

fn _copy_experiment(value: any) -> any {
    return
    persistence.experiment_value_from_map(persistence.experiment_value_to_map(value))
}
```

Every value is round-tripped through the persistence adapters. That gets you two things at once: a genuinely new value with no region shared with the source, and a full revalidation through the domain constructors. A quantity that somehow carried invalid state into the record cannot survive the trip, because `persistence.lat` reconstructs it from `{value, unit}` and asserts the reconstructed SI value matches.

24.4.2 The JSON round trip as a region detacher

The subtlest three lines in the Ballistics Lab close `verified_run_to_map`:

Listing 24.12: verified_artifacts.lat:629–632

```
    // Strings copied out of a shared crystal can retain that region even inside an
    // otherwise fluid Map. A JSON data-model round trip detaches every scalar too,
    // ensuring callers receive no hidden reference to the VerifiedRun crystal.
    return json_parse(json_stringify(out))
```

Building a fresh `Map` is not enough. The *strings you put into it* may still be handles into the artifact's crystal region. Serializing to JSON and parsing back produces scalars that are unambiguously new. The same reasoning drives the manual flag-array rebuild a hundred lines earlier:

Listing 24.13: verified_artifacts.lat:530–537

```
    // Build an ordinary detached Array. clone() of a frozen sub-array may retain
    // the artifact's shared crystal region, which is not appropriate for a
    // programmable serialization Map.
    flux flags = []
    for flag in value.flags {
        flags.push(flag)
    }
    out.set("flags", flags)
```

It works. Take a live artifact, convert it to a `Map`, mutate the Map, and the artifact is untouched:

Listing 24.14: Detachment, demonstrated (verbatim)

```
--- to_map is detached from the crystal ---
map phase        : unphased
artifact still   : 0.32.0
```

(The `Map` had its `engine_version` set to `"tampered"` between those two lines.)

24.4.3 `phase_of` as an authenticity check

Before any artifact is serialized, it must prove it is a crystal:

Listing 24.15: verified_artifacts.lat:372–377

```
fn _reconstruct_verified(value: any) -> VerifiedRun {
    _require_struct(value, "VerifiedRun", "value")
    if phase_of(value) != "crystal" {
        _fail("value", "VerifiedRun must be a crystal created by verify_run or a
    loader")
    }
```

This is the only use of `phase_of` in the Ballistics Lab's library modules, and it is doing real work. A struct literal typed by hand has the right `struct_name` and the wrong phase. Struct names are not a trust boundary — `persistence.lat` says so in a comment, and this line is the same idea enforced with the phase system. It is not cryptography; a determined caller can `freeze()` a hand-built struct. It

is a guardrail against the ordinary accident of building a value the long way round and skipping the checks.

24.5 The document

Six functions carry an artifact in and out of bytes, in three pairs:

- `verified_run_to_map` and `verified_run_from_map`
- `verified_run_to_json` and `verified_run_from_json`
- `save_verified_run` and `load_verified_run`

Listing 24.16: `verified_artifacts.lat:712–724`

```
export fn save_verified_run(path: any, value: any) {
    let checked_path = _path(path, "save_verified_run")
    // Validate and serialize completely before mutating the filesystem.
    let source = verified_run_to_json(value)
    if !write_file(checked_path, source) {
        _fail("save_verified_run", "unable to write '" + checked_path + "'")
    }
}

export fn load_verified_run(path: any) -> VerifiedRun {
    let checked_path = _path(path, "load_verified_run")
    return verified_run_from_json(read_file(checked_path))
}
```

The comment is the rule the whole Ballistics Lab follows: complete every check and build the entire output *before* touching the filesystem. A validation failure leaves no half-written file behind.

The round trip is exact:

Listing 24.17: Save, load, re-serialize (verbatim)

```
document bytes   : 4947
reloaded phase   : crystal
round trip equal : true
top-level keys   : 13
```

The artifact JSON is deterministic, but it is not canonical

`verified_run_to_json` calls `json_stringify` on a `Map`, so member order follows the runtime's `Map` implementation — the file starts with `engine_version`, not `kind`. That is a real difference from `analysis_export.lat`, which hand-assembles canonical JSON precisely to avoid this. It matters less here because the loader is key-driven and order blind. And the bytes are stable where it counts: writing the same artifact under the stack VM and the register VM produced files with an identical SHA-256, `9bbda938d42065ef...`, on both backends.

24.6 Reading a hostile document

A saved artifact is a file on disk. Files get edited, truncated, generated by other programs, and mailed around. `verified_run_from_map` therefore treats its input as untrusted.

24.6.1 Closed schema

`_require_exact_keys` rejects in *both* directions — a missing field and an unknown field are equally fatal:

Listing 24.18: `verified_artifacts.lat:104–118`

```
fn _require_exact_keys(object: Map, expected: Array, path: String) {
    for key in expected {
        if !object.has(key) {
            _fail(path, "missing required field '" + key + "'")
        }
    }
    for key in object.keys() {
        if typeof(key) != "String" {
            _fail(path, "object keys must be strings")
        }
        if !expected.contains(key) {
            _fail(path, "unknown field '" + key + "'")
        }
    }
}
```

Rejecting unknown fields is the unusual half, and it is deliberate. A tolerant loader that silently ignores `"operator"` will also silently ignore `"engine_versoin"`, and then you have an artifact that loaded cleanly and lost a field. Fail closed.

24.6.2 Five tampering attempts

Here are five edits made to a real 4,947-byte artifact file, and what the loader said. Each block is the message carried by the thrown error, wrapped where it did not fit the page.

Listing 24.19: Adding a field the schema does not know

```
--- tamper: unknown field ---
assertion failed: verified artifact $: unknown field 'operator'
```

Listing 24.20: Deleting a field — even an optional-looking one

```
--- tamper: drop a field ---
assertion failed: verified artifact $: missing required field 'note'
```

`note` may be `null`, but it may not be *absent*. Optional in the domain does not mean optional on disk.

Listing 24.21: Making the top-level termination disagree with the summary

```
--- tamper: termination disagrees with summary ---
assertion failed: verified artifact $.termination: does not match $.summary.termination
```

The same redundancy that made the `RunRecord` a poor document makes the artifact a good one: the duplicate is checked.

Listing 24.22: Claiming a future schema

```
--- tamper: schema version ---
assertion failed: verified artifact $.artifact_schema_version: unsupported value 2
```

Listing 24.23: A non-finite number inside the embedded request

```
--- tamper: resolved request altitude ---
assertion failed: resolved request v1:
$.resolved_request.atmosphere.altitude_m: must be finite
```

Look at the prefix on that last one. It came from a different module.

24.7 `resolved_request_v1.lat`: the request at rest

An artifact embeds the `resolved_request` — the engine's own statement of every parameter it actually used, after defaults were applied. That object arrives as a plain `Map` decoded from JSON, and inside a loaded document it is entirely under the file's control. So before a single reconstructed domain value is allocated:

Listing 24.24: `verified_artifacts.lat:335–337`

```
// Validate the untrusted transport snapshot before allocating any reconstructed
// crystal graph. This keeps hostile requests outside the artifact boundary.
let resolved_request =
resolved_v1.validate_resolved_request_v1(record.resolved_request)
```

`resolved_request_v1.lat` is 335 lines and exports exactly one function. It validates all nine sections of a solve-json v1 request — projectile, rifle, shot, atmosphere, wind, wind segments, solver, effects, sampling — plus two cross-object relationships (the ground threshold must sit below the muzzle height; Coriolis requires a latitude). Then:

Listing 24.25: The last line of the module, :334

```
return freeze(clone(object))
```

((x)) — the detachment idiom

`clone` makes a value that shares nothing with the caller's; `freeze` makes it deeply immutable. Together they hand back a detached crystal while leaving the caller's `Map` independently mutable. The module header says it outright: *"The returned Map is a detached crystal… The caller's Map remains independently mutable."*

Yes, this duplicates `process_backend.lat`

`resolved_request_v1.lat` is close to a line-for-line twin of `process_backend.lat:141–329`. Before you file that as a defect, notice what it buys: the artifact path has *no dependency on the process backend*. You can load, validate, and inspect an artifact in a program that never spawns a child, never speaks the wire protocol, and never imports the transport. One module validates responses *in flight*; the other validates snapshots *at rest*. They will drift only when the protocol changes, and when it does, both must be edited — which is exactly the moment you want two deliberate edits rather than one shared function quietly changing meaning for two callers.

24.8 What verification is not

The word does a lot of work in ballistics, so let us be precise about what it means here — and what it does not.

It is not truing. Verifying a run says nothing about whether the run matched what your rifle did on paper. It says the run is internally consistent and completely recorded. Truing changes the *inputs*, and has its own chapter earlier in this book. Verification changes nothing.

It is not a signature. There is no hash, no key, no chain of custody. An artifact is a document that refuses to load if it has been edited into an inconsistent state — not one that proves who wrote it. Editing the `note` field, or changing a number in a way that keeps every cross-check satisfied, produces a file that loads perfectly.

It is not the `verified` flag on a `Trajectory`. `domain.trajectory` has a `verified` parameter that defaults to `false`, and `process_backend.lat` always passes `false`. Every table you print says `verified: no` and always will. The module's own comment is blunt about it: *"A Trajectory is never accepted directly, regardless of its legacy* `verified` *flag."* If you set that flag by hand and hand the trajectory to `verify_run`, you get the `expected RunRecord, got Trajectory` refusal shown earlier. The flag is vestigial; the boundary is the function.

There is no `:verify` command

The colon grammar has eleven commands and none of them makes an artifact. Verification is a deliberate act you perform in LATTICE:

```
let record = sessions.current_run(lab)
let artifact = artifacts.verify_run(record, nil, "600 yd check, 12 mph L-R")
artifacts.save_verified_run("/tmp/run-2026-08-05.json", artifact)
```

That is a design position, not an omission. The same argument keeps `:set` out of the grammar: state-changing operations stay visible LATTICE calls rather than a second, shadow command language.

24.9 The tests that hold it in place

`test_verified_artifacts.lat` is 432 lines and five tests, and its names are a fair summary of the module's contract:

Test	What it pins
`explicit_verification_and_default_selection`	the promotion and the first/middle/last rule
`explicit_selection_note_and_index_validation`	custom indices, notes, bad indices
`map_json_and_file_round_trips`	to/from Map, JSON, and file
`strict_documents_and_hostile_values`	the closed schema and every tamper
`drafts_and_legacy_verified_trajectories_are_rejected`	phase and the vestigial flag

It runs in the ordinary hermetic gate, on both virtual machines:

Listing 24.26: From `make test-ballistics-lab` (verbatim)

```
=== ballistics_lab backend --stack-vm ===
ballistics_lab verified artifacts: all tests passed
=== ballistics_lab backend --regvm ===
ballistics_lab verified artifacts: all tests passed
```

Chapter 25 takes that command apart.

24.10 A working recipe

Putting it together — this is the whole workflow, and it is short:

Listing 24.27: Solve, verify, save, reload

```
// 1. Solve. :solve does this for you; in source it is one call.
let record = sessions.solve_session(lab)

// 2. Promote. Pick the samples that carry the point of the run.
let artifact = artifacts.verify_run(
record,
[0, 50, 100],
"175 gr match, 45 deg N, Coriolis on"
)

// 3. Write. Fully validated and serialized before the file is touched.
artifacts.save_verified_run("runs/2026-08-05-175gr.json", artifact)

// 4. Later, anywhere: read it back. Any inconsistency throws here.
let reloaded = artifacts.load_verified_run("runs/2026-08-05-175gr.json")
print(reloaded.engine_version + " / " + reloaded.termination)
```

Step 4 is the reason for the other three. A year from now, on a different machine, possibly with a different engine build on the PATH, that file still tells you exactly which engine produced the numbers, exactly what it assumed, and exactly what request it resolved — or it refuses to load and tells you which field is wrong. That is the entire promise, and 724 lines is what it costs.

Chapter 25

Testing the Laboratory

The Ballistics Lab is 8,295 lines of library and application code and 5,767 lines of tests. About seven lines of implementation for every five lines of test. That ratio is unremarkable for scientific software and remarkable for a REPL, and it exists because of a claim the Ballistics Lab makes on every table it prints: *these numbers came from this engine under these assumptions*. A tool that makes that claim has to be able to prove it.

This chapter is about how those 5,767 lines are written, what they actually test, and — most usefully — how to run them yourself in seven seconds.

25.1 There are no test blocks

The first thing to unlearn

Lattice has a `test` keyword and `test { }` blocks. The Ballistics Lab uses **none of them**. There is no `tests/` directory, no runner harness, and no discovery mechanism. Every test module is an ordinary Lattice program with a `main()`, run by `clat` like any other script, that exits non-zero when an assertion throws.

Why? Because the Ballistics Lab's tests need to do things a block-scoped test facility makes awkward: spawn child processes, read fixture files from paths relative to the repository root, pass command-line arguments to select a scenario, dump snapshots to stdout for regeneration, and run the same file twice under two different virtual machines. An ordinary program does all of that for free.

The convention is uniform across all eighteen modules:

Listing 25.1: The shape of every test module, from `test_units.lat:1–26`

```
// Source-level tests for ballistics_lab/units.lat.
// Run from the repository root:
//   ./clat examples/ballistics_lab/test_units.lat
//   ./clat --regvm examples/ballistics_lab/test_units.lat

import "examples/ballistics_lab/units" as u

fn assert_close(actual: Number, expected: Number, tolerance: Number) {
    let delta = abs(actual - expected)
    assert(
    delta <= tolerance,
    "expected " + to_string(actual) + " to be within " +
    to_string(tolerance) + " of " + to_string(expected)
    )
}

fn capture_error(action: Fn) -> String {
    return try {
        action(nil)
        // Deliberately after the action: a validation path that falls through
        // returns this sentinel and fails the caller's expected-message check.
        "__NO_ERROR__"
    } catch e {
        e.message
    }
}
```

A header comment naming the two commands that run it. One import of the module under test. A few local helpers. Then a sequence of `fn test_*()` functions, and a `main()` that calls each in order and prints one success line:

Listing 25.2: `test_units.lat:263–285`, condensed

```
fn main() {
    test_quantities_preserve_si_and_display_metadata()
    test_known_distance_conversions()
    test_known_velocity_and_mass_conversions()
    ...
    test_dimension_annotations_reject_mismatched_quantities()
    print("units: 19 tests passed")
}
```

25.1.1 The two helpers you will meet everywhere

`assert_close` (or `assert_near`) is obvious. The interesting one is `capture_error` — sometimes called `caught` — which turns a thrown assertion into a string so a test can assert on the *message*. Two details make it correct.

First, it takes a closure, so the failing expression is not evaluated until the helper is ready to catch. Second, the sentinel string is returned *after* the action, not before: if the code under test fails to throw, the caller's `assert_contains(message, "...")` fails against `"__NO_ERROR__"` instead of silently passing. That ordering is the whole trick, and it is documented in the source.

Closures in Lattice need a space

`| |` is an empty parameter list. `||` is logical OR, and writing `caught(|| { ... })` is a parse error. The Ballistics Lab's helpers sidestep this by taking one ignored parameter — `capture_error( |_| { ... })` — which reads better anyway.

25.1.2 The assertions

Four builtins do the work, registered by the runtime itself: `assert`, `assert_eq`, `assert_type`, and `assert_contains`. A failed assertion throws; an uncaught throw exits non-zero. That is the entire test framework.

Listing 25.3: All four, from `test_units.lat:28–41`

```
fn test_quantities_preserve_si_and_display_metadata() {
    let distance = u.yd(300)
    assert_type(distance, "Distance")
    assert_close(distance.si_value, 274.32, 0.000000001)
    assert_eq(distance.display_value, 300)
    assert_eq(distance.display_unit, "yd")

    assert_type(u.fps(2700), "Velocity")
    assert_type(u.gr(175), "Mass")
    assert_type(u.mil(1), "Angle")
    ...
}
```

25.2 The eighteen modules

One test module per library module, plus four that test the assembled application. Fifteen are hermetic and run in the default gate; three require a real engine and are opt-in.

File	Covers	Lines	Gate
`test_units.lat`	`units.lat`	285	hermetic
`test_domain.lat`	`domain.lat`, `protocol_v1.lat`	347	hermetic
`test_reference_import.lat`	module isolation	13	hermetic
`test_backend.lat`	`process_backend.lat`, `backend.lat`	343	hermetic
`test_analysis.lat`	`analysis.lat`	287	hermetic
`test_analysis_export.lat`	`analysis_export.lat`	399	hermetic
`test_session.lat`	`session.lat`	532	hermetic
`test_persistence.lat`	`persistence.lat`	716	hermetic
`test_resolved_request_v1.lat`	`resolved_request_v1.lat`	287	hermetic
`test_verified_artifacts.lat`	`verified_artifacts.lat`	433	hermetic
`test_render.lat`	`render.lat`	747	hermetic
`test_command_parser.lat`	`command_parser.lat`	256	hermetic
`test_commands.lat`	`commands.lat`	442	hermetic
`test_repl_support.lat`	`repl_support.lat`	13	hermetic
`test_ballistics_repl.lat`	the whole application	370	hermetic
`test_engine_backend.lat`	real-engine smoke	41	opt-in
`test_ballistics_repl_engine.lat`	real-engine REPL	146	opt-in
`test_conformance.lat`	engine ↔ fixtures	110	opt-in

Two are worth noticing for their size. `test_repl_support.lat` is thirteen lines, because the module it tests is six — one pure predicate that answers *is this value the EOF sentinel?* It was extracted from

the REPL loop specifically so it could be tested without a terminal, and the test builds a `Unit` value the only way you can:

Listing 25.4: `test_repl_support.lat`, in full

```
import "examples/ballistics_lab/repl_support" as repl_support

fn unit_value() {
    let ignored = 1
}

fn main() {
    assert(repl_support.is_eof_value(nil), "Nil must be recognized as EOF")
    assert(repl_support.is_eof_value(unit_value()), "Unit must be recognized as legacy
    EOF")
    assert(!repl_support.is_eof_value(""), "an empty input line is not EOF")
    assert(!repl_support.is_eof_value(0), "an ordinary value is not EOF")
    print("ballistics_lab REPL support: all tests passed")
}
```

`test_reference_import.lat` is also thirteen lines, and its header explains why it exists: it imports `reference_experiment.lat` and *nothing else*, so that any module-environment leak — a module that only works because some other importer happened to bind a name — fails loudly and alone. §26.9 of the next chapter shows a live leak that this kind of isolation test is designed to catch.

25.2.1 Phase assertions

Scattered through the suite is a family of assertions you will not find in most codebases:

Listing 25.5: From `test_backend.lat:50–53` and `test_analysis.lat:191`

```
fn assert_error(result: any, code: String) {
    assert(backends.solve_failed(result), "expected LaboratoryError, got " +
    typeof(result))
    assert(result.code == code, "expected error " + code + ", got " + result.code + ":
    " + result.message)
    assert(phase_of(result) == "crystal")
}

    assert(phase_of(table) == "crystal")
```

These pin the BALLISTICS LAB's freeze contracts. A refactor that stopped freezing an `AnalysisTable` would still produce correct numbers and would still render identically — and would break the immutability guarantee the rest of the architecture leans on. `phase_of` is how that gets caught.

25.3 Test doubles

25.3.1 The fake engine is a real process

`fixtures/fake_solve_json_engine.lat` is 295 lines of LATTICE, and the process backend talks to it exactly the way it talks to `ballistics-engine`: over `exec_argv`, with argv and a request on stdin.

Listing 25.6: `test_backend.lat:34–47`

```
fn fake_backend(scenario: String, expected_version: any = nil) -> any {
    return process.process_backend_with_argv(
    clat_executable(),
    [FAKE_ENGINE, scenario],
    expected_version,
    limits()
    )
}

fn fake_solve(scenario: String, expected_version: any = nil) -> any {
    return backends.solve_with(
    fake_backend(scenario, expected_version),
    fixture.representative_fixture_experiment()
    )
}
```

The executable is `./clat` and the first argv element is the fake engine's source path. So the test spawns a second LATTICE interpreter, which reads the request, validates it, and emits one of roughly forty-five named scenarios chosen by its own `args()[1]`. That is heavier than a mock object and it buys something a mock cannot: the request really is serialized, really crosses a pipe, and really comes back through `json_parse`.

The scenarios fall into families, and reading the list is the fastest way to understand what "fail closed" means in practice:

Family	Scenarios
honest	`success`, `success-stderr`
transport	`timeout`, `stdout-overflow`, `stderr-overflow`
protocol errors	`validation-error`, `located-error`, `resource-error`, `internal-error`, `exit-mismatch`
framing attacks	`contaminated`, `trailing`, `concatenated`, `truncated`, `duplicate-envelope-key`
numeric attacks	`nonfinite`, `nonfinite-resolved`, `nonfinite-sample`, `oversized-integer`
sample-array attacks	`empty-samples`, `missing-terminal`, `early-terminal`, `multiple-terminal`, `sample-limit-exceeded`
summary agreement	`summary-distance-mismatch`, `summary-time-mismatch`, `summary-speed-mismatch`, `summary-energy-mismatch`
version/schema	`unknown-schema`, `unknown-status`, `wrong-version`

Most of them are produced by *string-replacing the good fixture*, which keeps every hostile case exactly one edit away from a document known to be valid. And the test that consumes them is a single loop:

Listing 25.7: `test_backend.lat:124–140`

```
fn test_fail_closed_responses() {
    for scenario in [
        "contaminated", "trailing", "concatenated", "truncated", "nonfinite",
        "nonfinite-resolved", "invalid-resolved-altitude", "invalid-resolved-ground",
        "invalid-resolved-effects", "invalid-resolved-coriolis",
    "invalid-resolved-zero-range",
        "nonfinite-sample", "empty-samples", "missing-terminal",
        "early-terminal", "multiple-terminal", "summary-distance-mismatch",
        "summary-time-mismatch", "summary-speed-mismatch", "summary-energy-mismatch",
        "sample-limit-exceeded", "unknown-status", "unknown-field", "exit-mismatch",
        "malformed-error-location", "unknown-error-field", "success-stderr"
    ] {
        assert_error(fake_solve(scenario), "invalid_engine_response")
    }
    ...
}
```

Twenty-seven hostile responses, one expected outcome. Note that `success-stderr` is in that list: an otherwise perfect success envelope that also wrote to stderr is an invalid response, because the protocol did not promise it.

25.3.2 Driving the fake engine by hand

`test_backend.lat` is dual-mode. Given an argument it runs one scenario and prints a line; given none it runs the suite. That makes debugging a protocol change a one-liner:

Listing 25.8: Single-scenario mode — verbatim output, shell prompts added, last message wrapped

```
$ ./clat examples/ballistics_lab/test_backend.lat success
success: 6 samples passed
$ ./clat examples/ballistics_lab/test_backend.lat sample-limit-exact
sample-limit-exact: 10000 samples passed
$ ./clat examples/ballistics_lab/test_backend.lat trailing
vm error: assertion failed: trailing failed: engine stdout is not exactly one
JSON envelope: json_parse error at position 3692: unexpected trailing content
```

`sample-limit-exact` is the boundary case: exactly 10,000 samples, accepted; `sample-limit-exceeded` is 10,001 and rejected.

25.3.3 Injected closure backends

The fake engine tests the transport. Everything above the transport uses a backend that is just a closure, and counts its own invocations through a `Ref`:

Listing 25.9: `test_commands.lat:74–82`

```
fn successful_backend(calls: any) -> any {
    return backends.engine_backend("command-backend", |experiment| {
        let run_number = calls.get() + 1
        calls.set(run_number)
        return trajectory_for(experiment, run_number)
    }
    )
}
```

Because `backend.lat` defines a backend as "an immutable struct wrapping one solve closure," any closure of the right shape is a backend. The command layer never imports `process_backend.lat` at all, so tests at that level run with no process, no engine, and no I/O — and can then make assertions no integration test could, such as *the backend was invoked exactly once*.

`test_session.lat` takes the same idea further with four injected backends: one that succeeds, one that returns a `LaboratoryError`, one that throws, and one that returns a trajectory whose experiment does not match the session snapshot. All four are two-line closures.

25.4 Golden fixtures

Six JSON files at the root of `fixtures/` and twelve more under `fixtures/conformance/` are checked into the repository.

The most consequential is 1,032 bytes long:

Listing 25.10: `test_backend.lat:56–64`

```
fn test_golden_request() {
    let experiment = fixture.representative_fixture_experiment()
    let actual = json_parse(protocol.v1_request_json(experiment))
    let expected =
     json_parse(read_file("examples/ballistics_lab/fixtures/representative.request.json"))
    assert(
    actual == expected,
    "canonical v1 request drifted from fixture"
    )
}
```

That is the entire Ballistics Lab-to-engine contract in nine lines. Change a field name in the protocol module, change a unit, add a key — and this test fails before anything else does.

Two representative experiments, and they are not the same load

`reference_experiment.lat` is the *documentation* load: 175 gr G7 match bullet, imperial units, 1,000 yd, Coriolis on, 45° latitude. It is what the REPL starts with and what most of this book uses. `fixture_experiment.lat` is the *test* load: raw SI throughout, 0.01134 kg, 823 m/s, a 50 m maximum range, no latitude, Coriolis off. The tiny range is the point — it makes `representative.success.json` six samples and 3,691 bytes, small enough to audit by eye. Mixing the two up will put wrong numbers in your prose.

25.5 Snapshot tests, and how to regenerate them

`test_render.lat` is the largest test module at 747 lines, and most of that bulk is expected strings. Rendering is exactly the kind of code where a snapshot test earns its keep: the output is deterministic, human-meaningful, and tedious to assert field by field.

The danger with snapshots is that regenerating them becomes a manual chore and people stop doing it. So the module ships a dump mode:

Listing 25.11: `test_render.lat:736–747` (re-indented)

```
fn main() {
    if args().contains("dump") {
        dump_snapshots()
        return
    }
    test_profiles_and_validation()
    test_notice_and_error_snapshots()
    test_fixed_fallback_and_flag_escaping()
    test_renderer_shapes_and_signs()
    test_row_map_order_and_validation()
    print("ballistics_lab render: all tests passed")
}
```

`dump_snapshots()` prints every renderer's output between marker lines:

Listing 25.12: `./clat examples/ballistics_lab/test_render.lat dump` (verbatim, excerpt)

```
===DOPE_METRIC===
DOPE: snapshot experiment
Distance (m) | Drop (m) | Come-up (mil) | Velocity (m/s) | Energy (J) | Time (s) | Mach
-------------+----------+---------------+----------------+------------+----------+-----
      100.00 |    +0.19 |         +1.87 |         708.33 |    2550.00 |     0.15 | 2.06
      200.00 |    +0.45 |         +2.25 |         610.00 |    2050.00 |     0.31 | 1.82
Signs: drop + below/- above; come-up + correction up/- correction down
Provenance
  [0] snapshot experiment
    engine: snapshot-engine-b (schema 1)
    solver: rk45
    verified: yes
    termination: velocity_floor
    actual range: 200.00 m
    Assumptions
      - model_default: Used G7 \| model. [$.projectile.drag_model]
    Warnings
      - range_notice: Line one.\nLine two. [$.shot.max_range]
```

The whole dump is 311 lines across thirteen markers. Change a renderer deliberately, run the dump, paste the new section into the expected-string function, and the diff in code review is the diff a reader will see on screen.

Two things in that excerpt are themselves test data. The notice text contains a pipe and a newline, so the snapshot pins the escaping chain. And the fixture's drop is *positive* with a *positive* come-up — the sign relationship `analysis.lat` implements, frozen into a string so that a change to it cannot pass unnoticed.

25.5.1 When a snapshot pins the wrong answer

That last claim is worth stopping on, because until LATTICE commit `cd92f9e` it said the opposite — and every test that touched the come-up column agreed with it.

Section 23.6 tells the story from the rendering side: the come-up column had the sign of its `atan2` argument backwards, so it told a shooter to dial down for a target that needed up. What matters here is how thoroughly the test suite endorsed it.

One assertion pinned the sign directly. `test_analysis.lat` asserted `come_up < 0` for a fixture with a positive drop, under the message "positive fixture drop requires a negative come-up." That is not a gap in coverage. It is a specification, and it was wrong.

The fixtures agreed with the assertion. The synthetic trajectories in `test_render.lat` — and the ones behind `test_commands.lat` and `test_verified_artifacts.lat` — supplied drop values that grew more *negative* with distance:

Listing 25.13: `test_render.lat:81–85` as it stood before `cd92f9e`

```
observations = [
    sample_observation(0, 0, 800, 3000, -0.00004, 0.00004, 2.30, false),
    sample_observation(100, 0.14, 700, 2500, -0.12, -0.03, 2.05, false),
    sample_observation(200, 0.30, 600, 2000, -0.50, -0.08, 1.80, true)
]
```

Those fifth arguments are drop in metres. Fed a drop that goes negative downrange, the inverted expression produces a *positive* come-up, and the snapshot recorded a plausible-looking dope table. The legends agreed with the fixtures: `:dope` printed `drop + above/- below`, `:at` printed `drop/windage + above/right, - below/left`, and `:compare` printed `impact + above/right` over columns that are actually drop and windage. Everything was consistent. Nothing was right.

Those three strings are quoted here as evidence and nowhere else. All three were deleted outright by `cd92f9e`, and none of them survives anywhere in `examples/ballistics_lab/`. If you meet one in a transcript, that transcript came from the broken build.

What a snapshot test actually protects

A snapshot test proves that today's output equals yesterday's output. It says nothing about whether either is correct. When the fixture, the assertion, the legend and the implementation all come from the same author on the same afternoon, a snapshot suite will lock in that author's belief with impressive rigour — across both virtual machines, on every commit, for as long as nobody checks it against reality.

The fix touched nine files. Two carried the defect itself — `analysis.lat` for the sign, `protocol_v1.lat` for the zero datum. One more corrected three sign legends (`render.lat`), and one was the README. The remaining *five* were test files: the assertion that pinned the sign, the fixture generators that manufactured drop with the wrong sign, and the snapshots that had recorded the result. Every one of them had to be corrected to the engine's convention before the real fix could pass.

The corrective is not "write better snapshots." It is to hold at least one assertion against something the suite did not produce. This suite has two tests of that kind — the golden request fixture of §25.4, and the conformance runner of §25.8, which replays engine-authored fixtures through a real engine — and neither of them reached this defect. The come-up column is derived inside the Ballistics Lab and has no counterpart anywhere on the wire, so no external witness for it exists in the repository at all. The zero datum *is* on the wire, and the fixtures spell it out correctly; the tests still missed it, for a reason §25.8 takes apart.

That is the shape of the gap. The further a value sits from anything the Ballistics Lab did not author itself, the more of its correctness rests on somebody having checked it by hand — and in this case somebody eventually did, by running the Ballistics Lab against the engine's own `come-ups` subcommand while this book was being written. Section 23.6 carries the before-and-after numbers.

The lasting change is not the two corrected expressions. It is that the suite now says the true thing about each column, so the next person to touch `analysis.lat` inherits a specification instead of a preserved mistake. `render.lat` emits four sign legends and no others, and `test_render.lat` pins all four by snapshot:

Listing 25.14: Every sign legend the Ballistics Lab prints, from `render.lat`:577, 954, 969 and 994

```
Signs: drop + below/- above; windage + right/- left
Signs: drop + below/- above; come-up + correction up/- correction down
Signs: windage + right/- left; wind hold + correction right/- correction left
Signs: drop + below/- above; windage + right/- left; deltas are series minus baseline [0]
```

Read those against the retired three above. Each corrected legend names the columns of the table it sits under — and the fourth, the wind card's, is unchanged, because windage and its hold were

signed correctly all along. Section 23.6 works through why one of the two derived angles negates and the other must not.

25.5.2 The Map-order test

The one test in that file that is not a snapshot comparison is the most paranoid:

Listing 25.15: `test_render.lat:597–621` (re-indented)

```
fn test_row_map_order_and_validation() {
    let source = metric_dope(sessions.current_run(lab))
    flux reversed_rows = []
    for source_row in source.rows {
        let reversed = Map::new()
        flux index = len(source.columns) - 1
        while index >= 0 {
            let column = source.columns[index]
            reversed.set(column.key, source_row.get(column.key))
            index = index - 1
        }
        reversed_rows.push(reversed)
    }
    let reordered = AnalysisTable { ..., rows: reversed_rows, ... }
    let reordered_text = render.render_dope(reordered, metric)
    assert(reordered_text == expected_dope_metric(),
    "row Map insertion order affected rendering:\n" + reordered_text)
}
```

Every row is rebuilt with its keys inserted in reverse, and the rendered output must be byte-identical. That is the executable form of "never iterate a Map."

Declare local structs in tests, not imported ones

Notice that test builds an `AnalysisTable` from a *locally declared* struct with the same field names, and the file says why: *"Struct-name validation is intentionally exercised without selectively importing exported struct types, which is not portable across both VM backends."* A qualified struct literal such as `analysis.AnalysisTable { ... }` fails with `unknown struct 'AnalysisTable'`. Since the Ballistics Lab validates by `struct_name`, a local declaration works everywhere.

25.6 Transcript tests

`test_ballistics_repl.lat` tests the assembled application by *being a user*. It builds a newline-joined script of REPL input, runs the whole app as a child process, and asserts on the captured stdout.

Listing 25.16: `test_ballistics_repl.lat:44–51`

```
fn run_app(backend: String, scenario: String, input_text: any) -> Map {
    return exec_argv(
    executable(),
    app_args(backend, scenario),
    input_text,
    child_limits()
    )
}
```

The transcript itself is sixty lines of input, and it reads like a real session — because it is one. Structs are inspected, a projectile is swapped and swapped back, an array is destructured, a module is imported *at the prompt*, a multi-line function is defined and called later, four different kinds of error are provoked and recovered from, and then the whole colon surface is exercised in order.

The assertion helper that makes it a specification rather than a grep is this one:

Listing 25.17: `test_ballistics_repl.lat:89–99`

```
fn assert_in_order(label: String, output: String, expected: Array) {
    flux remaining = normalize_output(output)
    for value in expected {
        let position = remaining.index_of(value)
        assert(
        position >= 0,
        label + " missing ordered fragment '" + value + "': " + output
        )
        remaining = remaining.substring(position + len(value), len(remaining))
    }
}
```

Each fragment is searched for in what remains *after* the previous match, so the assertion pins sequence, not just presence. The ordered list of expected fragments runs to forty-odd entries and is, effectively, an executable specification of a full laboratory session.

25.6.1 The security assertions

The most valuable four lines in the whole suite are these. The transcript deliberately feeds `:load` two hostile paths — one shaped like Lattice source, one shaped like a shell command substitution:

Listing 25.18: `test_ballistics_repl.lat:105–106 and 216–226`

```
let source_shaped_path = "assert(false, 'executed')"
let shell_shaped_path = "$(touch " + shell_marker + ")"
...
let shell_marker_exists = file_exists(shell_marker)
...
assert(saved_exists, label + " :save did not create its literal path: " + output)
assert(!shell_marker_exists,
label + " executed shell-shaped command text: " + output)
```

If any layer of the Ballistics Lab ever interpolated a user string into a shell command or evaluated it as source, that marker file would appear and the test would fail. This is the empirical backing for the claim that user values never reach a shell — not an argument, a file that is not there.

Side effects are observed and cleaned before assertions

The test reads `file_exists` for both markers, deletes what it finds, and only *then* asserts. A failing assertion therefore cannot leave litter behind for the next run to trip over. The comment in the source calls this out explicitly. It is a small habit that makes a suite re-runnable after a failure.

25.6.2 Pinning the environment from inside the test

Listing 25.19: `test_ballistics_repl.lat:361–366`

```
// Pin the checked fixture contract rather than inheriting a developer's
// ambient real-engine version requirement into nested children.
env_set("BALLISTICS_ENGINE_VERSION", "0.24.1")

check_backend("stack-vm", "")
check_backend("regvm", "--regvm")
```

The fixtures were generated by engine 0.24.1. A developer with 0.32.0 on the PATH, and the environment variable `BALLISTICS_ENGINE_VERSION` exported to match, would otherwise poison every child

process. So the test sets the variable itself. Note also the last two lines: this module runs both virtual machines from *inside*, spawning its own children, which is why the Makefile invokes it only once.

25.7 Running the tests

25.7.1 The default gate

One command, no engine required, no network:

Listing 25.20: `make test-ballistics-lab` — verbatim, register-VM block condensed; wall time from `/usr/bin/time`

```
$ make test-ballistics-lab
sh tests/test_ballistics_lab_launcher.sh
Running POSIX ballistics Lab launcher tests...
  ok: default stack-vm and repository-local discovery
  ok: CLI discovery and backend precedence with spaced paths
  ok: environment discovery and backend precedence
  ok: long and shorthand backend selection
  ok: relative PATH entries survive the bundle-root chdir
  ok: help, VM-only backend validation, and option errors do not launch clat
  ok: invalid explicit overrides fail without fallback
  ok: incompatible PATH engines are skipped but explicit engines fail
  ok: clat exit status is preserved
Results: 9 launcher test groups passed
=== ballistics_lab backend --stack-vm ===
units: 19 tests passed
ballistics_lab domain: all tests passed
ballistics_lab isolated reference import: passed
ballistics_lab process backend: all tests passed
ballistics_lab analysis: all tests passed
ballistics_lab analysis export: all tests passed
ballistics_lab session: all tests passed
ballistics_lab persistence: all tests passed
ballistics_lab resolved request v1: all tests passed
ballistics_lab verified artifacts: all tests passed
ballistics_lab render: all tests passed
ballistics_lab command parser: all tests passed
ballistics_lab commands: all tests passed
ballistics_lab REPL support: all tests passed
experiment=175 gr match at 1,000 yards
schema_version=1
mass_kg=0.0113398
muzzle_velocity_mps=822.96
max_range_m=914.4
=== ballistics_lab backend --regvm ===
units: 19 tests passed
...
=== ballistics_lab REPL transcripts ===
./clat examples/ballistics_lab/test_ballistics_repl.lat
stack-vm: ballistics laboratory transcript passed
regvm: ballistics laboratory transcript passed
real 6.96
```

Seven seconds, wall clock, for the entire Ballistics Lab. Four things are happening in there.

The launcher is tested first, by 412 lines of POSIX shell that copy `ballistics-lab.sh` into disposable bundles and exercise discovery, relocation, backend selection, and paths containing spaces — without ever invoking a real `clat` or a real engine.

Every module runs twice, once on the stack VM and once on the register VM. The Makefile loop is literally `for backend in "" "--regvm"`.

`reference_experiment.lat` is run as a test. Those five `experiment=` lines are its `main()`. It asserts nothing; running without failing is the assertion, and it catches import-time breakage in the load every other test depends on.

The transcript test runs once, last, because it spawns both backends itself.

Backend divergence is a Lattice defect, not a Ballistics Lab bug

The Ballistics Lab supports two execution backends: the stack VM (default) and the register VM (`--regvm`). The tree-walking interpreter is deliberately not supported. Any difference in Ballistics Lab behaviour between the two supported backends is defined as a defect in the language runtime, which is precisely why every test file runs twice.

25.7.2 One file at a time

The header of every test module tells you how, and it always says "from the repository root" because import paths are root-relative:

Listing 25.21: The developer loop

```
$ ./clat examples/ballistics_lab/test_render.lat
ballistics_lab render: all tests passed
$ ./clat --regvm examples/ballistics_lab/test_render.lat
ballistics_lab render: all tests passed
```

25.7.3 The opt-in engine suite

The default gate never touches `ballistics-engine`. To test against the real thing you must say where it is:

Listing 25.22: `make test-ballistics-lab-engine` against engine 0.32.0 — verbatim, second conformance block condensed

```
$ make test-ballistics-lab-engine \
    BALLISTICS_ENGINE=/Users/alexjokela/.local/bin/ballistics \
    BALLISTICS_ENGINE_VERSION=0.32.0
=== ballistics_lab real engine --stack-vm ===
ballistics_lab real engine: 0.32.0 passed
=== ballistics_lab real engine --regvm ===
ballistics_lab real engine: 0.32.0 passed
=== ballistics_lab real-engine REPL transcripts ===
stack-vm: real-engine ballistics REPL transcript passed
regvm: real-engine ballistics REPL transcript passed
=== ballistics_lab cross-interface conformance --stack-vm ===
conformance calm-air: ok (engine 0.32.0)
conformance crosswind: ok (engine 0.32.0)
conformance segmented-wind: ok (engine 0.32.0)
conformance zeroed: ok (engine 0.32.0)
conformance representative: ok (engine 0.32.0)
conformance early-termination: ok (engine 0.32.0)
ballistics_lab conformance: all fixtures passed
=== ballistics_lab cross-interface conformance --regvm ===
conformance calm-air: ok (engine 0.32.0)
...
ballistics_lab conformance: all fixtures passed
real 0.25
```

If `BALLISTICS_ENGINE` is unset the target hard-fails with a usage message rather than silently testing nothing. `BALLISTICS_ENGINE_VERSION` is optional: leaving it empty disables the version pin while keeping strict schema validation.

25.8 Conformance: the fixtures are shared with the engine

This is the most interesting test in the Ballistics Lab, and it is only 110 lines. The six request/response pairs under `fixtures/conformance/` are the *same files* the engine repository uses to pin its own Rust service and CLI transport tests. The Ballistics Lab replays each request through the same runtime primitives its process backend uses, and compares the parsed envelope with the checked-in expected envelope.

Listing 25.23: `test_conformance.lat:1–12`

```
// MBA-1309: cross-interface conformance. The six checked-in fixture pairs are shared
// with the engine repository (tests/fixtures/solve_json_v1/), where the Rust service
// and CLI-transport tests pin them. This runner replays each REQUEST through the same
// runtime primitives the laboratory's process backend uses (exec_argv + json_parse)
// against a real ballistics executable and asserts the parsed envelope matches the
// checked-in SUCCESS envelope:
//   - exactly for schema_version, status, assumptions, warnings, and termination;
//   - numerically within tolerance (relative 1e-6 with an absolute floor of 1e-9)
//     for every number, so identical-version engines match tightly while minor
//     engine evolution surfaces as a reviewable fixture update;
//   - engine_version is informational and exempt (fixtures record the generator's).
// The executable path arrives as argv and is never interpolated into a shell.
```

The tolerance policy is the design decision worth studying. Strings, booleans, statuses, and notice codes must match *exactly*. Numbers must match to a relative 10^{-6} with an absolute floor of 10^{-9}. And `engine_version` is exempt entirely.

Listing 25.24: `test_conformance.lat:18–25`

```
fn numbers_match(expected: Number, actual: Number) -> Bool {
    let difference = abs(actual - expected)
    flux tolerance = abs(expected) * 0.000001
    if tolerance < 0.000000001 {
        tolerance = 0.000000001
    }
    return difference <= tolerance
}
```

Set the tolerance to zero and every engine patch release breaks the build. Set it too loose and a real physics regression slips through. At 10^{-6} relative, an engine that changed a drag coefficient would fail loudly, while an engine that reordered a floating-point summation would not.

The comparison itself is a recursive structural walk that checks both directions — every expected key must be present, and every actual key must be expected — so an engine that *added* a field to the envelope is a conformance failure, not a silent pass.

The fixtures knew about the zero datum. The tests still missed it.

Every checked-in request in this repository — all six conformance requests and the golden `representative.request.json` — carries `"target_height_m": 1.25` beside `"muzzle_height_m": 1.2` and `"sight_height_m": 0.05`. The zero solved to the line of sight, $1.2 + 0.05$, spelled out.

And the BALLISTICS LAB still failed to send that field on every interactive solve until `cd92f9e`, which is the second half of the defect §25.5.1 describes. The reason is worth writing down. `fixture_experiment.lat` builds its `Shot` with `m(1.25)` in the target-height slot, so the golden-request test always exercised the branch of `protocol_v1.lat` that copies a supplied value. The branch that mattered — what the encoder emits when the domain `Shot` leaves target height `nil`, which is what every experiment a user types at the prompt does — had no fixture at all.

A hundred percent of the fixtures agreeing about a field proves nothing if none of them reaches the code path that omits it.

These fixtures were generated by engine 0.24.1 and still pass on 0.32.0

Eight minor versions later, all six pairs match within tolerance against the engine on this machine. That is a meaningful statement about the stability of solve-json v1 — and it is also why the comparison exempts `engine_version`. If a future engine does change a number materially, the failure names the fixture, the JSON path, and both values, and the correct response is a reviewed fixture update, not a widened tolerance.

25.9 Reading a failure

Assertions throw, and LATTICE prints a stack trace. Here is a real one, from a scenario deliberately run in single-scenario mode:

Listing 25.25: A failing assertion — verbatim, first message wrapped

```
vm error: assertion failed: trailing failed: engine stdout is not exactly one
JSON envelope: json_parse error at position 3692: unexpected trailing content
stack trace (most recent call last):
  [line 322] in <script>
  [line 328] in main()
```

Three things to read, in order. The **message** is the assertion's own string — the BALLISTICS LAB writes these as full sentences with a path, so the message alone usually identifies the field. The **deepest frame** is the last line. And the `<script>` frame is the module's top level.

When the failure comes out of a caught error rather than an uncaught throw, you get the error object serialized instead, which carries the same information plus the whole call chain:

Listing 25.26: A caught error object (verbatim, wrapped)

```
{"message": assertion failed: render.table.columns[2].unit: expected 'mil',
got 'moa', "line": 12, "stack": [_fail() at line 12, _expect_columns() at line
807, render_dope() at line 946, <closure> at line 33, caught() at line 11,
main() at line 33, <script> at line 18]}
```

One backend difference you will see in stack traces

The `<script>` frame reports a real line number under the stack VM and `line 0` under the register VM. Everything else in that error object — message, codes, function frames — is identical. It is cosmetic, but if you are diffing test output across backends, that is the line that will differ.

25.10 Writing a test for your own change

The recipe follows from everything above.

1. **Create `test_<module>.lat`** next to the module, with a header naming both run commands.
2. **Import only what you need**, and declare local structs rather than importing exported struct types.
3. **Copy the two helpers** — an `assert_close` and a `capture_error` with a sentinel.
4. **Write `fn test_*()` functions** that are self-contained: no shared mutable state between them, so any one can be commented out.
5. **Assert on phase** where the module makes a freeze promise.
6. **Inject a closure backend** rather than a process, unless you are specifically testing the transport.
7. **End `main()` with one print** — the suite reads as a list of module names, and a missing line is as informative as a failure.
8. **Add the file to `BALLISTICS_LAB_TESTS`** in the Makefile so it runs on both backends.

Chapter 26 does exactly this: it adds a command to the Ballistics Lab, and then adds a 182-line test module for it that passes on both virtual machines.

Chapter 26

Extending the Laboratory

The Ballistics Lab ships with eleven colon commands. Sooner or later you will want a twelfth — some question you ask often enough that typing the Lattice for it every time becomes friction. This chapter adds one, end to end, and then runs it, tests it, and breaks it on purpose.

We will build `:mach <value>`: *at what range does this bullet fall to a given Mach number?* It is a real question — transonic transition is where group dispersion tends to open up — and it exercises every layer of the Ballistics Lab without being trivial. When we are done the change will be 88 lines across four modules, plus a 182-line test.

26.1 Decide where the change belongs

Before writing anything, work out which layers your feature touches. The Ballistics Lab's module graph is a directed acyclic graph with six roots and a single sink, and that shape is what makes the question answerable.

If your change is…	it belongs in…
a new unit or conversion	`units.lat`
a new field of a shot, rifle, or load	`domain.lat`
a new thing to send the engine	`protocol_v1.lat`
a new thing the engine sends back	`process_backend.lat`
a new calculation over a trajectory	`analysis.lat`
a new way to print a result	`render.lat`
a new machine-readable format	`analysis_export.lat`
a new colon command	`command_parser.lat` + `commands.lat`
a new kind of session state	`session.lat`
a new document on disk	`persistence.lat` or `verified_artifacts.lat`
a new transport to a solver	`backend.lat` (a closure)

A new command is the interesting case because it is the one that touches four layers at once:

1. **Calculation** — `analysis.lat` learns to compute the answer.
2. **Grammar** — `command_parser.lat` learns to recognize the command and turn its arguments into inert data.
3. **Presentation** — `render.lat` learns to print it.
4. **Routing** — `commands.lat` connects the three.

Note what is *not* on that list. `session.lat` does not change: the new command reads the current run and mutates nothing. `domain.lat`, `units.lat`, `protocol_v1.lat`, and `process_backend.lat` do not change: no new quantity crosses the wire. And `ballistics-repl.lat` does not change either — the REPL has no command table of its own; it forwards every colon line to `commands.execute_command`.

Work in a copy first

Copy `examples/ballistics_lab/` somewhere writable, preserving the `examples/ballistics_lab/` path (import paths are repository-root relative), symlink `clat` and `build/process_fixture` into the new root, and you have a fork you can run and test exactly like the original without touching your checkout. Everything in this chapter was done that way.

26.2 Layer one: the calculation

`analysis.lat` already has everything we need. A trajectory's observations carry `mach`; `_interpolate` does linear interpolation; `_distance_from_si` builds a `Distance` in a chosen display unit; and `trajectory_at` will produce a complete interpolated `Observation` once we know the range.

So the new function finds the bracket where Mach crosses the target on the way down, interpolates the *distance*, and delegates:

Listing 26.1: Added to `analysis.lat` — 37 lines

```
// Locate the first descending crossing of a Mach number and return the
// interpolated Observation there. Mach is monotonically decreasing over the
// usual supersonic trajectory, so the first bracket is the answer.
export fn mach_crossing(trajectory_value: any, mach: any) -> any {
    _validate_trajectory(trajectory_value, "mach_crossing.trajectory")
    let kind = typeof(mach)
    if kind != "Int" && kind != "Float" {
        assert(false, "mach_crossing.mach: expected Number, got " + kind)
    }
    if is_nan(mach) || is_inf(mach) || mach <= 0 {
        assert(false, "mach_crossing.mach: must be a positive finite number")
    }
    let observations = trajectory_value.observations
    flux index = 1
    while index < len(observations) {
        let left = observations[index - 1]
        let right = observations[index]
        if left.mach >= mach && right.mach < mach {
            let span = left.mach - right.mach
            let fraction = (left.mach - mach) / span
            return trajectory_at(
            trajectory_value,
            _distance_from_si(
            _interpolate(left.distance.si_value, right.distance.si_value, fraction),
            left.distance.display_unit
            )
            )
        }
        index = index + 1
    }
    assert(
    false,
    "mach_crossing: trajectory never falls to Mach " + to_string(mach) +
    "; terminal Mach is " + to_string(observations[len(observations) - 1].mach)
    )
    return nil
}
```

Five habits from the surrounding code are worth naming, because copying them is most of what "fits the codebase" means here.

Validate the trajectory first, with the module's existing `_validate_trajectory`. That one call guarantees non-empty observations, strictly increasing distances, and exactly one terminal sample at the end — so the loop below needs no defensive checks.

Take `any`, not `Number`, and check the type yourself. The BALLISTICS LAB does this everywhere: a declared parameter type produces a runtime message in the language's vocabulary, and a hand-written check produces one in the module's.

Name the path in every message. `"mach_crossing.mach: ..."` matches the `<function>.<argument>` convention used throughout, which is what makes an error legible six frames down a stack.

Delegate rather than duplicate. We interpolate one number — the distance — and let `trajectory_at` interpolate the other six fields. That also means the returned observation gets the empty-flags treatment already established for interpolated points.

Make the failure diagnostic actionable. "Never falls to Mach 1.2" is half an answer; "terminal Mach is 1.25598" tells the user their trajectory is too short, which is the actual problem.

Why the guard is `left.mach >= mach && right.mach < mach`

It brackets a *descending* crossing, and it treats an exact hit on the left sample as a crossing so that querying a Mach value that lands precisely on a grid point returns that point rather than skipping past it. A trajectory that is already subsonic at the muzzle finds no bracket at all and gets the diagnostic. The book's 140 gr load at 2,000 yd crosses Mach 1.2, 1.0, and 0.9 in that order, so the first bracket found is always the right one.

26.3 Layer two: the grammar

`command_parser.lat` produces inert data. It never evaluates anything, never touches the session, and never calls the layer above it. Adding a command means adding a parse function and one dispatch line.

Our command takes one positive number, so it is modelled on `_parse_at` but simpler — no unit token:

Listing 26.2: Added to `command_parser.lat` — 23 lines

```
fn _parse_mach(name: String, tokens: Array) -> any {
    let arity_error = _arity(name, len(tokens), 1)
    if arity_error != nil {
        return arity_error
    }
    let mach = _number(tokens[0], 0)
    if parse_failed(mach) {
        return mach
    }
    if mach <= 0 {
        return _path_error(
        "command_argument",
        ":mach value must be greater than zero",
        "arguments[0]"
        )
    }
    flux args = Map::new()
    args.set("mach", mach)
    return _command(name, args)
}
```

Listing 26.3: …and three lines in `parse_command`

```
    if name == "mach" {
        return _parse_mach(name, arguments)
    }
```

Notice that nothing throws. Every failure is a `LaboratoryError` *returned* as a value, tagged with a code and a JSON-path-style `path` pointing at the offending argument. The caller tests with `parse_failed()`. This is the Ballistics Lab's rule at module seams: failure is data, not control flow.

`_number` and `_arity` already exist and already produce the canonical messages. Reusing them is why the new command's errors are indistinguishable in style from the built-in ones:

Listing 26.4: The new command's error surface (verbatim)

```
:mach 0
Error
  code: command_argument
  message: :mach value must be greater than zero
  path: arguments[0]
:mach
Error
  code: command_arity
  message: :mach expects 1 argument(s), got 0
  path: arguments
:mach abc
Error
  code: command_argument
  message: expected a numeric argument, got 'abc'
  path: arguments[0]
```

The middle and last of those three came entirely from existing helpers. We wrote one error message; the parser supplied the other two.

26.4 Layer three: presentation

The result is an `Observation`, which `render.lat` already knows how to print. All we need is a heading that says which Mach number was asked for, formatted at the session's precision:

Listing 26.5: Added to `render.lat` — 6 lines

```
export fn render_mach_crossing(mach: any, observation: any, display: any) -> String {
    let value = _require_finite(mach, "render.mach")
    let profile = _profile(display)
    return "Mach " + _fixed(value, profile.precision) + " crossing\n" +
    render_observation(observation, profile)
}
```

Six lines, and three of them are conventions. `_require_finite` is the module's own numeric guard. `_profile` revalidates and copies the display preferences, so the new renderer cannot be handed a half-built profile. And `_fixed` means the heading obeys the same precision as everything else on screen — at precision 2 it reads `Mach 1.20 crossing`, at precision 0 it would read `Mach 1 crossing`.

Place new functions before line 678

`render.lat`'s indentation drifts progressively rightward from line 678 to the end of the file. Adding a function above that point — next to `render_observation`, where this one logically belongs — keeps your diff readable.

26.5 Layer four: routing

`commands.lat` is where the three pieces meet, and it takes four edits. Miss any one and the command fails in a different, instructive way (§26.8).

One: the name list. `_validate_command` refuses any name not in this array, because `dispatch_command` accepts hand-built `ColonCommand` values that never went through the parser.

Listing 26.6: `commands.lat:23–28`

```
fn _command_names() -> Array {
    return [
        "help", "show", "solve", "at", "mach", "dope", "wind-card", "compare",
        "save", "load", "reset", "quit"
    ]
}
```

Two: the help text. There is no generated help; `:help` prints a literal string.

Listing 26.7: `commands.lat:30–44`

```
    ":at <distance> <unit>\n" +
    ":mach <value>\n" +
    ":dope <start> <stop> <interval> <unit>\n" +
```

Three: re-validation. The parser already checked the argument. The dispatcher checks it again, because it is a public entry point:

Listing 26.8: Added to `_validate_command`

```
    if name == "mach" {
        _require_exact_args(args, ["mach"], name)
        let mach = _number(args.get("mach"), "dispatch_command.mach.args.mach")
        if mach <= 0 {
            _fail("dispatch_command.mach.args.mach: must be greater than zero")
        }
        return command
    }
```

`_require_exact_args` is the closed-schema check again: an args map with a missing key *or* an extra key is rejected.

Four: the dispatch branch.

Listing 26.9: Added to `_dispatch_checked`

```
    if name == "mach" {
        let current = _state_run(lab, false, name)
        if command_failed(current) {
            return current
        }
        let observation = analysis.mach_crossing(current.result, args.get("mach"))
        return _response(
        name,
        render.render_mach_crossing(args.get("mach"), observation, profile),
        observation
        )
    }
```

`_state_run` is the shared helper that fetches the current run and returns a `command_state_error` when there isn't one — so `:mach` before `:solve` behaves exactly like `:at` before `:solve`, for free.

And `_response` carries three things: the rendered text, the *value* (so a caller working in source gets the `Observation`, not a string), and the quit flag.

Everything after this is already handled

The new branch does not need a `try`. `dispatch_command` wraps validation in one `try` and execution in another, converting a throw into either a `command_validation_error` or a `command_execution_error`. Our assertion inside `mach_crossing` becomes a rendered error and the REPL survives — which is exactly what happens when the trajectory never gets slow enough.

26.6 Running it

That is the whole change: 88 added lines across four files. Nothing else in the Ballistics Lab was touched.

Listing 26.10: The diff, by file — paths abridged

```
$ diff -rq ../lattice/examples/ballistics_lab fork/examples/ballistics_lab
Files ... analysis.lat and ... analysis.lat differ
Files ... command_parser.lat and ... command_parser.lat differ
Files ... commands.lat and ... commands.lat differ
Files ... render.lat and ... render.lat differ
Only in fork/examples/ballistics_lab: test_mach.lat

added lines:  analysis 37   command_parser 23   commands 22   render 6
```

Run the fork the way the launcher does — `clat`, the REPL script, and the engine path as argv:

Listing 26.11: Launching the fork

```
$ cd fork
$ ./clat examples/ballistics_lab/ballistics-repl.lat \
      /Users/alexjokela/.local/bin/ballistics
```

`:help` now lists the command in place:

Listing 26.12: `:help` in the fork (verbatim)

```
Commands:
:help
:show
:solve
:at <distance> <unit>
:mach <value>
:dope <start> <stop> <interval> <unit>
:wind-card <start> <stop> <interval> <unit>
:compare
:save "<path>"
:load "<path>"
:reset
:quit
```

And here it is doing its job. The load is the book's 140 gr ELD-M, pushed out to a 2,000-yard maximum range so the bullet actually goes subsonic:

Listing 26.13: `:mach` against engine 0.32.0 — verbatim, run summary elided

```
:solve
  termination      : max_range
  actual range     : 2000.00 yd

:mach 1.2
Mach 1.20 crossing
Observation
  distance: 1268.24 yd
  time    : 2.00 s
  velocity: 1333.93 ft/s
  energy  : 553.07 ft-lb
  drop    : +567.73 in
  windage : +88.25 in
  Mach    : 1.20
  flags   : none
Signs: drop + below/- above; windage + right/- left

:mach 1.0
Mach 1.00 crossing
Observation
  distance: 1531.95 yd
  time    : 2.65 s
  velocity: 1111.61 ft/s
  energy  : 384.08 ft-lb
  drop    : +968.74 in
  windage : +141.09 in
  Mach    : 1.00
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Transonic at 1,268 yards, subsonic at 1,532. Sanity-check it against `:at 1000 yd` in the same session, which reports `Mach : 1.43` — comfortably supersonic, as it must be.

On the unmodified 1,200-yard load the command refuses, and says why:

Listing 26.14: A trajectory that stays fast — verbatim, message wrapped

```
:mach 1.2
Error
  code: command_execution_error
  message: mach: assertion failed: mach_crossing: trajectory never falls to
           Mach 1.2; terminal Mach is 1.25598
```

The REPL is still running. That is the two-stage `try` in `dispatch_command` doing its job.

26.7 Testing it

Following the recipe from Chapter 25, the new command gets its own module: `test_mach.lat`, 182 lines, five tests, no engine.

The synthetic trajectory is built entirely from `domain.lat` constructors with Mach values chosen so the arithmetic is checkable by hand — 2.40 at 0 m, 1.80 at 400, 1.40 at 800, 1.00 at 1200, 0.80 at 1600:

Listing 26.15: `test_mach.lat`, the interpolation test

```
fn test_crossing_is_interpolated() {
    let trajectory = synthetic_trajectory()
    let transonic = analysis.mach_crossing(trajectory, 1.2)
    // Between the 800 m (Mach 1.40) and 1200 m (Mach 1.00) samples, half way.
    assert(abs(transonic.distance.si_value - 1000.0) < 0.000001,
    "expected 1000 m, got " + to_string(transonic.distance.si_value))
    assert(abs(transonic.mach - 1.2) < 0.000001)

    let exact = analysis.mach_crossing(trajectory, 1.0)
    assert(abs(exact.distance.si_value - 1200.0) < 0.000001)
    assert(exact.flags == [])
}
```

Mach 1.2 sits exactly halfway between 1.40 and 1.00, so the crossing must be at 1000 m. Mach 1.0 sits exactly on a sample, so the crossing must be 1200 m — and its flags must be empty, because `trajectory_at` does not manufacture flags for interpolated points.

The dispatch test uses an injected closure backend with a call counter, so the whole command path runs with no process at all:

Listing 26.16: `test_mach.lat`, the dispatch test

```
fn test_dispatch_renders_the_crossing() {
    flux calls = Ref::new(0)
    let backend = backends.engine_backend("mach-test", |experiment| {
        calls.set(calls.get() + 1)
        return synthetic_trajectory(experiment)
    }
    )
    let lab = sessions.new_session(reference.representative_experiment(), backend)

    let before = commands.execute_command(lab, ":mach 1.2")
    assert(commands.command_failed(before))
    assert(before.code == "command_state_error")

    let solved = commands.execute_command(lab, ":solve")
    assert(commands.command_succeeded(solved))
    assert(calls.get() == 1)

    let response = commands.execute_command(lab, ":mach 1.2")
    assert(commands.command_succeeded(response), "dispatch must succeed")
    assert(response.name == "mach")
    assert(!response.quit)
    assert_contains(response.output, "Mach 1.20 crossing")
    assert_contains(response.output, "distance: 1093.61 yd")
    assert(struct_name(response.value) == "Observation")
    assert(calls.get() == 1)

    let too_slow = commands.execute_command(lab, ":mach 0.5")
    assert(commands.command_failed(too_slow))
    assert(too_slow.code == "command_execution_error")

    let help = commands.execute_command(lab, ":help")
    assert_contains(help.output, ":mach <value>")
}
```

Four assertions in there are worth their weight. `calls.get() == 1` *after* the `:mach` proves the command reads the stored run rather than re-solving. `before.code == "command_state_error"` proves the no-run path works. `distance: 1093.61 yd` is 1000 m rendered under the default imperial profile — a check on the calculation and the unit policy at once. And the `:help` assertion catches the edit people forget.

Listing 26.17: Both backends (verbatim)

```
$ ./clat examples/ballistics_lab/test_mach.lat
ballistics_lab mach command: all tests passed
$ ./clat --regvm examples/ballistics_lab/test_mach.lat
ballistics_lab mach command: all tests passed
```

And the existing suite still passes with the new command in place — all fourteen hermetic modules, on both virtual machines:

Listing 26.18: Regression check in the fork — verbatim, condensed

```
=== --stack-vm ===
units: 19 tests passed
ballistics_lab domain: all tests passed
ballistics_lab process backend: all tests passed
...
ballistics_lab commands: all tests passed
ballistics_lab REPL support: all tests passed
=== --regvm ===
units: 19 tests passed
...
ballistics_lab REPL support: all tests passed
```

The last step is to add `test_mach.lat` to `BALLISTICS_LAB_TESTS` in the Makefile so it runs in the gate.

26.8 What happens when you skip a layer

Four edits in `commands.lat` is easy to get to three. Here is what each omission looks like, so you can recognize it.

Forget `_command_names()`. The parser produces a valid `ColonCommand`; the dispatcher refuses it. Verified live:

Listing 26.19: The name list not updated — verbatim, message wrapped

```
:mach 1.2
Error
  code: command_validation_error
  message: assertion failed: dispatch_command.command.name: unsupported
           command 'mach'
```

Forget the parser. You get `code: unknown_command, message: unknown colon command ':mach'` — the parse layer never even built a command.

Forget `_validate_command`. The dispatcher's args check is what refuses unknown keys; without a branch for your name, execution falls through to the path-shaped commands and fails on a missing `"path"`.

Forget the help text. Nothing fails. That is the point of the `:help` assertion in the test.

Two error codes, two audiences

`command_validation_error` means *your command was malformed*. `command_execution_error` means *your command was fine and running it failed*. The two-stage `try` in `dispatch_command` is what separates them, and getting the distinction right for a new command costs nothing — put argument checks in `_validate_command` and everything else in the dispatch branch.

26.9 Gotchas

Four things cost real time while building this chapter's example. All four are LATTICE facts rather than BALLISTICS LAB facts, and all four are cheap once you know them.

26.9.1 A module's own import alias may not resolve

A live backend divergence in Lattice v0.6.4

Under the **stack VM** — the default — a module's aliased import can fail to resolve unless the *importing* program happens to bind the same alias name. Under the register VM and the tree-walking interpreter it resolves correctly.
Three files reproduce it:

```
// leaf.lat
export fn greet() -> String {
    return "hello"
}

// mid.lat
import "leaf" as leaf

export fn call_leaf() -> String {
    return leaf.greet()
}

// top.lat
import "mid" as mid

fn main() {
    print(mid.call_leaf())
}
```

```
$ clat top.lat
vm error: undefined variable 'leaf' (did you mean 'lerp'?)
stack trace (most recent call last):
  [line 3] in <script>
  [line 4] in main()
  [line 4] in call_leaf()
$ clat --regvm top.lat
hello
$ clat --tree-walk top.lat
hello
```

Per the Ballistics Lab's own policy, a stack-VM/register-VM difference is a defect in the language runtime, not in the Ballistics Lab. It is invisible in normal use because `ballistics-repl.lat` imports every module under its canonical alias. It bites the moment you write a *new* program that imports, say, `render.lat` without also importing `session.lat` `as sessions` — which is exactly what a first extension script does. **Workaround: import the whole chain under the canonical aliases.** And note what `test_reference_import.lat` is for: it exists to catch precisely this class of leak.

26.9.2 An empty closure needs a space

`||` is logical OR. An empty parameter list is `| |`, with a space. `caught(|| { ... })` is a parse error:

```
error: 32:18: parse error: unexpected token '||' in expression
```

26.9.3 You cannot write a struct literal through a module alias

`analysis.AnalysisTable { ... }` fails with `vm error: unknown struct 'AnalysisTable'`, even though `analysis.lat` exports the type. Declare a structurally identical local struct instead — which works because the BALLISTICS LAB validates by `struct_name`, and which `test_render.lat` does deliberately, noting in a comment that selective struct-type imports are "not portable across both VM backends."

26.9.4 `export` is a keyword

`import "..." as export` does not compile. Neither does any other use of a reserved word as an alias — `anneal` and `test` are the other two that catch people. Name the alias `exporter`, `exports`, or anything else.

26.10 Extending the other layers

Three more extension points, and they do not cost what the layering would lead you to guess: the one that touches the most modules is not the deepest change, and the one that swaps the engine out from under the whole application is the smallest.

26.10.1 A new unit

Cheap and mechanical. `units.lat` is a table: add a constructor and a case to the converter. To add furlongs to `Distance` you would write `export fn furlongs(value: Number) -> Distance` next to `yd`, add a `"furlong"` case in `distance_as` *before* the trailing `_unsupported` call, and add the string to the parser's `_distance_units()` list if you want it available to `:at` and `:dope`.

The converter's tail is the piece that makes this safe:

Listing 26.20: The pattern every `*_as` ends with

```
    _unsupported("distance_as", unit, "m, km, yd, ft, in")
```

Miss the list update and the diagnostic tells users the unit does not exist — which is true, and better than silently accepting it:

Listing 26.21: Verbatim, message wrapped

```
bad unit : assertion failed: distance_as: unsupported unit 'furlong';
           supported: m, km, yd, ft, in
```

26.10.2 A new field on the wire

Most expensive, because a new engine parameter crosses six modules: `domain.lat` (the field, its validation, and any relational rule), `protocol_v1.lat` (the request encoding), `process_backend.lat` (the resolved-request validator), `resolved_request_v1.lat` (the same validator, at rest), `persistence.lat` (the save/load adapter), and `render.lat` (the summary line). Plus the golden request fixture, which will fail first and tell you the byte-level change was intentional.

That is a lot of edits, and the Ballistics Lab's design makes them all mechanical: each module has exactly one place the field belongs, and the test that catches an omission runs in seven seconds.

26.10.3 A new backend

Cheaper than either of the above, and the most interesting. A backend is a struct wrapping one closure:

Listing 26.22: `backend.lat`, the entire interface

```
export struct EngineBackend {
    name: fix String,
    solve_fn: fix Fn
}
```

`solve_with` guarantees the caller sees either a `Trajectory` or a `LaboratoryError`, converting a throw into a `backend_contract_error` and rejecting any other return type by name. So a cache, a recorder, a second solver, or a stub is:

Listing 26.23: A recording backend, in eight lines

```
fn recording_backend(inner: any, log: any) -> any {
    return backends.engine_backend("recording", |experiment| {
        let result = backends.solve_with(inner, experiment)
        log.set(log.get() + 1)
        return result
    }
    )
}
sessions.set_backend(lab, recording_backend(sessions.backend_of(lab), Ref::new(0)))
```

`session.lat` imports `backend.lat` and nothing about processes. `commands.lat` imports neither `backend.lat` nor `process_backend.lat`. Both are entirely unaware that a subprocess exists, which is precisely why swapping the transport is a one-line change.

26.11 Why not a native extension?

The dylib was designed, not built

`design/ballistics-extension.md` in the LATTICE tree lays out a different BALLISTICS LAB: a Rust `cdylib` installed at `~/.lattice/ext/ballistics.dylib`, statically linking `ballistics-engine` and binding its Rust API. Its status line still reads **"Proposed. Architecture decided; pre-implementation"**, there is no `extensions/ballistics/` directory in the tree, and it was **never built**. What ships — and what every recipe in this chapter extends — is a **subprocess** driven through `exec_argv`, speaking **solve-json v1**.

Section 20.7 weighs that proposal against the shipped architecture in full: the versioned wire contract, the process as a trust boundary, distribution, and the 3.4 ms a solve actually costs. Read it there. The question *this* chapter has to answer is narrower, and it will occur to you the first time the BALLISTICS LAB cannot do something you want: is a native extension how you get the missing capability? Three answers, none of them about architecture.

26.11.1 Your structs would not survive the trip

Seven value kinds cross LATTICE's extension boundary — `Int`, `Float`, `Bool`, `String`, `Array`, `Map`, and `Nil`. Structs and closures do not, and the boundary force-copies everything that does cross, in both directions, on every call: a `dlopen`'d library cannot participate in the refcounting behind crystal-by-reference, so a shared handle must never escape into one.

Read that against what we built earlier in this chapter. The first thing `mach_crossing` does is hand its argument to `_validate_trajectory`; the last thing it does is return an `Observation`, which the dispatcher passes to the renderer and hands back to the caller as a *value*, not a string. Behind a dylib all of it is untyped Maps at the seam: nothing to validate on the way in, nothing to name on the way out, and `struct_name` — the Ballistics Lab's whole decoupling technique — with no struct to report on. Eighty-eight lines were enough for `:mach` because types survived every layer.

26.11.2 You would lose the fake engine

The Ballistics Lab tests its whole engine boundary against a fake engine written in Lattice: `fixtures/fake_solve_json_engine.lat`, 295 lines that impersonate `ballistics-engine` across the same seam. Chapter 25 drives roughly forty-five hostile scenarios through it — truncated JSON, duplicate object keys, non-finite numbers, exit-code disagreement — in a suite that finishes in seven seconds with no compiler, no linker, and no engine on the machine. That is possible only because the seam is argv plus a JSON document on stdin. The equivalent for a dylib is hostile Rust, rebuilt per case. Extending a Ballistics Lab you can fake is a different activity from extending one you cannot.

26.11.3 The capability you want is a verb, not a library

The engine on this machine has thirty-six subcommands. The Ballistics Lab's backend seam uses exactly one of them — `process_backend.lat:856` passes the single argument `"solve-json"` — so `zero`, `monte-carlo`, `estimate-bc`, `true-velocity`, `mpbr` and the rest are reachable from your shell, as Part IV showed, and not from a colon command. None of them needs an extension to reach. They need a second verb.

That costs more than `:mach` did, and the reason is worth seeing rather than asserting. solve-json v1 is what lets the shipped decoder be strict; the other subcommands do not speak it:

Listing 26.24: `zero` emits JSON, but not an envelope — verbatim

```
$ ballistics zero -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --target-distance 100 -o json
{
  "max_ordinate": 0.05664098769060267,
  "point_blank_range": 178.70138513201405,
  "sight_adjustment_moa": 4.301379931707232,
  "units": "imperial",
  "zero_angle_degrees": 0.07168966552845386,
  "zero_angle_moa": 4.301379931707232,
  "zero_angle_mrad": 1.2512207031250004
}
```

A bare result object: no `schema_version` to check, no `status` to branch on, no `resolved_request` to prove what the engine thought it was asked, no `engine_version`, and no unit vocabulary beyond the string `"imperial"`. So a second verb is four pieces of work, in the order the BALLISTICS LAB would want them:

1. **A request encoder** beside `protocol_v1.lat`. For a CLI verb it builds an argv array rather than a JSON document, which makes it the module that decides what is allowed to become an argument.
2. **A decoder that fails closed** on a document with no envelope around it — its own key-set check, its own finiteness checks, its own error code (Chapter 22).
3. **A second backend**, not a second branch in the existing one. `EngineBackend.solve_fn` promises a `Trajectory` or a `LaboratoryError`, and a zero angle is neither.
4. **A command**, by the four-layer recipe above.

That is the honest price of a strict seam, paid once per verb — and a dylib does not lower it. It replaces a decoder you can test with a string literal by a type-flattening boundary you cannot.

The one idea worth stealing from the unbuilt design

The design document's §7 proposed using the phase system to model draft versus verified dope: a card is `flux` while you are truing it, and `freeze()` makes it a crystal that a field-mode command will accept. That idea *did* ship — as `verified_artifacts.lat`, whose `phase_of(value) != "crystal"` check at line 374 is exactly the proposed contract (Chapter 24). The architecture changed completely; the best idea in it survived intact.

26.12 A checklist

For a new colon command, in order:

1. Write the calculation in `analysis.lat` (or wherever it belongs), validating inputs with the module's existing helpers and naming paths in every message.
2. Add a `_parse_*` function and a dispatch line to `command_parser.lat`. Return errors as values.
3. Add a renderer to `render.lat` that goes through `_profile` and `_fixed`.
4. Make four edits in `commands.lat`: name list, help text, `_validate_command`, `_dispatch_checked`.
5. Write `test_<name>.lat` with a synthetic trajectory and an injected closure backend. Assert on the help text.
6. Run it on both virtual machines.
7. Add the test to `BALLISTICS_LAB_TESTS` in the Makefile.
8. Run `make test-ballistics-lab` and confirm nothing else moved.

Eighty-eight lines, four files, seven seconds to know you did not break anything. That is the whole argument for the architecture in this book: not that a source-level laboratory is faster than a native one, but that it is small enough to change, and testable enough that changing it is not frightening.

Appendix A

The Lab Command Reference

This appendix is the page to photocopy and tape inside the lid of the range box. It lists every colon command the Ballistics Lab accepts, its exact arity, its argument ranges, what it prints, and how it fails. Every cell was checked twice: once against the parser source in `command_parser.lat` and the dispatcher in `commands.lat`, and once by actually typing the command into a running Lab and reading what came back.

Conventions used in this appendix

Transcripts were captured by piping a file of commands into the launcher:

```
sh scripts/ballistics-lab.sh < commands.txt
```

Under a pipe, GNU readline suppresses the `ballistics>` prompt, so the transcripts below contain no prompts. The three-line startup banner is elided from every transcript in this appendix; it is shown once, in full, in Section A.1. Unless a section says otherwise, the session is the one you get on a bare launch: the shipped reference experiment, a 175 gr G7 match load at 1,000 yards.

A.1 The card

Here is the whole surface, printed by the Lab itself. This is the complete output of a session whose entire input was `:help` followed by `:quit` — banner included, because you only see it once per launch.

Listing A.1: The complete colon surface, as the Lab prints it

```
Ballistics laboratory --- Lattice v0.6.4
Engine: /Users/alexjokela/projects/ballistics-engine/target/release/ballistics
Type :help for commands or enter Lattice. Ctrl-D to exit.

Commands:
:help
:show
:solve
:at <distance> <unit>
:dope <start> <stop> <interval> <unit>
:wind-card <start> <stop> <interval> <unit>
:compare
:save "<path>"
:load "<path>"
:reset
:quit
Quit requested.
```

One character we had to change

The first banner line contains a real em dash between `laboratory` and `Lattice`. Typesetting a raw em dash inside a verbatim listing is not possible in this book's font setup, so the listing above renders it as three hyphens. That is the only character in any transcript in this appendix that is not byte-for-byte what the Lab printed. Every number is untouched.

Table A.1 is the same information with the arities and ranges filled in. Read it as: *name*, how many arguments the parser demands (exactly, not at least), the argument types, and what the command needs to be true about the session before it will do anything.

Three facts about that table earn a sentence each.

Arity is exact. There is no such thing as an optional colon argument. `:help extra` is not tolerated with a shrug; it is a hard `command_arity` error. The parser compares `len(arguments)` against a single expected count and refuses anything else (`command_parser.lat:286`).

The distance unit is one of five. `m`, `km`, `yd`, `ft`, `in` — that list is a literal in `command_parser.lat:54–56` and nothing else is accepted.

Nothing on the card changes the load. No colon command can alter the projectile, rifle, atmosphere, wind, shot, or solver. The card queries and persists; setting up a rifle is done by typing Lattice at the same prompt. That division is deliberate and is the subject of Section A.5.

Table A.1: Every colon command, its arity, and its preconditions

Command	Args	Argument types	Needs
`:help`	0	—	nothing
`:show`	0	—	nothing
`:solve`	0	—	a reachable engine
`:at`	2	number $\geq$ 0, distance unit	a current run
`:dope`	4	3 numbers, distance unit	a current run
`:wind-card`	4	3 numbers, distance unit	a current run
`:compare`	0	—	a current *and* a previous run
`:save`	1	path string	a writable path
`:load`	1	path string	a readable experiment document
`:reset`	0	—	nothing
`:quit`	0	—	nothing

A.2 How a line becomes a command

The REPL routes a line to the command parser only when *two* things hold: the buffer is empty (you are at a fresh prompt, not continuing a multi-line Lattice expression), and the line's first non-space character is a colon (`ballistics-repl.lat:160`). Everything else is Lattice source.

That second condition is checked after `trim_start()`, so leading whitespace is fine:

Listing A.2: Leading whitespace before the colon is allowed

```
   :help
```

runs the command. And the first condition means a colon inside a multi-line Lattice value is inert:

Listing A.3: A colon line inside an incomplete expression is source, not a command

```
let note = "
:help
"
len(note)
```

Listing A.4: … and the Lab prints the string length, not the help text

```
7
```

Seven bytes: a newline, the five characters `:help`, and a newline. The help text never appeared, because the buffer was not empty when that line arrived.

A.2.1 Tokens, quoting, and escapes

After the leading colon, the parser splits the rest of the line into tokens. The first token is the command name; the remainder are the arguments.

- **Separators** are the space and the tab. Both work, and runs of them collapse.
- **Quotes** are the double quote and the single quote, and they are symmetric — either may quote a token. A quoted token may contain spaces.
- **Escapes inside a quoted token** are exactly two: the matching quote and the backslash. *Every other backslash is preserved byte for byte* (`command_parser.lat:197–198`). That is what lets a Windows path be pasted in as an inert argument without doubling anything.
- **A quoted token must end the token.** Text glued onto a closing quote is a syntax error, and so is a quote glued onto unquoted text.
- **The command name must not be quoted**, and must not be empty.

All of that is verifiable in one session. Here are the failures, each on its own input line:

Listing A.5: Quoting failures, with 1-based line:column locations

```
Error
  code: command_syntax
  message: unmatched quote
  location: 1:7
Error
  code: command_syntax
  message: quoted argument must be followed by whitespace or end of input
  location: 1:132
```

The `location` field is the parser's column tracking: the `1` is always the line (a command is always one line), and the column is 1-based into the raw input, computed by carrying the `trim_start()` delta forward (`command_parser.lat:106–113`).

Command names are case-sensitive. `:HELP` is not `:help`:

Listing A.6: The command table is matched exactly

```
Error
  code: unknown_command
  message: unknown colon command ':HELP'
  path: command
```

A.2.2 Bounds

The parser has two hard ceilings. Both are enforced before any evaluation, and both are reported as `resource_limit`.

Table A.2: Parser bounds, both verified at the boundary

Bound	Limit	Behaviour at the edge
Line length	4096 bytes	4096 accepted, 4097 rejected
Argument count	16	16 accepted, 17 rejected

Both edges were probed, not assumed. A `:dope` line padded to exactly 4096 bytes reaches the argument checks and fails on the argument (proving the length check passed); one more byte never gets that far:

Listing A.7: The 4096-byte line limit is inclusive

```
Error
  code: command_argument
  message: expected a numeric argument, got 'yyyyyyyyyyyyyyyyyyyyyyyyyyyy...
  path: arguments[0]
Error
  code: resource_limit
  message: command exceeds the 4096-byte input limit
  path: command
```

Listing A.8: Sixteen arguments pass the bound and fail on arity; seventeen do not pass the bound

```
Error
  code: resource_limit
  message: command exceeds the 16-argument limit
  path: arguments
Error
  code: command_arity
  message: :at expects 2 argument(s), got 16
  path: arguments
```

The internal check is `len(tokens) > 17` because token zero is the command name — sixteen arguments plus the name.

A.2.3 Numbers

A numeric argument goes through `parse_float` inside a `try`/`catch`; a parse failure and a non-numeric result both become `command_argument`. A parsed value that is NaN or infinite is rejected separately with *numeric argument must be finite*. Quoted numbers are accepted — `:at "100" yd` parses — because quoting affects tokenization only, never interpretation.

A.3 Command by command

Each entry below gives the syntax, the arity, the argument rules, what the command actually calls, and what it prints. The examples run against the shipped reference experiment unless stated.

A.3.1 `:help`

Syntax `:help`
Arity 0
Calls `_help_text()` (`commands.lat:30`)

Prints the literal command list shown in Section A.1. It takes no arguments and does not tolerate one:

Listing A.9: `:help` is zero-arity, strictly

```
Error
  code: command_arity
  message: :help expects 0 argument(s), got 1
  path: arguments
```

The text is a hard-coded string in `commands.lat`, not generated from the dispatch table. If you extend the Lab with a new command, you must add it in both places.

A.3.2 `:show`

Syntax `:show`
Arity 0
Calls `render.render_session_summary(lab)`
Returns the current `Experiment` as the response value

`:show` is the command you type most. It prints a session header followed by a complete expansion of the active experiment — every field, in your display units, with unset optional fields printed as `default`.

Listing A.10: : show on a fresh session, banner elided

```
Ballistics laboratory session
  active experiment: 175 gr match at 1,000 yards
  backend          : process-json-v1
  display          : yd, ft/s, ft-lb, mil (2 decimals)
  history          : 0/50
  current run      : none
  previous run     : none

Experiment: 175 gr match at 1,000 yards
  projectile           : 175 gr match
  mass                 : 175.00 gr
  diameter             : 0.31 in
  length               : 1.24 in
  drag model           : G7
  ballistic coefficient: 0.24
  rifle                : 24-inch match rifle
  sight height         : 1.50 in
  muzzle height        : 0.00 in
  twist rate           : 8.00 in
  twist direction      : right
  altitude             : 273.40 yd
  temperature          : 59.00 F
  pressure             : 29.92 inHg
  humidity             : 50.00 %
  latitude             : +45.00 deg
  muzzle velocity      : 2700.00 ft/s
  maximum range        : 1000.00 yd
  zero distance        : 100.00 yd
  muzzle angle         : default
  aim azimuth          : default
  shot azimuth         : default
  shooting angle       : default
  cant angle           : default
  target height        : default
  ground threshold     : default
  solver               : rk45
  time step            : default
  sampling interval    : 10.00 yd
  Magnus               : default
  Coriolis             : on
  enhanced spin drift  : default
  wind[0]              : 10.00 mph from 90.00 deg
```

Two lines in that block reward attention.

`history : 0/50` is *runs stored* over *history limit*. The limit defaults to 50 and is set with `sessions.set_history_limit`, not by any colon command.

`altitude : 273.40 yd` is 250 metres rendered in the display distance unit. The Lab applies one distance unit to every length, including vertical ones, so a station altitude reads in yards. It is arithmetically correct and visually surprising; if it bothers you, switch to metric display preferences.

A.3.3 `:solve`

Syntax `:solve`
Arity 0
Calls `sessions.solve_session(lab)`
Returns a `RunRecord`, or a `LaboratoryError`

`:solve` is the only command that starts a child process. It serialises the current experiment into a solve-json v1 request, spawns the engine with a fixed argv, feeds the request on stdin, validates the entire response, and — only if everything checks out — publishes a new session state containing the run.

Listing A.11: `:solve` on the reference experiment

```
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
  schema           : 1
  verified         : no
  termination      : max_range
  actual range     : 1000.00 yd
  maximum height   : +1.62 in
  time of flight   : 1.71 s
  terminal velocity: 1145.96 ft/s
  terminal energy  : 510.20 ft-lb
  stability factor : 3.80
  spin drift       : not reported
Assumptions
  - default_applied: Aim azimuth defaulted to 0 rad. [$.shot.aim_azimuth_rad]
  - default_applied: Shot azimuth defaulted to 0 rad (north). [$.shot.shot_azimuth_rad]
  - default_applied: Shooting angle defaulted to 0 rad. [$.shot.shooting_angle_rad]
  - default_applied: Cant angle defaulted to 0 rad. [$.shot.cant_angle_rad]
  - default_applied: Ground threshold defaulted to -100 m. [$.shot.ground_threshold_m]
  - default_applied: Constant vertical wind speed defaulted to 0 m/s. [$.wind.vertical_speed_mps]
  - default_applied: Solver time step defaulted to 0.001 s. [$.solver.time_step_s]
  - default_applied: Magnus effect defaulted to disabled. [$.effects.magnus]
  - default_applied: Enhanced spin drift defaulted to disabled. [$.effects.enhanced_spin_drift]
Warnings
  none
```

Nine assumptions is the healthy state

Every `default_applied` line is the engine telling you which field it filled in for you. A fully specified experiment still produces nine of them, because the Lab's domain model has optional fields the engine has defaults for. `verified : no` and `spin drift : not reported` are likewise normal: verification is an explicit later step, and `spin_drift_m` only appears when enhanced spin drift is enabled.

The assumption that is not there

One line you might expect is absent: `Target height defaulted to 0 m above the local ground datum`. `target_height_m` is the height above ground that the engine solves the zero *to*, and the engine's own default is ground level. The Lab fills the field in before the request leaves — `protocol_v1.lat` sets it to muzzle height plus sight height, the height of your line of sight — so the engine never has to default it, and it never reports it. That is why a conventional zero comes back as a conventional zero: at 100 yards the reference load reads `drop +0.00 in`. A missing assumption is doing real work here, which is a good argument for reading the list rather than skipping it.

`:solve` never leaves the session half-modified. A backend failure returns a `LaboratoryError` across the seam, is rendered, and the prior state is untouched — so the previous run, if there was one, still answers `:at`.

Termination is the field to read. `max_range` means you got the distance you asked for. Anything else means the solve stopped early, and the `actual range` line tells you where:

Listing A.12: A 5,000 yd request that hits the ground at 2,094.02 yd

```
Run: 175 gr match at 1,000 yards
  backend          : process-json-v1
  engine           : 0.37.0
  schema           : 1
  verified         : no
  termination      : ground_threshold
  actual range     : 2094.02 yd
  maximum height   : +1.62 in
  time of flight   : 5.36 s
  terminal velocity: 766.51 ft/s
```

That run was produced by evaluating `sessions.replace_shot(lab, shot(fps(2700), yd(5000), yd(100)))` and then `:solve`. It is not an error. The engine solved correctly; the bullet landed first.

A.3.4 :at

Syntax `:at <distance> <unit>`
Arity 2
Rules distance ≥ 0; unit in `m km yd ft in`
Calls `analysis.trajectory_at(current.result, distance)`

`:at` interpolates the current run at one distance. It is not restricted to sample points — ask for any distance inside the computed trajectory and you get a linear interpolation between the two bracketing samples; ask for an exact sample and you get that sample, with your requested display unit preserved.

Listing A.13: `:at 500 yd` on the reference run

```
Observation
  distance: 500.00 yd
  time    : 0.67 s
  velocity: 1841.19 ft/s
  energy  : 1317.05 ft-lb
  drop    : +58.83 in
  windage : -20.13 in
  Mach    : 1.65
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Read the legend literally

The two halves use opposite directions, and the legend says so. *Drop* is positive **below** the line of sight — it is a depth, so a bigger positive number means the bullet is lower. *Windage* is positive **right**. Section D.6 in Chapter D works through both conventions on a real run, including the one case where a come-up goes negative.

The unit argument converts *into* the display unit, so the echoed distance is not necessarily what you typed. All five units on one run:

Listing A.14: `:at` in five units, on one solve, display unit `yd`

```
Observation
  distance: 0.00 yd
  time    : 0.00 s
  velocity: 2700.00 ft/s
  energy  : 2832.25 ft-lb
  drop    : +1.50 in
  windage : +0.00 in
  Mach    : 2.42
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 109.36 yd
  time    : 0.13 s
  velocity: 2496.27 ft/s
  energy  : 2420.96 ft-lb
  drop    : +0.13 in
  windage : -0.83 in
  Mach    : 2.23
  flags   : none
Signs: drop + below/- above; windage + right/- left
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.96 ft/s
  energy  : 510.20 ft-lb
  drop    : +392.91 in
  windage : -102.04 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
```

Those three blocks came from `:at 0 yd`, `:at 100 m`, and `:at 3000 ft` respectively. One hundred metres echoes as `109.36 yd`; three thousand feet is exactly one thousand yards, so the third block is the terminal sample and carries the `terminal` flag. In the same session `:at 0.1 km` reproduced the `109.36 yd` block byte for byte, and `:at 3600 in` gave the 100 yd row.

The muzzle row is worth a second look. At `0.00 yd` the drop is `+1.50 in`, which is exactly the rifle's sight height, because that is where the bore sits relative to the line of sight before the bullet has gone anywhere. A little past the 100 yd zero, at `109.36 yd`, the bullet has crossed and is back to `+0.13 in` below the sight line.

Out of range is an error, not a clamp.

Listing A.15: Asking beyond the computed trajectory

```
Error
  code: command_execution_error
  message: at: assertion failed: trajectory_at.distance: 1828.8 m is outside computed trajectory
     [0, 914.4] m
```

The message reports SI metres and names the actual interval, which is the fastest way to discover that your solve terminated early.

A.3.5 :dope

Syntax `:dope <start> <stop> <interval> <unit>`
Arity 4
Rules `start > 0`, `stop > start`, `interval > 0`
Calls `analysis.sample` then `analysis.dope`

A dope card: distance, drop, come-up, velocity, energy, time, Mach. The four arguments describe a grid in the unit you name; the dispatcher samples the trajectory on that grid, converts every sampled distance into the display unit, and asks the analysis layer for a table in exactly the units the renderer will later demand.

Listing A.16: `:dope 100 500 100 yd`, provenance block elided

```
DOPE: 175 gr match at 1,000 yards
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2513.36 |        2454.23 |     0.12 | 2.25
       200.00 |     +4.01 |         +0.56 |         2334.06 |        2116.56 |     0.24 | 2.09
       300.00 |    +14.41 |         +1.33 |         2162.46 |        1816.78 |     0.37 | 1.93
       400.00 |    +32.26 |         +2.24 |         1998.49 |        1551.71 |     0.52 | 1.79
       500.00 |    +58.83 |         +3.27 |         1841.19 |        1317.05 |     0.67 | 1.65
Signs: drop + below/- above; come-up + correction up/- correction down
```

Every table also prints a `Provenance` block: engine version, schema version, solver method, verification status, termination, actual range, and the full assumption and warning lists. It is elided above for width but it is always there, and it is the reason a Lab table can be pasted into a notebook without a sidecar note about which engine produced it.

Two rows to sanity-check before you dial

The zero row and the trend. At the zero distance — 100 yd here — both drop and come-up read `+0.00`, because that is what a zero *is*: the bullet is on the line of sight and the dial is where you left it. Past the zero both columns grow positive and keep growing, and positive come-up means *raise the sight*. If either of those two things is not true of a table you are about to shoot, stop and find out why before you dial.

The stop is clipped to the trajectory; the start is not. Asking for a stop past the end of the solve silently truncates the grid and appends the true terminal row:

Listing A.17: `:dope 900 2000 200` yd on a 1,000 yd solve

```
DOPE: 175 gr match at 1,000 yards
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       900.00 |   +289.87 |         +8.95 |         1270.53 |         627.15 |     1.46 | 1.14
      1000.00 |   +392.91 |        +10.91 |         1145.96 |         510.20 |     1.71 | 1.03
Signs: drop + below/- above; come-up + correction up/- correction down
```

while a start past the end is refused outright:

Listing A.18: `:dope 1500 2000 100` yd on the same run

```
Error
  code: command_execution_error
  message: dope: assertion failed: sample.start: must be inside the computed trajectory
```

The argument checks run before any of that. All three are strict inequalities:

Listing A.19: Argument validation for `:dope`

```
Error
  code: command_argument
  message: :dope stop must be greater than start
  path: arguments[1]
Error
  code: command_argument
  message: :dope interval must be greater than zero
  path: arguments[2]
```

A.3.6 `:wind-card`

Syntax `:wind-card <start> <stop> <interval> <unit>`
Arity 4
Rules identical to `:dope`
Calls `analysis.wind_card([current.result], ...)`

The same grid, a different table: series index, trajectory name, wind speed, wind direction, distance, windage, and wind hold. Only the current run is passed in, so the series index is always zero, but the table's shape supports several series — the shape is shared with the multi-trajectory wind cards the analysis module can build directly.

Listing A.20: `:wind-card 200 1000 200 yd` on the reference run

```
Wind card
Series | Trajectory                 | Wind (mph) | From (deg) | Distance (yd) | Windage (in) | Wind hold (
     mil)
-------+----------------------------+------------+------------+---------------+--------------+----------------

  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |        200.00 |        -2.87 |
     +0.40
  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |        400.00 |       -12.39 |
     +0.86
  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |        600.00 |       -30.23 |
     +1.40
  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |        800.00 |       -58.91 |
     +2.05
  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |       1000.00 |      -102.04 |
     +2.83
Signs: windage + right/- left; wind hold + correction right/- correction left
```

Read that as physics: a 10 mph wind *from* 90 degrees is a wind from the shooter's right, so the bullet is pushed left — negative windage — and the hold is to the right, positive.

That sign flip between the drift and the hold is the honest difference between this table and the dope table, and it is worth understanding rather than memorising. Windage is positive-*right*, so correcting it means holding the other way: the hold negates the drift. Drop is positive-*below*, so correcting *it* means going the same way: the come-up does not negate anything. Two columns, two conventions, and the legend on each table names its own.

The `Series` column rendering as `0.00` rather than `0` is cosmetic: the index is carried as a number through the generic table machinery and formatted with the display precision.

A.3.7 `:compare`

Syntax `:compare`
Arity 0
Needs a current *and* a previous run
Calls `analysis.compare_trajectories([previous, current], ...)`

`:compare` aligns the last two runs on a shared distance grid and prints both, with deltas. Because it takes no arguments, the only way to use it is to solve, change something, and solve again:

Listing A.21: The `:compare` idiom

```
:solve
sessions.replace_winds(lab, [wind(mph(20), deg(90))])
:solve
:compare
```

The previous run is deliberately series **0** — the baseline — so every delta reads *current minus previous* (`commands.lat:318–319`). The first and last rows of that session:

Listing A.22: `:compare`, first two and last two rows (wide; wraps)

```
  0.00 | 175 gr match at 1,000 yards |          0.00 |     0.00 |         2700.00 |
     2832.25 |     +1.50 |        +0.00 | 2.42 |          +0.00 |                +0.00 |
             +0.00 |           +0.00 |              +0.00 |      +0.00
  1.00 | 175 gr match at 1,000 yards |          0.00 |     0.00 |         2700.00 |
     2832.25 |     +1.50 |        +0.00 | 2.42 |          +0.00 |                +0.00 |
             +0.00 |           +0.00 |              +0.00 |      +0.00
  0.00 | 175 gr match at 1,000 yards |       1000.00 |     1.71 |         1145.96 |
     510.20 |   +392.91 |      -102.04 | 1.03 |          +0.00 |                +0.00 |
            +0.00 |           +0.00 |              +0.00 |      +0.00
  1.00 | 175 gr match at 1,000 yards |       1000.00 |     1.71 |         1146.02 |
     510.26 |   +392.92 |      -206.82 | 1.03 |          +0.00 |                +0.06 |
            +0.06 |           +0.01 |            -104.78 |      +0.00
```

The physics checks out: doubling the beam wind from 10 to 20 mph moves 1,000 yd windage from −102.04 in to −206.82 in, a delta of −104.78 in, while drop changes by 0.01 in. Wind moves the bullet sideways and leaves the drop alone.

`:compare` has no range argument

It emits the *entire* sampled trajectory, two rows per distance. On the reference experiment — 1,000 yd sampled every 10 yd — that is roughly 200 table rows and a 300-line session. If you want a bounded comparison, call `analysis.compare_trajectories` directly with a distance list.

Both series carry the same `Trajectory` label when only a component changed, because `replace_*` carries the experiment name forward. If you are comparing two loads, rename the experiment between solves.

A.3.8 `:save`

Syntax `:save "<path>"`
Arity 1
Rules path must be a non-empty token
Calls `persistence.save_experiment(path, experiment_of(lab))`

Writes the active experiment — and nothing else — to a versioned JSON document. No run, no history, no session state, no closures, no Lattice source.

Listing A.23: Three saves: quoted, unquoted, and a path containing a space

```
Saved experiment to /private/tmp/.../app/ref.json
Saved experiment to /private/tmp/.../app/ref2.json
Saved experiment to /private/tmp/.../app/with space.json
```

The quotes in the help text are advice, not syntax. An unquoted path with no spaces or quote characters tokenises as one argument and works. A path containing a space *must* be quoted, or you get an arity error, because the space splits it into two tokens:

Listing A.24: An unquoted path containing a space

```
Error
  code: command_arity
  message: :save expects 1 argument(s), got 2
  path: arguments
```

An empty quoted path is caught by the parser before the dispatcher sees it:

Listing A.25: `:save ""`

```
Error
  code: command_argument
  message: :save path must not be empty
  path: arguments[0]
```

Write failures surface as `command_execution_error` with the path quoted back:

Listing A.26: An unwritable destination

```
Error
  code: command_execution_error
  message: save: assertion failed: save_experiment: unable to write '/proc/nope/x.json'
```

The document is validated and serialised in full *before* the file is touched, so a validation failure never leaves a partial file behind.

A.3.9 `:load`

Syntax `:load "<path>"`
Arity 1
Calls `persistence.load_experiment` then `sessions.replace_experiment`

Reads a saved experiment document, reconstructs every value through the domain constructors (so a hand-edited file gets the same validation a programmatically-built one does), renders the summary, and only then performs the single session mutation. On success it prints the loaded experiment in the same format as the second half of `:show`.

`:load` does not clear the run state

After a `:load`, `:show` reports the loaded experiment but `:at`, `:dope`, `:wind-card`, and `:compare` still answer from the *previous* trajectory. The safe idiom is `:load`, then `:reset`, then `:solve`. This is demonstrated with numbers in Section D.7.

Three failure modes, all reported as `command_execution_error`:

Listing A.27: Missing file, wrong document, and unparseable file

```
Error
  code: command_execution_error
  message: load: assertion failed: experiment JSON: expected String, got Nil
Error
  code: command_execution_error
  message: load: assertion failed: $: missing required field 'schema_version'
Error
  code: command_execution_error
  message: load: json_parse error at position 0: unexpected character
```

The first of those is a rough edge worth knowing: a *missing file* reports an internal type assertion rather than a file-not-found message. The `read_file` call returns `Nil`, and the string check downstream is what fails.

A.3.10 `:reset`

Syntax `:reset`
Arity 0
Calls `sessions.reset_session(lab)`

Listing A.28: The entire response

```
Session run state reset.
```

The wording is exact. Table A.3 is what it does and does not touch.

Table A.3: What `:reset` clears

State	Cleared?	After reset
History	yes	`0/limit`
Current run	yes	none
Previous run	yes	none
Experiment	**no**	unchanged, including every edit
Experiment name	**no**	unchanged
Backend	**no**	unchanged
Display preferences	**no**	unchanged
History limit	**no**	unchanged

If you want a clean rifle as well as a clean notebook, reload a saved experiment or relaunch the Lab.

A.3.11 `:quit`

Syntax `:quit`
Arity 0
Effect sets `quit: true` on the response; `main()` breaks

Listing A.29: The entire response, printed before the process exits 0

```
Quit requested.
```

Ctrl-D at a fresh prompt does the same thing without the message: the REPL's EOF predicate accepts both `Nil` and `Unit` from `input()`, prints a blank line, and breaks.

A.4 The error block

Every failure — parse, validation, execution, engine — is rendered as the same block. It has one required shape and four optional lines, emitted in a fixed order:

Listing A.30: The maximal error block

```
Error
  code: <code>
  message: <message>
  path: <json path or argument path>       (optional)
  location: <line>:<column>                (optional)
  stderr: <engine diagnostics>             (optional)
  exit code: <n>                           (optional)
```

`path` and `location` are mutually exclusive by construction: the domain constructor refuses to build an error carrying both, which encodes the protocol rule that a diagnostic points either at a field or at a source position, never both.

Table A.4 lists every code the colon surface can produce, with the layer that raises it. All fourteen were observed live.

Alongside those, the engine's own error codes pass through the seam unchanged and are rendered in the same block. There are ten of them — `invalid_json`, `unsupported_schema_version`, `unknown_field`, `missing_field`, `invalid_value`, `conflicting_fields`, `resource_limit`, `solve_failed`, `io_error`, `internal_error` — and they are catalogued in Chapter C.

Table A.4: Error codes reachable from the colon surface

Code	Raised by	Meaning
`command_syntax`	parser	malformed line; carries `location`
`unknown_command`	parser	no such colon command
`command_arity`	parser	wrong number of arguments
`command_argument`	parser	bad value, unit, or range
`resource_limit`	parser	4096-byte or 16-argument bound
`command_validation_error`	dispatcher	a hand-built command failed re-validation
`command_state_error`	dispatcher	no current or previous run
`command_execution_error`	dispatcher	I/O or analysis threw during execution
`evaluation_error`	REPL	a Lattice line threw
`repl_command_error`	REPL	the dispatcher itself threw or returned junk
`invalid_engine_response`	backend	the engine broke the protocol
`incompatible_engine_version`	backend	version pin mismatch
`process_error`	backend	spawn, timeout, or output-size failure
`backend_contract_error`	backend seam	backend returned a non-domain value

The REPL never dies

A session fed sixteen deliberately broken commands in a row printed sixteen error blocks and still exited 0. Every error path in `ballistics-repl.lat` renders and continues; the only line that ends the loop is `:quit` or EOF. You can hammer on the command line without losing your session.

A.5 What the card cannot do

There is no `:set`, no `:velocity`, no `:wind`. The colon grammar is deliberately closed: eleven commands, four shapes, no evaluation. Everything that *changes* the experiment is ordinary Lattice typed at the same prompt, against the top-level `lab` binding.

The constructors are imported unqualified into the REPL environment precisely so this reads well. Table A.5 is the working set.

A complete rifle setup is six lines:

Table A.5: Session mutators — the other half of the command surface

Call	Replaces
`sessions.replace_projectile(lab, v)`	the projectile
`sessions.replace_rifle(lab, v)`	the rifle
`sessions.replace_atmosphere(lab, v)`	the atmosphere
`sessions.replace_winds(lab, [v, ...])`	the wind list
`sessions.replace_shot(lab, v)`	the shot
`sessions.replace_solver(lab, v)`	the solver config
`sessions.replace_experiment(lab, v)`	everything, including the name
`sessions.set_backend(lab, v)`	the engine backend
`sessions.set_display_preferences(lab, v)`	units and precision
`sessions.set_history_limit(lab, n)`	the history bound

Listing A.31: Setting up a load at the Lab prompt

```
sessions.replace_projectile(lab, projectile("140 gr ELD-M", gr(140), inches(0.264), 0.315, "G7",
    inches(1.34)))
sessions.replace_rifle(lab, rifle("Tikka T3x CTR 24 in", inches(1.8), m(0), inches(8), "right"))
sessions.replace_atmosphere(lab, atmosphere(m(1200), celsius(12), hpa(880), 0.35, deg(44)))
sessions.replace_winds(lab, [wind(mph(8), deg(270))])
sessions.replace_shot(lab, shot(fps(2710), yd(1200), yd(100)))
sessions.replace_solver(lab, solver_config("rk45", nil, yd(25), nil, true, nil))
```

`replace_*` keeps the old name

Those six lines change every component of the experiment and leave the *name* alone: the session will still call it `175 gr match at 1,000 yards`. Renaming requires a whole-experiment replacement:

```
let cur = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("140 gr ELD-M at 1,200 yd", cur.projectile, cur.
    rifle, cur.atmosphere, cur.winds, cur.shot, cur.solver))
```

A notebook built on `replace_*` without that step mislabels every run it produces.

Validation failures on these lines come back as `evaluation_error`, because the domain constructors throw rather than return:

Listing A.32: Domain validation at the prompt

```
Error
  code: evaluation_error
  message: assertion failed: projectile.drag_model: unsupported value 'G9'; expected one of [G1,
     G6, G7, G8]
Error
  code: evaluation_error
  message: assertion failed: shot: zero_distance and muzzle_angle cannot both be supplied
```

The unit constructors available unqualified at the prompt are `m km yd ft inches`, `mps fps`, `kg grams gr`, `rad deg mil moa`, `kelvin celsius fahrenheit`, `pa hpa inhg`, `joules foot_pounds`, and `wind_mps mph kph`. The domain constructors are `projectile rifle atmosphere standard_atmosphere wind calm_wind shot`, `solver_config experiment observation assumption warning`, `trajectory_summary trajectory laboratory_error`.

A.6 Reproducing this appendix

Every transcript above came from a file of input lines piped into the launcher. To regenerate any of them:

Listing A.33: The capture command

```
sh /Users/alexjokela/projects/lattice/scripts/ballistics-lab.sh \
    --engine /Users/alexjokela/projects/ballistics-engine/target/release/ballistics \
    < input.txt > out.txt 2> err.txt
```

The engine is named rather than discovered, because the capture machine also carries an older `ballistics` on `PATH` that discovery would otherwise find first (Section 2.11).

The launcher writes one line to stderr and nothing else:

Listing A.34: The stderr line, present on every run

```
ballistics-lab: backend=stack-vm, engine=/Users/alexjokela/projects/ballistics-engine/target/
     release/ballistics
```

The transcripts in this appendix were produced against Lattice v0.6.4 and ballistics-engine 0.37.0. The Lattice version is printed in the banner; the engine version is printed by `:solve` and in every

table's provenance block. If your `engine` line reads something else, expect the assumption lists to match and the trajectory numbers to be worth re-checking — Chapter D covers how to pin a version and what to do when they disagree.

Appendix B

Engine Subcommand Index

The Ballistics Lab uses exactly one of the engine's subcommands: `solve-json`. The other forty-one exist for the times you want to work the engine directly — a quick recoil number, a tall-target correction factor, a PDF dope card, a saved profile you can reuse from a phone app. This appendix is the index to all of them.

Everything here was captured by running `ballistics <cmd> -h` against the binary on this machine's `PATH`, and by running the commands themselves. Nothing is transcribed from the engine's documentation, because in several places the documentation is ahead of the shipped binary.

B.1 Which binary is this?

Two engines can be installed at once, and they are not the same program

This book's stated engine is **0.30.1**. On the machine where these appendices were written, `ballistics` on `PATH` reports **0.32.0**, while the crate's release build reports 0.30.1. Both are present, both work, and the Ballistics Lab picks whichever the launcher finds first. Check yours before you trust a subcommand list:

```
$ ballistics --version
ballistics 0.32.0
```

The good news is that the difference is small, bounded, and was measured rather than assumed.

Table B.1: Every difference between the two shipped engines, measured

	0.30.1	0.32.0
Subcommands (excluding `help`)	30	36
Extra subcommands	—	`login`, `logout`, `recommend-powder`, `recommend-twist`, `recommend-col`, `calibrate-bc`
Extra sub-subcommand	—	`reticle import`
Help text for the shared 30		identical, byte for byte

That table is the output of a mechanical diff of `<cmd> -h` for all thirty shared subcommands: only `reticle` differed, by one added line. Everything else in this appendix therefore applies to both versions.

More importantly for a ballistics book: **the two engines produce identical numbers**. The same solve-json request was fed to both, twice, and the responses were compared byte for byte after normalising the version string:

Listing B.1: Two engines, one request, one answer

```
$ sed 's/"engine_version":"0.32.0"/"engine_version":"XX"/' resp_0320.json > a.txt
$ sed 's/"engine_version":"0.30.1"/"engine_version":"XX"/' resp_0301.json > b.txt
$ cmp a.txt b.txt && echo "identical apart from engine_version"
identical apart from engine_version
```

This held for a plain 1,000 m solve and for a second request exercising segmented wind, an explicit RK45 time step, and enhanced spin drift. Each binary is also deterministic run to run: five consecutive invocations produced five identical MD5 sums.

B.1.1 And now a third: 0.37.0

The census above was taken against 0.30.1 and 0.32.0. The engine has since reached **0.37.0**, and an appendix built on a subcommand census owes you a re-run of that census rather than the assumption that it held. It did not hold — but it moved in the one direction that is easy to state: **six subcommands were added and none were removed**.

That takes the count from thirty-six to **forty-two** excluding `help`, which is the number the rest of this appendix counts from. The thirty-six that were already there are all still present under the same names, and — this was checked mechanically, the same way the 0.30.1/0.32.0 diff was — not one of them changed its `Usage:` line. Every "Required" entry in Table B.8 therefore still holds on 0.37.0. Help text moved in exactly one of the thirty-six: `monte-carlo` gained a confidence-controlled

Table B.2: What 0.37.0 adds to the 0.32.0 census

Subcommand	Since	What it does
`explain`	0.33.0	Attributes the difference between two solved firing solutions to input groups
`error-budget`	0.33.0	Ranks declared input uncertainties by their own share of impact variance
`tolerance`	0.33.0	Bounds how wrong one input may be before the shot leaves the target
`dial-plan`	0.33.0	Ranks dial, hold, and hybrid ways to execute an angular correction
`adaptive-card`	0.33.0	A range card carrying a *measured* worst-case reconstruction error
`bc-convert`	0.33.2	Converts a published BC between the G1 and G7 families

`--adaptive` sampling mode, a `--seed` for reproducible runs, and `json` as the advertised name of the output mode it used to call `full` — which still works, so no command line in this appendix has gone stale.

The docs were ahead of both binaries, and the engine caught up

When the census above was taken, the engine repository's `CLI_USAGE.md` and `docs/SOLVE_JSON_V1.md` described a working tree that neither shipped binary matched, and two concrete examples were checked. Both have since resolved in the documentation's favour, which is worth recording precisely, because the resolution is the whole reason the preceding subsection exists.

`atmosphere.pressure_reference` was rejected as `unknown_field` by *both* 0.30.1 and 0.32.0. The cause was narrow and slightly embarrassing: `decode_solve_request_v1` carries a hand-maintained list of the field names a request may contain, and that list was never updated when the field shipped, even though the struct and the atmosphere resolver had handled it correctly all along. The effect was that the entire QNH-pressure feature was unreachable through the JSON wire path. 0.33.0 fixed exactly that, and 0.37.0 accepts the field and performs the reduction.

The `explain`, `error-budget`, `tolerance`, `dial-plan`, and `adaptive-card` subcommands existed in neither binary. All five shipped together in 0.33.0 as the decision-support train, and all five run on 0.37.0 with the flags the manual documents — they are the first five rows of Table B.2.

The advice survives its own examples. A manual is written against a working tree and a binary is a frozen artefact, so the two will part company again at the next release; when they disagree, believe `--help`. What has changed is only which of the two was ahead.

B.2 Invocation conventions

B.2.1 The two global-ish options

`--units <metric|imperial>` is accepted by all forty-two subcommands, including `completions`, and defaults to **imperial**. It selects the interpretation of every dimensional flag and the rendering of table and CSV output. Internal calculation is SI regardless. That uniformity is not an accident of the census: the six subcommands added since 0.32.0 each accept it too, so the switch has stayed genuinely global across five releases. Forty of the forty-two also accept the short form `-u`; `true-velocity` and `true-wind` take the long form only, which is worth knowing before you paste a command line between them.

`-o, --output <json|csv|table|pdf>` is offered by most subcommands and defaults to `table`. The parser accepts all four values everywhere it appears, but what happens next is per-command: on `trajectory`, `-o pdf` without `--output-file` exits 1 with `Error: "PDF output requires --output-file"`, while on `stability` the same flag falls back to the table and exits 0.

Prefer `-o json` or `-o csv` when scripting

The default table output is drawn with box-drawing characters and is intended for a terminal. The JSON and CSV forms are plain ASCII, stable, and are what the examples in this appendix use whenever the table would not survive the printed page.

B.2.2 Exit codes

Table B.3: Engine CLI exit codes, all observed

Code	Source	Example
0	success	any completed run
1	runtime refusal	`trajectory -o pdf` with no `--output-file`
2	argument parsing	unrecognised subcommand; a missing required flag
2	`solve-json`	malformed JSON, schema or semantic validation failure
3	`solve-json`	resource limit or solve failure

B.2.3 Where stdout ends and stderr begins

Diagnostics go to stderr, results to stdout, and the separation is clean enough to pipe. `true-velocity` prints `Calculating effective velocity via API...` on stderr before writing its JSON to stdout; redirecting stderr leaves a parseable document. `trajectory` likewise writes notes such as `note: twist rate not supplied; assumed 1:11.3"...` to stderr.

B.3 The machine interfaces

Two subcommands exist for programs rather than people. One of them is the entire basis of this book.

B.3.1 `solve-json`

Listing B.2: The complete help text

```
Solve one explicit-SI v1 JSON request from stdin and write one JSON envelope to stdout

Usage: ballistics solve-json [OPTIONS]

Options:
  -u, --units <UNITS>  Unit system for input/output (metric or imperial) [default: imperial] [
     possible values: metric, imperial]
  -h, --help           Print help (see more with '--help')
```

One request in, one compact envelope plus a newline out. No profiles, no files, no network, no shell. `--units` is accepted and has no effect: the protocol is SI at every field.

This is the only subcommand the Ballistics Lab ever invokes, and it invokes it with a fixed, literal argv — the string `"solve-json"` and nothing else (`process_backend.lat:849`). Chapter C is the full schema.

B.3.2 `mcp`

Listing B.3: The complete help text

```
Run a Model Context Protocol (MCP) server over stdio, exposing solve and engine_info tools

Usage: ballistics mcp [OPTIONS]
```

Exposes two tools over stdio: `solve`, whose arguments are a solve-json v1 request object, and `engine_info`. It is the same service behind a different transport — useful if you want the engine reachable from an agent harness rather than from a subprocess.

B.4 Core solvers

B.4.1 `trajectory`

Calculate a single trajectory. This is the flagship, and it is enormous: 107 options, more than every other subcommand's options put together. Every physics toggle in the engine lives here and nowhere else.

Listing B.4: A trajectory run and its default table (box drawing removed for print)

```
$ ballistics trajectory -v 2700 -b 0.243 -m 175 -d 0.308 \
      --drag-model g7 --auto-zero 100 --max-range 300
 Max Range:           300.00 yd
 Max Height:            1.72 yd
 Zero Angle:          0.0717 deg
 Time of Flight:       0.373 s
 Impact Velocity:    2162.56 fps
 Impact Energy:      1816.94 ft-lb
 Stability (SG):        2.00
 Status:             assumed
```

Listing B.5: … and the stderr note that came with it

```
note: twist rate not supplied; assumed 1:11.3" for the stability/spin-drift estimate -- Sg shown
     is not an evaluated stability verdict.
```

Two characters substituted, and only two

The engine draws its terminal tables with box-drawing characters and prints a degree sign and an em dash. Those cannot be typeset inside a verbatim listing in this book's font setup. In the two listings above, the box borders are removed, the degree sign is written `deg`, and the em dash is written as two hyphens. *No digit anywhere in this appendix has been altered.* Where a command's table would not survive that treatment, this appendix shows its `-o json` or `-o csv` form instead, which is plain ASCII.

Table B.4 groups the flags you will reach for. The full list is `ballistics trajectory --help`.

Table B.4: `trajectory` flags, by job

Job	Flags
The load	`-v -b -m -d –drag-model –bullet-length –twist-rate –twist-right`
The shot	`–max-range –auto-zero –shooting-angle –cant –shot-direction –latitude`
Sight geometry	`–sight-height –sight-offset –bore-height –zero-poi-up –zero-poi-right`
Atmosphere	`–temperature –pressure –pressure-type –humidity –altitude –density-altitude`
Wind	`–wind-speed –wind-direction –wind-ref –wind-vertical –wind-segment –enable-wind-shear –wind-shear-model`
Physics	`–enable-magnus –enable-coriolis –enable-spin-drift –enable-aerodynamic-jump –enable-pitch-damping –enable-precession –use-powder-sensitivity`
Integrator	`–use-euler –use-rk4-fixed –time-step`
Drag data	`–drag-table –cd-scale –bc-adjustment –bc-reference –bc-segment –use-bc-segments –bc-table-dir –bc-table-auto`
Zero-day conditions	`–zero-velocity –zero-temperature –zero-pressure –zero-humidity –zero-altitude –zero-powder-temp`
Output	`-o –full –plot –sample-trajectory –sample-interval –drops-reference –with-drag-coefficient –ignore-ground-impact`
Profiles	`–saved-profile –profile + –profile-row –location + –site –zero-set`
PDF card	`–output-file –font-scale –font-preset –bold-data –adjustment-unit –windage-unit`

Ground impact truncates the run by default

The same command with `--max-range 1000` does not necessarily give you 1,000 yards. Measured, on a .308 175 gr G7 at 2750 fps with a 100 yd zero:

```
default                   max_range = 524.8493662123586 yd
--ignore-ground-impact    max_range = 1000.0 yd
```

The bullet reached the ground at 525 yards, and the solver stopped there — correctly. If you want the ballistic path past impact, say so.

Two more surprises worth knowing before you script this command. `--sample-interval` is **always metres**, in both unit systems: a PDF card built with `--sample-interval 100` reported `4 ranges from 109 to 437 yards`, which is 100 m to 400 m. And PDF output needs two flags, not one:

Listing B.6: The PDF dope card path, both refusals and the success

```
$ ballistics trajectory ... -o pdf
Error: "PDF output requires --output-file"
$ ballistics trajectory ... -o pdf --output-file card.pdf
Error: "PDF output requires --sample-trajectory flag for trajectory data"
$ ballistics trajectory ... --sample-trajectory --sample-interval 100 \
      -o pdf --output-file card.pdf
PDF dope card written to: card.pdf
  4 ranges from 109 to 437 yards
```

B.4.2 `zero`

Calculate zero angle for a target — or, with `--from-angle`, the two ranges a given bore angle crosses the line of sight.

Listing B.7: Solving the zero angle for a 100 yd zero

```
$ ballistics zero -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --target-distance 100 --sight-height 1.5 -o json
{
  "max_ordinate": 0.0449668153920438,
  "point_blank_range": 170.77255889626645,
  "sight_adjustment_moa": 3.824031670993379,
  "units": "imperial",
  "zero_angle_degrees": 0.06373386118322298,
  "zero_angle_moa": 3.824031670993379,
  "zero_angle_mrad": 1.11236572265625
}
```

Listing B.8: Running it backwards: one angle, two crossings

```
$ ballistics zero -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --from-angle 0.0637 --sight-height 1.5 -o json
{
  "far_zero_range": 99.82120920392995,
  "max_ordinate": 0.044919331689567764,
  "near_zero_range": 58.68505834231162,
  "point_blank_range": 170.77255897343304,
  "primary_crossing": "far",
  "sight_adjustment_moa": 3.8220000000000014,
  "units": "imperial",
  "zero_angle_degrees": 0.06370000000000002,
  "zero_angle_moa": 3.8220000000000014,
  "zero_angle_mrad": 1.1117747335203882,
  "zero_range": 99.82120920392995
}
```

The two calls are inverses to five figures: feeding the first call's 0.0637 degrees back in returns a far zero of 99.82 yd. Note that `--target-distance` and `--from-angle` are mutually exclusive, and that `--target-height` is measured in yards or metres — a height in units of range, which invites being mis-entered as inches.

B.4.3 `monte-carlo`

Run Monte Carlo simulation. Propagates input uncertainty — muzzle velocity, launch angle, BC, wind speed, wind direction — through the solver and reports a dispersion distribution, or with `--wez` a hit probability swept against range.

Key flags: `-n/--num-sims` (default 1000); the five standard deviations `--velocity-std --angle-std --bc-std --wind-std --wind-direction-std`; the target (`--target-distance --target-radius`, or `--target-size WxH` for the WEZ sweep); and `--wez-start --wez-end --wez-step --wind-call-error` for the sweep itself. Output modes are `summary`, `full`, and `statistics`.

B.4.4 `mpbr`

MPBR

Maximum point-blank range: the farthest distance at which a bullet zeroed for the purpose stays inside a vital zone of a given size without any elevation hold — neither too high at mid-range nor too low at the end.

Listing B.9: MPBR for an 8-inch vital zone

```
$ ballistics mpbr -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --vital-zone 8 --sight-height 1.5 -o json
{
  "far_zero": 255.7036809019185,
  "impact_energy": 1813.9959509474554,
  "impact_energy_unit": "ft-lb",
  "impact_velocity": 2160.8073605821496,
  "impact_velocity_unit": "fps",
  "max_ordinate": 4.00458849578977,
  "max_ordinate_distance": 139.0,
  "max_ordinate_unit": "in",
  "mpbr": 300.8519598885623,
  "mpbr_unit": "yd",
  "near_zero": 21.52071820766701,
  "optimal_zero": 255.66406250000006,
  "vital_zone": 8.0,
  "vital_zone_unit": "in"
}
```

Read that as: zero at 256 yd, hold centre out to 301 yd, and the bullet never leaves a four-inch half-zone — peaking 4.00 in high at 139 yd.

B.4.5 `compare`

Compare multiple loads side-by-side at the same conditions. Two to eight loads, each described by a colon-delimited `--load` spec of `NAME:DRAG:BC:MASS:VELOCITY[:DIAMETER]`, or by a saved `--profile NAME`. `--zero-distance` is required.

Listing B.10: Two loads, one table

```
$ ballistics compare --load "A:g7:0.243:175:2700" --load "B:g7:0.290:200:2600" \
      --zero-distance 100 --start 300 --end 600 --step 300 -o csv
range_yd,A_drop_in,A_drop_MIL,A_drift_in,A_drift_MIL,A_velocity_fps,A_energy_ft-lb,A_tof_s,
     B_drop_in,B_drop_MIL,B_drift_in,B_drift_MIL,B_velocity_fps,B_energy_ft-lb,B_tof_s
300,14.41,1.33,-6.91,-0.64,2162,1817,0.373,15.21,1.41,-6.01,-0.56,2155,2062,0.380
600,95.69,4.43,-31.10,-1.44,1690,1109,0.843,97.32,4.51,-26.52,-1.23,1756,1369,0.843
```

The CSV widens with each load rather than adding rows — one row per range, one column block per load. Contrast the Ballistics Lab's `:compare`, which puts one row per series per distance and adds delta columns.

B.5 Cards and tables

All five share a family resemblance: a required `--zero-distance` (or its equivalent), a `--start`/`--end`/`--step` grid, and turret-unit controls.

B.5.1 `range-table`

Generate comprehensive range table (drop, wind, velocity, energy, ToF). The everything card.

Listing B.11: A three-row range table

```
$ ballistics range-table -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --zero-distance 100 --start 200 --end 600 --step 200 -o csv
range_yd,drop_in,drop_mil,wind_in,wind_mil,velocity_fps,energy_ft-lb,tof_s
200,3.5,0.487,-3.0,-0.41,2334,2117,0.24
400,30.8,2.136,-12.8,-0.89,1999,1552,0.52
600,93.2,4.314,-31.1,-1.44,1690,1109,0.84
```

Defaults you will want to override: `--start 100 --end 1200 --step 50` and a standing `--wind-speed 10 --wind-direction 90`. Turret units come from `--adjustment-unit` and the independent `--windage-unit`; scope tracking correction from `--elevation-cf` and `--windage-cf`.

B.5.2 `come-ups`

Generate come-up (elevation adjustment) table only. The narrow version of `range-table`: range, drop, come-up, velocity, energy, time.

Listing B.12: A come-up table in CSV

```
$ ballistics come-ups -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --zero-distance 100 --start 200 --end 600 --step 200 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
200,0.487,0.487,2334,2117,0.239
400,2.136,1.649,1999,1552,0.517
600,4.314,2.178,1690,1109,0.843
```

The come-up column is an increment, not a dial setting

Check the arithmetic: $2.136 - 0.487 = 1.649$, and $4.314 - 2.136 = 2.178$. The `come_up` value is the change from the *previous row*, not the absolute correction from zero. The terminal table makes this visible by printing a dash in the first row; the CSV instead repeats the drop. If you dial 1.649 mil at 400 yards from a 100 yard zero you will shoot low — the absolute correction is 2.136 mil, which is the `drop_mil` column.

This table and the Ballistics Lab's `:dope` agree, once you line the right columns up — and lining them up takes a moment, because the two programs print a column of the same name meaning different things. Pin the mapping down before comparing anything. `:dope`'s `Come-up (mil)` is the **total** dial from the zero. Its counterpart here is therefore `drop_mil` — and this command's `come_up_mil` column, being an increment, has no counterpart in the Ballistics Lab at all.

With that mapping in hand the two paths agree. Here is one 140 gr 6.5 mm load — BC 0.326 G7, 2710 ft/s, 1.75-inch sight height, 100 yard zero, 4000 ft, 50 °F, 35% humidity — driven twice: once over this flag interface, once over `solve-json` from the Ballistics Lab. Bullet length is omitted on both sides, because `come-ups` has no input for it and we want the two solving the same problem.

Listing B.13: The reference load through `come-ups`

```
$ ballistics come-ups --velocity 2710 --bc 0.326 --mass 140 --diameter 0.264 \
      --drag-model g7 --zero-distance 100 --sight-height 1.75 --altitude 4000 \
      --temperature 50 --humidity 35 --start 100 --end 600 --step 100 \
      --adjustment-unit mil -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,0.000,0.000,2587,2080,0.113
200,0.478,0.478,2467,1892,0.232
300,1.168,0.689,2350,1717,0.357
400,1.949,0.781,2237,1555,0.487
500,2.804,0.855,2127,1406,0.625
600,3.730,0.926,2020,1268,0.770
```

Listing B.14: The same load through the Ballistics Lab; `:solve` header and provenance block elided

```
DOPE: 140 gr 6.5 mm reference load
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     +0.00 |         +0.00 |         2587.03 |        2080.17 |     0.11 | 2.34
       200.00 |     +3.44 |         +0.48 |         2467.13 |        1891.81 |     0.23 | 2.23
       300.00 |    +12.61 |         +1.17 |         2350.38 |        1717.00 |     0.36 | 2.12
       400.00 |    +28.07 |         +1.95 |         2236.87 |        1555.16 |     0.49 | 2.02
       500.00 |    +50.47 |         +2.80 |         2126.71 |        1405.77 |     0.63 | 1.92
       600.00 |    +80.57 |         +3.73 |         2019.81 |        1267.98 |     0.77 | 1.82
Signs: drop + below/- above; come-up + correction up/- correction down
```

Station for station: 0.478 against +0.48, 1.168 against +1.17, 1.949 against +1.95, 2.804 against +2.80, 3.730 against +3.73 — agreement everywhere, to the two decimals the Ballistics Lab prints. The velocity and energy columns match as well. There is no reason to prefer one path over the other for elevation; pick whichever fits the job. `come-ups` is a flag away; the Ballistics Lab carries the provenance.

Why the two land on the same zero

`come-ups` takes `--sight-height` and solves the zero to the line of sight. Over `solve-json` the same datum is an explicit field, `shot.target_height_m` — the height above ground the zero is solved to, which the engine defaults to 0.0, the ground. The Ballistics Lab therefore states it: when an experiment sets no target height of its own, the wire adapter fills in muzzle height plus sight height, which is the height of the line of sight (`protocol_v1.lat:74-92`). Both paths are then zeroing the same rifle to the same place, which is why the tables above line up.

B.5.3 `wind-card`

Generate wind drift card (wind deflection at multiple speeds). One row per range, one column per wind speed.

Listing B.15: Drift in mils at two wind speeds

```
$ ballistics wind-card -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --zero-distance 100 --wind-speeds 5,10 --start 200 --end 600 --step 200 -o csv
range_yd,wind_5_mph,wind_10_mph
200,-0.2,-0.4
400,-0.4,-0.9
600,-0.7,-1.4
```

Defaults to `--wind-speeds 5,10,15,20` at `--wind-angle 90`. Several angles at once with `--wind-angles 30,60,90`.

B.5.4 `lead`

Moving-target lead table (hold in the direction of target travel). `--target-speed` is required; `--target-angle` defaults to 90 (a pure crosser).

Listing B.16: Leads for a 10 mph crossing target

```
$ ballistics lead -v 2700 -b 0.243 -m 175 -d 0.308 --drag-model g7 \
      --target-speed 10 --start 100 --end 300 --step 100 -o csv
range_yd,tof_s,lead_yd,lead_mil,intercept_yd,iterations,error
100,0.115,0.56,5.631,100.0,0,
200,0.239,1.17,5.843,200.0,0,
300,0.373,1.82,6.072,300.0,0,
```

The `intercept_yd` column is the honest part: the solver iterates to the point where the bullet and the target arrive together, so the lead is not the product of speed and time of flight.

B.5.5 `hold-corridor`

Robust hold corridors across named segmented-wind scenarios. You give it a JSON file of named wind scenarios and a list of ranges; it returns the hold that survives every scenario — a minimax rather than an average.

Listing B.17: The `--scenarios` file shape

```
{"version":1,
 "nominal":"low",
 "scenarios":[{"name":"low","segments":["5:90:400"]}]}
```

Required: `--scenarios <FILE>` and `--ranges R1,R2,...`. Optional target geometry with `--target rect:WxH` or `circle:D`. With `-o json` the output is a versioned `RobustHoldReportV1`.

B.6 Truing and calibration

Truing

Adjusting a model's inputs — usually muzzle velocity or ballistic coefficient — until its predictions match what the rifle actually did on paper. It is not curve-fitting: you are correcting a physical parameter you measured badly, not inventing a fudge factor.

B.6.1 `estimate-bc`

Estimate BC from trajectory data (drop and/or velocity), for G1, G7, or both.

Listing B.18: Fitting a G7 BC to two observed drops

```
$ ballistics estimate-bc -v 2700 -m 175 -d 0.308 \
      --data "300,14.4;600,95.7" --drag-model g7 --zero-range 100 -o json
{
  "diameter": 0.007823199999999999,
  "mass": 0.01133980925,
  "variants": [
    {
      "drag_model": "G7",
      "estimated_bc": 0.23,
      "fit_basis": "drop",
      "fit_rms": 0.519,
      "fit_rms_unit": "in",
      "n_points": 2,
      "reliable": true
    }
  ],
  "velocity": 822.96
}
```

Note that the echoed `diameter`, `mass`, and `velocity` are SI even though the input was imperial. `--data` takes `dist,drop;dist,drop` pairs; `--velocity-data` takes `dist,vel` pairs; `--drag-model both` fits G1 and G7 and returns two variants.

B.6.2 `true-velocity`

Calculate effective muzzle velocity from observed drop.

Listing B.19: Truing MV against a 4.5 mil drop at 600 yd

```
$ ballistics true-velocity --measured-drop 4.5 --range 600 \
      -b 0.243 -m 175 -d 0.308 --drag-model g7 --zero-distance 100 \
      --drop-unit mil -o json 2>/dev/null
{
  "calculated_drop_mil": 4.5,
  "confidence": "high",
  "effective_velocity": 2652.0,
  "final_error_mil": -0.003,
  "iterations": 7,
  "measured_drop_mil": 4.5,
  "used_bc_table": false,
  "velocity_unit": "fps"
}
```

With one or more `--observed RANGE:DROP[:SIGMA]` entries it upgrades to a joint MV-and-BC Bayesian fit with `--mv-prior MEAN:SIGMA` and `--bc-prior MEAN:SIGMA`, and can predict a new range with `--predict-range`. `--chrono-velocity` and `--chrono-distance` let you correct a chronograph reading taken a few feet downrange. This command is G1/G7 only.

B.6.3 `true-wind`

Back-solve the effective crosswind from an observed horizontal miss. You tell it where the shot landed left or right; it tells you what the wind actually was.

`--miss RANGE:RIGHT[:SIGMA]` is repeatable. **The RIGHT and SIGMA fields are always linear inches**, regardless of `--units`, because they are tape measurements off a target. **`--twist-rate` is required** — without it the solver cannot separate spin drift from wind and will attribute the drift to the wind call. Supply `--latitude` and `--shot-direction` to subtract Coriolis, and `--called-wind` to get a correction factor for your own wind reading.

B.6.4 `plan-truing`

Recommend target ranges for an identifiable MV + BC truing experiment. Run this *before* you go to the range. Given your measurement resolution (`--measurement-resolution`, required) it picks the distances at which a velocity error and a BC error produce distinguishable drops — so the fit has a unique answer. `--candidate-ranges` or `--range-grid START:END:STEP` bound the search; `--observation-count` defaults to 3.

B.6.5 `dsf`

Derive a Mach-keyed drop-scale-factor (DSF) point from an observed drop, and save it to a profile's DSF table. All three of `--saved-profile`, `--range`, and `--observed-drop` are required, and the drop is a *unit-suffixed literal with no separator*: `5.1mil`, `17.4moa`, `42.0in`. This is the one command that writes back into stored data as its primary effect.

B.6.6 `tall-target`

Compute a scope tracking correction factor from a tall-target test. Pure arithmetic: correction factor equals measured travel divided by dialed travel. No trajectory is solved.

Listing B.20: A scope that tracks 3 percent slow

```
$ ballistics tall-target --dialed 10 --measured 34.9 --range 100

Tall-Target Test Result
  Dialed travel:   10.00 MIL
  Actual travel:   9.69 MIL (34.90 in at 100 yd)
  Correction factor (actual / dialed): 0.9694
  Enter this as --elevation-cf / profile elevation_cf (or the windage equivalents); dial
     solutions are divided by it.
```

The correction factor is bounded to $(0.5, 1.5)$ where it is consumed, and it divides dial-unit outputs only — raw linear inches are untouched.

B.6.7 `generate-bc-segments`

Generate BC segments for velocity-dependent BC. Takes `-b -m -d` and an optional `--model` name, and emits the `--bc-segment VMIN:VMAX:BC` triples that `trajectory` consumes.

B.7 Standalone calculators

None of these integrates a trajectory. They are the arithmetic you would otherwise do on a phone, with the engine's constants.

B.7.1 `powder`

Resolve the powder-temperature-adjusted muzzle velocity without running a trajectory. Two mutually exclusive models, and the curve wins when both are given:

1. **Linear**: `--powder-temp-sensitivity` in fps per degree F, against a reference `--powder-temp`.

2. **Measured curve**: `--powder-temp-curve "40:2620,70:2700,100:2760"`, piecewise-linear between the given knots and *clamped* at the endpoints — it never extrapolates.

A verified linear run, with `-v 2750 --temperature 20 --powder-temp 70 --powder-temp-sensitivity 1.2 -m 175`, reported a resolved velocity of 2690.0 fps: a 50 degree drop times 1.2 fps per degree is 60 fps, exactly the reported shift. `--sweep START:END:STEP` prints a table across temperatures.

B.7.2 `recoil`

Free recoil energy, velocity, and impulse from a SAAMI momentum balance.

Listing B.21: An 11 lb rifle, 175 gr over 45 gr of powder

```
$ ballistics recoil -b 175 -c 45 -v 2700 -f 11
Free Recoil (SAAMI Momentum Balance)
====================================
  Firearm weight:      11.00 lb
  Bullet weight:       175.0 gr
  Charge weight:       45.0 gr
  Muzzle velocity:     2700.0 fps
  Gas velocity model:  saami-rifle  (4725.0 fps)

  Recoil velocity:     8.90 fps
  Recoil energy:       13.53 ft-lb
  Recoil impulse:      3.042 lb-s
```

All four of `-b -c -v -f` are required. `--firearm-type rifle|pistol|shotgun-average|shotgun-long` selects the gas-velocity multiplier; `--gas-velocity-factor` and `--gas-velocity` override it.

B.7.3 `power-factor`

Power factor (bullet weight times velocity divided by 1000) with per-organization USPSA/IDPA/SASS rulebook pass/fail.

Listing B.22: A rifle load run through the pistol rulebooks

```
$ ballistics power-factor -w 175 -v 2700
Power Factor
============
  Weight:              175.0 gr
  Velocity:            2700.0 fps
  Power factor (raw):  472.50
  Power factor (scored): 472

  Org       Class                          Min PF    Pass      Velocity limit
  USPSA     Minor                             125    PASS                   -
  USPSA     Major                             165    PASS                   -
  IDPA      BUG                                95    PASS                   -
  IDPA      Stock Revolver / CCP              105    PASS                   -
  IDPA      SSP / ESP / CO                    125    PASS                   -
  IDPA      PCC                               135    PASS                   -
  IDPA      Enhanced Revolver                 155    PASS                   -
  SASS      Smokeless - Pistol-Revolver        60    FAIL        400-1000 fps
  SASS      Smokeless - Rifle                  60    FAIL        400-1400 fps
```

The SASS rows fail not on power factor but on the velocity ceiling, which the table reports in its own column. `--organization` filters to one body.

B.7.4 `stability`

Analyze gyroscopic stability (Miller stability factor). `-l/--length` and `-t/--twist-rate` are required.

Listing B.23: Sg for the reference match load

```
$ ballistics stability -l 1.24 -t 8 -m 175 -d 0.308 -v 2700 -o json
{
  "bullet_diameter": 0.308,
  "bullet_length": 1.24,
  "bullet_mass": 175.0,
  "min_twist_sg_1_0": 15.61,
  "min_twist_sg_1_5": 12.65,
  "muzzle_velocity": 2700.0,
  "stability_factor": 3.8,
  "status": "Fully Stable",
  "twist_rate": 8.0,
  "twist_rate_unit": "in/turn",
  "units": "imperial"
}
```

That 3.80 is the same number the BALLISTICS LAB prints as `stability factor` for the reference experiment — the engine computes it once and reports it through every surface.

B.7.5 `drag-curve`

Print a built-in reference drag curve as (Mach, Cd) data. The default model is `g7`, not `g1`, which surprises people.

Listing B.24: The head of the G7 table

```
$ ballistics drag-curve --drag-model g7 -o csv
mach,cd
0,0.1198
0.05,0.1197
0.1,0.1196
0.15,0.1194
0.2,0.1193
```

Nine families are available — `G1 G2 G5 G6 G7 G8 GI GS RA4` — each backed by its own table. All nine are accepted by `solve-json` as well, which was not true when this appendix was first written: 0.30.1 and 0.32.0 took only `G1 G6 G7 G8` through the JSON path and rejected the other five with `invalid_value`. The widening landed with 0.33.1 and shipped in 0.33.2 — a side effect of making `monte-carlo --wez` able to attribute variance for every built-in family, which it could not do for a model the v1 request shape had no way to name.

The BALLISTICS LAB still restricts itself to the original four, but that is now *its* choice rather than the engine's refusal; Chapter 10 gives the three places it enforces it.

B.8 Optics and reticles

B.8.1 `reticle`

Reticle hold points and parametric reticle generation. A command with sub-commands:

Table B.5: `reticle` sub-commands

Sub-command	Purpose
`hold`	Place a firing solution in a reticle and name the nearest mark
`generate`	Build a parametric reticle description (`mil-grid`, `tree`, `bdc`)
`import`	**0.32.0 only.** Convert a third-party "Ventum" reticle spec into the engine's description

`reticle generate -o json` emits exactly the document that `reticle hold -reticle-json`, `mark-to-range -reticle-json`, and `profile save -reticle-json` consume. `hold` takes either a solution directly (`--drop-mil --wind-mil`) or a range plus the full load arguments, along with `--mag`.

B.8.2 `mark-to-range`

Map each reticle mark to the range where it lands. The inverse question: not "what do I hold at 600?" but "what is each hash worth?". `--mark <MIL>` is repeatable, or point it at a `--reticle-json` file. `--focal-plane ffp|sfp` and `--mag` matter here, because on a second-focal-plane scope the marks scale with magnification and the trajectory does not.

B.8.3 `bdc-match`

Solve the magnification that makes an SFP BDC reticle match this load. Give it at least two `--mark -range MIL:RANGE` pairs and it finds the magnification at which the reticle's marks line up with the load's drops. First-focal-plane is rejected with an explanation, which is correct: on an FFP scope the mark subtension does not change, so there is nothing to solve. `--residual-warn` defaults to 0.2 mil.

B.8.4 `optimal-zero`

Solve the one zero that makes a whole stage of targets holdable. Between two and sixteen `--target RANGE[:HEIGHT]` entries, with `--vital <SIZE>` as the fallback height. This is MPBR generalised to an arbitrary set of targets rather than one vital zone.

B.9 Storage and shell

B.9.1 `profile`

Manage saved ballistic profiles. Profiles live in `~/.ballistics/profiles/<NAME>.json`, one file per profile. There is no environment override and no per-project store.

Values are stored *in the units they were entered in* and converted on load: the serialised record carries a `units` field alongside the numbers (`ProfileData` in `src/profile.rs`), so a metric-saved profile consumed by an imperial command produces the same physics. Note also, from `profile save --help` itself, that the stored atmosphere fields are kept for round-trip parity: `trajectory -saved-profile` does not read them back.

"About forty more optional" is a bigger number than it was, and the growth is worth a sentence because it changes what a profile *is*. In 0.32.0 `profile save -h` printed thirty option lines; on 0.37.0 it prints forty-four. The fourteen new ones are all one idea: the profile no longer describes only the load and the atmosphere it was fired in, it also describes the *optic*. Twelve of them record turret mechanics and

Table B.6: `profile` sub-commands

Sub-command	Notes
`save <NAME>`	`--velocity --bc --mass --diameter` required; about forty more optional. Re-saving carries the DSF table and attached reticle forward unless cleared
`import <FILE>`	`.a7p` (ArcherBC2). `--zero-click` is needed because the format stores zeroing as raw click counts
`list`	Name, velocity, BC, mass, diameter, drag model
`show <NAME>`	Boxed detail view
`delete <NAME>`	
`zero-set add|remove|list`	Named alternate zeros and per-load dial corrections, selected at solve time with `--zero-set NAME`

hold extent — `--clicks-per-rev`, `--zero-stop`, `--travel-up/-down`, `--windage-travel-left/-right`, `--hold-up/-down/-left/-right`, and `--turret-elev/--turret-wind` for the turret's current dialed state — and the remaining two, `--clear-turret` and `--clear-click`, exist because every optional profile field is carried forward unchanged on re-save, so removing one has to be asked for explicitly. That is what `dial-plan` reads when you give it `--profile` instead of spelling out an optic on the command line.

Two different `-profile` flags

This is the sharpest edge in the whole CLI. On every command *except* `trajectory`, `--profile <NAME>` means a saved profile. On `trajectory`, `--profile <FILE>` means a CSV gun-profile file and requires `--profile-row <NAME>`; the saved-profile flag there is `--saved-profile <NAME>`. Passing a profile name to `trajectory -profile` does not load the profile — it demands a row selector for a file that does not exist.

B.9.2 `completions`

Generate shell completions. One positional argument: `bash`, `elvish`, `fish`, `powershell`, or `zsh`.

B.10 Online reverse solvers (0.32.0 only)

These six subcommands do not exist in 0.30.1; there they return `error: unrecognized subcommand`. All of them call a remote service (override with `--api-url`) and need a token stored by `login`, read from `BALLISTICS_API_TOKEN` or from `~/.ballistics/credentials.toml`.

Table B.7: The 0.32.0 online subcommands

Subcommand	Required flags
`login` / `logout`	`--token` (prompts if omitted)
`recommend-powder`	`--cartridge --bullet-weight --desired-velocity`
`recommend-twist`	`--caliber --weight --overall-length --nose-length`
`recommend-col`	`--cartridge`
`calibrate-bc`	`--diameter --length --weight`

None of this reaches the Lab

The BALLISTICS LAB drives `solve-json`, which performs no filesystem and no network access at all. Nothing in this section can be triggered by a Lab session, and no Lab run depends on a token, a subscription, or connectivity.

B.11 The complete index

Table B.8 is every subcommand in one place, with the flags you cannot omit. The "Ver" column is the first release in which the subcommand exists: "all" means it was already there in 0.30.1, "0.32" marks the six that 0.30.1 does not have, and "0.33" marks the six added since. Required arguments were re-checked against 0.37.0 and none of the original thirty-six changed.

Table B.8: All 42 subcommands and their required arguments

Subcommand	Ver	Required
`solve-json`	all	— (one JSON request on stdin)
`mcp`	all	—
`trajectory`	all	— (but a load is needed for a useful answer)
`zero`	all	`-v -b -m -d`, then `-target-distance` *or* `-from-angle`
`monte-carlo`	all	`-v -b -m -d`
`mpbr`	all	—
`compare`	all	`-zero-distance`
`range-table`	all	`-zero-distance`
`come-ups`	all	`-zero-distance`
`wind-card`	all	`-zero-distance`
`lead`	all	`-target-speed`

Subcommand	Ver	Required
`hold-corridor`	all	`--scenarios --ranges`
`estimate-bc`	all	`-v -m -d`
`true-velocity`	all	`--measured-drop --range -b -m -d`
`true-wind`	all	`--miss -v -b -m -d --twist-rate`
`plan-truing`	all	`--measurement-resolution`
`dsf`	all	`--saved-profile --range --observed-drop`
`tall-target`	all	`--dialed --measured --range`
`generate-bc-segments`	all	`-b -m -d`
`powder`	all	—
`recoil`	all	`-b -c -v -f`
`power-factor`	all	`-w -v`
`stability`	all	`-l -t`
`drag-curve`	all	—
`reticle`	all	a sub-command
`mark-to-range`	all	`--mark` *or* `--reticle-json`
`bdc-match`	all	`--mark-range` (at least two)
`optimal-zero`	all	`--target` (two to sixteen)
`profile`	all	a sub-command
`completions`	all	a shell name
`login`	0.32	— (`--token` prompts)
`logout`	0.32	—
`recommend-powder`	0.32	`--cartridge --bullet-weight --desired-velocity`
`recommend-twist`	0.32	`--caliber --weight --overall-length --nose-length`
`recommend-col`	0.32	`--cartridge`
`calibrate-bc`	0.32	`--diameter --length --weight`
`explain`	0.33	`-a -b --ranges`
`error-budget`	0.33	`--request --ranges --sigma`
`tolerance`	0.33	`--request --range --target --axis`
`dial-plan`	0.33	`--range`, plus `--elevation` *or* `--windage`
`adaptive-card`	0.33	`--start --end`, plus a load and `--zero-distance`
`bc-convert`	0.33	`--source-model --target-model`, then `--bc` *or* `--bc-segment`

B.12 The traps, ranked

Every item below was reproduced on the shipped binary.

1. **`trajectory -profile` takes a FILE; everything else takes a NAME.** Use `--saved-profile` on `trajectory`.
2. **Ground impact truncates runs by default.** `--max-range 1000` returned 524.85 yd. Use `--ignore-ground-impact`.
3. **`come-ups`' come-up column is an increment**, not the absolute dial. The absolute correction is the drop column — and it is the drop column that the Ballistics Lab's *total*-dial `Come-up (mil)` has to be compared against.
4. **`--sample-interval` is always metres**, in both unit systems.
5. **`solve-json` ignores `--units`** entirely — it is SI at every field, in both directions.
6. **PDF output needs two flags**: `--output-file` and `--sample-trajectory`.
7. **The default `--drag-model` for `drag-curve` is G7**, not G1.
8. **`solve-json` takes all nine drag families, but the Ballistics Lab still takes four.** Since 0.33.2 the engine accepts G2, G5, GI, GS and RA4 through the JSON path alongside G1, G6, G7 and G8. A load in one of the five is refused by the Lab's own validators, not by the engine, so the error you get names a Lab field path and no process is ever spawned.
9. **The documentation and the binary drift apart in both directions.** The gap that was open when this appendix was written has closed, and a new one will open at the next release. Verify against `--help` before you rely on a flag.

Appendix C

The solve-json v1 Schema

This is the contract. One JSON request on stdin, one JSON envelope on stdout, SI at every field, no files and no network. It is the whole surface between the Ballistics Lab and the `ballistics-engine`, and it is the reason the Lab can be written in ordinary Lattice with no native extension and no C ABI.

Every field in this appendix was probed against the shipped binary — one field per request, accepted or rejected, and the response inspected. Where the engine's own documentation and the binary disagree, this appendix follows the binary and says so.

Which engine produced this

This appendix was first probed against `ballistics 0.32.0`, and cross-checked against 0.30.1: after normalising the `engine_version` string the two were byte-identical, down to the assumption texts and the trajectory numbers.
Between 0.32.0 and 0.37.0 the wire surface grew, so that provenance no longer holds everywhere. Every passage covering something that changed — the drag model list, `pressure_reference`, `wind_shear_model`, the `zero_distance_m` / `muzzle_angle_rad` pairing, the `resolved_request` echo, and the notice-code table — was re-probed against 0.37.0, and says so where it quotes an envelope. Everything else is an unretouched 0.32.0 capture. Nothing in v1's invariants permits those to have changed meaning, but they have not been re-run, and this appendix does not pretend otherwise.

C.1 Transport

Listing C.1: The entire calling convention

```
$ ballistics solve-json < request.json > response.json
```

One process, one solve. Standard input carries exactly one JSON document; standard output carries exactly one compact envelope followed by a newline. The process exits with a code that is part of the contract.

Table C.1: `solve-json` exit codes, all reproduced

Exit	Meaning
0	`status: "ok"` envelope
1	`io_error` or `internal_error`
2	malformed JSON, schema violation, or semantic validation failure
3	resource limit, or the solve itself failed

Exit code and error code must agree

The Lab treats a disagreement between the envelope's error code and the process exit status as a hard protocol violation, not a warning. The mapping is fixed: parse and validation codes require exit 2, `resource_limit` and `solve_failed` require exit 3, `io_error` and `internal_error` require exit 1 (`process_backend.lat:530–544`). An engine that reports `invalid_value` and exits 3 gets its whole response rejected.

C.1.1 The input size limit

Standard input is capped at 1,048,576 bytes, and the limit is inclusive. Both sides were probed by padding a valid request with whitespace to exact byte counts:

Listing C.2: The 1 MiB boundary

```
1048576 bytes -> exit 0  {"schema_version":1,"engine_version":"0.32.0","status":"ok",...
1048577 bytes -> exit 3  {"schema_version":1,"status":"error","error":{"code":"resource_limit",
                          "message":"solve-json input exceeds the 1048576-byte limit",...
```

The Lab enforces the same number on its own side before spawning anything, so an oversized request never reaches a child process (`MAX_REQUEST_BYTES`, `process_backend.lat:25-34`).

C.2 The request: nine required members

A v1 request has exactly nine top-level members and all nine are mandatory — `schema_version` plus the eight sections. Sections may be empty objects, but they must be present.

The engine reports them one at a time, in a fixed order, which makes the requirement discoverable by bisection:

Listing C.3: Required-member discovery, one probe per line

```
{}                                                missing required field `schema_version`  $.
     schema_version
{"schema_version":1}                              missing required field `projectile`      $.
     projectile
{... "projectile":{}}                             missing required field `rifle`           $.
     rifle
{... "rifle":{}}                                  missing required field `shot`            $.
     shot
{... "shot":{}}                                   missing required field `atmosphere`      $.
     atmosphere
{... "atmosphere":{}}                             missing required field `wind`            $.
     wind
{... "wind":{}}                                   missing required field `solver`          $.
     solver
{... "solver":{}}                                 missing required field `effects`         $.
     effects
{... "effects":{}}                                missing required field `sampling`        $.
     sampling
{... "sampling":{}}                               missing required field `mass_kg`         $.
     projectile.mass_kg
```

Continuing the same bisection inside the sections finds exactly six required leaf fields:

Listing C.4: Required leaf fields, discovered the same way

```
+ projectile.mass_kg                missing required field `diameter_m`
+ projectile.diameter_m             missing required field `drag_model`
+ projectile.drag_model             missing required field `ballistic_coefficient`
+ projectile.ballistic_coefficient  missing required field `muzzle_velocity_mps`
+ rifle.muzzle_velocity_mps         missing required field `max_range_m`
+ shot.max_range_m                  ok, samples=31
```

That final line is the smallest request that solves. Written out:

Listing C.5: The minimal valid request

```
{"schema_version":1,
 "projectile":{"mass_kg":0.01134,"diameter_m":0.00782,
               "drag_model":"G7","ballistic_coefficient":0.243},
 "rifle":{"muzzle_velocity_mps":823.0},
 "shot":{"max_range_m":300.0},
 "atmosphere":{},"wind":{},"solver":{},"effects":{},"sampling":{}}
```

Thirty-one samples came back, because an empty `sampling` object defaults `interval_m` to 10 and 300 metres at 10-metre spacing is thirty-one points including both ends.

C.2.1 Three rules that hold everywhere

1. **Every object rejects unknown fields.** A typo is `unknown_field` with a JSON path, not a silently ignored key.
2. **Numbers must be finite.** No NaN, no infinity.
3. **Explicit `null` is not the same as omission.** It is `invalid_value: {"mass_kg":null}` gives `expected a number` at `$.projectile.mass_kg`.

SI at the wire

Every dimensional field carries its unit in the field name: `_kg`, `_m`, `_mps`, `_j`, `_s`, `_pa`, `_k`, `_rad`. There is no unit selector; the `--units` flag has no effect on this subcommand. `relative_humidity` is the one exception to the naming rule — it is a dimensionless *fraction* from 0 to 1, not a percentage.

C.3 Request field reference

The tables below give, for each field: whether it is required, its default when omitted, and whether the engine *echoes* it back in `resolved_request`. That last column matters more than it looks; see Section C.6.2.

C.3.1 `projectile`

The accepted set is `G1`, `G2`, `G5`, `G6`, `G7`, `G8`, `GI`, `GS`, and `RA4` — every reference family the engine's own drag tables carry. It was four names when this appendix was first written, and the protocol refused the other five outright. MBA-1442 widened it, and 0.33.1 shipped the widening: the wire list is now

Table C.2: `$.projectile`

Field	Req	Default and notes
`mass_kg`	yes	—
`diameter_m`	yes	—
`drag_model`	yes	one of nine reference families; see below
`ballistic_coefficient`	yes	for the named reference model
`length_m`	no	estimated from mass and diameter; emits `estimated_projectile_length`

the engine list, so a caller no longer has to discover that solve-json is a narrower door into the same tables than the CLI is.

Widening a set is the one kind of growth where "accepted" and "honoured" can quietly part company, so it is worth showing that the five new names are not aliases of the old four. Below, one request is solved nine times with nothing changed but the wire name — same 11.34 g bullet, same 823 m/s, same 91.44 m zero, same still air, same 1,000 m. The coefficient is held at 0.243 throughout, which is only physically meaningful for `G7`; the point of the run is that the name selects a different table, not that these are nine plausible descriptions of one bullet.

Listing C.6: Nine names, nine trajectories — `time_of_flight_s` and `terminal_speed_mps`, engine 0.37.0

```
G1    2.780379253567015    233.13540157780926
G2    1.9659154844043625   319.57264638360175
G5    2.216691951062771    296.4504142850786
G6    2.1453240282452186   296.0910661178944
G7    1.9621807353615615   323.89078933844905
G8    2.0821299308963312   304.05095375594897
GI    2.7616030992370844   234.41322747813592
GS    4.320993712448362    120.29484017222198
RA4   2.5885317338628893   251.5552131695923
```

The list is still closed, and the names are still case-sensitive. A name outside it is refused rather than coerced, and the message enumerates the current set, which makes the envelope itself the authority on what this engine accepts:

Listing C.7: A drag model the protocol does not carry — engine 0.37.0

```
{"code":"invalid_value",
 "message":"invalid value `G3`; expected one of G1, G2, G5, G6, G7, G8, GI, GS, RA4",
 "path":"$.projectile.drag_model","line":null,"column":null}
```

Do not hard-code this list

An enum that grew once can grow again — Section C.8 records that v1 explicitly reserves the right, and this section is the reason that rule exists. A client that pins its own copy of the nine names will reject a load the engine would have solved, and it will do so on the client's word rather than the engine's. Send the name through and let the `invalid_value` envelope be the refusal; its message names the set the engine in front of you actually has.

When `length_m` is omitted, the estimate appears in the assumption message and **not** in `resolved_request.projectile`:

Listing C.8: An estimated length is reported, not materialised

```
{"code":"estimated_projectile_length",
 "message":"Projectile length was omitted; the engine estimated 0.031066768041 m from mass and
     diameter.",
 "path":"$.projectile.length_m"}
```

C.3.2 `rifle`

Table C.3: `$.rifle`

Field	Req	Default	Notes
`muzzle_velocity_mps`	yes	—	
`sight_height_m`	no	`0.05`	above the bore
`muzzle_height_m`	no	`0`	above the ground datum
`twist_rate_m_per_turn`	no	`0.3048`	one turn in 12 inches
`twist_direction`	no	`"right"`	`"left"` or `"right"`
`sight_offset_lateral_m`	no	`0`	positive = sight right of bore; echoed when supplied

Table C.4: `$.shot`

Field	Req	Default	Notes
`max_range_m`	yes	—	requested termination distance
`zero_distance_m`	no	absent	solve elevation for this zero
`muzzle_angle_rad`	no	`0`	supply elevation directly
`aim_azimuth_rad`	no	`0`	horizontal aim offset
`shot_azimuth_rad`	no	`0`	compass bearing; 0 is north
`shooting_angle_rad`	no	`0`	uphill/downhill line of sight
`cant_angle_rad`	no	`0`	clockwise positive
`target_height_m`	no	`0`	**absolute** height above the ground datum
`ground_threshold_m`	no	`-100`	stop below this height
`zero_poi_up_m`	no	`0`	Kestrel “zero height”; echoed when supplied
`zero_poi_right_m`	no	`0`	Kestrel “zero offset”; echoed when supplied
`drops_reference`	no	`"los"`	`"los"` or `"target"`; echoed when supplied

C.3.3 shot

`zero_distance_m` and `muzzle_angle_rad` are no longer exclusive

Through 0.32.0 the two fields were alternatives: supplying both was a `conflicting_fields` refusal at `$.shot`, even though the engine's own protocol document already described a `zero_distance_elevation_not_resolved` *warning* for the case. The warning is what ships now. The combination is accepted, exit 0, and the envelope says plainly which of the two won:

Listing C.9: Both supplied — a warning, not a refusal, engine 0.37.0

```
{"code":"zero_distance_elevation_not_resolved",
 "message":"muzzle_angle_rad was supplied together with zero_distance_m: the elevation was
     used directly and the elevation search did not run. zero_distance_m is retained in
     resolved_request and still affects the windage-convergence bias (sight_offset_lateral_m
      / zero_poi_right_m), summary.equivalent_horizontal_range_m, and downrange wind-
     coverage validation.",
 "path":"$.shot.zero_distance_m"}
```

The change is not a loosening for its own sake; it is what makes the resolved request replayable. A zero-distance solve reports *both* the zero the caller asked for and the `muzzle_angle_rad` the search converged on, so its echo necessarily carries both, and a rule that refused both on the way in would have made the engine's own output illegal as input. Accepting the pair with `muzzle_angle_rad` winning outright reproduces the original angle bit for bit rather than re-converging on it.

Read the warning's second sentence carefully before treating `zero_distance_m` as decorative in that case. The elevation search does not run, but the zero distance is not inert: it still sets the windage convergence range that `sight_offset_lateral_m` and `zero_poi_right_m` bias, it still gates `summary.equivalent_horizontal_range_m`, and it still widens the range a segmented wind deck has to cover before `partial_wind_coverage` fires.

The Ballistics Lab never sends the pair — its domain model still treats the two as alternatives one layer earlier, at `domain.lat:391–393`, and refuses without a process spawn. That is now a Lab rule rather than an engine rule, and a Lab user meets it in the same words either way.

`target_height_m` is absolute, not relative to the muzzle

Raising `muzzle_height_m` without raising `target_height_m` asks the solver to zero a rifle held 1.5 m up onto a target sitting on the ground, and the elevation search will not converge:

```
{"code":"solve_failed",
 "message":"Zero angle did not converge: residual height error too large (target not
     reachable / not bracketed)",
 "path":null,"line":null,"column":null}
```

If you raise the muzzle, raise the target.

C.3.4 `atmosphere`

Table C.5: `$.atmosphere`

Field	Req	Default	Notes
`altitude_m`	no	`0`	station altitude
`temperature_k`	no	ICAO at `altitude_m`	authoritative when present
`pressure_pa`	no	ICAO at `altitude_m`	station pressure unless declared QNH
`pressure_reference`	no	`"absolute"`	or `"qnh"`; echoed when supplied
`relative_humidity`	no	`0.5`	fraction 0 to 1
`latitude_rad`	no	absent	**required** when `effects.coriolis` is true

Omitting temperature and pressure is not silent. Two distinct assumption codes name the ICAO substitution and quote the numbers to twelve decimal places:

Listing C.10: The atmosphere assumptions from the minimal request

```
{"code":"icao_standard_temperature",
 "message":"Station temperature was omitted; ICAO standard temperature at 0.000000 m is 288.15000
     0000000 K.",
 "path":"$.atmosphere.temperature_k"}
{"code":"icao_standard_pressure",
 "message":"Station pressure was omitted; ICAO standard pressure at 0.000000 m is 101325.00000000
     0000 Pa.",
 "path":"$.atmosphere.pressure_pa"}
```

Requesting Coriolis without a latitude is a cross-field validation failure with the path pointing at the *missing* field, not at the effect that needed it:

```
{"code":"invalid_value",
 "message":"latitude_rad is required when the Coriolis effect is enabled",
 "path":"$.atmosphere.latitude_rad","line":null,"column":null}
```

`pressure_pa` is *station* pressure — what a barometer sitting next to the rifle reads. A weather report is usually not that. METAR altimeter settings, aviation QNH, and the "pressure" most phone apps show are sea-level-corrected: the reading is what the barometer *would* say if it were carried down to sea level along a standard column of air. At 250 m that correction is worth about 3 kPa, which is roughly 3% of air density, and feeding it in raw overstates the density and understates the drop.

`atmosphere.pressure_reference` is how you say which one you have. It takes `"absolute"` (the default, and what an omitted field has always meant) or `"qnh"`. This appendix once recorded the field as documented but unshipped — both 0.30.1 and 0.32.0 rejected it as `unknown_field`. It ships now, and it does the reduction for you:

Listing C.11: 101,325 Pa declared as QNH, at 250 m — engine 0.37.0

```
{"code":"qnh_reduced_to_station_pressure",
 "message":"Station pressure was declared as a QNH (sea-level-corrected altimeter setting) of 101
    325.000000 Pa; reduced to station pressure 98357.651659113268 Pa at 250.000000 m via the
    ICAO inverse-barometric formula.",
 "path":"$.atmosphere.pressure_pa"}
```

The reduction is an *assumption*, not a warning, because nothing about the request is questionable — the engine is recording a value it derived, in the same voice it uses for an ICAO standard pressure. The echo then carries the reduced number, not the one you sent: `pressure_pa` comes back as `98357.65165911327` with `pressure_reference: "qnh"` alongside it. That pairing matters when you replay a resolved request; see Section C.6.2.

Declaring `"absolute"` explicitly changes nothing but the echo, and an unrecognised mode is refused rather than treated as absolute, so a typo can never silently hand you a 3% density error:

Listing C.12: An unrecognised pressure reference — engine 0.37.0

```
{"code":"invalid_value","message":"invalid value `msl`; expected one of absolute, qnh",
 "path":"$.atmosphere.pressure_reference","line":null,"column":null}
```

The field has no effect when `pressure_pa` is omitted: an absent pressure resolves to the ICAO standard *station* pressure for the requested altitude either way, and there is nothing to reduce.

C.3.5 `wind`

The `wind` object has exactly two legal shapes, and an empty object is a third: still air.

Constant wind takes `speed_mps` and `direction_from_rad`, which must be supplied *together*, plus an optional `vertical_speed_mps`. All three failure modes were reproduced:

Listing C.13: Half a constant wind is not a wind — three probes, summarised

```
speed_mps alone            -> conflicting_fields: speed_mps and direction_from_rad must be
     supplied together
direction_from_rad alone   -> conflicting_fields: speed_mps and direction_from_rad must be
     supplied together
vertical_speed_mps alone   -> conflicting_fields: vertical_speed_mps requires speed_mps and
     direction_from_rad
```

Segmented wind takes a `segments` array, whose `until_distance_m` boundaries must strictly increase. It conflicts with all three constant fields.

Listing C.14: A two-segment wind deck, as sent

```
"wind": {
  "segments": [
    {"until_distance_m": 400.0, "speed_mps": 4.4704,
     "direction_from_rad": 1.5707963267948966},
    {"until_distance_m": 800.0, "speed_mps": 2.2352,
     "direction_from_rad": 4.71238898038469}
  ]
}
```

Listing C.15: … and as echoed, with the vertical component materialised

```
"wind": {
  "segments": [
    {"until_distance_m": 400.0, "speed_mps": 4.4704,
     "direction_from_rad": 1.5707963267948966, "vertical_speed_mps": 0.0},
    {"until_distance_m": 800.0, "speed_mps": 2.2352,
     "direction_from_rad": 4.71238898038469, "vertical_speed_mps": 0.0}
  ]
}
```

Coverage is checked against the farther of `max_range_m` and `zero_distance_m`, because the zero trial flies through the same air. A deck that ends short does not fail — it warns, and the air beyond the last boundary is calm:

Listing C.16: The `partial_wind_coverage` warning, verbatim

```
{"code":"partial_wind_coverage",
 "message":"Segmented wind ends at 800.000000000000 m; wind is calm from that boundary through
     the required 1000.000000000000 m solve/zero range.",
 "path":"$.wind.segments"}
```

`wind_reference` accepts `"shooter"` (the default) or `"compass"`. Compass bearings need a shot azimuth to be re-referenced against, and the engine refuses rather than guessing:

```
{"code":"conflicting_fields",
 "message":"wind_reference \"compass\" requires shot.shot_azimuth_rad (earth-fixed bearings are
     re-referenced against the shot azimuth; omitting it would silently treat bearings as shooter
     -relative)",
 "path":"$.wind.wind_reference","line":null,"column":null}
```

Wind-from

Wind direction is the direction the wind is coming *from*, measured clockwise from the shooter's line of fire. Zero radians is a headwind, $\pi/2$ is a wind from the shooter's right, π is a tailwind. A wind from the right pushes the bullet left, so windage goes negative.

C.3.6 `solver`, `effects`, `sampling`

Table C.6: The last three sections

Field	Default	Notes
`solver.method`	`"rk45"`	`rk45`, `rk4`, `euler`
`solver.time_step_s`	`0.001`	fixed step for `rk4`/`euler`; accepted and *ignored* by `rk45`, which warns
`effects.magnus`	`false`	warns `experimental_effect` when true
`effects.coriolis`	`false`	requires `atmosphere.latitude_rad`
`effects.enhanced_spin_drift`	`false`	warns when true; cannot be true with `magnus`
`effects.wind_shear_model`	absent (= `"none"`)	`none`, `logarithmic`, `power_law`, `ekman_spiral` (alias `ekman`); echoed when supplied
`sampling.interval_m`	`10`	at most 10,000 samples

The Magnus / enhanced-spin-drift conflict is a hard error rather than a silent suppression, so the resolved request can never misstate which physics ran:

```
{"code":"conflicting_fields",
 "message":"magnus and enhanced_spin_drift cannot both be enabled",
 "path":"$.effects","line":null,"column":null}
```

`effects` is no longer three booleans. `wind_shear_model` was 0.36.0's headline, and it is the odd member of the section: a string rather than a flag, and a choice of physics rather than a switch on some. It makes the request's wind a function of height above the ground instead of a single value used at every point of the flight.

An omitted field means no altitude dependence at all, byte-identical to every request written before the field existed. An explicit `"none"` solves identically and differs only in the echo. `"logarithmic"` applies the boundary-layer profile $\ln(z/z_0)/\ln(z_{\text{ref}}/z_0)$; `"power_law"` applies $(z/z_{\text{ref}})^{1/7}$. Both take the request's wind as the speed at the 10 m meteorological reference height — the height a reported wind is measured at, and the value a flat-fire solver assumes uniformly for the whole flight.

Shear shows nothing until the bullet climbs

The profile is clamped at 1: it scales the wind *up* above 10 m and never down below it. That is deliberate — flat fire spends its whole flight under the reference height, and a profile allowed to shrink there would quietly disagree with the uniform-wind answer every other solver gives for the same inputs. The consequence is that on a short shot the field looks broken. Below are four solves of one 4.4704 m/s full-value wind, each stretched further out and each zeroed at its own range so the trajectory actually lofts, with the terminal windage in metres:

Listing C.17: Apex governs whether shear is visible at all — engine 0.37.0

```
range   apex_m    none          logarithmic   power_law
1000     4.7553   -3.337879668  -3.337879668  -3.337879668
1500    18.3128   -8.289423559  -8.835284821  -8.761559748
2000    45.8415  -14.714462080 -17.407397907 -17.170019442
2500    92.9236  -22.832459113 -29.337779602 -29.027594988
```

At 1,000 m the three agree to the last digit, because the whole trajectory stays below 10 m and the clamp holds the ratio at 1. Past that the models separate: by 2,500 m the logarithmic profile adds 6.5 m to an unsheared 22.8, nearly 30% more drift, and the two profiles differ from each other by about 0.3 m. If you enable shear on a flat-fire test case and see no change, the model is working; the trajectory is simply too low to care.

`"ekman_spiral"` is accepted for parity with the CLI's model names but this solve path has no near-ground closed form for it, so it leaves the wind at the operative value and says so with a `wind_shear_model_not_modeled` warning rather than passing silently. The alias `"ekman"` is accepted and echoed in its canonical spelling: `"ekman"` in, `"ekman_spiral"` out. An unrecognised name is refused outright rather than falling back to no shear, on the same reasoning as `pressure_reference` — a typo must never return unsheared numbers that look sheared:

Listing C.18: An unrecognised shear model — engine 0.37.0

```
{"code":"invalid_value",
 "message":"invalid value `log`; expected one of none, logarithmic, power_law, ekman_spiral,
     ekman",
 "path":"$.effects.wind_shear_model","line":null,"column":null}
```

The Ballistics Lab does not send this field, and its envelope validator rejects unknown keys inside `resolved_request` — so, exactly as with `reticle_hold` below, teaching the Lab about shear is a two-sided change and not a matter of passing one more string through.

C.3.7 The sample ceiling

The sampling limit is evaluated against the range the solve *actually reached*, and the service errors rather than thinning the output. The boundary is exact: 10,000 samples pass, 10,001 do not.

Listing C.19: Probing the ceiling on a 1,000 m solve

```
interval 0.1000100010001 m   exit 0   samples = 10000
interval 0.10001 m           exit 3   trajectory observation limit of 10000 exceeded (would
     produce 10001)
interval 0.1 m               exit 3   trajectory observation limit of 10000 exceeded (would
     produce 10001)
```

The error names the count it would have produced and points at `$.sampling.interval_m`, which is the field you can actually change.

C.3.8 The optional `reticle` block

A request may carry a top-level `reticle` object. When it does — and only then — the success envelope gains a top-level `reticle_hold`. All three of `range_m`, `magnification`, and `description` are required and no other key is accepted; `description` is the same document `ballistics reticle generate -o json` emits.

Listing C.20: A reticle hold, captured

```
"reticle_hold": {
  "range_m": 600.0, "magnification": 5.0,
  "down_mil": 5.583643794078713, "right_mil": 0.0, "mark_scale": 2.0,
  "nearest_mark_index": 1, "nearest_mark_label": "600",
  "nearest_mark_distance_mil": 1.5836437940787134, "off_reticle": true
}
```

The Lab cannot consume a reticle response

The Lab's envelope validator demands *exactly* the eight documented top-level success keys with no optional extras (`process_backend.lat:598–605`). A response carrying `reticle_hold` would be rejected as `invalid_engine_response`. This is not a defect — the Lab never sends a `reticle` block — but it is the shape of the fail-closed rule, and it means adding reticle support to the Lab is a two-sided change.

A supplied `reticle` is also echoed onto `resolved_request.reticle`, description and all. This appendix once recorded the echo as promised by the protocol document but absent from both shipped binaries, and pending for 0.33.0; 0.33.0 landed four releases back, and a probe on 0.37.0 finds `reticle` among the resolved request's members carrying the whole mark list verbatim. It belongs there for the same reason the rest of Section C.6.2 does: a hold-point is only reproducible if the reticle it was taken against travels with it.

C.4 A complete worked exchange

Here is one full request and its full response, captured together. The load is the book's reference match bullet at 1,000 metres with a 100-yard zero, a 10 mph wind from the right, and Coriolis on at 45 degrees north. Sampling is set to 250 metres so the whole exchange fits on the page.

Listing C.21: `req.json` — the complete request

```
{
  "schema_version": 1,
  "projectile": {
    "mass_kg": 0.01134,
    "diameter_m": 0.00782,
    "length_m": 0.0315,
    "drag_model": "G7",
    "ballistic_coefficient": 0.243
  },
  "rifle": {
    "muzzle_velocity_mps": 823.0,
    "sight_height_m": 0.0381,
    "muzzle_height_m": 0.0,
    "twist_rate_m_per_turn": 0.2032,
    "twist_direction": "right"
  },
  "shot": {
    "max_range_m": 1000.0,
    "zero_distance_m": 91.44
  },
  "atmosphere": {
    "altitude_m": 250.0,
    "temperature_k": 288.15,
    "pressure_pa": 101325.0,
    "relative_humidity": 0.5,
    "latitude_rad": 0.7853981633974483
  },
  "wind": {
    "speed_mps": 4.4704,
    "direction_from_rad": 1.5707963267948966
  },
  "solver": { "method": "rk45" },
  "effects": { "coriolis": true },
  "sampling": { "interval_m": 250.0 }
}
```

Listing C.22: The invocation

```
$ ballistics solve-json < req.json > resp.json
$ echo $?
0
$ wc -c resp.json
    3391 resp.json
```

The response is one compact line. Pretty-printed for the page, and split into its parts:

Listing C.23: The envelope head and the resolved request


```
{
  "schema_version": 1,
  "engine_version": "0.32.0",
  "status": "ok",
  "resolved_request": {
    "schema_version": 1,
    "projectile": {"mass_kg": 0.01134, "diameter_m": 0.00782, "length_m": 0.0315,
                   "drag_model": "G7", "ballistic_coefficient": 0.243},
    "rifle": {"muzzle_velocity_mps": 823.0, "sight_height_m": 0.0381,
              "muzzle_height_m": 0.0, "twist_rate_m_per_turn": 0.2032,
              "twist_direction": "right"},
    "shot": {"max_range_m": 1000.0, "zero_distance_m": 91.44,
             "muzzle_angle_rad": 0.0006958007812500001,
             "aim_azimuth_rad": 0.0, "shot_azimuth_rad": 0.0,
             "shooting_angle_rad": 0.0, "cant_angle_rad": 0.0,
             "target_height_m": 0.0, "ground_threshold_m": -100.0},
    "atmosphere": {"altitude_m": 250.0, "temperature_k": 288.15,
                   "pressure_pa": 101325.0, "relative_humidity": 0.5,
                   "latitude_rad": 0.7853981633974483},
    "wind": {"speed_mps": 4.4704, "direction_from_rad": 1.5707963267948966,
             "vertical_speed_mps": 0.0},
    "solver": {"method": "rk45", "time_step_s": 0.001},
    "effects": {"magnus": false, "coriolis": true, "enhanced_spin_drift": false},
    "sampling": {"interval_m": 250.0}
  },
```


The one number in that block the solver had to *compute* is `muzzle_angle_rad: 0.0006958007812500001` — the elevation it found for a 91.44 m zero. That is the whole point of the resolved request: it is a *complete* description of the run, not an echo of the input.

This request zeroes to the ground, and the BALLISTICS LAB does not

`req.json` above is hand-written and omits `target_height_m`, so the resolved request comes back with `target_height_m: 0.0` and the elevation was solved onto a target sitting in the dirt 91.44 m away. That is the engine behaving exactly as documented. It is *not* what the BALLISTICS LAB sends: every Lab request carries the field, set to the height of the line of sight (Section C.5).

The difference is not cosmetic, and it is worth seeing on this exact exchange. Add `"target_height_m": 0.0381` to `req.json` — the sight height, with the muzzle on the ground — and re-run it. The bore angle rises from `0.0006958007812500001` to `0.00111236572265625` rad, the assumption list drops from ten entries to nine, `maximum_height_m` goes from `0.016219219303429212` to `0.04112147563296846`, and the 1,000 m sample's `drop_m` goes from `13.452161435887655` to `13.035429622842196`. That is 0.4167318130454 59 m of drop removed, against a sight line subtended over the shot of $0.0381 \times 1000/91.44$, which is exactly $5/12 = 0.41666\ldots$ m — agreement to four decimal places. Read the summary and samples that follow as a demonstration of the ground datum, not as figures the BALLISTICS LAB would print.

Listing C.24: The assumptions — ten, all defaults

```
"assumptions": [
  {"code": "default_applied", "message": "Aim azimuth defaulted to 0 rad.",
   "path": "$.shot.aim_azimuth_rad"},
  {"code": "default_applied", "message": "Shot azimuth defaulted to 0 rad (north).",
   "path": "$.shot.shot_azimuth_rad"},
  {"code": "default_applied", "message": "Shooting angle defaulted to 0 rad.",
   "path": "$.shot.shooting_angle_rad"},
  {"code": "default_applied", "message": "Cant angle defaulted to 0 rad.",
   "path": "$.shot.cant_angle_rad"},
  {"code": "default_applied",
   "message": "Target height defaulted to 0 m above the local ground datum.",
   "path": "$.shot.target_height_m"},
  {"code": "default_applied", "message": "Ground threshold defaulted to -100 m.",
   "path": "$.shot.ground_threshold_m"},
  {"code": "default_applied",
   "message": "Constant vertical wind speed defaulted to 0 m/s.",
   "path": "$.wind.vertical_speed_mps"},
  {"code": "default_applied", "message": "Solver time step defaulted to 0.001 s.",
   "path": "$.solver.time_step_s"},
  {"code": "default_applied", "message": "Magnus effect defaulted to disabled.",
   "path": "$.effects.magnus"},
  {"code": "default_applied",
   "message": "Enhanced spin drift defaulted to disabled.",
   "path": "$.effects.enhanced_spin_drift"}
],
"warnings": [],
```

Listing C.25: The summary

```
"summary": {
  "actual_range_m": 1000.0,
  "maximum_height_m": 0.016219219303429212,
  "time_of_flight_s": 1.9621901907481842,
  "terminal_speed_mps": 323.89758274449713,
  "terminal_energy_j": 594.8376820908197,
  "stability_factor": 3.7937725097048576,
  "termination": "max_range"
},
```

Listing C.26: The samples — five rows on a 250 m grid

```
  "samples": [
    {"distance_m": 0.0, "time_s": 0.0, "speed_mps": 823.0,
     "energy_j": 3840.45543, "drop_m": 0.0381, "windage_m": 0.0,
     "mach": 2.4152756269212587, "flags": []},
    {"distance_m": 250.0, "time_s": 0.3360402350298456,
     "speed_mps": 672.8314981281616, "energy_j": 2566.8216150321,
     "drop_m": 0.38256767238348494, "windage_m": -0.14008525703887478,
     "mach": 1.9745729264324, "flags": []},
    {"distance_m": 500.0, "time_s": 0.7510429801374595,
     "speed_mps": 539.4176117528427, "energy_j": 1649.8076104580268,
     "drop_m": 2.1052848883852073, "windage_m": -0.623496188632351,
     "mach": 1.5830403528538461, "flags": []},
    {"distance_m": 750.0, "time_s": 1.276458574436444,
     "speed_mps": 419.5448819780584, "energy_j": 998.0215383258833,
     "drop_m": 5.985379338704224, "windage_m": -1.5883252940257553,
     "mach": 1.231247299928506, "flags": []},
    {"distance_m": 1000.0, "time_s": 1.9621901907481842,
     "speed_mps": 323.89758274449713, "energy_j": 594.8376820908197,
     "drop_m": 13.452161435887655, "windage_m": -3.2539818828329987,
     "mach": 0.9505491339264831,
     "flags": ["transonic", "subsonic", "terminal"]}
  ]
}
```

Three things to read out of those samples. The muzzle sample's `drop_m` is `0.0381` — exactly the sight height, because drop is measured down from the sight's horizontal reference and the bullet starts that far below it. `windage_m` is negative throughout, because the wind is from the right. And the final sample carries both `transonic` and `subsonic`, because at Mach 0.95 it is in the transonic band (0.8 to 1.2 inclusive) *and* below Mach 1.

C.5 What the Lab actually sends

The Lab builds its request from the domain model, reading only canonical SI fields, so display metadata cannot leak onto the wire. Here is the request for the shipped reference experiment, printed by the Lab's own module:

Listing C.27: Asking the Lab what it would send

```
$ ./clat examples/ballistics_lab/reference_experiment.lat --json
```

Listing C.28: The result, reformatted for the page

```
{"sampling":{"interval_m":9.1440000000000001},
 "rifle":{"twist_direction":"right",
          "muzzle_velocity_mps":822.96000000000004,
          "muzzle_height_m":0,
          "sight_height_m":0.038099999999999995,
          "twist_rate_m_per_turn":0.20319999999999999},
 "schema_version":1,
 "shot":{"max_range_m":914.39999999999998,
         "zero_distance_m":91.439999999999998,
         "target_height_m":0.038099999999999995},
 "atmosphere":{"altitude_m":250,
               "latitude_rad":0.78539816339744828,
               "relative_humidity":0.5,
               "pressure_pa":101325,
               "temperature_k":288.14999999999998},
 "wind":{"direction_from_rad":1.5707963267948966,"speed_mps":4.4703999999999997},
 "solver":{"method":"rk45"},
 "projectile":{"drag_model":"G7","diameter_m":0.0078231999999999989,
               "length_m":0.031495999999999996,
               "ballistic_coefficient":0.24299999999999999,
               "mass_kg":0.011339809249999999},
 "effects":{"coriolis":true}}
```

Every imperial input has become an exact SI value: 1.5 inches is 0.0381 m (spelled `0.038099999999999995` by a round-trip-safe float formatter), 1,000 yards is 914.4 m, 2,700 fps is 822.96 m/s, 10 mph is 4.4704 m/s.

One field on that wire is not a translation of anything the user typed. `target_height_m` appears in every request the Ballistics Lab builds, and the shipped reference experiment does not set it — `:show` reports `target height : default` for it, because the domain `Shot.target_height` really is `nil`. The field names the height above the local ground datum that the engine solves the zero *to*, and the engine's documented default for it is `0.0`: ground level. Nobody zeroes a rifle to the dirt, so `protocol_v1.lat:74–92` supplies the height of the line of sight instead — muzzle height plus sight height — whenever the experiment leaves the field unset. Here `muzzle_height_m` is 0 and the sight height is 1.5 inches, which is why `target_height_m` and `sight_height_m` print the same digits.

Omission is meaningful

Notice what is *not* there. There is no `muzzle_angle_rad`, no `cant_angle_rad`, no `time_step_s`, no `magnus`. The Lab's protocol module omits optional fields rather than serialising them as null, which is why a solve of this experiment produces exactly nine `default_applied` assumptions — one fewer than the ten in the hand-written exchange of Section C.4, whose request omits `target_height_m` and collects the engine's default for it. That is a deliberate contrast with the Lab's *persistence* format, which does the opposite and writes explicit nulls.

C.6 The response

C.6.1 Top level

A success envelope has exactly eight members: `schema_version` (always 1), `engine_version`, `status` (`"ok"`), `resolved_request`, `assumptions`, `warnings`, `summary`, and `samples` — plus `reticle_hold` when, and only when, the request carried a reticle.

C.6.2 `resolved_request`, and what replaying one means

`resolved_request` is a distinct type, not a replay of the input. Every documented default is materialised as a concrete value so the run is reproducible, and `temperature_k`, `pressure_pa`, and the effective `shot.muzzle_angle_rad` are always present.

This appendix used to carry a table here headed "fields the engine honours but whose names it does not echo". Five request fields — `rifle.sight_offset_lateral_m`, `shot.zero_poi_up_m`, `shot.zero_poi_right_m`, `shot.drops_reference`, and `wind.wind_reference` — changed the answer and left no trace of themselves in the echo, so an archived resolved request was not always enough to reproduce the run it described. That gap is closed. The 0.33.0 decision-support work needed to take a resolved request, perturb one input, and re-solve, which is impossible from a lossy echo; closing the gap was a precondition of the feature rather than a tidy-up. All five are echoed on 0.37.0, each present only when the raw request supplied it, so a request that never mentioned them still produces the byte-identical response it always did.

One request carrying all five, and the echo it comes back with:

Listing C.29: All five echoed — engine 0.37.0, one request, the members that matter

```
rifle       "sight_offset_lateral_m": 0.01
shot        "zero_poi_up_m": 0.0254, "zero_poi_right_m": 0.0254,
            "drops_reference": "target"
atmosphere  "pressure_reference": "qnh"
wind        "wind_reference": "compass"
```

Each still earns its place — these are not inert labels. Against an otherwise identical 1,000 m baseline (823 m/s, G7 0.243, 91.44 m zero, still air, whose `muzzle_angle_rad` is `0.0006958007812500001` and whose terminal `windage_m` is `0.0`), supplying one field at a time moves the numbers:

Listing C.30: One field at a time, against the baseline above — engine 0.37.0

```
sight_offset_lateral_m = 0.01
    terminal windage_m   0.0 -> 0.09936133026975426
zero_poi_up_m = 0.0254
    muzzle_angle_rad     0.0006958007812500001 -> 0.000973578559027778
zero_poi_right_m = 0.0254
    terminal windage_m   0.0 -> 0.27777778492226773
drops_reference = "target"
    output-mode toggle; this flat-fire baseline is unchanged
```

Two echoed fields are provenance, not inputs — do not replay them

The echo is complete now, but two of its members are statements about how a sibling value was *entered*, and the sibling has already been transformed by the time you read it. Replaying the resolved request verbatim applies the transform a second time:

- `atmosphere.pressure_reference: "qnh"` sits next to a `pressure_pa` that is *already* the reduced station pressure. Send the pair back and it is reduced again.
- `wind.wind_reference: "compass"` sits next to a `direction_from_rad` that is *already* converted to shooter-relative. Send the pair back and it is re-referenced against the shot azimuth again.

Both are visible in one round trip. Taking the resolved request from the five-field run above and feeding it straight back to `solve-json`:

Listing C.31: A verbatim replay is not a reproduction — engine 0.37.0

```
original  pressure_pa 98357.65165911327   direction_from_rad 0.7854
          windage_m -1.6575617863967538   drop_m 12.91472809861563
replay    pressure_pa 95477.20345319976   direction_from_rad 0.0
          windage_m  0.3782064218814722   drop_m 12.610857410738019
```

The engine's own resolved-to-request conversion drops exactly these two fields and states the rest absolutely, which is the behaviour to copy: a replay should say `pressure_pa` as absolute and `direction_from_rad` as shooter-relative, because that is what the resolved values already are. Every other member — `sight_offset_lateral_m`, the two `zero_poi` offsets, `drops_reference`, `wind_shear_model`, `reticle` — is carried straight across, because those name the physics that ran rather than the units it was typed in, and re-supplying them reproduces the same solve instead of compounding a conversion. `zero_distance_m` and `muzzle_angle_rad` both travel too; the angle wins, which is what reproduces the original elevation bit for bit rather than re-converging on it. The Ballistics Lab sends none of the five and replays nothing, so Lab artifacts are unaffected either way.

A consequence worth naming, because this appendix previously recorded the opposite: sending `"drops_reference": "los"` — the documented default — is no longer byte-identical to omitting it. The trajectory is identical, but the response gains `"drops_reference": "los"` inside `resolved_request.shot`. The protocol document always warned that an explicit default changes the bytes by adding an echo field; now that the echo field exists, the warning is simply true.

C.6.3 `assumptions` and `warnings`

Both are arrays of `{code, message, path}` objects emitted in deterministic request-field order. Assumptions record what the engine filled in; warnings record something you should know about the run you asked for.

Table C.7: Every assumption and warning code stable in v1, as of 0.37.0

Code	Emitted when
Assumptions	
`default_applied`	any documented default was substituted
`icao_standard_temperature`	`temperature_k` omitted
`icao_standard_pressure`	`pressure_pa` omitted
`qnh_reduced_to_station_pressure`	`pressure_reference` was `"qnh"`, so `pressure_pa` was reduced to station pressure
`estimated_projectile_length`	`length_m` omitted
Warnings	
`partial_wind_coverage`	a segmented deck ends short of the solve range
`rk45_time_step_ignored`	`time_step_s` supplied with `rk45`
`experimental_effect`	`magnus` or `enhanced_spin_drift` enabled
`zero_distance_elevation_not_resolved`	`muzzle_angle_rad` supplied together with `zero_distance_m`, so no elevation search ran
`wind_shear_model_not_modeled`	an accepted `wind_shear_model` this solve path has no profile for, i.e. `ekman_spiral`
`bc5d_drag_model_coerced`	a `corrections.bc5d_table_path` request whose drag model is outside the table's G1/G7 planes

Four of those — one assumption and three warnings — arrived after this appendix was first written, and the shape of the arrivals is worth noticing. None of them is a new failure: each one names a request the engine now *accepts* where it once refused, or a corner where it does less than the name suggests and says so rather than passing quietly. That is the direction v1 grows in (Section C.8), and it is why a consumer must treat an unrecognised notice code as information rather than as an error.

`corrections` is not covered here

`bc5d_drag_model_coerced` belongs to an optional top-level `corrections` block, whose one field names a local BC5D table file. From it the engine generates a velocity-keyed BC segment ladder for the load and solves — zero search and trajectory both — against that schedule instead of the request's single coefficient. It is echoed at `resolved_request.corrections` when supplied. This appendix does not document it: it needs a table file, which puts it outside the "no files and no network" contract everything else here rests on, and the Ballistics Lab never sends it. It is listed in the notice table only so that table stays honest about the codes a client can receive.

A request that triggers three warnings at once — a short wind deck, an explicit RK45 time step, and enhanced spin drift — returns them in field order:

Listing C.32: Three warnings, captured verbatim

```
"warnings": [
  {"code": "partial_wind_coverage",
   "message": "Segmented wind ends at 800.000000000000 m; wind is calm from that boundary through
      the required 1000.000000000000 m solve/zero range.",
   "path": "$.wind.segments"},
  {"code": "rk45_time_step_ignored",
   "message": "Adaptive RK45 owns its time step; the explicitly supplied fixed time step is
     retained in resolved_request but ignored by integration.",
   "path": "$.solver.time_step_s"},
  {"code": "experimental_effect",
   "message": "The enhanced spin drift effect is an opt-in experimental v1 model.",
   "path": "$.effects.enhanced_spin_drift"}
]
```

C.6.4 `summary`

Table C.8: `$.summary`

Field	Presence	Meaning
`actual_range_m`	always	distance actually reached
`maximum_height_m`	always	greatest *world-vertical* height above the same datum as `muzzle_height_m`
`time_of_flight_s`	always	
`terminal_speed_mps`	always	
`terminal_energy_j`	always	
`stability_factor`	when computable	muzzle gyroscopic Sg
`spin_drift_m`	only with `enhanced_spin_drift`	signed, positive right, excludes wind
`equivalent_horizontal_range_m`	inclined shots with a zero	angular-match “shoot-to” range
`termination`	always	`max_range`, `ground_threshold`, `time_limit`, `velocity_floor`

`spin_drift_m` is already inside `windage_m`

Do not add them. On a calm-air 1,000 m run with enhanced spin drift enabled, the two numbers are *equal to the last digit*:

```
summary.spin_drift_m        = 0.5443583308953567
samples[-1].windage_m       = 0.5443583308953567
```

Spin drift is folded into the lateral solution, not reported alongside it. The summary field exists so you can see how much of the windage it accounts for.

C.6.5 `samples`

Each sample is `{distance_m, time_s, speed_mps, energy_j, drop_m, windage_m, mach, flags}`.

- `drop_m` is **positive below** the sight's horizontal reference. At the *muzzle* it equals the sight height, because the bore starts that far below the sight line. At the **zero distance it is zero** — provided the request says what height the zero is solved to. `shot.target_height_m` carries that datum, and the engine defaults it to `0.0`, the ground; a request that omits it zeroes to the dirt and every sample then carries a sight height $\times\, d/d_{\text{zero}}$ bias. The BALLISTICS LAB always sends the height of the line of sight (§C.5), so its cards read zero at the zero. Section D.6 demonstrates both halves.

- `windage_m` is **positive to the shooter's right**.

- `flags` may contain `transonic` (Mach 0.8 to 1.2 inclusive), `subsonic` (below Mach 1 — so both appear between 0.8 and 1.0), `terminal`, and `ground_threshold`.

- **The terminal sample appears exactly once**, even when it falls off the interval grid.

C.7 The error envelope

An error response has exactly three top-level members and a fixed error shape:

Listing C.33: The error envelope, always this shape

```
{"schema_version":1,"status":"error",
 "error":{"code":"...","message":"...",
          "path":"$..."|null,"line":N|null,"column":N|null}}
```

`path` and the `(line, column)` pair are mutually exclusive. A structurally valid document that fails validation gets a JSON path and null line/column. Malformed JSON gets 1-based line and column and a null path. Location-free failures get null for all three.

Table C.9 is every code, each with the envelope actually captured from the shipped binary.

Table C.9: Every solve-json error code, captured live

Code	Exit	Captured message
`invalid_json`	2	`expected value at line 1 column 1`
`invalid_json`	2	`trailing comma at line 1 column 160`
`unsupported_schema_version`	2	`unsupported schema_version 2; expected 1`
`unknown_field`	2	`unknown field ‘foo‘`
`missing_field`	2	`missing required field ‘schema_version‘`
`invalid_value`	2	`invalid value ‘G3‘; expected one of G1, G2, G5, G6, G7, G8, GI, GS, RA4`
`invalid_value`	2	`expected a number` (explicit null)
`invalid_value`	2	`invalid value ‘msl‘; expected one of absolute, qnh`
`invalid_value`	2	`latitude_rad is required when the Coriolis effect is enabled`
`conflicting_fields`	2	`magnus and enhanced_spin_drift cannot both be enabled`
`conflicting_fields`	2	`speed_mps and direction_from_rad must be supplied together`
`resource_limit`	3	`trajectory observation limit of 10000 exceeded (would produce 10001)`
`resource_limit`	3	`solve-json input exceeds the 1048576-byte limit`
`solve_failed`	3	`Zero angle did not converge: residual height error too large (target not reachable / not bracketed)`
`io_error`	1	documented; not reproduced
`internal_error`	1	documented; a contained panic. Payloads and backtraces are never placed in the envelope

Branch on `code`, never on `message`

The human-readable `message` is diagnostic and explicitly outside the compatibility surface. Two of the messages in that table are near-identical in code but differ in wording between error families, and all of them are free to change. A client branches on `code`, and uses `path`, `line`, and `column` to point at the offending input.

C.8 Compatibility, and what the Lab adds

C.8.1 The engine's promises

Version 1's invariants hold absolutely. Removing a field, changing a unit or a sign, renaming an enum value, or changing a field's meaning all require v2. v1 grows only by addition, under three rules:

1. A new *request* field must be optional and default to the exact pre-existing behaviour. `zero_poi_up_m`, `zero_poi_right_m`, `sight_offset_lateral_m`, `drops_reference`, `wind_reference`, `pressure_reference`, and `wind_shear_model` all arrived this way.

2. A new *response* field must be omitted whenever its feature is inactive, so an unexercising response is byte-identical to one produced before the field existed. `reticle_hold` is the clearest case: it appears only when the request carried a `reticle` block.

3. An existing *request* enum may gain a new accepted value when that value names newly supported behaviour. The new value is opt-in and may be echoed; requests using the older values keep identical behaviour and identical output. The nine drag models of Table C.2 are the worked example — `G2`, `G5`, `GI`, `GS`, and `RA4` were added to a set of four under this rule.

The third rule is the one that dates a document. A reader who pins the enum lists out of an appendix like this one has pinned an engine version rather than the contract — which is exactly what happened to this appendix's own drag-model section between 0.32.0 and 0.37.0. The same logic applies to responses: a consumer that wants to parse across engine versions must tolerate response fields it does not recognise rather than reject them. One documented exception to rule 2 is worth knowing about before you write a strict decoder: `summary.equivalent_horizontal_range_m` arrives with no opt-in field of its own. It appears whenever a request carries both a non-zero `shooting_angle_rad` and a `zero_distance_m` — an incline-corrected shoot-to range needs a look angle to correct and a zero to correct against, and it is meaningful exactly when it has both. Supply the incline without the zero, or the zero without the incline, and the member is absent, so flat-fire responses are unaffected and the rule holds everywhere it was already relied on.

Not part of the contract: member order, whitespace, the human-readable text of messages, `engine_version`, and the exact decimal spelling of floats. Regression comparison uses a tolerance rather than string equality: $|a - e| \le 10^{-10} + 10^{-9}\max(|a|, |e|)$.

C.8.2 What the Lab requires on top

The Lab is stricter than the protocol demands, deliberately. Anything the protocol did not promise is rejected by its decoder, under the single code `invalid_engine_response`:

- **Exactly one JSON value on stdout.** A banner before it, a trailer after it, two concatenated envelopes, or a truncated one all fail at the parse.
- **No stderr on success.** `status: "ok"` with anything on stderr is a hard failure.
- **Exit code must match the error code.**
- **No unknown fields anywhere**, including inside `resolved_request`.
- **The resolved request must be internally consistent**: altitude in range, humidity a fraction, ground threshold below muzzle height, Coriolis implies latitude, Magnus and enhanced spin drift not both true.
- **Exactly one terminal sample**, and its distance, time, speed, and energy must agree with the summary within $10^{-10} + 10^{-9} \max(|a|, |b|)$.
- **At most 10,000 samples.**

Why be stricter than the contract?

Because the Lab's whole claim is that a number it prints came from a run it can describe. A response it only half-understood would break that claim quietly. The failure modes are catalogued, with the exact message each produces, in Section D.3.

Appendix D

Troubleshooting

Everything in this appendix was made to happen on purpose. Each entry gives the **symptom** exactly as it appears on your screen, the **cause** in the code or the protocol, and the **fix**. Where a symptom involves a number, the number came from a real run.

Start here

Three questions answer most problems. *Did the Lab start?* If the failure printed `ballistics-lab:` and exited 2, it never launched — see Section D.1. *Did* `:solve` *succeed?* Check the `termination` line before you distrust any number. *Have you solved since your last change?* Every table and every `:at` reads the last *run*, not the current experiment.

D.1 The Lab will not launch

Launcher failures all look alike: one line on stderr beginning `ballistics-lab:`, and exit status 2. Nothing else is printed and no Lattice runs.

D.1.1 ballistics was not found

Listing D.1: Symptom (the launcher's exhausted-search message, `ballistics-lab.sh:228`)

```
ballistics-lab: ballistics was not found; install it, build ../ballistics-engine, or set
    BALLISTICS_ENGINE
```

Cause. The launcher's discovery order found nothing usable. It tries, in order: an explicit `--engine`; the `BALLISTICS_ENGINE` environment variable; `ballistics` on `PATH`; `$CARGO_TARGET_DIR/release` then `debug`; a sibling `../ballistics-engine/target/release` then `debug`; the caller's own `target/release` and `target/debug`; and finally `~/.cargo/bin/ballistics`.

Fix. Point at it directly, either way:

```
$ sh scripts/ballistics-lab.sh --engine /path/to/ballistics
$ BALLISTICS_ENGINE=/path/to/ballistics sh scripts/ballistics-lab.sh
```

D.1.2 The named engine does not exist

Listing D.2: Symptom

```
ballistics-lab: ballistics executable '/no/such/ballistics' was not found or is not executable
```

Cause. An explicit `--engine` or `BALLISTICS_ENGINE` named a path that is not a file, or is not executable.

Fix. Check the path and the permission bit. A path containing a slash is taken literally; a bare name is looked up on `PATH`.

D.1.3 The engine exists but is the wrong program

Listing D.3: Symptom, produced with `--engine /bin/echo`

```
ballistics-lab: ballistics executable '/bin/echo' does not provide the required solve-json v1
     command
```

Cause. The launcher runs `<engine> solve-json --help` and requires the phrase `explicit-SI v1 JSON request` in the output. Anything else — an old engine, a different tool with the same name, a wrapper script that eats the flag — fails this probe.

Fix. Confirm by hand:

```
$ ballistics solve-json --help | head -1
Solve one explicit-SI v1 JSON request from stdin and write one JSON envelope to stdout
```

During discovery this is a skip, not a failure

When the launcher is *searching*, an incompatible candidate produces `ballistics-lab: ignoring incompatible engine ...` on stderr and the search continues. Only an *explicitly named* engine turns the same probe into a fatal error. That asymmetry is deliberate: you asked for that one.

D.1.4 clat was not found

Listing D.4: Symptoms — the first from an explicit `CLAT=/no/such/clat`, the second the launcher's exhausted-search message

```
ballistics-lab: clat executable '/no/such/clat' was not found or is not executable
ballistics-lab: clat was not found; build the checkout with 'make' or set CLAT
```

Cause. The Lab is Lattice source; it needs the interpreter. The launcher looks for `--clat`, then `$CLAT`, then `./clat` in the checkout root, then `clat` on `PATH`.

Fix. Run `make` in the Lattice checkout, or set `CLAT`.

D.1.5 Launcher option mistakes

Listing D.5: Three symptoms, all exit 2

```
ballistics-lab: unknown option: --frobnicate
ballistics-lab: unknown backend 'turbo' (expected stack-vm or regvm)
ballistics-lab: conflicting backend selections: stack-vm and regvm
```

Fix. `sh scripts/ballistics-lab.sh --help` prints the whole option set. There are five: `--backend`, `--stack-vm`, `--regvm`, `--clat`, `--engine`.

D.2 Engine version problems

D.2.1 The engine version is not what the book says

Symptom. `:solve` reports `engine : 0.32.0` but the book, or a colleague's notebook, says 0.30.1.

Cause. More than one engine is installed and `PATH` decides. Check:

```
$ which ballistics
/Users/alexjokela/.local/bin/ballistics
$ ballistics --version
ballistics 0.32.0
```

Fix, and the good news. For solve-json work there is usually nothing to fix. The same request was fed to both shipped versions and the responses were compared after normalising the version string:

```
$ cmp a.txt b.txt && echo "identical apart from engine_version"
identical apart from engine_version
```

That held for a plain 1,000 m solve and for one exercising segmented wind, an explicit RK45 time step, and enhanced spin drift. The trajectory numbers, the defaults, and the assumption texts are the same. What differs is six extra subcommands and one reticle sub-subcommand in 0.32.0 — none of which the Lab uses. See Section B.1.

If you must be certain, run the specific engine you want:

```
$ sh scripts/ballistics-lab.sh --engine ~/projects/ballistics-engine/target/release/ballistics
```

D.2.2 Pinning a version, and the error when it does not match

The Lab can require an exact engine version. Set `BALLISTICS_ENGINE_VERSION` and every solve checks the envelope's `engine_version` against it:

Listing D.6: Pinning to a version that is not installed

```
$ BALLISTICS_ENGINE_VERSION=0.30.1 sh scripts/ballistics-lab.sh
```

Listing D.7: Symptom, on the next `:solve`

```
Error
  code: incompatible_engine_version
  message: expected engine version '0.30.1', got '0.32.0'
  exit code: 0
```

Cause. A deliberate refusal, not a fault. The check happens after the envelope has been read and before any domain value is constructed, so no trajectory is built from an engine you did not authorise.

Fix. Either install the pinned version, or unset the variable. Setting it to the version you actually have makes the same session solve normally.

Pin it in a lab notebook, leave it unset at the bench

The pin exists so a reproducibility run fails loudly rather than quietly producing near-identical numbers from a different build. For everyday work it adds friction without adding safety.

D.3 The engine ran but the Lab rejected it

Every entry in this section produces the code `invalid_engine_response`. That code means: *the child process returned something the protocol does not promise, so the Lab refused to build a trajectory from it.* It is a protocol violation, not a physics failure.

Each one below was reproduced by pointing the Lab at the repository's fake engine, which serves deliberately hostile responses on demand:

Listing D.8: How these were produced

```
$ ./clat examples/ballistics_lab/ballistics-repl.lat \
      ./clat examples/ballistics_lab/fixtures/fake_solve_json_engine.lat <scenario>
```

D.3.1 Framing failures

Cause. `solve-json` reserves stdout exclusively for the envelope. Anything else on that stream — a progress message, a deprecation notice, a shell wrapper's `echo` — breaks the frame.

Fix. If you wrapped the engine in a script, make the script silent on stdout. Diagnostics belong on stderr, and only on an error path.

Table D.1: stdout is not exactly one JSON envelope

What the engine did	Message
Printed a banner first	`json_parse error at position 0: unexpected character`
Printed a trailer after	`json_parse error at position 3692: unexpected trailing content`
Printed two envelopes	`json_parse error at position 3692: unexpected trailing content`
Truncated the envelope	`json_parse error at position 34: expected ',' or '}' in object`
Emitted a duplicate key	`json_parse error at position 88: duplicate object key`
Emitted an oversized integer	`json_parse error at position 41: integer out of signed 64-bit range`

D.3.2 Stream and status disagreements

Listing D.9: A success that also wrote to stderr

```
Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response: status 'ok' must not write diagnostics to
     stderr
  stderr: unexpected success diagnostic\n
  exit code: 0
```

Listing D.10: An error code that disagreed with the exit status

```
Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response: $.error.code 'invalid_value' requires exit
     2, got 3
  exit code: 3
```

Cause. Two hard rules. A successful solve writes nothing to stderr. An error envelope's code determines the exit status: parse and validation codes require 2, `resource_limit` and `solve_failed` require 3, `io_error` and `internal_error` require 1.

Fix. These indicate a broken engine build or a wrapper that alters either stream. Note that *whitespace-only* stderr on an error path is tolerated — the Lab trims before deciding a diagnostic exists.

D.3.3 Unknown fields

Listing D.11: A future field at the top level

```
Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response: $: unknown field 'future_field'
  exit code: 0
```

Listing D.12: A future field inside the resolved request

```
Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response: $.resolved_request.rifle: unknown field '
     sight_offset_lateral_m'
  exit code: 0
```

Cause. The Lab's decoder uses exact key sets everywhere, with no optional-extras allowance. This is stricter than the protocol requires — solve-json v1's compatibility rules explicitly permit new response fields, and tell consumers to tolerate them.

This is the Lab's one real forward-compatibility risk

A future engine that adds a response field the Lab does not know about will break every solve, with the message naming the field. Two consequences worth planning for: a request carrying a `reticle` block would produce a `reticle_hold` the Lab rejects, and if the engine ever starts echoing `sight_offset_lateral_m` the second message above becomes real rather than synthetic. The fix in both cases is a Lab update (`process_backend.lat`), not a workaround.

D.3.4 Internally inconsistent responses

Cause. The Lab cross-checks the response against itself before building anything. The terminal sample must agree with the summary on distance, time, speed, and energy, within a tolerance of $10^{-10} + 10^{-9}\max(|a|, |b|)$ — the same tolerance the engine's own regression fixtures use. A summary that is off by one part in 10^9 passes; a summary that is off by 1.0 metre does not.

Table D.2: Payload checks, each reproduced

Defect	Message tail
No samples at all	`$.samples: must contain a terminal observation`
No terminal flag	`$.samples: expected exactly one terminal flag, got 0`
Too many samples	`$.samples: exceeds the 10000-sample v1 limit`
Non-finite summary value	`$.summary.actual_range_m: must be finite`
Terminal disagrees with summary	`$.samples: terminal distance_m does not match $.summary`
Impossible effect combination	`$.resolved_request.effects: magnus and enhanced_spin_drift cannot both be enabled`

D.3.5 An unknown status or schema

Listing D.13: A schema the Lab does not implement

```
Error
  code: unsupported_schema_version
  message: unsupported response schema_version 2; expected 1
  path: $.schema_version
  exit code: 0
```

Listing D.14: A status the protocol does not define

```
Error
  code: invalid_engine_response
  message: assertion failed: solve-json v1 response: $.status: unsupported value 'future'
  exit code: 0
```

Note the difference. A future *schema* gets its own clean code, because that is a foreseen situation. A future *status* inside schema 1 is a protocol violation, because v1 defined exactly two.

D.4 Process failures: timeouts and size limits

These produce `process_error` rather than `invalid_engine_response`: the child never delivered a complete response at all.

Table D.3: The process backend's default limits

Limit	Default	Option key
Wall-clock timeout	30,000 ms	`timeout_ms`
Maximum stdout	8,388,608 bytes	`max_stdout_bytes`
Maximum stderr	262,144 bytes	`max_stderr_bytes`

D.4.1 The engine timed out

Listing D.15: Symptom, reproduced with a 200 ms limit against a deliberately slow engine

```
Error
  code: process_error
  message: exec_argv: timed out after 200 ms
```

Cause. The child did not exit within `timeout_ms`. At the default of 30 seconds this means a genuinely enormous solve, a hung engine, or an engine waiting on something — a network call, a prompt, a lock.

Fix. First reduce the work: a smaller `max_range`, a coarser `sampling_interval`. If the solve legitimately needs longer, build a backend with a larger budget and install it:

Listing D.16: Raising the timeout from the Lab prompt

```
flux opts = Map::new()
opts.set("timeout_ms", 120000)
let slow = process.process_backend(engine_executable, nil, opts)
sessions.set_backend(lab, slow)
```

D.4.2 The engine produced too much output

Listing D.17: Symptom, reproduced with a 1024-byte cap

```
Error
  code: process_error
  message: exec_argv: stdout exceeded max_stdout_bytes (1024 bytes)
```

Cause. The reader stopped at the cap. At the 8 MiB default this needs an engine that is streaming something it should not be. A response at the protocol's own ceiling — 10,000 samples — measured 2,046,081 bytes, under 2 MiB, so the default leaves a factor of four in hand.

Fix. Coarsen the sampling interval. If the engine is printing debug output on stdout, that is the actual problem, and it will show up as a framing failure the moment you raise the cap.

The three options are the only ones accepted

`process_backend` validates the option map strictly: only `timeout_ms`, `max_stdout_bytes`, and `max_stderr_bytes`, each a positive `Int`. A misspelled key throws `process_backend.options: unknown option '...'` at construction time, not at solve time.

D.5 Command errors

D.5.1 "no current run is available"

Listing D.18: Symptom

```
Error
  code: command_state_error
  message: at: no current run is available
  path: $.session.current_run
```

Cause. `:at`, `:dope`, `:wind-card`, and `:compare` all read a stored run. A fresh session has none.

Fix. `:solve` first. For `:compare` you need *two* runs, and the message names which one is missing:

```
Error
  code: command_state_error
  message: compare: no previous run is available
  path: $.session.previous_run
```

D.5.2 Wrong number of arguments

Listing D.19: Symptom

```
Error
  code: command_arity
  message: :at expects 2 argument(s), got 1
  path: arguments
```

Cause. Colon arities are exact. The most common way to hit this accidentally is a path containing a space: `:save /tmp/my load.json` is two arguments.

Fix. Quote the path. Either quote character works.

D.5.3 Bad unit or out-of-range argument

Listing D.20: Symptoms

```
Error
  code: command_argument
  message: unsupported distance unit 'furlongs'; expected one of [m, km, yd, ft, in]
  path: arguments[1]
Error
  code: command_argument
  message: :at distance must be non-negative
  path: arguments[0]
Error
  code: command_argument
  message: expected a numeric argument, got 'abc'
  path: arguments[0]
```

Fix. The five distance units are the whole list. Note that `:dope` and `:wind-card` require `start > 0` strictly, while `:at` accepts zero.

D.5.4 "is outside computed trajectory"

Listing D.21: Symptom

```
Error
  code: command_execution_error
  message: at: assertion failed: trajectory_at.distance: 1828.8 m is outside computed trajectory
     [0, 914.4] m
```

Cause. You asked past the end of the run. The Lab interpolates; it does not extrapolate.

Fix. The message hands you the answer: the trajectory ended at 914.4 m, which is 1,000 yards. Either ask inside that interval, or find out why the solve stopped there — Section D.8.

D.5.5 Resource limits

Listing D.22: Symptoms

```
Error
  code: resource_limit
  message: command exceeds the 16-argument limit
  path: arguments
Error
  code: resource_limit
  message: command exceeds the 4096-byte input limit
  path: command
```

Cause. The parser's two hard bounds, enforced before any evaluation. Both boundaries are inclusive: exactly 4096 bytes and exactly 16 arguments are accepted.

D.5.6 `:load` failures

Listing D.23: Three symptoms, all `command_execution_error`

```
message: load: assertion failed: experiment JSON: expected String, got Nil
message: load: assertion failed: $: missing required field 'schema_version'
message: load: json_parse error at position 0: unexpected character
```

Cause, in order. The file does not exist. The file is JSON but is not a Lab experiment document. The file is not JSON at all.

Fix. The first message is the one that misleads: it is a file-not-found, reported through an internal type assertion because `read_file` returned `Nil`. Check the path before you go looking for a corrupt document.

D.5.7 `:save` failures

Listing D.24: Symptom

```
Error
  code: command_execution_error
  message: save: assertion failed: save_experiment: unable to write '/proc/nope/x.json'
```

Cause. The directory does not exist, or is not writable.

Fix. Note what did *not* happen: the document was fully validated and serialised before the write was attempted, so there is no partial file to clean up.

D.6 The number looks wrong: signs

Nothing in this section errors, which is why it is worth reading before you need it. Every table and every `:at` block ends with a legend naming its own conventions, and those conventions are not all the same direction. Drop counts *downward*. Windage counts *rightward*. Once you know that, the legends are literal and you can read a sign off them without translating.

D.6.1 Drop is positive *below* the line of sight

```
Signs: drop + below/- above; windage + right/- left
```

Drop is a *depth*, not a height. A larger positive number means the bullet is lower relative to your line of sight. Negative drop means the bullet is above it.

Demonstration. Move the zero out to 300 yards and query 100 yards. With a 300 yard zero the bullet at 100 yards is unambiguously *above* the line of sight — that is what a 300 yard zero means — and the Lab prints it negative:

Listing D.25: 100 yd on a 300 yd zero: above the sight line, so negative

```
Observation
  distance: 100.00 yd
  time    : 0.12 s
  velocity: 2513.36 ft/s
  energy  : 2454.22 ft-lb
  drop    : -4.80 in
  windage : -0.69 in
  Mach    : 2.25
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

Confirmation. Query the zero distance itself in the same run:

Listing D.26: At the zero, the bullet is on the line of sight

```
Observation
  distance: 300.00 yd
  time    : 0.37 s
  velocity: 2162.46 ft/s
  energy  : 1816.77 ft-lb
  drop    : +0.00 in
  windage : -6.71 in
  Mach    : 1.93
  flags   : none
Signs: drop + below/- above; windage + right/- left
```

That is the check to run whenever you doubt a table: **at the zero distance, drop is zero**. If it is not, something about the zero is not what you think it is.

Where the convention comes from. The engine's `drop_m` is measured downward from the line of sight, and the Ballistics Lab passes it straight through (`process_backend.lat:403`) without flipping it. The one place this surprises people is the muzzle, where drop equals the sight height exactly — `+1.50 in` on this rifle — because before the bullet has gone anywhere the bore sits precisely its sight height below the sight line.

D.6.2 Come-up is positive when the sight must be raised

```
Signs: drop + below/- above; come-up + correction up/- correction down
```

Come-up is the angle the drop subtends at that distance — the elevation you dial. Positive means *up*. Past the zero it grows, monotonically, because the bullet keeps falling further below the line of sight.

The case where it legitimately goes negative is a target closer than your zero, where the bullet is still above the sight line and the correction is genuinely downward. The 300 yard zero shows both halves in one table:

Listing D.27: A 300 yd zero: dial down inside the zero, up beyond it

```
DOPE: 175 gr match at 1,000 yards
Distance (yd) | Drop (in) | Come-up (mil) | Velocity (ft/s) | Energy (ft-lb) | Time (s) | Mach
--------------+-----------+---------------+-----------------+----------------+----------+-----
       100.00 |     -4.80 |         -1.33 |         2513.36 |        2454.22 |     0.12 | 2.25
       200.00 |     -5.59 |         -0.78 |         2334.06 |        2116.55 |     0.24 | 2.09
       300.00 |     +0.00 |         +0.00 |         2162.46 |        1816.77 |     0.37 | 1.93
       400.00 |    +13.06 |         +0.91 |         1998.49 |        1551.70 |     0.52 | 1.79
       500.00 |    +34.83 |         +1.93 |         1841.19 |        1317.05 |     0.67 | 1.65
       600.00 |    +66.88 |         +3.10 |         1689.83 |        1109.40 |     0.84 | 1.51
       700.00 |   +111.15 |         +4.41 |         1544.02 |         926.21 |     1.03 | 1.38
       800.00 |   +170.05 |         +5.90 |         1403.93 |         765.77 |     1.23 | 1.26
       900.00 |   +246.65 |         +7.61 |         1270.55 |         627.18 |     1.46 | 1.14
      1000.00 |   +344.89 |         +9.58 |         1145.99 |         510.23 |     1.71 | 1.03
Signs: drop + below/- above; come-up + correction up/- correction down
```

Both columns cross zero together at 300 yards, and they have to: the come-up is the drop expressed as an angle, so they share a sign.

Why come-up follows drop but wind hold opposes windage

These look inconsistent and are not. Drop is positive-*below*, so a bullet that is low needs the sight raised — correction and quantity point the same way, and come-up carries drop's sign. Windage is positive-*right*, so a bullet that has drifted right needs a hold to the left — correction and quantity point opposite ways, and the hold carries the opposite sign. Two conventions, two behaviours, and each table's legend states its own.

D.6.3 Windage and wind hold

Listing D.28: A 10 mph wind from 90 degrees — from the shooter's right

```
  0.00 | 175 gr match at 1,000 yards |      10.00 |      90.00 |       1000.00 |      -102.04 |
             +2.83
Signs: windage + right/- left; wind hold + correction right/- correction left
```

Wind from the right pushes the bullet left, so windage is negative; the hold is to the right, so wind hold is positive. Both agree with the legend and with the physics.

D.6.4 My come-ups disagree with the engine's `come-ups` subcommand

They agree. The `come-ups` subcommand takes flags rather than solve-json, so it is an independent path through the same engine, and it is the best available cross-check on a dope card. Fed the same load, the two match at every distance.

Listing D.29: The engine's own come-up table

```
$ ballistics come-ups --velocity 2710 --bc 0.326 --mass 140 --diameter 0.264 \
      --drag-model g7 --zero-distance 100 --sight-height 1.75 --altitude 4000 \
      --temperature 50 --humidity 35 --start 100 --end 1000 --step 100 \
      --adjustment-unit mil
```

Table D.4: The Lab's come-up column against the engine's own, same load

Distance	Lab `Come-up (mil)`	Engine `Drop (MIL)`
100 yd	+0.00	0.000
200 yd	+0.48	0.478
300 yd	+1.17	1.168
400 yd	+1.95	1.949
500 yd	+2.80	2.804
600 yd	+3.73	3.730
700 yd	+4.73	4.730
800 yd	+5.81	5.809
900 yd	+6.97	6.975
1000 yd	+8.24	8.235

Compare against the right column

In the engine's table the **total dial is the column headed `Drop (MIL)`**. The column headed `Come-Up` is the *incremental* step from the previous row — $0.478 + 0.689 = 1.168$, and $1.168 + 0.781 = 1.949$. The Lab's `Come-up (mil)` column is a total, so it lines up with the engine's `Drop (MIL)`. Comparing it against the engine's `Come-Up` column will show a disagreement that is not there.

If they genuinely disagree, check two things. First, the **sight height** — it is an input to both, and `come-ups` will not read it from your experiment. Second, the **target height**: the field `target_height_m` is the height above ground that the engine solves the zero *to*, and its own default is ground level. The Lab fills it in with muzzle height plus sight height — the height of your line of sight — so that a zero

means what shooters mean by a zero. If you have set `target_height` explicitly on the shot, you have overridden that, and a zero solved to a different datum will shift every come-up by roughly the datum offset divided by the zero distance.

Two paths, one engine, slightly different inputs

The agreement above is exact because that load was set up so both paths solve the same problem: no explicit bullet length and no explicit pressure, which `come-ups` has no flag for anyway. Where your experiment specifies things the subcommand cannot take — a bullet length, a station pressure — expect a few hundredths of a mil of difference from the atmosphere and the drag, not from the signs. A disagreement in the *direction* of the dial is a real problem; a disagreement in the third decimal is two surfaces with different defaults, the same genre as Section D.8's range mismatch.

D.7 The number looks wrong: state

D.7.1 `:show` and `:at` disagree after a `:load`

Symptom. You load a saved experiment, `:show` confirms it, and the queries return numbers from a completely different rifle.

Demonstration. Solve at 2,700 fps, save, change the muzzle velocity to 3,300 fps, solve, then load the saved file back and query:

Listing D.30: After `:load` without `:reset` — the 3,300 fps answer

```
Observation
  distance: 1000.00 yd
  time    : 1.33 s
  velocity: 1544.95 ft/s
  energy  : 927.32 ft-lb
  drop    : +239.44 in
  windage : -71.90 in
  Mach    : 1.38
  flags   : terminal
Signs: drop + below/- above; windage + right/- left
```

Listing D.31: After `:reset` and `:solve` — the 2,700 fps answer

```
Observation
  distance: 1000.00 yd
  time    : 1.71 s
  velocity: 1145.96 ft/s
  energy  : 510.20 ft-lb
  drop    : +392.91 in
  windage : -102.04 in
  Mach    : 1.03
  flags   : transonic, terminal
Signs: drop + below/- above; windage + right/- left
```

A 400 ft/s error in terminal velocity, silently, with `:show` insisting the rifle is the loaded one.

Cause. `:load` calls `sessions.replace_experiment` and nothing else (`commands.lat:339`). That mutator preserves `current_run`, `previous_run`, and the history — by design, because replacing a component should not silently discard a run you may still want to compare against.

Fix. Make it a habit:

Listing D.32: The safe load idiom

```
:load "/path/to/experiment.json"
:reset
:solve
```

The same caution applies to every `sessions.replace_*` call. Nothing invalidates a stored run except `:reset` or a new `:solve`.

D.7.2 `:reset` did not reset what I expected

Symptom. After `:reset` the rifle is still the one you edited.

Cause. The response string is precise: `Session run state reset.` It clears the history, the current run, and the previous run. It does not touch the experiment, its name, the backend, the display preferences, or the history limit. Table A.3 is the full list.

Fix. To get a clean experiment as well, `:load` a saved one or relaunch.

D.7.3 Every run is labelled with the wrong load

Symptom. You replaced the projectile, rifle, atmosphere, wind, shot, and solver, and `:show` still says `175 gr match at 1,000 yards`.

Cause. `_experiment_with` (`session.lat:285`) always carries `current.name` forward. None of the `replace_*` mutators can change the name.

Fix. Rename with a whole-experiment replacement:

```
let cur = sessions.experiment_of(lab)
sessions.replace_experiment(lab, experiment("140 gr ELD-M at 1,200 yd", cur.projectile, cur.rifle
     , cur.atmosphere, cur.winds, cur.shot, cur.solver))
```

Do this *before* you solve, or the run record — and every table's provenance block — carries the wrong name.

D.7.4 `:compare` shows two identically named series

Cause. The same root: only the wind, or only the velocity, changed between the two solves, and the name did not.

Fix. Rename between solves. Series 0 is always the *previous* run and series 1 the current, so the ordering is unambiguous even when the labels are not.

D.8 The number looks wrong: physics

D.8.1 "actual range" is shorter than I asked for

Listing D.33: Symptom: a 5,000 yd request

```
  termination      : ground_threshold
  actual range     : 2094.02 yd
```

Cause. Not an error. The bullet reached the ground threshold — default −100 m below the muzzle — before it reached the requested range, and the solver stopped, correctly.

Fix. Read the `termination` line first, always. `max_range` means you got the distance you asked for; the other three values (`ground_threshold`, `time_limit`, `velocity_floor`) mean the solve ended on its own terms. If you want to model a longer flight, lower `ground_threshold` or raise the muzzle — and if you raise the muzzle, read the next entry first.

D.8.2 “Zero angle did not converge”

Listing D.34: Symptom

```
Error
  code: solve_failed
  message: Zero angle did not converge: residual height error too large (target not reachable /
     not bracketed)
  exit code: 3
```

Cause. `target_height_m` is the height above the ground datum that the engine solves the zero *to*, and it is **absolute** — not a height relative to the muzzle. Ask the solver to zero a rifle held at chest height onto a target lying on the ground at 100 yards and the elevation search has no bracket, which is what this message reports.

You have to ask for it explicitly. Raising the muzzle alone does not produce this. The Lab supplies a target height when the shot does not carry one, and it supplies the height of the line of sight — muzzle height plus sight height (`protocol_v1.lat`) — so the target tracks the rifle. A 1.5 m muzzle height on its own solves normally. The failure above was reproduced by overriding that default with a target at ground level:

Listing D.35: Muzzle at chest height, target explicitly on the ground

```
sessions.replace_rifle(lab, rifle("24-inch match rifle", inches(1.5), m(1.5), inches(8), "right")
    )
sessions.replace_shot(lab, shot(fps(2700), yd(1000), yd(100), nil, nil, nil, nil, nil, m(0), nil)
    )
:solve
```

Fix. Leave `target_height` unset unless you mean it, and the zero is solved to your line of sight. If you do set it — to model shooting at a target on the deck, or off a tower — set it to a height the bullet can actually reach at the zero distance from the muzzle height you gave it.

D.8.3 A sample count I did not ask for

Listing D.36: Symptom, produced by asking for a 0.05 m sampling interval over 1,000 yards

```
Error
  code: resource_limit
  message: trajectory observation limit of 10000 exceeded (would produce 18289)
  path: $.sampling.interval_m
  exit code: 3
```

Cause. The engine refuses rather than thinning your data. The limit is evaluated against the range actually reached, so a solve that goes further than you expected can trip it.

Fix. Coarsen the sampling interval on the solver config. The message tells you the exact count it would have produced, so the required factor is arithmetic — 18,289 over 10,000 means at least a 1.83× coarser grid. The command that produced the error above was:

```
sessions.replace_solver(lab, solver_config("rk45", nil, m(0.05), nil, true, nil))
:solve
```

D.8.4 A DOPE table stops short of where I asked

Cause. `:dope` and `:wind-card` clip the requested stop to the trajectory and append the true terminal row. A request for `:dope 900 2000 200 yd` on a 1,000 yd solve returns two rows: 900 and 1,000.

Fix. Nothing to fix — but if the *start* is past the end you get a hard error instead:

```
Error
  code: command_execution_error
  message: dope: assertion failed: sample.start: must be inside the computed trajectory
```

And if you cross-check it against the CLI. From engine 0.37.0 the engine's own card subcommands stop short too, and they announce it, so a reader comparing a Lab card against `come-ups` or `range-table` will see a line the Lab never prints:

```
$ ballistics come-ups -v 1050 -b 0.125 -m 40 -d 0.224 --drag-model g1 \
      --zero-distance 50 --sight-height 1.5 --start 100 --end 1200 --step 100 -o csv
range_yd,drop_mil,come_up_mil,velocity_fps,energy_ft-lb,time_s
100,2.177,2.177,900,72,0.311
...
1000,109.264,20.525,362,12,5.350
warning: card truncated at 1000 yd: 1200 yd was requested, but this load's solved flight reaches
     only 1000 yd; the rows past that are left out rather than filled in from a range the bullet
     does reach
```

That is not an error — the command exits 0, and the rows it printed are identical to the ones a `--end 1000` card would have given. The warning goes to stderr so `-o csv` and `-o json` stay machine-clean; `-o json` carries the same facts as a `truncated` object instead. The two card systems also stop in different places: the Lab appends the true terminal row off-grid, the engine stops at the last *requested* row and puts the flight's terminal distance in the notice's `reach` field. Expect the final labels to differ by up to one row spacing, and reconcile against `reach`. Section 13.7 has the whole contract.

The engine still refuses outright when *no* requested row is reachable, which is the counterpart of the Lab's start-past-the-end error above:

```
$ ballistics come-ups ... --start 1100 --end 1200 --step 100 -o csv
Error: "no trajectory sample at 1100 yd: this load's solved flight reaches only 1000 yd"
$ echo $?
1
```

D.8.5 The engine gave me a shorter range than the Lab did

Cause. They are different termination models. On the CLI's `trajectory` subcommand, ground-impact detection is on by default and uses a bore height that defaults to 60 inches. The same load at `--max-range 1000` returned:

```
default                   max_range = 524.8493662123586 yd
--ignore-ground-impact    max_range = 1000.0 yd
```

Through solve-json the equivalent control is `ground_threshold_m`, which defaults to −100 m — effectively off for a level shot. The two surfaces are consistent with themselves and differ from each other by default.

Fix. When cross-checking a Lab number against a CLI number, pass `--ignore-ground-impact` or set the bore height and ground threshold to match.

D.9 Backend differences

Symptom you might fear. Two Lattice VMs, floating-point formatting, and a physics book.

What was measured. An identical session — `:solve`, `:at 1000 yd`, `:dope 100 1000 100 yd`, `:windcard 200 1000 200 yd` — run on both backends:

Listing D.37: Byte-for-byte agreement

```
$ sh scripts/ballistics-lab.sh          < in.txt > out_stack.txt 2>/dev/null
$ sh scripts/ballistics-lab.sh --regvm  < in.txt > out_regvm.txt 2>/dev/null
$ cmp -s out_stack.txt out_regvm.txt && echo "byte-identical"
byte-identical
$ shasum -a 256 out_stack.txt out_regvm.txt
89de770ac400800ecc16cef1997dce3fba1db40f99b7811d801aa7b817792f35  out_stack.txt
89de770ac400800ecc16cef1997dce3fba1db40f99b7811d801aa7b817792f35  out_regvm.txt
```

The only difference between the two runs is the launcher's stderr line:

```
ballistics-lab: backend=stack-vm, engine=/Users/alexjokela/.local/bin/ballistics
ballistics-lab: backend=regvm, engine=/Users/alexjokela/.local/bin/ballistics
```

What that proves, and what it does not

The trajectory integration happens in Rust, not in Lattice. So identical output proves the two VMs agree on *request construction, unit conversion, interpolation, table assembly, and float formatting* — everything the Lab itself computes. It does not test the ODE arithmetic, because neither VM does any.

Fix, if you ever do see a difference. You have found a Lattice VM bug, not a ballistics bug. Capture both outputs and diff them; the first differing line names the operation.

D.10 Reproducing a problem for a bug report

A useful report needs five things, all of which the Lab will tell you:

1. **The Lattice version**, from the banner's first line.

2. **The engine path**, from the banner's second line, and the **engine version**, from any `:solve` or any table's provenance block.
3. **The backend**, from the launcher's stderr line.
4. **The experiment**, saved to a file with `:save` — it is a complete, validated, re-loadable description of the load.
5. **The exact input**, as a file you can pipe back in.

Listing D.38: A self-contained reproduction

```
$ sh scripts/ballistics-lab.sh < repro.txt > out.txt 2> err.txt
$ head -2 out.txt          # Lattice version and engine path
$ cat err.txt              # backend and resolved engine
$ grep engine out.txt      # engine version
```

If the problem is in the protocol rather than in the Lab, reduce it further: take the request the Lab would send and run it through the engine directly.

Listing D.39: Reducing to a bare engine call

```
$ ./clat examples/ballistics_lab/reference_experiment.lat --json > req.json
$ ballistics solve-json < req.json > resp.json; echo $?
0
```

That pipeline returned 101 samples and this summary:

Listing D.40: The reference experiment, solved outside the Lab

```
{"actual_range_m": 914.4,
 "maximum_height_m": 0.04111754369481144,
 "time_of_flight_s": 1.7064080077984438,
 "terminal_speed_mps": 349.2875788381682,
 "terminal_energy_j": 691.7386422597805,
 "stability_factor": 3.7979628593634613,
 "termination": "max_range"}
```

Compare that against what the Lab printed for the same experiment: `terminal velocity: 1145.96 ft/s`, `terminal energy: 510.20 ft-lb`, `stability factor: 3.80`, `time of flight: 1.71 s`. The numbers are the same number, rendered twice — 349.2876 m/s is 1145.96 ft/s, and 691.7386

J is 510.20 ft-lb. If your two sides ever disagree, the fault is in the Lab's request construction or its rendering, and you have localised it.

Check the request, not only the response

The `req.json` in the middle of that pipeline is the most useful file in a bug report, because it is exactly what the Lab asked for. Two fields repay a look before you blame the engine: `zero_distance_m`, and `target_height_m` — the height above ground the zero is solved to. In the request above the latter reads `0.0381`, which is the rifle's sight height with the muzzle at ground level: the zero is solved to the line of sight. If a trajectory looks uniformly shifted, that is the field to read first.

Two files, one exit code, no Lattice — and the person reading your report can run it in one line.

Index