HYBRILIT WORKSHOP 2026

NDMSPC: Exploring Multi-Dimensional Space Analysis

pronounced **“ndm-space”** *— n-dimensional space*

JINR, Dubna

Dedicated to 70 years of JINR and 60 years of MLIT

Martin Vala

Institute of Physics · Faculty of Science
Pavol Jozef Šafárik University in Košice

Štefan Korečko

LIRKIS laboratory · Dept. of Computers and Informatics, FEEI
Technical University of Košice, Slovakia

Outline

THE TALK

1. **The problem** — n-dimensional data at TB/PB scale
2. **From histograms to sparse bins** — why dense grids fail
3. **The core** — `THnSparse` mapping + `TTree` storage, binnings, one function per bin
4. **Working with a bin** — projections, ranges, parameters
5. **Running it** — `single` / `ipc` / `tcp`, and what that means on a batch system
6. **In practice** — systematics as axes, imports

The first half is background — what a histogram really is, and why a sparse, n-dimensional store changes what you can ask of it.

THE SOFTWARE

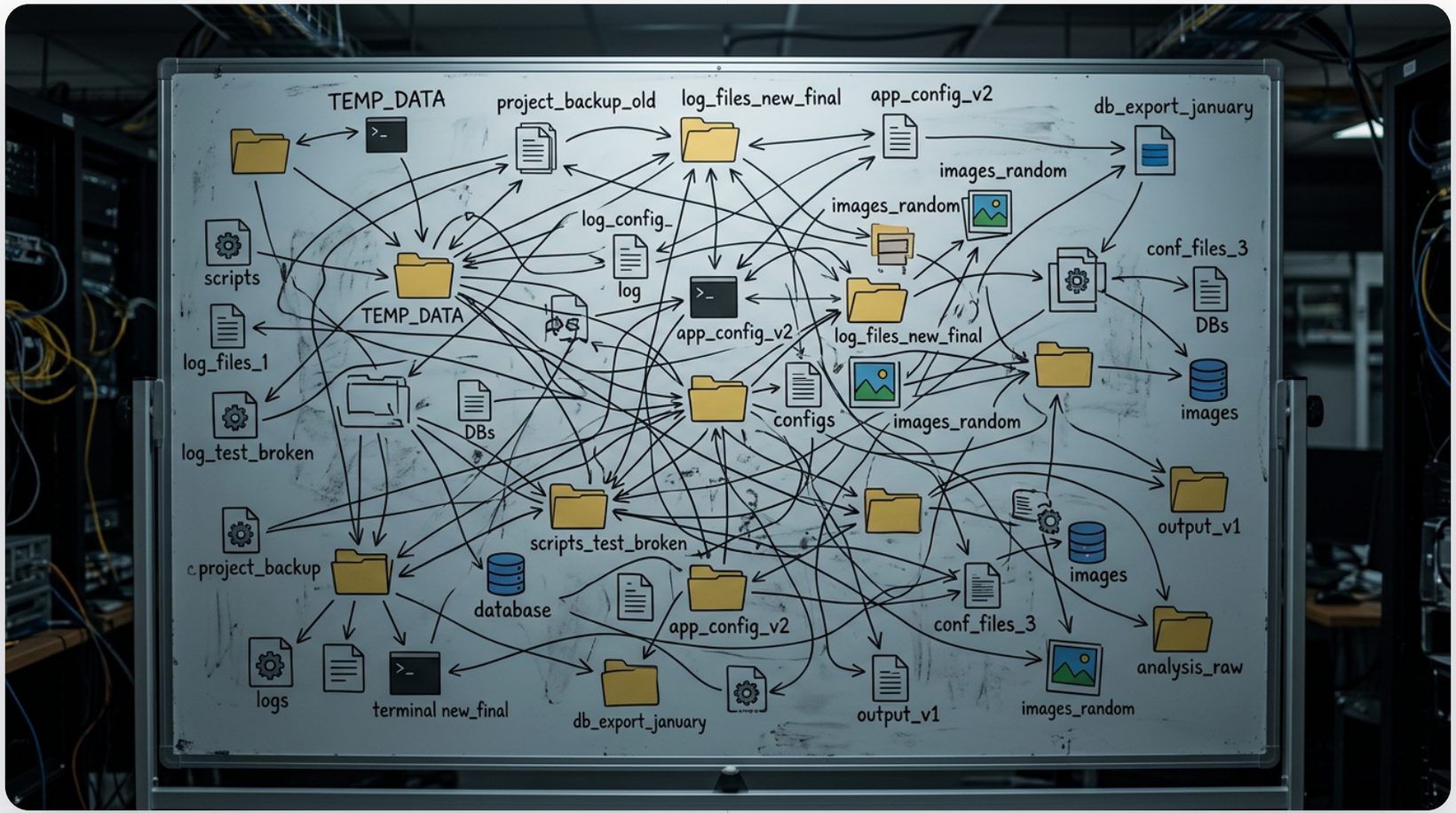
NDMSPC — a ROOT extension for N-dimensional, sparse analysis: many binnings over one dataset, a process function per bin, and results stored with the bin they belong to.

NDMVR — its 3D and VR counterpart: the same data, drawn in the browser with JSROOT and in VR, where higher dimensions can actually be looked at.

One dataset, one process function, and the same code from a laptop to a cluster — that is the thread through the next thirty minutes.

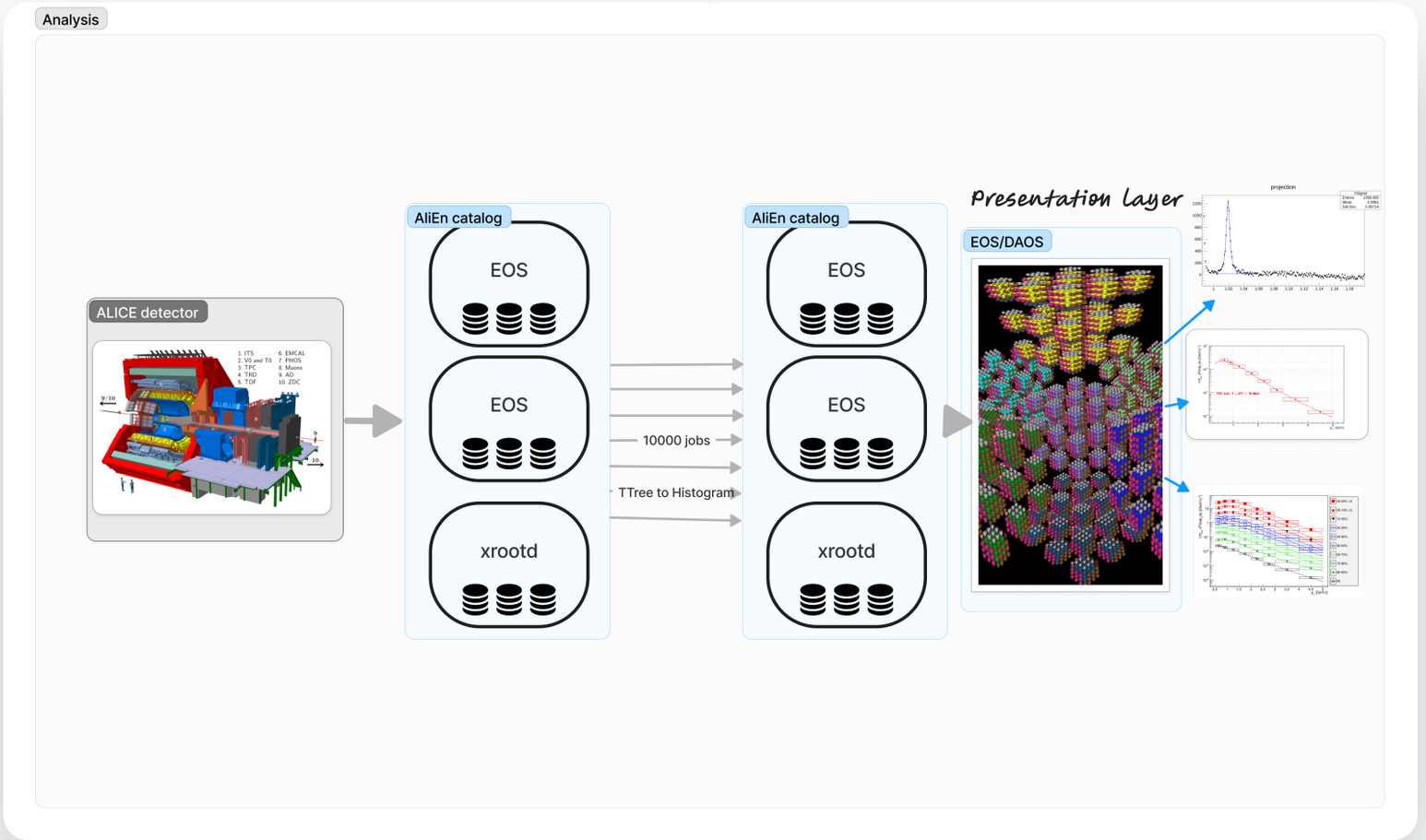
The data we have to keep

Many observables per event, every one of them needing to stay usable.



NDMSPC and EOS to rescue

Sparse storage on object storage — TB to PB scale, without allocating the full grid.

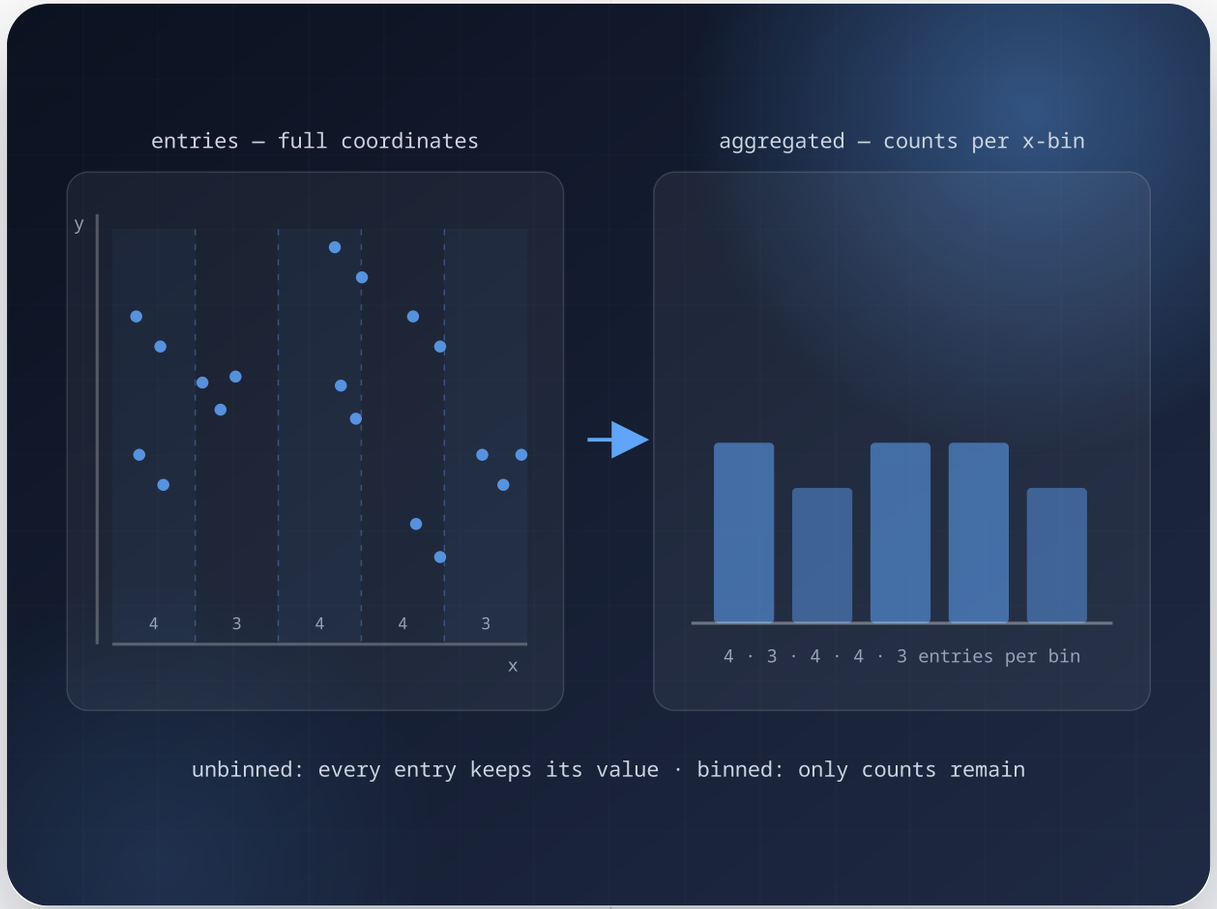


Unbinned data is an idealization

Keeping every entry with its full coordinates is the conceptual baseline.

- Each entry keeps its own measured values — no grouping
- The natural output of detectors and simulations
- No aggregation, no information lost (conceptually)
- Digital storage breaks the ideal: every number has finite resolution

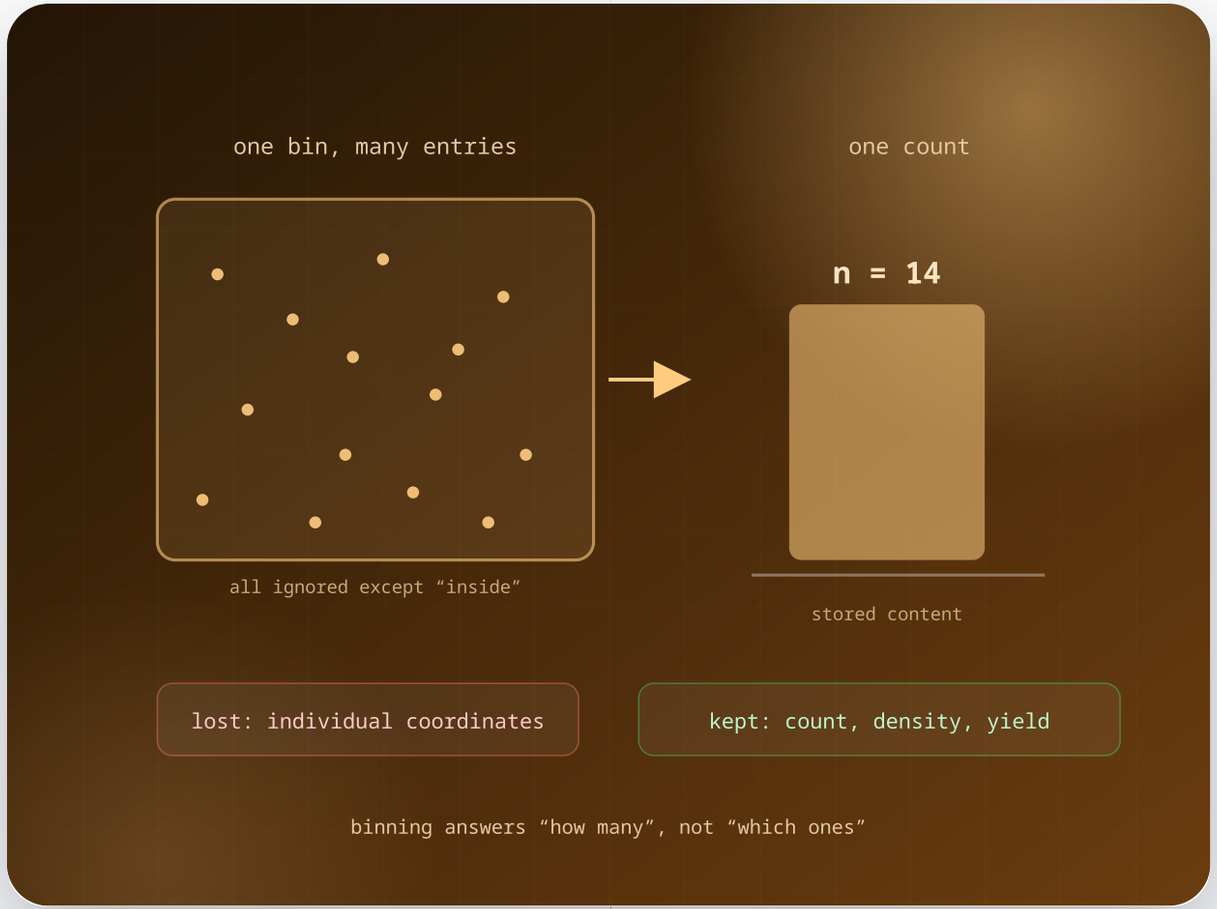
“Unbinned” describes the intent, not the bits.



Binning is lossy

Aggregation replaces coordinates with counts.

- Everything inside a bin becomes one number
- Individual coordinates are lost — irreversibly
- Counts, densities and yields are preserved
- Coarse bins hide structure; fine bins cost statistics
- The trade-off is part of the analysis choice

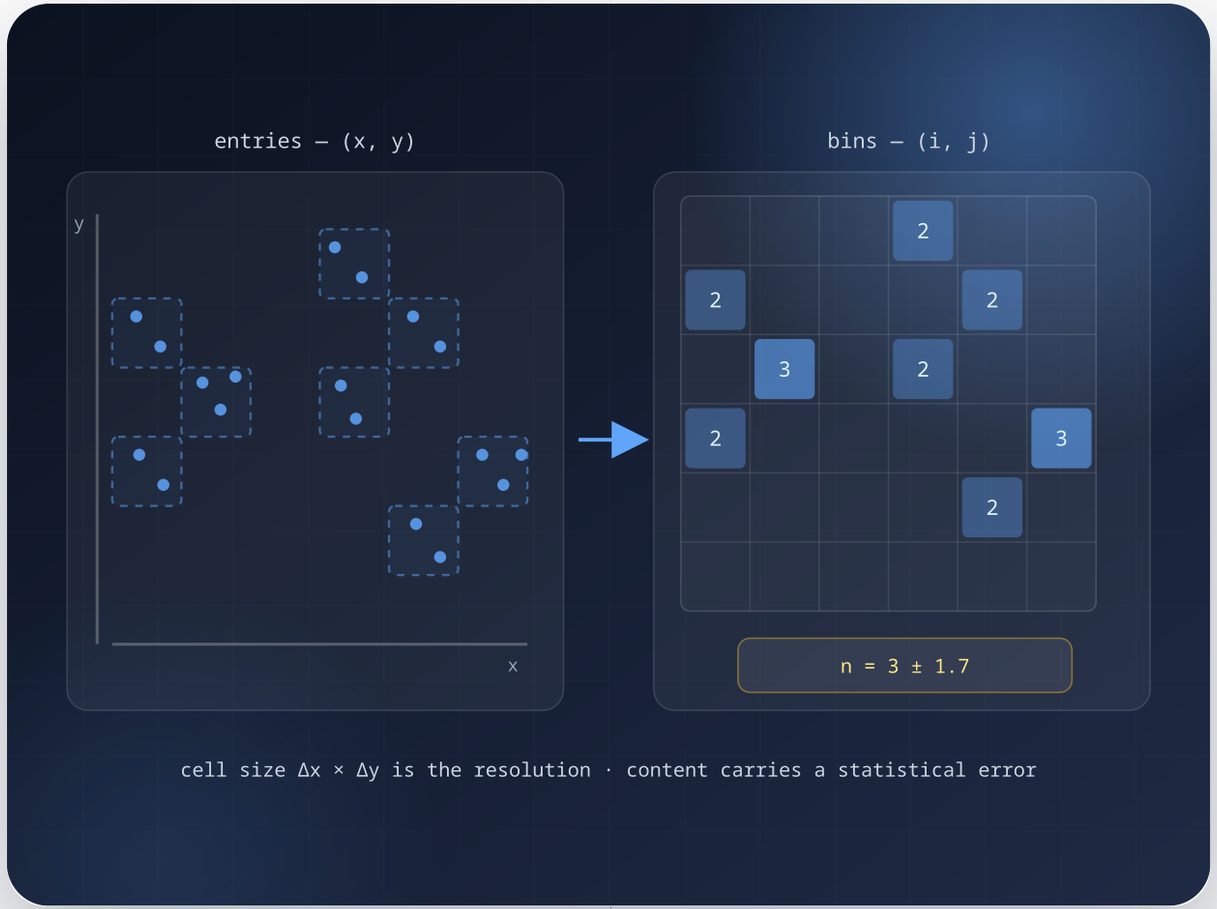


The same data in two dimensions

A second axis turns bins into cells — each cell has a value and an error.

- Two observables → cells `(i, j)` instead of bins `i`
- Content n_{ij} = count or sum of weights
- Statistical error $\sigma_{ij} \approx \sqrt{n_{ij}}$ (or from the weights)
- Cell size $\Delta x \times \Delta y$ is the resolution
- Empty cells are meaningful — that is sparsity

The histogram becomes a grid of numbers ± errors.

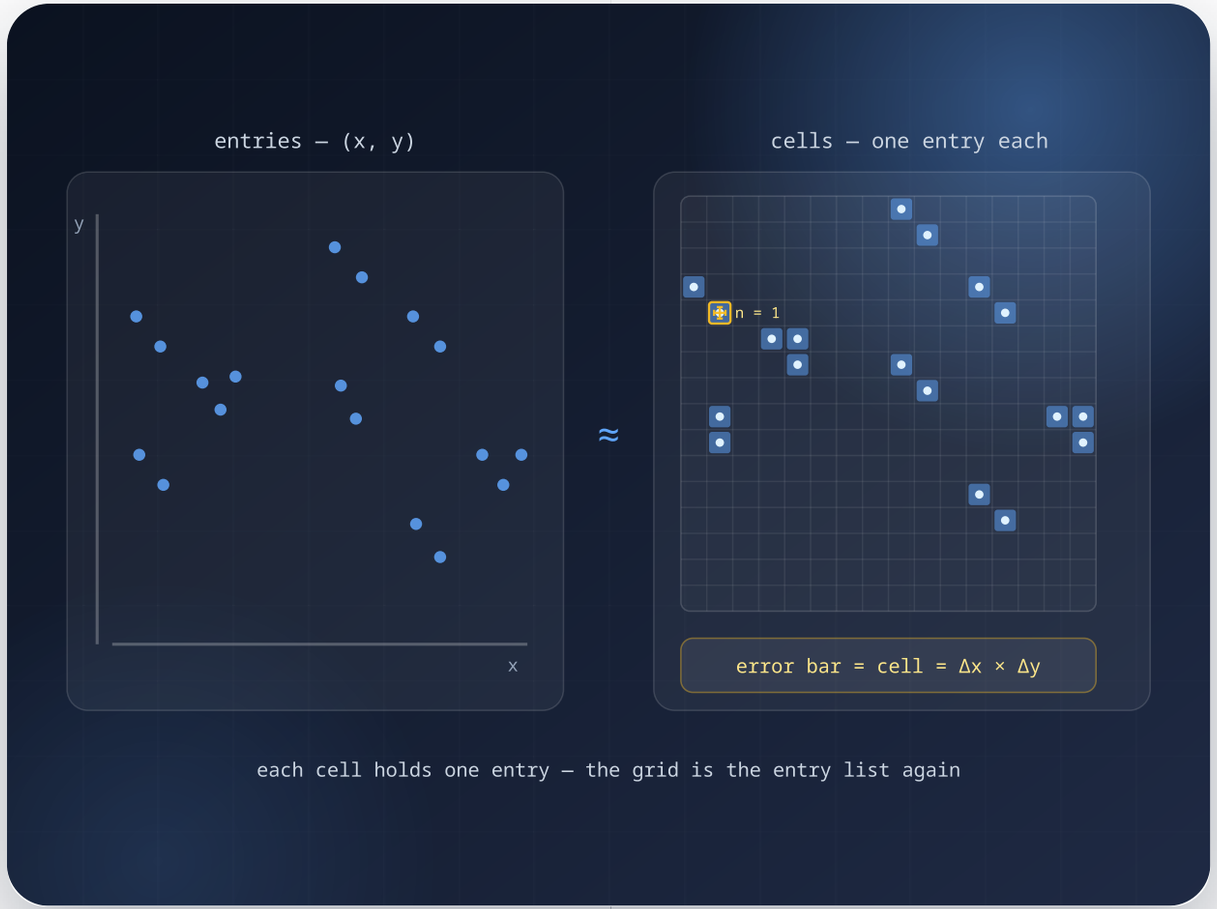


One entry per bin — unbinned again

If no cell holds more than one entry, binning has nothing left to aggregate.

- Each cell holds at most one entry
- Set the cell size to the data resolution Δx, Δy
- Content is 1; the error bar is just the cell
- The grid reproduces the scatter at that resolution
- Binning only loses information when cells are coarser than Δx

The limiting case: histogram ≡ entry list, up to the resolution.

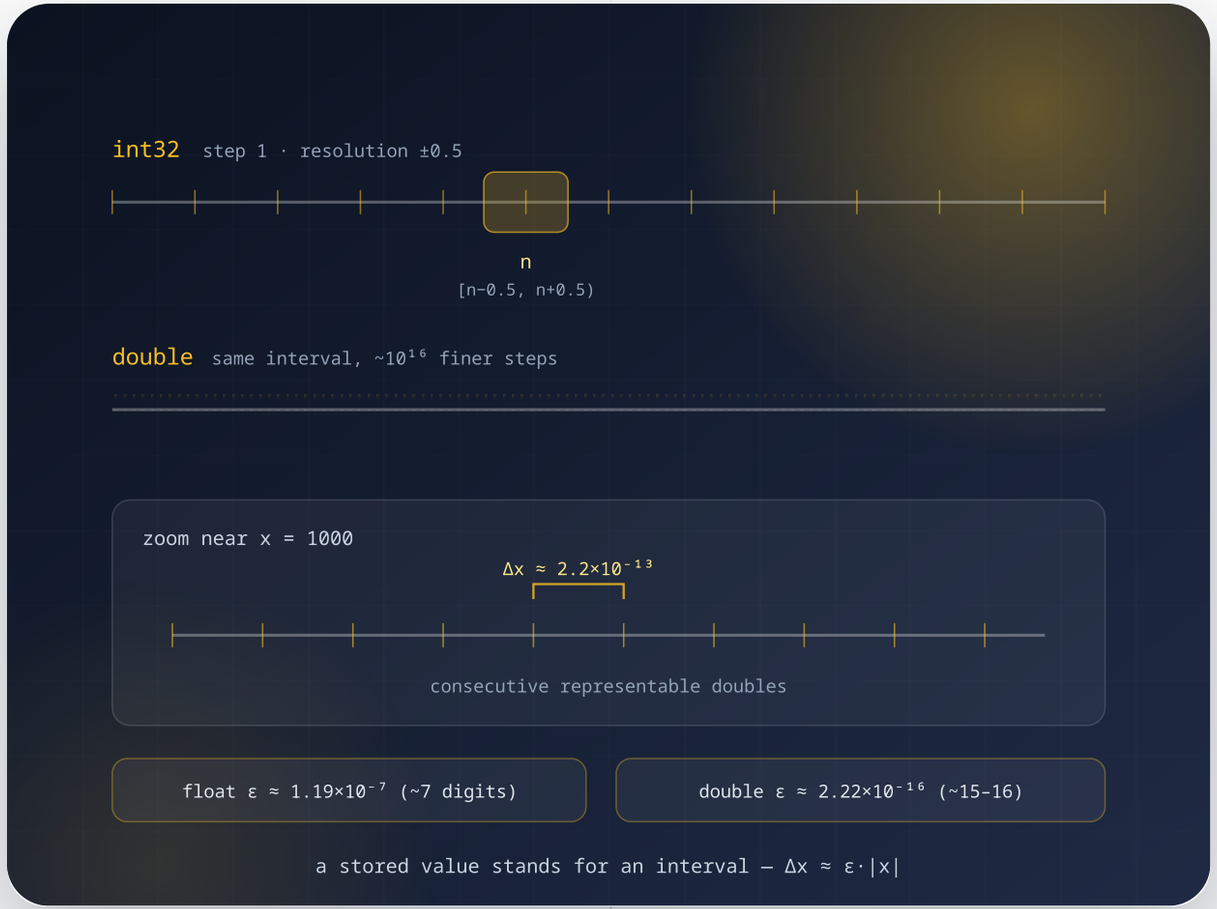


Every stored number is a bin

A value on disk is not a real number — it is a cell of a finite grid.

- `int n` means `[n−0.5, n+0.5)`: a bin of width 1
- `double x`: neighbours are $\Delta x \approx \varepsilon \cdot |x|$ apart, $\varepsilon \approx 2.22\times10^{-16}$
- At $x = 1000$, $\Delta x \approx 2.2\times10^{-13}$ — tiny, but not zero
- `float` is coarser: $\varepsilon \approx 1.19\times10^{-7}$ (~7 digits)

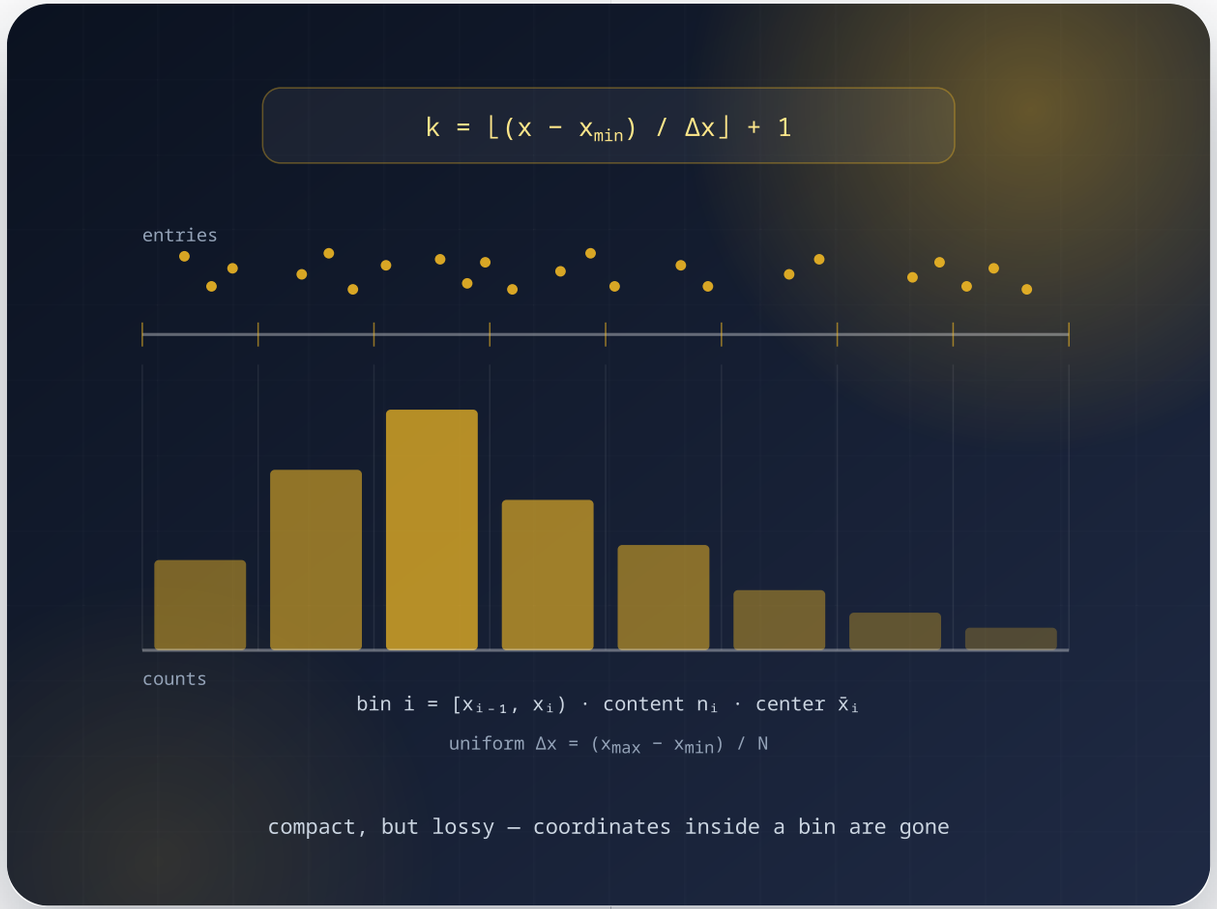
So a stored dataset is a set of intervals — it is **already binned** by its type.



Binning: intervals and indices

Binning aggregates entries into intervals along each axis.

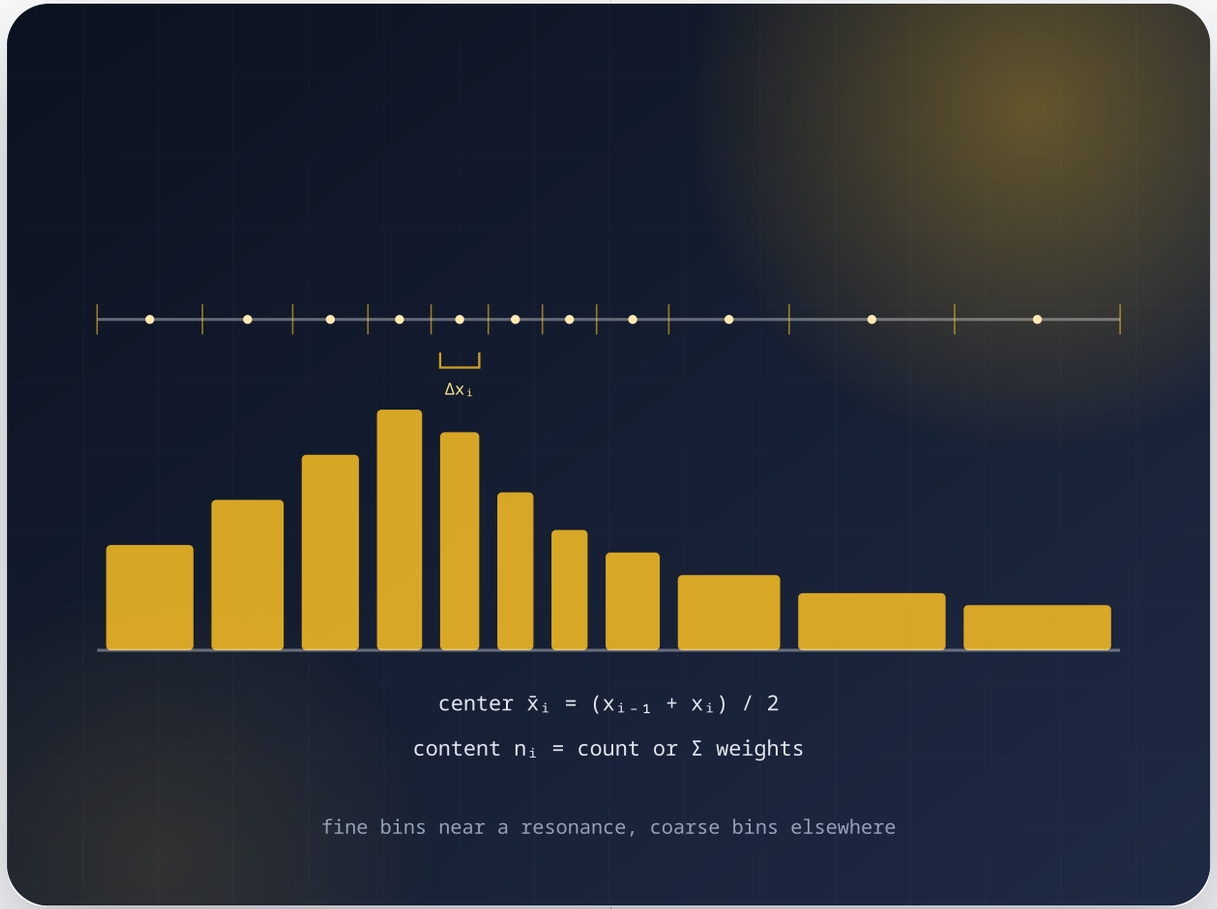
- Edges $x_0 < x_1 < \dots < x_N$; bin i covers $[x_{i-1}, x_i)$
- Uniform binning: $\Delta x = (x_{max} - x_{min})/N$
- Index $k = \lfloor (x - x_{min})/\Delta x \rfloor + 1$
- Each bin accumulates a count or a sum of weights
- A histogram is a discrete approximation of a continuous distribution



Variable bins and bin content

Bins need not be uniform — only the edges must increase.

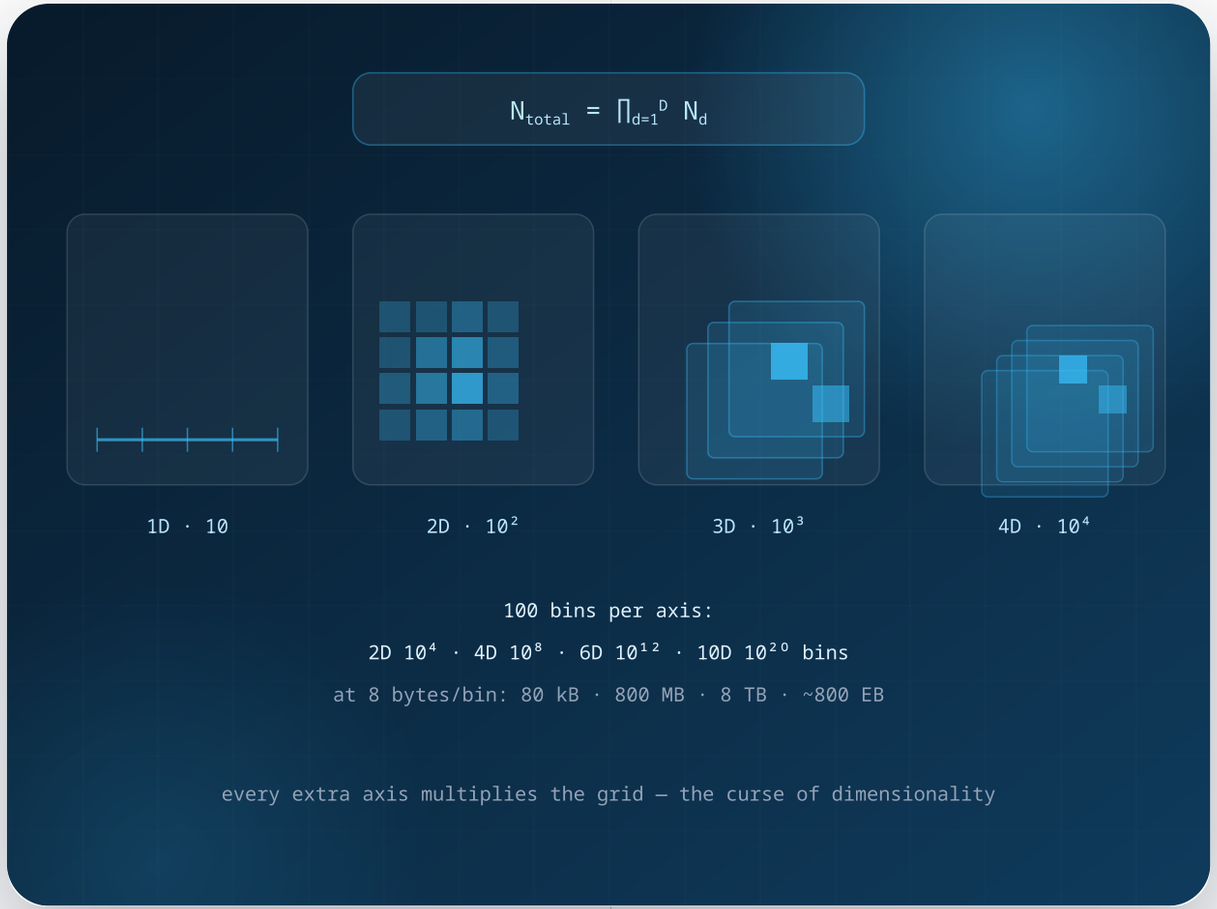
- Finer bins where the physics is: resonances, thresholds
- Coarser bins elsewhere to keep statistical errors down
- Center $\bar{x}_i = (x_{i-1} + x_i)/2$
- Content n_i = count or Σw; error from the weights
- The binning is a choice — and it shapes what you can see



Multi-dimensional binning

Each axis multiplies the grid: $N_{total} = \prod_{d=1}^{D} N_d$.

- D axes partition the space into hypercubic cells
- A bin is the tuple $(i_1, \ldots, i_D)$
- 100 bins/axis: 10^4 (2D), 10^8 (4D), 10^{12} (6D), 10^{20} (10D)
- At 8 bytes/bin: 80 kB → 800 MB → 8 TB → ~800 EB
- This is the **curse of dimensionality**

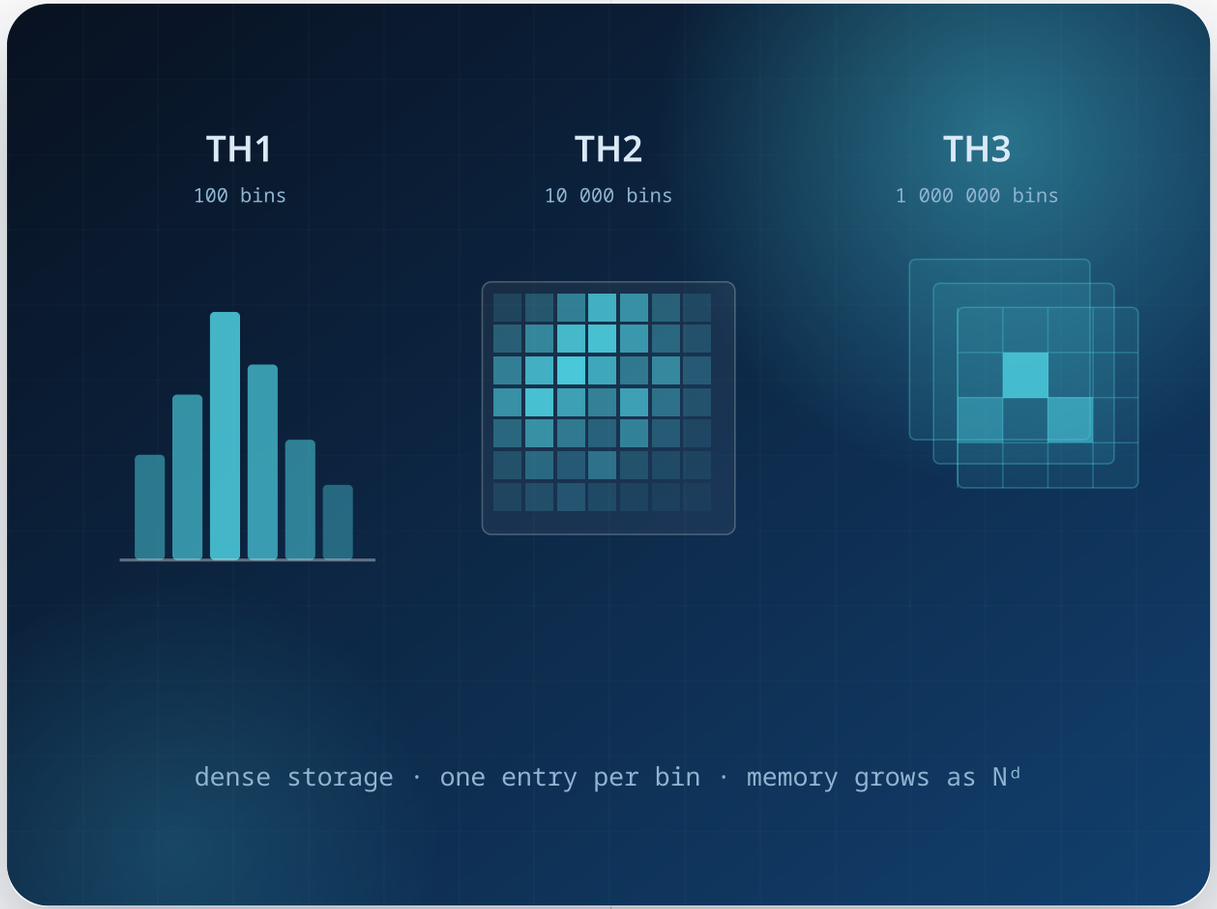


ROOT histograms: TH1, TH2, TH3

Fixed, dense binning — every bin of the grid exists in memory.

```
TH1D h1("h1", "", 100, 0, 1);                    // 100 bins
TH2D h2("h2", "", 100, 0, 1, 100, 0, 1);         // 10 000 bins
TH3D h3("h3", "", 100, 0, 1, 100, 0, 1,
        100, 0, 1);                              // 1 000 000 bins
```

Memory grows as N^d: a fourth 100-bin axis would already need 10^8 bins.

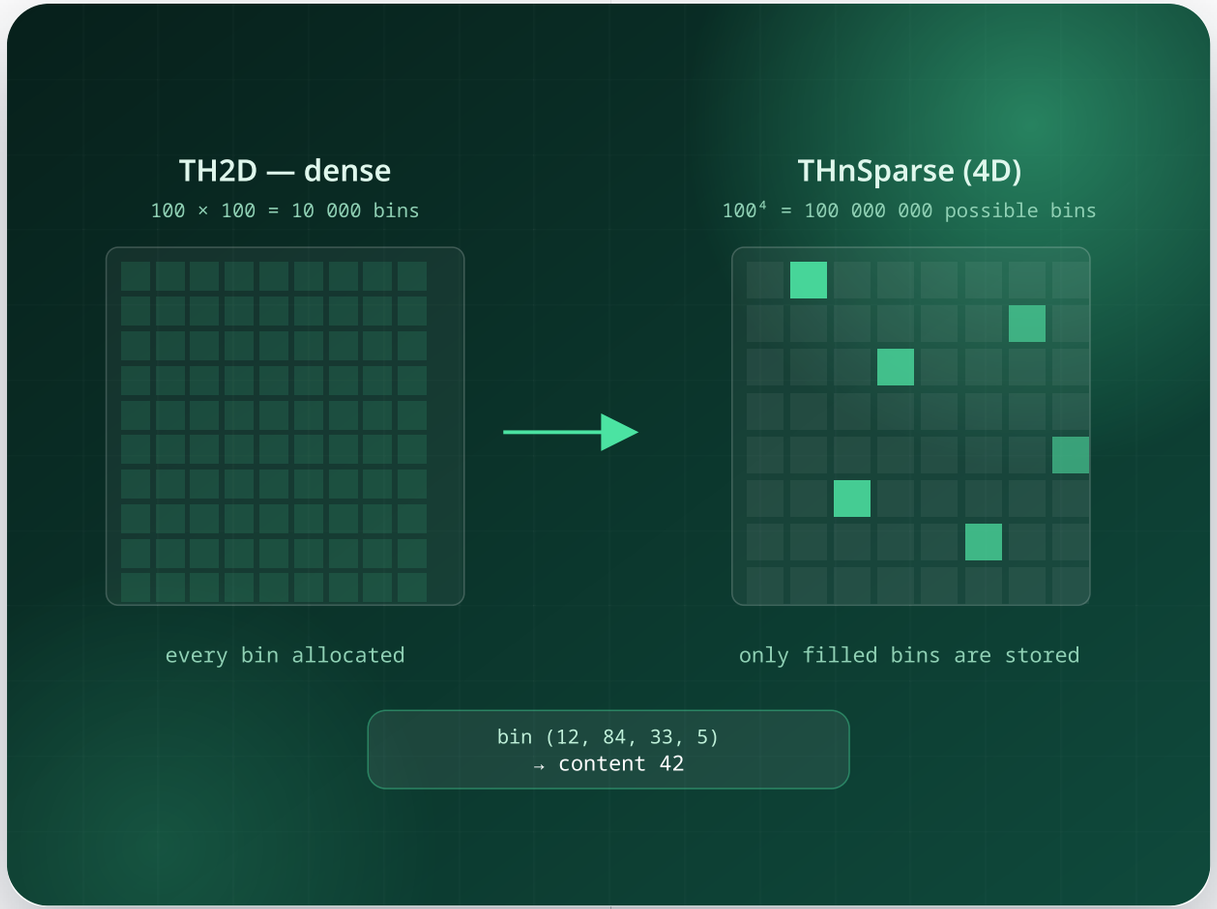


THnSparse: sparse n-dimensional data

- Same axes and bin edges as TH1/TH2/TH3 — but **n** dimensions
- Only bins that were actually filled are stored
- $100^4 = 10^8$ possible bins, a tiny fraction typically filled

```
Int_t nbins[4]   = {100, 100, 100, 100};
Double_t xmin[4] = {0., 0., 0., 0.};
Double_t xmax[4] = {1., 1., 1., 1.};
THnSparseD hs("hs", "4D", 4, nbins, xmin, xmax);

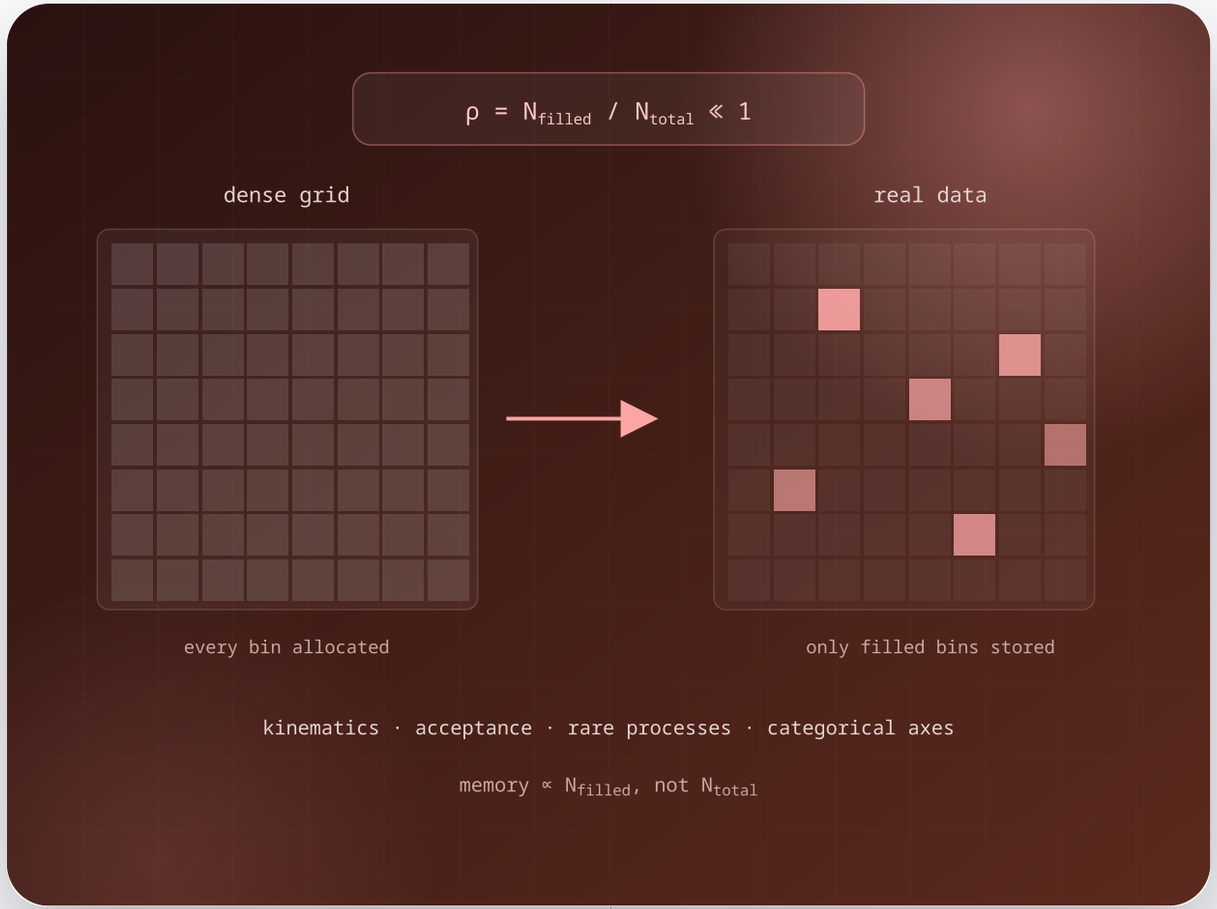
Double_t x[4] = {0.12, 0.84, 0.33, 0.05};
hs.Fill(x);   // stores one bin, not 10^8
```



Sparsity: real data does not fill the grid

Only a small fraction of bins is ever populated.

- Occupancy $\rho = N_{filled} / N_{total} \ll 1$
- Causes: kinematic constraints, detector acceptance, rare processes, categorical axes
- Allocating every bin is wasteful
- A sparse map stores `coordinate → content` only for filled bins
- Memory scales with N_{filled}, not N_{total}



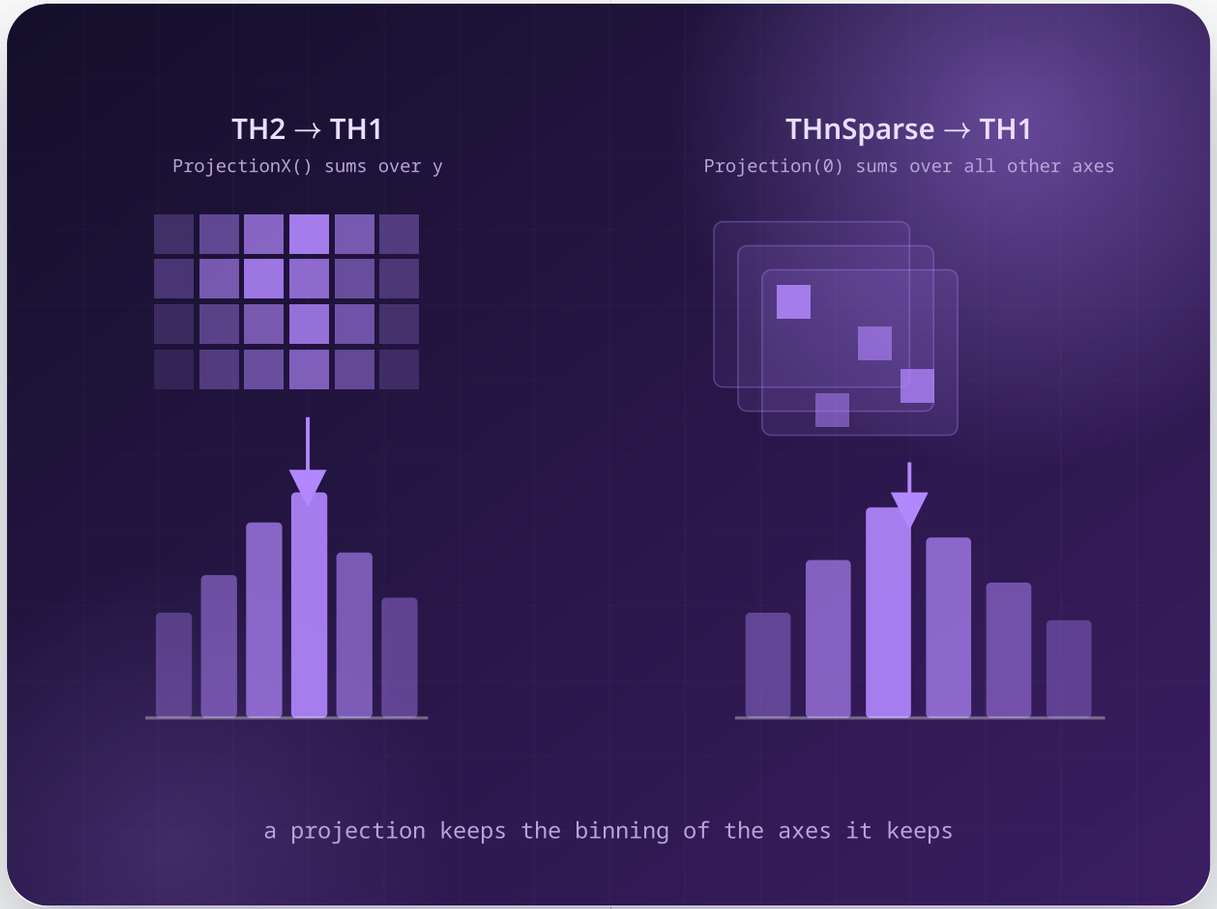
Projections: fewer dimensions

Summing over one or more axes keeps the binning of the axes that remain.

```
TH2D *h2 = ...;
TH1D *px = h2->ProjectionX();      // sum over y
TH1D *py = h2->ProjectionY();      // sum over x

THnSparseD *hs = ...;
TH1D *p0  = hs->Projection(0);     // sum over 1,2,3
TH2D *p01 = hs->Projection(1, 0);  // y=1, x=0
```

For `THnSparse` the first argument is **y**. The result is a normal `TH1`/`TH2` — fit it, draw it, save it.



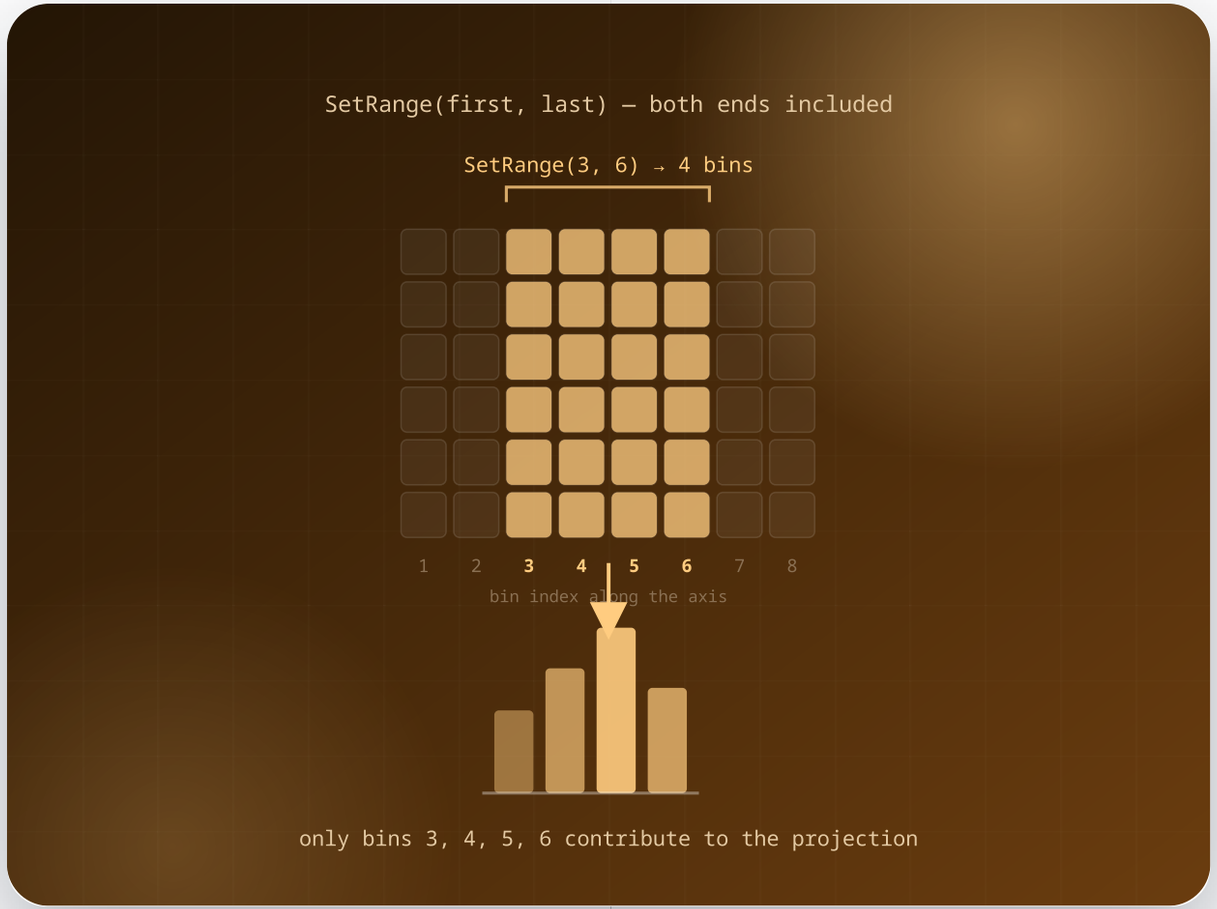
Selecting a range: `SetRange()`

Ranges are set in bin numbers, and **both ends are included**: `SetRange(3, 6)` is bins 3, 4, 5 and 6.

```
// TH1 / TH2 / TH3 — range on an axis
h2->GetYaxis()->SetRange(3, 6);    // bins 3..6, both included
TH1D *px = h2->ProjectionX();
h2->GetYaxis()->SetRange(0, 0);    // 0,0 -> full range

// THnSparse — per dimension (0-based)
hs->GetAxis(1)->SetRange(2, 8);    // bins 2..8, both included
TH1D *p0 = hs->Projection(0);
```

`SetRangeUser(xmin, xmax)` selects by value instead of bin index.



Ranges on several axes

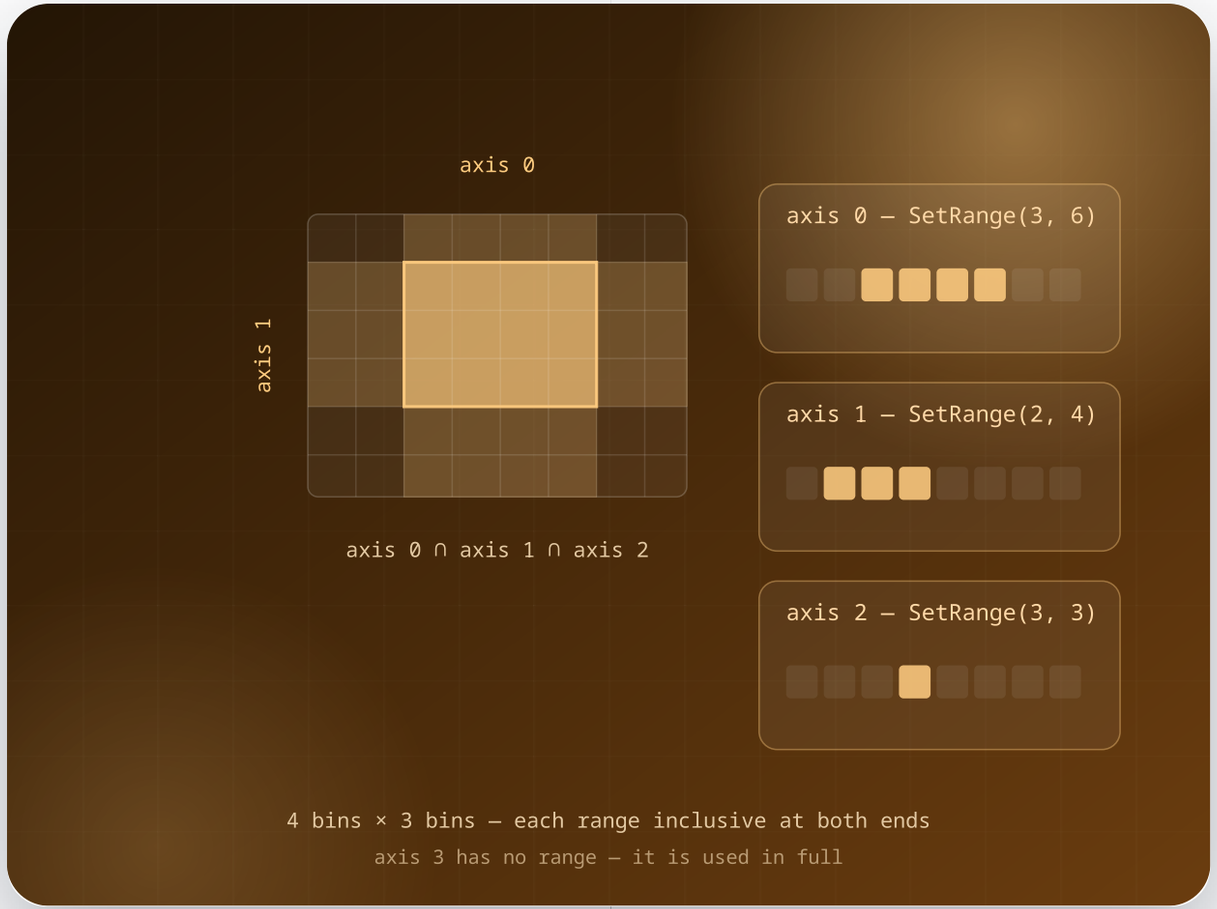
Each axis keeps its own range; a bin is used only if it is inside all of them.

```
hs->GetAxis(0)->SetRange(3, 6);    // pT:    bins 3..6
hs->GetAxis(1)->SetRange(2, 4);    // cent.: bins 2..4
hs->GetAxis(2)->SetRange(3, 3);    // a single bin
                                   // axis 3: full

TH1D *p = hs->Projection(0);       // sums axes 1,2,3

hs->GetAxis(0)->SetRange(0, 0);    // reset axis by axis
hs->GetAxis(1)->SetRange(0, 0);
hs->GetAxis(2)->SetRange(0, 0);
```

Ranges are **intersected** (logical AND), and each is inclusive at both ends — `SetRange(3, 3)` is one bin, not none. There is no global reset: clear each axis with `SetRange(0, 0)`.



One THnSparse, any dimension

A projection keeps the axes you ask for — the others are summed over.

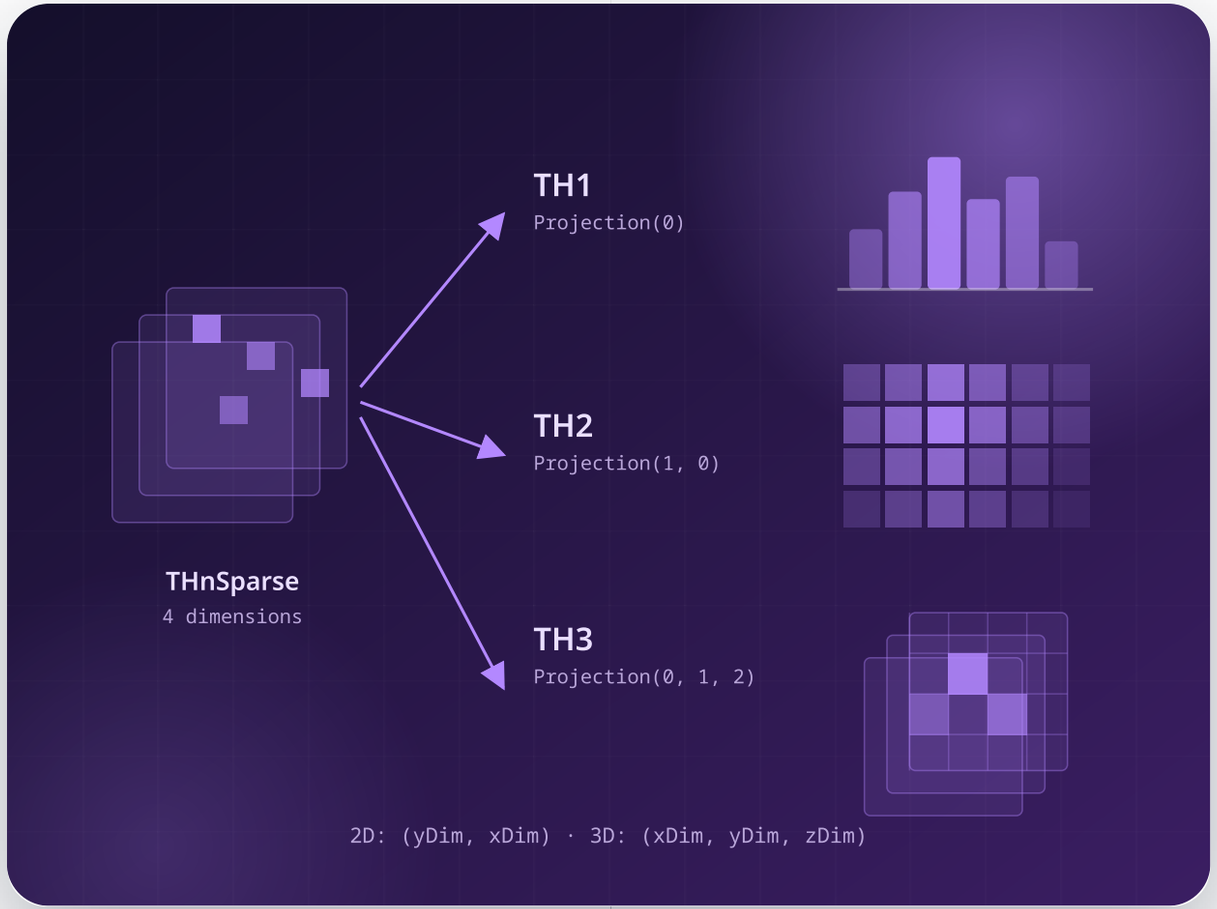
```
THnSparseD *hs = ...;                // 4 dimensions

TH1D *h1 = hs->Projection(0);        // keep axis 0
TH2D *h2 = hs->Projection(1, 0);     // y=1, x=0
TH3D *h3 = hs->Projection(0, 1, 2);  // x=0, y=1, z=2

THnSparseD *h4 = hs->ProjectionND(4, dims); // stay nD
```

Note the argument order: **`TH2 takes (yDim, xDim)`**, while **`TH3 takes (xDim, yDim, zDim)`**.

The result is an ordinary TH1/TH2/TH3 — draw it, fit it, save it.



Mapping and storage

Two classes carry the whole idea: one decides where a bin is, the other keeps what is in it.

`Ndmspc::NGnTree`

- `THnSparse` as the **mapping**
- `TTree` as the **data storage**

`Ndmspc::NBinning`

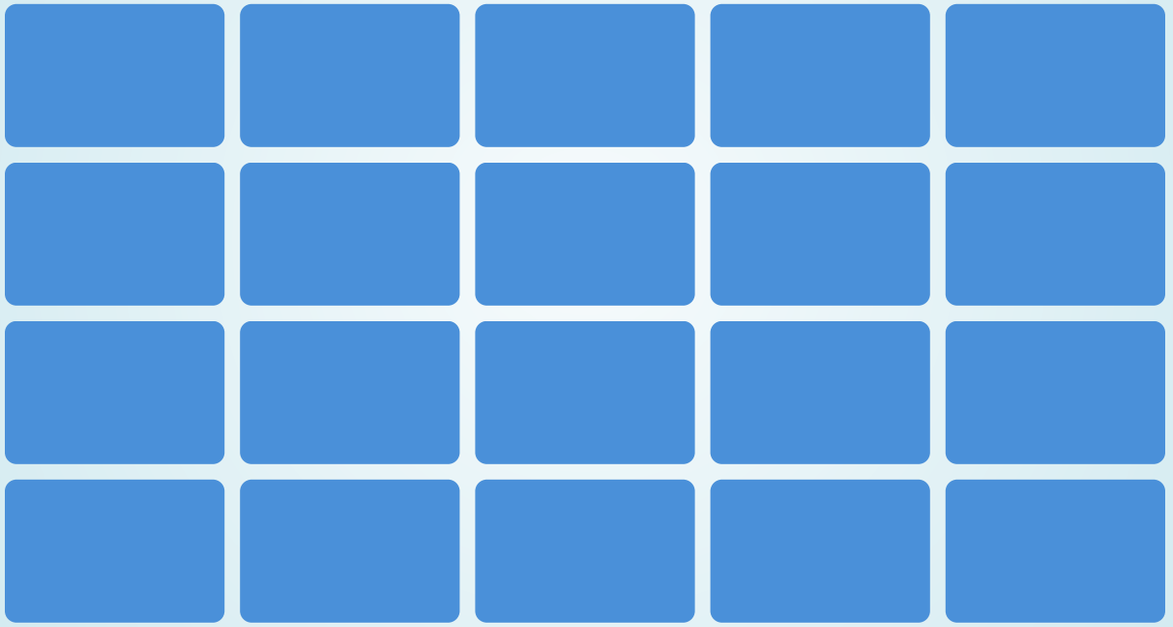
- The user defines the finest **base binning**
- Other binnings are **rebinning** of that base
- **Multiple** binnings over the same stored data

The mapping is sparse, the storage is linear, and the binnings are views — nothing is rewritten when the aggregation changes.

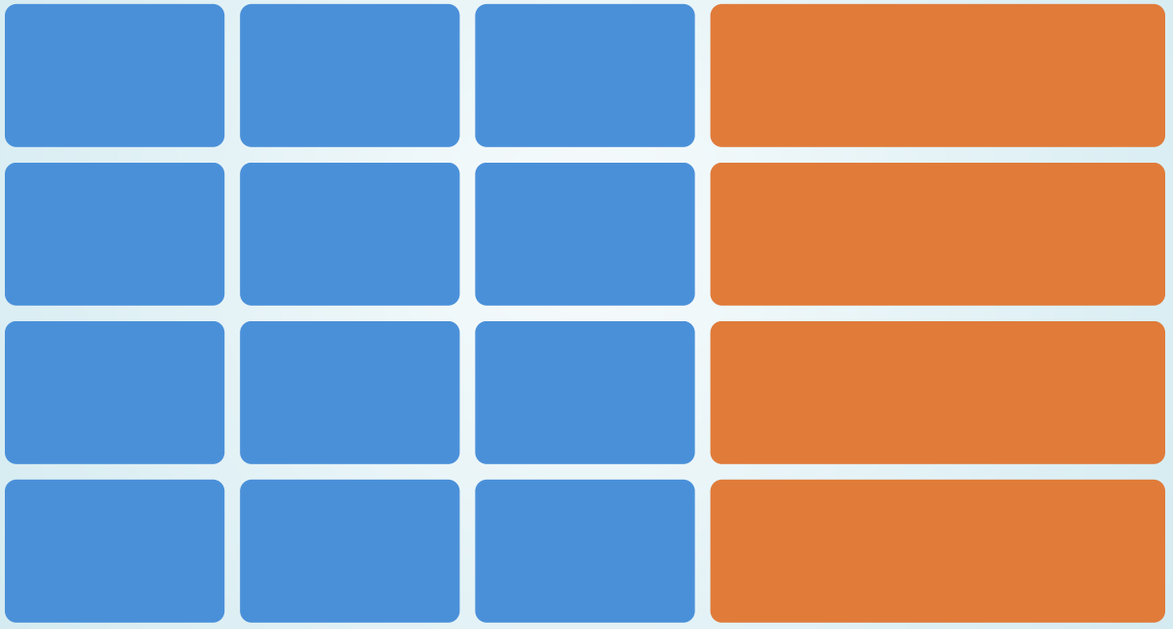
Integrated or single bin



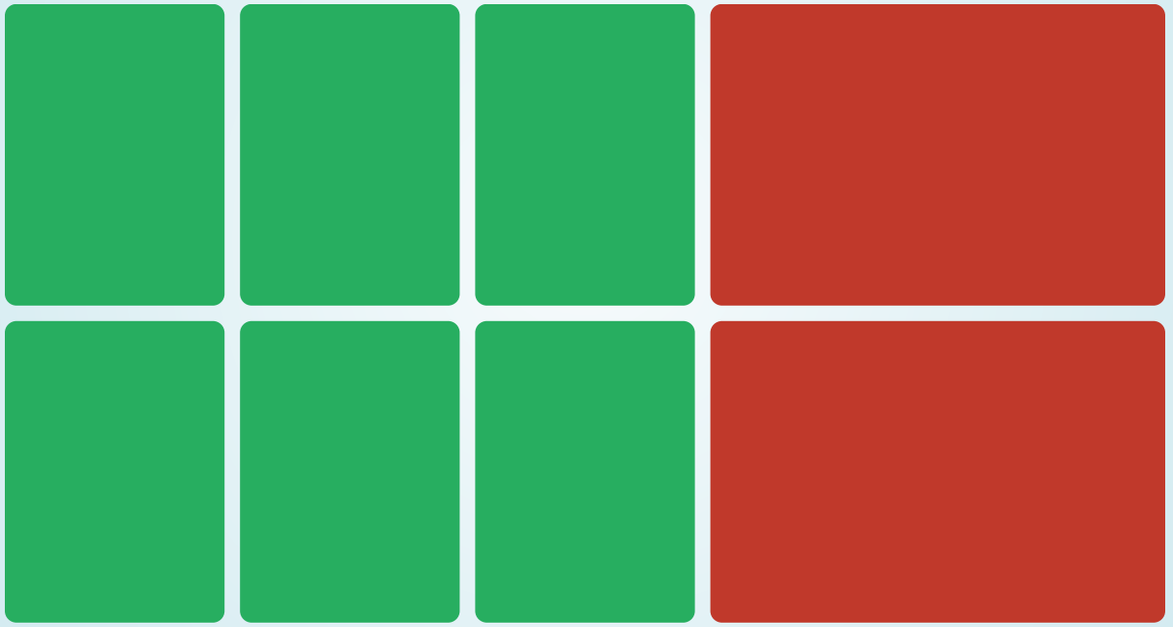
b0 — Uniform "base" Binning



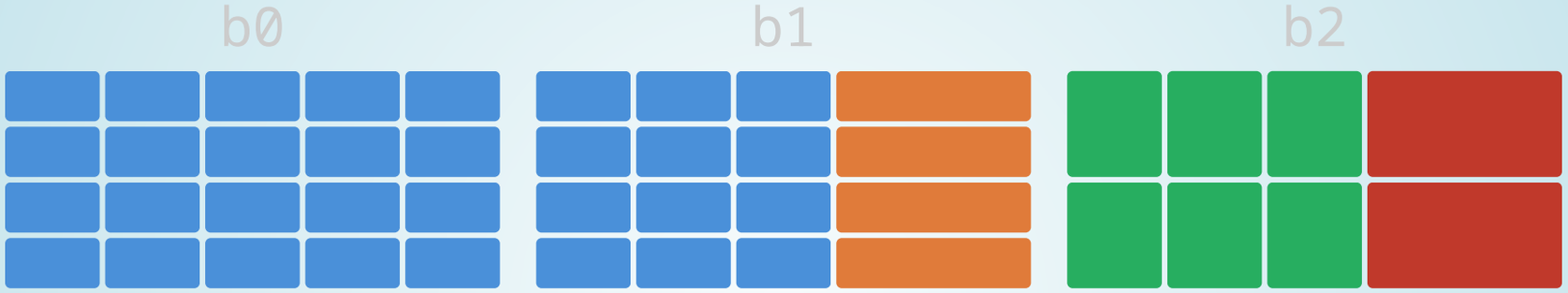
b1 — Rebin x (right)



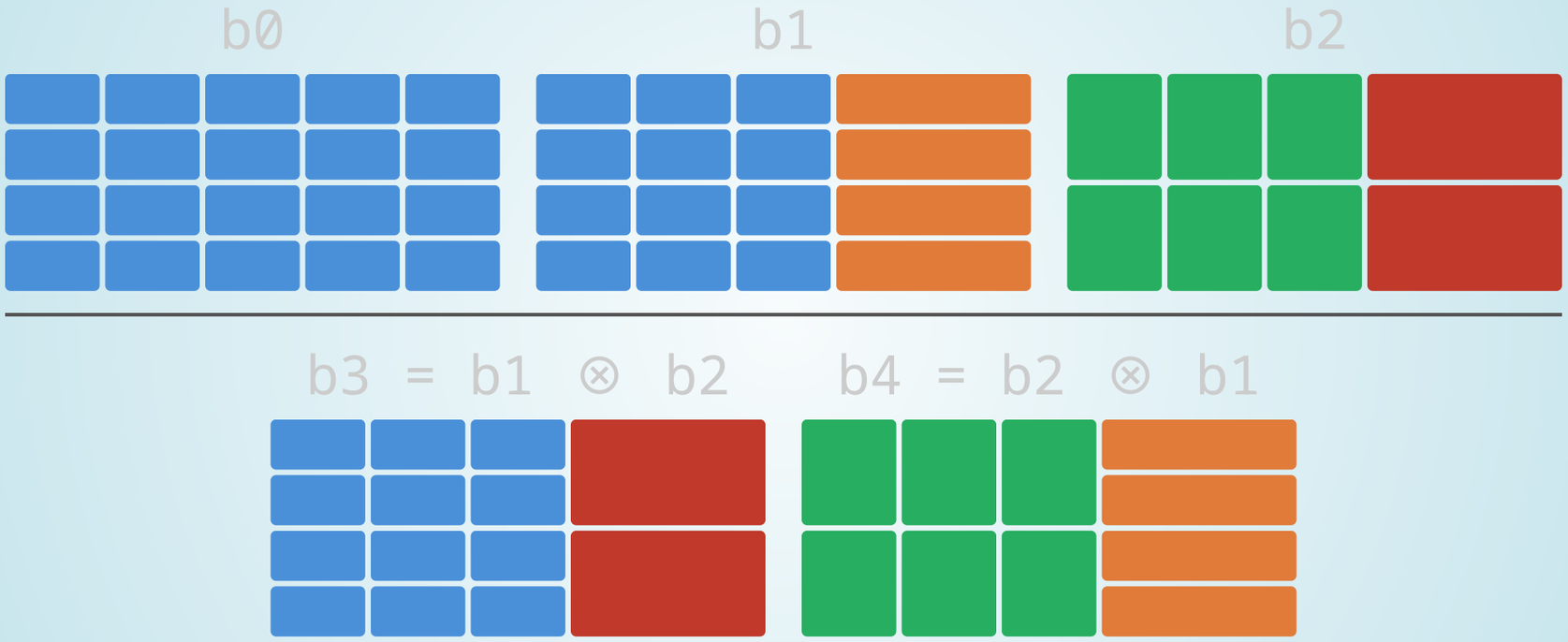
b2 — Rebin x+y



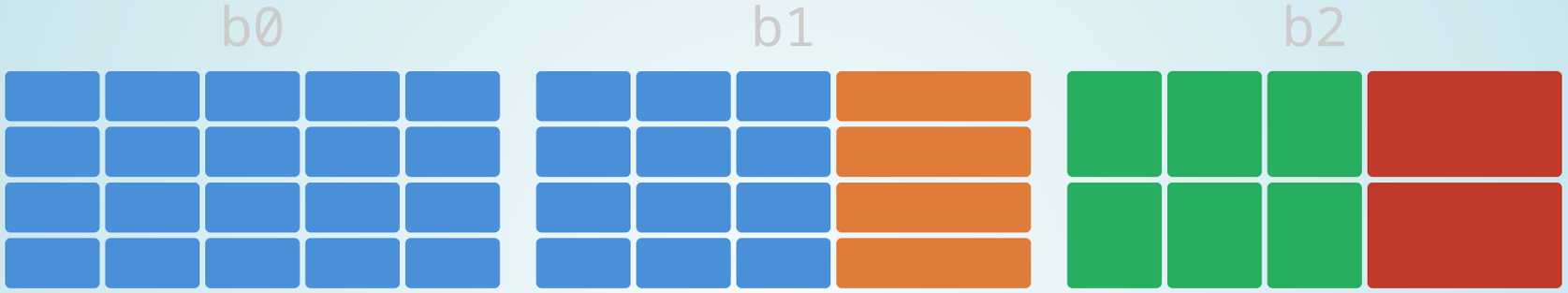
Multiple binnings



Multiple binnings

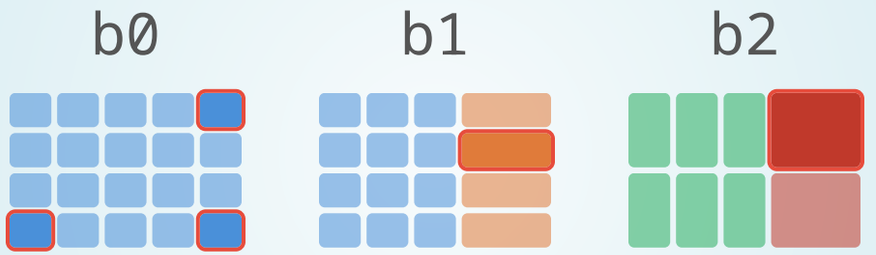


Multiple binnings



NGnTree mapping → `THnSparse`

NGnTree → TTree Storage



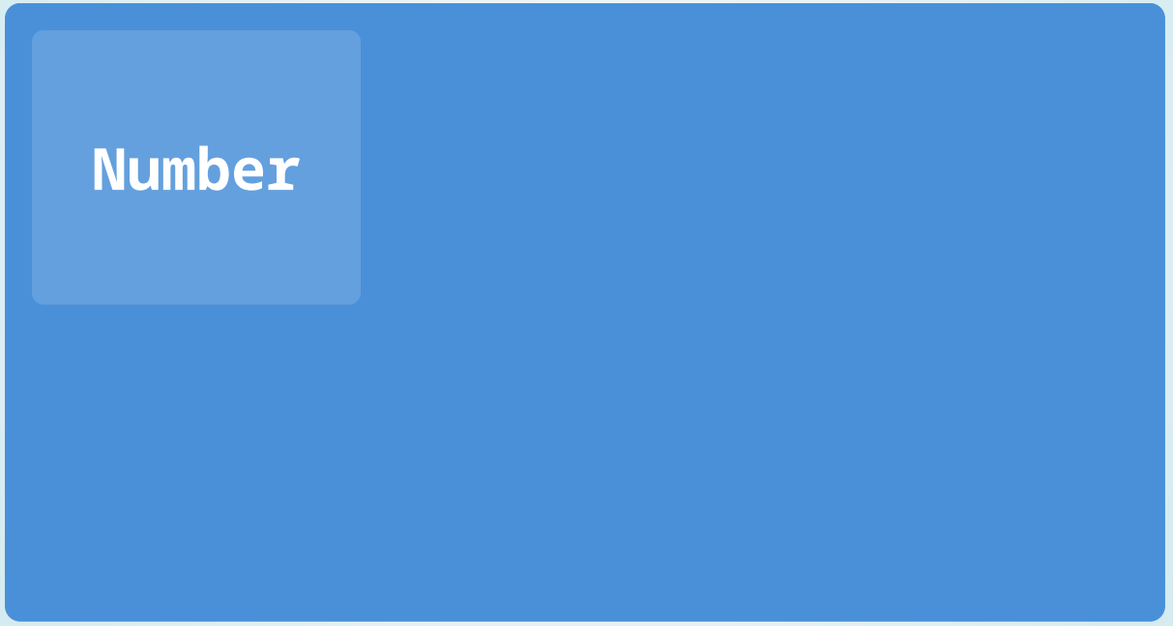
TTree (linear storage)

Entry 0 ... Entry 5 ... Entry 20 ... Entry 23 ... Entry 32

Single bin



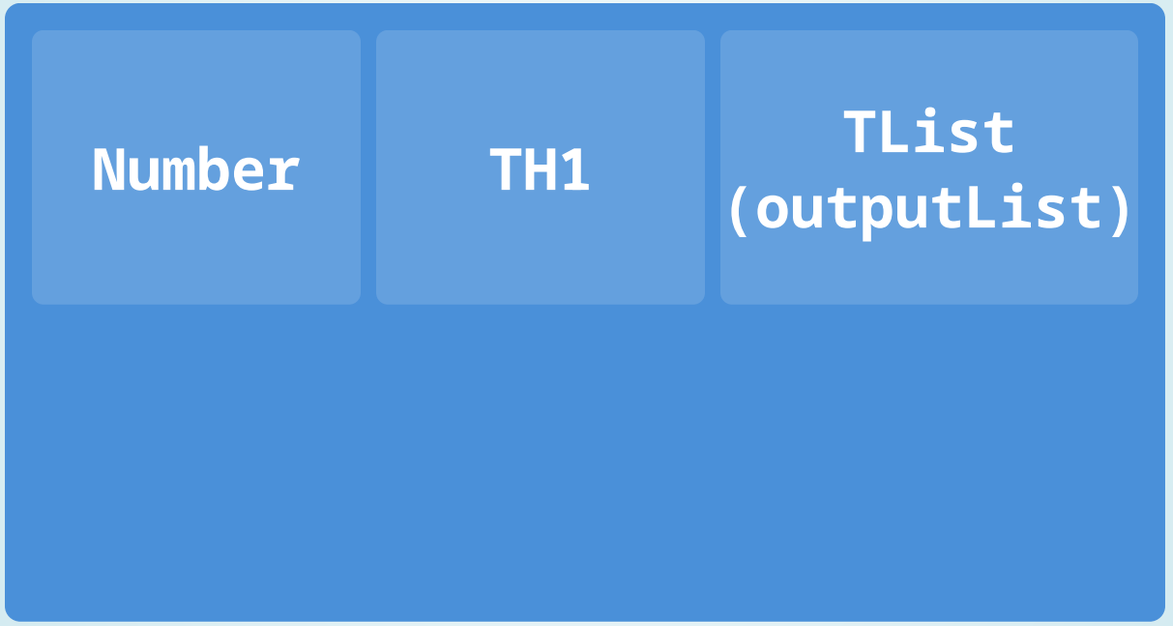
Single bin



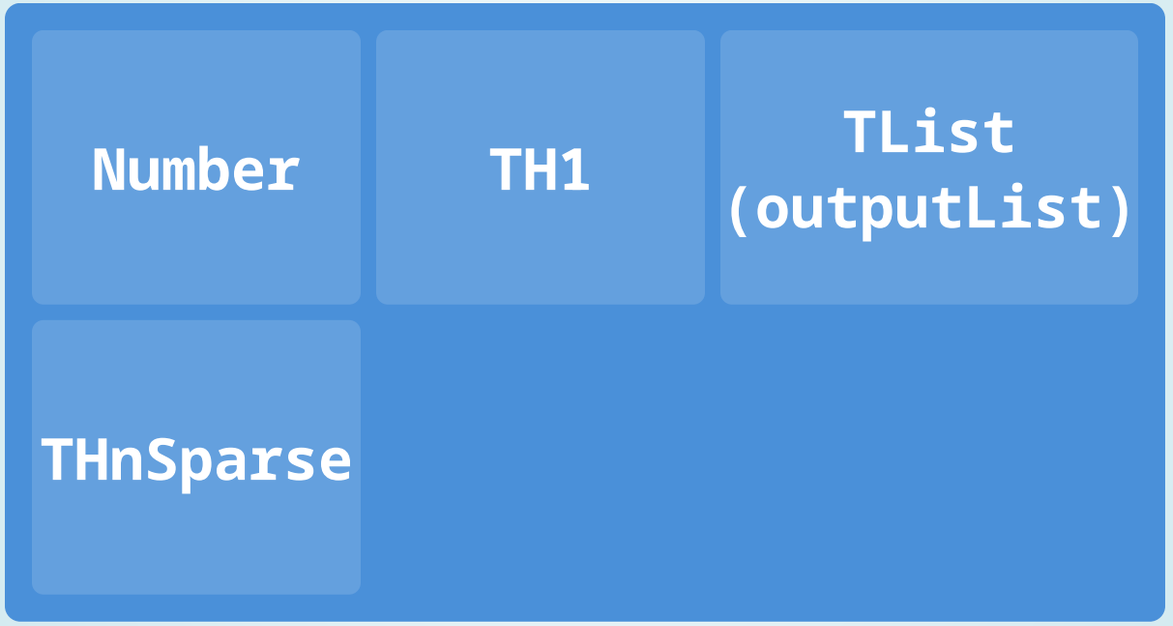
Single bin



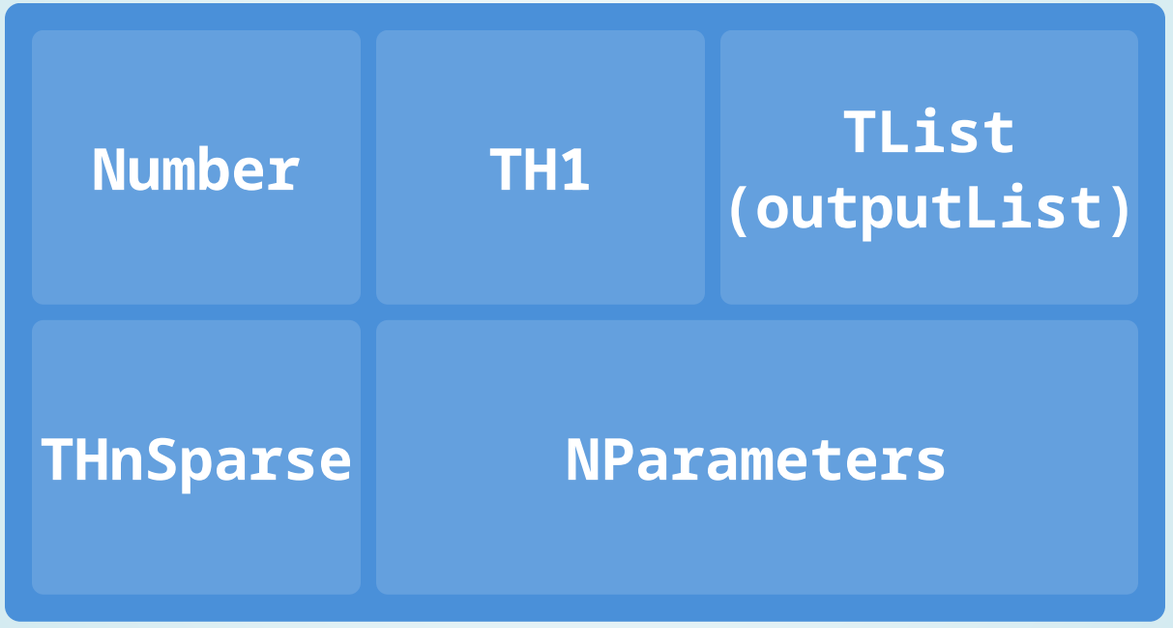
Single bin



Single bin



Single bin



Single bin

NParameters (TH1)

Label	Value	Error

Single bin

NParameters (TH1)

Label	Value	Error
p1	10.2	0.88

Single bin

NParameters (TH1)

Label	Value	Error
p1	10.2	0.88
p2	-4.2	0.1

Single bin

NParameters (TH1)

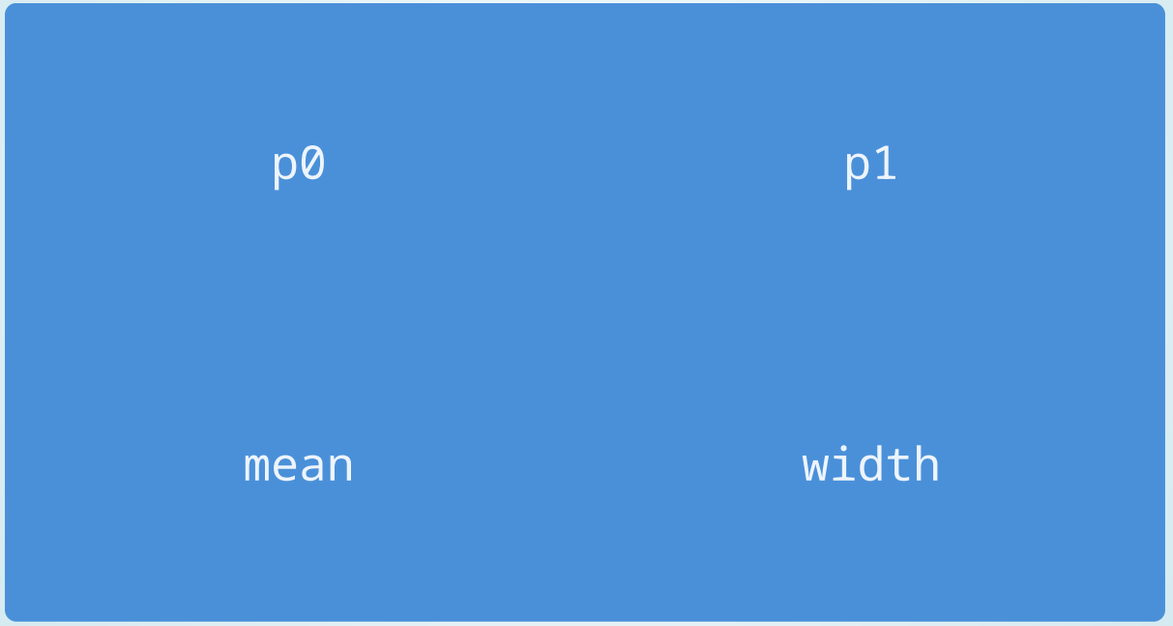
Label	Value	Error
p1	10.2	0.88
p2	-4.2	0.1
mean	1.5	0.25

Single bin

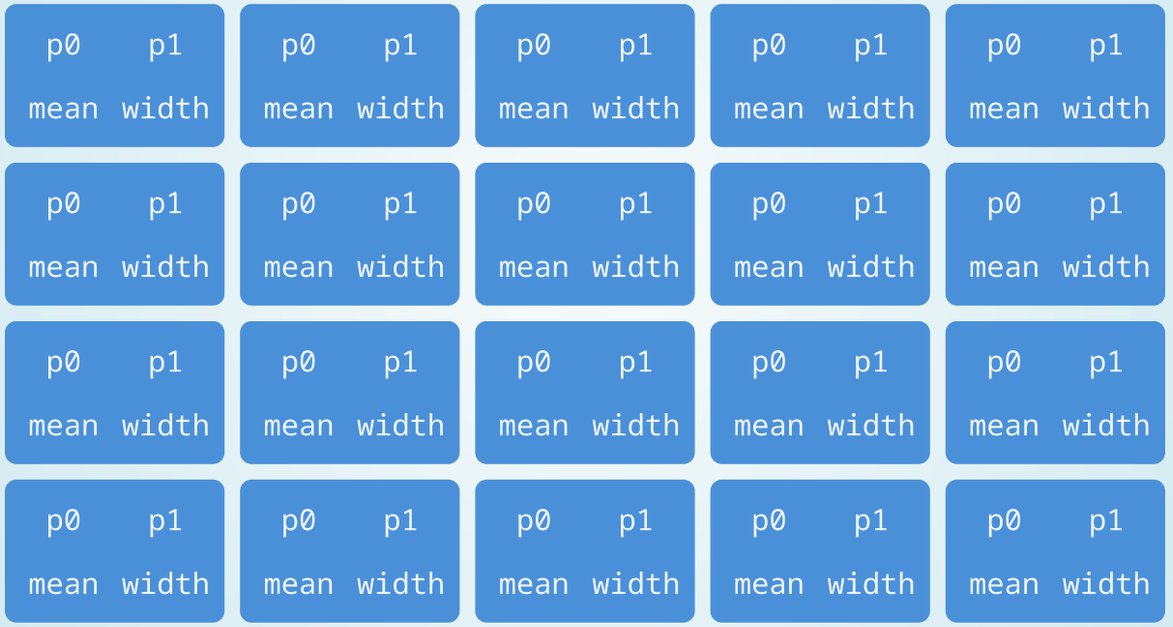
NParameters (TH1)

Label	Value	Error
p1	10.2	0.88
p2	-4.2	0.1
mean	1.5	0.25
width	4.5	0.65

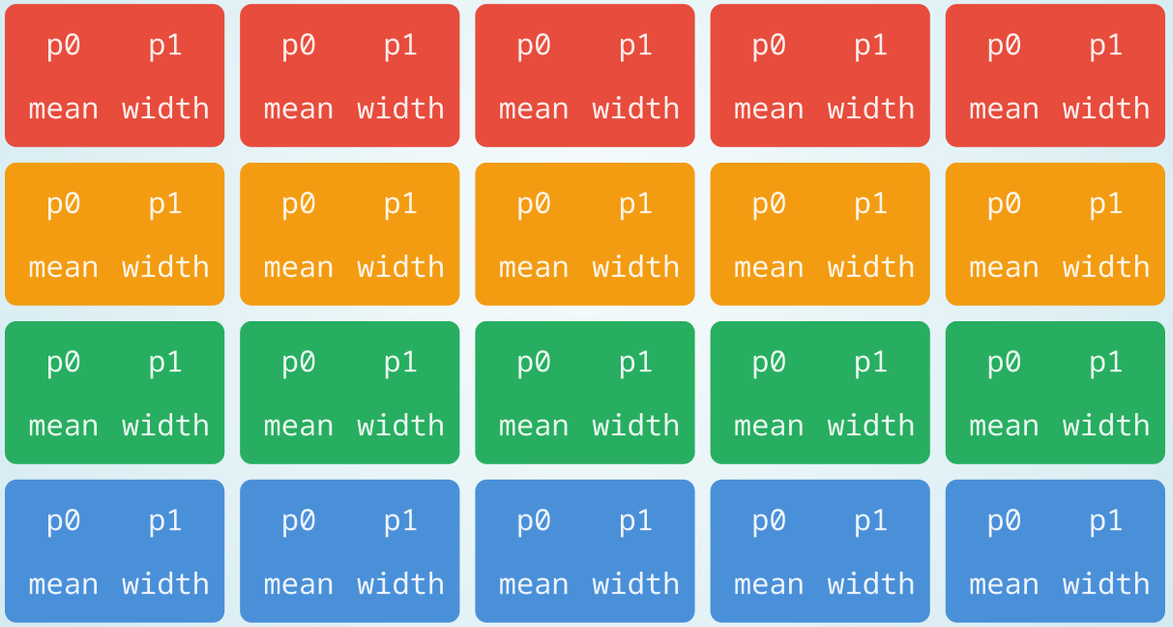
Single bin (with parameters)

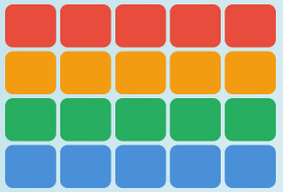


b0 binning → Spectras

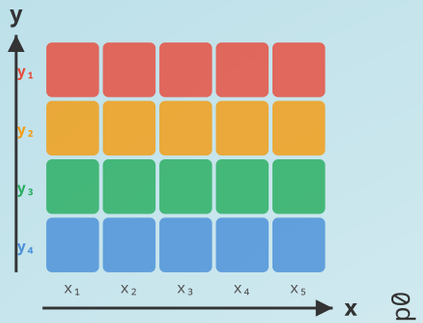


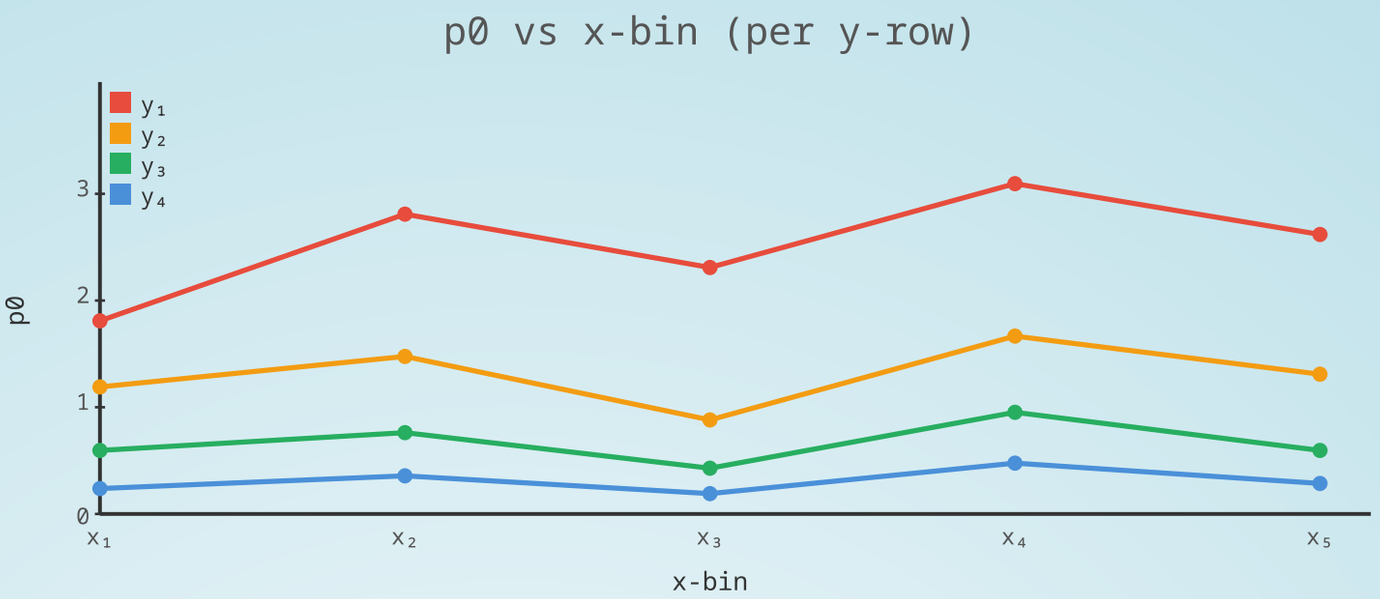
b0 binning → Spectras



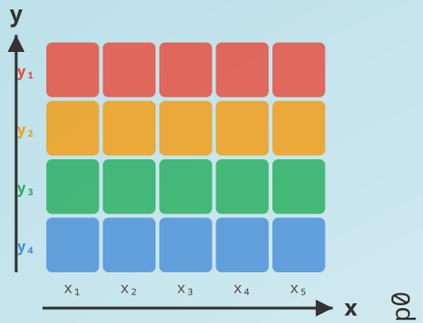


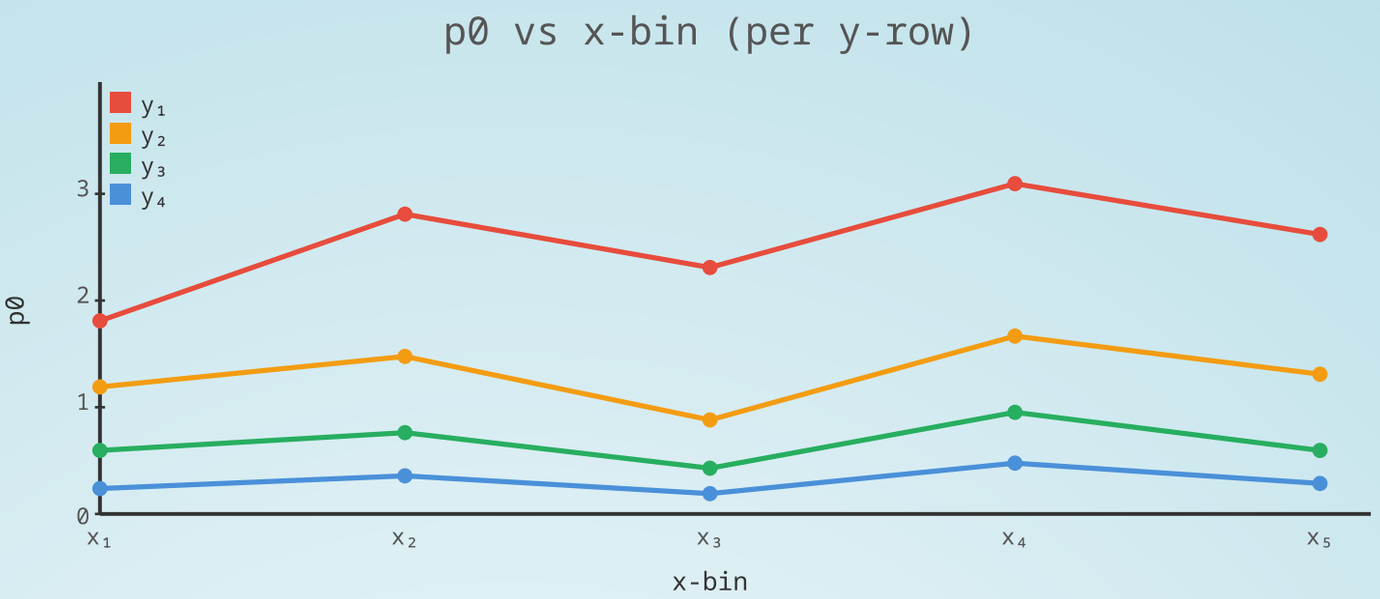
Spectras of p0 parameter

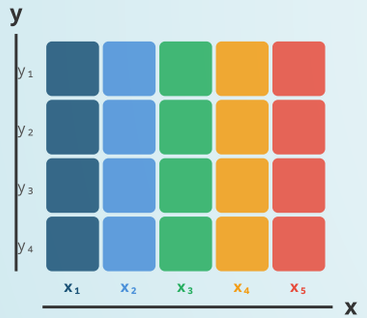


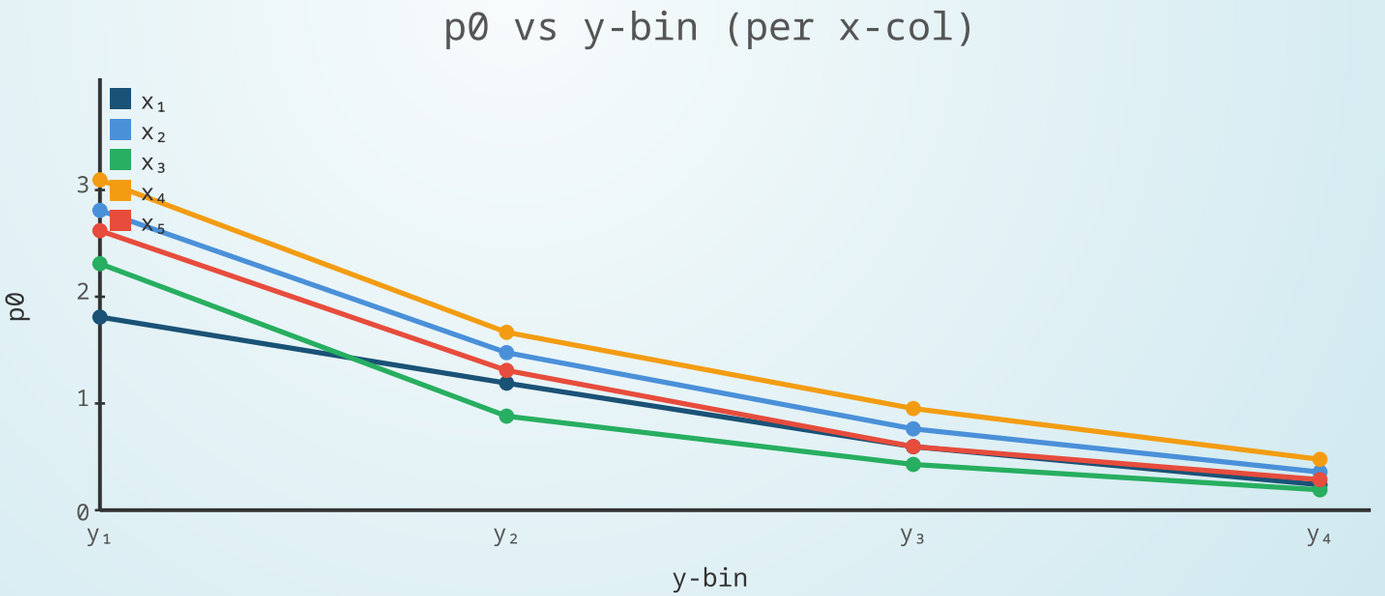


Spectras of p0 parameter

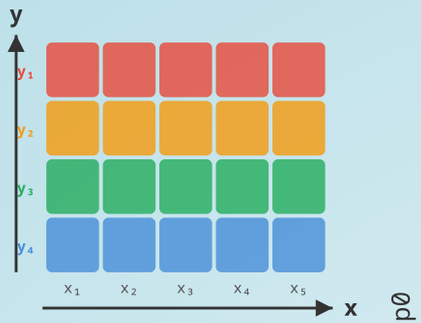


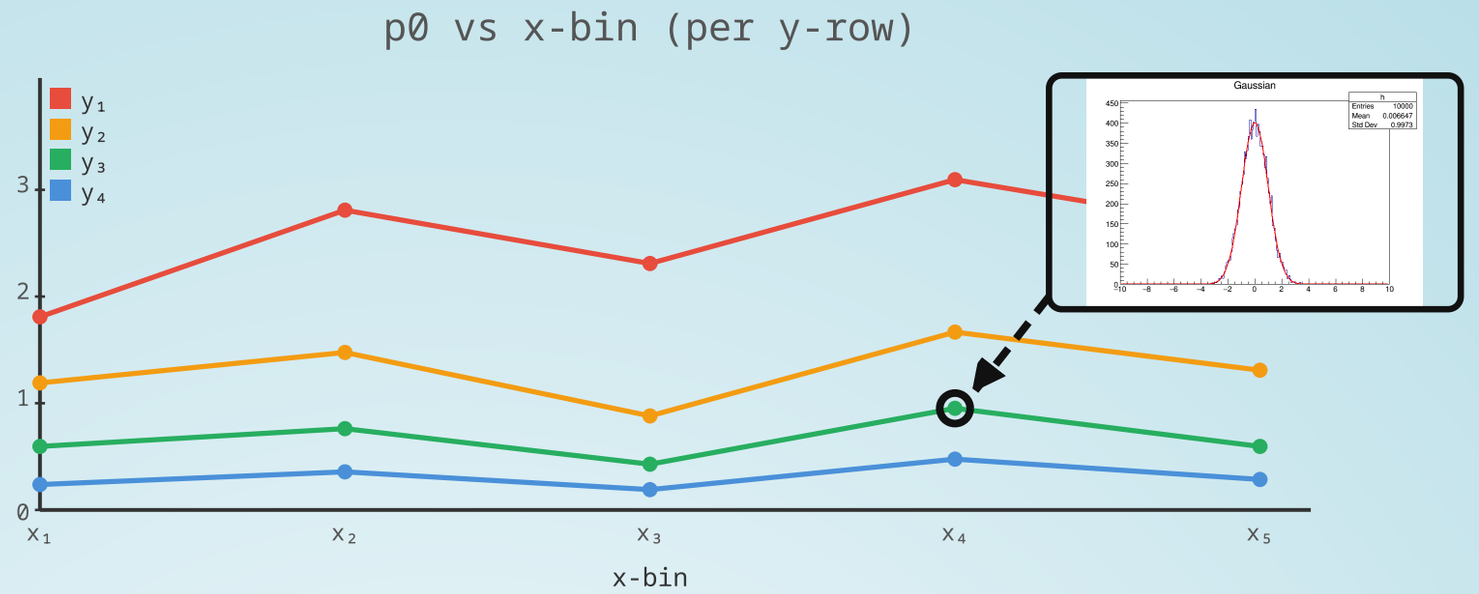



y
y1
y2
y3
y4
x1
x2
x3
x4
x5
x

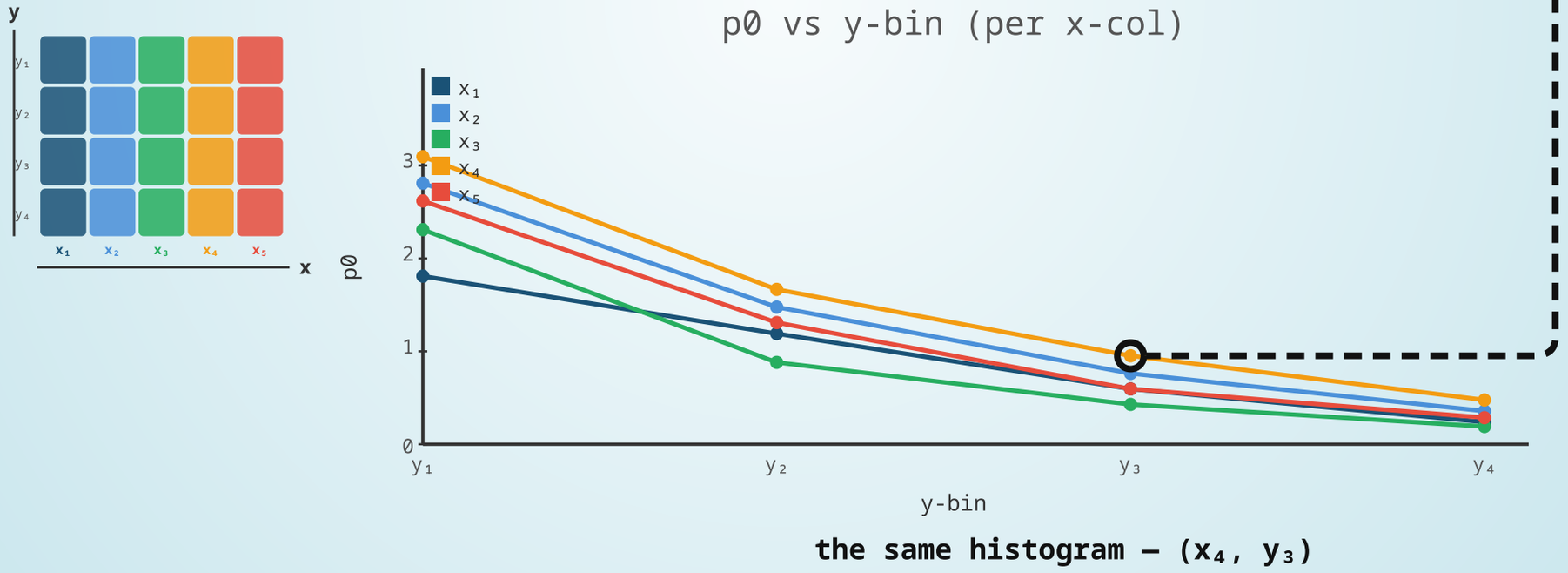

p0 vs y-bin (per x-col)
x1
x2
x3
x4
x5
p0
3
2
1
0
y1
y2
y3
y4
y-bin

Spectras of p0 parameter



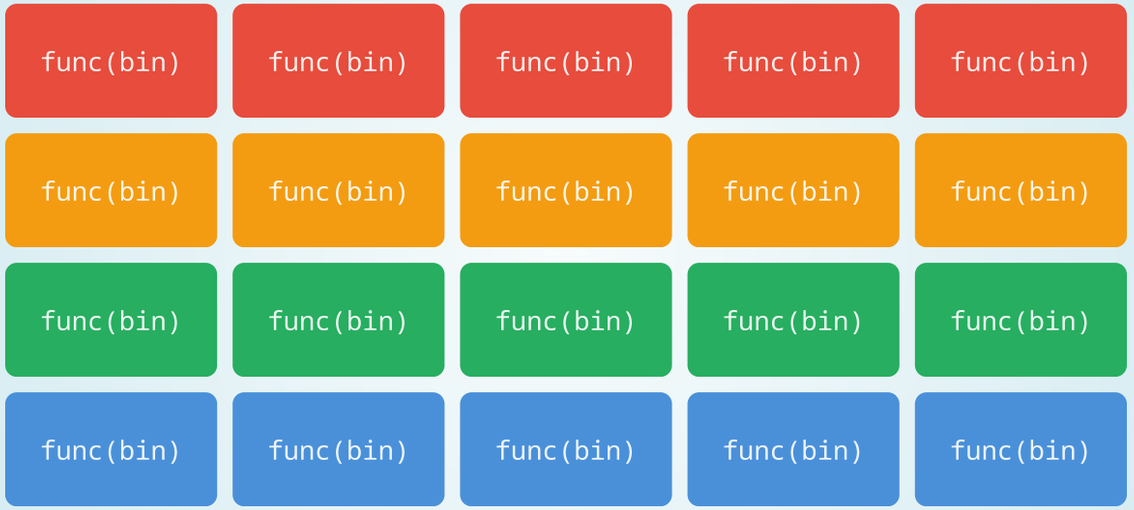


histogram stored in bin (x_4, y_3)



the same histogram — (x_4, y_3)

b0 binning → execute **func** for every bin




```
ngnt->Process([](Ndmspc::NBinningPoint *bin,...) {
  // user function – called once per bin
});
```

Simple usage: build the tree

Two axes, one NGnTree, two binning definitions.

- The axes define the base binning
- Each `AddBinningDefinition` adds another view
- `b1` is a plain rebinning
- `b2` mixes rebin and variable bins
- All of them use the same stored entries

```
TObjArray *axes = new TObjArray();
axes->Add(new TAxis(5, 0.0, 0.5));   // a1
axes->Add(new TAxis(20, 0.0, 1.0));  // a2

auto *ngnt = new Ndmspc::NGnTree(axes);

// b1 — no rebinning
std::map<std::string, std::vector<std::vector<int>>> b;
b["a1"] = {{1}};
b["a2"] = {{1}};
ngnt->GetBinning()->AddBinningDefinition("b1", b);

// b2 — rebin a1, variable bins in a2
std::map<std::string, std::vector<std::vector<int>>> b2;
b2["a1"] = {{2}};
b2["a2"] = {{1,4}, {2,3}, {5}};
ngnt->GetBinning()->AddBinningDefinition("b2", b2);
```

Simple usage (2): process function

The lambda runs once per bin and stores its result with the point.

- `point` — the current bin: coordinates, labels, parameters
- `outputPoint` — objects stored with this point
- `output` — shared and merged at the end
- `Process(func, "b1")` runs one binning; `""` runs all

```
auto func = [](Ndmspc::NBinningPoint *point,
               TList *output, TList *outputPoint,
               int threadId)
{
  double x = point->GetBinCenter("a1");
  auto label = point->GetBinLabel("a2");

  outputPoint->Add(canvas);

  point->GetParameters()->SetParameter(
      "meanFit", fit->Parameter(1), fit->Error(1));
};

ngnt->Process(func, "b1");   // "" → all binnings
```

Execution mode: single

One process runs the whole loop.

```
root -b -q macro.C
# or
ndmspc-run macro.C
```

Simplest to run and to debug — a good default while developing.

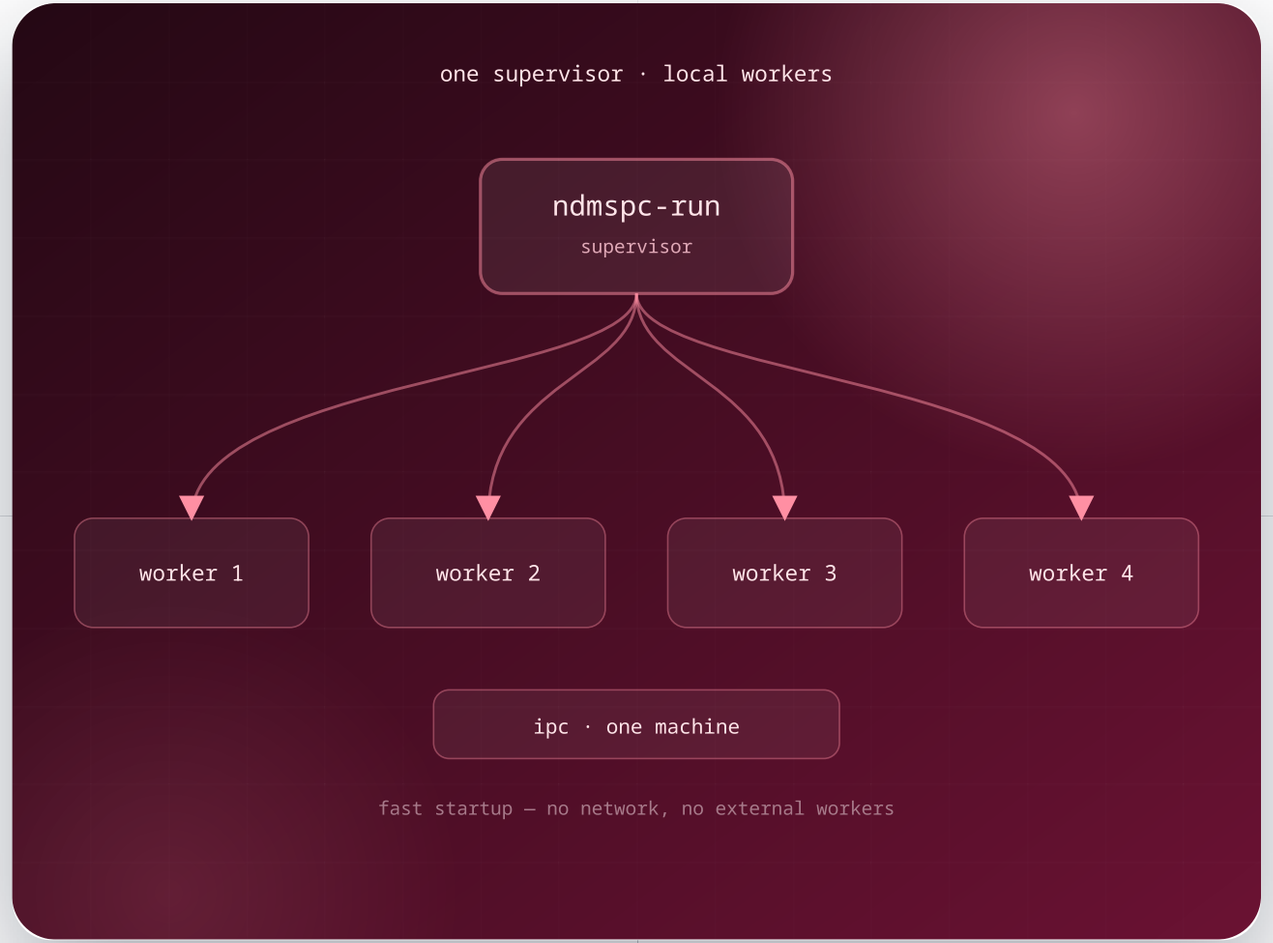


Execution mode: ipc

Several local processes share the bins.

```
export NDMSPC_MAX_PROCESSES=4
root -b -q macro.C
# or
ndmspc-run --mode ipc -n 4 macro.C
```

No network and no external workers — fast startup.



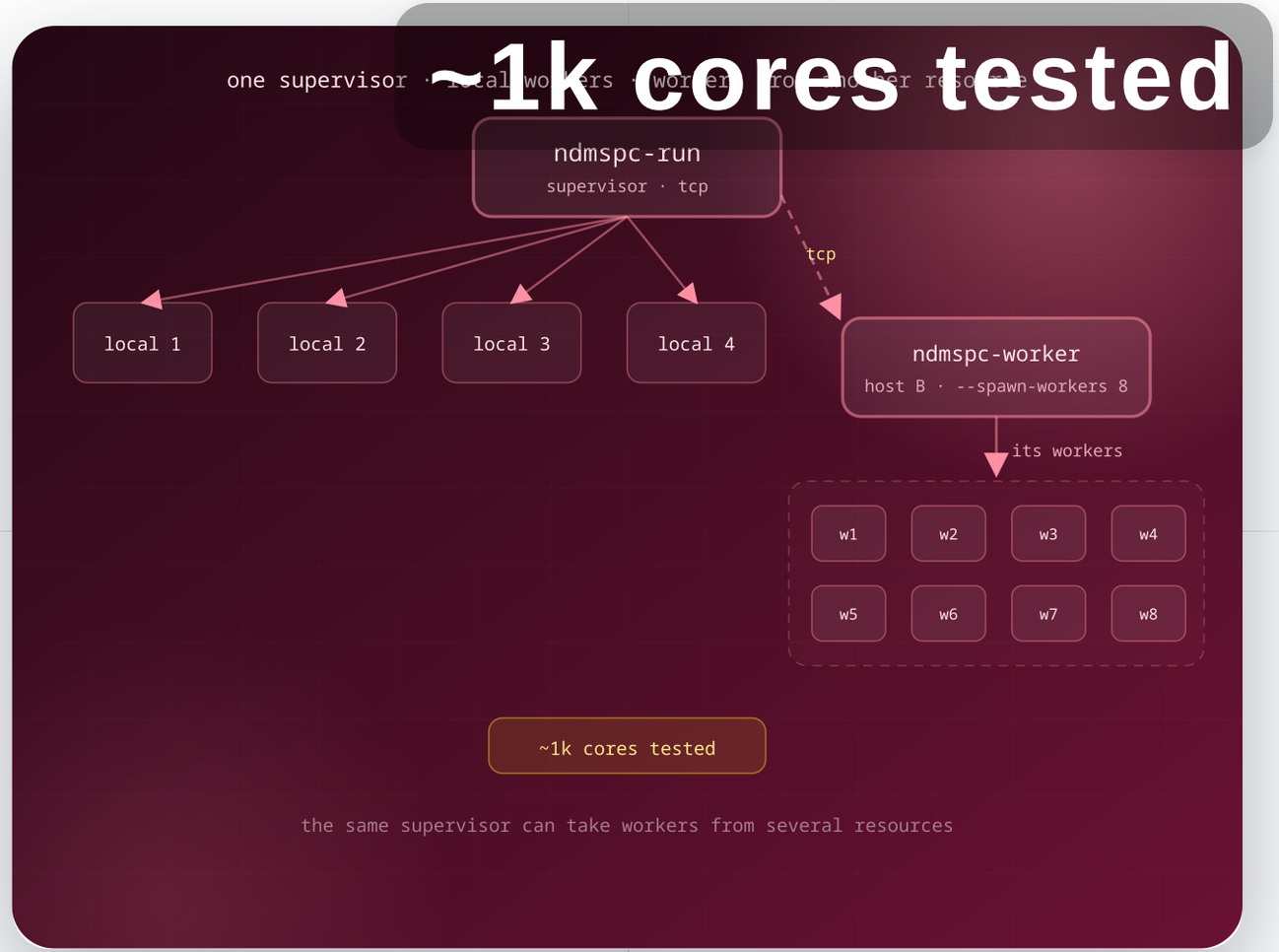
Execution mode: tcp

Workers can live on other machines.

```
ndmspc-run --mode tcp -n 12 macro.C \
    --spawn-workers 4

ndmspc-worker --spawn-workers 8 \
    --endpoint tcp://supervisor-host:5555
```

The same process function; workers just join the supervisor.



From one node to Govorun

`ipc` fills one machine. `tcp` crosses the machine boundary — which is what a batch system asks you to do.

	ipc	**tcp**
Multiple machines	no	**yes**
Process isolation	yes	yes
Filesystem for results	local disk or EOS	shared: NFS / EOS
Worker startup	auto (fork)	`ndmspc-worker`
Workers join late	no	**yes**

- The supervisor binds **one TCP port**; workers connect to `tcp://host:5555` and **may join while the run is in progress** — no restart, no re-scheduling.
- `--spawn-workers 0` is the batch-system mode: start the supervisor, then run `ndmspc-worker --endpoint …` in each job step or on each node.
- Only `NDMSPC_TMP_RESULTS_DIR` must be on shared storage. Each worker writes locally to `NDMSPC_TMP_DIR`, then copies the result to the shared path.
- Macros may be HTTP(S) URLs, so the macro needs no shared filesystem — only the results directory does.

```
# supervisor — one job step
ndmspc-run --mode tcp -n 32 --spawn-workers 0 \
  --tcp-port 5555 --results-dir /scratch/$USER/ndmspc/results \
  NTaxiConvert.C

# on each worker node — another job step
ndmspc-worker --endpoint tcp://supervisor-host:5555 --spawn-workers 8
```

Tuning a cluster run

The knobs that matter once the run is spread over a shared filesystem.

Variable	Default	Set it when
`NDMSPC_TMP_DIR`	local / `/tmp`	always — node-local scratch, never `$HOME`
`NDMSPC_CACHE_DIR`	`~/.cache`	`$HOME` is small — the reshape cache lives here
`NDMSPC_TTREE_BASKET_SIZE`	32000	writing to EOS/NFS → `1048576` for fewer, bigger writes
`NDMSPC_IPC_BATCH_SIZE`	1	many fine-grained bins → batch tasks per message
`NDMSPC_TCP_MAX_INFLIGHT_PER_WORKER`	1	raise for throughput, keep low for fairness
`NDMSPC_TTREE_AUTOFLUSH`	ROOT	long runs, many entries
`NDMSPC_SINGLE_WORKER_FAST_MERGE`	off	single-worker runs — skips `Merge()`

TWO TRAPS

- `ROOT_MAX_THREADS` is threads **inside one process**; `NDMSPC_MAX_PROCESSES` is worker **processes**. Set both high and you oversubscribe the node.
- Counts nest with colons: `NDMSPC_MAX_PROCESSES=2:5` is 2 workers × 5 nested — that is how per-bin work maps onto a multi-core node.
- Auto-selection trap: `NDMSPC_MAX_PROCESSES=1` selects **`ipc`**, not thread mode.

Same process function in all three modes — only the launch changes.

Visualization

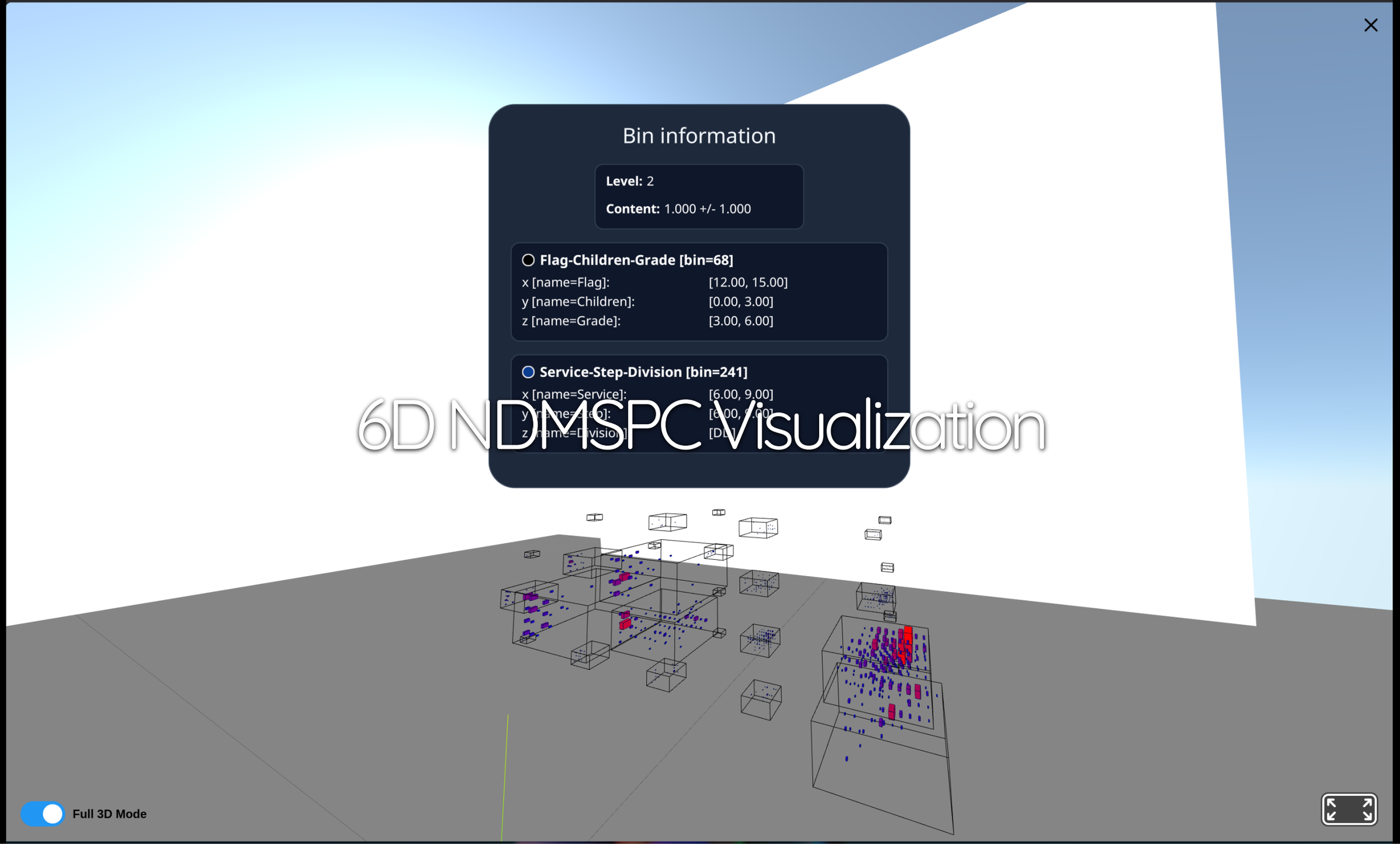
The result is meant to be looked at — in the browser, and in 3D.

NAVIGATOR

- `Ndmspc::NGnNavigator` class
- **Reshape** — `{{1,4,5},{3,6,9}}`
- **Export** — JSON, websocket

NDMVR

- N-dimensional visualization in **3D / VR**
- React library `@ndmspc/ndmvr`, `<NdmspcEnv />` component
- Interactions with custom functions
- The NDMSPC user interface


Bin information
Level: 2
Content: 1.000 +/- 1.000
Flag-Children-Grade [bin=68]
x [name=Flag]: [12.00, 15.00]
y [name=Children]: [0.00, 3.00]
z [name=Grade]: [3.00, 6.00]
Service-Step-Division [bin=241]
x [name=Service]: [6.00, 9.00]
6D NDMSPC Visualization
Full 3D Mode

`ndmspc-server` — one process, every front end

The same binary is what a room runs and what a local install starts.

WHAT IT SERVES

Web page + static assets	`/`
HTTP API	`/api/…`
WebSocket	`/ws/root.websocket`
MCP endpoint	`POST /api/mcp`
Room router	`--rooms`
Tools	handlers from tool macros

The base actions are built in; everything else is a macro you pass with `-m`.

THE BINARIES

- `ndmspc-server` — the process above
- `ndmspc-run` — run a macro with N workers
- `ndmspc-worker` — attach to an IPC/TCP supervisor
- `ndmspc-mcp` — MCP over stdin/stdout

```
ndmspc-server                 # port 8080, default tools
ndmspc-server -p 9000         # another port
ndmspc-server -m myTool.C     # your macro instead
ndmspc-run --mode ipc -n 4 m.C
```

MCP, agents, and your own tool

The server already speaks MCP — so a model can drive the same live session, and the tools you write are what it sees.

MCP endpoint

- Model Context Protocol over JSON-RPC 2.0
- `initialize` · `tools/list` · `tools/call`
- `POST /api/mcp`, or stdio via `ndmspc-mcp`
- The tools **are** the UI's handlers — same session

An agent on top

- Any MCP client: an editor, or a terminal agent
- It works on the **live session** — the file you opened is the file it reshapes
- The model sees **only** what you register
- Cost is the model's; the server stays yours

Your tool

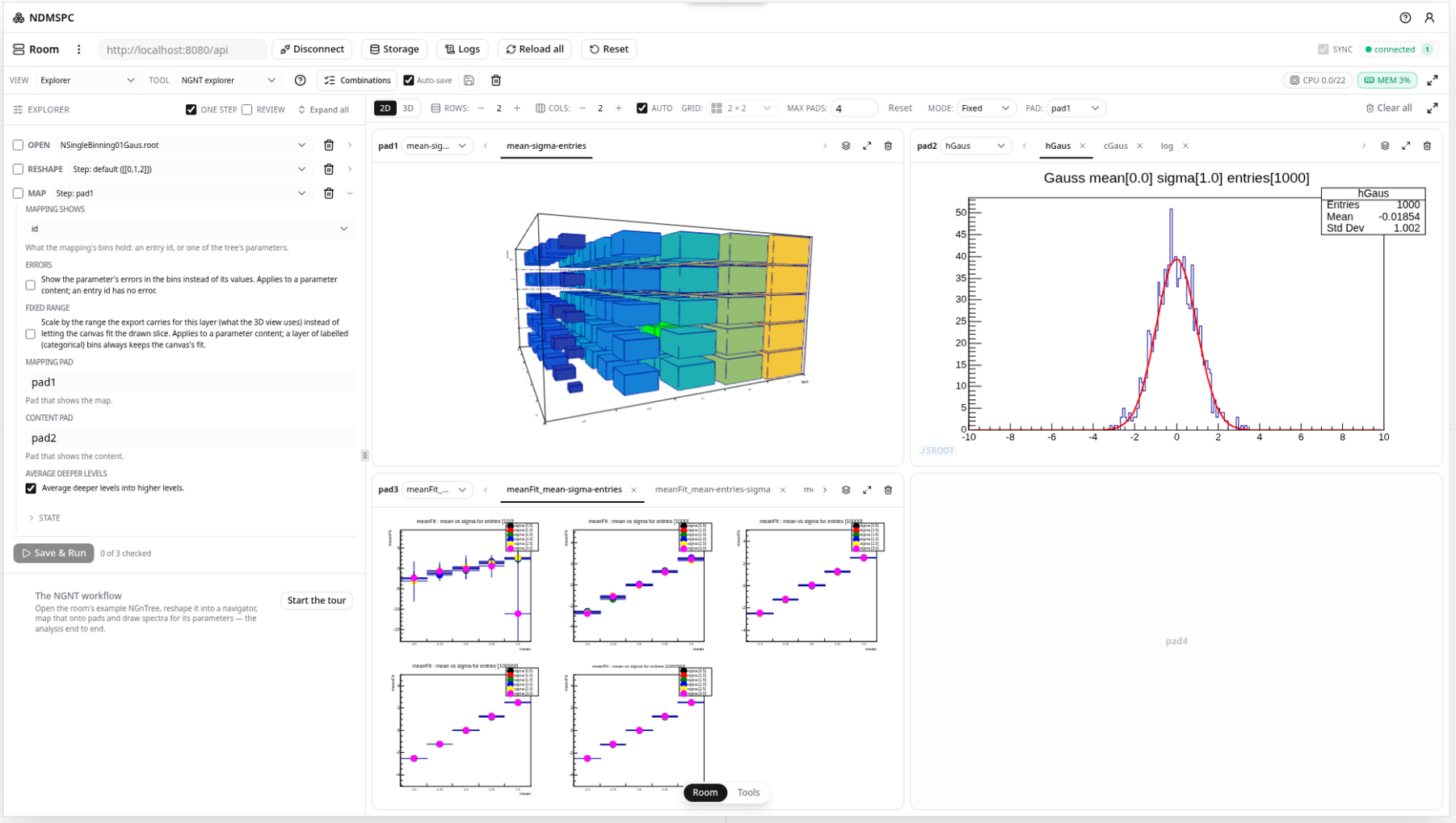
- A ROOT macro calling `RegisterMcpTool`
- Key `<ns>/<group>/<action>` → route, tool name, group
- `inputSchema` → the form the UI renders
- `dependsOn` → an ordered pipeline

```
Ndmspc::RegisterMcpTool("myname/gaus/generate", {
  .description = "Fill a Gaussian and fit it.",
  .methods     = {"POST"},
  .inputSchema = {{"properties",
    {{"entries", {{"type", "integer"},
                  {"title", "Entries"},
                  {"default", 10000}}},
     {"sigma",   {{"type", "number"},
                  {"title", "Sigma"},
                  {"default", 1.0}}}}}},
});
```

Register the handler once; the transports are derived. The same declaration gives you the HTTP route, the WebSocket action and the MCP tool — and the form the browser renders. Nothing in the C++ server is recompiled.

Open → reshape → map → spectra

The same Gaussian example in the NDMSPC UI — one bin clicked to show its stored fit in the second pad.



Summary

THE CORE

- **Mapping & storage** — `NGnTree`: a `THnSparse` map over a `TTree`
- **Binning** — one base binning, rebinnings, many definitions; nothing is rewritten
- **Per bin** — projections, ranges, parameters; the result stays with its bin
- **Processing** — one function per bin, across every binning
- **Execution** — `single`, `ipc`, `tcp`

THE FRONT ENDS

- **`ndmspc-server`** — one process serving the page, the API, the WebSocket and MCP
- **The UI** — open → reshape → map → spectra, on the live session
- **NDMVR** — the same data in 3D and VR
- Every front end is the **same session**, not a copy

EXTENDING IT

- A tool is a **macro** calling `RegisterMcpTool` — no C++ recompile
- One declaration → **HTTP route, WebSocket action, MCP tool and the UI form**
- So an **agent** can drive your tools on the live session, with only what you register

One dataset · many views · one process function · laptop to cluster

Documentation: `ndmspc.gitlab.io/docs`

Thank you for your attention!

Questions, or a tool of your own to show?

`gitlab.com/ndmspc`