

Demolab — contents

From a capacitor to the LIF neuron	2
The Streamlit spiking-neuron playground	4
The Hodgkin–Huxley membrane, in equations	6
Runbooks	9
Getting help	10
The vocabulary	11
The folder structure	12
Authoring slides	13
Writing style	14
The contract	15
Getting started	16
Introduction	18
LIF voltage traces and a recurrent network raster	21
EIF voltage traces and a recurrent network raster	25
A passive cartpole in MuJoCo	29
A chaotic double pendulum in MuJoCo	30

From a capacitor to the LIF neuron

30 May 2026

A neuron's membrane is a thin lipid bilayer separating two salty solutions. Charge piles up on either side, ions leak through embedded channels, and currents are injected by synapses or an experimenter's electrode. Electrically, this is just an RC circuit. The leaky integrate-and-fire (LIF) neuron is what you get when you write Kirchhoff's current law for that circuit and bolt a threshold on top.

The membrane is a capacitor

Two conductors separated by an insulator form a capacitor. The bilayer is the insulator; the intracellular and extracellular fluids are the conductors. The defining relation for a capacitor is

$$Q = C_m V,$$

where $V = V_{\text{in}} - V_{\text{out}}$ is the membrane potential, Q is the charge stored on the inner face, and C_m is the membrane capacitance (typically $\approx 1\mu\text{F}/\text{cm}^2$).

Differentiating in time gives the capacitive current, the current required to change the voltage at rate $\frac{dV}{dt}$:

$$I_C = \frac{dQ}{dt} = C_m \frac{dV}{dt}.$$

The membrane is also a leaky resistor

Ion channels punctuate the bilayer. Even at rest some are open, so the membrane has a finite conductance g_L (equivalently, a leak resistance $R_m = 1/g_L$). The leak current is driven by how far the voltage sits from its reversal potential V_{rest} :

$$I_L = g_L(V - V_{\text{rest}}) = \frac{V - V_{\text{rest}}}{R_m}.$$

When $V > V_{\text{rest}}$, the leak current is positive: charge flows outward and pulls V back down. The capacitor and the leak resistor sit in parallel between the inside and the outside of the cell.

Kirchhoff's current law

Now inject an external current I . Charge is conserved at the membrane node: whatever flows in must either charge the capacitor or escape through the leak.

$$I = I_C + I_L = C_m \frac{dV}{dt} + \frac{V - V_{\text{rest}}}{R_m}.$$

Multiply through by R_m and define the membrane time constant $\tau_m = R_m C_m$:

$$\tau_m \frac{dV}{dt} = -(V - V_{\text{rest}}) + R_m I.$$

This is the subthreshold LIF equation. Nothing has been assumed beyond Kirchhoff's law and linear, passive channels. The dynamics are first-order: any voltage perturbation decays exponentially toward $V_{\text{rest}} + R_m I$ with time constant τ_m .

Adding a spike

Real neurons don't just relax. When V crosses a threshold, voltage-gated sodium channels open, the membrane briefly depolarizes to nearly +40 mV, and potassium channels repolarize it. The

full mechanism is the Hodgkin–Huxley model. LIF throws all of that away and replaces it with a rule:

$$V(t) \geq V_{\text{th}} \implies \text{spike at } t, \quad V \leftarrow V_{\text{reset}}.$$

Two parameters, no extra differential equations. Often a refractory period is added during which V is clamped at V_{reset} .

What was thrown away

The derivation makes the model’s assumptions explicit:

- **Linear leak.** Real channels are voltage- and time-dependent; g_L is treated as constant.
- **Point neuron.** The whole membrane is collapsed to one node: no dendrites, no axonal delay, no spatial structure.
- **Phenomenological spike.** The action potential is replaced by a reset, so the model says nothing about spike shape, sodium dynamics, or bursting.

In exchange you get a single linear ODE with a reset, which integrates in a handful of lines and scales to networks of thousands of neurons. That is the LIF neuron simulated in exp000: same equation, same RC circuit, now with current injected and a threshold to cross.

The Streamlit spiking-neuron playground

16 June 2026

► **Open the live playground:** localhost:8501. Start it with `uv run streamlit run experiments/playground.py` (details below).

Most of this lab runs *batch-style*: a tool command takes a fixed set of arguments, runs a simulation, and drops a directory of artifacts that an experiment then publishes. That is the right shape for a result you want to pin down and cite.

The Streamlit app is the opposite end of that spectrum: a live, interactive surface for *building intuition* before you commit to a run. It re-simulates a single integrate-and-fire neuron on every slider drag and redraws the voltage trace immediately.

What it does

A radio button at the top of the sidebar picks the model, and the app drives the matching neuron subcommand: the **same** CLI the experiments run:

- **LIF**, the leaky integrate-and-fire neuron (`neuron lif`), a hard threshold with a spike-and-reset rule. This is the model behind exp000.
- **EIF**, the exponential integrate-and-fire neuron (`neuron eif`), where an explicit exponential term replaces the hard threshold. This is the model behind exp001.

Every parameter is a slider in the sidebar. Most are shared between the two models:

- **Input**, tonic current I and membrane resistance R_m .
- **Membrane**, time constant τ_m , and the resting / reset potentials.
- **Simulation**, duration and the integration timestep Δt .

The spiking mechanism, by contrast, is model-specific: LIF exposes a single hard threshold V_{th} , while EIF exposes the soft threshold V_T , the slope factor Δ_T , and the peak cutoff V_{peak} . As you drag, the app shows the active model, the spike count, and the firing rate in Hz, flagged *spiking* or *silent*.

The LIF membrane equation it integrates is

$$\tau_m \frac{dV}{dt} = -(V - V_{rest}) + R_m I,$$

with a spike-and-reset rule whenever V crosses V_{th} ; EIF adds the $\Delta_T \exp((V - V_T)/\Delta_T)$ term and resets when V reaches V_{peak} .

Starting the server

Launch it through `uv` (never call `python` / `streamlit` directly):

```
uv run streamlit run experiments/playground.py
```

Streamlit serves on `http://localhost:8501` by default and opens a browser tab. To pick another port:

```
uv run streamlit run experiments/playground.py --server.port 3001
```

Stop it with Ctrl-C in the terminal.

How it fits the rest of the lab

The playground deliberately sits **outside** the tool $\rightarrow$ experiment contract: it writes no `config.json` / `output.json` / `manifest.json` of its own to the record, produces no committed

artifacts, and isn't bundled by any experiment runner. It's a thinking tool, not a reproducible run, but it never reimplements the science: each slider settle shells out to the tool CLI and reads the trace back from `temp/neuron/<model>/`, the same files the experiments consume. When a slider configuration produces something worth keeping, reproduce it as a pinned run with the tool instead:

```
uv run python tools/neuron/tool.py eif --current 2.5 --duration 100 --delta-t 2
```

That writes the artifacts an experiment can publish.

The Hodgkin–Huxley membrane, in equations

3 July 2026

Almost all equations, but not quite: a few lines of prose between the blocks so the eye has somewhere to rest. The subject is the four-dimensional system for a space-clamped patch of excitable membrane, the model Hodgkin and Huxley fit to the squid giant axon in 1952 [1], with state $\mathbf{x} = (V, m, h, n)^\top$: one voltage and three gating variables. Everything below is that one system, taken apart one relation at a time.

Current balance

The membrane is a capacitor in parallel with three voltage-gated conductances [1, 10]. Charge conservation across it gives the equation that ties the whole model together: the rate of change of voltage is set by the sum of ionic currents plus whatever is injected:

$$C_m \dot{V} = - \underbrace{\bar{g}_{\text{Na}} m^3 h (V - E_{\text{Na}})}_{I_{\text{Na}}} - \underbrace{\bar{g}_{\text{K}} n^4 (V - E_{\text{K}})}_{I_{\text{K}}} - \underbrace{g_{\text{L}} (V - E_{\text{L}})}_{I_{\text{L}}} + I_{\text{ext}}(t)$$

C_m , membrane capacitance; $\bar{g}_\bullet$, peak conductances; $E_\bullet$, reversal potentials; $m, h, n \in [0, 1]$, gating variables; I_{ext} , injected current.

Each conductance is the peak value scaled by gates raised to a power: the exponents (m^3 , n^4) are the number of independent subunits that must all be open at once. The gates are what make the patch excitable; the next few blocks say how they move.

Gating kinetics

Each gate relaxes toward a voltage-dependent target, written two equivalent ways, as forward/backward rates, or as a target value x_∞ approached with time constant τ_x :

$$\dot{x} = \alpha_x(V)(1 - x) - \beta_x(V)x = \frac{x_\infty(V) - x}{\tau_x}(V), \quad x_\infty = \frac{\alpha_x}{\alpha_x + \beta_x}, \quad \tau_x = \frac{1}{\alpha_x + \beta_x}, \quad x \in \{m, h, n\}.$$

$\alpha_x(V)$, $\beta_x(V)$, voltage-dependent opening/closing rate constants; x_∞ , steady-state (in)activation; τ_x , its relaxation time constant; $x \in \{m, h, n\}$, the three gating variables.

Rate functions (V in mV, rates in ms^{-1})

The rates themselves are the empirical heart of the model: six functions fit to voltage-clamp data, no first-principles derivation behind them [1]. They are what turns the abstract kinetics above into concrete dynamics:

$$\begin{aligned} \alpha_m(V) &= \frac{0.1(V + 40)}{1 - \exp(-(V + 40)/10)}, & \beta_m(V) &= 4 \exp(-(V + 65)/18), \\ \alpha_h(V) &= 0.07 \exp(-(V + 65)/20), & \beta_h(V) &= \frac{1}{1 + \exp(-(V + 35)/10)}, \\ \alpha_n(V) &= \frac{0.01(V + 55)}{1 - \exp(-(V + 55)/10)}, & \beta_n(V) &= 0.125 \exp(-(V + 65)/80). \end{aligned}$$

The α_m and α_n expressions look singular: numerator and denominator both vanish at one voltage, so a naive evaluation returns 0/0. The function is smooth there anyway: the singularity is removable, and the value is just the limit [9]:

$$\alpha_m(V) = \begin{cases} \frac{0.1(V+40)}{1 - \exp(-(V+40)/10)} & \text{if } V \neq -40 \\ 1 & \text{if } V = -40 \end{cases} \quad \text{since} \quad \lim_{V \rightarrow -40} \alpha_m(V) = 1.$$

Steady-state gating as a Boltzmann sigmoid

Plotted against voltage, each x_∞ is an S-curve, and it is often more convenient to fit it directly as a Boltzmann sigmoid with a half-activation voltage $V_{1/2}$ and a slope factor k than to go through the rate functions [7, 8]. Its derivative has the tidy logistic form that makes the slope at $V_{1/2}$ exactly $1/(4k)$:

$$x_\infty(V) = \frac{1}{1 + \exp(-(V - V_{1/2})/k)}, \quad \frac{dx_\infty}{dV} = \frac{x_\infty(1 - x_\infty)}{k}.$$

$V_{1/2}$, half-activation voltage; k , slope factor.

Equilibrium potentials

The reversal potentials $E_\bullet$ that appear in the current balance are not free parameters: they are set by the ionic concentrations either side of the membrane. Nernst gives the single-ion equilibrium; Goldman–Hodgkin–Katz generalises to the current carried by a partially permeant ion and to the resting potential of a membrane permeable to several ions at once [2, 3]:

$$E_S = \frac{RT}{z_S F} \ln \frac{[S]_{\text{out}}}{[S]_{\text{in}}}, \quad I_S = P_S z_S^2 \frac{VF^2}{RT} \cdot \frac{[S]_{\text{in}} - [S]_{\text{out}} e^{-z_S VF/RT}}{1 - e^{-z_S VF/RT}},$$

$$V_m = \frac{RT}{F} \ln \frac{P_K[K]_{\text{out}} + P_{\text{Na}}[\text{Na}]_{\text{out}} + P_{\text{Cl}}[\text{Cl}]_{\text{in}}}{P_K[K]_{\text{in}} + P_{\text{Na}}[\text{Na}]_{\text{in}} + P_{\text{Cl}}[\text{Cl}]_{\text{out}}}.$$

E_S , Nernst potential of ion S ; V_m , resting potential of the mixed-permeability membrane; I_S , GHK current carried by S ; R , gas constant; T , absolute temperature; F , Faraday constant; z_S , valence of S ; $P_\bullet$, membrane permeabilities; $[S]_{\text{in}}$, $[S]_{\text{out}}$, intra- and extracellular concentrations.

Linear stability

With the vector field fully specified, the question of whether the rest state is quiet or poised to spike becomes a matter of eigenvalues [4, 5, 6]. Write the system as $\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x})$ and linearise about a fixed point $\mathbf{x}^*$ where $\mathbf{F}(\mathbf{x}^*) = \mathbf{0}$. The Jacobian $\mathbf{J} = \partial \mathbf{F} / \partial \mathbf{x}$ is

$$\mathbf{J} = \begin{bmatrix} \partial_V \dot{V} & \partial_m \dot{V} & \partial_h \dot{V} & \partial_n \dot{V} \\ \partial_V \dot{m} & -1/\tau_m & 0 & 0 \\ \partial_V \dot{h} & 0 & -1/\tau_h & 0 \\ \partial_V \dot{n} & 0 & 0 & -1/\tau_n \end{bmatrix}, \quad \partial_V \dot{V} = -\frac{g_L + \bar{g}_{\text{Na}} m^3 h + \bar{g}_{\text{K}} n^4}{C_m},$$

and the rest state is stable iff every eigenvalue has negative real part: $\det(\mathbf{J} - \lambda \mathbf{I}) = 0 \implies \text{Re}(\lambda_i) < 0$ for all i .

$\mathbf{F}$, the model's vector field; $\mathbf{x}^*$, a fixed point; $\mathbf{J}$, the Jacobian there; λ_i , its eigenvalues; $\partial_a \dot{b}$, the partial of $\dot{b}$ with respect to a .

Synaptic drive and subthreshold response

So far I_{ext} has been an anonymous forcing term. In a network it is synaptic: presynaptic spikes open conductances that pull the membrane toward a synaptic reversal potential [7]. A spike train $\{t_k\}$ arriving through a synapse of reversal E_{syn} adds

$$I_{\text{syn}}(t) = g_{\text{syn}}(t)(V - E_{\text{syn}}), \quad g_{\text{syn}}(t) = \bar{g} \sum_{k: t_k \leq t} \frac{t - t_k}{\tau_s} e^{1-(t-t_k)/\tau_s},$$

where each presynaptic spike contributes an alpha-function conductance transient of peak $\bar{g}$ (the synaptic weight) that rises and decays on the synaptic time constant τ_s . Below threshold, where

the membrane is a leaky integrator $\tau \dot{V} = -(V - V_{\text{rest}}) + RI$ with membrane time constant τ and input resistance R , the voltage is the convolution

$$V(t) = V_{\text{rest}} + \frac{R}{\tau} \int_{-\infty}^t e^{-(t-s)/\tau} I(s) ds, \quad \hat{Z}(\omega) = \frac{R}{1 + i\omega\tau},$$

with $\hat{Z}(\omega)$ the input impedance, a first-order low-pass whose cutoff sits at $\omega_c = 1/\tau$.

g_{syn} , synaptic conductance; E_{syn} , synaptic reversal potential; $\{t_k\}$, presynaptic spike times; $\bar{g}$, peak conductance (synaptic weight); τ_s , synaptic time constant; τ , R , membrane time constant and input resistance; $\hat{Z}(\omega)$, input impedance; ω_c , low-pass cutoff.

References

1. Hodgkin AL, Huxley AF (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol* 117:500–544. doi:10.1113/jphysiol.1952.sp004764
2. Goldman DE (1943). Potential, impedance, and rectification in membranes. *J Gen Physiol* 27:37–60. doi:10.1085/jgp.27.1.37
3. Hodgkin AL, Katz B (1949). The effect of sodium ions on the electrical activity of the giant axon of the squid. *J Physiol* 108:37–77. doi:10.1113/jphysiol.1949.sp004310
4. FitzHugh R (1961). Impulses and physiological states in theoretical models of nerve membrane. *Biophys J* 1:445–466. doi:10.1016/S0006-3495(61)86902-6
5. Nagumo J, Arimoto S, Yoshizawa S (1962). An active pulse transmission line simulating nerve axon. *Proc IRE* 50:2061–2070. doi:10.1109/JRPROC.1962.288235
6. Ermentrout GB, Terman DH (2010). *Mathematical Foundations of Neuroscience*. Springer. doi:10.1007/978-0-387-87708-2
7. Gerstner W, Kistler WM, Naud R, Paninski L (2014). *Neuronal Dynamics*. Cambridge University Press. doi:10.1017/CBO9781107447615
8. Izhikevich EM (2007). *Dynamical Systems in Neuroscience*. MIT Press. doi:10.7551/mitpress/2526.001.0001
9. Sterratt D, Graham B, Gillies A, Willshaw D (2011). *Principles of Computational Modelling in Neuroscience*. Cambridge University Press. doi:10.1017/CBO9780511975899
10. Cole KS, Curtis HJ (1939). Electric impedance of the squid giant axon during activity. *J Gen Physiol* 22:649–670. doi:10.1085/jgp.22.5.649

Runbooks

7 July 2026

Runbooks are demolab’s **procedure layer**: named, repeatable jobs the agent runs step by step, showing each result and confirming before it moves on. Where a guide is always-on reference, a runbook is invoked when you want it: type its name in capitals (**LINT**, **DOCTOR**, **GETTING-STARTED**) and the agent opens it and drives it. Say **HELP** for the full menu. The canonical text is the Markdown source, linked below.

Getting in

- **GETTING-STARTED**: set up a fresh lab end to end: scaffold, brand, pick a stack, get it building and optionally published, then land a first experiment.
- **TOUR**: a guided walk through an existing repository, so you (or a new collaborator) learn where everything lives.

Checking the work

- **LINT**: audit the prose *and* the figures against the house style.
- **DOCTOR**: audit the repository’s structure against the rules.
- **RED-TEAM**: attack a result’s validity, hard, to surface what would break it.
- **STEELMAN**: build the strongest honest case for a result.
- **GROUND-CLAIMS**: back every claim with a run or a citation.
- **NEXT**: suggest what to run next.

Bringing work in

- **MIGRATE-CODE**: wrap an existing codebase, one experiment at a time, without rewriting the science.
- **MIGRATE-STACK**: switch the language your tools are written in (MATLAB, R, Julia, Octave).
- **FROM-JUPYTER**: convert a notebook into a tool, a runner, and a write-up.
- **FROM-PAPER**: reproduce a paper’s result from scratch.

Housekeeping

- **EMBED-DOCS**: use demolab as a docs subfolder inside another project.
- **UPDATE**: vendor the latest engine into your lab and review what changed between versions.

Their always-on counterpart, the **guides**, each has its own page in this collection.

Getting help

7 July 2026

Stuck, or found a bug? Try self-service first: the runbooks under `demolab-engine/runbooks/` and the other guides cover most operating questions. When you still need a human, here's where to go.

Where to ask

- **GitHub issues:** `github.com/eoinmurray/demolab/issues`. Use these for bugs, feature requests, and questions about the framework. Prefer this: issues are searchable, so your question helps the next person and the fix lands in the open.
- **Email:** `em586@cam.ac.uk`. Use this for private matters, or if you can't or don't want to post publicly.

Writing a report that gets answered

Before opening an issue, run the doctor (say “*doctor the repo*”), check the guides and runbooks, and search existing issues. Then include, in order:

- **What you ran and what happened:** the exact command and the *full* error output, not a paraphrase.
- **Toolchain versions:** `uv --version`, `typst --version`, `task --version`.
- **The commit:** demolab stamps the git SHA into the page/PDF footer, so paste that line or run `git rev-parse --short HEAD`.
- **Framework or content?:** state whether it's the engine (a demolab bug) or your own tool/experiment/writing. Ask if you're unsure.
- **A minimal repro** if you can: the smallest writing or tool that triggers it.

You can also just type `SUPPORT` in capitals and the coding agent will walk you through all of this interactively.

The full reference lives in `SUPPORT.md`.

The vocabulary

7 July 2026

demolab has a small vocabulary for its parts, and using the words precisely saves a lot of confusion. This page defines the ones you will meet most, with the distinctions that are easy to mix up called out explicitly. If you would rather be walked through it interactively, type GLOSSARY in capitals and the coding agent will take you term by term.

The core pieces

- **Tool:** a small program (Python by default) holding *reusable* science: a model or solver behind a small CLI that writes the contract files. It emits data, not plots, and reuse is the bar for making one.
- **Experiment:** a runner plus writeup sharing an id: `experiments/exp<NNN>.py` runs a tool and stages results, and `writings/exp<NNN>.typ` is the writeup. A tool is *reusable* science; an experiment is *one run* of it. Do not confuse the two.
- **Runner:** the `experiments/exp<NNN>.py` file: it runs a tool's CLI (or computes inline), renders figures from the data, and stages the record. It never imports a tool.
- **Article:** a prose-only writeup, `writings/ar<NNN>.typ`, with no runner and no data pipeline. Contrast an experiment, which does run one.
- **Writing:** any `writings/<id>.typ`: an experiment's writeup, an article, or a deck. It is a *meta* plus *body* pair the engine discovers and publishes.
- **Deck:** a `writings/<id>.slide.typ` Touying slide deck. It is paged-only, compiled to a stand-alone PDF and grouped under the `slides` collection (or its `meta.collection`, if set): listed, but not an entry.
- **Entry:** a published item that becomes its own page, meaning an experiment or an article. Decks are listed but are not entries.

The record and its wiring

- **Contract:** the file-based, language-neutral interface between a tool and an experiment: the tool writes a fixed set of files, the runner reads them by running the tool's CLI.
- **Artifacts:** the committed record under `artifacts/`: `artifacts/data/<id>/` holds a run's figures and `numbers.json`, `artifacts/pdfs/` holds shareable PDFs, and `artifacts/site/` is the gitignored web build.
- `numbers.json`: the aggregated, committed record of a run: each command's config plus headline metrics plus a provenance stamp. Writings read it so the numbers cannot drift.
- **Provenance:** the block stamped into every run: git commit SHA, a *dirty* flag, and a UTC timestamp, surfaced as a page and PDF footer.
- **Collection:** a group of entries sharing a `collection:` slug in their *meta*, grouped together on the homepage.

The full reference lives in GLOSSARY.md.

The folder structure

7 July 2026

A demolab repo has a place for everything, and the layout is the same in every repo. Once you know the four content directories, the engine, and the reconciled root files, you can find anything by name. A fresh clone ships *engine-only* (just `demolab-engine/` and the root files) and task scaffold lays down the empty content tree below.

```
demolab/
├─ tools/           the science - one directory per tool (tool.py + tests)
├─ experiments/     the runners - one expNNN.py per experiment
├─ writings/        the writeups - one .typ per entry, by id
├─ artifacts/       the committed record of every run
│   ├─ data/<id>/   figures + numbers.json - the neutral record
│   ├─ pdfs/        compiled PDFs (shareable)
│   └─ site/        the built web site - GITIGNORED
├─ demolab.yaml     optional branding + collections config
├─ HOUSESTYLE.local.md optional house-style overrides
├─ demolab-engine/  the engine - the BLACK BOX
├─ AGENTS.md · README.md · Taskfile.yml · pyproject.toml
└─ temp/           short-lived run scratch - GITIGNORED
```

The four content directories

These are yours. `tools/` holds the science, one directory per tool, each with its CLI in `tool.py` and a test beside it. `experiments/` holds the runners: one `expNNN.py` per experiment, which invokes a tool, renders figures, and stages its output. `writings/` holds the writeups, one `.typ` file per entry: an `expNNN.typ` reads an experiment's run, an `arNNN.typ` is a prose-only article, and an `arNNN.slide.typ` compiles to a standalone slide deck. `artifacts/` is the committed record: `data/<id>/` keeps each run's figures and `numbers.json`, and `pdfs/` keeps the shareable PDFs. Both of those are in git; `artifacts/site/` and `temp/` are gitignored because the build regenerates them.

What ties an experiment together is its *id*, not a folder. The same `exp000` threads through all three: `experiments/exp000.py` runs it, `artifacts/data/exp000/` records it, `writings/exp000.typ` writes it up.

The engine and the root files

`demolab-engine/` is the *black box*. It holds the Typst publisher, the scaffold templates, the agent runbooks, and the guides, and you never hand-edit it. When you update demolab, the whole engine is swapped wholesale. Everything you own lives *outside* it.

Two optional files sit at the root: `demolab.yaml` for branding and collections, and `HOUSESTYLE.local.md` for house-style overrides. The remaining root files, `AGENTS.md`, `README.md`, `Taskfile.yml`, `pyproject.toml`, and the CI workflow, are framework files that are reconciled by diff on update rather than swapped, so your local edits survive.

Learning it interactively

Type **STRUCTURE** in capitals and the coding agent will walk you through this tree, path by path, in your own repo.

The full reference lives in `STRUCTURE.md`.

Authoring slides

7 July 2026

A deck in demolab is a talk, not a writeup. It lives at `writings/<id>.slide.typ` and is built with Touying into a standalone PDF. The `.slide.typ` filename is the marker: the build compiles the deck on its own, links it from the site, and deliberately excludes it from the HTML pages and the book. Touying is paged-only and does not survive HTML export, so a deck stays a PDF and nothing else.

One structural difference from a prose entry: a deck declares `#let meta` so the site can list and link it, but it never declares a `#let body`. Its figures come from the same committed run outputs the writeups read, out of `artifacts/data/`, never from ad-hoc images.

Size against the canvas

A slide is a fixed rectangle: 842 by 474 points at 16:9. Once the title header and footer margins take their share, you have roughly 350 points of usable height on a titled slide. That number is the budget you lay everything out against. Size images in absolute points, not percentages: a percentage resolves against whatever region encloses it, which inside a grid cell is rarely the whole slide, so percent-sized figures come out unpredictably small and can overflow. Give each figure row a height that suits its shape, sum the rows, gutters, and captions, and confirm the total stays under budget. If a slide is mostly whitespace the figures are too small; if things do not fit, split the slide rather than shrinking text below readable sizes.

Check the page count

Overflow is silent. Oversized content does not clip or raise an error; Touying quietly spills it onto an extra page, and the deck grows without warning. So after every layout change, count the pages and compare against the number of slides you meant to have. Render the deck to images and count them, rather than trusting cached metadata from Finder or Spotlight, which goes stale. Then eyeball the dense slides at higher resolution to catch clipped captions or terms swallowed into a subscript.

Lift layouts from the catalog

You do not re-derive a layout from scratch. Every layout is a named, tested block in the gallery deck, and the guide carries an index of names paired with when to use each one: bullets, two-column comparisons, code panels, equation-with-terms, figure grids, big-number statements, section dividers, and more. You find the name, copy that block out of the gallery, and swap in your own content. The gallery holds the single page-count-checked copy of each layout, and a test keeps the index and the gallery in sync, so an entry can never point at a missing block.

You can also just type `SLIDES` in capitals and the coding agent will walk you through all of this interactively.

The full reference lives in `SLIDES.md`.

Writing style

7 July 2026

HOUSESTYLE is the taste demolab writes with, so the whole lab reads as one voice. It is **style**, softer than the hard invariants, and you follow it unless you have a reason not to. If you type HOUSESTYLE in capitals, the coding agent will walk you through it interactively.

How the prose reads

A write-up is written for a scientist, not a changelog. The title is a plain sentence about the science, never prefixed with the entry id or date, since those already sit in the meta strip. Lead with the claim in plain language, then let the figure and the numbers carry the evidence, and keep the motivation short. Report results honestly, including partial and negative ones, folded into the prose or a caption rather than hidden. One notable habit: no em-dashes in running prose. They are the loudest tell of machine-written text, so a comma, colon, parentheses, or full stop stands in.

How math is set

Equations are written in native Typst so they render as selectable math on the web and typeset in the PDF, never pasted as images. The single most important rule: after an equation, define *every* symbol in a list. A reader should never have to guess what a term means. For “approximately” use $\approx$ in prose and **approx** in math, never a tilde, because in Typst a tilde is a non-breaking space and quietly vanishes.

Figures, captions, and numbers

Every figure is drawn by a runner from the tool’s data and embedded in both the web page and the PDF from one rendered asset, so the plot style lives in one shared place instead of being reset per figure. Line plots go to vector SVG, dense or photographic content to high-dpi PNG, always on a white background with alt text. Ink is near-black by default; a single accent colour is earned only to separate traces sharing an axis. Captions are written to be read cold, standing on their own: a lead clause saying what the figure shows, a body naming each axis with units and defining every symbol, then one honest takeaway.

Numbers a run produced are never hand-typed, in the prose or in a caption. They come from the run’s recorded output, so a stated result always traces back to the computation and cannot drift. Likewise, citations use the citation helpers rather than typed brackets, so numbering, links, and hover-popovers stay correct as you reorder.

Making it your own

This is demolab’s *default* style, and you can override it. Drop a HOUSESTYLE.local.md in the repo root: by default your rules add to and supersede these, or you can tell it to replace them entirely and inherit none of demolab’s taste. Either way the underlying tool-to-experiment contract still holds; only the style is yours to bend.

The full reference lives in HOUSESTYLE.md.

The contract

7 July 2026

RULES is the single source of truth for how a demolab repo is put together. It covers two things at once: the principles a repo follows, and the contract that lets an experiment call a tool and turn its output into a published page. This is the plain-English tour. If you want the coding agent to walk you through it live, just type **RULES** in capitals and it will.

The toolchain

demolab publishes with a small, fixed stack. Python runs through `uv`, you never call `python` directly, you say `uv run python ...`, and `uv sync` after pulling. Publishing is done entirely with the `typst` CLI: it compiles the whole repository into a website and a set of PDFs in one pass, no Node or bundler involved. Common commands are wrapped in a `Taskfile`, so reach for `task build` or `task dev` rather than the raw tools. Python is the default, not a hard requirement: the contract below is file-based and language-neutral, so the tool layer could be MATLAB, R, or Julia instead.

The engine/content firewall

A demolab repo has a clear line between framework and your work. The engine, the `Typst` build, the runbooks, the guides (this file included), and the scaffold, is a black box: you never edit it, and it is swapped wholesale when you run an update. A thin middle layer (the root `README.md`, `Taskfile`, `pyproject.toml`, CI) is reconciled by diff rather than replaced. A couple of root files like `demolab.yaml` are your overrides and are never touched by updates. Everything under `tools/`, `experiments/`, `writings/`, and `artifacts/` is 100% yours, freely deletable and replaceable. Knowing which zone a file sits in tells you whether an update will overwrite it.

The tool-to-experiment contract

A *tool* holds reusable science and speaks only through files: a runner reaches it by running its CLI as a subprocess, never by importing it. Tools emit machine-readable *data*, not finished plots; drawing the figure is the runner's job, so a result can always be redrawn. A tool writes a fixed file set (`config.json`, `output.json`, `manifest.json`, and its data) into scratch under `temp/`; the runner reads the manifest to learn the headline metrics, renders the figures, and aggregates everything into a committed `numbers.json` under `artifacts/data/`. Because each write-up reads its own `numbers.json` and figures straight from that run, a number on the page can't drift from the code that produced it.

To **add an experiment**: add or reuse a tool subcommand, write an `expNNN.py` runner that declares its commands and renders figures, then write an `expNNN.typ` write-up that reads the run. To **add a tool**: give it a directory under `tools/`, reuse the `setup_run_dir` / `write_output` pattern, and ship tests. A **writing** is just a `meta` + `body` pair in a `.typ` file, and an article needs no tool at all.

The full reference lives in `RULES.md`.

Getting started

7 July 2026

demolab is a lab notebook for computational science. You write the science as code, an experiment runs it and emits data, a write-up reads that data, and one build turns the whole repository into a website and a set of PDFs. Numbers and figures come from the run, never retyped, so a result on the page can't quietly drift from the code that produced it. A coding agent operates it in plain language; you stay in the loop.

Install

Open a coding agent in an empty folder and paste:

Clone github.com/eoinmurray/demolab and follow its `GETTING-STARTED.md` strictly.

It runs the `GETTING-STARTED` runbook and sets everything up with you: the toolchain — `uv` for Python, `Typst` for publishing, and `go-task` as the command runner — then your own copy of the repository, the scaffold, and your first experiment. You drive everything through `task`, which wraps `uv` and `typst`; you never call `pip` or `python` directly.

Your first run

The agent drives this: `GETTING-STARTED` stands the lab up, then builds your first experiment with you — your own science, or a suggested starter — and you watch it land on a live page. By hand, the loop is:

```
task install      # resolve dependencies
task scaffold    # lay down writings/ experiments/ tools/ artifacts/
task dev         # serve the site at localhost:3000, live-reloading on save
task run -- exp000 # run an experiment end to end (once you've written one)
```

Change a parameter, `task run` again, and watch the page update — figures and numbers, no prose touched. That's the whole point. `task build` compiles everything once; `task test` runs the suite. (Want a worked example in your tree? `task add-demo-content` overlays the same demo this site is built from; `task clear-demo-content` removes it.)

How a lab is laid out

Four content directories, each with one job:

- `tools/`: the **science**. Reusable models and solvers, one directory per tool. A tool emits **data** (a `numbers.json` plus the data behind each figure), never plots, and stays language-agnostic.
- `experiments/`: the **runners**. One `expNNN.py` per experiment: it calls a tool (or computes inline), renders the figures into `artifacts/data/expNNN/`, and stages a `numbers.json` of the headline metrics.
- `writings/`: the **write-ups**. One `.typ` per entry. It reads the run, `#let run = json("/artifacts/data/expNNN/numbers.json")`, embeds the figures, and pulls every number from that file. Prose-only articles (like this one) and slide decks live here too.
- `artifacts/`: the committed **record**: `data/<id>/` (figures + numbers) and `pdfs/` (a PDF per entry, plus a bound book). The built website (`site/`) is regenerated by CI, so it's gitignored.

The engine itself lives in `demolab-engine/`: a black box you never edit, swapped wholesale when you update.

The loop

The discipline is one rule with teeth: *nothing on the page is typed by hand*.

1. A tool computes and writes data files.
2. A runner turns that data into figures and a `numbers.json`.
3. A write-up reads the `numbers.json` and embeds the figures.
4. `task build` compiles the repository into a website, a PDF per entry, and a book.

Change the code, rebuild, and the prose, figures, and numbers update together. Every result carries the git provenance of the run that made it.

Publish

One build emits three things from a single source: a **website** (HTML with real, selectable math), a **PDF per entry**, and a bound **book**. To put it online, `task deploy-setup` drops a GitHub Pages workflow; enable Pages in the repository settings and push. The site uses relative links, so it works under any path.

Driving it with an agent

demolab is meant to be run by a coding agent. You type a **name in capitals** and it acts:

- **HELP**: list everything it can do.
- A **guide** name (`RULES`, `HOUSESTYLE`, `SLIDES`, `STRUCTURE`, `GLOSSARY`, `SUPPORT`): it walks you through that reference.
- A **runbook** name (`LINT`, `DOCTOR`, `MIGRATE-STACK`, ...): it runs that procedure step by step.

The reference layers are documented right here in this collection: a page for each **guide** (always-on conventions), plus a **runbooks** index (the on-demand procedures).

Introduction

9 July 2026

Here is an observation. Software engineers use coding agents constantly; scientists, far less so. That gap is not about talent or curiosity, it is about tooling: the agent arrived in a form built for shipping software, not for running experiments and writing them up. demolab is an attempt to close that gap. It is a lab notebook for computational science in which the science lives as code, an agent operates it in plain language, and one build turns the whole repository into a website and a set of PDFs. This page makes the case for it, in the same two acts as the talk it is drawn from: first the agents, then the workflow built on them.

Coding agents got good

A coding agent is not autocomplete. It reads a repository, runs programs, edits files, and checks its own work; it can do anything a terminal can do. The line runs autocomplete (2021), to chat (2023), to agents that do real work (2025), and only the last step made it useful for anything as unforgiving as science.

The clearest evidence is SWE-bench Verified [1, 2], a benchmark of real, human-validated GitHub issues that an agent must resolve end to end. The trajectory, shown below, is the point: at the benchmark's 2024 launch the best systems closed about a third of the issues, and by late 2025 that figure had climbed past three quarters. A tool that was a novelty two years ago now does a large fraction of a working engineer's tickets unaided. Many such agents exist, commercial and open, so there is no vendor lock-in.

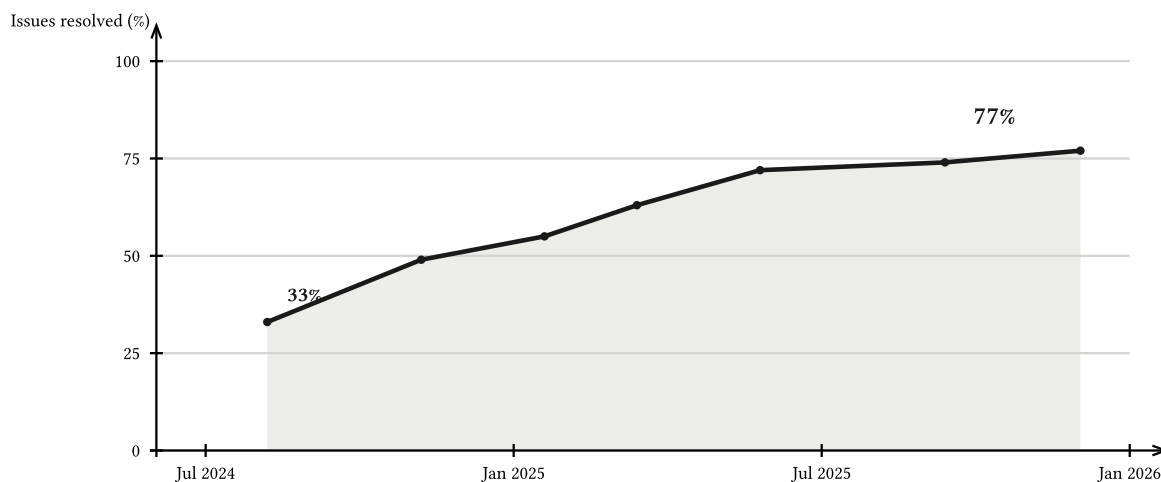


Figure 1: Share of real, human-validated GitHub issues a coding agent resolves end to end, on SWE-bench Verified (swebench.com), from the benchmark's 2024 launch to late 2025. The vertical axis is percent of issues resolved; the horizontal axis is date. The resolved fraction climbs from 33% to 77% in about eighteen months. These are software bugs, not science, so the level does not transfer to research; the slope is what matters.

It would be dishonest to sell only the strengths. Agents do the same work faster, clear the backlog of things you never had time for, document as they go, and read a paper straight into runnable code, and they work far better over a whole repository than over a snippet. They are also confidently wrong: they invent APIs, numbers, and citations. They leave you with comprehension debt, code you shipped but do not understand. Their judgement is weak; an agent will not kill a bad experiment on its own. So the honest summary is short. A coding agent

is a powerful tool that needs supervision, and a framework worth having is one that supplies the rails.

What demolab is

demolab is that framework. It writes code, runs programs, and documents the results, and it holds the whole thing to one loop: run, produce data, write it up, publish. One build compiles the repository into a website (with real, selectable math) and a PDF per entry plus a bound book. The discipline underneath is a single rule with teeth: nothing on the page is typed by hand. Numbers and figures come from the run, so a result cannot quietly drift from the code that produced it, and every result carries the git provenance of the run that made it. It is reproducible by default, and it stays reproducible a year later because one command rebuilds all of it.

Those properties answer real failure modes. Results drift when numbers are retyped and figures go stale; here they come from the run. Reproducibility rots until, a year on, nothing builds; here one command rebuilds everything. Code and paper drift apart into separate repositories; here the experiment and its write-up live together. And agents need rails; here they operate in plain language while the science stays yours.

A few principles hold the shape together:

1. **Tools compute, experiments analyse.** A clean split between the reusable science and the runner that exercises it.
2. **One experiment is a runner plus a write-up.**
3. **Components talk through files, never imports.** A tool emits data; a runner reads it.
4. **Raw data is disposable; the record is committed.**
5. **Bring any stack.**
6. **Provenance is built in.**
7. **Documentation is everything.**

The shape of a repository

A lab is four content directories, each with one job:

- **tools/:** the science, as reusable models and solvers. A tool emits data (a `numbers.json` plus the data behind each figure) and stays language-agnostic.
- **experiments/:** the runners. One `expNNN.py` per experiment calls a tool, renders the figures, and stages a `numbers.json` of the headline metrics.
- **writings/:** the write-ups. One `.typ` per entry reads the run and embeds its figures and numbers. Prose articles like this one, and slide decks, live here too.
- **artifacts/:** the committed record, holding the figure data and a PDF per entry.

The engine itself lives in `demolab-engine/`, a black box you never edit and swap wholesale when you update. Publishing is built on Typst, a modern take on LaTeX: instant incremental compiles instead of slow multi-pass ones, readable errors with line numbers instead of macro soup, and one source that renders to both web and PDF instead of PDF alone. Because Typst imports files, the numbers on the page are the numbers from the run.

Driving it with an agent

demolab is meant to be run by a coding agent, though you can operate every command by hand. You type a name in capitals and it acts. **HELP** lists everything it can do. A guide name (**RULES**, **HOUSESTYLE**, **SLIDES**, **STRUCTURE**, **GLOSSARY**, **SUPPORT**) walks you through an always-

on convention. A runbook name (`GETTING-STARTED`, `LINT`, `DOCTOR`, `FROM-PAPER`, `MIGRATE-STACK`, and the rest) runs an on-demand procedure step by step: read a paper into code, convert a notebook, migrate the language, update the engine. Underneath it all is an ordinary command runner, so `task run -- exp000` runs an experiment, `task dev` serves the site with live reload, and `task build` compiles the lot. The default stack is Python, but tools talk through files rather than imports, so you can switch to MATLAB, R, Julia, or Octave by asking, and the contract and the typesetting stay put.

Where it's going

Everything above is demolab today: implemented and ready to use. Two directions define where it goes next.

The first is autoresearch. Give the agent a goal and some compute, and it runs the loop itself: goal, code, compute, read the data, document, iterate. Because the rails already exist, every attempt lands as a committed experiment and write-up with provenance and a published trail, for free. It is built, and still being tested.

The second is scale. Version one is one person's lab notebook. The next step is a lab on a GitHub organisation: many researchers, one shared flow, with questions and answers, code review, and shared code compounding across the group. The notebook becomes the lab.

References

1. Jimenez CE, Yang J, Wettig A, Yao S, Pei K, Press O, Narasimhan K (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR 2024*. doi:10.48550/arXiv.2310.06770
2. OpenAI (2024). Introducing SWE-bench Verified. Live leaderboard at swebench.com.

LIF voltage traces and a recurrent network raster

13 May 2026

A single leaky integrate-and-fire (LIF) neuron driven by a steady supra-threshold current should fire periodically; wire 200 of them together with sparse excitation and noisy per-neuron bias and the population should sustain ongoing, irregular firing rather than fall silent or lock into synchrony. We ran both (one neuron, then the network) and report the voltage trace, the raster, and the measured firing rates.

Methods

The LIF membrane voltage decays toward a resting potential and is pushed around by injected current; a threshold crossing emits a spike and resets the voltage.

- **Subthreshold dynamics.** Integrate the leaky-membrane ODE

$$\tau_m \frac{dV}{dt} = -(V - V_{\text{rest}}) + R_m I,$$

with V the membrane potential, τ_m the membrane time constant, V_{rest} the resting potential, R_m the input resistance, and I the injected current.

- **Spike-and-reset.** When $V(t) \geq V_{\text{th}}$, record a spike at t and set $V \leftarrow V_{\text{reset}}$, with V_{th} the threshold and V_{reset} the reset potential.
- **Integration.** Forward Euler at timestep Δt ,

$$V \leftarrow V + \frac{\Delta t}{\tau_m} [-(V - V_{\text{rest}}) + R_m I].$$

- **Single-neuron run.** Drive one neuron with a constant supra-threshold tonic current.
- **Network run.** Wire 200 neurons with sparse, all-excitatory recurrence. Each neuron i takes total current $I_i(t) = I_i^{\text{bias}} + s_i(t)$, with noisy bias $I_i^{\text{bias}} \sim \mathcal{N}(\mu, \sigma^2)$ and synaptic current s_i that decays between spikes and is kicked by incoming ones,

$$s_i \leftarrow s_i e^{-\Delta t / \tau_{\text{syn}}} + \sum_{j \in \text{spiked}} W_{ij},$$

where τ_{syn} is the synaptic time constant and $W_{ij} = w C_{ij}$ with connectivity $C_{ij} \sim \text{Bernoulli}(p)$, weight w , and no self-connections. Each neuron is held at V_{reset} for a refractory period after spiking.

Parameter values are tabulated below.

Results

With a steady tonic input above threshold, the single neuron fires periodically at 90 Hz. Each upward sweep is the membrane charging through its RC time constant; each vertical line is a reset after a spike.

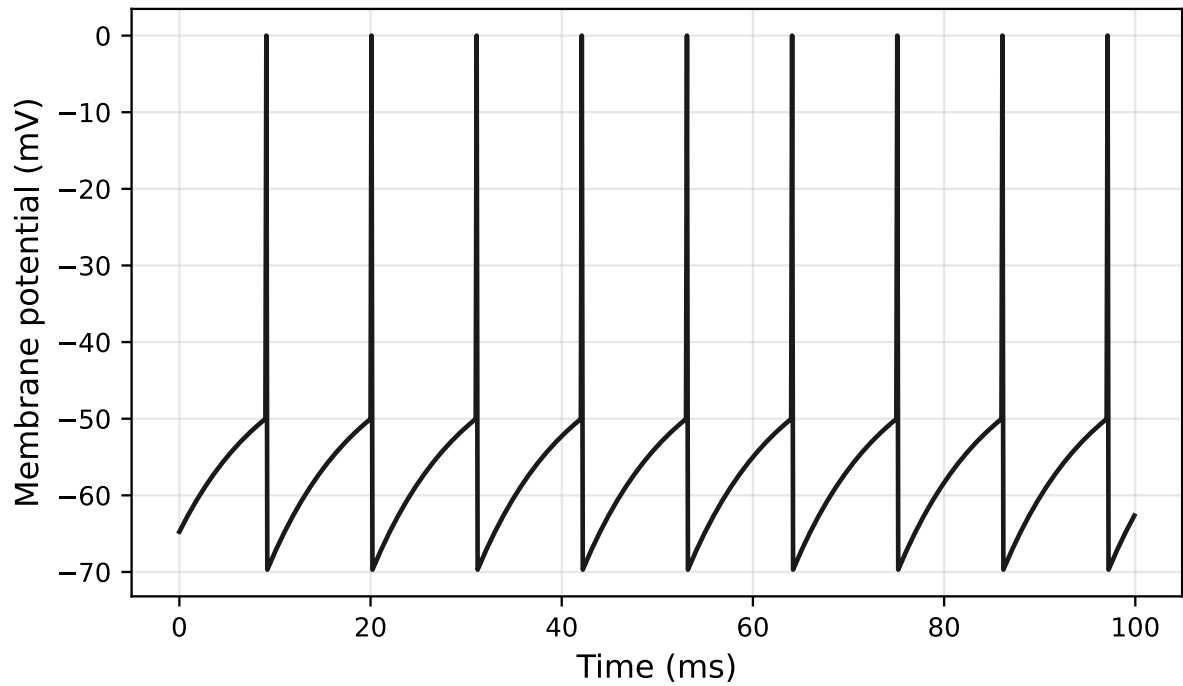


Figure 2: Single LIF neuron under tonic input: membrane potential (mV) against time (ms). The membrane charges toward its steady state and resets at each threshold crossing, giving periodic firing at 90 Hz.

The network sustains ongoing irregular firing: a population mean of 104 Hz, spanning 56–148 Hz across neurons, no silence, no runaway synchrony. The raster shows every spike from every neuron over half a second; the panel beneath tracks the network’s mean total input current.

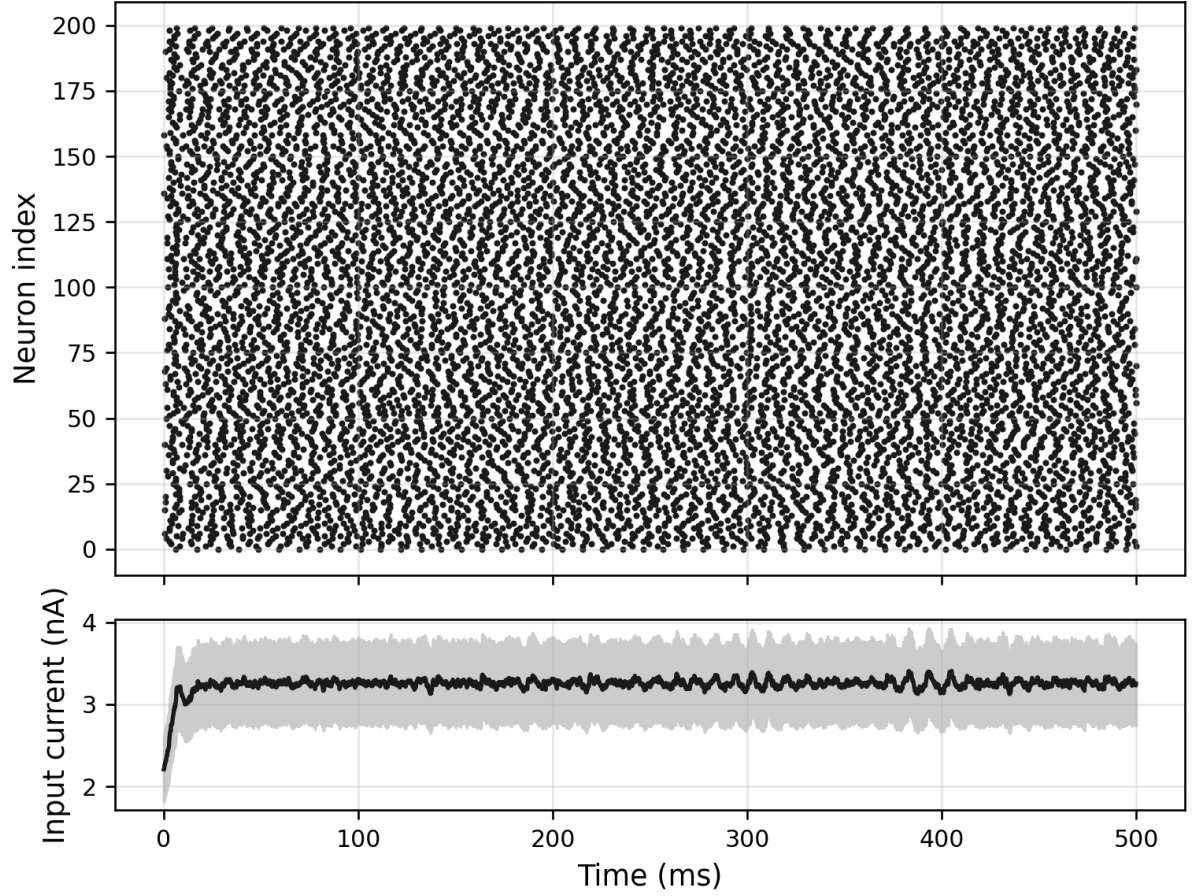


Figure 3: Network raster (top: spike time vs neuron index) and mean input current (bottom, nA vs ms) across 200 recurrently coupled LIF neurons over 500 ms. Mean firing rate 104 Hz; the shaded band is ± 1 s.d. of the per-neuron total current across the population.

LIF run parameters

Parameter	Value
current	2.5
duration	100
dt	0.1
tau_m	10
r_m	10
v_rest	-65
v_reset	-70
v_thresh	-50
firing_rate_hz	90

Network run parameters

Parameter	Value
n	200
duration	500
dt	0.1
seed	0
mean_firing_rate_hz	104.2
min_firing_rate_hz	56
max_firing_rate_hz	148

Generated from commit db62207 (uncommitted changes) · 5 July 2026

EIF voltage traces and a recurrent network raster

13 May 2026

The exponential integrate-and-fire (EIF) neuron replaces LIF’s hard threshold with an explicit exponential term that models the upswing of a spike. Under the same tonic drive as exp000 it should fire at a different rate than LIF (the exponential accelerates spike initiation, but the upswing itself takes time) while a network of them should behave much like the LIF network with a softened spike onset. We ran the single neuron and the network and compare against exp000.

Methods

The EIF neuron keeps LIF’s one-variable spirit but grows an exponential term, so the threshold becomes a soft inflection point rather than a discontinuity.

- **Subthreshold dynamics.** Integrate

$$\tau_m \frac{dV}{dt} = -(V - V_{\text{rest}}) + \Delta_T \exp\left(\frac{V - V_T}{\Delta_T}\right) + R_m I,$$

with V the membrane potential, τ_m the membrane time constant, V_{rest} the resting potential, R_m the input resistance, I the injected current, V_T the soft threshold, and Δ_T the slope factor.

- **Peak-and-reset.** When $V(t) \geq V_{\text{peak}}$, record a spike at t and set $V \leftarrow V_{\text{reset}}$, with V_{peak} the peak cutoff and V_{reset} the reset potential.
- **Threshold behaviour.** V_T is no longer a discontinuity: it is the point at which the exponential term takes over and the voltage blows up on its own. Small Δ_T approximates LIF; larger Δ_T smooths spike initiation.
- **Integration.** Forward Euler at timestep Δt ,

$$V \leftarrow V + \frac{\Delta t}{\tau_m} \left[-(V - V_{\text{rest}}) + \Delta_T \exp\left(\frac{V - V_T}{\Delta_T}\right) + R_m I \right].$$

- **Single-neuron run.** Drive one neuron with a constant tonic current.
- **Network run.** Structurally identical to exp000: 200 all-excitatory neurons, sparse p random connectivity, weight w , exponentially-decaying synaptic input with time constant τ_{syn} , noisy per-neuron bias $I_i^{\text{bias}} \sim \mathcal{N}(\mu, \sigma^2)$, and a refractory period, with every neuron now obeying the EIF dynamics above.

Parameter values are tabulated below.

Results

Under the same tonic input, the EIF neuron fires at 60 Hz, against 90 Hz for the LIF neuron in exp000: the exponential term accelerates spike initiation, but the upswing takes time, so the inter-spike period shifts.

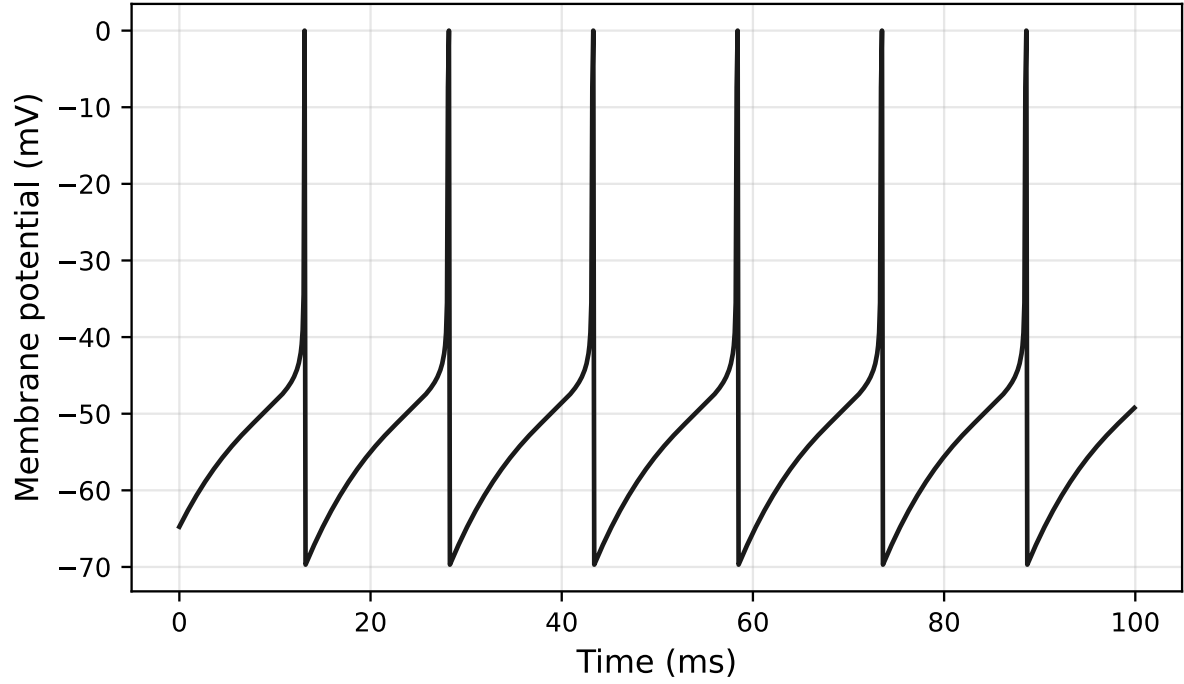


Figure 4: Single EIF neuron under tonic input: membrane potential (mV) against time (ms). The soft exponential threshold produces a curved spike upswing rather than LIF's hard reset, and periodic firing at 60 Hz.

The EIF network sustains irregular firing at a population mean of 69 Hz, spanning 32–98 Hz across neurons, the same asynchronous-irregular regime as the LIF network, shifted in rate.

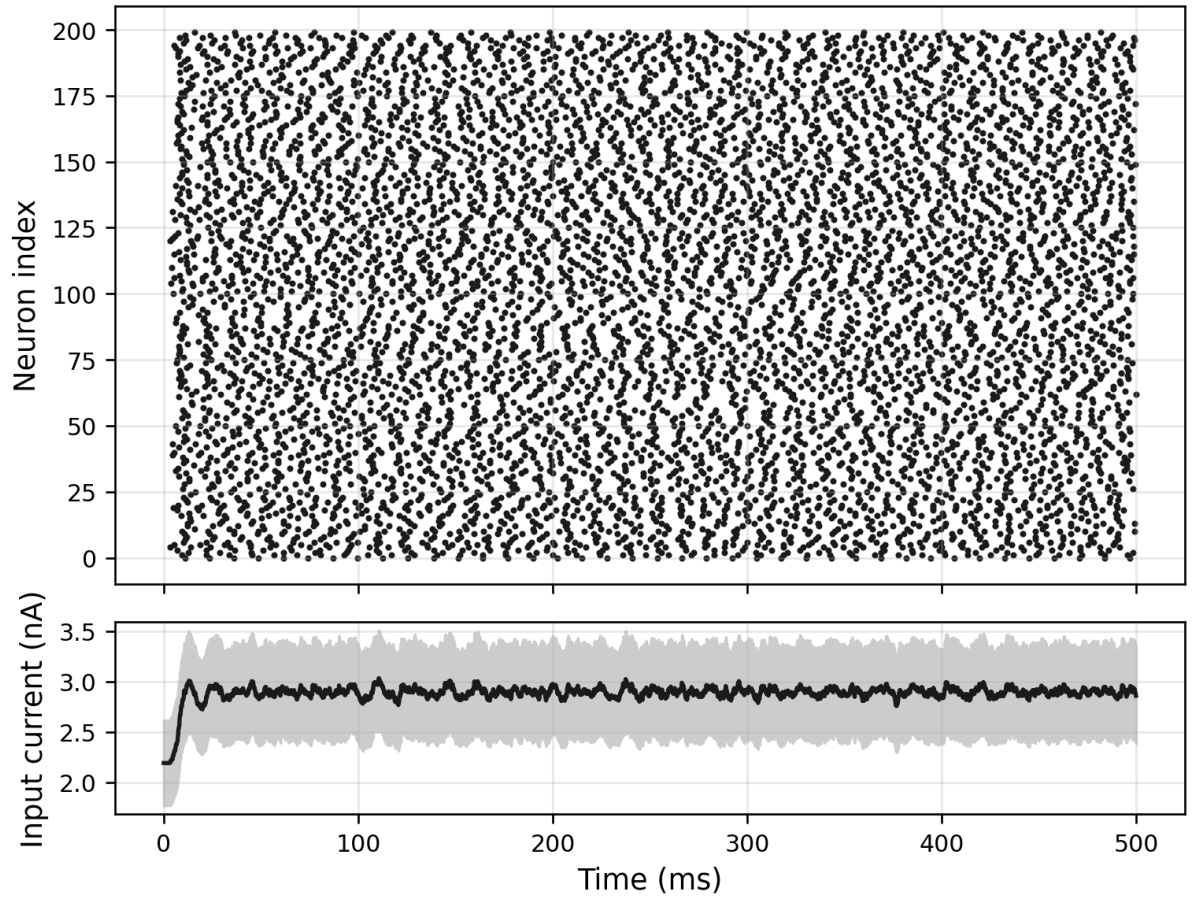


Figure 5: EIF network raster (top: spike time vs neuron index) and mean input current (bottom, nA vs ms) across 200 neurons over 500 ms. Mean firing rate 69 Hz; the shaded band is ± 1 s.d. of the per-neuron total current.

EIF run parameters

Parameter	Value
current	2.5
duration	100
dt	0.1
tau_m	10
r_m	10
v_rest	-65
v_reset	-70
v_t	-50
delta_t	2
v_peak	0
firing_rate_hz	60

EIF network run parameters

Parameter	Value
n	200
duration	500
dt	0.1
seed	0
mean_firing_rate_hz	69.11
min_firing_rate_hz	32
max_firing_rate_hz	98

Generated from commit db62207 (uncommitted changes) · 5 July 2026

A passive cartpole in MuJoCo

5 June 2026

The cartpole is the simplest physics-engine model that still looks like something: a cart slides on a frictionless rail with a thin pole hinged on top. With no controller, released from a small offset, the pole should topple under gravity while the cart barely moves, since the rail absorbs almost all of the hinge's reaction force. We ran a four-second passive simulation and recorded the fall.

Methods

The model is a small MJCF cartpole:

- one slide joint for the cart along the x-axis
- one hinge joint for the pole about the y-axis
- damping coefficients 0.05 and 0.005 for the cart and pole
- timestep 0.005 s, simulation length 4 s
- initial pole offset $\theta_0 = 0.15$ rad ($\approx 8.6^\circ$); everything else starts at rest

The simulation is stepped and a frame sampled every 1/60 s into the video below.

Results

[Video — Passive cartpole released from a small offset of $\theta_0 = 0.15$ rad: with no controller the pole topples under gravity while the cart stays almost stationary on the rail. · view the web edition to play.]

The pole crosses 60° from vertical at $t \approx 0.6$ s, ends the run essentially horizontal (final angle 89.5°), and the cart barely moves, only $\approx 3.1 \times 10^{-5}$ m of recoil over the full four seconds, because almost all the angular momentum of the falling pole is absorbed by the rail's reaction force rather than by translating the cart.

Run parameters

Parameter	Value
theta0	0.15
duration	4
fps	60
fall_time_s	0.595
final_angle_deg	89.52964154260019
max_cart_displacement_m	0.00003051021168075732

A chaotic double pendulum in MuJoCo

5 June 2026

The double pendulum is the textbook example of a deterministic system that is, in practice, unpredictable: two arms hinged in series have four state variables $(\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2)$ and a Lagrangian with one nonlinear coupling term, and that is enough for trajectories that start arbitrarily close together to separate exponentially. We ran two identical pendulums side by side, started 10^{-3} rad apart, and measured when and how far they diverge.

Methods

The demo runs two double pendulums at once, side by side in the same MuJoCo scene, with identical mass, length, and damping. The only difference is the initial angle of the upper arm,

$$\theta_1^A(0) = 2.0, \quad \theta_1^B(0) = 2.0 + 10^{-3},$$

in radians. The MJCF model is two double-pendulum chains:

- two chains of two hinge arms each, anchored at $x = \pm 0.6$ m
- arms are 0.4 m capsules; a small site marks each tip so its world position can be read
- timestep 0.002 s with the RK4 integrator, important in the chaotic regime, where forward Euler would let small numerical errors swamp the actual divergence
- damping 0, so total energy is conserved (give or take RK4’s drift)

Tip separation is measured in each pendulum’s *anchor-local* frame (each tip position minus its own anchor) so the constant 1.2 m horizontal offset between the two scenes doesn’t contaminate the metric.

Results

For the first two seconds or so the two pendulums trace what looks like the same motion. Around $t \approx 3$ s the tip-to-tip distance crosses the 0.1 m threshold, and from then on they are visibly doing completely different things. The Lyapunov-like blow-up is the whole story.

[Video — Two near-identical double pendulums (started 10^{-3} rad apart) in one MuJoCo scene: they track each other for the first couple of seconds, then diverge visibly once the tip separation passes the 0.1 m threshold. · view the web edition to play.]

The maximum separation hits 1.21 m, basically the full extent of a single pendulum (0.8 m), meaning that at some point one tip is at the top of its swing while the other’s is at the bottom. The final separation of 0.56 m is just where they happen to be at $t = 8$ s; with a chaotic system, “where they end up” is meaningless past the divergence time.

Run parameters

Parameter	Value
theta1	2
theta2	0
epsilon	0.001
duration	8
fps	60
separation_threshold	0.1
separation_time_s	2.968
max_separation_m	1.2096309484781214
final_separation_m	0.5648867078342311

Generated from commit 60d7d7a · 3 July 2026