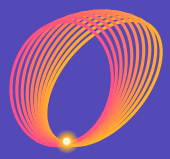

ODEROM

Oriented Differential Exterior calculus for Relativity, Operators and Manifolds

User manual · a notebook for tensor calculus

Rafael Camargo Rodrigues de Lima

UDESC — Universidade do Estado de Santa Catarina

rafael.lima@udesc.br

Revised · 3 September 2026

Contents

1. About this manual, and about ODEROM

ODEROM does symbolic computation in differential geometry. Give it a manifold, a coordinate chart and a metric, and it computes curvature — Christoffel symbols, the Riemann and Ricci tensors, invariant scalars such as Kretschmann — in closed form, with no numerical evaluation anywhere. The main interface is a notebook: a stack of code blocks you edit and run one at a time, each with its typeset result below it.

There is a second half to the program, and it is worth naming early because it is easy to miss. Alongside the *component* engine just described, ODEROM manipulates tensor expressions in *abstract indices* — canonicalising a monomial under its declared symmetries, collecting like terms, and deciding whether a sum vanishes under identities you declare (the Bianchi identities, metric compatibility). That is the `simplify` verb, section 9.18. The two halves share a language and a window but never each other's machinery: one knows what R_{trtr} equals for your metric, the other knows that $R_{abcd} + R_{acdb} + R_{adbc}$ is zero for every metric.

This document covers the notebook and how it behaves, how to run the program in three different places, the file format, the complete grammar, eight examples executed end to end, and the command line.

> **How this manual was verified.** The grammar, the query list, the error messages and every example were captured by running the real binary — never written from memory, never copied from a design document. Every command shown was executed at the revision this manual describes, and its output pasted back in. Where something could not be verified that way (build prerequisites on macOS and Windows, which this repository does not exercise in CI), the text says so instead of guessing.
>
> **A note on language.** The manual is in English; the program's *error messages* are in Portuguese in places, and deliberately so — they name what you wrote wrong, in the author's language. Interface labels follow the language selector (section 3); error text does not. Messages quoted in this manual are quoted exactly as the program emits them, mixed languages included.

2. Running ODEROM: browser, desktop, terminal

There are three ways to run ODEROM, and they share one engine. The same file, the same grammar and the same results — only the surface differs.

In a browser, installing nothing

https://oderom.pages.dev opens the notebook in a browser tab. No download, no installer, no administrator rights, no operating-system dependency — it behaves the same in a university lab, on a borrowed laptop, and on a machine where you are not allowed to install anything.

The computation runs *on the machine that opened the page*, compiled to WebAssembly (the `oderom-wasm` crate). Nothing is sent to any server; there is no account and no session to lose. The notebook opens with the Reissner–Nordström example loaded and nothing executed — `Shift`+`Enter` runs a block.

Two differences from the desktop application, both imposed by the platform:

- **Saving downloads a file.** A browser page cannot write where it likes, so *Save* hands you an `.od` download, in exactly the format the desktop application reads. Opening works the same way, through the file picker.
- **Opening a file needs the picker.** The path field is how the desktop application addresses the filesystem; in the browser the file dialog stands in for it.

Everything else is the same, including the parts that once were not: a long computation no longer freezes the tab, and *Cancel* genuinely stops it. The work happens in a Web Worker, so the page keeps answering while a block runs — this is verified by a test that drives the real page in a real browser (`oderom-wasm/tests/navegador.rs`), not by inspection.

As a desktop application

ODEROM is a Cargo workspace; the notebook application (`oderom-app`) is a Tauri 2 shell over a static front-end — no Node.js, no npm, no JavaScript build step. `tauri.conf.json` declares neither `devUrl` nor `beforeBuildCommand`: `build.frontendDist` points straight at `oderom-app/dist/`, and CodeMirror 5 and KaTeX are vendored there as files (not a CDN, not an npm dependency) — the window opens with no network access.

Prerequisites

- A stable Rust toolchain (edition 2021). No minimum version is pinned in `rust-toolchain.toml`; any recent stable works.
- **Linux** — the WebKitGTK system libraries, taken from this project's own CI pipeline (`.github/workflows/ci.yml`, the `TAURI_APT_DEPS` variable), not from generic Tauri documentation:

```
sudo apt-get install libwebkit2gtk-4.1-dev build-essential curl wget file \
  libxdo-dev libssl-dev libayatana-appindicator3-dev librsvg2-dev
```

- **macOS / Windows** — this repository runs no CI on those platforms (every job is `ubuntu-latest`), so there is no prerequisite *verified by this project* to quote. The standard Tauri 2 requirements apply (Xcode Command Line Tools on macOS; the WebView2 Runtime and the MSVC build tools on Windows) — stated here as general knowledge, not as something confirmed against this repository.
- Node.js is not needed on any platform.

The launcher

The repository root has an `./oderom` script that dispatches on an explicit first word:

```
./oderom                    # opens the notebook window
./oderom repl               # the terminal REPL
./oderom build-ui           # packages the UI binary
./oderom kretschmann f.od   # anything else goes to the CLI verbatim
```

Building and running the window by hand is the same thing without the script:

```
cargo build -p oderom-app
./target/debug/oderom-app
```

Distributable packages (Linux)

`cargo tauri build` packages all three Linux targets the tauri-cli recognises:

```
cargo tauri build -b deb
cargo tauri build -b rpm
cargo tauri build -b appimage
```

None of the three introduces a Node.js or npm dependency — bundling is entirely native. Package metadata is declared explicitly in `tauri.conf.json`:

Field	Value
`identifier`	`com.oderom.notebook` — reverse-DNS, deliberately not ending in `.app`, which collides with the macOS bundle extension
`productName`	ODEROM
`bundle.category`	Education (Tauri's enum has no dedicated "Science" category)
`bundle.shortDescription`	Symbolic differential geometry notebook for general relativity
`bundle.license`	MIT OR Apache-2.0 — the same licence as the Cargo workspace. The bundler does not inherit this field from `Cargo.toml`; it had to be declared separately.

Installing from a package — the end user's path, with no Rust installed:

```
# .deb (Debian/Ubuntu and derivatives)
sudo apt install ./ODEROM_0.1.0_amd64.deb

# .AppImage (any modern Linux distribution, installs nothing)
chmod +x ODEROM_0.1.0_amd64.AppImage
./ODEROM_0.1.0_amd64.AppImage

# .rpm (Fedora/openSUSE and derivatives)
sudo rpm -i ODEROM-0.1.0-1.x86_64.rpm
```

The `.deb` and `.rpm` install a system binary plus a menu entry, and depend on `libwebkit2gtk-4.1-0` and `libgtk-3-0` already being present. The `.AppImage` installs nothing — one self-contained executable (~77 MB against ~4 MB for the others, because it carries the whole GTK/WebKit runtime).

> **Code signing.** None of the three packages is signed, and on Linux that blocks nothing. It is a known open item for a future macOS/Windows packaging round: unsigned, macOS Gatekeeper and Windows SmartScreen warn or refuse. Not solved here, only recorded.

On the command line

Two binaries, both separate from the notebook and from each other: `oderom` (one command per process) and `oderom-repl` (an interactive session). Both use the grammar of chapter 9 and the same engine as the notebook. Chapter 11 covers them in full.

First run

The window opens titled "ODEROM", with a dark header carrying the mark, and the notebook already holding the built-in Reissner–Nordström example — four blocks, none of them executed (`[ ]` in the gutter), the first already focused:

```
manifold M dim 4
bundle TM on M dim 4
%%
chart schw on M coords (t, r, theta, phi)
%%
metric g on schw bundle TM {
  [t,t] = -(1 - 2*M/r + Q^2/r^2),
  [r,r] = 1/(1 - 2*M/r + Q^2/r^2),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
%%
kretschmann
```

The status bar reads `ready · 4 blocks · 0 executed`. Nothing is computed until you press a run key — not even in the block that already has focus.

3. Anatomy of the notebook

The notebook is a vertical stack of blocks. Each block has three parts:

- **The gutter**, on the left, with an execution indicator in brackets. An empty `[ ]` means never run; `[n]` means this was the *n*-th block run in this session. The numbering reflects the real order of execution, not the position of the block on the page.
- **The editor**, where you type. ODEROM's grammar is syntax-highlighted.
- **The output**, which appears below the editor after a run, mathematically typeset. Some outputs combine mathematics with a plain-text summary line.

At the bottom of the stack there is always an empty block with a dashed border. It is a permanent target to start typing in and a reminder of the run shortcut; it is not a "real" block until it has content.

The header

Left to right, the header carries the mark and the document name, then the controls:

Control	What it does
Language selector	Chooses the interface language — Portuguese, English, Italian, French. See below.
Path field	The file path that *Open* and *Save* act on (chapter 8). In the browser, saving downloads instead.
Open, Save	Read and write a notebook in the `%%` format.
Gallery	Opens a panel of known metrics to load without typing them (chapter 13).
Export	Opens a panel that writes an `export` line into a new block (section 9.17, and below).
Clear execution	Returns every block to not-executed, keeping the text.
New notebook	Deletes every block. Asks for confirmation.

The footer is a status bar reflecting the notebook's state live, including whether a block is currently running.

> **Nothing recomputes by itself.** This is the notebook's central principle. Opening a file never runs anything; every block arrives not-executed. No edit, no navigation and no interface action triggers a computation. Only the run keys, pressed deliberately, execute a block — and only that block. Every control described in this chapter obeys it, the Export panel included: it writes a line, it does not run it.

The language selector

The dropdown at the left of the header switches the language of the *interface* — button labels, tooltips, panels, the status bar. Four languages: Português, English, Italiano, Français. The choice is remembered between sessions.

What it deliberately does not translate is anything the engine produces as *content*: formulas, orbit annotations, summary lines and, above all, error messages. An error names what you wrote wrong, and it does that in Portuguese regardless of the selector. The one exception is the gallery's prose (each entry's description and invariant), which is translated where a translation exists and falls back to the original otherwise — translating may never invent or hide a metric, and a test checks that the gallery still lists the same six entries in every language.

The Export panel

`export sympy kretschmann` has always worked, and nothing on screen said so: the syntax highlighter only colours the word after you have typed it, which helps only those who already know. The Export button opens a panel listing the output formats and the queries, showing the exact line each choice produces *before* you click.

Clicking writes that line into a new block at the end of the notebook and focuses it. It does not run it. Running stays `Shift`+`Enter`, as for every other block — a button that also computed would be the only place in the program where a click starts a calculation. The panel teaches the syntax instead of hiding it, so that the second time you can write the line yourself.

Clear execution and New notebook: two ways to start over, two different risks

The two controls at the right of the header both return the notebook to a clean state, but they are not the same thing — each exists for a different risk, which is why they are separate buttons, visually distinct (the second in red), and never one control with options.

Button	What it does	What it keeps	Confirms?
Clear execution	Returns every block to not-executed (`[ ]`) and discards all session state: the computation cache, declared aliases, the execution counter (the next run is `[1]` again), staleness marks, every displayed result.	The text of every block, untouched.	No — reversible: run again and any result comes straight back.
New notebook	Deletes every block and starts empty (a single blank block, like a freshly created notebook), clearing all session state too.	Nothing of the previous content.	Yes — destructive: deleted text cannot be recovered.

Clear execution is the analogue of restarting a kernel: the state after clicking is indistinguishable from opening that same text as a fresh file — same text, no executions, no session memory. Being entirely reversible, it asks nothing; asking would be friction with nothing to protect.

New notebook deletes text you wrote, so it asks — a plain panel (not a blocking modal dialog) explaining exactly what is lost before anything happens. Only an explicit click on the red confirm button performs it; closing the panel any other way (Escape, clicking outside, the Cancel button) cancels without touching anything.

Neither one touches a file. If the current notebook came from a file, neither *Clear execution* nor *New notebook* modifies it on disk — both act only on what is on screen. After *New notebook* the original file is exactly where it was, ready to be reopened (chapter 8) if the blank notebook was not what you wanted.

4. Running blocks: the keyboard model

ODEROM follows Jupyter's model, not Mathematica's. There are three run keys, and what separates them is what happens to the focus afterwards:

Key	What it does
`Shift` + `Enter`	Runs the current block and moves focus to the next one. If this is the last block, creates a new empty block below and focuses that instead.
`Ctrl` + `Enter`	Runs the current block and keeps focus on it. Creates nothing. This is the shortcut for iterating in place.
`Alt` + `Enter`	Runs the current block and inserts a new empty block immediately below, even if a block already follows, moving focus to the new one.

Moving focus does not run anything. The destination block is focused and stays not-executed — there is never a chain reaction. Each run is a separate gesture.

On opening, the first block takes focus automatically, so the keys work without clicking inside an editor first.

5. The states of a block

The gutter and the block's appearance distinguish five states:

State	Indicator	Meaning
Never run	`[ ]`	The block has not been executed. Every block is in this state when a notebook opens.
Run and current	`[n]`	It ran, and its result matches the current code.
Run and stale	`[n]` amber + ▌ + ↻	It ran, but the code has changed since (or a block above it changed). The old result stays visible, marked as out of date.
Running	—	The computation is under way. The block offers a way to cancel.
Cancelled	—	The user interrupted it. Any earlier result stays on screen as stale.

Staleness

When an already-run block has its code edited, it becomes stale: the result on screen was computed from text that no longer exists. Beyond the block itself, every executed block *below* it also goes stale, because they may depend on what changed.

The staleness mark is purely informative — it never recomputes and never deletes. The old result stays readable; it is simply shown as old. The distinction uses amber (never red, which is reserved for errors) plus non-colour signals — a side stripe, a ↻ glyph in the gutter, and a text label above the output — so that it reads independently of colour perception.

The mark clears when the block is run again (only that block; the ones below stay marked until each is run), or when the edit is undone and the text is once again exactly what it was at the moment of the run.

6. One block at a time, and cancelling

The notebook runs one block at a time. While a computation is under way, the run keys in any other block are refused — they do not start a second calculation. The refusal is always signalled: a momentary highlight appears on the status bar and on the block that is occupying the engine, showing who is blocking. It is never silent and never a modal dialog.

The exclusion holds for any combination — a query does not start while another runs, and neither does a declaration. The refused block does not change state and does not lose its previous result.

A running computation can be cancelled. The calculation genuinely stops. The cancelled block takes the cancelled state — it does not revert to looking never-run, because an attempt happened. If the block already had a result from an earlier run, that result stays on screen, marked stale.

While a block runs, the interface stays responsive: the other blocks accept focus and editing normally. Blocking execution does not block editing. This holds in the browser as well as on the desktop.

> **Getting out of a block.** If you try to run a block and the run is refused, there are two ways out: wait for the running block to finish, or cancel it. As soon as the running computation ends or is cancelled, the next block runs normally.

7. Creating, editing and deleting blocks

Each block carries a single control in its top-right corner: delete — and that same space becomes the cancel button while the block is running (chapter 6). There is no dedicated button for inserting a block anywhere: that happens only as an effect of the run keys — `Shift`+`Enter` on the last block, or `Alt`+`Enter` on any block (chapter 4). Editing is direct: an ordinary code editor with highlighting. Blocks cannot be reordered — creation order is the only way to decide position in v1.

Inserting or deleting a block marks the executed blocks below the change as stale, by the same logic as chapter 5 — the structure changed, so what came after may no longer correspond.

8. Saving and opening: the `%%` format

Notebooks are saved and opened in a text format that separates blocks with a `%%` marker. The path field at the top of the window, with the Open and Save buttons, controls this. A reopened notebook comes back with every block not-executed — consistent with "nothing recomputes by itself": opening a file never triggers a computation.

The exact rule (`oderom-notebook/src/persist.rs`): `%%` delimits blocks as a line whose content, after stripping a trailing `\r`, is exactly `%%` — not a prefix, no label or metadata beside it. *N* blocks need exactly *N*−1 delimiter lines; the first block has no delimiter before it. This is safe because an `.od` comment is `#`, never `%%` — but if a block's own text contains a line that is exactly `%%`, saving is refused with an error rather than corrupting the file by writing it anyway.

The block's kind is never stored. Only each block's source text is saved — never the output, never whether it is a declaration or a query. On reopening, each block starts as if freshly created (never run), and its kind is rediscovered from scratch by `classify_block` from the text itself — so it can never drift out of sync with the real content, because it is never stored separately from it.

A real example, produced by the serialisation function itself (`persist::render`) from a four-block notebook — not reconstructed by hand:

```
manifold M dim 4
bundle TM on M dim 4
%%
chart schw on M coords (t, r, theta, phi)
%%
metric g on schw bundle TM {
  [t,t] = -(1 - 2*M/r),
  [r,r] = 1/(1 - 2*M/r),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
%%
kretschmann
```

File extension: `.od`, one for everything — notebooks and command-line files alike. The path field shows the placeholder `path/file.od` and the browser version downloads with that suffix; none of this is validation, and `save_notebook` / `open_notebook` accept any path.

The shared extension does not mean the two uses are interchangeable, and it is worth knowing in what sense each direction works.

A command-line `.od`, opened in the notebook: works. With no `%%` line at all it becomes a single block holding everything (`parse_sources` returns a one-element vector when it finds no delimiter) — never one block per declaration. `Shift`+`Enter` runs the lot; to work block by block, split it by hand.

A notebook, handed to the command line: also works. `parse_model` blanks every line that is exactly `%%` before reading the rest, so a file saved by the notebook runs in the terminal with no editing. The lines are *blanked*, not removed, so that line numbers in error messages keep pointing at the right place. Only a line that is exactly the delimiter counts — a lone `%` keeps whatever meaning it has anywhere else.

9. The ODEROM language

A notebook block (or an `.od` file) holds a sequence of declarations and queries. The parser decides which is which by looking at nothing but the first word (`classify_block`, `oderom-cli/src/parser.rs`) — eight declaration keywords, and fifteen words that start a query:

```
manifold  bundle  head  chart  metric  connection  axiom      (declaration, 10)
tensor  transition  equation

christoffel  riemann  ricci  scalar  kretschmann  geodesic     (query, 17)
accel  einstein  riccisquare  gaussbonnet  weyl  weylsquare
export  simplify  eval  invariant  solve
```

Section 9.23 lists every one of them, together with the structural words that appear inside a declaration and the function names the expression parser reserves.

A seventh declaration form has no keyword of its own: an alias, `NAME := expression` (section 9.9), recognised by the `:=` rather than by its first word. A blank block is not an error and not a kind — it classifies as empty and does nothing. Any other opening word is an explicit error, never a silent guess.

9.1 Manifold, bundle and chart

```
manifold M dim 4
bundle TM on M dim 4
chart schw on M coords (t, r, theta, phi)
```

`manifold NAME dim N` declares a manifold of dimension *N*. `bundle NAME on MANIFOLD dim N` declares a bundle over an already-declared manifold — `dim` here is the fibre dimension, which in general need not match the manifold's, though for the tangent bundle (the only one used in this manual's examples) it always does. `chart NAME on MANIFOLD coords (c1, c2, ...)` declares a coordinate chart; the number of coordinate names must match the manifold's dimension exactly — `chart c on M coords (x, y)` against a `manifold M dim 3` is rejected with `chart `c` has 2 coordinates, but manifold `M` has dimension 3` (the real message, captured by running the parser).

The `tangent` marker

A bundle declaration may end with the word `tangent`:

```
bundle TM on M dim 4 tangent
```

This marks which bundle a covariant derivative acts on. It matters only for abstract-index work (sections 9.3 and 9.18), where writing `T[a,b;c]` means the derivative index `c` lives in *some* bundle over `M`, and the program has to know which. The resolution rule, in order:

- Exactly one bundle over that manifold — it is used, marked or not. The marker is unnecessary in the common case, which is why the examples throughout this manual omit it.
- More than one bundle, exactly one marked `tangent` — the marked one is used.

- More than one bundle and none marked, or more than one marked — an error naming the candidates, never a guess:

```
error: cannot tell which bundle a covariant derivative acts on over
  manifold `M`: more than one bundle and none is marked
  (candidates: TM, E) -- mark exactly one with `tangent`, as in
  `bundle TM on M dim N tangent`
```

9.2 Metric, connection, and components of any tensor

```
metric g on schw bundle TM {
  [t,t] = -(1 - 2*M/r + Q^2/r^2),
  [r,r] = 1/(1 - 2*M/r + Q^2/r^2),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
```

`metric NAME on CHART bundle BUNDLE { ROW, ROW, ... }`. `bundle` here names which already-declared bundle carries the metric tensor's indices — the metric declares, on its own, an abstract rank-2 head, symmetric, both indices covariant; this happens automatically and is never written by hand. Each *ROW* inside `{ }` is `[i,j] = EXPR`, comma-separated, with no comma after the last. The bracketed indices are coordinate names from the referenced chart (or a literal integer). The alternative LaTeX form uses a subscript instead of brackets, echoing the declaration's own name: `g_{tt} = ...` (see 9.4) — the two forms produce an identical tree, verified by comparing the `kretschmann` of the same metric written both ways (`schwarzschild_ascii.od` and `schwarzschild_latex.od`, `oderom-cli/tests/fixtures/`).

```
connection Gamma on flat2 {
  [x,y,y] = x
}
```

`connection NAME on CHART { ROW, ... }` — the same block structure, but declaring the Christoffel symbols (rank 3) directly, without going through a metric. Christoffel is not a tensor, so there is no head, no declared symmetry; it is stored as a raw grid. A connection needs no `bundle` at all. It is the alternative entrance to `christoffel` / `riemann` / `ricci` with no metric ever declared — but `scalar` / `kretschmann`, which must invert a metric to raise an index, refuse in that case (see 9.5 and chapter 10, example 4).

Components for any tensor

Until recently, `metric` was the only way in the whole language to give a tensor components — every other tensor could have a signature (`head`) and nothing else. `tensor` lifts that:

```
tensor p : TM on mink { [t] = E, [x] = p1 }

tensor F : TM*, TM* symmetry antisymmetric on mink {
```

```
  [t,x] = -Ex, [t,y] = -Ey, [t,z] = -Ez,
  [x,y] = Bz,  [y,z] = Bx,  [z,x] = By
}
```

The syntax is the signature of a `head` (section 9.3), followed by the chart and component block of a `metric`. Neither half is new; the declaration is the seam between them, and it declares the head as a side effect — you do not write both.

The variance is the point. `TM` without an asterisk is contravariant, an index upstairs, p^{μ}; `TM*` is covariant, downstairs. That mark is what lets section 9.19 evaluate `p[a] p[a]` without you writing a metric in the expression.

Components you do not write are zero — the same rule `metric` already follows, and it is why the example above omits `[y]` and `[z]`. A declared symmetry governs what you did not write: `F` above gives only `[t,x]`, and `[x,t]` is $-Ex$ because antisymmetry says so, not because anyone typed it.

One chart per declaration. The same tensor in two charts is two declarations, and it is better that way: it is you stating that you know the components in both places, rather than the program guessing one from the other by transport.

Rank 0 is refused, and the message says where to go: a scalar with a value is an alias (`S := EXPR`, section 9.9), and a second way to say the same thing does not pay for itself.

Defining a tensor by an expression

Components can also be *computed* rather than written out:

```
tensor T^{\mu\nu} := A^\mu B^\nu
tensor S^{\mu\nu} := T^{\mu\nu} + T^{\nu\mu}
```

The left-hand side gives the name, the order of the indices and the variance of each. The right-hand side gives everything else: the bundle and dimension of each index come from the slots of the factors it uses, and the chart comes from those factors too. That is why there is no `on CHART` here — repeating it would be a second source for a fact the expression already fixes.

The evaluation happens at the declaration, not at the use, which is what makes `T` components like any other `tensor`. A definition may use another declared above it, since declarations are read top to bottom. Writing a variance or an index count that disagrees with what the expression produces is an error, naming the index.

Such a tensor carries no declared symmetry. $A \otimes B$ has none, and inferring one from the computed components would be guesswork.

Curvature as a named tensor

The third form of `tensor` takes a curvature query as its right-hand side:

```
tensor G  := einstein g
tensor Rc := ricci g
```

```
tensor R4 := riemann g
tensor C  := weyl g
```

Until this existed, `ricci`, `einstein`, `riemann` and `weyl` were queries that *printed*: the result had no name, and so could not be contracted, compared with `invariant`, exported, or placed on one side of an equation. Now it can:

```
eval G[a,b] G[a,b]
  →  0                    (Schwarzschild is a vacuum)

simplify Rc[a,b] - Rc[b,a]
  →  0
```

That second line is worth a moment. `simplify` never sees a component — it works in abstract indices. It collapses because the declaration *asserts* the symmetry on the head, and the assertion is checked: every cell is written, including the ones symmetry would already determine, and a disagreement would be reported rather than absorbed.

Query	Rank	Variance	Declared symmetry
`ricci`, `einstein`	2	both down	`(1 2)+`
`riemann`, `weyl`	4	all down	`(1 2)- (3 4)- (1 3)(2 4)+`

The metric name is optional when the document declares only one, as everywhere else. Two forms are refused, each for its own reason: `christoffel`, because it is not a tensor (the same refusal the query itself gives); and the five scalar-valued queries (`scalar`, `kretschmann`, `ricciquare`, `gaussbonnet`, `weylsquare`), because `tensor` declares a tensor — an alias (`NAME := EXPR`) is where a named scalar goes.

The bundle and chart come from the metric the query was asked of. Evaluation happens at the declaration, as with the other two forms, so a document that declares this pays the cost once, on load.

9.3 `head` — abstract construction

```
head R : TM*, TM*, TM*, TM* symmetry (1 2)- (3 4)- (1 3)(2 4)+
head eps : TM*, TM*, TM* symmetry antisymmetric
head Rs : on M
```

`head NAME : SLOT, SLOT, ... [symmetry GENERATOR ...]` declares an abstract tensor head against a bundle, with no concrete components — the level of `oderom canon` and `oderom simplify` (chapter 11), not of the curvature queries. Each *SLOT* is a bundle name, alone (contravariant) or followed by `*` (covariant). `symmetry` accepts `antisymmetric`, `symmetric`, or an explicit list of generators — each a sequence of cycles in parentheses (1-based slot positions) followed by `+` or `-` for the sign. `metric` and `connection` never need a hand-written `head`: they declare their own.

Rank 0: a head with no slots

A scalar head — the Ricci scalar, an action, any named invariant — has no slots, so there is no bundle in the declaration to say which manifold it lives on. It says so directly:

```
head Rs : on M
```

The `on MANIFOLD` clause is how a rank-0 head is declared, and it is *only* for rank 0: a head that lists slots takes its manifold from them, and writing both is a syntax error. A rank-0 head is written without brackets in an expression — `Rs`, never `Rs[]` — and it takes a covariant derivative in the ordinary way, with an empty index list before the semicolon:

```
$ oderom simplify --prelude prelude.od "2 Rs T[a,b] - Rs T[a,b]"
Rs T[a,b]

$ oderom simplify --prelude prelude.od "Rs[;a] + Rs[;a]"
2 Rs[;a]
```

A `symmetry` clause on a rank-0 head is refused, because there are no slots for a permutation to act on:

```
error: parse error: `Rs` tem rank 0 (um escalar), entao nao ha slots
  para uma simetria permutar -- remova a clausula `symmetry`
```

9.4 Scalar expressions

There is one expression grammar (`oderom-cli/src/expr_parser.rs`), with two spellings for every token that has a LaTeX equivalent — never two parsers:

Form	ASCII	LaTeX
sum, difference	`a + b`, `a - b`	same
product	`a * b` or juxtaposition (`2M`)	juxtaposition
division	`a / b`	`\frac{a}{b}`
power (integer exponent only)	`a^n`, `a^-n`, `a^{-n}`	`a^{n}`
grouping	`( ... )`	`( ... )` or `\left( ... \right)`
sine, cosine	`sin(x)`, `cos(x)`	`\sin(x)`, `\sin^2(x)`, `\sin{x}`
exponential	`exp(x)`	`\exp(x)`
hyperbolic sine, cosine	`sinh(x)`, `cosh(x)`	`\sinh(x)`, `\cosh^2(x)`
Greek letter	literal identifier (`theta`)	`\theta`, `\phi`

Thirteen functions are known. Five have an atom of their own in the engine — `sin`, `cos`, `exp`, `sinh`, `cosh`. The other eight are the reciprocals and quotients of those five, and enter as exactly that:

Written	Is	Written	Is
`tan(x)`	sin/cos	`tanh(x)`	sinh/cosh
`cot(x)`	cos/sin	`coth(x)`	cosh/sinh
`sec(x)`	1/cos	`sech(x)`	1/cosh
`csc(x)`	1/sin	`csch(x)`	1/sinh

They are not new atoms, and the difference is visible: a result containing a tangent comes back written as `sin(x)/cos(x)`, never as `tan(x)`. This is the same rule aliases follow (section 9.9) — after expansion nothing survives. What you gain for that price is that the derivative needs no new rule and cannot be wrong independently: d(sin/cos) = (cos² + sin²)/cos², which the engine's existing sin²+cos²=1 identity reduces to 1/cos² — the textbook sec². Every one of the eight is checked against its closed form in the suite.

The form `NAME(...)` is always syntactically a function call (never read as juxtaposition); a name outside that list of five is one of two things, never a generic error. If it has no standard mathematical meaning (`f`, `g`, `h`), it is an *indeterminate function* (section 9.10) — an opaque symbol, accepted and representable, with no value or derivative known in advance. If it is a known elementary function not yet implemented as callable (`tan`, `sqrt`, an inverse trigonometric function), it is rejected explicitly — never accepted as if it were indeterminate. The exponent of `^` is always a literal integer, never a subexpression.

`exp` is its own derivative; `sinh` and `cosh` differentiate into each other with no sign change (unlike `sin` / `cos`). They satisfy the crossed identity $\cosh(x)^2 - \sinh(x)^2 = 1$ — the same simplification `sin` / `cos` already perform with sin²+cos²=1, sign flipped. This is what makes the hyperbolic-plane metric of chapter 10 reduce to a constant Ricci scalar of −2 at all. `exp` did *not* get the identity $\exp(a)\cdot\exp(b) = \exp(a+b)$: no real use case in this project needs it (integer powers of the same `exp(...)` already combine without any identity), and it would require merging two distinct atoms rather than reducing the power of one — new machinery, not a small extension of what exists.

9.5 Queries

`QUERY := QUERY_NAME IDENTIFIER? VARIANCE_MARKER?` — the keyword alone, or followed by the name of a declared metric or connection to disambiguate when there is more than one (`kretschmann g`), and optionally an index-variance marker in brackets, always last (`riemann g [up,down,down]` — section 9.16). Without the identifier the usual rule applies: if there is exactly one metric (or connection, when there is no metric), it is used; more than one without disambiguating is an error. `geodesic` and `accel` (sections 9.11 and 9.12) are the two exceptions to this grammar — each requires a second identifier, always last, the name of the affine parameter, and neither accepts a variance marker (they produce equations, not indexed tensor objects).

The twelve, with what each returns — verified by running every one of them against `schwarzschild_ascii.od`, `reissner_nordstrom.od`, `hyperbolic_plane.od` and `connection_only.od`:

Query	Returns	Works from a bare `connection`?
`christoffel`	The Christoffel symbols Γ^{a}_{bc} (label `Gamma`).	Yes.
`riemann`	The Riemann tensor. From a metric, fully covariant, R_{abcd} (the first index is lowered); from a bare connection, in mixed form R^{a}_{bcd}, since there is no metric to lower with. Label `R` either way.	Yes (in mixed form).
`ricci`	The Ricci tensor $R_{bd} = R^{a}_{bad}$ (label `Ricci`) — the contraction needs no inverse metric.	Yes.
`scalar`	The Ricci scalar $R = g^{bd}R_{bd}$ — a single expression, no label.	No — `` `scalar` needs a metric to invert (only a connection was declared) ``.
`kretschmann`	The Kretschmann scalar $R_{abcd}R^{abcd}$ — a single expression, no label.	No — same error.
`geodesic` (9.11)	The geodesic equations, one per coordinate. No label — a list of equations, not tensor components.	Yes — it needs only Γ, never g_{ab}.
`accel` (9.12)	The same equations solved for each coordinate's second derivative. The form a numerical integrator consumes directly. No label.	Yes — same reason.
`einstein` (9.13)	The Einstein tensor $G_{ab} = R_{ab} - \tfrac{1}{2}g_{ab}R$, fully covariant, listed by independent components exactly like `ricci` (label `G`).	No — g_{ab} appears literally in the formula, and R already needs g^{bd}.
`ricciscalar` (9.14)	The invariant $R_{ab}R^{ab}$ — a single expression, no label.	No — both Ricci indices must be raised.
`gaussbonnet` (9.14)	The Gauss–Bonnet/Euler density $R_{abcd}R^{abcd} - 4R_{ab}R^{ab} + R^2$ — a pure recombination of the three scalars, no label.	No — inherited from the three it combines.
`weyl` (9.15)	The Weyl (conformal) tensor C_{abcd}, fully covariant, listed by independent components exactly like `riemann` (label `C`).	No. Also refuses in exactly 2 dimensions.
`weylsquare` (9.15)	The invariant $C_{abcd}C^{abcd}$ — a single expression, no label.	No — same as `weyl`, whose dimension barrier it inherits too.

Two further query words are not curvature commands and take no metric of their own:

- `export TARGET QUERY` (section 9.17) wraps any of the twelve above, emitting it in another tool's syntax instead of this project's typography. It is not a thirteenth query; it is an additive wrapper, and it composes with the variance marker of 9.16.
- `simplify EXPRESSION` (section 9.18) is the other half of the program entirely: abstract indices, not components. It never looks at a metric's components and never computes curvature.

There is no holonomy query. The numerical integration of geodesics and parallel transport (`oderom-components::holonomy`) is wired to no keyword, no notebook block and no CLI subcommand — it is exercised only by a dedicated Rust test, with a hand-pinned mesh. `geodesic` and `accel` are the symbolic path: they write and isolate algebraically, and never integrate numerically. The two never blur; each lives in its own module.

9.6 Output targets

`Target::Unicode` (default), `Target::Latex`, `Target::Json`. In the notebook the choice is not the user's: each result computes and stores both texts, Unicode and LaTeX, at the same time (`EntryResult`, `oderom-session/src/entry.rs`) — the window always typesets the LaTeX through KaTeX. Only the command line exposes the choice, through `--target` (chapter 11). Despite the name, `Unicode` does not convert Greek names to glyphs — `theta` stays "theta" (plain console text); it is the `Latex` target that produces `\theta`, `\sin\left(...\right)`, `\frac{...}{...}`. Verified by running the same query with both flags:

```
--target unicode:
  Gamma[^theta,phi,phi] = -sin(theta)*cos(theta)

--target latex:
  \Gamma^{\theta}_{\phi \phi} = -\sin\left(\theta\right) \cos\left(\theta\right)
```

`Target::Json` produces a serialised expression tree (for a lone scalar) or an object with the component summary (for `christoffel` / `riemann` / `ricci`) — not a text format for reading, a format for another program to consume. Unlike the other two it keeps raw integer indices (`"indices":[2,3,3]`), and for the same reason it did not get the typography described below.

`--target mathematica` and `--target sympy` also exist as values on the ordinary CLI queries — a quick way to look at the translated syntax without going through the `export` grammar. The real difference: no reserved-word collision detection or renaming, and none of the `symbols(...)` line SymPy needs, both of which `export` adds. `--target sympy` is a quick look, never the "paste it and it runs" guarantee that only `export` makes.

9.7 Component typography and click-to-copy

A query that lists components never shows a raw numeric index — `R_{0,1,0,1}` appears nowhere. The indices are the chart's own coordinate names (Greek letters such as `\theta` in the LaTeX target, spelled out in Unicode), separated by a single space in LaTeX — `R_{t r`

`t r}` — never by a comma. The label itself goes through the same conversion: `christoffel` is `\Gamma`, the real Greek letter, not the five loose Roman letters `Gamma` (which KaTeX would typeset as five multiplied variables). The Unicode target, having no mathematical typography at all, keeps the comma (`R[t,r,theta,phi]`) and the spelled-out label.

The space between names in LaTeX is not cosmetic — it is what stops a Greek macro from swallowing the next letter. `\theta` is a control word: TeX reads the backslash and keeps consuming letters while there are letters, so `\theta` immediately followed by `r` is read as a single macro `\thetar`, which does not exist, and KaTeX shows it raw in red instead of typesetting it. A space ends the macro at the right place without changing the visual result (math mode ignores whitespace for spacing). That exact bug — `\thetar` showing raw on screen — is what exposed both problems at once: the `Gamma` label with no backslash, and the concatenation with no separator.

An index is shown in exactly the variance the component was actually computed in — never adjusted for looks. `riemann` from a metric has already had its first index lowered, so it comes out fully covariant, `R_{t r t r}`; from a bare connection it comes out mixed, `R^{a}_{bcd}`. `christoffel` is always mixed, `\Gamma^{a}_{bc}`; `ricci` is always fully covariant, whether it came from a metric or a connection.

The annotation saying how many raw indices an independent component stands for (`(N components by symmetry)`) never shares a line with the formula — it is an ordinary English sentence, never mathematics, so it never goes through the LaTeX typesetter with the formula. In the notebook it appears as small grey text beside the formula (never inside what gets copied); on the command line and in the REPL, as its own indented line below. The same separation applies to the trailing summary lines.

In the notebook, clicking any component of a result copies that component's clean LaTeX to the clipboard — just the typeset equality (`R_{t r t r} = \frac{...}{...}`), ready to paste into a LaTeX document: never the symmetry annotation, never the summary line, never another component, never the whole block. A small "copied" badge appears briefly beside the component clicked. Behind that interaction, each result reaches the JavaScript with the formula and the annotation as separate fields (never one concatenated string) — which text is formula and which is caption is decided entirely in Rust (`oderom_components::RenderedComponent`), before the screen exists.

9.8 Common error messages

All captured by running the real parser and CLI against a file built to provoke each one:

> **The messages are in two languages, and this manual does not translate them.** The project's convention is that an error names, in Portuguese, what the author wrote wrong — that is the language the person at the keyboard is writing the document in. The English messages below are older ones, from before that convention was settled, and they are quoted as they really appear. Where this manual shows a message, it is the real bytes: translating one would make the page pleasant and the search for it fail.

Situation	Real message
Derivative mark (`'`) on a known function	`` `sin` is a known function with a known derivative -- write `diff(sin(...), var)`, not `sin'(...)` or `sin_{...}(...)` ``
Prime notation on an indeterminate function of several arguments	`` `h'` (prime notation) is ambiguous for a function of 2 arguments -- use `h_{var,...}(...)` instead ``
Derivative subscript naming a non-argument	`` `h`'s derivative subscript names `x`, which is not one of its own arguments ``
Derivative mark with no argument list after it	`` `f` with a derivative mark must be followed by an argument list, e.g. `f'(r)` or `f_{r}(r)` ``
Chart with the wrong number of coordinates	`` chart `c` has 2 coordinates, but manifold `M` has dimension 3 ``
Name already declared	`` chart `c` is already declared ``
No metric and no connection in the file	`no metric or connection found in the file`
More than one metric, not disambiguated	`the file declares more than one metric (g1, g2); pick one with --metric`
A metric-requiring query with only a connection	`` `scalar` needs a metric to invert (only a connection was declared) `` (same, with the command's own name substituted)
`weyl` / `weylsquare` in 2 dimensions	`the Weyl tensor is not defined in 2 dimensions (the standard formula's 1/(n-2) term is undefined at n=2)`
Unrecognised opening keyword (notebook block)	`expected a declaration (manifold/bundle/head/chart/metric/connection/axiom/tensor/transition/equation), an alias (NOME := expression), or a query (christoffel/riemann/ricci/scalar/kretschmann/geodesic/accel/einstein/riccisquare/gaussbonnet/weyl/weylsquare/export/simplify/eval/invariant/solve), found ...`
A name never declared as an alias	none — it stays an ordinary free variable, as it always was
An alias used before its own declaration	`` alias `f` is used before its own `f := ...` declaration ``

Redeclaring an alias name	`alias `f` is already declared`
`geodesic` / `accel` with no parameter name	`` `geodesic` needs an affine parameter name, e.g. `geodesic tau` or `geodesic g tau` ``
Affine parameter colliding with a chart coordinate	`affine parameter `r` collides with chart coordinate `r` -- geodesic needs a distinct name`
Affine parameter colliding with a free variable	`affine parameter `M` collides with a free variable already used in this metric/connection -- geodesic needs a distinct name`
`symmetry` on a rank-0 head	`` `Rs` tem rank 0 (um escalar), entao nao ha slots para uma simetria permutar -- remova a clausula `symmetry` ``
Ambiguous bundle for a covariant derivative	`cannot tell which bundle a covariant derivative acts on over manifold `M`: more than one bundle and none is marked (candidates: TM, E) -- mark exactly one with `tangent``

> **Mixed languages, on purpose and by accident.** Messages written for the abstract-index half are in Portuguese; the older component-engine messages are in English. Both are quoted above exactly as emitted. The mixture is a known inconsistency in the program, not in this manual.

9.9 Expression aliases

```
f := 1 - 2*M/r + Q^2/r^2

manifold M dim 4
bundle TM on M dim 4
chart schw on M coords (t, r, theta, phi)
metric g on schw bundle TM {
  [t,t] = -f,
  [r,r] = 1/f,
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
```

`NAME := EXPRESSION` declares an alias: a short name for a scalar subexpression that would otherwise be repeated in full in every component using it — above, the Reissner–Nordström metric with $f(r)$ written once. An alias block classifies as a declaration and so takes part in the notebook's declaration reconstruction: an alias's scope is the whole notebook, top to bottom, from the point of its declaration — it is not local to a block. An alias may reference another alias declared before it.

An alias is pure syntactic sugar, expanded by textual substitution inside the expression parser, before anything reaches the normalisation engine. Nothing survives the expansion:

the tree from `[t,t] = -f` is byte-for-byte the one `[t,t] = -(1 - 2*M/r + Q^2/r^2)` produces by hand — verified by comparing the `kretschmann` of both forms (`reissner_nordstrom.od` against `reissner_nordstrom_alias.od`): the same result, byte identical, `(-96*M*Q^2*r + 48*M^2*r^2 + 56*Q^4)/r^8`. Declaring or expanding an alias never triggers a computation on its own.

An alias is used without parentheses — just `f`, never `f(...)`. The parenthesised form is always a function call (sections 9.4 and 9.10): `f(r)` is the indeterminate function *f* applied to *r*, never read as an alias call, even if `f` is also declared as an alias in the same file — the two never collide because they live in different syntactic positions. An alias name must be declared before use (a forward reference is an error, not a free variable); a name never declared as an alias anywhere stays an ordinary free variable, exactly as *M* or *Q* always were. Redeclaring an alias name is an error.

9.10 Indeterminate functions and derivatives

This section only represents and differentiates — it never solves anything. There is no query and no operation that resolves an indeterminate function to closed form or integrates an equation. What exists is: writing `f(r)` without giving a definition, and correctly differentiating an expression containing it.

```
metric g on schw bundle TM {
  [t,t] = -f(r),
  [r,r] = 1/f(r),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
```

`f(r)` declares an indeterminate function: a new functional symbol, undefined, which the engine treats as an opaque leaf — never evaluated, never expanded, never given an invented algebraic identity. It is deliberately different from an alias and from a known function, and the three never collapse into one: an alias expands to a known expression; a known function has a value and a derivative; an indeterminate function has neither — its only property is having derivatives, which are themselves opaque symbols (f', f''), never simplified beyond the chain rule. The same name can be all three in different positions with no collision: `f` alone is the alias (if declared), `f(r)` is always the indeterminate function, `sin(x)` is always the known one — syntactic position decides, never a silent reinterpretation.

Not every name outside the list of five becomes indeterminate, though. A reserved set of names with standard mathematical meaning is kept out of both groups and rejected explicitly. The reason is concrete: before that reservation existed, `tan(theta)` in a metric was accepted as an indeterminate function — and differentiated as `tan'`, some opaque symbol, never the real derivative $\sec^2$. For someone who wrote `tan(theta)` meaning the actual tangent, the result looked plausible and was silently wrong — exactly the class of failure (a number that passes for correct) this project treats as worse than a loud error.

The reserved set today: both common spellings of every inverse trigonometric and hyperbolic function (`asin`/`arcsin` and so on); `log` and `ln`; `sqrt`; `atan2`; `abs`. Left out are

the non-differentiable or discrete functions with no natural derivative (`floor`, `ceil`, `sign`) — outside the scope of a project centred on differentiable curvature.

The eight reciprocals (`tan`, `cot`, `sec`, `csc` and their hyperbolic analogues) were reserved until recently and are now implemented — section 9.4. They needed no new machinery, being quotients of functions the engine already had. What is still reserved needs a genuinely new atom, and splits in two: `log` and `atan` have *rational* derivatives (1/x and $1/(1+x^2)$) and would fit the rational form the engine is built on; `sqrt` — and every inverse function whose derivative contains one — is algebraic, not rational, which is a different kind of change and not a larger version of the same one.

The reservation means "not yet", never "forbidden forever": each name in it is a real function this project may implement as known in a future round (the same way `exp`, `sinh` and `cosh` went from unavailable to known). Calling a reserved name with parentheses is an explicit error: `` `tan` is a known elementary function, not yet implemented as callable -- reserved names cannot be used as an indeterminate function ``. A reserved name used *without* parentheses stays an ordinary free variable.

The derivative of a one-variable indeterminate function uses prime notation only — `f'(r)`, `f''(r)`, one prime per order, exactly as on paper. Prime notation is ambiguous for a function of several variables, so a function such as `h(t, r)` uses a braced subscript, always comma-separated, naming the variable(s) to differentiate by: `h_{t}(t, r)` is $\partial h/\partial t$, `h_{t,r}(t, r)` the mixed partial. The comma is always required inside the braces (never the run-together `h_{tr}` that `NAME_{...}` accepts for a metric component index) — this grammar has no chart in scope to split a run-together name back into coordinates.

Differentiating an expression containing an indeterminate function produces the correct chain rule, generalised over the order rather than fixed to a sibling function: the derivative of *f* at order *k* is *f* at order *k*+1. `diff(f(r), r)` gives `f'(r)`; `diff(f(r)^2, r)` gives `2*f(r)*f'(r)`; `diff(f(r)*g(r), r)` gives `f'(r)*g(r) + f(r)*g'(r)`; and the derivative of `f(r)` with respect to a coordinate other than *r* is zero — not as a special case, but because the chain rule multiplies by d*r*/d(other) = 0. Verified by running the real engine: a metric with an unknown `f(r)` is accepted, and `christoffel` on it produces real symbols containing `f(r)` and `f'(r)`.

9.11 `geodesic` — the geodesic equation

This query only writes the equation — it never solves it. There is no separate mode for null or timelike geodesics, and none for non-affine parametrisation: the equation $\ddot{x}^a + \Gamma^a{}_{bc}\dot{x}^b\dot{x}^c = 0$ is the same in all three cases. What changes is the initial condition, and an initial condition is not something this query takes or produces.

Syntax: `geodesic PARAMETER` or `geodesic TARGET PARAMETER` — the affine parameter's name (typically `tau`) always comes last; the target, when present, comes before it. Unlike the other queries the parameter is not optional: `geodesic` alone is an error naming the two valid forms. `accel` shares this grammar exactly.

The collision rule. The affine parameter's name may not match any coordinate of the chart in use, nor any free variable already used in the metric or connection. Matching would make the equation genuinely ambiguous — `t(r)` would be at once "the coordinate *t*

as a function of the parameter *r*" and "the free coordinate *t* times the free coordinate *r*" — so both cases are refused explicitly, never guessed:

```
$ oderom geodesic schwarzschild_ascii.od --param r
error: parse error: affine parameter `r` collides with chart coordinate `r`
  -- geodesic needs a distinct name

$ oderom geodesic schwarzschild_ascii.od --param M
error: parse error: affine parameter `M` collides with a free variable already
  used in this metric/connection -- geodesic needs a distinct name
```

Two incarnations of the same coordinate name. Inside one geodesic equation, *r* means two different things at once: inside a Christoffel coefficient it is the ordinary free coordinate; in the velocity/acceleration term it is *r* as a function of the affine parameter, with a derivative of its own. The two never collapse: they are different tree shapes from construction — the same indeterminate function machinery of section 9.10, applied here to the coordinate name itself. What changes between output targets is only the typography of that second incarnation: in Unicode, spelled out with the parameter explicit (`t'(tau)`), because a dot over a letter does not survive plain text; in LaTeX, dot notation with the parameter implicit (`\dot{t}`, `\ddot{r}`).

```
--target unicode:
  (2*M*r*t''(tau) - 2*M*r'(tau)*t'(tau) - r^2*t''(tau))/(2*M*r - r^2) = 0

--target latex:
  \frac{(2 M r \ddot{t} - 2 M \dot{r} \dot{t} - r^{2} \ddot{t})}{(2 M r - r^{2})} = 0
```

Since it needs only Γ and never g^{ab}, `geodesic` works from a bare `connection` just as `christoffel` / `riemann` / `ricci` do.

9.12 `accel` — the geodesic solved for the acceleration

This is algebraic isolation, not ODE solving: `accel` never integrates, never produces a trajectory, never chooses an initial condition. Each geodesic equation is linear in its own second derivative, so "solve for $\ddot{x}^a$" means dividing by the coefficient of $\ddot{x}^a$ and moving the rest across — nothing more. It is a separate command from `geodesic`, never a mode of it.

Syntax and collision rule are identical to `geodesic`. The output is $\ddot{x}^a = f^a(x, \dot{x})$, one equation per coordinate — the form a numerical integrator consumes directly.

Before dividing, each equation has the form $C_a \cdot \ddot{x}^a + R_a = 0$. `accel` checks structurally — never assumes — that $\ddot{x}^a$ appears at degree exactly 1 (never squared, never inside another function, never in more than one factor of the same product), via `oderom_expr::isolate_linear`, a purely structural analysis of the expression tree. That linearity is not a property of the chosen metric — it is a mathematical fact about what a geodesic equation is, so no genuine geodesic equation can fail it. Accordingly a violation there is not a catchable `Result` but an assertion that fails loudly: no legitimate input can reach it.

The coefficient C_a is different: it could in principle vanish for a degenerate metric or chart, which is a property of the choice, not of the equation. So it stays a real catchable error naming the coordinate, rather than an assertion. Today, for every real metric and connection, that coefficient is always exactly 1 and never zero — the check remains because it guards against a future change in how the equation is assembled, not because a real case needs it.

The output goes through the same canonical rational normalisation as the rest of the engine — never a fraction inside a fraction. The radial Schwarzschild equation, whose Christoffel coefficient is not a trivial monomial:

```
--target unicode:
  r''(tau) = (-M*r^2*r'(tau)^2 + M*r^2*t'(tau)^2 + 4*M*r^4*sin(theta)^2*phi'(tau)^2
    + 4*M*r^4*theta'(tau)^2 - 4*M^2*r*t'(tau)^2 - 4*M^2*r^3*sin(theta)^2*phi'(tau)^2
    - 4*M^2*r^3*theta'(tau)^2 + 4*M^3*t'(tau)^2 - r^5*sin(theta)^2*phi'(tau)^2
    - r^5*theta'(tau)^2)/(2*M*r^3 - r^4)
```

A single reduced fraction, not a fraction of fractions, despite the non-trivial denominator — checked both by inspection and by a dedicated acceptance test.

9.13 `einstein` — the Einstein tensor

$G_{ab} = R_{ab} - \tfrac{1}{2}g_{ab}R$, fully covariant, listed by independent components exactly as `ricci` is, with label `G`. It needs a real metric: g_{ab} appears literally in the formula, and R already needs g^{bd}. Against a vacuum solution it is identically zero in every component; against Reissner–Nordström it is not (chapter 10, example 7).

9.14 `riccisquare` and `gaussbonnet`

`riccisquare` is $R_{ab}R^{ab}$, a single expression. `gaussbonnet` is the Gauss–Bonnet/Euler density $R_{abcd}R^{abcd} - 4R_{ab}R^{ab} + R^2$ — a pure recombination of `kretschmann`, `riccisquare` and `scalar`, and it inherits their requirement of a metric.

9.15 `weyl` and `weylsquare` — the Weyl tensor

The conformal tensor C_{abcd}, fully covariant, listed like `riemann` with label `C`, and its invariant $C_{abcd}C^{abcd}$. Both need a metric. Both also refuse in exactly two dimensions, and the message says why rather than returning something meaningless:

```
the Weyl tensor is not defined in 2 dimensions
  (the standard formula's 1/(n-2) term is undefined at n=2)
```

9.16 The index-variance marker

By default each query returns its tensor in a fixed variance (the table in 9.5). An optional marker in brackets, last in the query, asks for whichever variance you want instead — one `up` or `down` per index, in order:

```
riemann [up,down,down,down]
ricci g [up,up]
weyl [down,down,up,down]
```

`up` raises that index with g^{ab} if it is not already contravariant; `down` lowers it with g_{ab} if it is not already covariant — an index already in the requested variance is left alone. The target still comes before the marker: `ricci g [up,up]`, not `ricci [up,up] g`. Without a marker nothing changes: the marker is additive, never a change of default behaviour.

```
oderom> riemann [up,down,down,down]
inverting the metric...
computing Christoffel symbols...
[1] riemann [up,down,down,down] -- [OK]
R[^t,r,t,r] = 2*M/(-2*M*r^2 + r^3)
R[^t,r,r,t] = 2*M/(2*M*r^2 - r^3)
R[^t,theta,t,theta] = M/(-r)
...
232 independent components identically zero
```

One visible and expected difference: a marked query shows every independent component raw, without the symmetry grouping that an unmarked `riemann` / `ricci` shows. The symmetry group of a generic mixed-variance tensor is not necessarily that of the fully covariant one, so the marked query does not try to reuse it. The values are identical either way; only the number of displayed lines changes.

Which queries accept the marker: `riemann`, `ricci`, `einstein`, `weyl` — the four real tensors. The marker always needs a declared metric, never a bare connection (even for `riemann` / `ricci`, which unmarked work from both), because raising and lowering use g^{ab}/g_{ab}. The number of `up` / `down` must match the tensor's rank exactly — never silently padded or truncated. A scalar has no indices, and the marker on one is a clear error rather than being silently ignored:

```
oderom> scalar [up]
error: `scalar` is a scalar -- it has no indices, so an index-variance
  marker doesn't apply. Use it on riemann, ricci, einstein, or weyl
  instead.
```

9.17 `export` — symbolic translation to Mathematica and SymPy

`export TARGET QUERY` wraps any of the twelve queries and emits the result in an external tool's syntax instead of this project's typography:

```
export sympy kretschmann
export mathematica riemann g [up,down,down,down]
export sympy geodesic tau
```

It is not a thirteenth query — it is an additive wrapper, and it composes with the variance marker of 9.16 and with everything else the wrapped query accepts. Unlike `--target`

`sympy`, `export` also detects and renames collisions with the target language's reserved words and emits SymPy's `symbols(...)` line, which is what makes the output paste-and-run rather than just readable.

In the notebook, the Export button (chapter 3) writes one of these lines into a new block instead of requiring you to remember the syntax. It writes; it does not run.

9.18 `simplify` — abstract indices

Everything up to here computes *components*: give it a metric, get numbers and closed-form expressions indexed by coordinate names. `simplify` is the other half of the program. It works on tensor expressions in *abstract* indices, where `R[a,b,c,d]` is not a component of anything — *a*, *b*, *c*, *d* are slot labels, and the only facts available are the ones declared with `head`.

```
simplify R[a,b,c,d] + R[b,a,c,d]
```

What it does, in order: canonicalise every monomial under its head's declared symmetry group, then collect like terms. That alone decides a large class of questions. Given the Riemann symmetries of `prelude.od`:

```
$ oderom simplify --prelude prelude.od "R[a,b,c,d] + R[b,a,c,d]"
0

$ oderom simplify --prelude prelude.od "R[a,b,c,d] - R[c,d,a,b]"
0

$ oderom simplify --prelude prelude.od "3 R[a,b,c,d] + -1 R[a,b,c,d]"
2 R[a,b,c,d]

$ oderom simplify --prelude prelude.od "eps[a,a,c]"
0
```

The last one is antisymmetry doing its job: a repeated slot in a totally antisymmetric head forces zero.

Notation

Monomials are juxtaposed, not multiplied with `*`: `g[a,b] R[b,c,d,e]`, never `g[a,b]*R[b,c,d,e]`. A repeated index within a monomial is a contraction. A coefficient may precede a monomial (`2 R[a,b,c,d]`, `-1 R[a,b,c,d]`, `1/3 R[a,b,c,d;e]`). A semicolon introduces covariant-derivative indices: `T[a,b;c]` is $\nabla_c T_{ab}$, and `g[a,b;c,d]` is the second derivative. A rank-0 head takes no brackets at all (`Rs`), and its derivative writes the empty index list explicitly (`Rs[;a]`) — see 9.3.

Axioms are declared, never inferred

Canonicalisation under declared symmetries cannot reach a multi-term identity such as the first Bianchi identity: it relates three distinct monomials, and its cyclic permutation is

not itself a symmetry of Riemann. Riemann's slot-symmetry group has order 8; the cyclic permutation has order 3, and 3 does not divide 8 — by Lagrange's theorem it simply cannot be a member, not merely happens not to be. So such identities are *declared*, on the command line, one flag each:

Flag	Declares
`--bianchi HEAD`	The first (algebraic) Bianchi identity for that head: $R_{a[bcd]} = 0$.
`--bianchi2 HEAD`	The second (differential) Bianchi identity: $R_{ab[cd;e]} = 0$. A separate axiom — neither implies the other. It takes the *base* head; the differentiated form is found structurally, so nobody writes `R;1`.
`--metric HEAD`	That head is a metric, so contracting it against an index raises or lowers rather than leaving a factor behind.
`--metric-compatible HEAD`	$\nabla_c g_{ab} = 0$ — a property of the connection being Levi-Civita, not of the metric's shape, so it is declared and not deduced. Distinct from `--metric`, which contracts an *undifferentiated* metric: a derivative of it is a different object, and assuming it vanishes is exactly this flag's job.

```
$ oderom simplify --prelude prelude.od --bianchi R \
    "R[a,b,c,d] + R[a,c,d,b] + R[a,d,b,c]"
0

$ oderom simplify --prelude prelude.od --bianchi2 R \
    "R[a,b,c,d;e] + R[a,b,d,e;c] + R[a,b,e,c;d]"
0

$ oderom simplify --prelude prelude.od --metric g "g[a,b] R[b,c,d,e]"
R[a,c,d,e]

$ oderom simplify --prelude prelude.od --metric-compatible g "g[a,b;c]"
0
```

Without the flag, the identity does not hold — and must not. The negative control matters as much as the positive one: a simplifier that returns `0` too eagerly is worse than one that returns nothing.

```
$ oderom simplify --prelude prelude.od "R[a,b,c,d] + R[a,c,d,b] + R[a,d,b,c]"
R[a,b,c,d] + -1 R[a,c,b,d] + R[a,d,b,c]
```

Two engines, and a way to check one against the other

`--engine` selects how the decision is made:

Value	Meaning

`egraph` (default)	Equality saturation over an e-graph, with cost-based extraction. Fast, and heuristic: it matches, it does not decide.
`linear`	Per-stratum linear algebra over $\mathbb{Q}$: enumerate the stratum's canonical basis, build the identity rows, row-reduce, and ask whether the expression's vector lies in the span. That is a decision, not a heuristic.
`both`	Runs both and fails if they disagree.

`both` is not a production mode — it is test coverage, closing the hole that the fast path is heuristic and will stay that way. An unknown value is refused by name:

```
$ oderom simplify --engine=xyz ...
error: parse error: --engine desconhecido: `xyz` (use egraph, linear ou both)
```

In the notebook

`simplify` is a query keyword like any other, so a block whose first word is `simplify` runs on `Shift`+`Enter`. The `head` declarations come from the notebook's own declaration blocks, which already accept the word `head`.

```
manifold M dim 4
bundle TM on M dim 4
head R : TM*, TM*, TM*, TM* symmetry (1 2)- (3 4)- (1 3)(2 4)+
%%
simplify R[a,b,c,d] + R[b,a,c,d]
```

Declaring an axiom in a document

The flags above are one of two ways to say the same thing. The other is a declaration, which is what a notebook block uses — a block has no command line:

```
axiom bianchi R
axiom bianchi2 R
axiom metric g
axiom compatible g
```

An axiom is a declaration and not an option, because it is a statement about the *geometry*: "the head R satisfies the first Bianchi identity" is the same kind of claim as the slot symmetry the `head` line already declares. It follows that its scope is the whole document, top to bottom; that it is saved in the `.od` file along with everything else; and that it means the same on all three surfaces, since all three read the same document.

The fourth is spelled `compatible` rather than `metric-compatible` for a mechanical reason: to the tokenizer a hyphen is its own symbol, and so is an underscore, so either spelling would arrive as three tokens instead of one word.

The head is resolved at the point of declaration, so a mistyped name fails on the line that wrote it rather than in a later query that has nothing wrong with it. Flags and declarations

combine: a document may declare its own axioms and still receive more on the command line.

`simplify` with no expression after it classifies as a query (the first word is the verb) and fails at execution, showing the expected shape rather than complaining about a token:

```
`simplify` precisa de uma soma de monomios,
  p.ex. `simplify R[a,b,c,d] + R[b,a,c,d]`
```

9.19 `eval` — the same expression, evaluated

`simplify` and `eval` take the same expression and ask different questions of it:

```
simplify p[a] p[a]   ->  p[a] p[a]       canonical form, no numbers
eval     p[a] p[a]   ->  -E^2 + p1^2     because the components exist
```

Same syntax, same parser, same monomial. What changes is what happens to it afterwards. Where `simplify` works on what the `head` declarations assert, `eval` works on what the `tensor` declarations contain (section 9.2).

A complete document is five lines:

```
manifold M dim 4
bundle TM on M dim 4
chart mink on M coords (t, x, y, z)
metric g on mink bundle TM { [t,t] = -1, [x,x] = 1, [y,y] = 1, [z,z] = 1 }
tensor p : TM on mink { [t] = E, [x] = p1 }
```

and then `eval p[a] p[a]` returns `-E^2 + p1^2`. The result is a component grid whose rank is the number of free indices — rank 0 for a scalar, so `eval T[a,b] V[b]` comes back as a list, one entry per value of *a*.

The textbook spelling

Indices may also be written the way they appear in a book, with the variance on the index itself:

```
eval g_{\mu \nu} V^\mu V^\nu
eval V^\mu W_\mu
eval T^{\mu}_{\nu}
```

A single index needs no braces, as in LaTeX. A Greek macro and a plain letter are the same label — `\mu` becomes `mu`, the same equivalence `\theta` and `theta` already have in scalar expressions — so `V^{a} W_{a}` and `V^\mu W_\mu` are the same monomial. The bracket form remains the complete one: covariant derivatives (`;`) and symmetrisation groups are written only there.

The variance you write is checked, not applied. Writing `V_\mu` for a head declared contravariant is an error, naming the index and the head. This is deliberate: a physicist

writing `V_\mu` means "lower this index with the metric", and doing that silently would settle the semantics of raising and lowering by notation without anyone having decided it. An error leaves that door open; a silent transformation closes it. To lower an index today, write the metric: `g[a,b] V[a]`.

> **Latin indices inside braces need a separator.** `g_{\mu\nu}` is unambiguous because a macro delimits itself. `g_{ab}` is not — one index called `ab`, or two? Write `g_{a b}` or `g_{a,b}`. It is the same ambiguity the derivative subscript already faced (`h_{t,r}`), and the same answer.

Moving an index

Writing an index in the variance opposite to the declared one moves it with the metric — which is what the notation means:

```
eval u_\mu        with u declared TM   ->  g_{\mu\rho} u^\rho
eval A^\mu        with A declared TM*  ->  g^{\mu\rho} A_\rho
```

This is sugar, expanded in the parser: the metric enters as a real factor with a fresh dummy index, and nothing downstream — evaluation, the type judgment, canonicalisation — sees anything but an expression already honest about variance. The price is the same one aliases pay: the output shows the metric you did not write.

Without a declared metric there is nothing to move an index with, and the attempt is refused rather than guessed at — moving an index *is* a claim about the geometry. Where a document declares more than one metric, it is refused too, naming the ambiguity.

One consequence worth knowing, because it falls out rather than being decided: the mixed form of the metric, `g^{\mu}{}_{\nu}`, expands to `g^{\mu\rho} g_{\rho\nu}` — the Kronecker delta. Contracted with a vector it returns the vector.

Symmetrising and antisymmetrising

Round brackets symmetrise an index group; square brackets antisymmetrise it — the standard notation:

```
eval P_{(\mu\nu)}     ->  1/2 (P_{\mu\nu} + P_{\nu\mu})
eval P_{[\mu\nu]}     ->  1/2 (P_{\mu\nu} - P_{\nu\mu})
```

The bracket form has the same groups — `P[(a,b)]` and `P[[a,b]]` — and produces identical components. Both normalise by $1/k!$ and both refuse nesting rather than guessing at it.

The inverse metric

The metric written with raised indices is its inverse:

```
eval g^{\mu\nu} A_\mu A_\nu
```

This is not "the metric with its indices raised" — raising the indices of g with g would give g back. `g^{ab}` is the inverse by definition, and that is the universal convention, so it is a head of its own rather than a reinterpretation of the original one.

The permission belongs to the metric alone. Any other head written in the opposite variance is still an error: raising and lowering an arbitrary index by notation does not exist yet. The mixed form, `g^{\mu}{}_{\nu}`, is refused too — it is neither the metric nor its inverse but the Kronecker delta, and naming it that way would settle a third convention nobody asked for.

Where the abstract meets the concrete

This is the part worth understanding, because it is what makes the example above work without a metric written anywhere in it.

Abstract-index notation here contracts without looking at variance: `R[a,b,c,d] R[a,b,c,d]` is legitimate, with the metric implied. In components that does not exist — you may only sum an upstairs index against a downstairs one. So `eval` decides pair by pair:

The contracted pair	What happens
dual variances (`TM` with `TM*`)	summed directly
both `TM*`	g^{ab} is inserted — raising one index
both `TM`	g_{ab} is inserted

And that is why $p^{\mu}p_{\mu}$ works: `p` is declared `TM`, as physics asks; both slots of `p[a] p[a]` are contravariant; g_{ab} enters on its own, and the sign comes from where it should, from $g_{tt} = -1$.

> **No emergency metric.** If the chart has no declared metric and a pair needs one, `eval` refuses and says so. Contracting two indices of the same variance *is* a claim about the geometry, and without a declared metric it has no value.

What it refuses, and why

Situation	What happens
A head with a signature but no components	Refused, naming the head. Returning the symbol would answer a different question — for that one there is `simplify`.
Charts that share no tensor	Refused, naming both. Evaluating in the wrong chart would return plausible, wrong numbers.

Every factor present in more than one chart	Refused as ambiguous, naming the charts. This is what `invariant` (9.20) is for.
A sum whose terms have different free indices	Refused: a sum of tensors of different types is not a tensor.
Same-variance contraction, no metric	Refused, as above.

The active chart is not chosen by syntax. It comes from the tensors in the expression, which already declared it — the one chart common to all of them. When a tensor is declared in several charts (9.20), that common chart is an intersection rather than a single answer, and one factor declared in a single chart is enough to settle it.

9.20 `transition` and `invariant` — is this really a tensor equation?

A tensor equation is supposed not to depend on the coordinates you happened to choose. Every textbook says so; no computer algebra system lets you *ask*. Cadabra, xAct and SymPy all leave the check as something you perform by running the calculation twice and comparing with your eyes.

The ODEROM turns it into one line of program. It takes two pieces: the same object written in two charts, and the coordinate change between them.

The same tensor in more than one chart

A `tensor` — and a `metric` — may be declared more than once, under one name, as long as each declaration names a different chart:

```
tensor p : TM on cart  { [x] = x, [y] = y }
tensor p : TM on shear { [u] = u, [v] = v }
```

The two declarations share a single head. That is the point: this is one geometric object written in two bases, not two objects that happen to be spelled the same. Re-declaring with a different signature is refused — that really would be two objects under one name — and so is repeating the same chart, which is a typing mistake rather than a second basis.

The coordinate change

```
transition cart to shear {
  u = 2*x,
  v = x + y
}
```

Each line gives one coordinate of the destination chart as a function of the source chart's coordinates. Every destination coordinate must appear exactly once.

Three decisions are worth naming. The word is `to` rather than an arrow, because the lexer would read `->` as two separate symbols. Each declaration carries one direction only:

deriving the reverse map would mean solving a symbolic system, and a program that silently attempted it would sometimes fail in ways you could not see. And the transition is your claim about the overlap of the two charts' domains — the ODEROM does not verify that overlap.

Asking the question

```
invariant g_{\mu\nu} p^\mu p^\nu
```

```
cart: x^2 + y^2
shear: -u*v + 1/2*u^2 + v^2
cart -> shear: concordam
invariante
```

The two expressions are visibly different, and that is the whole content of the answer: they are the same number written in two coordinate systems, and the ODEROM checked it rather than asserting it. On the command line a disagreement also exits with status 1, so a script can fail on it.

Free indices, and why variance decides

The question is not restricted to scalars. With free indices left over, each one is carried across by its own factor:

```
T_{i...}(x) = T_{k...}(phi(x)) * d(phi_k)/d(x_i)      covariant index
T^{i...}(x) = T^{k...}(phi(x)) * d(x_i)/d(phi_k)      contravariant index
```

A covariant index is carried by the Jacobian; a contravariant one by its inverse. That difference *is* what the two words mean, and swapping them would return plausible, wrong components. So:

```
invariant p^\mu                  invariant p_\mu

cart:                            cart:
[^x] = x                         [x] = x
[^y] = y                         [y] = y
shear:                           shear:
[^u] = u                         [u] = 1/2*u - 1/2*v
[^v] = v                         [v] = -1/2*u + v
cart -> shear: concordam          cart -> shear: concordam
invariante                       invariante
```

Both are invariant, and the components differ between the two — which is exactly what should happen, and what makes the check worth running. The inverse Jacobian is verified against the identity before it is used, for the same reason the metric inverse is: a wrong inverse does not fail, it answers.

> **On choosing test cases.** A rotation is a poor example here, and this manual deliberately does not use one. A rotation is orthogonal, so the inverse of its Jacobian equals the transpose — and covariant and contravariant would transform identically, hiding any confusion between them. The shear above does not have that symmetry.

What it refuses

Situation	What happens
The expression lives in a single chart	Refused, naming it. There is nothing to compare.
Two charts with no `transition` between them	Refused. Without the map there is no way to compare, and answering "invariant" would be a lie.
Different variance in different charts	Refused: those are two objects under one name.
A transition whose Jacobian does not invert	Refused, naming the entry of the product that failed to give the identity.

9.21 `equation` — a named equation

```
equation eq1 : x + y = 2
equation efe : G_{\mu\nu} = 8 T_{\mu\nu}
equation e   : T[a,b] + U[a,b] = S[a,b]
```

`equation NAME : LEFT = RIGHT` declares an equation and gives it a name. Both sides use the same expression grammar as `simplify`, `eval` and `invariant` — a sum of monomials, in brackets or in LaTeX — or, as in the first line, an ordinary scalar expression.

The name is not only for `solve`. A named equation is a first-class object: it can be referred to, and it stays in the document as a statement about the geometry rather than a step that scrolls away.

The third meaning of `=`

This is worth naming because the symbol is now overloaded three ways. Inside a `metric` or `tensor` block, `=` assigns a component (`[x,x] = 1`); inside `transition`, a coordinate (`u = 2*x`); `:=` defines (an alias, or a tensor). Inside `equation`, `=` *asserts*. There is no ambiguity — the keyword settles it — but the choice was made deliberately, not inherited.

One spelling is genuinely ambiguous and is refused by name: `equation e := ...`, which reads exactly like an alias definition. So is an equation with no `=`, one with more than one (`x = 2 = 3` has two readings and neither is chosen), and one with an empty side.

Where an equation ends

Both sides run to the start of the next declaration, or to the end of the block. It is the same problem the scalar alias and the tensor definition already have — an expression with no closing token — and the same answer.

Coefficients are written by juxtaposition

A trap worth knowing, and one this manual got wrong before the implementation corrected it. In the monomial grammar a coefficient sits next to its factor:

```
equation efe : G_{\mu\nu} = 8 T_{\mu\nu}       ✓
equation efe : G_{\mu\nu} = 8*T_{\mu\nu}       ✗  syntax error at `*`
```

And a *symbolic* coefficient must be a declared head of rank zero, because an undeclared name is `unknown tensor head`:

```
head L : on M
equation esp : Rc_{\mu\nu} = L g_{\mu\nu}
```

This does not apply to a scalar equation such as `x + y = 2`, which uses the ordinary scalar grammar, where `*` is multiplication as usual.

9.22 `solve` — the inverted question

Every other query in this program has the shape "given this, compute that": given the metric, the Christoffel symbols; given the components, the scalar. `solve` is the first that says "this equals that, now tell me what is missing".

The physics asks the inverted question constantly, and the commonest of all is: given a geometry, *what matter produces it?*

```
solve NAME for TARGET
```

The word `for` is required. It is what makes the line readable aloud, and what stops `solve eq1 y` from looking correct.

A scalar unknown

```
equation eq1 : x + y = 2
solve eq1 for y
  →  y = 2 - x
```

The algebra is not new: this is the same linear isolation the `accel` query already uses to turn the geodesic equation into $x'' = f(x, x')$. The target is collected from both sides, so `2*y = y + 5` gives `y = 5`, and a coefficient is divided out.

A tensor unknown, in components

This is the case that motivated the feature. Declare the geometry, name the Einstein tensor, declare the unknown, and state the field equations:

```
tensor G := einstein g
head T : TM*, TM* symmetry (1 2)+
equation efe : G_{\mu\nu} = 8 T_{\mu\nu}
solve efe for T_{\mu\nu}
```

Over Schwarzschild:

```
T:
16 independent components identically zero
```

Schwarzschild is a vacuum, and the program has just said so by solving rather than by being told. Over Reissner–Nordström the same four lines give the electromagnetic stress-energy tensor:

```
T:
[t,t] = (-1/4*Mass*Q^2*r + 1/8*Q^2*r^2 + 1/8*Q^4)/r^6
[r,r] = 1/8*Q^2/(2*Mass*r^3 - Q^2*r^2 - r^4)
[theta,theta] = 1/8*Q^2/r^2
[phi,phi] = 1/8*Q^2*sin(theta)^2/r^2
12 independent components identically zero
```

How it works is worth one paragraph, because it explains the refusals. The unknown is given *symbolic* components — one per orbit of its declared symmetry, so a symmetric rank-2 unknown in four dimensions has ten, not sixteen. The equation is then evaluated with those symbols in place, by the ordinary evaluator, and each cell of the resulting grid becomes one scalar isolation. Every piece of index work — inserting the metric, contracting, judging variance — stays where it already was and was already tested.

Asking for a different variance

The target is written with indices, and the variance written there says in what form the answer is wanted. If the equation's free indices disagree, the whole equation is carried to the target's variance first, by contracting with the metric one index at a time:

```
head T : TM, TM symmetry (1 2)+
equation e : A_{\mu\nu} = T_{\mu\nu}
solve e for T^{\mu\nu}
```

Without that step the unknown sits behind two metric factors — $g_{\mu a}\, g_{\nu b} T^{ab}$ — and the cell (μ,ν) becomes a sum over *all* components of T: a 16×16 symbolic system in four dimensions. Moving the equation costs n^{r+1} per index instead, and is what one does by hand.

> **A diagonal metric hides the need for this.** When `g` is diagonal the two metric factors survive only at a=μ, b=ν, so each cell has one unknown even without moving anything. Schwarzschild and Reissner–Nordström are diagonal; a metric with $g_{uv} \neq 0$ is not, and there the move is the difference between an answer and a refusal.

More equations than unknowns

A tensor equation of rank r is n^r equations. When the unknown has fewer degrees of freedom than that, the surplus is not waste — it is a check:

```
tensor Rc := ricci g
head L : on M
equation esp : Rc_{\mu\nu} = L g_{\mu\nu}
solve esp for L
```

Metric	Answer
de Sitter	`L = 3*H^2`
Schwarzschild	`L = 0`
Reissner–Nordström	refused: it asks for `L = -Q^2/r^4` *and* `L = Q^2/r^4`

Sixteen equations, one unknown. Either every cell agrees — and the program has just *proved* the manifold is an Einstein space, with Λ computed — or they do not, and the ansatz does not close. Both answers are useful, and the second is the one no other tool gives for free. The refusal shows the two conflicting values, which here are exactly the sign structure of the electromagnetic Ricci tensor.

The same machinery covers symmetry. A target declared symmetric, against an equation that is not, has cells (μ,ν) and (ν,μ) landing in the same orbit; if they ask for different values, the antisymmetric part would have to vanish and does not, and the refusal names the component where they disagree.

Without components: moving terms

When no head in the equation has components there are no numbers to give, and what remains is what one does on paper — move terms across:

```
head T : TM*, TM*
head U : TM*, TM*
head S : TM*, TM*
equation e : T[a,b] + U[a,b] = S[a,b]
solve e for T[a,b]
  →  T[a,b] = -1 U[a,b] + S[a,b]
```

Which of the two paths runs is decided by the *document*, not by trying one and falling back: if any head in the equation has components, the answer is numbers; if none does,

the answer is rearranged symbols. It is the same expression with different amounts of information — the relationship `simplify` and `eval` already have.

Like terms are collected before occurrences are counted, which is not decoration: without it `3 T[a,b] + -1 T[a,b]` would look like two occurrences of the target and be refused, when it is `2 T[a,b]`.

What it refuses, and why

Situation	What happens
Unknown equation name	Refused, listing the ones the document declares.
The target does not appear	Refused: the equation does not determine it.
Not linear in the target (`y^2 = 4`)	Refused. Two solutions exist, and returning one would hide the other.
A head of rank ≥ 1 written without indices	Refused, asking for them: the variance written is what says in what form the answer is wanted.
The target already has components	Refused: then it is not an unknown.
A cell involving several components of the target	Refused as a linear system, and the message says in which variance the target was declared — that is the fix.
A cell with no unknown but a non-zero value	Refused: there the equation asserts `1 = 0`.
Cells that disagree about the same component	Refused, showing both values.
Target rank neither equal to the equation's nor zero	Refused: neither an isolation nor an over-determination.
Moving an index with no metric declared	Refused rather than inventing one.
The target in several permuted monomials	Refused. See below.

That last one deserves its reason stated. In

```
equation e : T[a,b] + T[b,a] = S[a,b]
```

isolating *T* stops being isolation and becomes a linear system over the algebra of the symmetry group, entangled with Butler–Portugal canonicalisation — and the system is genuinely *under*determined: the antisymmetric part of *T* is not fixed by that equation. Half an answer there would be one of infinitely many, offered without saying that there are infinitely many.

9.23 Every reserved word

The complete list, taken from the program rather than from memory. A name in any of these tables cannot be used as an ordinary variable or as an indeterminate function.

Declarations (10)

Word	Form	Section
`manifold`	`manifold NAME dim N`	9.1
`bundle`	`bundle NAME on MANIFOLD dim N [tangent]`	9.1
`chart`	`chart NAME on MANIFOLD coords (c, ...)`	9.1
`metric`	`metric NAME on CHART bundle BUNDLE { [i,j] = EXPR, ... }`	9.2
`connection`	`connection NAME on CHART { [i,j,k] = EXPR, ... }`	9.2
`tensor`	three forms: components, `:= EXPR`, `:= QUERY`	9.2
`head`	`head NAME : SLOT, ... [symmetry GEN ...]`	9.3
`axiom`	`axiom KIND HEAD`	9.18
`transition`	`transition FROM to TO { c = EXPR, ... }`	9.20
`equation`	`equation NAME : LEFT = RIGHT`	9.21

An eleventh declaration form has no keyword: an alias, `NAME := EXPR` (9.9).

Queries that name a target (12)

All take the shape `NAME [TARGET] [[up|down, ...]]`, where the target is a declared metric or connection and may be omitted when the document declares only one.

Word	Result	Rank
`christoffel`	Connection coefficients Γ^{a}_{bc}	3, not a tensor
`riemann`	Riemann tensor	4
`ricci`	Ricci tensor	2
`einstein`	$G_{ab} = R_{ab} - \tfrac{1}{2} g_{ab} R$	2
`weyl`	Weyl tensor (undefined in 2 dimensions)	4
`scalar`	Ricci scalar	0
`kretschmann`	$R_{abcd} R^{abcd}$	0
`riccisquare`	$R_{ab} R^{ab}$	0
`gaussbonnet`	The Euler density	0

`weylsquare`	$C_{abcd}C^{abcd}$	0
`geodesic`	The geodesic equations; needs `PARAM`	—
`accel`	The same, solved for the acceleration	—

Queries that take an expression (5)

Word	Form	Section
`simplify`	`simplify SUM_OF_MONOMIALS`	9.18
`eval`	`eval SUM_OF_MONOMIALS`	9.19
`invariant`	`invariant SUM_OF_MONOMIALS`	9.20
`solve`	`solve NAME for TARGET`	9.22
`export`	`export {mathematica|sympy} QUERY`	9.17

Structural words

These appear only inside a declaration or query, never on their own.

Word	Where
`on`	`bundle`, `chart`, `metric`, `connection`, `tensor`, `head` (rank 0)
`dim`	`manifold`, `bundle`
`coords`	`chart`
`bundle`	also inside `metric`, naming the bundle the metric lives on
`tangent`	`bundle` — marks it as the tangent bundle of its manifold (9.1)
`symmetry`	`head` and `tensor`, before the generators
`symmetric`, `antisymmetric`	shorthand for a full symmetry group, in place of generators
`to`	`transition FROM to TO`
`for`	`solve NAME for TARGET`
`up`, `down`	the index-variance marker (9.16)
`mathematica`, `sympy`	the two targets of `export`
`bianchi`, `bianchi2`, `metric`, `compatible`	the four kinds of `axiom`

Function names

Thirteen are callable, and are differentiated correctly:

```
sin  cos  exp  sinh  cosh
tan  cot  sec  csc  tanh  coth  sech  csch
```

The first five are atoms of the engine. The eight that follow are rewritten into the quotients they *are* — `tan(x)` enters as `sin(x) * cos(x)^-1` — so nothing downstream needs to know the word was written, and the derivative is the real one rather than a plausible wrong one.

Seventeen more are reserved and **refused** with "not yet implemented as callable":

```
asin  arcsin  acos  arccos  atan  arctan
asinh arcsinh acosh arccosh atanh arctanh
log  ln  sqrt  atan2  abs
```

This refusal is deliberate, and section 9.10 explains why: a name with no standard meaning (`f(r)`, `a(t)`) is accepted as an *indeterminate function* and differentiated by the chain rule, but a name that *does* have a standard meaning must not be absorbed into that same mechanism. Before this rule, `tan(theta)` was accepted as an opaque symbol and differentiated as if `tan′` were an unknown function, never the real $\sec^2$, producing a plausible and wrong Christoffel symbol.

Deliberately absent, and not by oversight: `floor`, `ceil` and `sign`, which have no natural derivative and are out of scope for a program built around differentiable curvature.

10. Worked examples

The ten examples below were executed on the command line (chapter 11) — the same engine the notebook uses — and the output pasted straight from the terminal, not rewritten from memory. The first four files are this repository's own fixtures (`oderom-cli/tests/fixtures/`); the fifth is written for this section; the last four reuse earlier metrics. A holonomy example is deliberately absent: as section 9.5 records, there is no holonomy query in the language today, so there would be no real `.od` to show.

Example 1 — Schwarzschild

```
manifold M dim 4
bundle TM on M dim 4
chart schw on M coords (t, r, theta, phi)
metric g on schw bundle TM {
  [t,t] = -(1 - 2*M/r),
  [r,r] = 1/(1 - 2*M/r),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
```

`ricci`:

```
10 independent components identically zero
```

`kretschmann`:

```
48*M^2/r^6
```

This shows the basic shape of a complete declaration and the queries in their simplest form, with a single parameter. The vanishing Ricci (a vacuum solution) beside a non-vanishing Kretschmann is the point of the example: the manifold is curved even where the Ricci tensor disappears entirely.

Example 2 — Reissner–Nordström

```
metric g on schw bundle TM {
  [t,t] = -(1 - 2*M/r + Q^2/r^2),
  [r,r] = 1/(1 - 2*M/r + Q^2/r^2),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}
```

`kretschmann` (Unicode, the default):

```
(-96*M*Q^2*r + 48*M^2*r^2 + 56*Q^4)/r^8
```

the same query with `--target latex`:

```
\frac{(-96 M Q^{2} r + 48 M^{2} r^{2} + 56 Q^{4})}{r^{8}}
```

A two-parameter metric, and the LaTeX target producing real `\frac{}{}` and exponents from the same tree. The result matches the textbook closed form, $48M^2/r^6 - 96MQ^2/r^7 + 56Q^4/r^8$, and this repository's permanent acceptance test.

Example 3 — the round 2-sphere S² (dimension 2, non-zero curvature)

```
manifold S2 dim 2
bundle TS2 on S2 dim 2
chart standard on S2 coords (theta, phi)
metric g on standard bundle TS2 {
  [theta,theta] = 1,
  [phi,phi] = sin(theta)^2
}
```

`ricci`, `scalar`, `kretschmann`:

```
Ricci[theta,theta] = 1
Ricci[phi,phi] = sin(theta)^2
1 independent component identically zero

2

4
```

And `weyl` on the same metric, refusing for the stated reason rather than returning something meaningless:

```
error: the Weyl tensor is not defined in 2 dimensions
  (the standard formula's 1/(n-2) term is undefined at n=2)
```

Example 4 — the hyperbolic plane H² (`sinh` / `cosh`)

```
manifold H2 dim 2
bundle TH2 on H2 dim 2
chart polar on H2 coords (chi, phi)
metric g on polar bundle TH2 {
  [chi,chi] = 1,
  [phi,phi] = sinh(chi)^2
}
```

`scalar`:

```
-2
```

The direct hyperbolic analogue of the sphere above, and an exact sign flip of its +2. Reaching a constant at all depends on the identity $\cosh^2 - \sinh^2 = 1$ firing during simplification (section 9.4); without it the scalar would not close.

Example 5 — a connection with no metric

```
manifold M dim 2
bundle TM on M dim 2
chart flat2 on M coords (x, y)
connection Gamma on flat2 {
  [x,y,y] = x
}
```

`christoffel` and `riemann` work; the index of Riemann stays mixed, because there is no metric to lower it with:

```
Gamma[^x,y,y] = x
7 independent components identically zero

R[^x,y,x,y] = 1
```

```
R[^x,y,y,x] = -1
14 independent components identically zero
```

`scalar` refuses, naming why:

```
error: `scalar` needs a metric to invert (only a connection was declared)
```

Example 6 — the geodesic equations of Schwarzschild

`geodesic tau` — one equation per coordinate, in canonical equals-zero form:

```
(2*M*r*t''(tau) - 2*M*r'(tau)*t'(tau) - r^2*t''(tau))/(2*M*r - r^2) = 0

(M*r^2*r'(tau)^2 - M*r^2*t'(tau)^2 + 2*M*r^3*r''(tau)
  - 4*M*r^4*sin(theta)^2*phi'(tau)^2 - 4*M*r^4*theta'(tau)^2
  + 4*M^2*r*t'(tau)^2 + 4*M^2*r^3*sin(theta)^2*phi'(tau)^2
  + 4*M^2*r^3*theta'(tau)^2 - 4*M^3*t'(tau)^2 - r^4*r''(tau)
  + r^5*sin(theta)^2*phi'(tau)^2 + r^5*theta'(tau)^2)/(2*M*r^3 - r^4) = 0

(-r*sin(theta)*cos(theta)*phi'(tau)^2 + r*theta''(tau)
  + 2*r'(tau)*theta'(tau))/r = 0

(r*sin(theta)*phi''(tau) + 2*r*cos(theta)*phi'(tau)*theta'(tau)
  + 2*sin(theta)*phi'(tau)*r'(tau))/(r*sin(theta)) = 0
```

Example 7 — the same geodesic solved for the acceleration

`accel tau` gives the radial equation of section 9.12, isolated for `r''(tau)` — the form a numerical integrator consumes. Note that this is still symbolic: nothing is integrated.

Example 8 — the Einstein tensor: vacuum against electrovacuum

`einstein` on Schwarzschild:

```
10 independent components identically zero
```

`einstein` on Reissner–Nordström:

```
G[t,t] = (-2*M*Q^2*r + Q^2*r^2 + Q^4)/r^6
G[r,r] = Q^2/(2*M*r^3 - Q^2*r^2 - r^4)
G[theta,theta] = Q^2/r^2
G[phi,phi] = Q^2*sin(theta)^2/r^2
6 independent components identically zero
```

The contrast is the example: vacuum gives zero everywhere; electrovacuum does not, and every surviving component carries Q. Setting $Q = 0$ collapses the second into the first.

Example 9 — Kerr, from the gallery

Kerr is the newest gallery entry, and the first non-diagonal metric in it — `[t,phi]` is the frame-dragging term that distinguishes Kerr from a spinning Schwarzschild. Load it with the Gallery button, or on the command line:

```
$ oderom load kerr
manifold M dim 4
bundle TM on M dim 4

Sigma := r^2 + a^2*cos(theta)^2

Delta := r^2 - 2*M*r + a^2

chart bl on M coords (t, r, theta, phi)

metric g on bl bundle TM {
  [t,t] = -(1 - 2*M*r/Sigma),
  [t,phi] = -2*M*a*r*sin(theta)^2/Sigma,
  [r,r] = Sigma/Delta,
  [theta,theta] = Sigma,
  [phi,phi] = (r^2 + a^2 + 2*M*a^2*r*sin(theta)^2/Sigma) * sin(theta)^2
}
```

`ricci` — the vacuum result, in about a second:

```
$ time oderom ricci kerr.od
10 independent components identically zero
real    0m1.14s
```

The entry also demonstrates aliases (section 9.9) carrying real weight: Σ and Δ appear in four of the five components, and writing them out each time would make the metric unreadable.

Example 10 — the matter that produces the geometry

The first nine examples ask the program to compute. This one asks it to *invert*: given a geometry, what stress-energy tensor does Einstein's equation require?

```
manifold M dim 4
bundle TM on M dim 4
chart schw on M coords (t, r, theta, phi)
metric g on schw bundle TM {
  [t,t] = -(1 - 2*Mass/r + Q^2/r^2),
  [r,r] = 1/(1 - 2*Mass/r + Q^2/r^2),
  [theta,theta] = r^2,
  [phi,phi] = r^2 * sin(theta)^2
}

tensor G := einstein g
head T : TM*, TM* symmetry (1 2)+
equation efe : G_{\mu\nu} = 8 T_{\mu\nu}
```

`solve efe for T_{\mu\nu}`:

```
T:
[t,t] = (-1/4*Mass*Q^2*r + 1/8*Q^2*r^2 + 1/8*Q^4)/r^6
[r,r] = 1/8*Q^2/(2*Mass*r^3 - Q^2*r^2 - r^4)
[theta,theta] = 1/8*Q^2/r^2
[phi,phi] = 1/8*Q^2*sin(theta)^2/r^2
12 independent components identically zero
```

That is the electromagnetic stress-energy tensor of a charged black hole, obtained without it being written down anywhere. Delete `Q^2/r^2` from the two metric components and the same four lines answer `16 independent components identically zero`: Schwarzschild is a vacuum, and the program says so by solving rather than by being told.

The *same* geometry answers a second, different question. Ask whether it is an Einstein space — whether the Ricci tensor is proportional to the metric:

```
tensor Rc := ricci g
head L : on M
equation esp : Rc_{\mu\nu} = L g_{\mu\nu}
```

`solve esp for L` over Reissner–Nordström:

```
error: parse error: a equacao `esp` nao tem solucao: ela pede
`L = -Q^2/r^4` e tambem `L = Q^2/r^4` -- o ansatz nao fecha
```

Sixteen equations, one unknown, and they disagree. The two values shown are the sign structure of the electromagnetic Ricci tensor, and the refusal is the informative answer: charged black holes are not Einstein spaces. Over de Sitter the same three lines return `L = 3*H^2`, and over Schwarzschild `L = 0`.

11. Command-line use

Two binaries, separate from each other and from the notebook: `oderom` (one command per process) and `oderom-repl` (a long-lived interactive session). Both use the grammar of chapter 9 and the same engine as the notebook — no language construct changes.

11.1 `oderom` — one command per run

The real usage message (`oderom-cli/src/error.rs`), reflowed here for the page:

```
usage: oderom canon [--prelude PATH] "<expression>"
   or: oderom simplify [--prelude PATH] [--metric HEAD]... [--bianchi HEAD]...
       "<sum of monomials>"
   or: oderom eval [--prelude PATH] "<sum of monomials>"
   or: oderom invariant [--prelude PATH] "<sum of monomials>"
   or: oderom solve [--prelude PATH] NAME TARGET
   or: oderom {christoffel|riemann|ricci|scalar|kretschmann|einstein|
       riccisquare|gaussbonnet|weyl|weylsquare} FILE
```

```
       [--metric NAME | --connection NAME]
       [--target unicode|latex|json|mathematica|sympy]
       [--max-lines N] [--max-nodes N] [--max-denominator-degree N]
       [--timeout SECONDS]
   or: oderom {geodesic|accel} FILE --param NAME
       [--metric NAME | --connection NAME]
       [--target unicode|latex|json|mathematica|sympy]
       [--max-lines N] [--max-nodes N] [--max-denominator-degree N]
       [--timeout SECONDS]
   or: oderom export {mathematica|sympy} {christoffel|riemann|ricci|
       scalar|kretschmann|geodesic|accel|einstein|riccisquare|
       gaussbonnet|weyl|weylsquare} FILE
       [--metric NAME | --connection NAME] [--param NAME]
       [--max-nodes N] [--max-denominator-degree N] [--timeout SECONDS]
   or: oderom load NAME
```

Typical use, with one of the queries of section 9.5 and an `.od` file:

```
oderom kretschmann my_metric.od
oderom riemann my_metric.od --metric g2 --target latex
oderom einstein my_metric.od
oderom weyl my_metric.od
```

`geodesic` and `accel` are the only two of the twelve with a flag of their own — `--param`, mandatory, the affine parameter's name (the notebook grammar's trailing identifier, here as a flag):

```
oderom geodesic schwarzschild_ascii.od --param tau
oderom accel schwarzschild_ascii.od --param tau --target latex
```

Limits and progress

Flag	Default	What it guards
`--timeout`	30 s	Wall time for the whole run.
`--max-nodes`	50000	Expression-tree size, checked after every stage.
`--max-denominator-degree`	30	Denominator degree, likewise.
`--max-lines`	—	Truncates the printed component list; the summary still reports the full count.

Every stage of the computation is announced on stderr as it begins, so if the timeout fires, the last announced line is exactly the stage in progress. One of those lines names which normalisation engine was chosen:

```
$ oderom ricci kerr.od
inverting the metric...
computing Christoffel symbols...
```

```
motor: localizado
computing the Riemann tensor (mixed)...
contracting to the Ricci tensor...
10 independent components identically zero
```

`motor: localizado` means the localised rational engine was used — the one that made metrics like Kerr practical (chapter 13). It is progress information, not a setting; there is no flag to choose between engines here. (`--engine` belongs to `simplify`, section 9.18, and selects something else entirely.)

The other three modes

`oderom canon [--prelude PATH] "expression"` canonicalises a single abstract monomial under its declared symmetries — the `head` / `symmetry` level of section 9.3, operating on `prelude.od` by default.

`oderom simplify` is section 9.18: a sum of abstract monomials, with axioms declared by flag.

`oderom load NAME` takes no `FILE`: it prints a known gallery metric's declarations to stdout as a complete, valid `.od` file. It composes with any other query by redirection:

```
oderom load antidesitter > antidesitter.od
oderom scalar antidesitter.od
-12*H^2
```

An unknown name fails and lists every known one in the error itself:

```
error: unknown gallery entry `naoexiste`
  (known: desitter, antidesitter, frw, schwarzschild,
   reissnernordstrom, kerr)
```

11.2 `oderom-repl` — an interactive session

The real commands, exactly as the REPL lists them: `:load` `:reload` `:defs` `:entries` `:recompute` `:save` `:timeout` `:quit`. A line without a leading `:` is a query (section 9.5), executed immediately against the loaded file.

```
$ oderom-repl
ODEROM REPL. :load <file.od> to start, :quit or Ctrl+D to leave.
oderom> :load schwarzschild_ascii.od
ok: 2 definition(s), 9ms
oderom> christoffel
inverting the metric...
computing Christoffel symbols...
[1] christoffel -- [OK] (147ms)
Gamma[^t,t,r] = -M/(2*M*r - r^2)
...
oderom> :entries
[1] christoffel -- [OK] (147ms)
```

`:reload` re-reads the same file from disk (useful after editing it in another editor) and marks as stale the entries that depended on what changed — the same staleness principle as chapter 5. `:recompute <n>` or `:recompute stale` recomputes one entry or every stale one.

The REPL has no gallery command of its own: `:load` reads a file from disk, not a gallery name. To use a gallery metric here, go through `oderom load NAME > file.od` first.

11.3 Cancellation and timeout: the two binaries differ

This is a real difference, verified by running both against the same metric that never finishes on its own (non-reciprocal g_{tt}/g_{rr}).

`oderom` ends the whole process when the timeout fires, naming the stage in progress:

```
$ oderom kretschmann non_reciprocal.od --timeout 3
inverting the metric...
computing Christoffel symbols...
computing the Riemann tensor (mixed)...
error: timed out after 3s -- last stage in progress:
  `computing the Riemann tensor (mixed)`
```

There is no continuing after that — there is no "after", since each invocation is one computation. The process ends.

`oderom-repl` cancels only the running query; the process stays up, ready for the next one, exactly as the notebook never loses its window over a cancelled block:

```
oderom> :timeout 3
timeout set to 3s
oderom> kretschmann
inverting the metric...
computing Christoffel symbols...
timed out after 3s -- cancelling...
[1] kretschmann -- [CANCELLED]
oderom> scalar
...
[2] scalar -- [CANCELLED]
```

This is the notebook's own deep cancellation (`oderom_expr::run_cancellable`, which checks inside the normalisation loop, not merely between components) — the REPL and the notebook go through the same `oderom-session::run_query`. The standalone `oderom` binary does not, which is why it has only a wall clock and no deep cancellation.

12. Quick reference

Run keys

Key	Action

`Shift` + `Enter`	Run; focus moves to the next block (creating one if this is the last).
`Ctrl` + `Enter`	Run; focus stays.
`Alt` + `Enter`	Run; insert a new block below and focus it.

Block states

Indicator	State
`[ ]`	Never run
`[n]`	Run, current (n = order of execution)
`[n]` amber + ▌ + ↻	Run, stale
—	Running (cancellable)
—	Cancelled

Language constructs

Construct	Syntax
`manifold`	`manifold NAME dim N`
`bundle`	`bundle NAME on MANIFOLD dim N [tangent]`
`head` (rank ≥ 1)	`head NAME : BUNDLE[*], ... [symmetry antisymmetric|symmetric|CYCLES±]`
`head` (rank 0)	`head NAME : on MANIFOLD` — no slots, no `symmetry`
`chart`	`chart NAME on MANIFOLD coords (c1, c2, ...)`
`metric`	`metric NAME on CHART bundle BUNDLE { [i,j] = EXPR, ... }`
`connection`	`connection NAME on CHART { [i,j,k] = EXPR, ... }`
`tensor`, components (9.2)	`tensor NAME : SLOTS [symmetry ...] on CHART { [i,j] = EXPR, ... }` — components for any tensor, not just the metric
`tensor`, by expression (9.2)	`tensor NAME^{\mu\nu} := EXPR` — the bundle, dimension and chart come from the factors used
`tensor`, from a query (9.2)	`tensor NAME := {ricci|einstein|riemann|weyl} [METRIC]` — curvature with a name, so it can be contracted, compared and solved for
`eval` (9.19)	`eval SUM_OF_MONOMIALS` — the same expression as `simplify`, evaluated on declared components
`axiom` (9.18)	`axiom {bianchi|bianchi2|metric|compatible} HEAD` — a declared multi-term identity, document-scoped

`transition` (9.20)	`transition FROM to TO { c = EXPR, ... }` — a coordinate change, one direction per declaration
`invariant` (9.20)	`invariant SUM_OF_MONOMIALS` — the same expression again, evaluated in every chart that carries it, and compared
`equation` (9.21)	`equation NAME : LEFT = RIGHT` — a named assertion; coefficients by juxtaposition, a symbolic one being a rank-0 head
`solve` (9.22)	`solve NAME for TARGET` — the inverted question. Target: a scalar name, or a head with indices (`T^{\mu\nu}`, `T[a,b]`)
alias (9.9)	`NAME := EXPR` — used as `NAME`, never `NAME(...)`
indeterminate function (9.10)	`f(r)`, `h(t, r)`; derivatives `f'(r)`, `h_{t,r}(t, r)`
query (9.5)	`NAME [TARGET] [[up|down, ...]]`
`geodesic` / `accel`	`geodesic [TARGET] PARAMETER`
`export` (9.17)	`export {mathematica|sympy} QUERY`
`simplify` (9.18)	`simplify SUM_OF_MONOMIALS` — juxtaposition, not `*`; `;` for covariant derivatives
gallery (13)	not a grammar construct — the Gallery button, or `oderom load NAME`. Names: `desitter antidesitter frw schwarzschild reissnernordstrom kerr`

13. The spacetime gallery

The gallery is a list of known metrics that can be loaded without typing them. The Gallery button (chapter 3) opens a panel with each entry, its name, a line describing the spacetime and the curvature invariant that characterises it. Clicking one loads it.

> **Loading pastes text; it runs nothing.** The chosen metric's declarations are inserted as new, editable blocks at the end of the notebook — exactly the text a hand-typed example would have, never an opaque metric behind a button. Nothing runs automatically; the blocks appear as `[ ]`, ready for `Shift`+`Enter`. Existing blocks are never touched.

Every metric carries symbolic parameters (M, H, a, Q). Fixing a value needs no new mechanism: edit the pasted text, or declare an alias in a block of its own (section 9.9).

Each gallery entry is also an acceptance test of the engine: its curvature was computed and checked against a known closed form before it entered the list. A gallery that loaded metrics the engine got wrong would be worse than no gallery — which is why the list below is deliberately small.

Name	What it is	Verified invariant

`desitter`	De Sitter, flat slicing (constant positive curvature)	$R = 12H^2$ (constant); Weyl identically zero
`antidesitter`	Anti-de Sitter, Poincaré coordinates (constant negative curvature)	$R = -12H^2$ (constant); $R_{ab} = -3H^2 g_{ab}$
`frw`	Spatially flat Friedmann–Robertson–Walker, generic scale factor a(t)	$R = 6(a''(t)/a(t) + a'(t)^2/a(t)^2)$
`schwarzschild`	Vacuum, static, spherically symmetric	$R_{ab} = 0$; Kretschmann $= 48M^2/r^6$
`reissnernordstrom`	Spherically symmetric, charged (electrovacuum, not pure vacuum)	$R = 0$ but $R_{ab} \neq 0$
`kerr`	Rotating black hole, Boyer–Lindquist coordinates (vacuum, non-diagonal)	$R_{ab} = 0$ in all 10 independent components

Two entries deserve a note of their own.

`frw` uses an indeterminate function (section 9.10) directly inside a metric component, not merely inside a geodesic equation. The engine differentiates Γ, R_{ab} and R correctly in terms of a(t), a′(t) and a″(t) without the user saying what a(t) is.

`kerr` is the first non-diagonal entry, and it is recent. Earlier revisions of this manual recorded that Kerr could not be in the gallery: its Christoffel and Riemann tensors did not finish in practical time against the metric's genuinely bivariate denominator. That is no longer true, and the fix was not in the gallery but two levels below it — a quadratic accumulator in polynomial multiplication, and the notebook session finally taking the localised rational engine that had already been chosen for it. `ricci` on Kerr now returns the vacuum result in about a second.

> **Diagonal metrics are no longer a restriction.** Metric inversion detects the structure of *g* at three levels — diagonal (the unchanged fast path), block-diagonal (each decoupled block inverted on its own, such as Kerr's 2×2 $g_{t\varphi}$ block) and general (adjugate/cofactors) — and verifies $g^{ab}g_{bc} = \delta^a{}_c$ before accepting the result.

Gödel remains outside the gallery, and for the reason Kerr no longer has: its Ricci scalar computes to an algebraically correct value that the normaliser does not reduce to closed form. No entry joins without a genuinely verified curvature test, so it stays out, documented, with no date attached.

On the command line, `oderom load NAME` reaches the same catalogue — printing the declarations to stdout instead of pasting blocks. The difference makes sense, since the CLI has no block list; the catalogue itself is one and the same (`oderom-cli/src/gallery.rs`), shared by both surfaces:

```
oderom load schwarzschild > schwarzschild.od
oderom kretschmann schwarzschild.od
```

How to cite

If ODEROM is useful in your work, please cite the archived record rather than the web address, so that the version you used stays identifiable:

```
de Lima, R. C. R. (2026). ODEROM: a symbolic engine for
differential geometry [Computer software].
Zenodo. https://doi.org/10.5281/zenodo.22262763
```

That identifier always resolves to the most recent release. Where the version matters, cite that version's own identifier instead — 0.1.0 is `10.5281/zenodo.22262764`.

A machine-readable `CITATION.cff` accompanies the source, and a technical description of the program — what it computes, how it represents expressions, and how the results are verified — is published alongside this manual as `ODEROM-paper.pdf`.

ODEROM · user manual. Every chapter re-verified against the source on 20 August 2026, and every example re-executed against the binary at that revision. The sections added since — `eval`, the LaTeX index notation, and 9.20 (`transition` / `invariant`) — carry outputs pasted from the terminal on the day each was written, most recently 2 September 2026.