# Contents



# CubeFit — LOSVD-convolved stellar population fitting for IFU cubes

CubeFit builds a **hypercube of convolved template spectra** for every `(spaxel S, component C, population P)` and solves for a global, non-negative mixture that best explains the observed IFU data cube. It is designed for large datasets (10^3–10^5 spaxels, 10^2–10^3 populations), is robust to restarts, and uses HDF5 end-to-end with streaming, tile-aligned I/O.

- **Hypercube builder:** FFT-based LOSVD convolution on the template grid, followed by flux-conserving rebin to the observed grid.
- **Normalization:** choose “model” (LOSVD amplitude) or “data” (per-spaxel observed flux with LOSVD-proportional splits). Convert between modes later without a rebuild.
- **Line emphasis:** optional λ-weights so absorption features drive the fit.
- **Solver:** multiprocess Kaczmarz with diagonal preconditioning, trust region, backtracking, global step caps, near-zero column freezing, optional de-correlation nudge, and a tile-local **weighted NNLS polish**.
- **Tracking:** streaming fit metrics and optional snapshots to a SWMR-friendly sidecar.
- **Utilities:** chunk-aware normalization conversion; global column energy; reconstruction and plotting helpers that never touch the giant `/HyperCube/models` unless needed.

---

## HDF5 layout (what lives where)

Core inputs:

- `/Templates` — `(P, T)` template spectra on the template grid (time domain).
- `/TemPix` — `(T,)` template grid in log-λ (natural log).
- `/ObsPix` — `(L,)` observed wavelength grid (log-λ).

- `/R_T` — `(T, L)` (or `(L, T)`) **flux-conserving** linear rebin operator mapping the template grid to the observed grid.
- `/LOSVD` — `(S, V, C)` LOSVD histograms (per spaxel and component).
- `/VelPix` — `(V,)` velocity grid in km/s (for LOSVD).

Optional inputs:

- `/Mask` — `(L,)` boolean wavelength mask; used consistently by builder and solver.
- `/HyperCube/data_flux` — `(S,)` masked mean data flux per spaxel (required for `norm="data"`).

Built artifacts:

- `/HyperCube/models` — `(S, C, P, L)` float32 convolved+rebinned spectra, chunked for streaming and resumable via a `_done` bitmap.
- `/HyperCube/col_energy` — `(C, P)` float64 global column energy `E[c,p] = Σ_{s,λ in mask} A^2`, for step-size control.
- `/HyperCube/norm/losvd_amp` — `(S, C)` LOSVD amplitude (sum or trapz).
- `/HyperCube/norm/losvd_amp_sum` — `(S,)` per-spaxel sum of amplitudes.
- `/HyperCube/lambda_weights` — `(L,)` optional λ-weights in `[floor, 1]` (generated by a median-DoG heuristic).

Fit outputs:

- `/X_global` — `(C*P,)` solution vector, row-major `(C, P)` flattened.

---

## Manager: creating and populating the HDF5 backbone

`hdf5_manager.py` provides:

- **Safe file open** with retries, optional SWMR, and lock-handling (`open_h5`).
- **Dataset creation** for the core arrays and dimensions (`H5Manager` + `H5Dims`).
- **Population** from NumPy arrays (`populate_from_arrays`), including grid checks.
- **Flux-conserving rebin** operator construction (`ensure_rebin_and_resample`).

Typical setup:

```python
from CubeFit.hdf5_manager import H5Manager, H5Dims

mgr = H5Manager("galaxy.h5", tem_pix=tem_loglam, obs_pix=obs_loglam)
mgr.init_base(H5Dims(nSpat=S, nLSpec=L, nTSpec=T, nVel=V, nComp=C, nPop=P))
mgr.populate_from_arrays(losvd=H_SVC, datacube=Y_SL, templates=T_PT)
mgr.ensure_rebin_and_resample()  # builds /R_T (and validates shapes)
```

Notes:

- `/R_T` is built by exact bin overlap in log-λ, preserving flux.
- `/Mask` (if present) is used later by both builder and solver.
- Manager helpers centralize chunk-cache setup so downstream is fast.

---

## Hypercube builder

`hypercube_builder.py` constructs `/HyperCube/models` in streaming, tile-aligned passes:

1. **Kernel from LOSVD:** each `(s,c)` LOSVD row □ **unit-area** kernel on the template grid (linear interpolation to integer pixel shifts). A separate scalar **amplitude** is computed per `(s,c)` (either sum or trapezoidal integral).
2. **Convolution (per p):** FFT-multiply template by kernel and crop the linear convolution back to the template length.
3. **Rebin to observed grid:** multiply by `/R_T` to get `(P, L)` for each `(s,c)`.
4. **Normalization:**
   - `norm="model"`: multiply each `(s,c,p,:)` by its LOSVD amplitude.
   - `norm="data"`: compute per-spaxel mean observed flux (from `/HyperCube/data_flux`), then split across components in proportion to their LOSVD amplitudes, preserving the (c) ratios determined by LOSVD at that spaxel.
5. **Global column energy:** simultaneously accumulate `E[c,p]=□_{s,□ in mask} A²` into `/HyperCube/col_energy` for solver preconditioning.

**Resumability & chunking:** the builder creates `/HyperCube/models` with chunks `(S_chunk, C_chunk, P_chunk, L)` and maintains a `_done` bitmap over the `(S_chunk, C_chunk, P_chunk)` tile grid; this lets you resume safely or inspect progress mid-build. Metadata is flushed regularly.

---

## Normalization conversion (post-hoc)

Flip between `norm="data"` and `norm="model"` after the build:

```
from CubeFit.hypercube_builder import convert_hypercube_norm
convert_hypercube_norm("galaxy.h5", to_mode="model", recompute_energy=True)
```

This reads `/HyperCube/norm/losvd_amp_sum` (and `/HyperCube/data_flux` when needed), computes a **per-spaxel scalar** `F[s]`, and scales `/HyperCube/models[s,:,:,:] *= F[s]` in **S-tiles** (chunk-aligned). It updates `/HyperCube/col_energy` (optional) and flips the `/HyperCube` attribute `norm.mode`. It also respects `/Mask` when deriving or validating the spaxel flux vector.

This is a safe, **no-FFT** operation. Use it to avoid rebuilding the hypercube when only a normalization flip is required.

---

## λ-feature emphasis (line weights)

Absorption features can be down-weighted in plain least squares. To emphasize them, the solver supports **λ-weights**:

- Build weights once via a **median spectrum** and a two-scale smoothing **difference-of-Gaussians** (implemented with boxcars), then map to `[min_w, 1]`. Store at `/HyperCube/lambda_weights`.
- During solving, multiply both **residuals** and **design matrix columns** by `sqrt(w)` — equivalent to solving in a diagonal metric where line pixels matter more.

Helpers in `cube_utils.py`:

```
from CubeFit.cube_utils import ensure_lambda_weights, read_lambda_weights

w = ensure_lambda_weights("galaxy.h5")  # writes /HyperCube/lambda_weights
w2 = read_lambda_weights("galaxy.h5")   # reads, with floor & mask handling
```

At solver startup you'll see a banner printing `min/max/mean` of the λ-weights used (after masking).

## Solver: multiprocess batched Kaczmarz (+ weighted NNLS polish)

Entry point:

```python
from CubeFit.pipeline_runner import PipelineRunner

runner = PipelineRunner("galaxy.h5")
x_global, stats = runner.solve_all_mp_batched(
    epochs=E,
    pixels_per_aperture=...,          # row batching (if used)
    lr=...,                           # base learning-rate
    project_nonneg=True,              # x >= 0
    processes=..., blas_threads=...,  # MP and BLAS knobs
    orbit_weights=...,                # optional (C,) ratio guidance
    # …other optional arguments…
)
```

High-level flow (`pipeline_runner.py` □ `kaczmarz_solver_cchunk_mp.py`):

1. **Tile scheduling:** sort S-tiles by data norm (coarse "hardest first" ordering).
2. **Workers:** each process takes a contiguous **c-band** for the active S-tile.
3. **Inside a worker (per band):**
   - **Sanitize** non-finite values in A and R to zeros.
   - **Apply λ-weights** by computing gradients and denominators on `□w · A` and `□w · R` (but updates and residuals are returned in the unweighted space).
   - **Diagonal preconditioning:** per-population step `dx_p □ g_p / □ (A_w)^2`, blended with global column energy `E[c,p]` for stability.
   - **Near-zero column freeze** (relative & absolute thresholds).
   - **Trust region** on ΔR in weighted space (`||□R_w|| ≤ □ ||R_w||`) + **backtracking** for monotone weighted RMSE drop.
   - **Optional de-correlation nudge** to avoid chasing flat null-space directions across P.
   - **Candidate set** for **tile-local weighted NNLS polish** (top-K by score).
4. **Parent process:**
   - Aggregates ΔR over bands and runs **tile-level backtracking**.
   - Applies a **global step cap** using `□ dx² · E[c,p]` vs. `||Y||`.
   - Optionally runs **NNLS polish** on the proposed (c,p) set using a λ-subsample (rows) and accepts only if it reduces **weighted RMSE** enough.
   - Commits the single global α for the tile: update R and x, enforce `x ≥ 0`.
   - Optional **ratio penalty** step nudges the component totals toward `orbit_weights` (if provided).
   - Streams metrics to `FitTracker` and snapshots (optional).

The result is written to `/X_global` (flattened `(C*P,)`), along with run stats.

---

## Reconstruction & plots (without storing a full ModelCube)

You can reconstruct `Ŷ = A·x` for diagnostics without materializing a full `/ModelCube`:

- **Reconstruction utilities** read A in the same chunk order and contract against x in λ-bands. They write a small target array you name (e.g., `/ReconTile`) or return the result.
- **Parallel spectral plots** pull just the selected rows from `/DataCube` and the reconstructed array and produce side-by-side data/model spectral plots for worst/best spaxels.

Both are tile-aligned and cache-aware; they never read /HyperCube/models beyond the needed slices.

---

## Fit tracking & snapshots

fit_tracker.py provides a **non-blocking** sidecar writer process:

- Destination: a sidecar H5 next to the main file (*.fit.<pid>.<ts>.h5).
- Bounded mp.Queue so the solver never blocks on I/O.
- SWMR enabled for dashboards.
- Streams **RMSE history**, **EWMA**, **progress**, and optional **x snapshots** (best/last/history).

Use tracker=FitTracker(...) in the solver; the runner wires this up for you.

---

## Utilities

- **Normalization conversion:** convert_hypercube_norm(h5, to_mode="model"|"data", recompute_energy=True) — chunk-aligned, mask-aware.
- **Global column energy:** created during build; can be recomputed later if you changed normalization.
- **Done-bitmap control:** functions to inspect and invalidate _done to resume or redo specific tiles.
- **Recompression:** rewrite /HyperCube/models with different compression/filter options (tile-wise copy).

---

## Environment variables (knobs)

These can be exported to tune performance, stability, or behavior. Defaults shown are what the code uses if you don't set them.

### HDF5 & dataset caching

| Variable | Default | Purpose |
|---|---:|---|
| HDF5_USE_FILE_LOCKING | FALSE | Avoid HDF5 file locking issues on shared filesystems. |
| CUBEFIT_RDCC_NSLOTS | 400003 | File-level raw chunk cache: number of slots. |
| CUBEFIT_RDCC_NBYTES | 4294967296 (4 GiB) | File-level raw chunk cache size in bytes. |
| CUBEFIT_RDCC_W0 | 0.9 | File-level raw chunk cache preemption policy. |
| CUBEFIT_RDCC_SLOTS | 1000003 | Per-dataset cache slots used by some readers. |
| CUBEFIT_RDCC_BYTES | 268435456 (256 MiB) | Per-dataset cache size used by some readers. |

### Multiprocessing & BLAS/OpenMP

| Variable | Default | Purpose |
|---|---:|---|
| CUBEFIT_MP_CTX | forkserver (fallback spawn) | MP start method for the solver pool. |
| OMP_NUM_THREADS | set by code | BLAS/OpenMP threads per worker (also OPENBL |
| KMP_INIT_AT_FORK | FALSE | Avoid OpenMP deadlocks with fork. |

| Variable | Default | Purpose |
|---|---|---|
| `SLURM_CPUS_PER_TASK` | (if present) | Used by reconstruction helpers to set BLAS threac |
| `PYTHONUNBUFFERED` | `1` (recommended) | Unbuffered stdout for real-time logs on HPC. |

#### Solver: worker-level stability & steps (in `kaczmarz_solver_cchunk_mp.py`)

| Variable | Default | Purpose |
|---|---|---|
| `CUBEFIT_BT_STEPS` | `3` | Backtracking steps per band (worker) for monotone weighted RM |
| `CUBEFIT_BT_FACTOR` | `0.5` | Multiplicative α shrink in backtracking. |
| `CUBEFIT_TRUST_TAU` | `0.7` | Trust-region cap: ‘ |
| `CUBEFIT_EPS` | `1e-12` | Numerical floor in denominators and divisions. |
| `CUBEFIT_ZERO_COL_REL` | `1e-12` | Relative threshold (vs median energy) to **freeze** near-zero colum |
| `CUBEFIT_ZERO_COL_ABS` | `1e-24` | Absolute threshold to **freeze** columns. |
| `CUBEFIT_DEBUG_SAFE` | `0` | Print sanitization/freeze stats per tile if `1`. |
| `CUBEFIT_NNLS_PROP_PER_BAND` | `6` | How many (c,p) candidates to propose for NNLS polish per band |

#### Solver: aggregation & NNLS polish (in `kaczmarz_solver_cchunk_mp.py`)

| Variable | Default | Purpose |
|---|---|---|
| `CUBEFIT_TILE_BT_STEPS` | `6` | Backtracking steps for the **aggregated** ΔR across bands |
| `CUBEFIT_TILE_BT_FACTOR` | `0.5` | Aggregated α shrink factor. |
| `CUBEFIT_GLOBAL_TAU` | `0.5` | Global step cap using □ `dx²·E[c,p]` vs ‘ |
| `CUBEFIT_GLOBAL_ENERGY_BLEND` | `1e-3` | Blend local `□(A_w)^2` with global column energy `E[c` |
| `CUBEFIT_NNLS_ENABLE` | `1` | Enable tile-local **weighted** NNLS polish. |
| `CUBEFIT_NNLS_EVERY` | `1` | Run NNLS polish every N tiles. |
| `CUBEFIT_NNLS_MAX_COLS` | `128` | Max columns K in the NNLS system. |
| `CUBEFIT_NNLS_MAX_BYTES` | `1000000000` | Approx memory cap for NNLS system (rows×8 per colu |
| `CUBEFIT_NNLS_SUB_L` | `0` | λ-subsample size for NNLS (0 □ use all). |
| `CUBEFIT_NNLS_SOLVER` | `nnls` | One of `nnls`, `lsq`, `mu`, `fista` (FISTA recommended f |
| `CUBEFIT_NNLS_MAX_ITER` | `50` | Iteration cap for iterative NNLS (`mu`, `fista`, `lsq`'s TRF |
| `CUBEFIT_NNLS_MIN_IMPROVE` | `0.999` | Required weighted-RMSE ratio to accept polish (< this a |

#### Solver: λ-weights (in `kaczmarz_solver_cchunk_mp.py` / `cube_utils.py`)

| Variable | Default | Purpose |
|---|---|---|
| `CUBEFIT_LAMBDA_WEIGHTS_ENABLE` | `1` | Enable λ-weights for solver (□ |
| `CUBEFIT_LAMBDA_WEIGHTS_DSET` | `/HyperCube/lambda_weights` | Dataset to read. |
| `CUBEFIT_LAMBDA_MIN_W` | `1e-6` | Floor on λ-weights; protects v |
| `CUBEFIT_LAMBDA_WEIGHTS_AUTO` | `1` | Auto-generate weights if miss |

#### Fit tracking (in `fit_tracker.py`)

| Variable | Default | Purpose |
|---|---|---|
| `FITTRACKER_START` | `spawn` (or best available) | Start method for the tracker process (`spawr` |

| Variable | Default | Purpose |
|---|---|---|
| `CUBEFIT_TRACKER_QSIZE` | 8192 | Max queue size; drops messages when full if |
| `CUBEFIT_TRACKER_FLUSH_EVERY` | 128 | Flush to disk after this many messages. |
| `CUBEFIT_TRACKER_FLUSH_SEC` | 5.0 | Or flush if this many seconds have passed. |
| `CUBEFIT_RMSE_STRIDE` | 16 | Only enqueue every Nth batch RMSE sample |

---

## Typical end-to-end workflow

```python
# 1) Build HDF5 backbone & rebin operator
mgr = H5Manager("galaxy.h5", tem_pix=tem_loglam, obs_pix=obs_loglam)
mgr.init_base(H5Dims(nSpat=S, nLSpec=L, nTSpec=T, nVel=V, nComp=C, nPop=P))
mgr.populate_from_arrays(losvd=H_SVC, datacube=Y_SL, templates=T_PT)
mgr.ensure_rebin_and_resample()

# 2) Build hypercube
from CubeFit.hypercube_builder import build_hypercube
build_hypercube("galaxy.h5", norm_mode="data")  # or "model"

# (Optional) Flip normalization later without rebuild
from CubeFit.hypercube_builder import convert_hypercube_norm
convert_hypercube_norm("galaxy.h5", to_mode="model", recompute_energy=True)

# 3) Create λ-weights
from CubeFit.cube_utils import ensure_lambda_weights
ensure_lambda_weights("galaxy.h5")

# 4) Solve
from CubeFit.pipeline_runner import PipelineRunner
runner = PipelineRunner("galaxy.h5")
x, stats = runner.solve_all_mp_batched(
    epochs=6, lr=0.6, project_nonneg=True, processes=8, blas_threads=2,
    orbit_weights=orbit_w  # optional (C,)
)
```

---

## Notes on normalization & physics

- `norm="model"` keeps each `(s,c)` model's scale proportional to the LOSVD amplitude — closest to "physical mass/flux" carried by LOSVD at that spaxel.
- `norm="data"` ties the per-spaxel model sum to the observed mean flux (masked), splitting across components in proportion to LOSVD amplitudes. This is convenient for direct data-scale comparisons but can blur absolute differences across datasets. You can switch modes post-hoc with `convert_hypercube_norm`.

The solver respects `/Mask` and λ-weights uniformly, so **absorption features** can dominate the fit when λ-weights are present (verify the printed min/max/mean at startup to ensure they are not all ones).

---

## Troubleshooting & sanity checks

- **Hypercube completeness:** use the `_done` bitmap helpers to inspect or reset tiles if needed.
- **Normalization flips:** use `convert_hypercube_norm` (streams in S-tiles and handles `/Mask`); do **not** read the entire dataset into RAM.
- **Line weights:** ensure `/HyperCube/lambda_weights` exists and spans a range (e.g., min ▯ 1, max ≈ 1); the solver prints min/max/mean.
- **Population mixture looks uniform across components:**
  - Confirm λ-weights are applied (banner shows they're not all 1).
  - Increase epochs modestly; keep `GLOBAL_ENERGY_BLEND` small but nonzero.
  - Enable NNLS polish with a moderate `CUBEFIT_NNLS_MAX_COLS`, a λ-subsample (`CUBEFIT_NNLS_SUB_L`), and `CUBEFIT_NNLS_SOLVER=fista` for sharper per-tile updates.

---

## Where to look in the code

- **HDF5 backbone:** `hdf5_manager.py` (safe open, grids, rebin operator).
- **Builder:** `hypercube_builder.py` (FFT conv, normalization, col_energy, chunking + resume).
- **Solver:** `kaczmarz_solver_cchunk_mp.py` (MP loop, λ-weights, trust/backtracking, NNLS polish).
- **Runner:** `pipeline_runner.py` (orchestration, tracking, `/X_global`).
- **Tracking:** `fit_tracker.py` (sidecar writer, RMSE history, progress, snapshots).
- **Recon/plots:** utilities in `kz_fitSpec.py` (tile-aligned recon) and safe plotting helpers.

---

## Building the docs

Place this file at `docs/CubeFit.md` and run:

```
make             # builds docs/CubeFit.html and docs/CubeFit.pdf via pandoc
mkdocs serve     # live preview with Material theme (served from docs/)
```

Pandoc targets (from your Makefile):

- `docs/CubeFit.html` via:
  `pandoc -s -f gfm -t html5 docs/CubeFit.md -o docs/CubeFit.html --metadata title="CubeFit" --toc`
- `docs/CubeFit.pdf` via XeLaTeX:
  `pandoc -s -f gfm docs/CubeFit.md -o docs/CubeFit.pdf --pdf-engine=xelatex -V geometry:margin=1in -V mainfont="Latin Modern Roman" -V monofont="Latin Modern Mono" --toc`

MkDocs (from your `mkdocs.yml`):

```yaml
site_name: CubeFit
theme:
  name: material
nav:
  - Home: index.md
  - CubeFit: CubeFit.md
docs_dir: docs
```