Phlux.rs

Monte Carlo N-Particle Transport Framework

SoHo Labs

January 2026

Contents

1 Foreword

1.1 Workspace Breakdown

Crate	Description
phlux	Monte Carlo Particle Transport
crater	Constructive Solid Geometry
xs	Nuclear Cross Section Service
rott	Row-Oriented Tensor Types
roam	Discrete Stochastic Processes

1.2 Quick Links

1.2.1 Demos

[TBD - Coming soon]

1.2.2 Resources

- Repository
- Documentation
- Demo Site
- Issue Tracker

1.3 Notation

1.3.1 General Tensor Notation

This workspace employs tensor notation throughout. Almost every tensor encountered in this book is of rank ≤ 2. Colloquially, these are classic scalars, vectors and matrices.

- Objects with no indices are scalar-valued. (e.g. λ, $f : \mathbb{R}^n \to \mathbb{R}$)
- Objects with one or more indices are tensor-valued. (e.g. x^i, X^{ij}, $F^i(X^{ij}) : \mathbb{R}^{m \times n} \to \mathbb{R}^m$)
- Rank-1 tensors (vectors) are represented in lower-case, rank-r where $r > 1$ tensors use upper-case. (e.g. x^j, X^{ij})
- Tensor operations use Einstein summation notation, eliding the summation symbol (e.g. $x^j x_j = \sum_j x^j x_j$)
- Element-wise or broadcasted multiplication is indicated by juxtaposition (e.g. $x^i y^i$). Shared indices indicate broadcasted multiplication (e.g. $X^{ij} y^i$ is the action of multiplying the i-th row of X^{ij} by the i-th element of y^i).
- When used, the spatial dimension is indexed by j, and the batch dimension is indexed by i. (e.g. x^j, X^{ij})

2 Phlux: Transport Theory

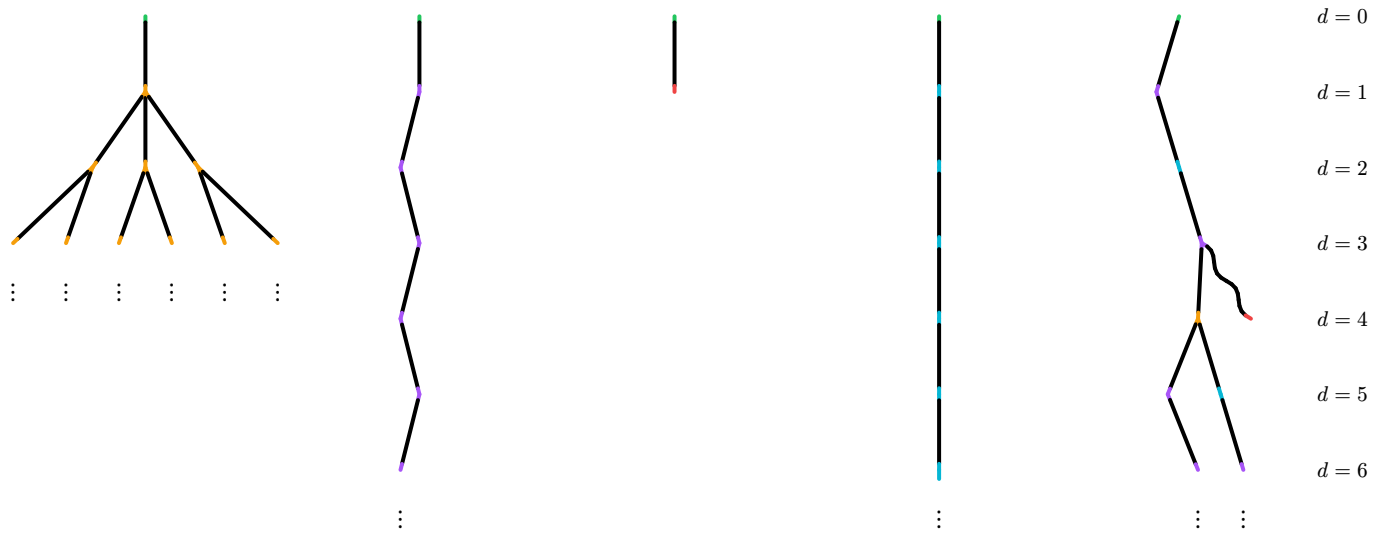
Particle transport can be formulated as a discrete stochastic process, where the system state evolves through random interactions with geometry and materials. The `phlux` crate builds on the `roam` framework (see the Roam chapter for background on discrete stochastic processes) to implement Monte Carlo particle transport.

2.1 Transport as a Stochastic Process

Particle transport is a discrete Markov process. Over this chapter we will formulate N-particle transport using the abstractions provided by `roam.rs` (discrete stochastic processes) and `rott.rs` (row-oriented tensor types).

To begin, consider the history of a single particle. Below are a number of example histories, all beginning from a single source at depth $d = 0$. For now, this initial particle is a neutron.

In these history diagrams, solid black lines indicate particles traveling through space-time. Each *kink* in the particle trajectory represents a concrete interaction between the particle and a nucleus, or some non-physical part of the system, such as a material boundary. These are called *events*, and they are colored by type (fission, scattering, etc.). Particle types (neutron, photon, etc.) are identified by the shape of the line, bearing resemblance to Feynmann diagrams.



(a) Supercritical **(b) Reflective** **(c) Absorptive** **(d) Transmissive** **(e) Diverse**

Figure 1: Contrived particle histories over time as depth increases.

In example *(a)*, our neutron collides with a fissile nucleus, generating 3 daughter progeny nucleus via fission. This chain reaction continues, and progeny neutrons induce subsequent fission events at $d = 2$. This is a highly supercritical system; the neutron population increases exponentially with depth.

In *(b)*, the neturon population is a constant; our source particle scatters elastically, inducing the creation of no additional progeny beyond itself. This history may represent a particle in a vaccuum encased by a perfectly reflective box.

Example *(c)* is trivial. The neutron is absorbed at its first collision.

Example *(d)* depicts a particle repeatedly *transmitting* between *cells* (uniform volumes of material) without interaction. This is non-physical; the particle experiences no change in state throughout the history.

Finally, in *(e)* we depict a more realistic particle history, containing a diverse set of events and particle types. Notably in depth $d = 3$, the lone neutron undergoes inelastic scattering with a nucleus, spawning a progeny *photon*, depicted by the squiggly line. This photon is immediately absorbed at its next event. In a real simulation, the particle's history is determined by the system geometry, and material *cross sections* (see xs.rs) from which event probabilites are computed.

It is **crucial** to reinforce that this is a *Markov* process; the event observed at $d = D$ is completely determined by the parent event at $d = D - 1$, and no states prior ($d < D - 1$). A fundamental assumption of `phlux.rs` is that particles do not interact, and thus all histories are independent from each other, as well.

2.1.1 Particles and Events: Structures-of-Arrays

Because transport is a Markov process, and each particle is independent, this problem is embarassingly parallel. We can simulate the histories of *batches* of particles through the system concurrently, leveraging techniques and hardware designed to accelerate such calculations.

Particles are grouped into batches, and their data are organized in a **Row-Oriented Tensor Type** (`Rott`). This is a classic *structure-of-arrays* approach, which differs from the *array-of-structures* model, where one would expect to see `Vec<Particle>`, or similar.

```
pub struct ParticleBatch<B: Backend> {
    origins: Tensor<B, 2, Float>,         // [M, 3] - birth positions
    directions: Tensor<B, 2, Float>,      // [M, 3] - normalized vectors
    particle_types: Tensor<B, 1, Int>,    // [M] - Neutron(0) or Photon(1)
    energies: Tensor<B, 1, Float>,        // [M] - MeV
    particle_ids: Tensor<B, 1, Int>,      // [M] - unique IDs
    depths: Tensor<B, 1, Int>,            // [M] - generation level
    origin_event_ids: Tensor<B, 1, Int>,  // [M] - parent event ID
    start_times: Tensor<B, 1, Float>,     // [M] - creation time
}
```

The events induced by these particles follow a similar memory model (i.e., a `Rott`)

```
pub struct EventBatch<B: Backend> {
    origins: Tensor<B, 2, Float>,      // [M, 3] - spatial positions
    event_types: Tensor<B, 1, Int>,    // [M] - categorical type
    particle_ids: Tensor<B, 1, Int>,   // [M] - incident particle ID
    event_ids: Tensor<B, 1, Int>,      // [M] - unique event ID
    depths: Tensor<B, 1, Int>,         // [M] - generation level
    times: Tensor<B, 1, Float>,        // [M] - time of flight
}
```

These tensor-based structures are used during GPU-accelerated simulation. After each simulation step, data is extracted and stored in a backend for querying (see Section 2.3).

In mathematical notation, we typeset these types according to the `rott.rs` standard:

$$\mathbb{P}^i \tag{1}$$

and

$$\mathbb{E}^i \tag{2}$$

The internal state of our Markov process is therefore $\mathbb{E}^{i(d)}$, representing collection of events across all histories in the batch at depth d.

2.2 The Transport Operator

The stochastic transport operator $\mathbb{T}^i$ advances the simulation from depth d to $d+1$:

$$\mathbb{E}^i(d+1) = \mathbb{T}^i(\mathbb{E}^j(d)) \tag{3}$$

This decomposes into two phases:

1. **Progeny Sampling**: From events $\mathbb{E}^j(d)$ and incident particles $\mathbb{P}^j(d-1)$, sample progeny $\mathbb{P}^i(d)$ from nuclear cross section distributions.
2. **Particle Casting**: Simulate the flight of $\mathbb{P}^i(d)$ through the geometry, producing events $\mathbb{E}^i(d+1)$.

Formally:

$$\mathbb{T}^i(\mathbb{E}^j(d)) = \mathbb{R}^i(\mathbb{P}^i(\mathbb{E}^j(d), \mathbb{P}^j(d-1))) \tag{4}$$

The transport loop iterates until all particles are terminated (absorbed or escaped):

```
while active_particles > 0 {
    let progeny = sample_progeny(&events, &particles);
    let cast_result = geometry.particle_cast(&progeny, rng);
    storage.store(&cast_result);
    events = cast_result.events;
    particles = progeny;
}
```

2.2.1 Cell Identification

Before casting, each particle must be assigned to a geometry cell. Given particle positions $\boldsymbol{r}^i$, we compute cell indices c^i via point-location queries against the CSG regions:

$$c^i = \text{argmax}_c \quad \chi_c(\boldsymbol{r}^i) \tag{5}$$

where χ_c is the indicator function for cell c. The implementation sorts cells by volume (smallest first) and terminates early once all particles are assigned, avoiding redundant CSG evaluations.

2.2.2 Homogenization

A particle in cell c is *guaranteed* to experience its next event *within* cell c. Note that a Transmission event at depth d means the particle enters a new cell c' in $d+1$.

This feature enables an important optimization: we can perform exactly one batched particle cast and resultant events are guaranteed to be within the cell.

Particles are *homogenized*, sorted by cell index so that particles in the same cell are contiguous:

$$\mathbb{P}^i \xrightarrow{\pi} \mathbb{P}^{\pi(i)} \quad \text{where} \quad c^{\pi(i)} \leq c^{\pi(i+1)} \tag{6}$$

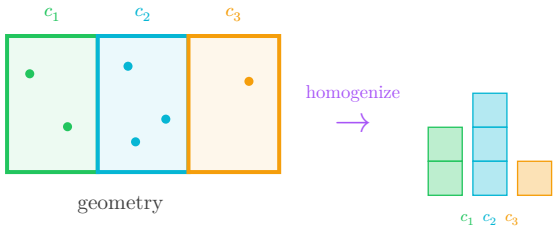


Figure 2: Homogenization: particles scattered across cells (left) are sorted into contiguous batches by cell (right).

```
// Homogenization: group particles by cell
let labeled = particles.map(|batch| {
    let labels = find_cell_indices(&cells, batch.origins());
    batch.partition_by_labels(labels)
});
let homogenized = labeled.concat_by_label(max_batch_size);
```

This is a *spatial-based acceleration structure* that eliminates unnecessary ray casts.

2.2.3 Batched Particle Casting

For each homogenized batch, ray casting determines where particles interact:

1. **Sample free-flight distance**: $s \sim \text{Exp}(\Sigma_t)$ where Σ_t is the total macroscopic cross section
2. **Ray-trace to cell boundary**: compute s_b, the distance to exit the current cell
3. **Compare distances**:
   - If $s < s_b$: nuclear event (collision) at distance s
   - If $s \geq s_b$: transmission event at boundary

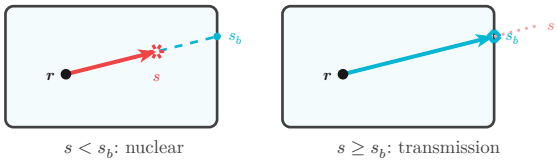


Figure 3: Ray casting: sampled nuclear distance s competes with boundary distance s_b. Here $s < s_b$, so a nuclear event occurs.

```
// Ray casting through material
let boundary_distances = cell.region.ray_cast(&rays);
let nuclear_distances = sample_nuclear_distances(&batch, &xs, rng);

let is_nuclear = nuclear_distances.lower_than(boundary_distances);
let event_distances = nuclear_distances.mask_where(is_nuclear, boundary_distances);
let event_origins = batch.origins() + batch.directions() * event_distances;
```

For nuclear events, the reaction type is sampled from cross section ratios:

$$P(\text{absorption}) = \frac{\Sigma_a}{\Sigma_t}, \quad P(\text{scattering}) = \frac{\Sigma_s}{\Sigma_t}, \quad P(\text{fission}) = \frac{\Sigma_f}{\Sigma_t} \tag{7}$$

Invariant: Every particle produces exactly one event. This 1:1 alignment simplifies bookkeeping and enables efficient parallel processing.

2.2.4 Progeny Sampling

At each event, progeny particles are sampled based on event type:

Event Type	Progeny Sampling
Source	Sample random direction and energy from source distribution
Scattering	Preserve energy; sample new isotropic direction
Fission	Sample ν neutrons from $\nu(E)$; each gets random direction and fission spectrum energy
Transmission	Push origin past boundary; preserve direction and energy
Absorption	No progeny (particle terminates)

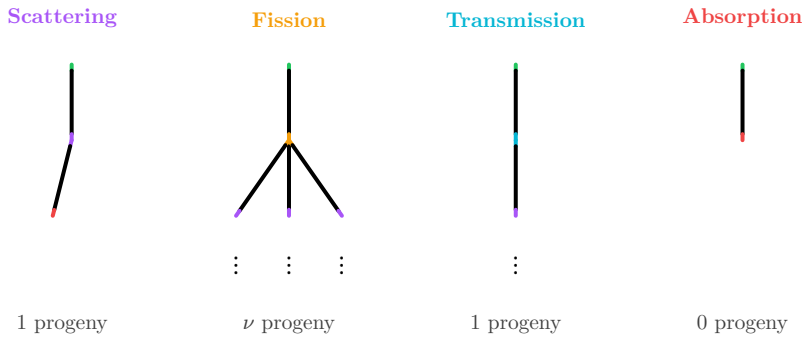


Figure 4: Progeny sampling by event type. Scattering and transmission produce one progeny each (with direction change or boundary crossing), fission produces ν progeny, and absorption terminates the particle.

Fission events require *replication*—a single event produces multiple progeny particles:

```
// Fission: replicate events for each progeny neutron
for (event_idx, nu) in nu_values.iter().enumerate() {
    for _ in 0..nu {
        replication_indices.push(event_idx);
    }
```

```
}
let progeny_origins = events.origins().select(replication_indices);
```

2.3 The Storage Backend

During simulation, particle and event data flows from tensor objects back to the CPU and to a persistent storage backend.

2.3.1 Data Flow

```
// Simulation phase (tensors, on device)
ParticleBatch<B> / EventBatch<B>
        ↓ extract to CPU, async
// Storage phase (Polars DataFrames)
Backend {
    TableStorage  // events & particles DataFrames
    GraphStorage  // particle lineage graph
}
        ↓ query
// Query phase
TransportData → Observable::measure()
```

2.3.2 TableStorage

The `TableStorage` module stores events and particles in columnar tables (Polars DataFrames).

Events DataFrame schema:

Column	Type	Description
`event_id`	`u64`	Unique identifier
`transport_id`	`String`	Simulation run ID
`depth`	`u64`	Depth in the stochastic process, a.k.a generation
`event_type`	`String`	Source, Fission, Scattering, Transmission, Absorption
`cell_id`	`u64`	Geometry cell
`origin_x/y/z`	`f32`	Position (cm)
`time`	`f32`	Timestamp (ns)

Particles DataFrame schema:

Column	Type	Description
`particle_id`	`u64`	Unique identifier
`particle_type`	`String`	Neutron, Photon
`origin_x/y/z`	`f32`	Birth position (cm)
`direction_x/y/z`	`f32`	Unit direction vector
`energy`	`f32`	Energy (MeV)
`origin_event_id`	`u64`	Event that created this particle
`caused_event_id`	`u64`	Event this particle caused

2.3.3 GraphStorage

The `GraphStorage` module maintains a directed graph of particle lineage using `petgraph`:

- **Vertices**: Events (indexed by `EventId`)
- **Edges**: Particles (connecting parent event → child event)

This enables efficient genealogical queries (parent-child relationships, ancestry tracing, branching factors, etc.) that would be expensive on tabular data alone.

2.4 Observables

The transport simulation produces genealogies $\mathbb{G}^i(d) = (\mathbb{E}^i(d), \mathbb{P}^i(d))$ encoding the complete history of particle interactions. Extracting physical measurements from this data requires defining **observables** (also called **tallies** in Monte Carlo literature).

2.4.1 The Observable Integral

An observable estimates an integral over phase space. The general form is:

$$T = \int_{\mathcal{D}} R(\boldsymbol{x})\psi(\boldsymbol{x})d\boldsymbol{x} \tag{8}$$

where:

- $\boldsymbol{x} = \left(\boldsymbol{r}, \hat{\Omega}, E, t\right)$ is the phase-space coordinate (position, direction, energy, time)
- $\psi(\boldsymbol{x})$ is the **angular flux** (particle density in phase space)
- $R(\boldsymbol{x})$ is the **response function** (what we measure at each point)
- $\mathcal{D}$ is the **integration domain** (phase-space region of interest)

2.4.1.1 Tensor-Valued Response Functions

The response function R can produce outputs of arbitrary tensor rank:

Rank	Integral	Example
0 (Scalar)	$T = \int_{\mathcal{D}} R\psi d\boldsymbol{x}$	Total flux, multiplication factor k
1 (Vector)	$T^i = \int_{\mathcal{D}} R^i\psi d\boldsymbol{x}$	Cell flux: $R^c = \chi_c(\boldsymbol{r})$ yields T^c
2 (Matrix)	$T^{ij} = \int_{\mathcal{D}} R^{ij}\psi d\boldsymbol{x}$	Energy-cell: $R^{cg} = \chi_c(\boldsymbol{r})\chi_g(E)$
3+ (Tensor)	$T^{ijk} = \int_{\mathcal{D}} R^{ijk}\psi d\boldsymbol{x}$	Spatial mesh: $R^{ijk} = \chi_{ijk}(\boldsymbol{r})$

The tensor rank of R determines the shape of the measurement output.

2.4.2 The Observable Pipeline

In `phlux`, the observable integral decomposes into two components:

Tally Component	phlux Concept	Mathematical Role
$\mathcal{D}$	`Filter`	Restricts the integration domain
$R^{i_1\cdots i_n}(\boldsymbol{x})$	`Response<T>`	Response function with output shape `T`

An `Observable` composes these two components:

1. `Filter`: Selects which events/particles contribute (defines $\mathcal{D}$)
2. `Response`: Maps filtered data to a measurement value (defines R with its tensor rank)

The key insight is that the response function $R^{i_1\cdots i_n}$ defines both what is measured and the structure of the data returned to the querier.

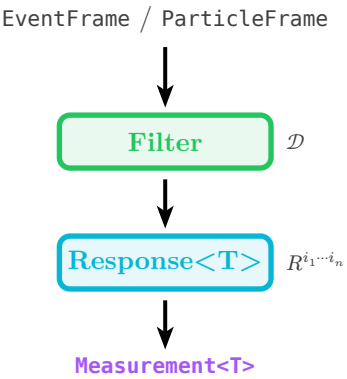


Figure 5: The observable pipeline: data flows through Filter and Response stages to produce a measurement with uncertainty.

2.4.3 Measurements

A `Measurement<T>` is a physical quantity with statistical uncertainty derived from the number of counts included in the `Filter`:

$$m = (\mu, \sigma) \tag{9}$$

where μ is the measured value and σ is the standard deviation.

Again, this concept extends to arbitrary tensors

$$m^{ijk} = \left(\mu^{ijk}, \sigma^{ijk}\right) \tag{10}$$

In `phlux`, this is implemented as:

```rust
pub struct Measurement<T> {
    pub value: T,
    pub uncertainty: T,
}

pub type Scalar = Measurement<f64>;
pub type Vector = Measurement<Vec<f64>>;
```

Additional measurement types:

Type	Description
`Scalar`	Single value T with uncertainty σ
`Vector`	Indexed values T^i with per-element uncertainty σ^i
`Histogram`	Binned values with bin edges and per-bin uncertainty
`SpatialMesh`	3D voxel grid T^{ijk} with per-voxel uncertainty
`TimeSeries`	Time-indexed values $T(t)$ with per-timestep uncertainty

Each measurement type carries uncertainty, enabling proper error propagation in downstream calculations.

2.4.4 Filters

A `Filter` selects which events or particles contribute to the measurement. Mathematically, a filter creates a boolean mask and applies masked selection:

$$m^j = \left[P^j(\mathbb{E}^j)\right] \quad \text{(boolean mask)} \tag{11}$$

$$\mathbb{E}^i = \mathbb{E}^j\left[m^j\right] \tag{12}$$

Note the different indices (i vs j) indicating that some counts may have been filtered out.

Filters implement the `Filter` trait:

```rust
pub trait Filter: Send + Sync {
    fn apply_events(&self, frame: EventFrame) -> EventFrame {
        frame  // Default: no filtering
    }

    fn apply_particles(&self, frame: ParticleFrame) -> ParticleFrame {
        frame  // Default: no filtering
    }
}
```

Common filter implementations:

- `EventTypeFilter` — Filter events by type (Source, Scattering, Absorption, Fission, Transmission)
- `ParticleTypeFilter` — Filter particles by type (Neutron, Photon)
- `DepthFilter` — Filter events at specific collision depth
- `CellFilter` — Filter events in specific geometric cell
- `EnergyFilter` — Filter particles by energy range $[\text{min}, \text{max})$

2.4.5 Response Functions

A `Response` maps filtered data to a measurement, encoding both the scoring logic and the output structure. The `Response` trait mirrors the `Filter` pattern:

```rust
pub trait Response: Send + Sync {
    type Output;
}

pub trait ResponseEvents: Response {
    fn respond(&self, frame: EventFrame) -> TransportResult<Self::Output>;
}

pub trait ResponseParticles: Response {
    fn respond(&self, frame: ParticleFrame) -> TransportResult<Self::Output>;
}
```

The response function $R^{i_1 \cdots i_n}$ determines both what each event contributes and the shape of the output. The tensor rank of R maps directly to `Output`:

Response	Output	Tensor Rank	Description
`Count`	`Scalar`	$R(\boldsymbol{x}) = 1$	Count events with Poisson uncertainty
`KEffective`	`Scalar`	$R = \frac{N_{d+1}}{N_d}$	Multiplication factor (ratio)
`CellFlux`	`Vector`	$R^c(\boldsymbol{x}) = \chi_c(\boldsymbol{r})$	Per-cell event counts
`EnergySpectrum`	`Histogram`	$R^g(\boldsymbol{x}) = \chi_g(E)$	Binned energy distribution
`SpatialFlux`	`SpatialMesh`	$R^{ijk}(\boldsymbol{x}) = \chi_{ijk}(\boldsymbol{r})$	3D voxel flux

Uncertainty is Poisson-derived for all counting responses ($\sigma = \sqrt{N}$). For ratio responses like k-effective:

$$\sigma_k = k\sqrt{\frac{1}{N_0} + \frac{1}{N_1}} \tag{13}$$

2.4.6 The Frame API

Observables query data through *frames*, lazy wrappers around the `TableStorage` backend that provide a fluent filtering API.

> **DataFrames**
>
> Both `ParticleFrame` and `EventFrame` operate over `LazyFrames`. This allows Polars to J-I-T optimize the query operations for us.

2.4.6.1 EventFrame

`EventFrame` provides methods for filtering and querying events:

```
let frame = data.events()
    .with_type(EventType::Fission)    // Filter by event type
    .at_depth(5)                       // Filter by depth
    .in_cell(cell_id)                  // Filter by geometry cell
    .in_time_range(0.0, 100.0);        // Filter by time window

let count = frame.count();             // Terminal: count events
let events = frame.collect();          // Terminal: collect Vec<Event>
let df = frame.dataframe();            // Terminal: get Polars DataFrame
```

Filter methods:

- `.with_type(EventType)` / `.exclude_type(EventType)` — filter by event type
- `.at_depth(u64)` — filter by generation depth
- `.in_cell(CellId)` — filter by geometry cell
- `.in_bbox(BoundingBox)` — filter by spatial bounds
- `.in_time_range(f32, f32)` — filter by time window

2.4.6.2 ParticleFrame

`ParticleFrame` provides similar methods for particles:

```
let frame = data.particles()
    .with_type(ParticleType::Neutron)  // Filter by particle type
    .energy_between(0.1, 10.0);        // Filter by energy range (MeV)

let count = frame.count();
let particles = frame.collect();
```

Filter methods:

- `.with_type(ParticleType)` — filter by particle type (Neutron, Photon)
- `.energy_between(f32, f32)` — filter by energy range
- `.from_event(EventId)` — particles originating from specific event

2.4.7 Canonical Observables

This section illustrates canonical `Observable` implementers in `phlux.rs`.

2.4.7.1 Count (Scalar)

The simplest observable counts events within a domain $\mathcal{D}$:

$$T = \int_{\mathcal{D}} R(\boldsymbol{x})\psi(\boldsymbol{x})d\boldsymbol{x} \quad \text{where} \quad R(\boldsymbol{x}) = 1 \tag{14}$$

With Poisson uncertainty $\sigma = \sqrt{N}$.

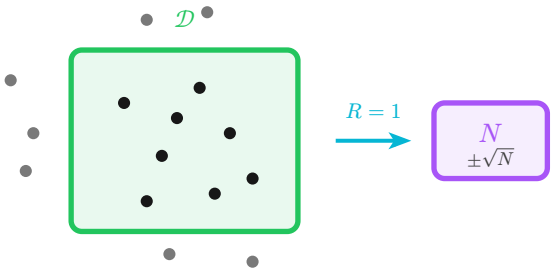


Figure 6: Count response: events within domain $\mathcal{D}$ produce a scalar $N \pm \sqrt{N}$.

```
// Count via Frame API
let n = data.events()
    .with_type(EventType::Fission)
    .at_depth(5)
    .count();
let uncertainty = (n as f64).sqrt();
println!("N = {} ± {:.2}", n, uncertainty);
```

2.4.7.2 K-Effective (Scalar Ratio)

The multiplication factor $k(d)$ measures the branching ratio between consecutive depths:

$$k(d) = \frac{N(d+1)}{N(d)} = \frac{|\mathbb{E}^i(d+1)|}{|\mathbb{E}^j(d)|} \tag{15}$$

With ratio error propagation:

$$\sigma_k = k\sqrt{\frac{1}{N_0} + \frac{1}{N_1}} \tag{16}$$

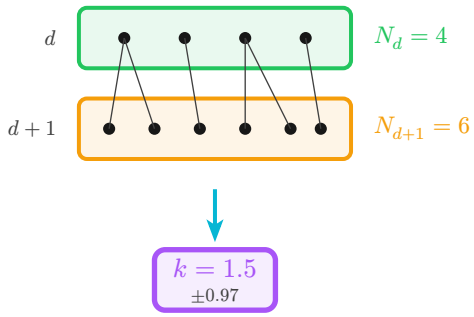


Figure 7: K-effective response: ratio of event counts at consecutive depths d and $d+1$.

Interpretation:

- $k(d) < 1$: Subcritical — population decreases with depth
- $k(d) = 1$: Critical — population remains constant
- $k(d) > 1$: Supercritical — population grows with depth

```
let keff = KEffective::at_depth(5);
let result: Scalar = data.measure(&keff)?;
println!("k = {} ± {}", result.value, result.uncertainty);
```

2.4.7.3 Cell Flux (Vector)

Cell flux counts events per geometric cell, producing a vector output:

$$T^c = \int_{\mathcal{D}} R^c(\boldsymbol{x})\psi(\boldsymbol{x})d\boldsymbol{x} \quad \text{where} \quad R^c(\boldsymbol{x}) = \chi_c(\boldsymbol{r}) \tag{17}$$

The indicator function $\chi_c(\boldsymbol{r}) = 1$ if $\boldsymbol{r} \in \text{cell}_c$, else 0. Per-cell Poisson uncertainty:

$$\sigma_c = \sqrt{T^c} \tag{18}$$

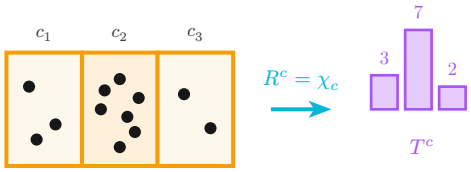


Figure 8: Cell flux response: events are binned by geometric cell, producing per-cell counts T^c.

```rust
let flux = CellFlux::from_cells(&geometry.cells());
let result: Vector = data.measure(&flux)?;

for (cell_id, (value, uncertainty)) in result.iter().enumerate() {
    println!("Cell {}: {} ± {}", cell_id, value, uncertainty);
}
```

2.4.7.4 Energy Spectrum (Histogram)

Energy spectrum bins particles by energy, producing a histogram:

$$T^g = \int_{\mathcal{D}} R^g(\boldsymbol{x})\psi(\boldsymbol{x})d\boldsymbol{x} \quad \text{where} \quad R^g(\boldsymbol{x}) = \chi_g(E) \tag{19}$$

The indicator function $\chi_g(E) = 1$ if $E \in \left[E_g, E_{g+1}\right)$, else 0. Per-bin Poisson uncertainty.

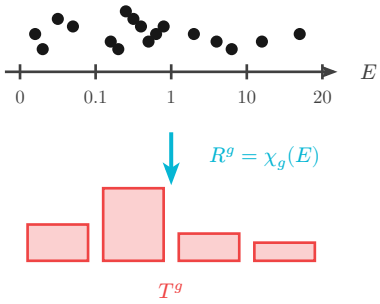


Figure 9: Energy spectrum response: particles are binned by energy, producing histogram counts T^g.

```rust
let spectrum = EnergySpectrum::with_bins(vec![0.0, 0.1, 1.0, 10.0, 20.0]);
let result: Histogram = data.measure(&spectrum)?;

for (bin, (value, uncertainty)) in result.iter().enumerate() {
    println!("Bin {}: {} ± {}", bin, value, uncertainty);
}
```

2.4.7.5 Spatial Flux (SpatialMesh)

Spatial flux bins events into a 3D voxel grid:

$$T^{ijk} = \int_{\mathcal{D}} R^{ijk}(\boldsymbol{x})\psi(\boldsymbol{x})d\boldsymbol{x} \quad \text{where} \quad R^{ijk}(\boldsymbol{x}) = \chi_{ijk}(\boldsymbol{r}) \tag{20}$$

The indicator function $\chi_{ijk}(\boldsymbol{r}) = 1$ if $\boldsymbol{r}$ is in voxel (i, j, k), else 0. Per-voxel Poisson uncertainty.

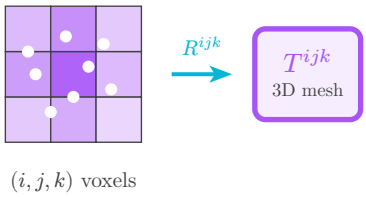


Figure 10: Spatial flux response: events are binned into 3D voxels, producing a mesh T^{ijk}.

```rust
let mesh = SpatialFlux::new(bounds, [10, 10, 10]);
let result: SpatialMesh = data.measure(&mesh)?;

for ((i, j, k), (value, uncertainty)) in result.iter() {
    println!("Voxel ({},{},{}): {} ± {}", i, j, k, value, uncertainty);
}
```

2.5 Transport as a Genealogy

The genealogy $\mathbb{G}^i(d) = (\mathbb{E}^i(d), \mathbb{P}^i(d))$ can be represented as a directed graph $G(E, P)$, where:

- **Vertices** correspond to events (rows in $\mathbb{E}^i(d)$)
- **Edges** correspond to particles (rows in $\mathbb{P}^i(d)$)

This forms a forest of trees, each rooted at a source event and extending until all progeny are terminated. The depth d of an event is the number of edges traversed from the root.

This graph representation enables:

1. Genealogical queries (parent-child relationships)
2. Criticality analysis (branching factors)
3. Path-based tallies (event sequences)
4. Lineage tracking (contribution from specific sources)

2.5.1 A Genealogical View on Criticality

The multiplication factor $k(d)$ is the average branching factor of the forest at depth d:

$$\begin{aligned} k(d) &= \mathbb{E}_{e \in E(d)}[b(e)] \\ &= \frac{\sum_{e \in E(d)} b(e)}{|E(d)|} \end{aligned} \tag{21}$$

where $b(e)$ is the outgoing degree of vertex e (number of progeny particles produced by the event).

The uncertainty in $k(d)$ follows the observable contract:

$$\Delta k(d)^2 = \frac{\sum_{e \in E(d)} (b(e) - k(d))^2}{|E(d)|} \tag{22}$$

3 Crater: Constructive Solid Geometry

`crater.rs` is a library for constructing and analyzing N-dimensional fields.

3.1 Scalar Fields

A **scalar field** is a function that assigns a number to each point in space:

$$f : \mathbb{R}^n \to \mathbb{R} \tag{23}$$

In `crater.rs`, types that implement `ScalarField<N>` behave like the class of functions above.

In one dimension, a scalar field is simply a function $f : \mathbb{R} \to \mathbb{R}$. Consider a simple quadratic:

$$f(x) = x^2 - r^2 \tag{24}$$

The parameter r defines a family of fields.

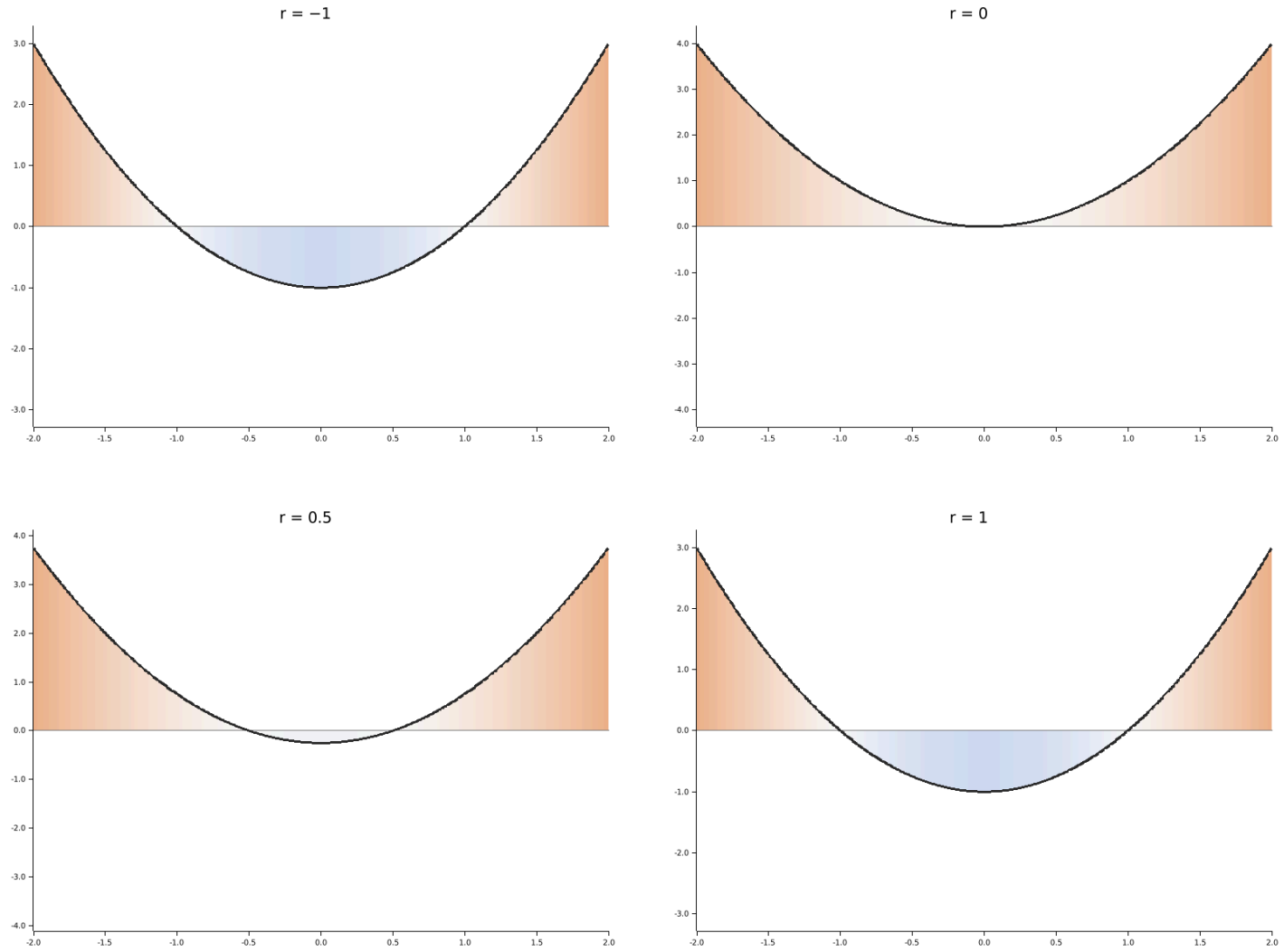


Figure 11: 1D scalar field $f(x) = x^2 - r^2$ for different values of r. Blue shading indicates $f < 0$ (inside), orange indicates $f > 0$ (outside).

In two dimensions, scalar fields become $f : \mathbb{R}^2 \to \mathbb{R}$. We can extend our earlier example naturally:

$$f(x, y) = x^2 + y^2 - r^2 \tag{25}$$

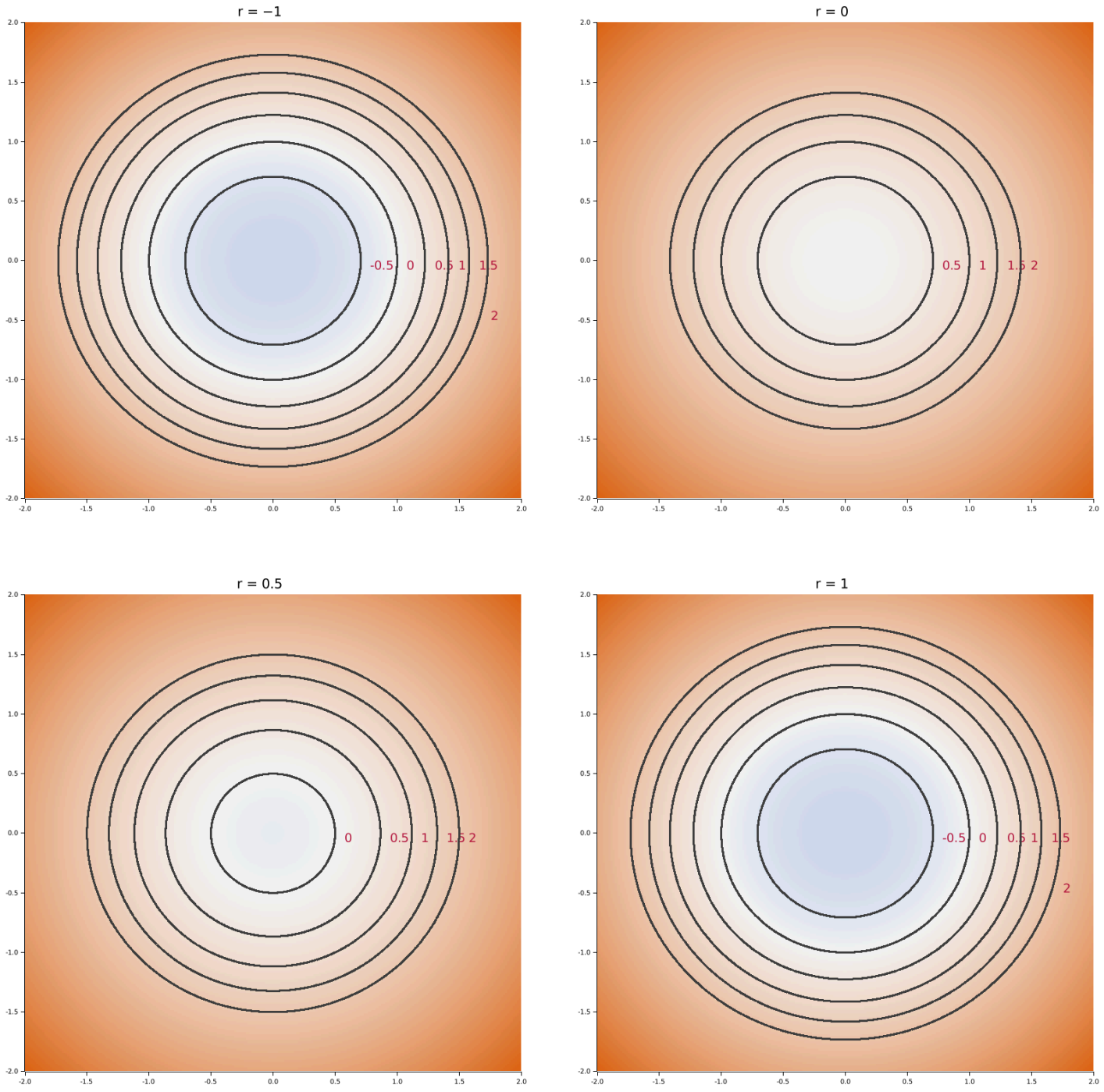


Figure 12: 2D scalar field $f(x, y) = x^2 + y^2 - r^2$ for different values of r. Isosurfaces are presented as dashed contours.

Isosurfaces

An **isosurface** is a locus of points where the scalar field is constant:

$$\partial f = \{x^j \in \mathbb{R}^n \mid f(x^j) = c\} \tag{26}$$

`crater.rs`'s `IsoSurface` type represents a pairing of a function $f : \mathbb{R}^n \to \mathbb{R}$ with a constant c.

Of course we are not restricted to just circular fields. Fields can be defined by any function $f : \mathbb{R}^n \to \mathbb{R}$.

```rust
/// f(x, y) = sin²(x) + cos²(y)
fn eval_trig(pts: Tensor<Backend, 2>) -> Tensor<Backend, 1> {
    let n = pts.dims()[0];
    let x = pts.clone().slice([0..n, 0..1]).squeeze::<1>();
    let y = pts.slice([0..n, 1..2]).squeeze::<1>();
    x.sin().powf_scalar(2.0) + y.cos().powf_scalar(2.0)
}

/// f(x, y) = x · y
fn eval_xy(pts: Tensor<Backend, 2>) -> Tensor<Backend, 1> {
    let n = pts.dims()[0];
    let x = pts.clone().slice([0..n, 0..1]).squeeze::<1>();
    let y = pts.slice([0..n, 1..2]).squeeze::<1>();
    x * y
}

/// f(x, y) = exp(-(x² + y²) / 10)
fn eval_gaussian(pts: Tensor<Backend, 2>) -> Tensor<Backend, 1> {
    let n = pts.dims()[0];
    let x = pts.clone().slice([0..n, 0..1]).squeeze::<1>();
    let y = pts.slice([0..n, 1..2]).squeeze::<1>();
    (x.clone() * x + y.clone() * y).div_scalar(2.0).neg().exp()
}

/// f(x, y) = sin(x * 4)
fn eval_sin_x(pts: Tensor<Backend, 2>) -> Tensor<Backend, 1> {
    let n = pts.dims()[0];
    let x = pts.slice([0..n, 0..1]).squeeze::<1>();
    x.mul_scalar(4).sin()
}
```

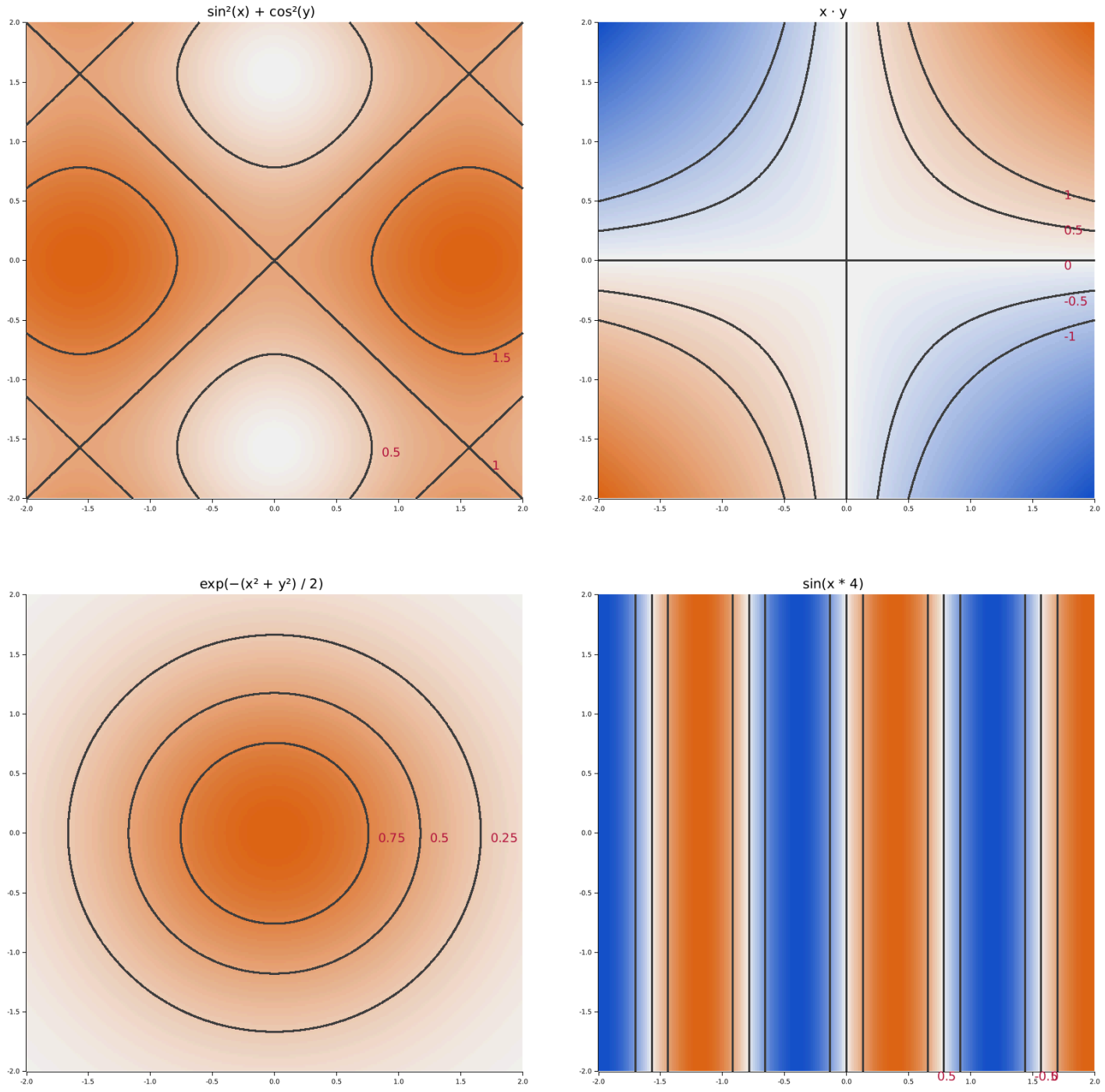


Figure 13: Exotic 2D scalar fields.

This abstraction is not limited to closed-form functions. Any programmable function can be used to define a field by implementing the `ScalarField` trait.

```
/// A Mandelbrot escape-time field.
///
/// For each point c = (x, y) in the complex plane, iterates
/// z_{n+1} = z_n² + c starting from z_0 = 0. Returns a normalized
/// escape time in [-1, 1], where -1 indicates points that never
/// escape (inside the set) and positive values indicate how
/// quickly the point escaped.
```

```rust
#[derive(Clone)]
struct MandelbrotField {
    max_iter: usize,
}

impl ScalarField<2, Backend> for MandelbrotField {
    fn evaluate(&self, origins: Tensor<Backend, 2>) -> Tensor<Backend, 1> {
        let n = origins.dims()[0];
        let device = origins.device();

        let cx = origins.clone().slice([0..n, 0..1]).squeeze::<1>();
        let cy = origins.slice([0..n, 1..2]).squeeze::<1>();

        let mut zx = Tensor::<Backend, 1>::zeros([n], &device);
        let mut zy = Tensor::<Backend, 1>::zeros([n], &device);
        let mut result = Tensor::<Backend, 1>::full([n], -1.0, &device);
        let mut escaped = Tensor::<Backend, 1>::zeros([n], &device);

        for iter in 0..self.max_iter {
            // z² = (zx + i·zy)² = (zx² - zy²) + i·(2·zx·zy)
            let zx2 = zx.clone() * zx.clone();
            let zy2 = zy.clone() * zy.clone();
            let new_zx = zx2.clone() - zy2.clone() + cx.clone();
            let new_zy = zx.clone() * zy * 2.0 + cy.clone();

            // Escape criterion: |z|² > 4
            let mag2 = zx2 + zy2;
            let just_escaped =
                mag2.greater_elem(4.0).float() * (escaped.clone().neg() + 1.0);

            // Record normalized iteration count for escaped points
            let iter_value = (iter as f32 / self.max_iter as f32) * 2.0 - 1.0;
            result = result.clone() * (just_escaped.clone().neg() + 1.0)
                + just_escaped.clone() * iter_value;
            escaped = escaped + just_escaped;

            zx = new_zx;
            zy = new_zy;
        }

        result
    }
}
```

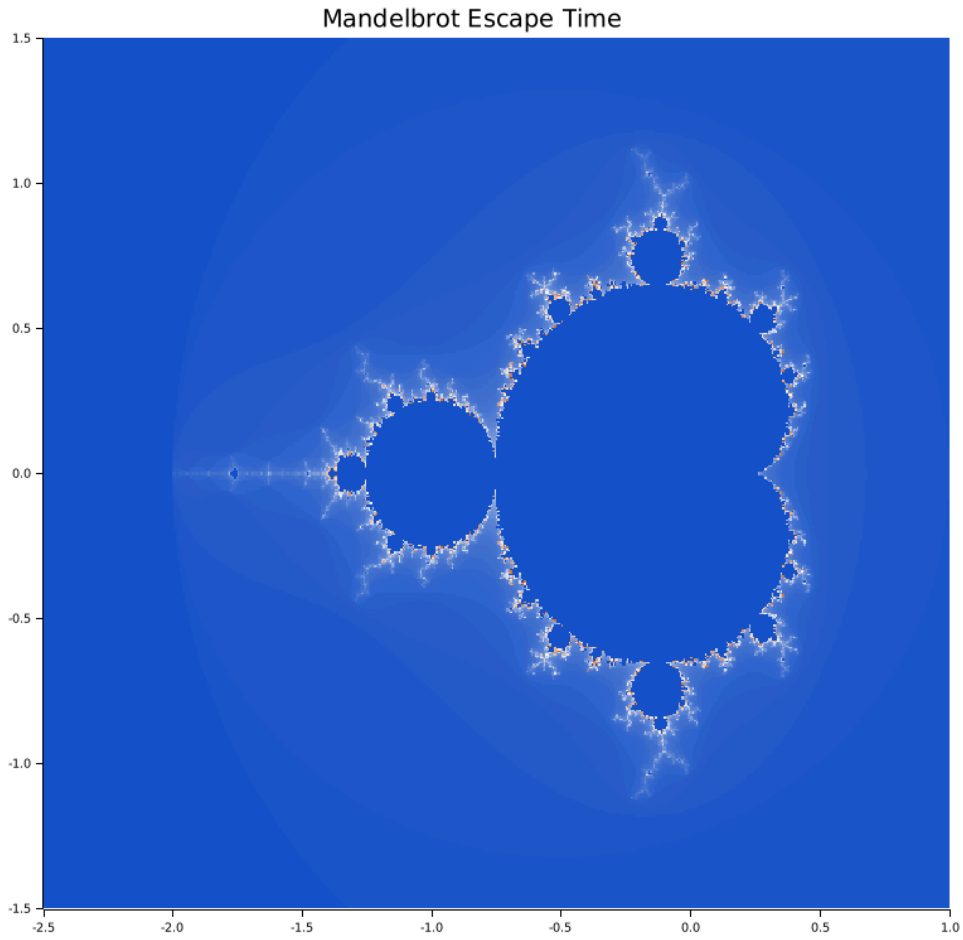


Figure 14: A field with no closed-form expression. This Mandelbrot field computes an iterative escape time at each point, generating a fractal pattern.

3.1.1 Batch Evaluation

Evaluation of scalar fields at points in $\mathbb{R}^n$ is an embarassingly parallel operation. Thus, we enable efficient batch evaluation by defining a tensor-valued function $F^i : \mathbb{R}^{m \times n} \to \mathbb{R}^m$:

$$F^i(X^{ij}) = v^i \tag{27}$$

where v^i is a rank-1 tensor with length m.

F^i satisfies the following two properties:

$$F^k\big(X^{kj}\big) = f(x^j) \quad \forall k \in [1, m] \text{ with } x^j = X^{kj} \tag{28}$$

and

$$\frac{\partial F^i}{\partial X^{kj}} = \delta^{ik} \frac{\partial F^i}{\partial X^{ij}} \tag{29}$$

The first property states that F^i is equivalent to evaluating f over each row of X^{ij} independently. The second property states the Jacobian is block diagonal—each output row is a function of only its corresponding input row. Colloquially, this means that all rows are evaluated independently.

For the remainder of this chapter, we will refer to scalar fields using the F^i notation to reinformce that our algorithms are operating on batches of input points.

3.2 Regions

A `Region` is the volume enclosed by an isosurface:

$$R_F = \{x^j \mid F^i(x^j) \leq 0\} \tag{30}$$

The simplest `Region` is a **halfspace**, which divides R^n into two sets: those point within the `Region` and those without it.

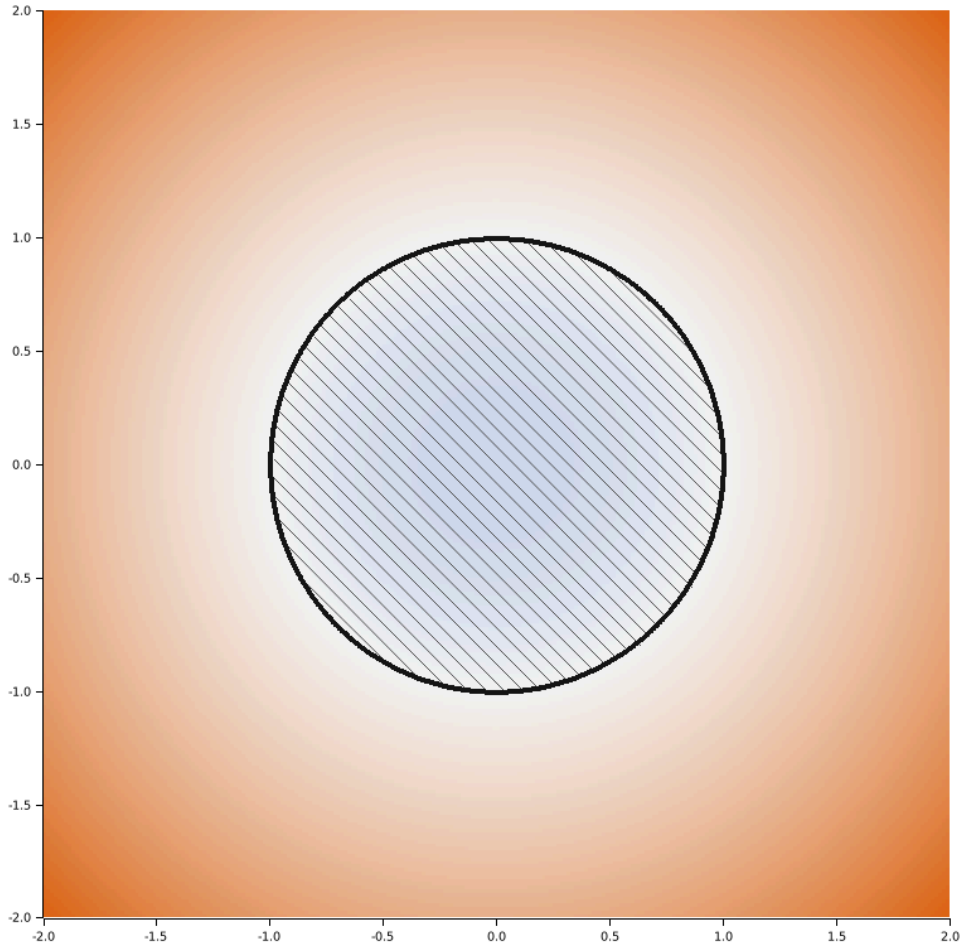


Figure 15: A circular `Region`. Hatching indicates the interior.

Contiguity

Depending on the behavior of F^i or isosurface value, a `Region` may not be contiguous:

```
// A field that produces non-contiguous regions: sin(πx) * sin(πy)
// This creates a checkerboard pattern where each cell is a separate region
let field = FnField::new(|pts: Tensor<Backend, 2>| {
    let n = pts.dims()[0];
    let x = pts.clone().slice([0..n, 0..1]).squeeze::<1>();
    let y = pts.slice([0..n, 1..2]).squeeze::<1>();
    (x * PI).sin() * (y * PI).sin()
});
```

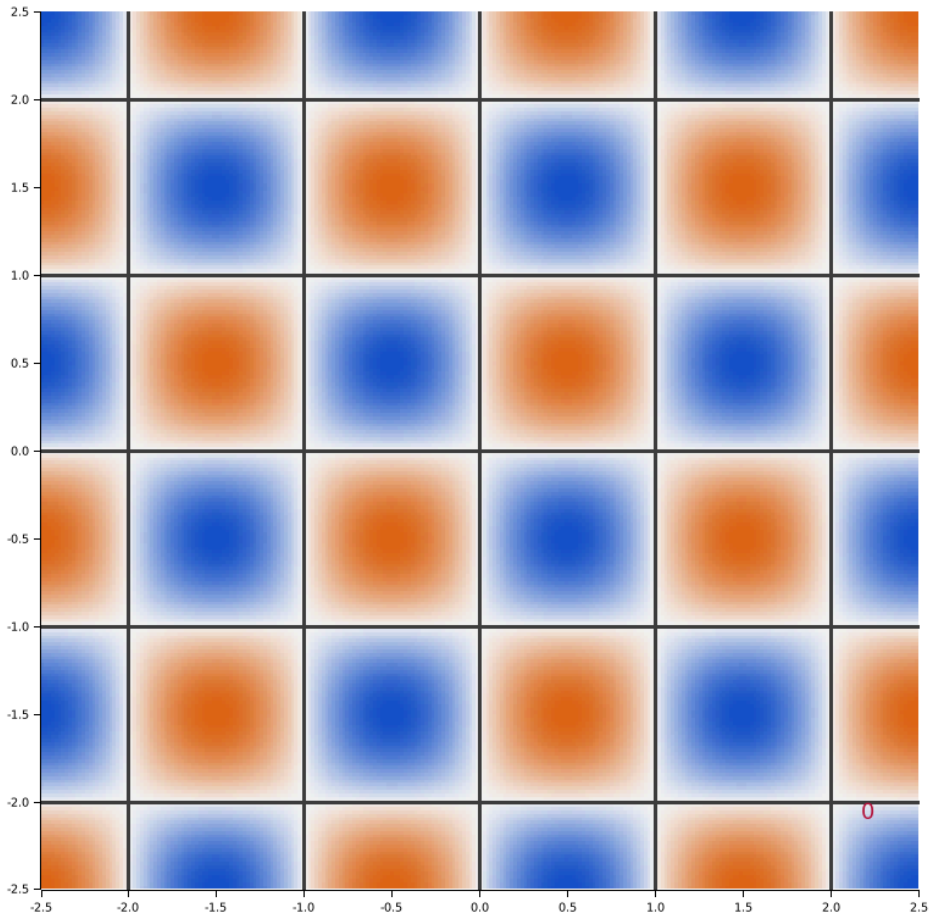


Figure 16: A non-contiguous `Region` from $f(x, y) = \sin(\pi x)\sin(\pi y)$. Every blue cell in the checkerboard pattern is a member of the same halfspace `Region`.

3.2.1 Constructive Solid Geometry

Constructive Solid Geometry (CSG) enables the creation of arbitrarily complex shapes through the algebraic combination of primitive `Region`s. A number of common primitives are implemented via the `FieldND<const N: usize>` enum. However, CSG can incorporate any `Region`, not just those made from our primitives.

CSG is defined by a collection of binary and unary algebraic operations. These operations bear resemblance to set-theoretic operations:

1. Union: $F^i \cup G^i = \min(F^i, G^i)$
2. Intersection: $F^i \cap G^i = \max(F^i, G^i)$
3. Complement: $\overline{F^i} = -F^i$

From this we derive the difference operation $F^i \setminus G^i = F^i \cap \overline{G^i}$

In direct set notation:

$$\begin{aligned} R_F \cup R_G &= \{x^j \mid F^i(x^j) \leq 0 \text{ or } G^i(x^j) \leq 0\} \\ R_F \cap R_G &= \{x^j \mid F^i(x^j) \leq 0 \text{ and } G^i(x^j) \leq 0\} \\ R_F \setminus R_G &= \{x^j \mid F^i(x^j) \leq 0 \text{ and } G^i(x^j) > 0\} \end{aligned} \tag{31}$$

Isosurface thickness

In practice, testing $F^i(x^j) = 0^i$ exactly is numerically fragile. Numerical methods are sensitive to the finite precision of the floating-point representation of the real numbers.

Consider a 1D field $f(x)$ where $f(x_1) < 0$, $f(x_2) > 0$ and x_1 and x_2 are adjacent representable floating point numbers (i.e., they differ by their least significant bit, only). In this case, there exists no x_* such that $f(x_*) = 0$ exactly and $x_1 < x_* < x_2$, without increasing our floating point precision.

To combat this, we use an **epsilon tolerance** ε to classify points into three categories:

- **Inside**: $f(x) < -\varepsilon$
- **On**: $|f(x)| \leq \varepsilon$
- **Outside**: $f(x) > \varepsilon$

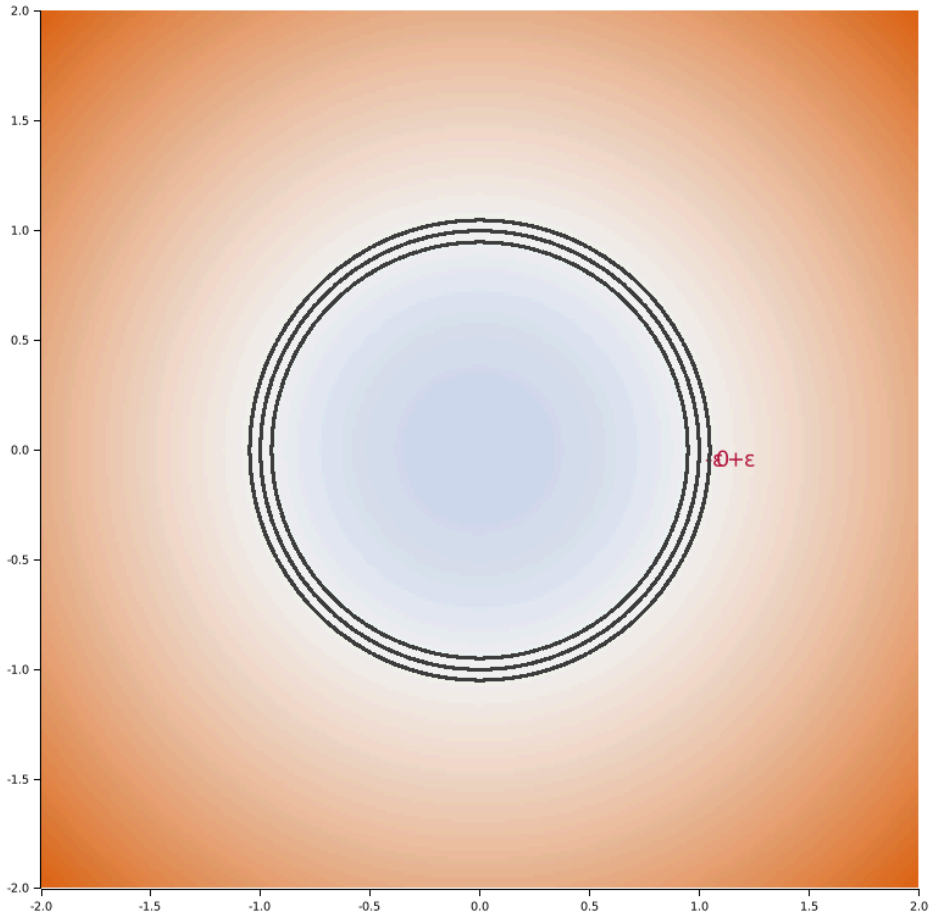


Figure 17: Classification bands around the zero isosurface of a circular field in 2D. The dashed contours show $f = -\varepsilon$ and $f = +\varepsilon$. Points between them are classified as "on" the surface.

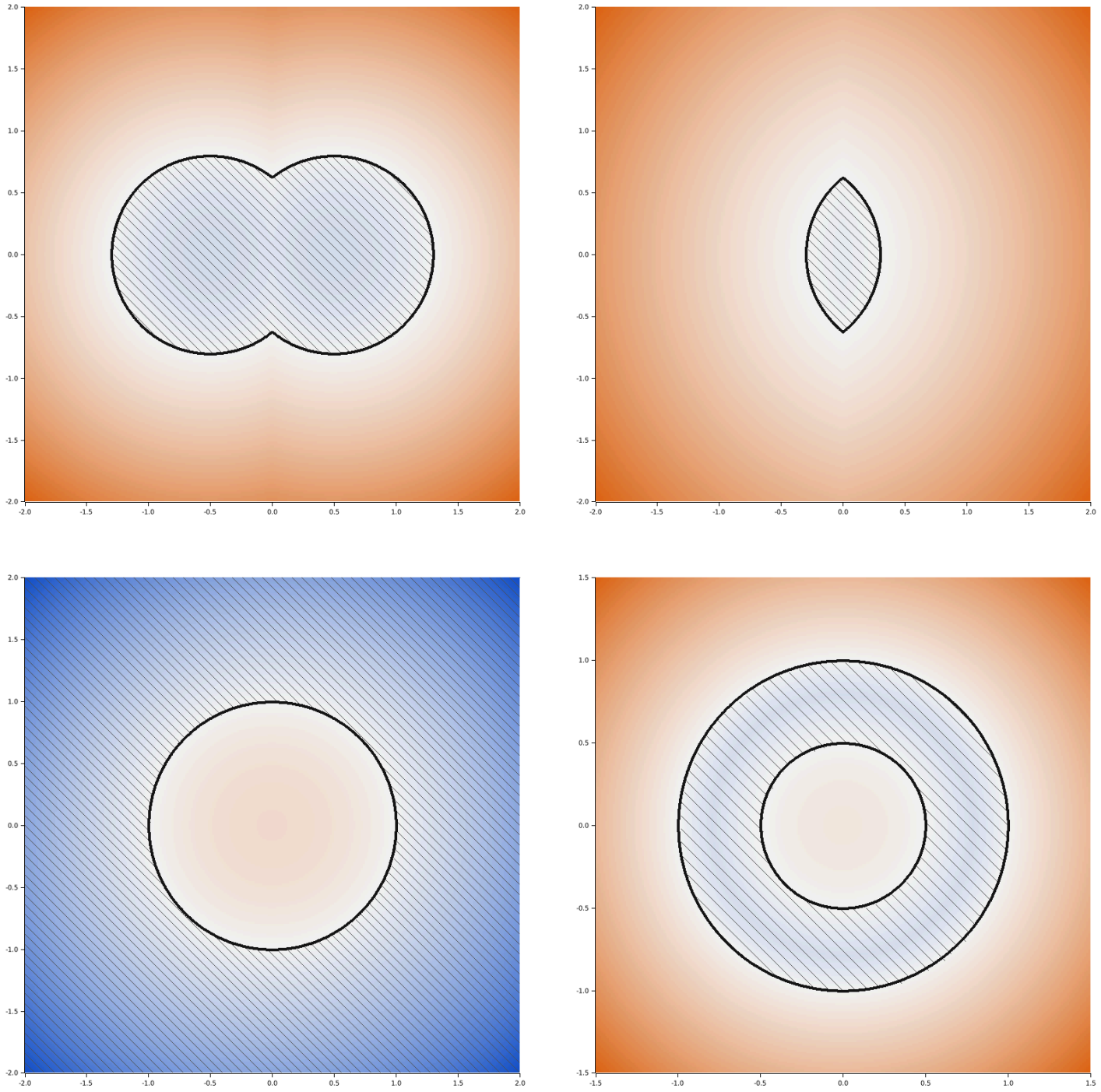


Figure 18: CSG operations: union (top-left), intersection (top-right), complement (bottom-left), difference (bottom-right).

These operations can be composed to create increasingly complex `Region`s. `Region` itself is an enum that behaves like a tree node:

```
// pub enum Region<const N: usize, B: Backend> {
//     HalfSpace(Isosurface<N, B>, Side),
//     Union(Box<Region<N, B>>, Box<Region<N, B>>),
//     Intersection(Box<Region<N, B>>, Box<Region<N, B>>),
// }
```

By composing these operations, we create a tree of operations that concretely define the composite `Region`:

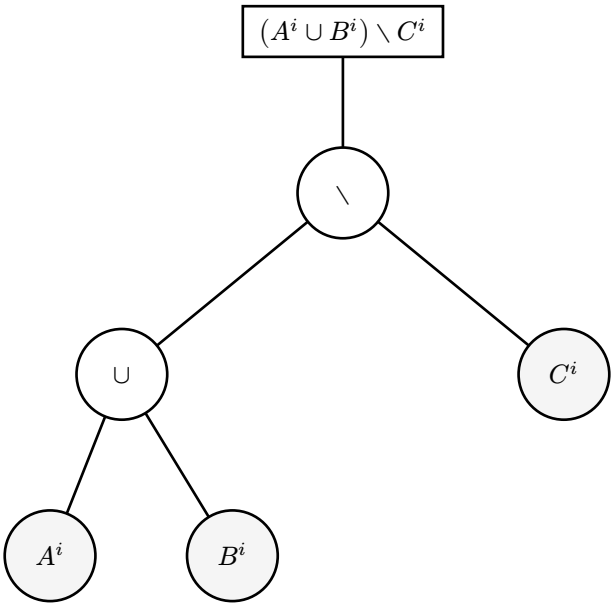


Figure 19: A CSG tree representing the functional composition $(A^i \cup B^i) \setminus C^i$. Leaf nodes are primitive `Region`s; internal nodes represent intermediate `Region`s.

These trees are unbounded in depth, enabling the procedural creation of complex `Region`s:

```rust
/// Constructs a 2D gear region with the specified number of teeth.
///
/// The gear is built from CSG primitives:
/// - A circular body (outer radius)
/// - A circular hole (inner radius)
/// - Teeth formed by intersecting pairs of halfspaces (lines)
///
/// Each tooth is a wedge created by intersecting two halfspaces
/// whose normals point inward, rotated around the gear center.
pub fn gear_2d<B: burn::tensor::backend::Backend>(
    outer_radius: f32,
    inner_radius: f32,
    num_teeth: usize,
    tooth_height: f32,
    tooth_width_fraction: f32, // fraction of tooth spacing (0..1)
) -> Region<2, B> {
    use std::f32::consts::PI;

    // Base body: outer circle minus inner hole
    let body: Region<2, B> = FieldND::circle(outer_radius).into_isosurface(0.0).region();
    let hole: Region<2, B> = FieldND::circle(inner_radius).into_isosurface(0.0).region();
    let mut gear = body.clone() & -hole;

    // Add teeth around the circumference
    let tooth_radius = outer_radius + tooth_height;
    let angle_per_tooth = 2.0 * PI / num_teeth as f32;
    let half_tooth_angle = angle_per_tooth * tooth_width_fraction * 0.5;

    for i in 0..num_teeth {
        let center_angle = i as f32 * angle_per_tooth;

        // Create a tooth as a wedge: intersection of two halfspaces
        // Left edge of tooth
        let left_angle = center_angle - half_tooth_angle;
        let left_normal = [left_angle.sin(), -left_angle.cos()];
        let left_plane: Region<2, B> =
            FieldND::line(left_normal).into_isosurface(0.0).region();
```

```
        // Right edge of tooth
        let right_angle = center_angle + half_tooth_angle;
        let right_normal = [-right_angle.sin(), right_angle.cos()];
        let right_plane: Region<2, B> =
            FieldND::line(right_normal).into_isosurface(0.0).region();

        // Tooth is the wedge between planes, bounded by outer and inner radii
        // The tooth only extends outward from the body (between outer_radius and tooth_radius)
        let outer_bound: Region<2, B> =
            FieldND::circle(tooth_radius).into_isosurface(0.0).region();
        let inner_bound: Region<2, B> = body.clone();
        let tooth = left_plane & right_plane & outer_bound & -inner_bound;

        gear = gear | tooth;
    }

    gear
}
```

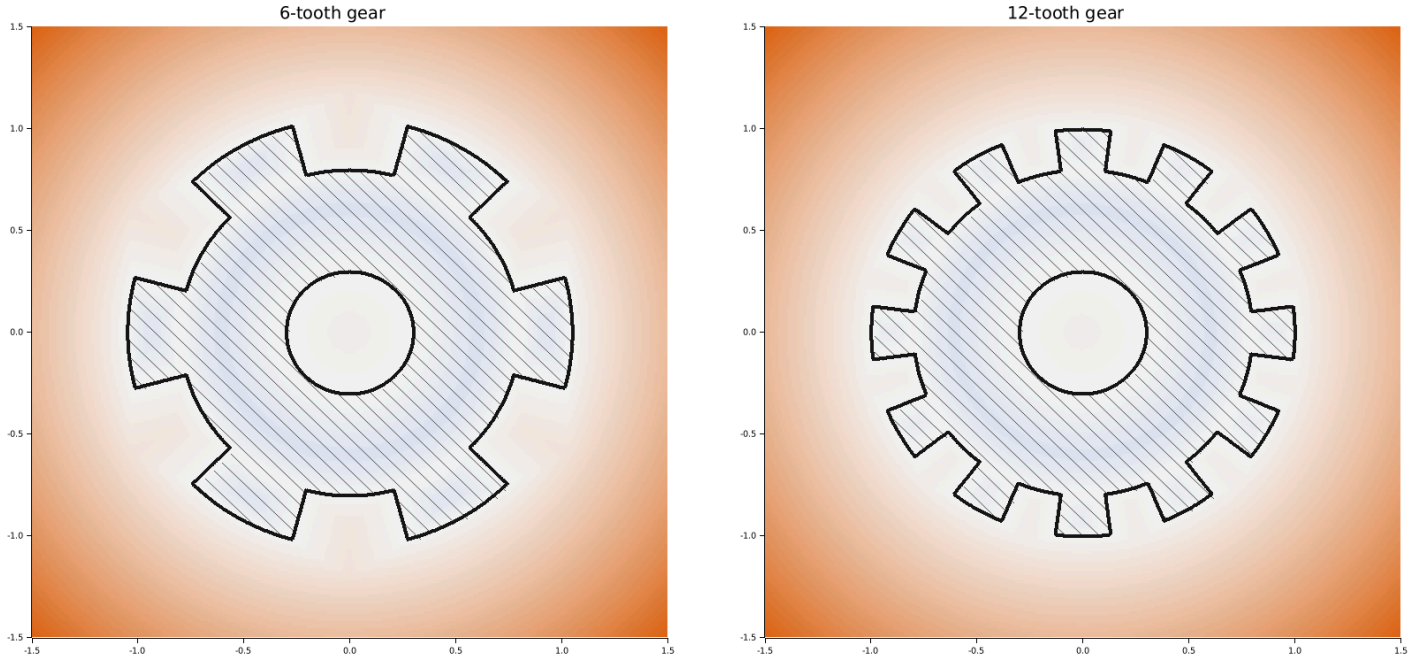


Figure 20: Procedurally generated gears with 6 and 12 teeth, built from circles and halfspaces.

3.2.1.1 Algebras

A `CSGAlgebra` is a concrete implementation of the CSG operations. In the mathematical sense:

$$\boldsymbol{A} = (P(D), \cup, \cap) \tag{32}$$

where $P(D)$ is the power set of domain D.

3.2.1.1.0.1 Differentiable Algebras

Differentiable approximations of CSG operations can be constructed:

$$\begin{aligned} F^i \cup G^i &= \frac{F^i + G^i + \sqrt{F^i F^i + G^i G^i - 2\alpha^i F^i G^i}}{1 + \alpha^i} \\ F^i \cap G^i &= \frac{F^i + G^i - \sqrt{F^i F^i + G^i G^i - 2\alpha^i F^i G^i}}{1 + \alpha^i} \end{aligned} \tag{33}$$

where α^i controls smoothness. $\alpha^i = 1$ reduces to the original min-max operations.

3.2.1.1.0.2 Blending

A custom blending function can be added to both $\cup$ and $\cap$ operators:

$$\varphi(F^i, G^i) = \frac{a_0}{1 + \left(\frac{F^i}{a_1}\right)^2 + \left(\frac{G^i}{a_2}\right)^2} \tag{34}$$

where a_0, a_1, a_2 are positive constants that control the shape of the blending function.

3.2.2 Primitives

`crater.rs` provides a number of dimension-generic primitive `Region`s via the `FieldND` interface.

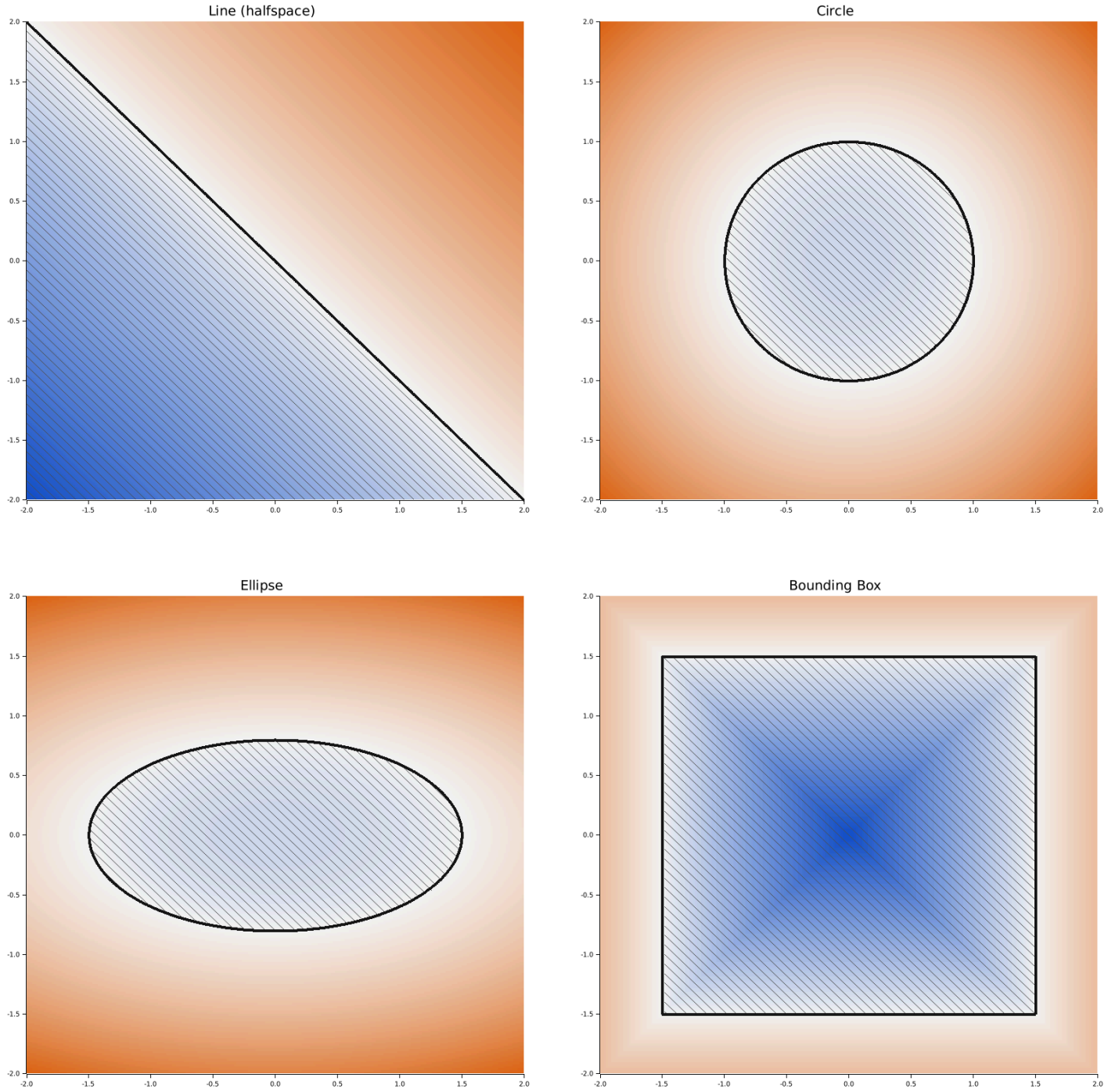


Figure 21: 2D primitives: line (halfspace), circle, ellipse, and bounding box.

3.3 Transformations

Until now, all fields we have discussed have been centered at the origin. We extend our framework to include arbitary **transformations** on our fields.

Any function $T^i : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ can perform covariant transformations on our fields without changing our definition of F^i:

$$F^i(X^{ij}) \mapsto F^i(T^i(X^{ij})) \tag{35}$$

In plain language, the transformation is on the **domain** of F^i, not its **image**.

`crater.rs` provides a number of transformations to use off-the-shelf via the `Transformation` enum.

3.3.1 Translation

`Transformation::Translate` implements a tensor-valued function $T^i : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$:

$$T^i(X^{ij}) = X^{ij} - t^j \tag{36}$$

where $t^j \in \mathbb{R}^n$ is the translation vector.

```rust
fn translate_before<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    FieldND::circle(0.8).into_isosurface(0.0).region()
}

fn translate_after<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    translate_before().transform(Transform::translate([1.0, 0.5]))
}
```

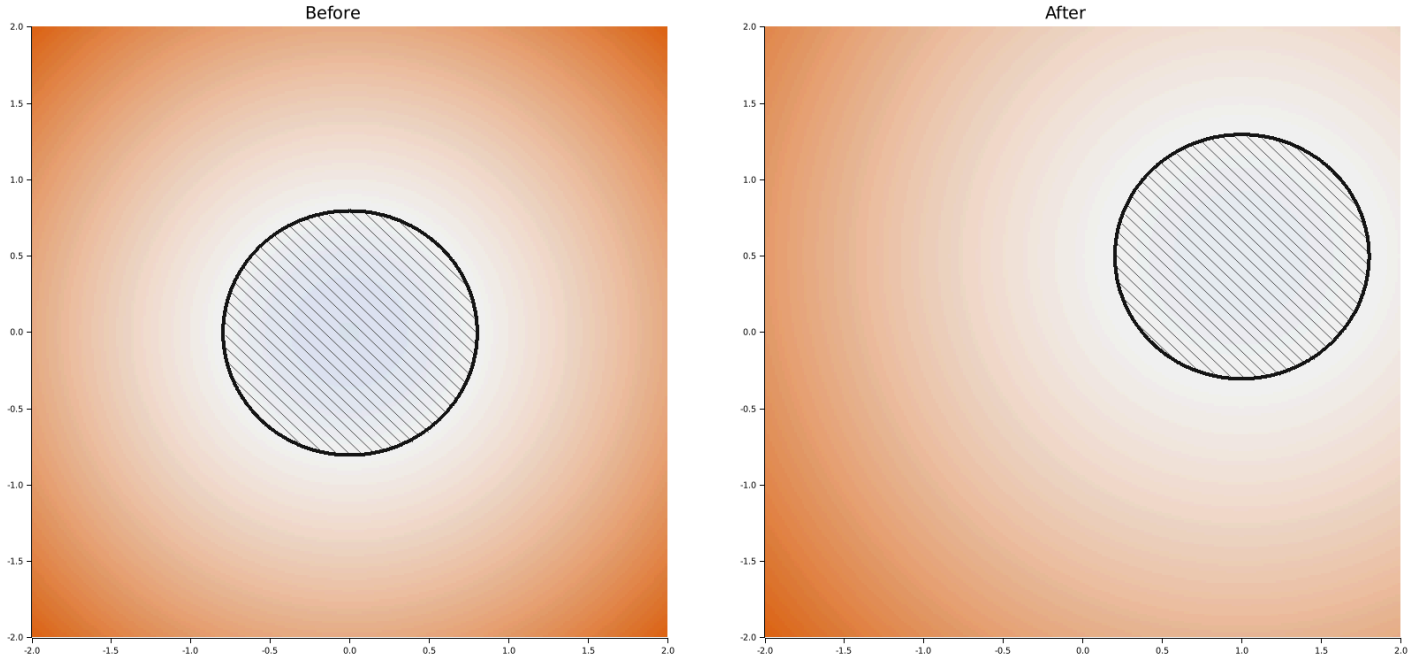


Figure 22: Left: original circle. Right: circle translated by $(1, 0.5)$.

3.3.2 Scaling

`Transformation::ScaleDim` implements a tensor-valued function $S^i : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$:

$$S^i(X^{ij}) = X^{ij} s_i \tag{37}$$

where $s_i \in \mathbb{R}$ is the scale factor for the ith dimesion, **only**.

```rust
fn scale_before<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    FieldND::circle(1.0).into_isosurface(0.0).region()
}

fn scale_after<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    scale_before()
        .transform(Transform::scale_dim(0, 1.5))
        .transform(Transform::scale_dim(1, 1.5))
}
```

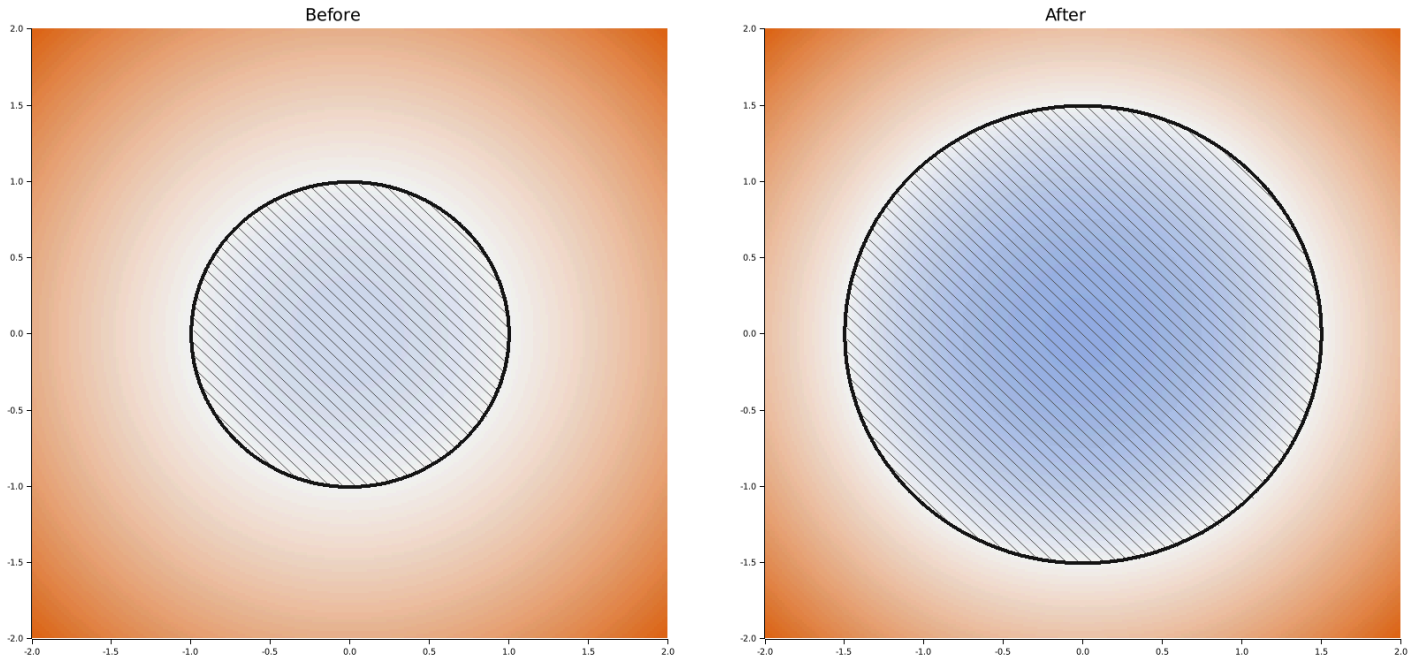


Figure 23: Left: unit circle. Right: scaled uniformly by factor 1.5, resulting in radius 1.5.

non-unitarity of scaling

The scaling transformations are not unitary operations. A circle of radius 1 scaled by factor 2 produces a *different field* than a circle constructed with radius 2, even though their boundaries coincide:

```rust
// Demonstrates that scaling is NOT a unitary operation on scalar fields.
// A circle(1) scaled by 2 produces a DIFFERENT field than circle(2).
// The boundary (isosurface at 0) is the same, but field values differ.

/// Circle of radius 1, scaled by factor 2 in both dimensions
fn scaled_circle<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    FieldND::circle(1.0)
        .into_isosurface(0.0)
        .region()
        .transform(Transform::scale_dim(0, 2.0))
        .transform(Transform::scale_dim(1, 2.0))
}

/// Circle of radius 2 (constructed directly)
fn direct_circle<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    FieldND::circle(2.0).into_isosurface(0.0).region()
}
```

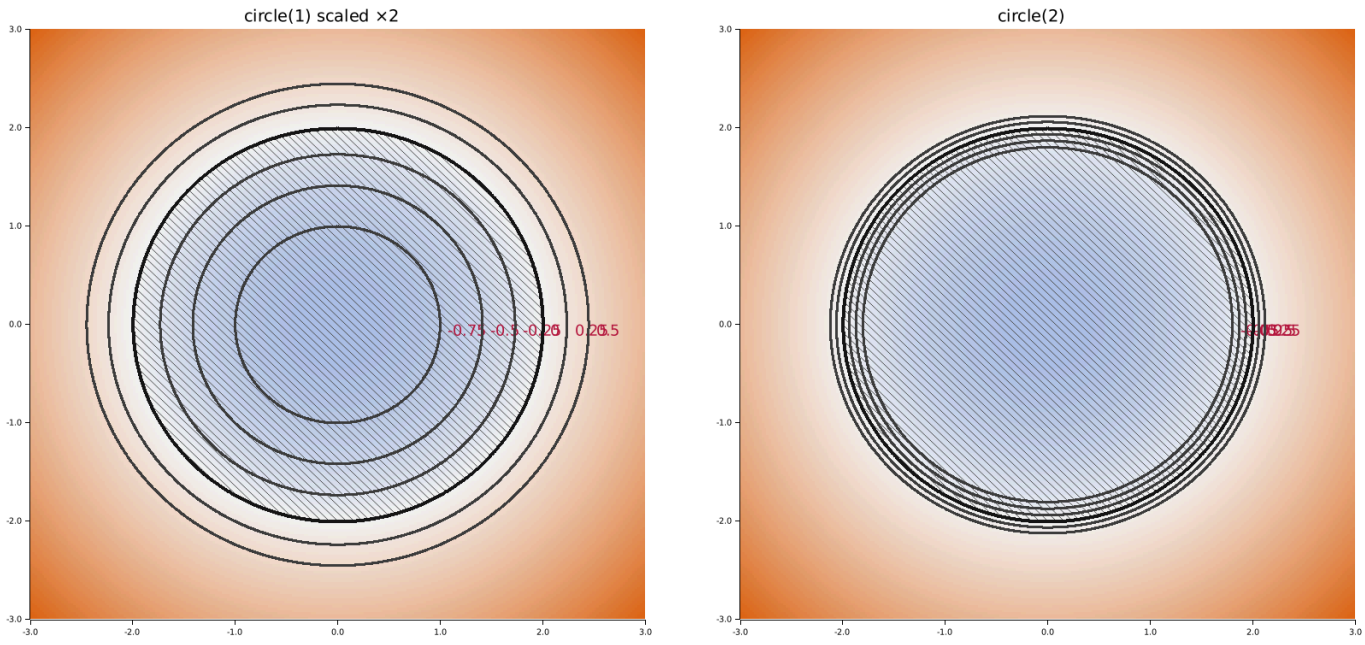


Figure 24: Left: `circle(1)` scaled by 2. Right: `circle(2)`. The boundaries match, but the field gradients differ.

3.3.3 Rotation

`Transformation::Rotate` N-dimensional rotation $R^i : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ with $R^i \in \mathrm{SO}(n)$ can be represented as a sequence of **plane rotations**.

A plane rotation is a rotation in a 2D plane spanned by coordinate axes a and b, parameterized by angle θ_{ab}. Call this $G(a, b, \theta_{ab})$.

The general N-dimensional rotation is a product of $\frac{n(n-1)}{2}$ plane rotations:

$$R^i = \prod_{a=2}^{n} \prod_{b=1}^{a-1} G(a, b, \theta_{ab}) \tag{38}$$

Intuitively, $G(a, b, 0)$ is the identity tensor.

```rust
fn rotate_before<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    FieldND::ellipse(1.2, 0.6).into_isosurface(0.0).region()
}

fn rotate_after<B: burn::tensor::backend::Backend>() -> Region<2, B> {
    rotate_before().transform(Transform::rotate(0, 1, std::f32::consts::FRAC_PI_4))
}
```

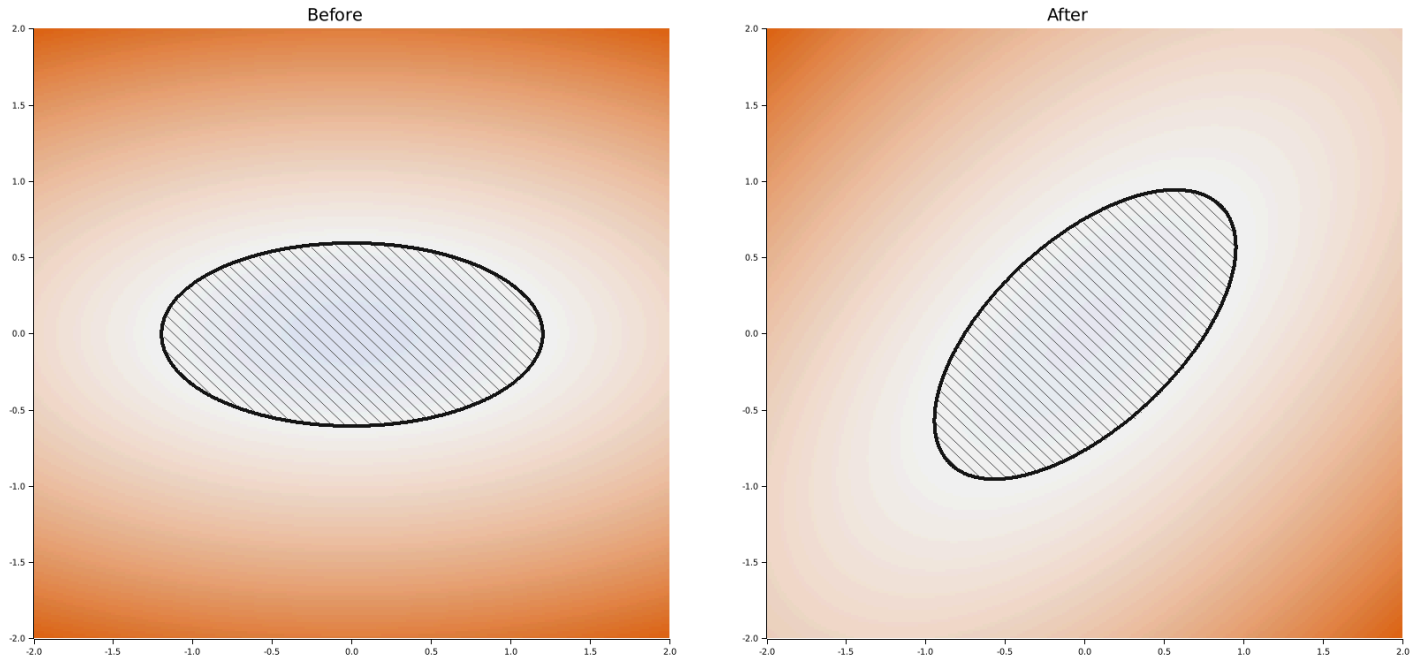


Figure 25: Left: original ellipse. Right: rotated by 45 deg in the xy-plane.

3.3.4 Non-standard Transformations

Transformations are not limited to those which are exposed by the `Transformation` enum. Any implementer of the `ApplyTransformation` trait can be used to covariantly tranform a field.

For example, the Poincaré disk model maps the hyperbolic plane $\mathbb{H}^2$ to the open disk $\mathbb{D}^2 = \left\{z \in \mathbb{C} : |z| < \frac{1}{\eta}\right\}$, compressing points far from the origin toward the boundary.

For a tensor of points $X^{ij} \in \mathbb{R}^{m \times n}$ with radial distance $r^i = \sqrt{X^{ij} X^i_j}$, the transformation $P^i : \mathbb{R}^{m \times n} \to \mathbb{D}^{m \times n}$ is:

$$P^i(X^{ij}) = X^{ij} \frac{\tanh(\eta r)}{\eta r} \tag{39}$$

where $\eta \in \mathbb{R}^+$ is the scale parameter controlling the disk radius $\frac{1}{\eta}$.

The inverse transformation $P^{-1} : \mathbb{D}^{m \times n} \to \mathbb{R}^{m \times n}$ used for covariant field evaluation is:

$$P^{-1}(X^{ij}) = X^{ij} \frac{\operatorname{arctanh}(\eta r)}{\eta r} \tag{40}$$

Points at $r^i \to \infty$ in hyperbolic space map to the disk boundary $r'^i \to \frac{1}{\eta}$.

```rust
use crater::analysis::prelude::RayField;
use crater::csg::prelude::ApplyTransformation;
use crater::csg::transformations::Covariant;

/// A Poincaré disk transformation that maps the infinite plane
/// into a bounded disk. Points far from the origin are compressed
/// toward the boundary.
///
/// Uses the formula: r' = tanh(r * scale) / scale
/// where r is the distance from the origin.
#[derive(Clone)]
pub struct PoincareDisk {
    pub scale: f32,
}
```

```rust
impl<B: burn::tensor::backend::Backend> ApplyTransformation<2, B> for PoincareDisk {
    fn apply_scalar_field(
        self,
        original: Box<dyn ScalarField<2, B>>,
    ) -> Box<dyn ScalarField<2, B>> {
        let scale = self.scale;
        Box::new(Covariant::ScalarField {
            field: original,
            // Forward: disk coords → hyperbolic coords (inverse of projection)
            transform: Box::new(move |points: Tensor<B, 2>| {
                let n = points.dims()[0];
                let x = points.clone().slice([0..n, 0..1]);
                let y = points.clone().slice([0..n, 1..2]);

                // r = sqrt(x² + y²)
                let r = (x.clone() * x.clone() + y.clone() * y.clone()).sqrt();
                let r_safe = r.clone().clamp_min(1e-8);

                // Inverse of tanh: arctanh(r * scale) / scale
                // arctanh(x) = 0.5 * ln((1+x)/(1-x))
                let rs = (r_safe.clone() * scale).clamp(-0.999, 0.999);
                let r_hyp =
                    ((rs.clone() + 1.0).log() - (-rs + 1.0).log()).mul_scalar(0.5 / scale);

                // Scale factor: r_hyp / r
                let factor = r_hyp / r_safe;
                let new_x = x * factor.clone();
                let new_y = y * factor;
                Tensor::cat(vec![new_x, new_y], 1)
            }),
            // Inverse: hyperbolic coords → disk coords
            inverse_transform: Box::new(move |points: Tensor<B, 2>| {
                let n = points.dims()[0];
                let x = points.clone().slice([0..n, 0..1]);
                let y = points.clone().slice([0..n, 1..2]);

                let r = (x.clone() * x.clone() + y.clone() * y.clone()).sqrt();
                let r_safe = r.clone().clamp_min(1e-8);

                // r' = tanh(r * scale) / scale
                let r_disk = (r_safe.clone() * scale).tanh() / scale;

                let factor = r_disk / r_safe;
                let new_x = x * factor.clone();
                let new_y = y * factor;
                Tensor::cat(vec![new_x, new_y], 1)
            }),
        })
    }

    fn apply_ray_field(self, original: Box<dyn RayField<2, B>>) -> Box<dyn RayField<2, B>> {
        // Poincaré disk ray field transformation not implemented
        original
    }
}
```

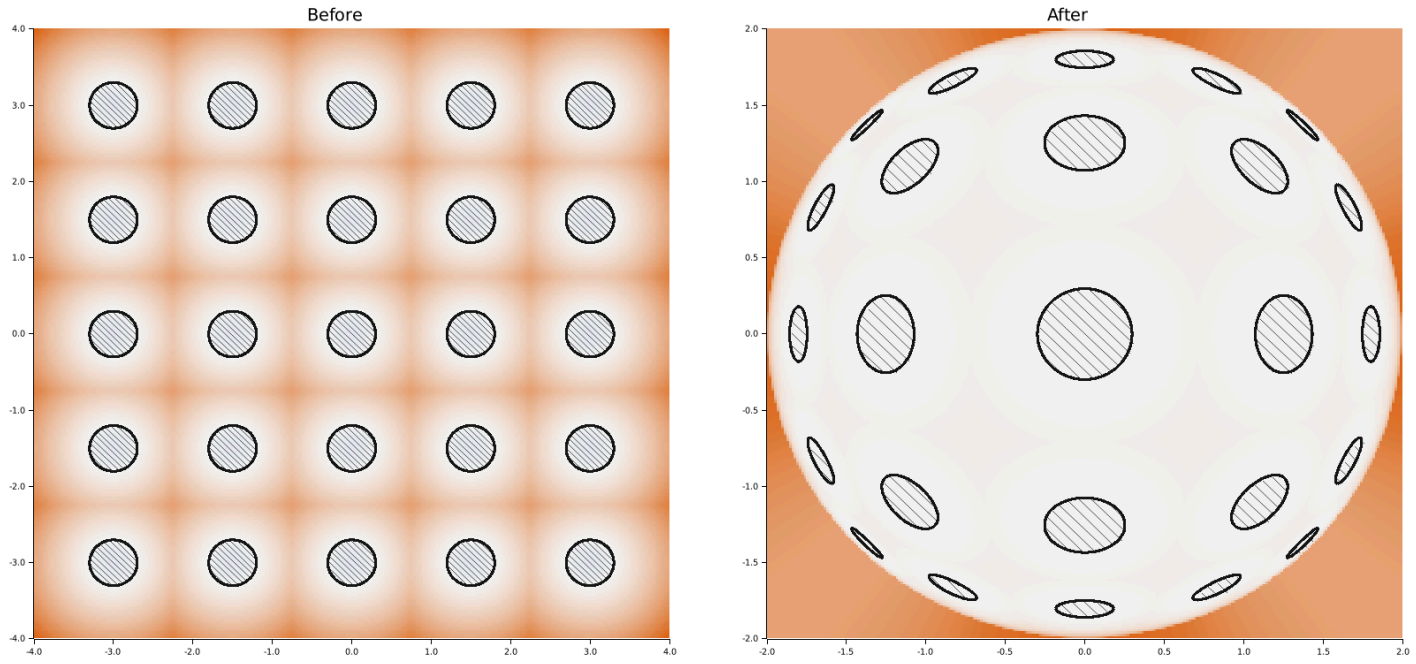


Figure 26: A grid of circles before (left) and after (right) Poincaré disk projection. Circles far from the origin appear smaller and compressed toward the disk boundary. Notice that the original bounds are $[-4, 4] \times [-4, 4]$, but in the transformed space, all points map to the disk of radius $\frac{1}{\eta}$ where η is the Poincaré disk scale. Field values outside of the disk are an artifact of our use of `clamp`.

3.4 Ray Casting

A batch of **rays** is defined by:

- Origins $X^{ij} \in \mathbb{R}^{m \times n}$
- Directions $U^{ij} \in \mathbb{R}^{m \times n}$ (unit vectors)

Ray casting is the computational problem of finding a tensor Λ^i such that $F^i(S^{ij}(\Lambda^i)) = 0^i$, where S^{ij} is the function representing ray extensions:

$$S^{ij}(\Lambda^i) = X^{ij} + \Lambda^i U^{ij} \tag{41}$$

There are a number of methods for computing Λ^i, either via analytical computation, or numerical iteration. The class of F^i determines which methods are available for use, and which are performant.

> **Row-Oriented Tensor Types**
>
> `crater.rs`'s API for ray casting is mediated by the `Rays` and `RayCastResult` types. These types are aliases for `Rotts<B, RayBatch<B, N>>` and `Rotts<B, RayBatchCastResult<B, N>>`, respectively (see Section 5).

3.4.1 Analytical Method

The analytical method computes intersection points directly by making the implicit function explicit:

$$F^i(S^{ij}(\Lambda^i)) = 0^i \tag{42}$$

is inverted to solve for Λ^i:

$$\Lambda^i = R^i(X^{ij}, U^{ij}) \tag{43}$$

Where R^i is a `RayField`. `RayField`s are similar to `ScalarField`s, but take an extra input parameter: a tensor of unit directions U^{ij}. The field returns the rank-1 tensor of distances from X^{ij} to the point at which $F^i(S^{ij}(\Lambda^i)) = 0^i$ along the direction U^{ij}.

3.4.1.1 Hyperplanes

For a hyperplane with normal vector n^j, define:

$$\alpha^i = X^{ij} n_j, \quad \beta^i = U^{ij} n_j \tag{44}$$

The ray field is:

$$R^i(X^{ij}, U^{ij}) = \frac{\alpha^i}{\beta^i} \tag{45}$$

Degenerate cases:
- If $\beta^i = 0$: ray is parallel to the hyperplane (undefined)
- If $\alpha^i = 0$: ray origin is on the hyperplane
- If $\frac{\alpha^i}{\beta^i} < 0$: intersection is behind the ray origin

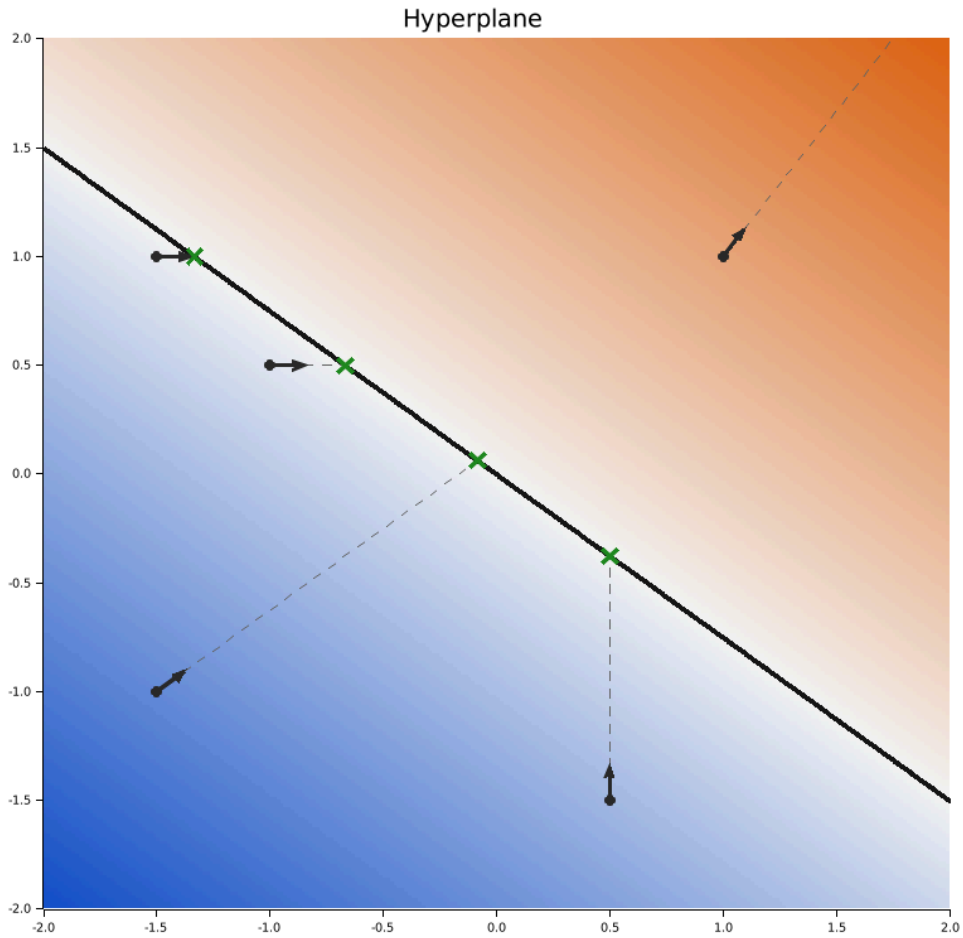


Figure 27: Ray-hyperplane intersections. Rays either hit the surface (green X) or miss (extending to bounds).

3.4.1.2 Hyperspheres

For a hypersphere with radius r, the primal scalar field is:

$$F^i(X^{ij}) = X^{ij} X^i_j - r^2 \tag{46}$$

Substituting the ray equation and expanding:

$$\begin{aligned} 0^i &= F^i(X^{ij} + \Lambda^i U^{ij}) \\ &= (X^{ij} + \Lambda^i U^{ij})\left(X^i_j + \Lambda^i U^i_j\right) - r^2 \\ &= \left(U^{ij} U^i_j\right)\Lambda^i \Lambda^i + \left(2 X^{ij} U^i_j\right)\Lambda^i + \left(X^{ij} X^i_j - r^2\right) \end{aligned} \tag{47}$$

This is a quadratic in Λ^i. With coefficients:

$$a^i = U^{ij}U^i_j, \quad b^i = 2X^{ij}U^i_j, \quad c^i = X^{ij}X^i_j - r^2 \tag{48}$$

The discriminant is $\Delta^i = b^i b^i - 4a^i c^i$, and solutions are:

$$\Lambda^i = \frac{-b^i \pm \sqrt{b^i b^i - 4a^i c^i}}{2a^i} \tag{49}$$

Cases:

- $\Delta^i < 0$: no intersection
- $\Delta^i = 0$: tangent (single point)
- $\Delta^i > 0$: two intersections; take smallest non-negative root

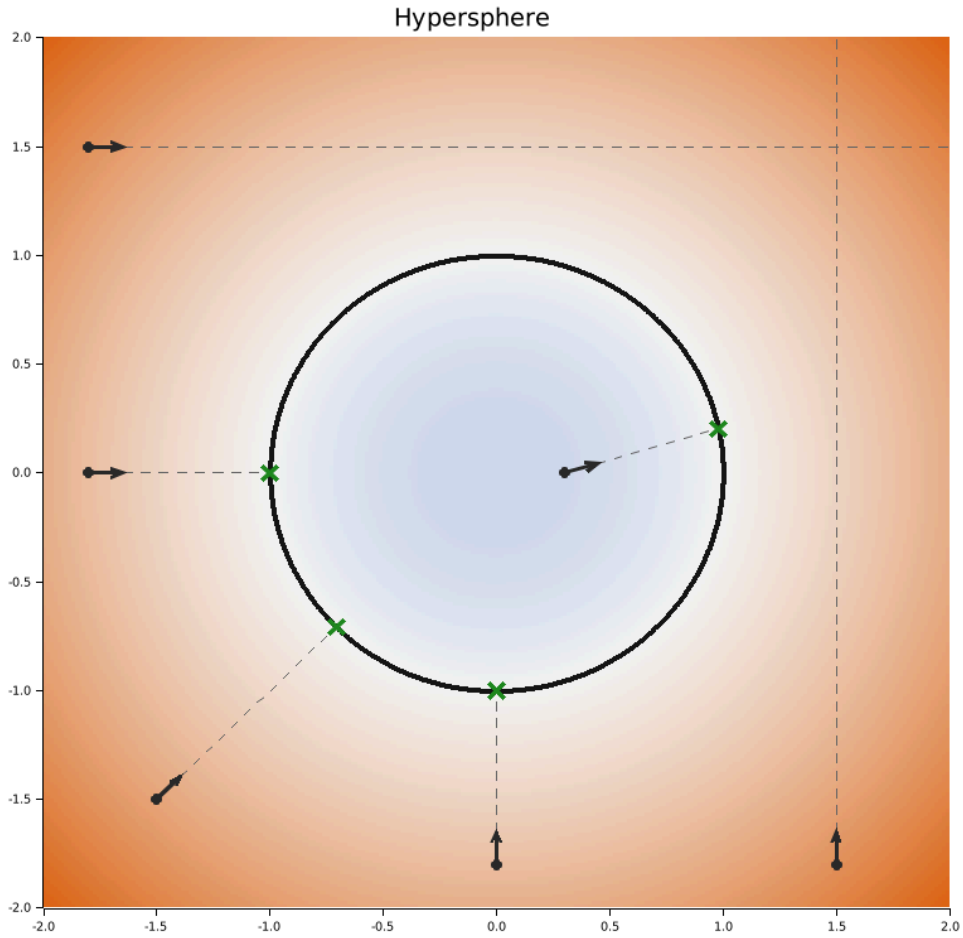


Figure 28: Ray-hypersphere intersections. Rays from outside hit the surface; rays from inside exit; some rays miss entirely.

3.4.1.3 Hypercones

For a cone with unit-axis a^j and opening angle θ:

$$F^i(X^{ij}) = \left(X^{ij}a_j\right)^2 - \cos(\theta)^2\left(X^{ij}X^i_j\right) \tag{50}$$

The ray intersection yields a quadratic with coefficients:

$$\begin{aligned} a^i &= \left(U^{ij}a_j\right)^2 - \cos(\theta)^2 U^{ij}U^i_j \\ b^i &= 2X^{ij}U^i_j\left(a_j - \cos(\theta)^2\right) \\ c^i &= \left(X^{ij}a_j\right)^2 - \cos(\theta)^2\left(X^{ij}X^i_j\right) \end{aligned} \tag{51}$$

Solutions follow the same quadratic formula, taking the smallest non-negative root.

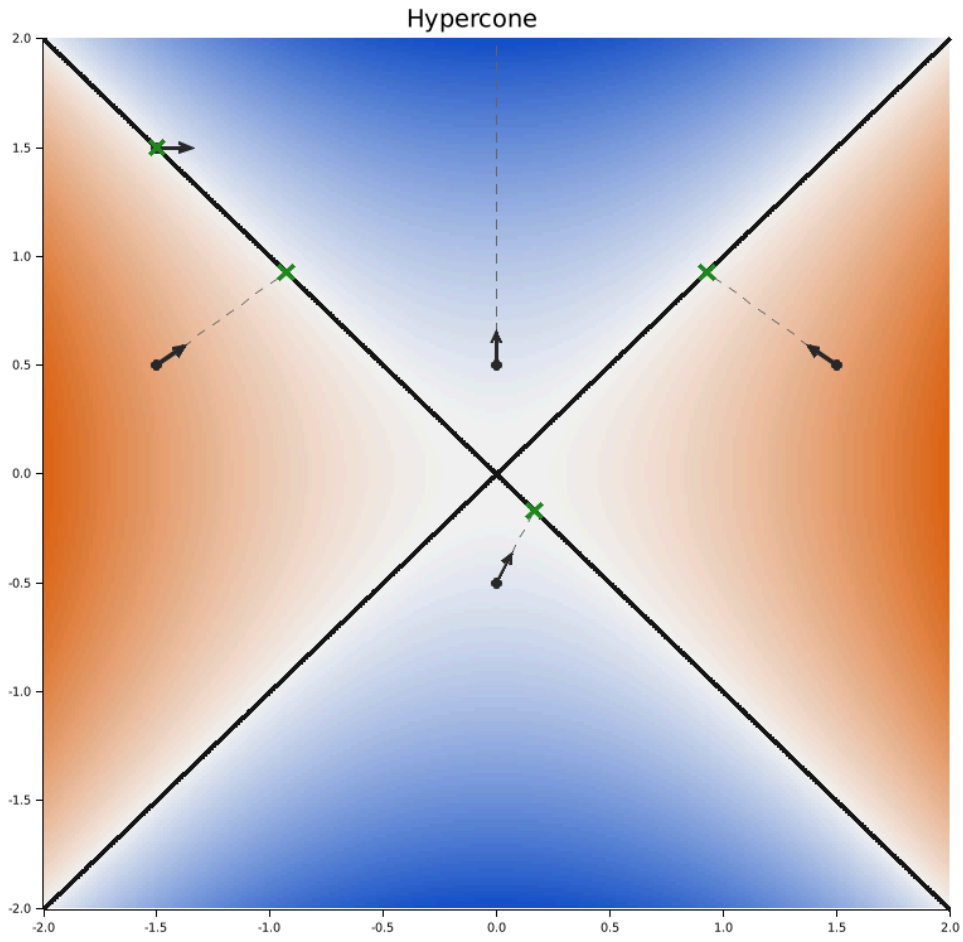


Figure 29: Ray-hypercone intersections. The cone extends from the apex; rays may hit either edge or pass through.

3.4.1.4 Hypercylinders

For a hypercylinder with radius r extending along the last dimension, project to the first $N-1$ dimensions:

Let $X'^{i(N-1)} = X^{ij}$ for $j \in [1, N-1]$ and $U'^{i(N-1)} = U^{ij}$ for $j \in [1, N-1]$.

The quadratic coefficients become:

$$a^i = U'^{i(N-1)} U'^i_{(N-1)}, \quad b^i = 2X'^{i(N-1)} U'^i_{(N-1)}, \quad c^i = X'^{i(N-1)} X'^i_{(N-1)} - r^2 \tag{52}$$

3.4.1.5 Composite CSG `Region`s

The analytical method can be extended to support any CSG `Region` that is composed entirely of primitives whose `RayField` is defined.

Given a batch of rays with origins X^{ij} and unit directions U^{ij}, the algorithm enumerates all K primitive halfspaces $\{F_k\}$ in the CSG tree. For each primitive, we evaluate its `RayField` R^i_k to obtain candidate intersection distances d^i_k. The final result selects the nearest distance where the ray actually intersects the composite region's boundary.

AnalyticalRayCast(X^{ij}, U^{ij}, F^i):

$\{F_k\}_{k=1}^{K} \leftarrow$ enumerate halfspaces **in** CSG tree
for $k \leftarrow 1$ **to** K **do**
    $d_k^i \leftarrow R_k^i(X^{ij}, U^{ij})$ // evaluate `RayField`
$d_*^i \leftarrow \infty$
for $k \leftarrow 1$ **to** K **do**
    if $F^i\big(S^{ij}(d_k^i)\big) = 0^i$ **and** $d_k^i < d_*^i$ **then**
        $d_*^i \leftarrow d_k^i$
return d_*^i

Algorithm 1: Analytical ray casting for composite CSG regions.

3.4.2 Bracket and Bisect Algorithm

For complex CSG regions without analytical solutions, numerical methods find intersections iteratively. The bracket-and-bisect approach proceeds in two phases: first bracketing to find an interval containing a root, then bisection to refine it.

3.4.2.1 Ray Bracketing

Bracketing marches along each ray in fixed steps until the field value changes sign, indicating the isosurface lies within the current interval. The inputs are ray origins X^{ij}, unit directions U^{ij}, the scalar field F^i, step size $\Delta\lambda$, and maximum search distance $\lambda_{\max}$. The algorithm maintains left and right bracket positions L^i and R^i, using a mask M^i to selectively update only those rays that haven't yet found a sign change.

RayBracket(X^{ij}, U^{ij}, F^i, $\Delta\lambda$, $\lambda_{\max}$):

$L^i \leftarrow 0^i$ // left bracket
$R^i \leftarrow L^i$ // right bracket
$F_L^i \leftarrow F^i(S^{ij}(L^i))$
while $R^i < \lambda_{\max}$ **do**
    $R^i \leftarrow L^i + \Delta\lambda$
    $F_R^i \leftarrow F^i(S^{ij}(R^i))$
    $M^i \leftarrow \mathbb{1}^i\{F_L^i F_R^i \geq 0^i\}$ // no sign change yet
    $L^i \leftarrow M^i R^i + (1 - M^i) L^i$
    $F_L^i \leftarrow M^i F_R^i + (1 - M^i) F_L^i$
return (L^i, R^i)

Algorithm 2: Ray bracketing: march along rays until a sign change is detected.

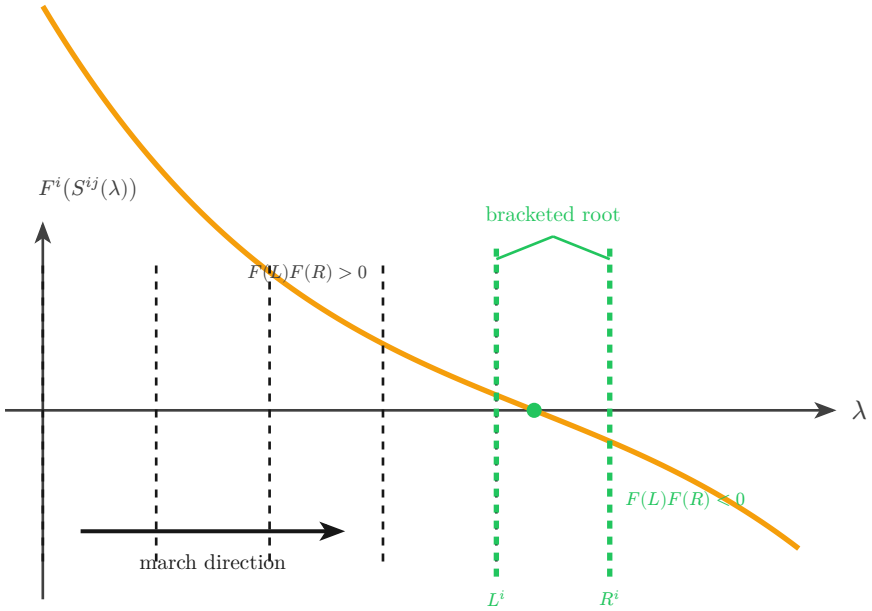


Figure 30: Ray bracketing: marching along the ray in steps of $\Delta\lambda$ until a sign change is detected, indicating the isosurface lies within the bracket $[L^i, R^i]$.

3.4.2.2 Ray Bisection

Once a bracket $[L^i, R^i]$ containing the isosurface is established, bisection refines it by repeatedly halving the interval. At each iteration, the midpoint C^i is evaluated and the bracket is narrowed to whichever half contains the sign change. The mask M^i determines whether the root lies in the right half (when F_L^i and F_C^i have the same sign) or the left half. The algorithm runs for a fixed number of iterations n; the final bracket width is $\Delta\lambda/2^n$ where $\Delta\lambda$ is the initial bracket width from the marching phase.

RayBisect$(L^i,\ R^i,\ F^i,\ n)$:

$F_L^i \leftarrow F^i(S^{ij}(L^i))$
for $k \leftarrow 1$ **to** n **do**
    $C^i \leftarrow \frac{L^i + R^i}{2}$
    $F_C^i \leftarrow F^i(S^{ij}(C^i))$
    $M^i \leftarrow \mathbb{1}^i\{F_L^i F_C^i > 0^i\}$ // root in right half
    $L^i \leftarrow M^i C^i + (1 - M^i) L^i$
    $R^i \leftarrow M^i R^i + (1 - M^i) C^i$
    $F_L^i \leftarrow M^i F_C^i + (1 - M^i) F_L^i$
return R^i

Algorithm 3: Ray bisection: iteratively halve the bracket to locate roots.

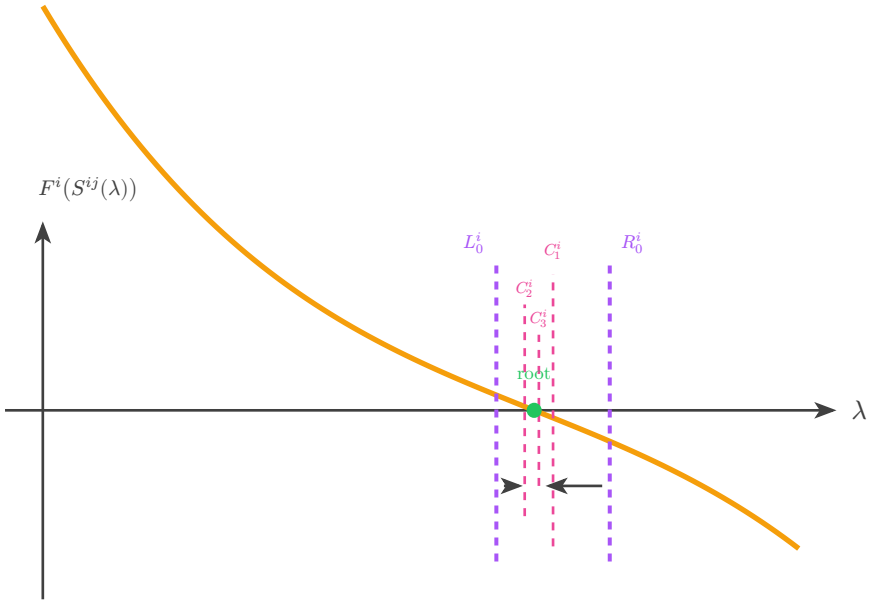


Figure 31: Ray bisection: iteratively halving the bracket $[L^i, R^i]$ to locate the root. Each iteration evaluates the midpoint C^i and updates the bracket to the half containing the sign change.

3.4.3 Newton's Method

For autodifferentiable fields, Newton's method provides quadratic convergence by using the gradient to predict where the function crosses zero. Given ray origins X^{ij} and directions U^{ij}, the algorithm iteratively refines position estimates. Each iteration computes the field value f^i and the directional derivative $g^i = \nabla F^i U^{ij}$ along the ray, then applies the Newton update to move toward the root. A step size parameter α allows damped updates for stability. Rays with zero gradient are nudged by δ to escape degenerate points.

NewtonRayCast(X^{ij}, U^{ij}, F^i, n, α, δ):

$P^{ij} \leftarrow X^{ij}$
for $k \leftarrow 1$ **to** n **do**
    $f^i \leftarrow F^i(P^{ij})$
    $g^i \leftarrow \nabla F^i(P^{ij})U^{ij}$ // directional derivative
    if $g^i = 0$ **then** $g^i \leftarrow \delta$ // nudge degenerate cases
    $P^{ij} \leftarrow P^{ij} - \alpha \frac{f^i}{g^i} U^{ij}$
return P^{ij}

Algorithm 4: Newton's method: use gradient information to converge quadratically.

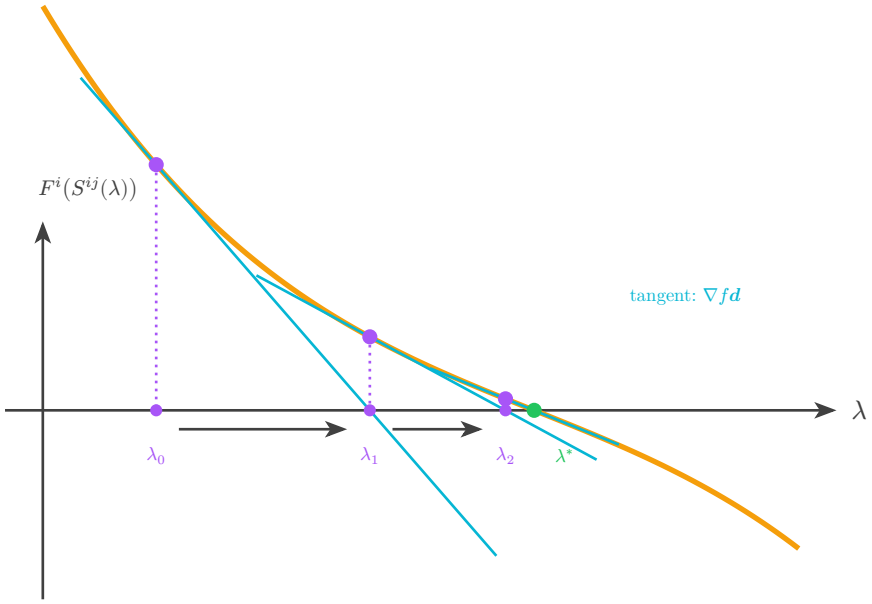


Figure 32: Newton's method: using the tangent line (directional derivative $\nabla f \boldsymbol{d}$) to compute successive approximations. Exhibits quadratic convergence—each iteration roughly doubles the number of correct digits.

3.5 Analysis Modules

3.5.1 Volume Estimation (Monte Carlo)

The algorithm samples N points uniformly from a bounding domain D and evaluates the scalar field F^i at each point. Points with $F^i \leq 0$ lie inside the region. The volume ratio equals the fraction of interior points, scaled by the domain volume.

MonteCarloVolume(F^i, D, N):

1 $X^{ij} \sim \text{Uniform}(D)$ // sample N points
2 $V^i \leftarrow F^i(X^{ij})$
3 $C \leftarrow \sum_{i=1}^{N} \mathbb{1}^i\{V^i \leq 0^i\}$
4 **return** $\text{Vol}(D)C/N$

Algorithm 5: Monte Carlo volume estimation.

The volume approximation:

$$\frac{V(R^i)}{V(D)} \approx \frac{\sum_{j=1}^{N} \mathbb{1}^i(F^i(X^{ij}) \leq 0^i)}{N} \tag{53}$$

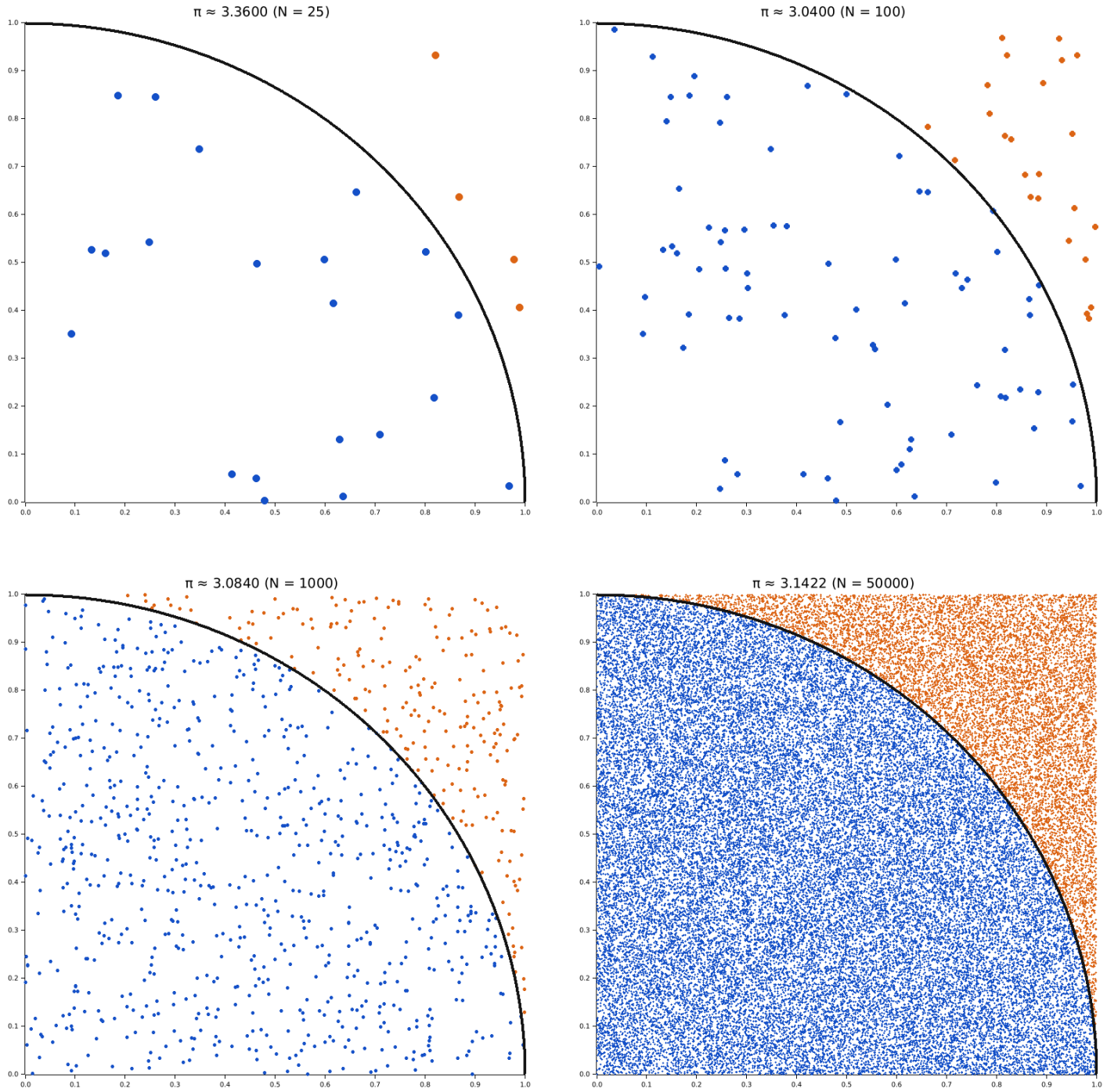


Figure 33: Monte Carlo estimation of π. Random points in $[0,1]^2$ are classified as inside (blue) or outside (orange) a quarter circle. As N increases, the estimate converges to π with error $O\left(1/\sqrt{N}\right)$.

3.5.1.1 Error Analysis

The error decreases as $O\left(\frac{1}{\sqrt{N}}\right)$. The standard error is:

$$\sigma_V \approx V(D)\sqrt{\frac{p(1-p)}{N}} \tag{54}$$

where $p = \frac{V(R^i)}{V(D)}$ is the proportion of domain occupied by the region.

3.5.2 Gradient Computation

The gradient of F^i at origins X^{ij} is:

$$\nabla F^i(X^{ij}) = \partial^j F^i(X^{ij}) = \begin{bmatrix} \partial^j F^i(X^{1j}) \\ \partial^j F^i(X^{2j}) \\ \vdots \\ \partial^j F^i(X^{mj}) \end{bmatrix} \tag{55}$$

Gradients are computed via automatic differentiation through the Burn framework.

3.5.2.1 Surface Normals

For origins X^{ij} on isosurface $F^i(X^{ij}) = c$, the unit normal is:

$$\hat{n} = \frac{\nabla F^i(X^{ij})}{|\nabla F^i(X^{ij})|} \tag{56}$$

3.5.3 Gradient Descent

Gradient descent finds points inside a region by iteratively stepping in the direction of steepest descent. Starting from origins X^{ij}, the algorithm updates positions:

$$X^{ij}_{k+1} = X^{ij}_k - \alpha \nabla F^i\left(X^{ij}_k\right) \tag{57}$$

where α is the step size. The process terminates when points satisfy the stopping condition (e.g., $F^i(X^{ij}) < 0$) or after a maximum number of steps.

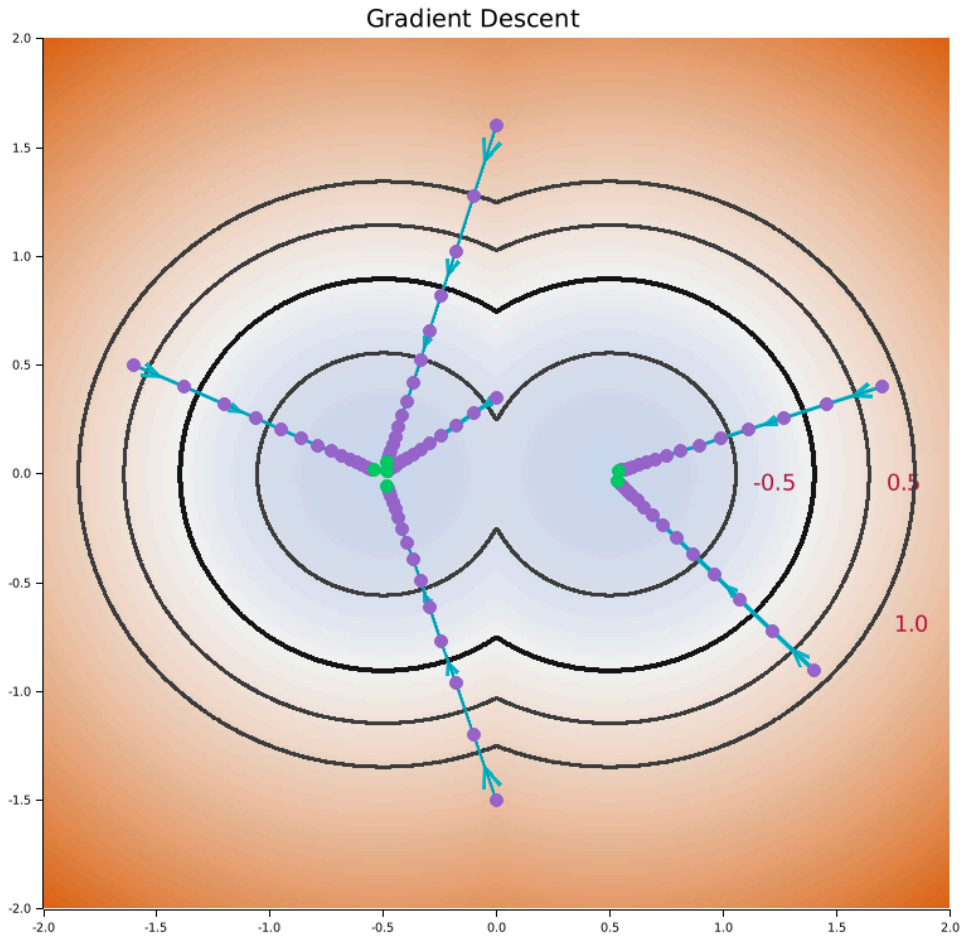


Figure 34: Gradient descent from an exterior point toward the region interior. Each step follows the negative gradient direction. Purple points show intermediate steps; green indicates convergence.

3.5.4 Root Finding (Newton's Method)

To find points *on* the isosurface (not just inside), Newton's method provides fast convergence. The update rule for finding $F^i(X^{ij}) = 0$:

$$X_{k+1}^{ij} = X_k^{ij} - F^i\left(X_k^{ij}\right) \frac{\nabla F^i\left(X_k^{ij}\right)}{\|\nabla F^i\left(X_k^{ij}\right)\|^2} \tag{58}$$

This is the multi-dimensional Newton step projected along the gradient direction.

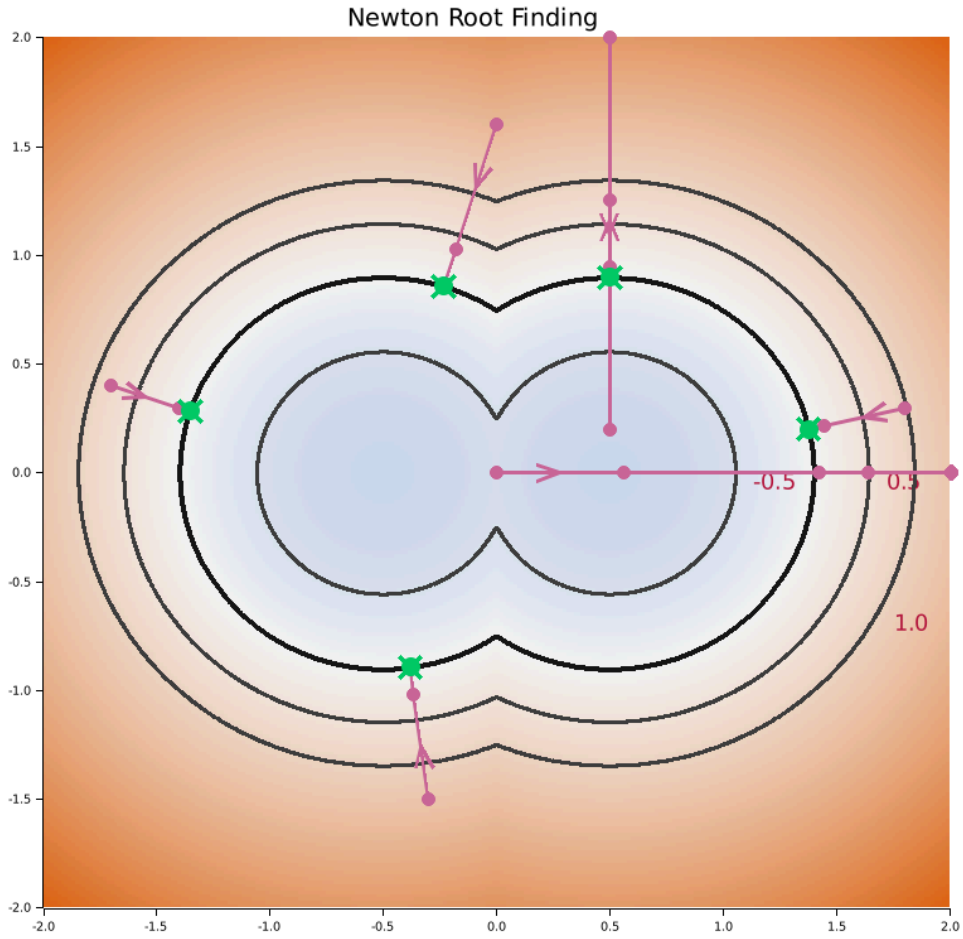


Figure 35: Newton root finding converging to the isosurface. The path shows rapid convergence characteristic of Newton's method. The green marker indicates successful convergence to the surface.

3.5.5 Adaptive Bounding

The bounding tree algorithm recursively subdivides space to find a tight bounding box around a region. Each cell is classified as:

- **Inside**: entirely contained within the region
- **Outside**: entirely outside the region
- **Boundary**: intersects the isosurface

Cells are subdivided until reaching maximum depth or becoming homogeneous. The tight bounding box is computed from all Inside and Boundary leaf cells.

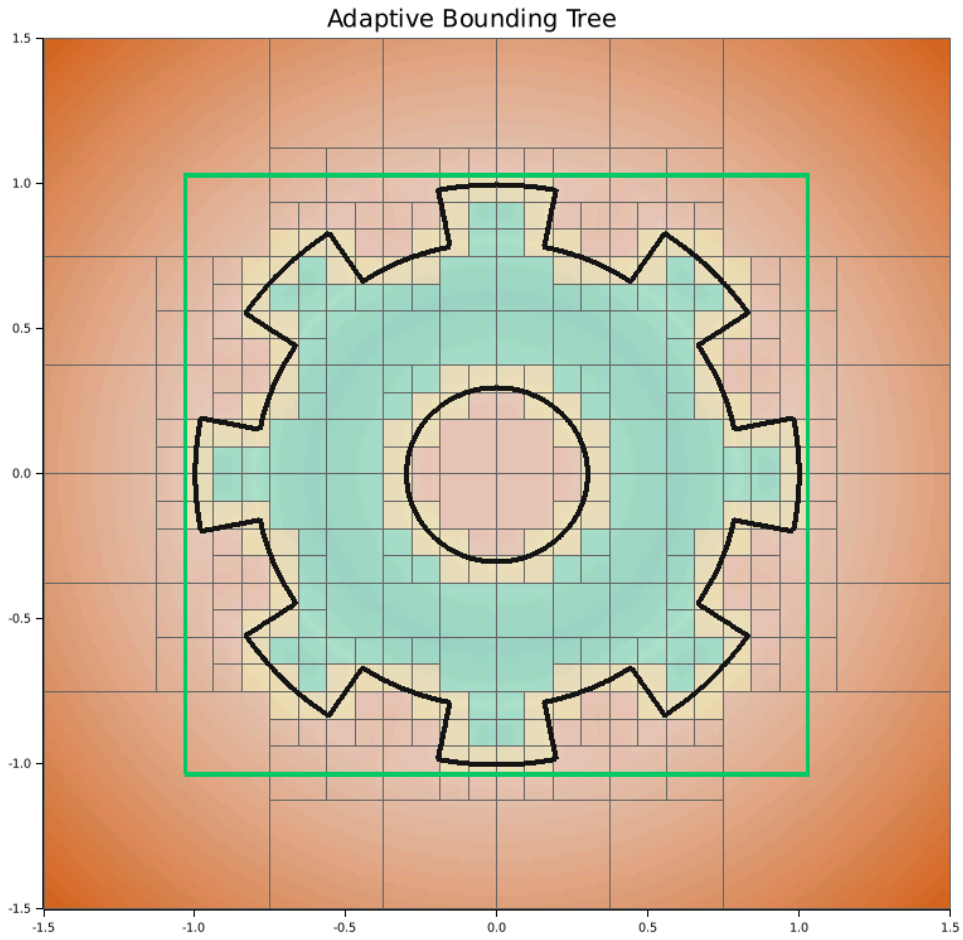


Figure 36: Adaptive quadtree bounding of a gear region. Green cells are inside, orange are outside, yellow intersect the boundary. The green outline shows the computed tight bounding box.

3.5.5.1 Cell Classification

Given a bounding box D with bounds $\left[x^j_{\min}, x^j_{\max}\right]$, we generate a uniform grid of s^n sample points where s is the samples per dimension. For the i-th sample point in the grid:

$$X^{ij} = x^j_{\min} + T^{ij}\left(x^j_{\max} - x^j_{\min}\right) \tag{59}$$

where $T^{ij} \in [0,1]^{m\times n}$ is the normalized grid coordinate for sample i in dimension j.

The cell is classified by evaluating $F^i(X^{ij})$:

- `Inside`: $F^i(X^{ij}) < 0 \forall i$
- `Outside`: $F^i(X^{ij}) > 0 \forall i$
- `Boundary`: otherwise (mixed signs or samples `On` surface)

3.5.5.2 Subdivision Scheme

Each Boundary cell is subdivided into 2^n children by splitting along each axis at the midpoint. In 2D this produces 4 quadrants (quadtree); in 3D it produces 8 octants (octree), and so on.

For any intermediate bounding box with center $c^j = \frac{x^j_{\min}+x^j_{\max}}{2}$, the k-th child ($k \in [0, 2^n - 1]$) has bounds determined by the binary representation of k. For each dimension j:

$$x^{(k)j}_{\min} = \begin{cases} x^j_{\min} \text{ if bit } j \text{ of } k = 0 \\ c^j \text{ otherwise} \end{cases} \tag{60}$$

$$x_{\max}^{(k)j} = \begin{cases} c^j \text{ if bit } j \text{ of } k = 0 \\ x_{\max}^j \text{ otherwise} \end{cases} \tag{61}$$

3.5.5.3 Algorithm

```
FindBoundingTree(F^i, D, d_max, s):
 1  nodes ← ∅, leaves ← ∅
 2  SubdivideRecursive(D, 0)                    // start at root with depth 0
 3  return (nodes, leaves)
 4
 5  SubdivideRecursive(B, d):
 6  X^ij ← uniform grid of s^N samples in B
 7      v^i ← F^i(X^ij)
 8      status ← Classify(v^i)
 9      if status = Boundary and d < d_max then
10          {B_k}_{k=0}^{2^N−1} ← split B into 2^N children
11          for k ← 0 to 2^N − 1 do
12              SubdivideRecursive(B_k, d + 1)
13      else
14          append (B, status) to leaves
```

Algorithm 6: Adaptive bounding tree construction. The algorithm recursively subdivides Boundary cells until reaching maximum depth $d_{\max}$ or finding homogeneous (Inside/Outside) cells.

3.6 Mesh Extraction

Mesh extraction converts an implicit surface representation (the zero level set of a scalar field) into an explicit geometric representation suitable for rendering, physics simulation, or export to standard formats.

3.6.1 Simplicial Complexes

The output of mesh extraction is a *simplicial complex*—a collection of simplices that approximate the isosurface. A k-simplex is the convex hull of $k+1$ non-degenerate points:

k	Name	Vertices	Use
0	Point	1	Isolated samples
1	Segment	2	Curves, contours
2	Triangle	3	Surface meshes
3	Tetrahedron	4	Volume meshes

In n dimensions, the isosurface $F^i = c$ is an $(n-1)$-dimensional manifold. Mesh extraction approximates this manifold using $(n-1)$-simplices:

- **2D fields**: Extract 1-simplices (line segments) forming a polyline contour
- **3D fields**: Extract 2-simplices (triangles) forming a surface mesh

3.6.2 The Marching Algorithm

The *marching* family of algorithms (marching squares, marching cubes) share a common structure:

1. **Discretize** the domain into a regular grid of hypercubes
2. **Evaluate** the scalar field at each grid vertex

3. **Classify** each cell by which corners are inside ($F^i < 0$) vs outside ($F^i \geq 0$)
4. **Lookup** the simplex configuration from a precomputed table
5. **Interpolate** vertex positions along edges where the isosurface crosses

The key insight is that each cell's topology depends only on the *sign pattern* of its corners, not the actual field values. This reduces the problem to a finite lookup table indexed by a bit pattern.

3.6.3 Hypercube Classification

Each hypercube corner is classified as inside (bit = 1) or outside (bit = 0). The corner bits are packed into an integer index:

$$\text{index} = \sum_{k=0}^{2^n-1} \text{bit}_k \cdot 2^k \tag{62}$$

Dimension	Corners	Index bits	Configurations
2D (squares)	4	4	$2^4 = 16$
3D (cubes)	8	8	$2^8 = 256$ (15 unique by symmetry)

The index refers to the bit pattern of the corner classification. For example, in 2D, the index `6` corresponds to the bit pattern `0110`, indicating that the 1st and 2nd corners are inside and the 3rd and 0th corners are outside. This, and the remaining configurations of 4 bits are shown below:

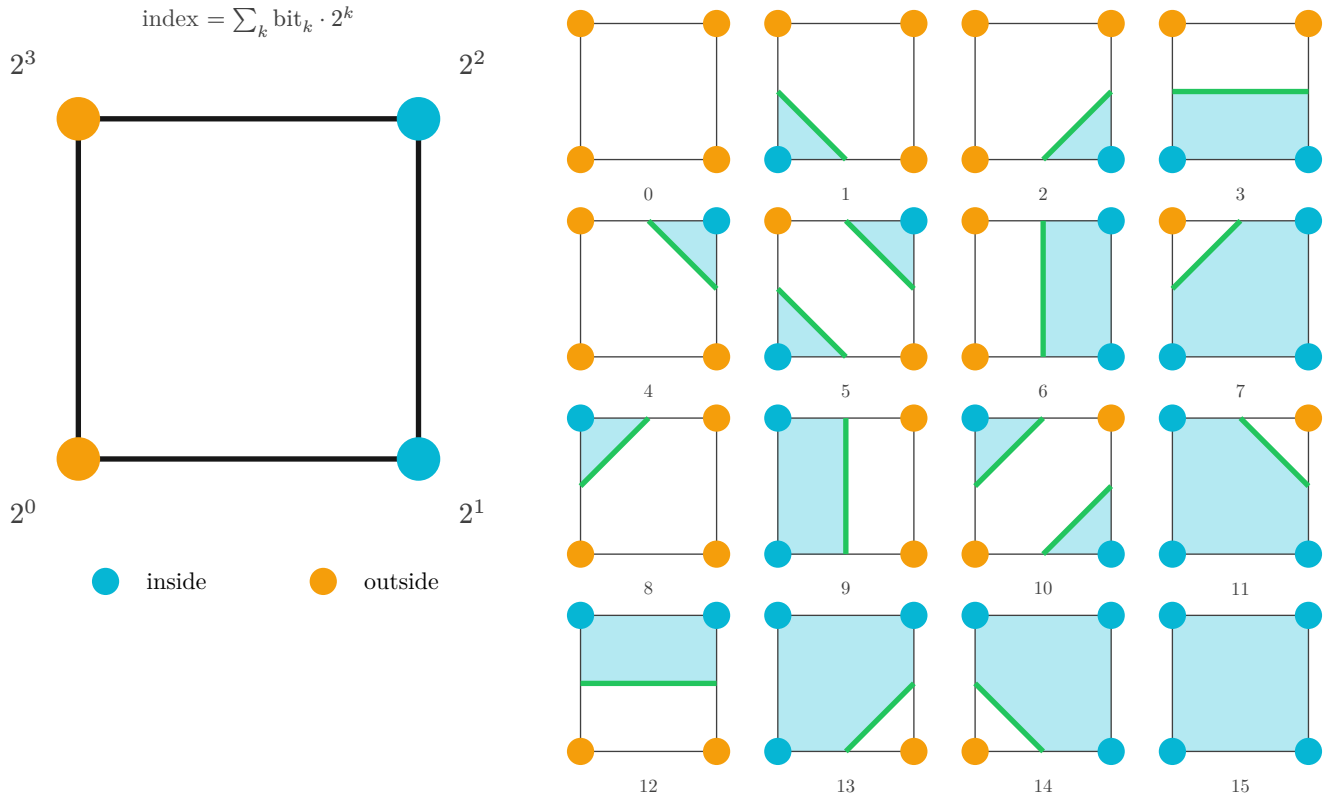


Figure 37: Marching squares: (left) cell with corner bit positions, (right) all 16 configurations with inside regions shaded and contour segments in green.

This process is naturally extended to 3D, where each cell is classified by its 8 corners. In this example, the hypercube is index `105`, corresponding to the bit pattern `01101001`, indicating that the 0th, 3rd, 5th, and 6th corners are inside and the rest are outside. This, and the remaining configurations of 8 bits, deduplicated by symmetries are shown below:

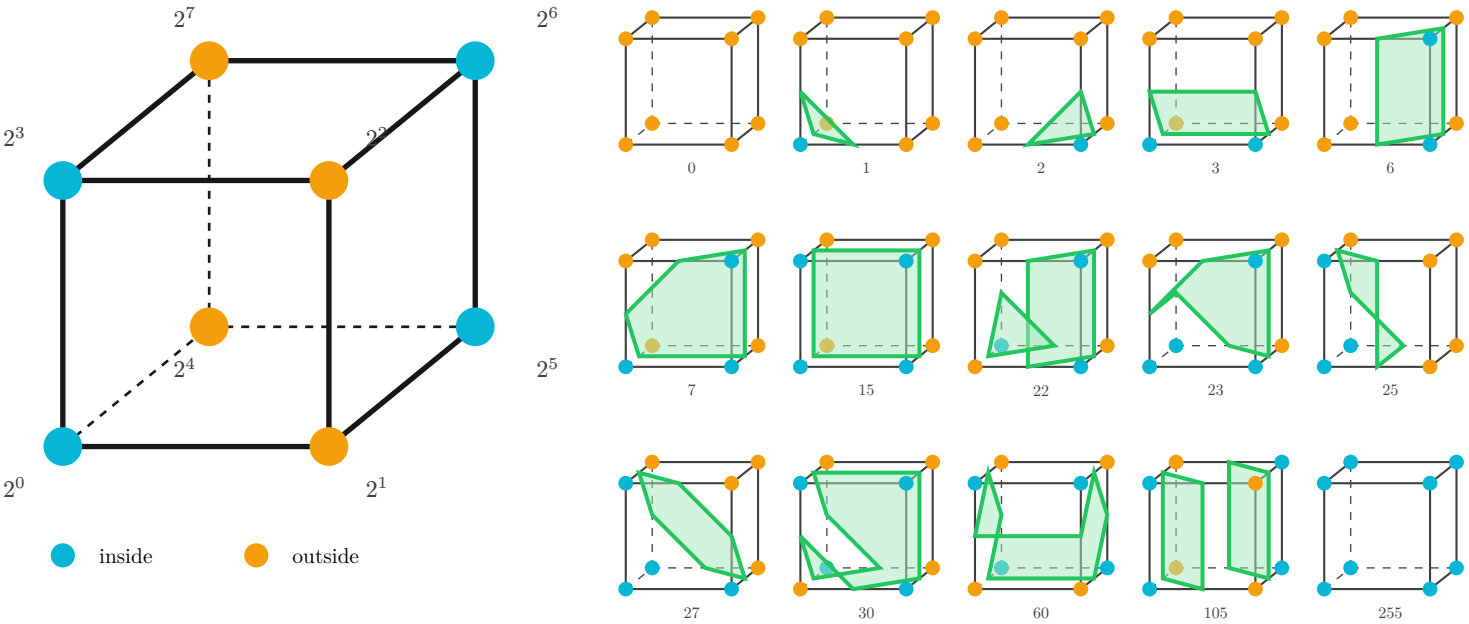


Figure 38: Marching cubes: (left) cell with 8 corners labeled by bit position, (right) representative cases showing triangulated surfaces. The 256 configurations reduce to 15 unique cases by symmetry. Triangulation of the surfaces in the diagram is elided for visual clarity.

3.6.4 Edge Interpolation

When the isosurface crosses an edge between vertices v_1, v_2 with field values f_1, f_2, linear interpolation locates the crossing point:

$$p = v_1 + t(v_2 - v_1) \quad \text{where} \quad t = \frac{-f_1}{f_2 - f_1} \tag{63}$$

This assumes locally linear field behavior. Higher-order interpolation could improve accuracy but is not yet implemented.

3.6.5 Resolution

The mesh approximation improves with grid resolution, but at increasing computational cost. Doubling resolution quadruples cell count in 2D and octuples it in 3D.

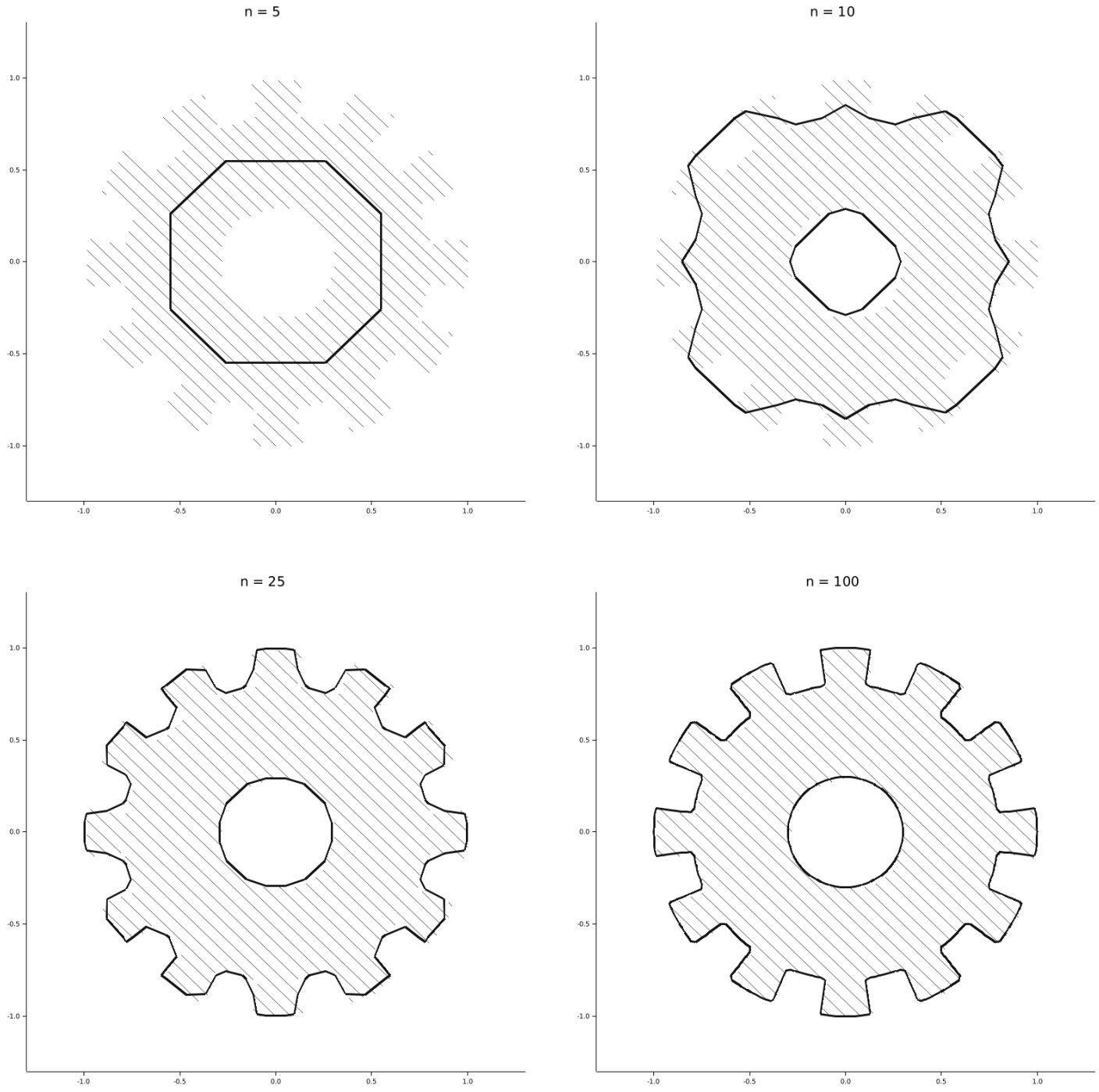


Figure 39: Effect of grid resolution on mesh extraction. A 12-tooth gear extracted at $n = 5, 10, 25, 100$. Low resolution produces a coarse approximation; higher resolution captures fine detail.

3.6.6 Algorithms

MarchingSquares(F^i, n_x, n_y, D):

$X^{ij} \leftarrow$ grid over D with $(n_x + 1) \times (n_y + 1)$ vertices
$V^{ij} \leftarrow F^i(X^{ij})$ // evaluate field at vertices
segments $\leftarrow \emptyset$
for each cell (c_x, c_y) **in** grid **do**
    idx $\leftarrow \sum_{k=0}^{3} \mathbb{1}\{V_k < 0\} 2^k$ // 4-bit classification
    edges $\leftarrow$ EdgeTable[idx]
    for each edge pair **in** edges **do**
        $p \leftarrow \text{Interpolate}(v_1, v_2, V_1, V_2)$
        append p **to** segments
return segments

Algorithm 7: Marching squares extracts a polyline contour from a 2D scalar field.

MarchingCubes(F^i, n_x, n_y, n_z, D):

$X^{ij} \leftarrow$ grid over D with $(n_x + 1) \times (n_y + 1) \times (n_z + 1)$ vertices
$V^{ij} \leftarrow F^i(X^{ij})$ // evaluate field at vertices
triangles $\leftarrow \emptyset$
for each voxel (c_x, c_y, c_z) **in** grid **do**
    idx $\leftarrow \sum_{k=0}^{7} \mathbb{1}\{V_k < 0\} 2^k$ // 8-bit classification
    config $\leftarrow$ TriTable[idx]
    for each triangle **in** config **do**
        for each edge **in** triangle **do**
            $p \leftarrow \text{Interpolate}(v_1, v_2, V_1, V_2)$
        append triangle **to** triangles
return triangles

Algorithm 8: Marching cubes extracts a triangle mesh from a 3D scalar field.

3.6.6.1 Tensor-Native Implementation

The naïve implemenatation of the marching algorithm iterates over hypercubes sequentially. `crater.rs` exploits tensor parallelism using convolutions. Instead of looping over cells, a specially constructed convolution kernel extracts all corner values simultaneously.

3.6.6.1.1 2D: Using `conv2d`

A 4-channel kernel $K \in \mathbb{R}^{4\times1\times2\times2}$ extracts corner values, where each output channel selects one corner via a one-hot 2×2 filter. Applying `conv2d` to the scalar field values produces a tensor containing all corner values for every hypercube:

$$\text{corners} = K * V \quad \rightarrow \quad \text{shape } 1 \times 4 \times n_x \times n_y \tag{64}$$

The cell indices are then computed via weighted sum:

$$\text{idx}_{i,j} = \sum_{k=0}^{3} \mathbb{1}\{(\text{corners})_k < 0\} \cdot 2^k \tag{65}$$

3.6.6.1.2 3D: Using `conv3d`

An 8-channel kernel $K \in \mathbb{R}^{8\times1\times2\times2\times2}$ extracts the 8 voxel corners:

$$\text{corners} = K * V \quad \rightarrow \quad \text{shape } 1 \times 8 \times n_x \times n_y \times n_z \tag{66}$$

$$\text{idx}_{i,j,k} = \sum_{c=0}^{7} \mathbb{1}\{(\text{corners})_c < 0\} \cdot 2^c \tag{67}$$

—

3.7 Rescaling Transformations

Rescaling transformations provide mathematically precise control over scalar field output ranges through univariate functions.

Given a scalar field $F^i : \mathbb{R}^{m \times n} \to \mathbb{R}^m$, a rescaling transformation produces:

$$G^i(\boldsymbol{x}) = \eta\big(F^i(\boldsymbol{x})\big) \tag{68}$$

where $\eta : \mathbb{R} \to \mathbb{R}$ is a univariate function applied element-wise.

3.7.1 Function Classes

1. **Hyperbolic Tangent Family**: Standard and scaled tanh functions
2. **Logistic (Sigmoid) Family**: Standard and scaled sigmoid functions
3. **Arctangent Family**: Standard and scaled arctan functions
4. **Error Function**: Gaussian-like transition with erf approximation
5. **Soft Clipping**: Linear preservation with asymptotic bounds

3.7.2 Properties of Rescaling

1. **Isosurface Preservation**: If $\mathcal{S}_c = \{\boldsymbol{x} : F^i(\boldsymbol{x}) = c\}$ is an isosurface of F^i, then the transformed field has isosurfaces at $\eta(c)$
2. **Monotonicity**: For strictly monotonic η, relative ordering of field values is preserved
3. **Bounded Output**: Many rescaling functions map $\mathbb{R}$ to bounded intervals
4. **Differentiability**: Most rescaling functions are smooth C^∞

3.7.3 Rescaling for Ray Cast Stability

Rescaling is crucial for numerical stability in ray casting. Consider $f(\lambda)$ representing the scalar field along a ray. Ray casting finds a root of $f(\lambda) = 0$ to tolerance ε.

The floating point number line is not uniformly distributed—differences between representable numbers increase with magnitude. For high-gradient fields, no representable λ may satisfy tolerance ε.

Rescaling maps field values to a closed interval, vertically compressing the field. This ensures at least one λ exists satisfying the tolerance, and for continuous fields, the converged λ is at most one floating point number from the true root.

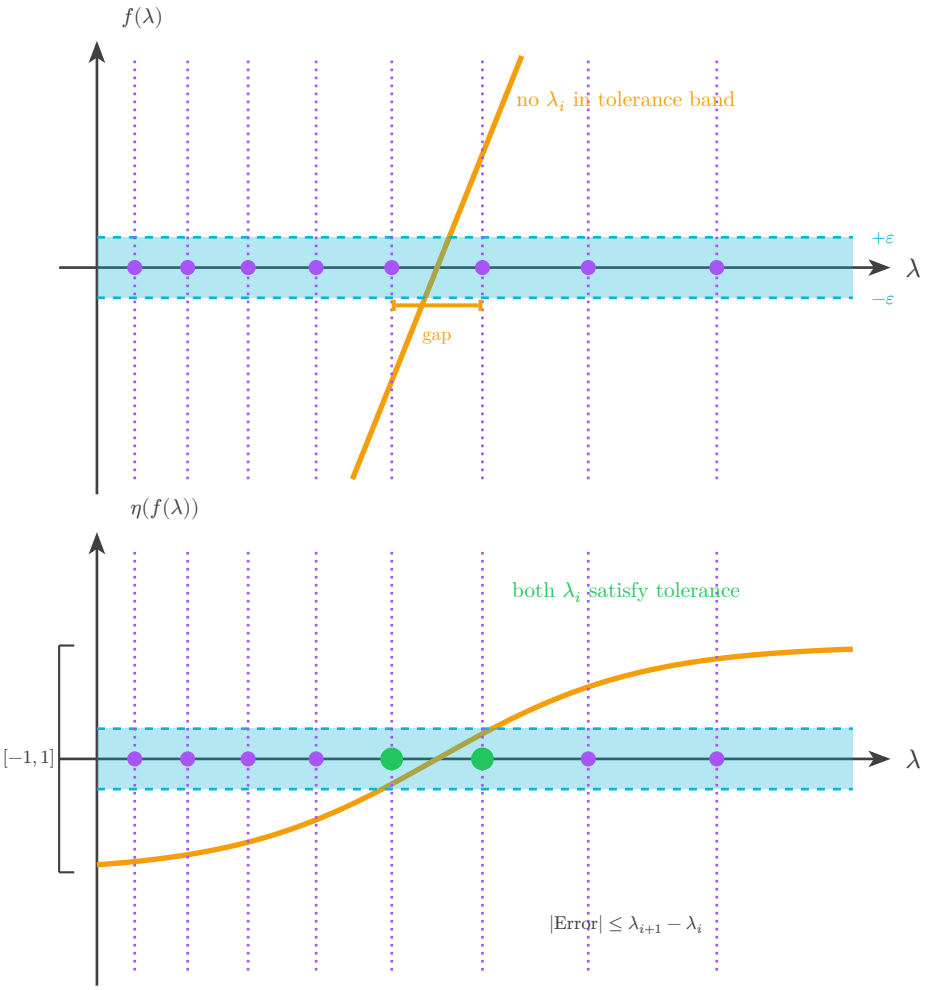


Figure 40: Floating point precision and rescaling. Top: a steep field $f(\lambda)$ may have no representable λ_i within the tolerance band $\pm\varepsilon$ because floating point gaps widen with magnitude. Bottom: rescaling via $\eta = \tanh$ compresses the field, reducing the gradient near the root so that at least one λ_i satisfies the tolerance.

4 XS: Nuclear Cross Section Service

Coming Soon

The `xs.rs` crate provides nuclear cross section data services for particle transport simulations.

Documentation is under development.

5 Rott: Row-Oriented Tensor Types

Data are considered "Row-oriented" if each row of the dataset is independent from the others. `rott.rs` enables a "structure-of-arrays" interface for common operations on row-oriented data. The core abstraction is a Row-Oriented Tensor Type (`Rott` - a Rust trait), which owns one or more tensor objects where the 0th dimensions of all tensors are equal.

5.1 Row-Oriented Tensor Type (ROTT)

A Row-Oriented Tensor Type (ROTT) owns one or more tensor objects where the 0th dimensions of all tensors are all equal.

Consider rank-2 tensors (matrices) of the form $X^{ij} \in \mathbb{R}^{m \times n}$, representing m independent rows with data dimension n. Such objects can represent a batch of particle positions or velocities ($n = 3$), energies ($n = 1$), and more. While the discussion herein is limited to tensors of rank 2, this is not a mandate. A `Rott` may own tensors of any rank ≥ 1.

We collapse these tensors into a compound type, employing a structure-of-arrays approach:

$$\mathbb{A}^i = (X^{ix}, Y^{iy}, Z^{iz}, ...) \tag{69}$$

The ith "row" of $\mathbb{A}^i$ can be thought of as a slice across the 0th index of all constituent tensors. The ith row is a collection of tensors, each of which has a rank reduced by 1. This is analogous to slices of single tensors, but extended to a collection of multiple, related tensors.

The `bb` notation is used throughout to denote a type as a `Rott`. Attributes of these objects are notated with the . syntax:

$$\mathbb{A}.X^{iu} \tag{70}$$

Let's implement a simple representation of some particles:

```rust
#[derive(Clone)]
pub struct ParticleBatch<B: Backend> {
    pub positions: Tensor<B, 2>,  // [batch, 3]
    pub directions: Tensor<B, 2>, // [batch, 3]
    pub energies: Tensor<B, 1>,   // [batch]
}
```

In the above, we associate three tensors: `positions`, `directions`, and `energies`, each of which has their own rank and shape. A row of this `Rott` is a triplet of a **single** particle's position (3-vector), direction (3-vector), and energy (scalar).

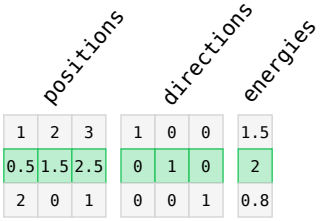


Figure 41: A `ParticleBatch` with 3 particles. The highlighted row represents a single particle.

The *cardinality* of such types is equal to the size of the 0-th dimension, notated as $|\mathbb{A}^i|$.

```rust
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[1.0, 2.0, 3.0], [0.5, 1.5, 2.5], [2.0, 0.0, 1.0]], device),
    directions: Tensor::from_floats(
        [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]],
        device,
    ),
    energies: Tensor::from_floats([1.5, 2.0, 0.8], device),
};
```

```
let cardinality = particles.cardinality(); // |P^i| = 3
assert_eq!(cardinality, 3);
```

This patterns enables more than tensor organization; it offers higher-level operations that "feel" like single-tensor operations, but really act over all constituent tensors.

5.2 Operations on ROTTs

5.2.1 Predicate Notation

Boolean masks are created from predicates using the Iverson bracket $[P]$. The resulting mask inherits the shape of the operands in the predicate.

For a predicate P^i evaluated element-wise over index i:

$$[P^i] = \begin{cases} 1 \text{ if } P^i \text{ is true} \\ 0 \text{ if } P^i \text{ is false} \end{cases} \tag{71}$$

The output is a boolean tensor $m^i \in \{0,1\}^M$ where M is the size of the indexed dimension.

```
let energies: Tensor<B, 1> = Tensor::from_floats([0.5, 2.0, 1.5, 0.3], device);

// m^i = [E^i > 1.0]
let mask = energies.greater_elem(1.0);

// [false, true, true, false]
let expected: Tensor<B, 1, Bool> =
    Tensor::from_data(TensorData::from([false, true, true, false]), device);
assert_eq!(mask.to_data(), expected.to_data());
```

5.2.2 `select`

Masked selection extracts rows of the `Rott` into a smaller `Rott`, according to a mask. Given a mask $m^i \in \{0,1\}^M$, the selection operation for a single tensor X^{ij}

$$X^{ij}[m^i] = X^{\text{true}(m^i)j} \tag{72}$$

where $\text{true}(m^k) = \{k : m^k = 1\}$ is the set of indices where the mask is true.

The tensor cardinality is reduced:

$$|A^i[m^i]| = \sum_i m^i \leq |A^i| \tag{73}$$

The same operation can be broadcasted over a `Rott` input:

$$\mathbb{A}^i[m^i] = (X^{ij}[m^i], Y^{ik}[m^i], Z^{il}[m^i]) \tag{74}$$

```
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [1.0, 0.0, 0.0],
            [2.0, 0.0, 0.0],
            [3.0, 0.0, 0.0],
            [4.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::from_floats(
```

```
        [
            [1.0, 0.0, 0.0],
            [0.0, 1.0, 0.0],
            [0.0, 0.0, 1.0],
            [1.0, 0.0, 0.0],
        ],
        device,
    ),
    energies: Tensor::from_floats([0.5, 2.0, 1.5, 0.3], device),
};

// m^i = [E^i > 1.0]
let mask = particles.energies.clone().greater_elem(1.0);

// P^i[m^i] -> cardinality reduced from 4 to 2
let selected = particles.select(mask);
assert_eq!(selected.cardinality(), 2);
assert_eq!(
    selected.energies.to_data().to_vec::<f32>().unwrap(),
    vec![2.0, 1.5]
);
```

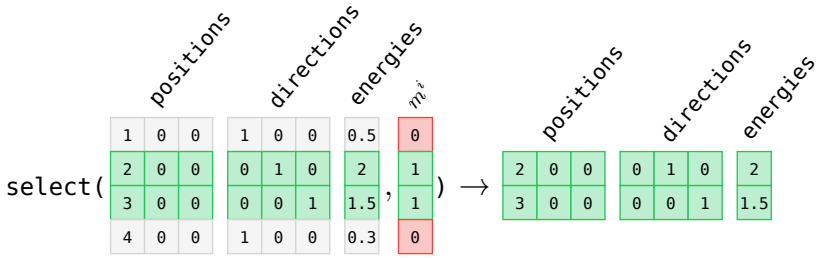


Figure 42: Masked selection extracts rows where the mask is true, producing a smaller `Rott`.

5.2.3 `mask_where`

Masked where selects between two ROTTs element-wise based on a mask:

$$\text{mask\_where}(\mathbb{A}^i, m^i, \mathbb{B}^i) = \begin{cases} \mathbb{B}^i \text{ if } m^i = 1 \\ \mathbb{A}^i \text{ if } m^i = 0 \end{cases} \tag{75}$$

```
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [1.0, 0.0, 0.0],
            [2.0, 0.0, 0.0],
            [3.0, 0.0, 0.0],
            [4.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::from_floats(
        [
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
        ],
        device,
    ),
    energies: Tensor::from_floats([1.0, 2.0, 3.0, 4.0], device),
};
```

```
let particles_to_overwrite: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [9.0, 9.0, 9.0],
            [9.0, 9.0, 9.0],
            [9.0, 9.0, 9.0],
            [9.0, 9.0, 9.0],
        ],
        device,
    ),
    directions: Tensor::from_floats(
        [
            [0.0, 1.0, 0.0],
            [0.0, 1.0, 0.0],
            [0.0, 1.0, 0.0],
            [0.0, 1.0, 0.0],
        ],
        device,
    ),
    energies: Tensor::from_floats([1.5, 2.5, 3.5, 4.5], device),
};

// Mask: overwrite indices 1 and 2
let overwrite: Tensor<B, 1, Bool> = Tensor::from_data([false, true, true, false], device);

// Select initial values where mask=false, updated values where mask=true
let result = particles.mask_where(overwrite, particles_to_overwrite);

// Energies: [1.0, 2.5, 3.5, 4.0] - particles 1,2 have new energies, positions and directions
assert_eq!(
    result.energies.clone().into_data().to_vec::<f32>().unwrap(),
    vec![1.0, 2.5, 3.5, 4.0]
);
```

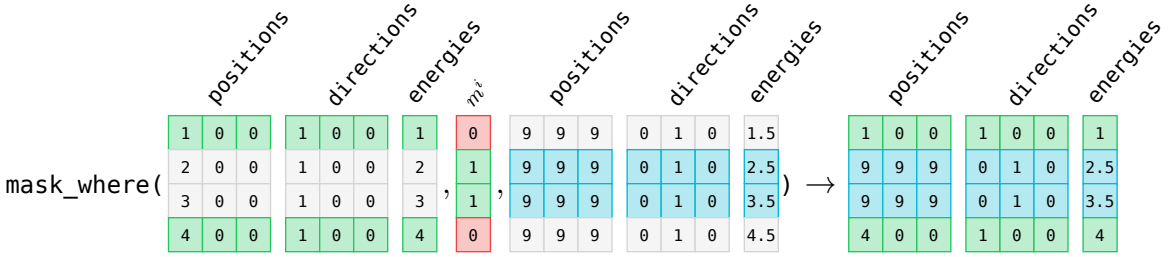


Figure 43: Masked where selects between two ROTTs based on a mask. Rows where $m^i = 0$ come from $\mathbb{A}^i$ (green), rows where $m^i = 1$ come from $\mathbb{B}^i$ (cyan).

5.2.4 `slice`

Slicing extracts a contiguous range of rows from a `Rott`, returning a smaller `Rott`:

$$\mathbb{A}^i[a:b] = \mathbb{A}^{[a,b)} \tag{76}$$

where a and b are integer bounds with $0 \leq a \leq b < |\mathbb{A}^i|$.

```
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [0.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [2.0, 0.0, 0.0],
```

```
            [3.0, 0.0, 0.0],
            [4.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::from_floats(
        [
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
        ],
        device,
    ),
    energies: Tensor::from_floats([1.0, 2.0, 3.0, 4.0, 5.0], device),
};

// P^i[1:4] - extract indices 1, 2, 3
let sliced = particles.slice(1, 4);

assert_eq!(sliced.cardinality(), 3);
assert_eq!(
    sliced.energies.clone().into_data().to_vec::<f32>().unwrap(),
    vec![2.0, 3.0, 4.0]
);
```

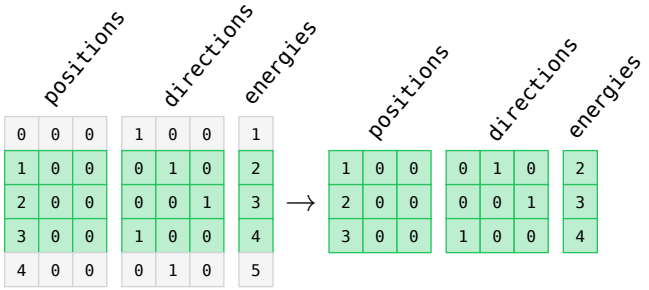


Figure 44: Slicing extracts a contiguous range of rows (indices 1–3 highlighted) into a new ROTT.

5.2.5 `reorder`

Reordering permutes rows according to an index tensor:

$$\text{reorder}(\mathbb{A}^i, \sigma^i) = \mathbb{A}^{\sigma^i} \tag{77}$$

where σ^i is a permutation of indices i.

```
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [2.0, 0.0, 0.0]], device),
    directions: Tensor::from_floats(
        [[1.0, 0.0, 0.0], [1.0, 0.0, 0.0], [1.0, 0.0, 0.0]],
        device,
    ),
    energies: Tensor::from_floats([1.0, 2.0, 3.0], device),
};

// Reorder: reverse the batch
let indices: Tensor<B, 1, Int> = Tensor::from_ints([2, 1, 0], device);
let reordered = particles.reorder(indices);

assert_eq!(reordered.cardinality(), 3);
```

```
assert_eq!(
    reordered
        .energies
        .clone()
        .into_data()
        .to_vec::<f32>()
        .unwrap(),
    vec![3.0, 2.0, 1.0]
);
```

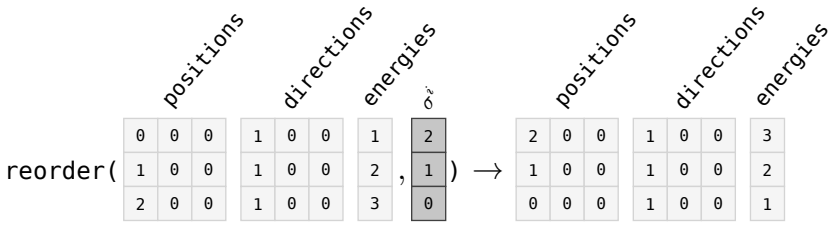


Figure 45: Reordering permutes rows according to an index tensor σ^i, here reversing the order with $\sigma = [2, 1, 0]$.

5.2.6 `concat`

Concatenation joins multiple ROTTs along the batch dimension:

$$\text{concat}(\mathbb{A}^i, \mathbb{B}^i) = \mathbb{C}^i \tag{78}$$

where $|\mathbb{C}^i| = |\mathbb{A}^i| + |\mathbb{B}^i|$.

```
let batch_a: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0]], device),
    directions: Tensor::from_floats([[1.0, 0.0, 0.0], [1.0, 0.0, 0.0]], device),
    energies: Tensor::from_floats([1.0, 2.0], device),
};
let batch_b: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[2.0, 0.0, 0.0], [3.0, 0.0, 0.0], [4.0, 0.0, 0.0]], device),
    directions: Tensor::from_floats(
        [[1.0, 0.0, 0.0], [1.0, 0.0, 0.0], [1.0, 0.0, 0.0]],
        device,
    ),
    energies: Tensor::from_floats([3.0, 4.0, 5.0], device),
};

// concat(A^i, B^i) - combine batches
let combined = ParticleBatch::concat(&[batch_a, batch_b]);

assert_eq!(combined.cardinality(), 5);
assert_eq!(
    combined
        .energies
        .clone()
        .into_data()
        .to_vec::<f32>()
        .unwrap(),
    vec![1.0, 2.0, 3.0, 4.0, 5.0]
);
```

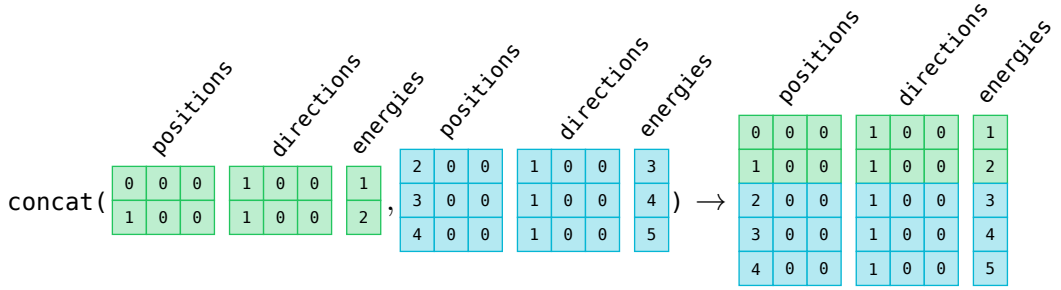


Figure 46: Concatenation joins two ROTTs along the batch dimension. Colors show the origin of each row in the result.

5.2.7 `concat_bounded`

Bounded concatenation combines two ROTTs up to a maximum size, returning any overflow:

$$\text{concat\_bounded}(\mathbb{A}^i, \mathbb{B}^i, n) = (\mathbb{C}^i, \mathbb{R}^i) \tag{79}$$

where $|\mathbb{C}^i| = \min(|\mathbb{A}^i| + |\mathbb{B}^i|, n)$ and $\mathbb{R}^i$ contains any remaining elements.

```rust
let batch_a: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0]], device),
    directions: Tensor::ones([2, 3], device),
    energies: Tensor::from_floats([1.0, 2.0], device),
};
let batch_b: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[2.0, 0.0, 0.0], [3.0, 0.0, 0.0], [4.0, 0.0, 0.0]], device),
    directions: Tensor::ones([3, 3], device),
    energies: Tensor::from_floats([3.0, 4.0, 5.0], device),
};

// Concatenate with max_size=4, overflow goes to remainder
let (merged, remainder) = batch_a.concat_bounded(batch_b, 4);

assert_eq!(merged.cardinality(), 4);
assert!(remainder.is_some());
assert_eq!(remainder.as_ref().unwrap().cardinality(), 1);
```

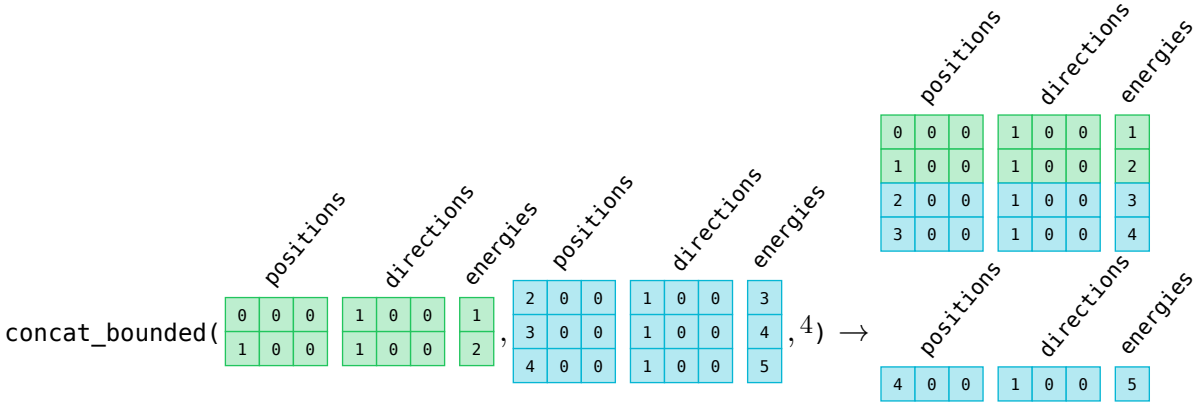


Figure 47: Bounded concatenation joins two ROTTs along the batch dimension, up to a maximum size. Colors show the origin of each row in the result.

5.2.8 `partition_uniform`

Uniform partitioning divides a `Rott` into smaller `Rotts` of at most a specified maximum size:

$$\text{partition\_uniform}(\mathbb{A}^i, n) = \left[\mathbb{A}^\alpha_0, \mathbb{A}^\beta_1, ..., \mathbb{A}^\omega_k\right] \tag{80}$$

where each $|\mathbb{A}^i_j| \leq n$ for all j, and all batches except possibly the last have exactly n elements.

```rust
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [0.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [2.0, 0.0, 0.0],
            [3.0, 0.0, 0.0],
            [4.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::ones([5, 3], device),
    energies: Tensor::from_floats([1.0, 2.0, 3.0, 4.0, 5.0], device),
};

// Partition into batches of at most 2
let batches = particles.partition_uniform(2);

assert_eq!(batches.num_rotts(), 3); // [2, 2, 1]
assert_eq!(batches.rotts()[0].cardinality(), 2);
assert_eq!(batches.rotts()[1].cardinality(), 2);
assert_eq!(batches.rotts()[2].cardinality(), 1);
```

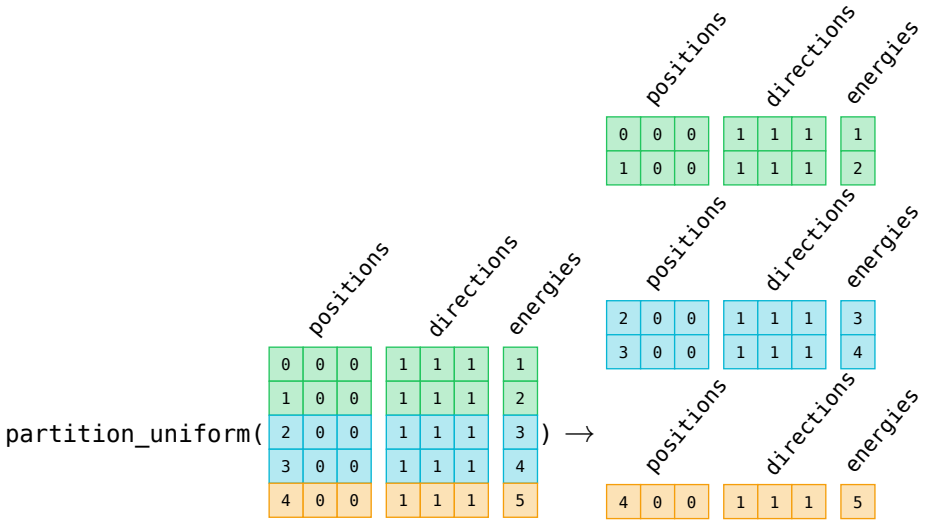


Figure 48: Splitting divides a ROTT into chunks of at most n rows. Colors indicate which chunk each row belongs to.

5.2.9 `partition_by_labels`

Label-based partitioning groups elements by integer labels:

$$\text{partition\_by\_labels}(\mathbb{A}^i, \ell^i) = [\mathbb{A}^i_0, \mathbb{A}^i_1, ..., \mathbb{A}^i_k] \tag{81}$$

where each $\mathbb{A}^i_j$ contains all elements with label j.

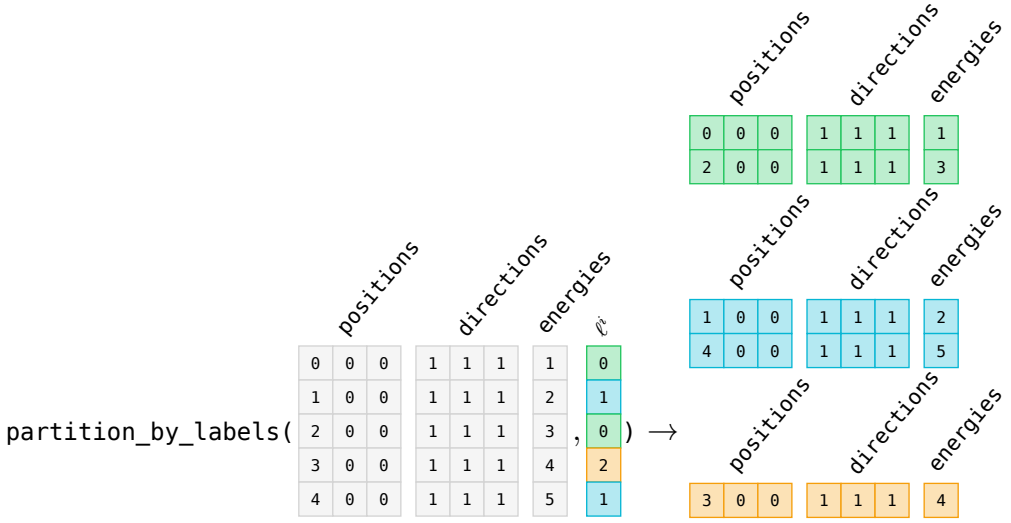


Figure 49: Partitioning by labels groups rows with matching labels into separate ROTTs.

```rust
let particles: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [0.0, 0.0, 0.0],
            [1.0, 0.0, 0.0],
            [2.0, 0.0, 0.0],
            [3.0, 0.0, 0.0],
            [4.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::ones([5, 3], device),
    energies: Tensor::from_floats([1.0, 2.0, 3.0, 4.0, 5.0], device),
};

// Labels: cell IDs for each particle
let cell_ids: Tensor<B, 1, Int> = Tensor::from_ints([0, 1, 0, 2, 1], device);

// Partition by cell ID
let batches = particles.partition_by_labels(cell_ids);

assert_eq!(batches.num_rotts(), 3); // 3 unique labels
```

5.3 The Rotts Struct

The `Rotts` struct provides a collection abstraction over multiple `Rotts`, enabling iteration over the batches.

A `Rotts` wraps a vector of ROTTs and provides parallel iterators and batch operations. Domain-specific collections like `Particles` and `Events` are type aliases for `Rotts`:

```rust
/// A collection of particle batches - type alias for Rotts.
pub type Particles<B> = Rotts<B, ParticleBatch<B>>;
```

Concretely, one can initialize a `Rotts` with a vector of `Rotts`:

```rust
let batch_a: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0], [2.0, 0.0, 0.0]], device),
    directions: Tensor::ones([3, 3], device),
    energies: Tensor::from_floats([1.0, 2.0, 3.0], device),
```

```
};
let batch_b: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats([[3.0, 0.0, 0.0], [4.0, 0.0, 0.0]], device),
    directions: Tensor::ones([2, 3], device),
    energies: Tensor::from_floats([4.0, 5.0], device),
};
let batch_c: ParticleBatch<B> = ParticleBatch {
    positions: Tensor::from_floats(
        [
            [5.0, 0.0, 0.0],
            [6.0, 0.0, 0.0],
            [7.0, 0.0, 0.0],
            [8.0, 0.0, 0.0],
        ],
        device,
    ),
    directions: Tensor::ones([4, 3], device),
    energies: Tensor::from_floats([6.0, 7.0, 8.0, 9.0], device),
};

let particles = Rotts::new(vec![batch_a, batch_b, batch_c]);

assert_eq!(particles.num_rotts(), 3); // 3 batches
assert_eq!(particles.cardinality(), 9); // 9 total particles
```

5.3.1 `map`

The `map` method applies a function to each ROTT in parallel, collecting results:

```
let particles: Particles<B> = Rotts::new(vec![
    ParticleBatch {
        positions: Tensor::from_floats([[0., 0., 0.], [1., 0., 0.], [2., 0., 0.]], device),
        directions: Tensor::ones([3, 3], device),
        energies: Tensor::from_floats([1.0, 2.0, 3.0], device),
    },
    ParticleBatch {
        positions: Tensor::from_floats([[3., 0., 0.], [4., 0., 0.]], device),
        directions: Tensor::ones([2, 3], device),
        energies: Tensor::from_floats([4.0, 5.0], device),
    },
    ParticleBatch {
        positions: Tensor::from_floats(
            [[5., 0., 0.], [6., 0., 0.], [7., 0., 0.], [8., 0., 0.]],
            device,
        ),
        directions: Tensor::ones([4, 3], device),
        energies: Tensor::from_floats([6.0, 7.0, 8.0, 9.0], device),
    },
]);

// Compute total energy per batch (in parallel)
let energies: Vec<f32> =
    particles.map(|batch| batch.energies.clone().sum().into_scalar().elem::<f32>());

// batch_a: 1+2+3=6, batch_b: 4+5=9, batch_c: 6+7+8+9=30
assert_eq!(energies, vec![6.0, 9.0, 30.0]);
```

5.3.2 `filter`

The `filter` method applies a mask-generating function to each ROTT in parallel, selecting elements within each batch:

```
let particles: Particles<B> = Rotts::new(vec![
    ParticleBatch {
        positions: Tensor::from_floats([[0., 0., 0.], [1., 0., 0.], [2., 0., 0.]], device),
        directions: Tensor::ones([3, 3], device),
        energies: Tensor::from_floats([1.0, 2.0, 3.0], device),
    },
    ParticleBatch {
        positions: Tensor::from_floats([[3., 0., 0.], [4., 0., 0.]], device),
        directions: Tensor::ones([2, 3], device),
        energies: Tensor::from_floats([4.0, 5.0], device),
    },
    ParticleBatch {
        positions: Tensor::from_floats(
            [[5., 0., 0.], [6., 0., 0.], [7., 0., 0.], [8., 0., 0.]],
            device,
        ),
        directions: Tensor::ones([4, 3], device),
        energies: Tensor::from_floats([6.0, 7.0, 8.0, 9.0], device),
    },
]);

// Filter particles with energy > 1.5 from each batch (in parallel)
let high_energy = particles.filter(|batch| batch.energies.clone().greater_elem(1.5));

// batch_a: [2,3] -> 2, batch_b: [4,5] -> 2, batch_c: [6,7,8,9] -> 4 = 8 total
assert!(high_energy.is_some());
assert_eq!(high_energy.as_ref().unwrap().cardinality(), 8);
let energies: Vec<Vec<f32>> = high_energy
    .as_ref()
    .unwrap()
    .rotts()
    .iter()
    .map(|b| b.energies.to_data().to_vec::<f32>().unwrap())
    .collect();
assert_eq!(
    energies,
    vec![vec![2.0, 3.0], vec![4.0, 5.0], vec![6.0, 7.0, 8.0, 9.0]]
);
```

6 Roam: Discrete Stochastic Processes

6.1 Stochastic Processes

A discrete stochastic process, $\mathbb{M}(d)$, is a sequence of states indexed by an integer parameter d. This is the process *depth*, or the number of *steps* that have occurred since the initial state, $\mathbb{M}(0)$. The states can be any type: a scalar $m(d)$, vector $m^i(d)$, tensor $M^{ij}(d)$, or more exotic types like `Rott` implementers or `Rotts` collections (see Chapter 5).

These processes can be *memory-less*, meaning $\mathbb{M}(d)$ depends on $\mathbb{M}(d-1)$ and system hyperparameters, **only**. Such processes are called *Markov processes*. Non-Markov processes may depend on the full history $[\mathbb{M}(0), \mathbb{M}(1), ..., \mathbb{M}(d-1)]$, enabling phenomena like reinforcement, lock-in dynamics, or path-dependent behavior.

6.2 The Stepper Trait

The core abstraction in `roam.rs` is the `Stepper` trait. Implementers compute new states `T` from the history of previous states `&[T]`, and a handle to an `Rng`.

```rust
pub trait Stepper<T, R: Rng> {
    type Error;

    /// Compute next state from history `[M(0), ..., M(d)]`.
    fn step(&self, history: &[T], rng: &mut R) -> Result<T, Self::Error>;
}
```

The `step` method enables the discrete stochasticity: given the full history of the process $[\mathbb{M}(0), ..., \mathbb{M}(d)]$ and a randomness source, produce the next state $\mathbb{M}(d+1)$ or signal a terminal condition via the error type.

6.3 Trajectories

The `Trajectory` struct manages a sequence of states with automatic history:

```rust
use rand::SeedableRng;
use rand::rngs::StdRng;
use roam::{Stepper, Trajectory};

struct CountingStepper;

impl<R: Rng> Stepper<usize, R> for CountingStepper {
    type Error = Infallible;

    fn step(&self, history: &[usize], _rng: &mut R) -> Result<usize, Self::Error> {
        Ok(history.len())
    }
}

let mut rng = StdRng::seed_from_u64(42);
let mut traj: Trajectory<usize, StdRng, _> = Trajectory::new(0, CountingStepper);

// Step through the process
traj.step(&mut rng).unwrap();
assert_eq!(traj.depth(), 1);
assert_eq!(*traj.current(), 1);

// Run for a fixed number of steps
traj.run_bounded(10, &mut rng).unwrap();
assert_eq!(traj.depth(), 11);

// Access state history
```

```
assert_eq!(traj.states().len(), 12);
assert_eq!(*traj.state_at(5).unwrap(), 5);
```

A `Trajectory` stores all states internally: $[\mathbb{M}(0), \mathbb{M}(1), ..., \mathbb{M}(d)]$.

6.4 Basic Markov Examples

6.4.1 The Trivial Stepper

The simplest possible `Stepper` does nothing. It always returns the initial state:

$$\mathbb{M}(d) = \mathbb{M}(0) \quad \forall d \tag{82}$$

```
/// A trivial stepper that always returns the initial state.
///
/// This is the simplest possible stepper: $bb(M)(d) = bb(M)(0)$ for all $d$.
pub struct TrivialStepper;

impl<T: Clone, R: Rng> roam::Stepper<T, R> for TrivialStepper {
    type Error = Infallible;

    fn step(&self, history: &[T], _rng: &mut R) -> Result<T, Self::Error> {
        Ok(history.first().unwrap().clone())
    }
}

fn trivial_stepper_example() {
    let mut rng = StdRng::seed_from_u64(42);
    let mut traj: Trajectory<f64, StdRng, _> = Trajectory::new(1.0, TrivialStepper);

    traj.run_bounded(100, &mut rng).unwrap();

    // All states equal the initial value
    assert!(traj.states().iter().all(|&s| s == 1.0));
}
```

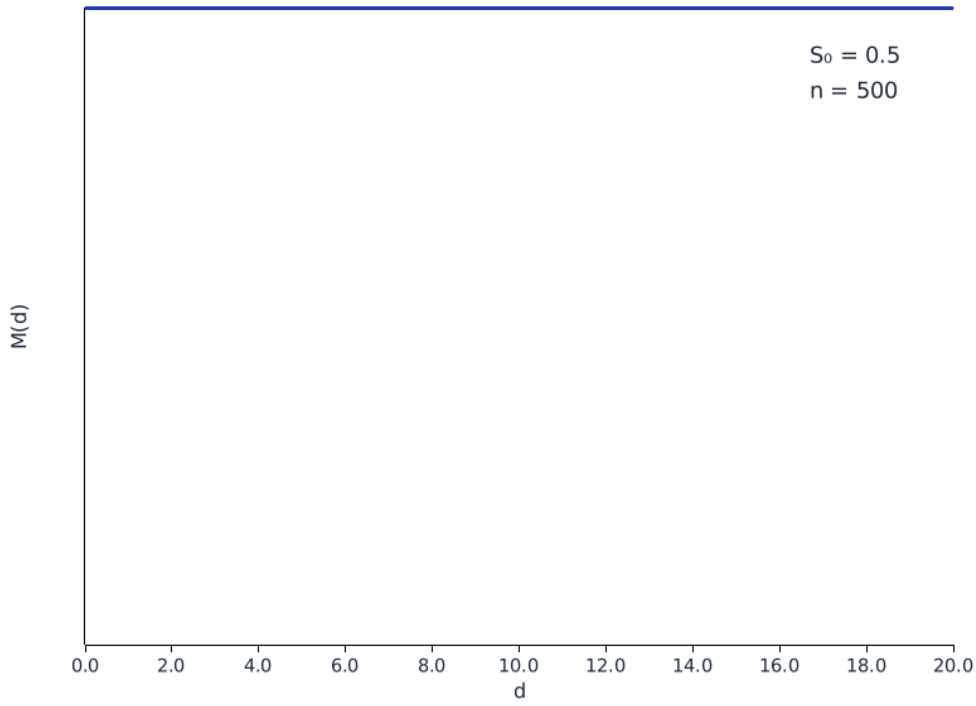


Figure 50: Trivial stepper trajectories. All remain constant at $\mathbb{M}(0)$.

6.4.2 A Counting Stepper

A slightly more interesting example: a counting stepper that returns the current depth:

$$\mathbb{M}(d) = d \tag{83}$$

```rust
/// A counting stepper that returns the current depth.
///
/// At each step, returns $d$ (the number of steps taken).
pub struct CountingStepper;

impl<R: Rng> roam::Stepper<usize, R> for CountingStepper {
    type Error = Infallible;

    fn step(&self, history: &[usize], _rng: &mut R) -> Result<usize, Self::Error> {
        Ok(history.len())
    }
}

fn counting_stepper_example() {
    let mut rng = StdRng::seed_from_u64(42);
    let mut traj: Trajectory<usize, StdRng, _> = Trajectory::new(0, CountingStepper);

    traj.run_bounded(10, &mut rng).unwrap();

    // States are [0, 1, 2, ..., 10]
    assert_eq!(traj.states(), &[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
}
```

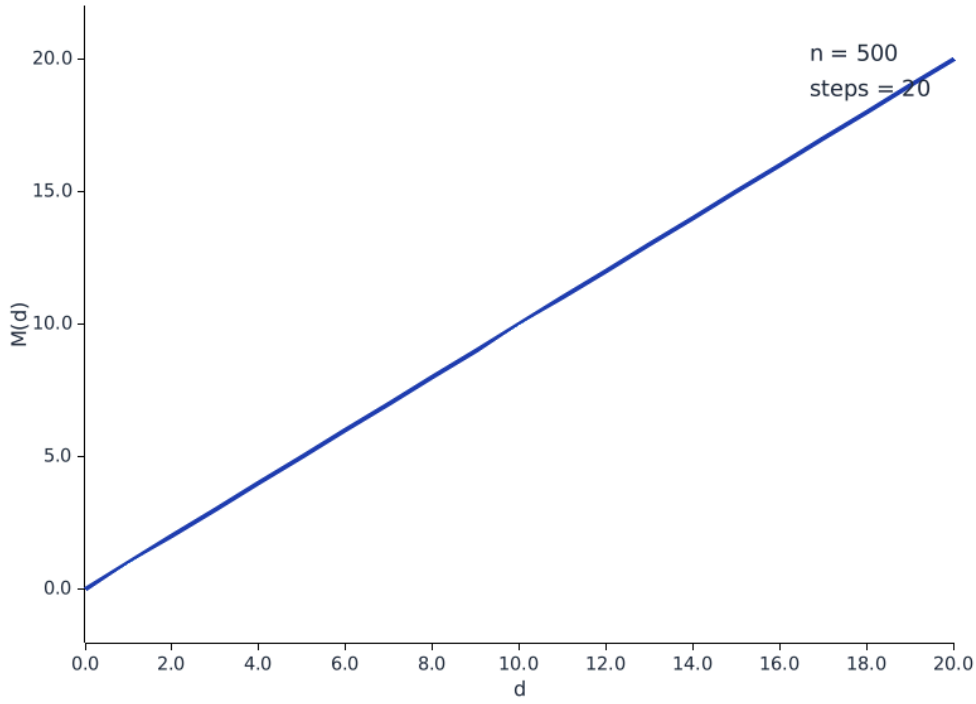


Figure 51: Counting stepper: all trajectories follow the same deterministic path $\mathbb{M}(d) = d$.

6.4.3 Closure-Based Steppers

For quick prototyping, `FnStepper` wraps a closure as a stepper:

```
use rand::SeedableRng;
use rand::rngs::StdRng;
use roam::{Trajectory, stepper};

let step_fn = |history: &[i32], rng: &mut StdRng| {
    let current = *history.last().unwrap();
    Ok::<_, Infallible>(current + rng.random_range(0..=1))
};

let mut rng = StdRng::seed_from_u64(42);
let mut traj = Trajectory::new(0, stepper(step_fn));

traj.run_bounded(10, &mut rng).unwrap();

assert_eq!(traj.depth(), 10);
```

The `stepper()` function is a convenience constructor for `FnStepper`.

6.4.4 Geometric Brownian Motion

As a pedagogical example, consider Geometric Brownian Motion (GBM), a discrete-time stochastic process commonly used to model stock prices and other financial quantities. The process evolves according to:

$$\begin{aligned} s(0) &= s_0 \\ s(d+1) &= s(d) \cdot \exp\left[\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t} z(d)\right] \end{aligned} \tag{84}$$

where s_0 is the initial value, μ is the drift rate, σ is the volatility, Δt is the time step size, and $z(d) \sim \mathcal{N}(0,1)$ is a standard normal random variable.

This is a Markov process; $s(d+1)$ depends only on $s(d)$ and the random draw $z(d)$, not on any earlier history.

We define the process as follows:

```
/// State for geometric Brownian motion.
#[derive(Clone, Debug)]
pub struct GBMState {
    pub value: f64,
    pub time: f64,
}
```

```
/// Standard GBM stepper (Markov).
pub struct GBM {
    pub drift: f64,
    pub volatility: f64,
    pub dt: f64,
}

impl<R: Rng> Stepper<GBMState, R> for GBM {
    type Error = &'static str;

    fn step(&self, history: &[GBMState], rng: &mut R) -> Result<GBMState, Self::Error> {
        let state = history.last().unwrap();

        if state.value <= 0.0 {
            return Err("value became non-positive");
        }

        let z = Normal::new(0.0, 1.0).unwrap().sample(rng);

        // S(t+1) = S(t) * exp((μ - σ²/2)Δt + σ√Δt * Z)
        let drift_term = (self.drift - 0.5 * self.volatility.powi(2)) * self.dt;
        let diffusion_term = self.volatility * self.dt.sqrt() * z;
        let new_value = state.value * (drift_term + diffusion_term).exp();

        Ok(GBMState {
            value: new_value,
            time: state.time + self.dt,
        })
    }
}
```

Simulation is done using the `Trajectory` API:

```
use rand::SeedableRng;
use rand::rngs::StdRng;
use roam::Trajectory;

let mut rng = StdRng::seed_from_u64(42);
let gbm = GBM {
    drift: 0.05,
    volatility: 0.2,
    dt: 1.0 / 252.0, // Daily steps
};

let mut traj: Trajectory<GBMState, StdRng, _> = Trajectory::new(
    GBMState {
        value: 100.0,
```

```
        time: 0.0,
    },
    gbm,
);

// Simulate one trading year (252 days)
traj.run_bounded(252, &mut rng).unwrap();

assert_eq!(traj.depth(), 252);
assert!(traj.current().value > 0.0);
```

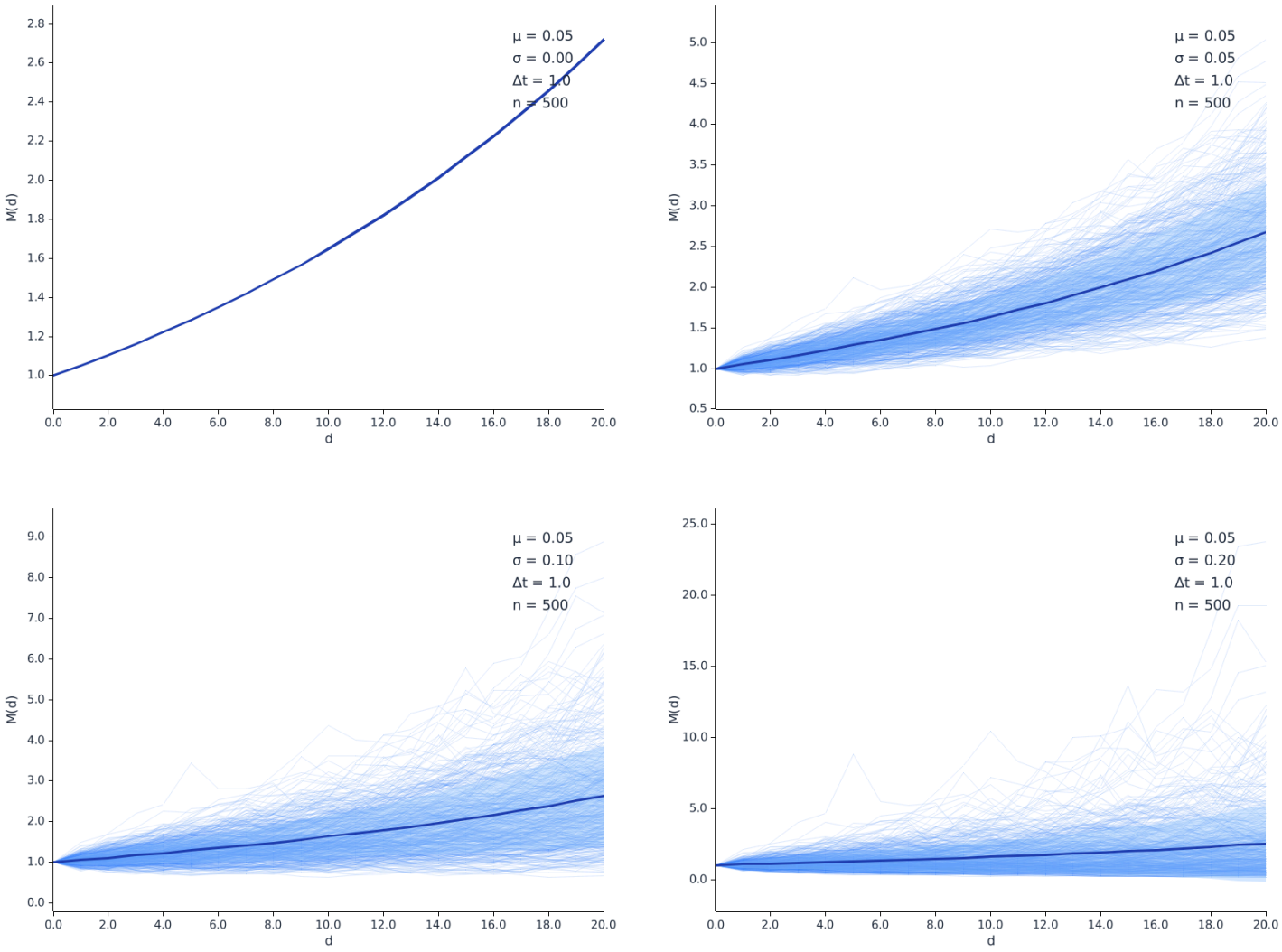


Figure 52: GBM trajectories with varying volatility σ. As volatility increases, the spread of possible outcomes widens dramatically. Each panel shows 500 traces with mean (dark line) and $\pm 1\sigma$ band (shaded).

6.5 Basic Non-Markov Examples

6.5.1 Pólya Urn Model

The Pólya urn is a classic model demonstrating *rich-get-richer* dynamics. An urn starts with one red and one blue ball. At each step:

1. Draw a ball uniformly at random
2. Return it along with one additional ball of the same color

The fraction of red balls converges to a limit, but that limit depends entirely on the random trajectory; different runs lock into different attractors, depending on their early behavior.

Below we define the internal state of the urn:

```rust
/// State for the Pólya urn model.
#[derive(Clone, Debug)]
pub struct UrnState {
    pub red: u32,
    pub blue: u32,
}

impl UrnState {
    pub fn new(red: u32, blue: u32) -> Self {
        Self { red, blue }
    }

    pub fn fraction_red(&self) -> f64 {
        self.red as f64 / (self.red + self.blue) as f64
    }
}
```

Using this, we implement `Stepper`:

```rust
/// Pólya urn stepper (non-Markov).
///
/// Draw a ball proportional to counts, return it with one more of the same color.
pub struct PolyaUrn;

impl<R: Rng> Stepper<UrnState, R> for PolyaUrn {
    type Error = Infallible;

    fn step(&self, history: &[UrnState], rng: &mut R) -> Result<UrnState, Self::Error> {
        let state = history.last().unwrap();
        let total = state.red + state.blue;

        // Draw proportional to current counts
        if rng.random_ratio(state.red, total) {
            Ok(UrnState {
                red: state.red + 1,
                blue: state.blue,
            })
        } else {
            Ok(UrnState {
                red: state.red,
                blue: state.blue + 1,
            })
        }
    }
}
```

Finally, we can sample random trajectories through this space as follows:

```rust
let mut rng = StdRng::seed_from_u64(42);
let mut traj: Trajectory<UrnState, StdRng, _> = Trajectory::new(
    UrnState::new(1, 1), // Start with 1 red, 1 blue
    PolyaUrn,
);

traj.run_bounded(10, &mut rng).unwrap();
```

Extending into deeper depths, we see the convergent behavior emerge:

```rust
let mut rng = StdRng::seed_from_u64(42);
let mut traj: Trajectory<UrnState, StdRng, _> = Trajectory::new(
    UrnState::new(1, 1), // Start with 1 red, 1 blue
    PolyaUrn,
);

traj.run_bounded(100, &mut rng).unwrap();

// Fraction converges to some limit (path-dependent!)
let final_fraction = traj.current().fraction_red();
assert!(final_fraction > 0.0 && final_fraction < 1.0);
```

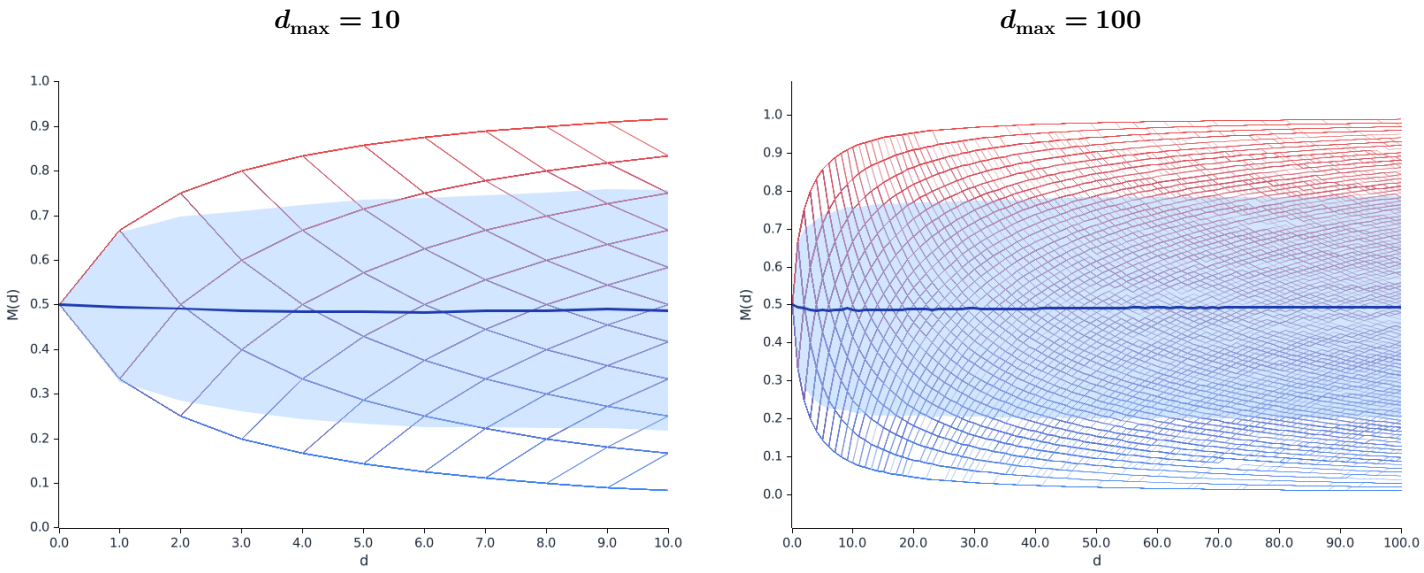


Figure 53: 500 Pólya urn trajectories showing fraction of red balls over time. Left: in early depths, samples greatly inflnence long-term behavior. Right: trajectories converge to common attractors. Notice that the mean remains 0.5 and the variance stabilizes.

6.6 Tensor States: The Ising Model

The Ising model demonstrates how `roam.rs` handles tensor-valued states using Burn `Tensors` as states. In the Ising model, a 2D lattice of spins evolves via *Metropolis dynamics*. The lattice is defined as:

$$S^{xy} = \{-1, +1\} \forall (x, y) \tag{85}$$

Each location in the lattice is *spin up* or *spin down*. Each lattice location randomly updates according to the spins of its neighbors.

6.6.1 Standard Ising Model (Markov)

The *energy* of a standard Ising model configuration is:

$$E = -J \sum_{\langle i,j \rangle} s_i s_j \tag{86}$$

where the sum runs over nearest-neighbor pairs and J is the coupling constant. The *Metropolis* algorithm proposes spin flips and accepts them with probability:

$$P(\text{accept}) = \min(1, \exp(-\Delta E/T)) \tag{87}$$

The state of the Ising model is implemented as below:

```rust
/// State for 2D Ising spin lattice using burn tensors.
///
/// Spins are stored as a 2D tensor with values +1 or -1 (as floats for GPU ops).
#[derive(Clone)]
pub struct SpinLattice<B: Backend> {
    pub spins: Tensor<B, 2>,
    pub size: usize,
}

impl<B: Backend> SpinLattice<B> {
    /// Create a new lattice with random spins.
    pub fn random(size: usize, device: &B::Device) -> Self {
        // Generate random values in [0, 1), convert to ±1
        let random = Tensor::<B, 2>::random(
            [size, size],
            burn::tensor::Distribution::Uniform(0.0, 1.0),
            device,
        );
        let threshold = Tensor::<B, 2>::full([size, size], 0.5, device);
        let mask = random.lower(threshold);
        let ones = Tensor::<B, 2>::ones([size, size], device);
        let neg_ones = ones.clone().neg();
        let spins = mask.clone().float() * neg_ones + mask.bool_not().float() * ones;
        Self { spins, size }
    }

    /// Create a lattice with all spins up.
    pub fn all_up(size: usize, device: &B::Device) -> Self {
        let spins = Tensor::<B, 2>::ones([size, size], device);
        Self { spins, size }
    }

    /// Compute total magnetization.
    pub fn magnetization(&self) -> f32 {
        self.spins.clone().sum().into_scalar().elem()
    }

    /// Compute magnetization per spin.
    pub fn magnetization_per_spin(&self) -> f64 {
        self.magnetization() as f64 / (self.size * self.size) as f64
    }

    /// Convert to Vec<Vec<i8>> for visualization.
    pub fn to_vec(&self) -> Vec<Vec<i8>> {
        let data = self.spins.clone().into_data();
        let flat: Vec<f32> = data.to_vec().unwrap();
        flat.chunks(self.size)
            .map(|row| {
                row.iter()
                    .map(|&v| if v > 0.0 { 1i8 } else { -1i8 })
                    .collect()
            })
            .collect()
    }

    /// Get the device this lattice is on.
    pub fn device(&self) -> B::Device {
        self.spins.device()
    }
}
```

The `Stepper` uses `conv2d` for efficient neighbor sum computation and checkerboard updates to avoid updating neighbors inconsitently across the lattice:

```
/// Standard Ising model stepper using burn tensors (Markov).
///
/// Uses parallel Metropolis updates on the full lattice each step.
/// Neighbor sums computed via conv2d for GPU efficiency.
pub struct IsingModel<B: Backend> {
    pub temperature: f64,
    pub coupling: f64,
    _phantom: PhantomData<B>,
}

impl<B: Backend> IsingModel<B> {
    pub fn new(temperature: f64, coupling: f64) -> Self {
        Self {
            temperature,
            coupling,
            _phantom: PhantomData,
        }
    }

    /// Compute sum of four neighbors using conv2d with a cross kernel.
    /// The kernel [[0,1,0], [1,0,1], [0,1,0]] sums the 4 neighbors.
    pub fn neighbor_sum(spins: &Tensor<B, 2>, device: &B::Device) -> Tensor<B, 2> {
        use burn::tensor::module::conv2d;
        use burn::tensor::ops::ConvOptions;

        let [rows, cols] = spins.dims();

        // Cross kernel: sums up, down, left, right neighbors
        let kernel_data: [f32; 9] = [0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0];
        // [out_channels, in_channels, h, w]
        let kernel = Tensor::<B, 1>::from_floats(kernel_data, device).reshape([1, 1, 3, 3]);

        // Use circular padding by manually wrapping edges
        let top = spins.clone().narrow(0, rows - 1, 1);
        let bottom = spins.clone().narrow(0, 0, 1);
        let padded_v = Tensor::cat(vec![top, spins.clone(), bottom], 0);

        let left = padded_v.clone().narrow(1, cols - 1, 1);
        let right = padded_v.clone().narrow(1, 0, 1);
        let padded = Tensor::cat(vec![left, padded_v, right], 1);

        let padded_4d = padded.reshape([1, 1, rows + 2, cols + 2]);

        // Conv2d with no padding (we already padded manually for circular boundaries)
        let options = ConvOptions::new([1, 1], [0, 0], [1, 1], 1);
        let result = conv2d(padded_4d, kernel, None, options);

        result.reshape([rows, cols])
    }

    /// Generate random tensor from external RNG for reproducibility.
    pub fn random_tensor<R: Rng>(size: usize, rng: &mut R, device: &B::Device) -> Tensor<B, 2> {
        let data: Vec<f32> = (0..size * size).map(|_| rng.random::<f32>()).collect();
        Tensor::<B, 1>::from_floats(&data[..], device).reshape([size, size])
    }

    /// Create a checkerboard mask for the given parity.
    pub fn checkerboard_mask(size: usize, parity: bool, device: &B::Device) -> Tensor<B, 2> {
        let mut data = vec![0.0f32; size * size];
```

```
        for i in 0..size {
            for j in 0..size {
                if ((i + j) % 2 == 0) == parity {
                    data[i * size + j] = 1.0;
                }
            }
        }
        Tensor::<B, 1>::from_floats(&data[..], device).reshape([size, size])
    }
}
```

The `Stepper` implementation performs the Metropolis updates:

```
impl<B: Backend, R: Rng> Stepper<SpinLattice<B>, R> for IsingModel<B> {
    type Error = Infallible;

    fn step(&self, history: &[SpinLattice<B>], rng: &mut R) -> Result<SpinLattice<B>, Self::Error> {
        let state = history.last().unwrap();
        let device = state.device();
        let mut spins = state.spins.clone();

        // Checkerboard decomposition: update black squares, then white squares
        // This ensures correct parallel Metropolis (neighbors don't change during update)
        for parity in [true, false] {
            // Compute neighbor sums
            let neighbors = Self::neighbor_sum(&spins, &device);

            // Energy change if we flip: ΔE = 2 * J * s_i * Σ neighbors
            let delta_e = spins
                .clone()
                .mul_scalar(2.0 * self.coupling as f32)
                .mul(neighbors);

            // Acceptance probability: min(1, exp(-ΔE/T))
            let neg_delta_e_over_t = delta_e.neg().div_scalar(self.temperature as f32);
            let accept_prob = neg_delta_e_over_t.exp().clamp_max(1.0);

            // Generate random numbers
            let random = Self::random_tensor(state.size, rng, &device);

            // Accept where random < accept_prob
            let accept_mask = random.lower(accept_prob);

            // Apply checkerboard mask (only update one color)
            let checker = Self::checkerboard_mask(state.size, parity, &device);
            let update_mask = accept_mask.float().mul(checker).bool();

            // Flip accepted spins
            let flipped = spins.clone().neg();
            spins = update_mask.clone().float() * flipped + update_mask.bool_not().float() * spins;
        }

        Ok(SpinLattice {
            spins,
            size: state.size,
        })
    }
}
```

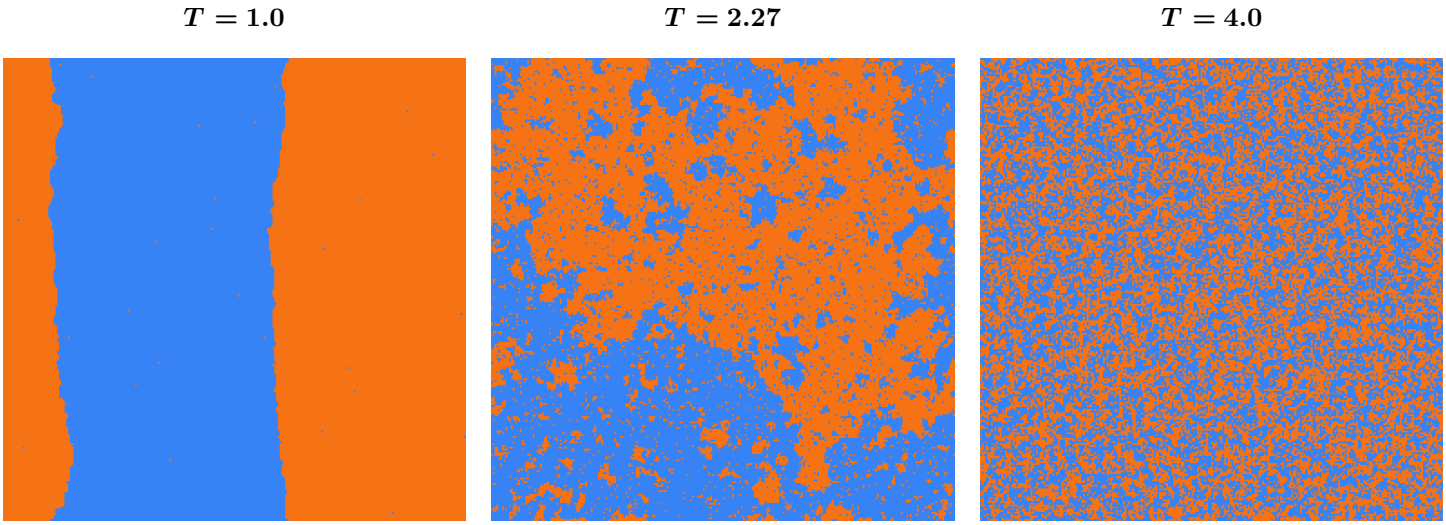


Figure 54: Ising model at different temperatures. Left: ordered phase with large coherent domains. Center: critical point with fractal structure. Right: disordered phase with random spins.

6.6.2 Magnetic Memory Ising (Non-Markov)

To demonstrate non-Markov dynamics with tensor states, we extend the Ising model with *magnetic memory*. The flip acceptance probability is biased by the historical average magnetization:

$$P(\text{accept}) = \min\Big(1, \exp\Big(-\Big(\Delta E - w\overline{M}s'\Big)/T\Big)\Big) \tag{88}$$

where $\overline{M}$ is the average magnetization over recent history and w is the memory weight. This creates "momentum" in the magnetization dynamics; the system resists flipping away from its historical trend.

```rust
/// Ising model with magnetic memory using burn tensors (non-Markov).
///
/// Flip acceptance depends on historical magnetization, creating "momentum".
/// Spins that align with the historical magnetization trend are favored.
pub struct MagneticMemoryIsing<B: Backend> {
    pub temperature: f64,
    pub coupling: f64,
    pub memory_weight: f64,
    _phantom: PhantomData<B>,
}

impl<B: Backend> MagneticMemoryIsing<B> {
    pub fn new(temperature: f64, coupling: f64, memory_weight: f64) -> Self {
        Self {
            temperature,
            coupling,
            memory_weight,
            _phantom: PhantomData,
        }
    }
}
```

The `Stepper` implementation biases acceptance by historical magnetization:

```rust
impl<B: Backend, R: Rng> Stepper<SpinLattice<B>, R> for MagneticMemoryIsing<B> {
    type Error = Infallible;

    fn step(&self, history: &[SpinLattice<B>], rng: &mut R) -> Result<SpinLattice<B>, Self::Error> {
        let state = history.last().unwrap();
```

```
        let device = state.device();
        let mut spins = state.spins.clone();

        // Compute historical average magnetization (last 10 states or all if fewer)
        let window_start = history.len().saturating_sub(10);
        let avg_mag: f64 = history[window_start..]
            .iter()
            .map(|s| s.magnetization_per_spin())
            .sum::<f64>()
            / (history.len() - window_start) as f64;

        // Checkerboard decomposition for correct parallel Metropolis
        for parity in [true, false] {
            // Compute neighbor sums
            let neighbors = IsingModel::<B>::neighbor_sum(&spins, &device);

            // Energy change if we flip: ΔE = 2 * J * s_i * Σ neighbors
            let delta_e = spins
                .clone()
                .mul_scalar(2.0 * self.coupling as f32)
                .mul(neighbors);

            // Proposed spins (flipped)
            let proposed = spins.clone().neg();

            // Memory bias: favor flips that align with historical magnetization trend
            let memory_bias = proposed
                .clone()
                .mul_scalar(self.memory_weight as f32 * avg_mag as f32);

            // Effective energy change
            let effective_delta_e = delta_e.sub(memory_bias);

            // Acceptance probability: min(1, exp(-ΔE_eff/T))
            let neg_delta_e_over_t = effective_delta_e.neg().div_scalar(self.temperature as f32);
            let accept_prob = neg_delta_e_over_t.exp().clamp_max(1.0);

            // Generate random numbers
            let random = IsingModel::<B>::random_tensor(state.size, rng, &device);

            // Accept where random < accept_prob
            let accept_mask = random.lower(accept_prob);

            // Apply checkerboard mask
            let checker = IsingModel::<B>::checkerboard_mask(state.size, parity, &device);
            let update_mask = accept_mask.float().mul(checker).bool();

            // Flip accepted spins
            spins = update_mask.clone().float() * proposed + update_mask.bool_not().float() * spins;
        }

        Ok(SpinLattice {
            spins,
            size: state.size,
        })
    }
}
```

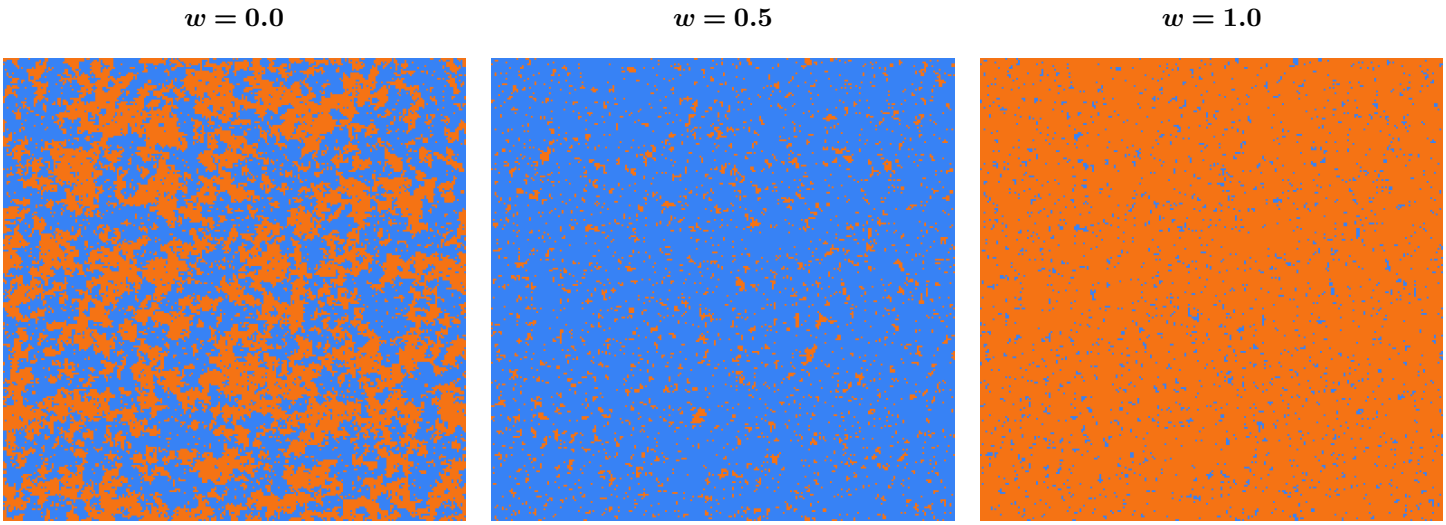


Figure 55: Magnetic memory Ising at $T = 2.5$ with varying memory weight w. Left: no memory (standard Ising). Center/ Right: increasing memory strength creates more coherent domain structures as the system develops "momentum" in its magnetization.

6.7 Error-Based Termination

Steppers can signal terminal conditions by returning an error. The `run()` method continues until an error is returned:

```
struct TerminatingStepper { max_depth: usize }

impl<R: Rng> Stepper<usize, R> for TerminatingStepper {
    type Error = &'static str;

    fn step(&self, history: &[usize], _rng: &mut R) -> Result<usize, Self::Error> {
        if history.len() >= self.max_depth {
            Err("max depth reached")
        } else {
            Ok(history.len())
        }
    }
}

// Usage
let mut traj = Trajectory::new(0, TerminatingStepper { max_depth: 100 });
let result = traj.run(&mut rng);
assert!(result.is_err());
assert_eq!(traj.depth(), 99);
```

This pattern is useful for:

- Absorbing states (particle absorbed, process terminates)
- Boundary conditions (value exceeds threshold)
- Convergence criteria (change falls below tolerance)

7 Phlux Viewer

The Phlux Viewer is a GPU-accelerated visualization application for exploring particle transport simulation results stored in `.phlux` archives.

7.1 Loading `.phlux` Files

Archives are loaded through the phlux API:

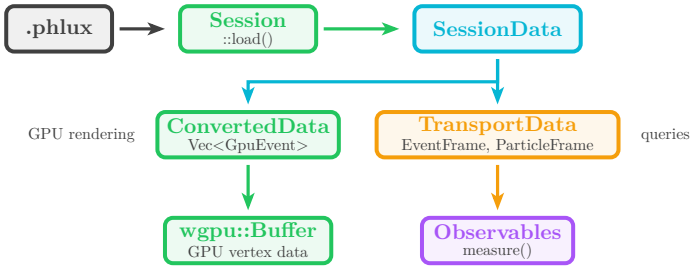


Figure 56: Loading pipeline: `.phlux` → `Session` → `SessionData` → `TransportData` → observables.

`Session::load()` extracts the archive to a temp directory, reads the manifest, and loads Parquet data into Polars DataFrames. `SessionData` provides access to transport data by ID. `TransportData` exposes `EventFrame` and `ParticleFrame` for querying, and the `measure()` method for computing observables.

7.2 User Interface

The viewer displays a 3D scene with a sidebar of interactive charts:

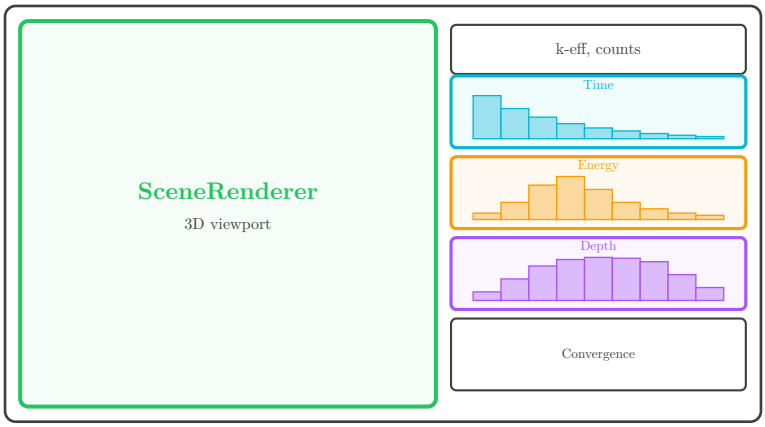


Figure 57: UI layout: 3D viewport with histogram sidebar. Each `HistogramChart` has a `RangeSelection` overlay for filtering.

3D Viewport — `SceneRenderer` renders particle tracks and collision events. Filtering happens in WGSL shaders via `GpuFilterSettings` uniforms.

Histogram Charts — Three `HistogramChart` widgets display time, energy, and depth distributions. Each chart has a `RangeSelection` overlay that users drag to define filter bounds.

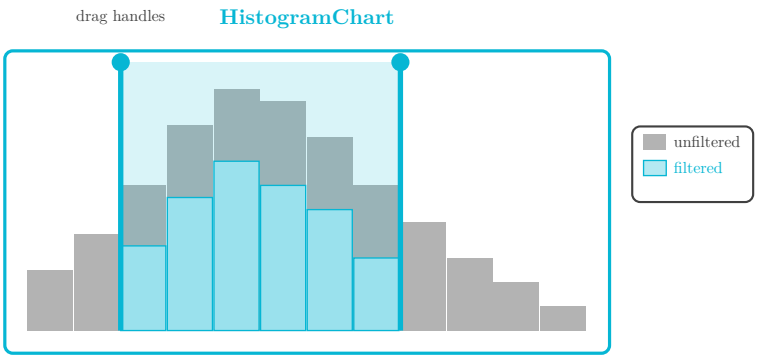


Figure 58: Dual-histogram rendering: background shows full distribution, foreground shows events passing all cross-filters. Foreground bars are shorter due to filtering from other histograms.

Each histogram renders two layers: a light background showing the full distribution, and a colored foreground showing events passing all active filters. Because filters cross-apply, the foreground bars may be shorter than the background even within the selection bounds—events excluded by other histograms (e.g., energy filter) reduce counts in this histogram (e.g., time). This dual-layer approach lets users see both the overall data shape and the effect of their combined filter selections.

Convergence Chart — `ConvergenceChart` displays k-effective and Shannon entropy versus collision depth.

7.3 Filter Bifurcation

`ViewerFilter` is the single source of truth for filter state. It bifurcates to two consumers:

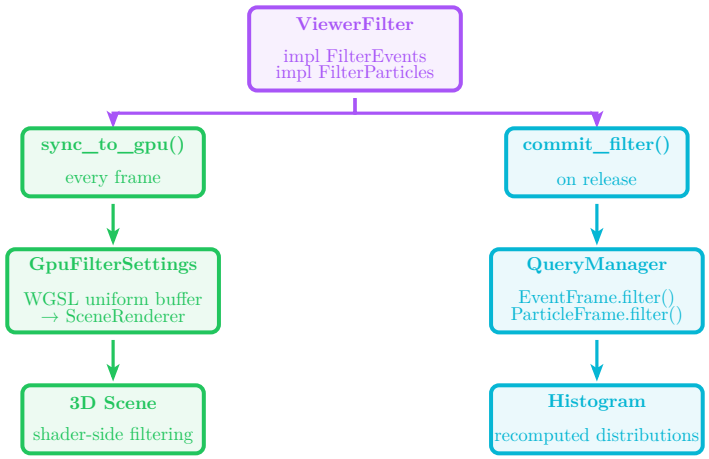


Figure 59: Filter bifurcation: `ViewerFilter` feeds both GPU rendering (immediate) and backend queries (debounced).

GPU Path — `sync_to_gpu()` converts `ViewerFilter` to `GpuFilterSettings` and uploads to the uniform buffer every frame during drag. WGSL shaders discard fragments outside filter bounds for immediate visual feedback.

Backend Path — `commit_filter()` triggers `QueryManager` to recompute histograms on selection release. The query applies `ViewerFilter` via `EventFrame.filter()` and `ParticleFrame.filter()` (Polars lazy expressions), then bins results into `Histogram` distributions.

8 Appendix

8.1 Citation

When using `phlux.rs` in academic work, please cite:

```
@software{phlux.rs,
  title={phlux.rs: Monte Carlo N-Particle Transport Framework},
  author={Huibregtse, Clyde},
  year={2026},
  url={https://gitlab.com/soho-labs/phlux.rs}
}
```

8.2 Contributing

8.2.1 Quick Start

Install developer tools:

```
cargo install --path phlux-tools
```

8.2.2 Commit Message Format

We use Conventional Commits with the format:

```
type(scope): description
```

8.2.2.1 Examples

```
feat(phlux): add particle tracking system
fix(crater): resolve memory leak in mesh generation
docs(xs): update API documentation
perf(phlux): optimize particle tracking loop
refactor(workspace): consolidate shared dependencies
```

8.2.3 Development Workflow

Before committing:

```
# Format code
cargo fmt --all

# Run linting
cargo clippy --workspace --all-targets -- -D warnings

# Validate a commit message
echo "feat(phlux): add new feature" | phlux-tools commit-check --stdin
```

8.2.4 Getting Help

- View phlux-tools commands: `phlux-tools --help`
- View commit schema: `phlux-tools commit-schema --show`
- Report issues: https://gitlab.com/soho-labs/phlux.rs/-/issues