YAMC

YAMC (Yet Another Monte Carlo) is an open-source Monte Carlo particle transport code for neutron and photon simulations, aimed at fusion neutronics workflows.

It supports constructive solid and mesh-based geometries, coupled neutron-photon transport, transmutation, shutdown dose rate analysis, CAD geometry import, flexible tallies, and convenient post-processing, all accessible through a Python API.

YAMC's priorities are:

- First-class support for fusion neutronics workflows
- Runs on Linux, macOS, and Windows, on x86 and ARM CPUs and AMD, NVIDIA, and Intel GPUs, and in the browser via WebAssembly
- Straightforward installation with `pip`, no compilation step
- A fully permissively licensed open-source dependency stack
- Batteries included: CAD geometry meshing, transmutation, convenient cross section data downloading, and visualization built in

What each of those means in practice is on Why YAMC.

Explore

- Why YAMC: what it does well, and where to read more about each
- Installation: install with `pip`, no compilation step
- Quick start: a minimal simulation end to end
- Usage guide: materials, geometry, sources, tallies, transmutation, CAD
- Nuclear data libraries: the six you can choose between, and how much is in each
- Python API: the type-annotated reference, generated from the package's stubs

To ask questions or get support, try the GitHub Discussions page.

Why YAMC

YAMC is a Monte Carlo neutron and photon transport code for fusion. These are the things it does well.

Easy

YAMC online runs the same transport engine in your browser, on your own machine, with no account and no upload, so a first run needs nothing installed.

`pip install yamc` on Linux, macOS or Windows, on x86 or ARM. The solver is a Rust core shipped inside the wheel behind thin PyO3 bindings that target Python's stable ABI (3.10+), so nothing is compiled at install time. The dependency-free core also builds to WebAssembly, which is what YAMC online runs. See Installation.

Nuclear data downloads itself. Pick a library and only the parts your model needs are fetched and cached, down to just the temperatures you set on your materials. To work offline, download the full data files and point YAMC at the directory. See Nuclear data.

YAMC is MIT or Apache-2.0, and nothing in its dependency stack is copyleft, including the GPU path.

`model.to_html()` writes one file that runs the simulation in any browser, for a colleague without Python or a demo in a course. See Export to a runnable HTML page.

Accurate

Cross sections are used on the evaluation's own energy grid, with no multigroup collapse. See Method.

Statistics are kept per history, with no batches to choose, and each tally result carries automatic reliability checks on its error, variance of variance and convergence. See Statistical reliability.

A transmutation can return every nuclide density, activity, decay heat and dose with an uncertainty propagated from the evaluation's own covariance data. See Nuclear-data uncertainty.

Fast

`compute="gpu"` runs on AMD, NVIDIA or Intel GPUs in double precision, through one cubecl/Vulkan code path rather than a proprietary single-vendor toolkit. See GPU transport.

Weight windows are generated automatically from the model, and delta tracking and survival biasing are one setting each. See Weight windows.

Set a target relative error on a tally and the run ends as soon as it is met. See Stopping on a precision target.

The dose at any cooling time, for any irradiation schedule, comes from a single transport run. See Shutdown dose rate.

Flexible

YAMC includes its own CAD mesher and mesh transport, so a CadQuery assembly or STEP file goes straight into a model, alongside or inside CSG cells. See Geometry.

A tokamak plasma source comes from plasma parameters, and tallies cover heating, DPA, tritium production and dose. See Sources and Tallies.

Materials can be depleted coupled to a transport run, or from a spectrum you already have, giving activity, decay heat, decay photons and waste clearance. See Simulate transmutation.

Cross sections, reactions, decay data and fission yields can each come from a different library, or from your own files. See Nuclear data settings.

Nuclear data, tally results and meshes are stored and exchanged as Apache Arrow, which is columnar and language-agnostic, so data files are portable and load quickly. Tallies convert to numpy, pandas, polars or xarray with little or no copying, and export to VTK-HDF for ParaView. See Results.

Installation

YAMC can be installed and used on Linux, Windows and MacOS.

Standard Installation (Recommended)

YAMC is a Rust back end package with a Python API. Installation requires no additional Python package dependencies. YAMC is available on PyPI.

Install from PyPI

To install, make sure you have pip installed, then run:

```
pip install yamc
```

This installs the core YAMC transport solver with multi-threading support (via Rayon). MPI is not required.

Optional extras

YAMC provides optional extras for additional functionality:

Extra	Install command	What it adds
cad	`pip install "yamc[cad]"`	CAD meshing pipeline, surface and tetrahedral (requires cadquery)
viz	`pip install "yamc[viz]"`	VTKHDF visualization export (h5py, numpy)
pandas	`pip install "yamc[pandas]"`	pandas DataFrame export of tally results
polars	`pip install "yamc[polars]"`	polars DataFrame export of tally results
xarray	`pip install "yamc[xarray]"`	xarray (labelled N-D array) export of tally results
all	`pip install "yamc[all]"`	cad and viz (does not include pandas, polars or xarray)

Extras combine, as in `pip install "yamc[all,pandas]"`. The quotes stop shells such as zsh, the default on macOS, from reading the square brackets as a file name pattern.

To install the CAD-to-mesh-to-transport pipeline and the visualization export together:

```
pip install "yamc[all]"
```

Virtual Environment (Recommended)

Further instructions are in the Python docs.

```
python3 -m venv .venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
pip install "yamc[all]"
```

Install the development version

`pip install yamc` gives the latest release. The unreleased changes on the main branch build from source, which needs a Rust toolchain:

```
pip install "yamc @ git+https://github.com/fusion-neutronics/yamc-and-yani.git#subdirectory=packages/yamc"
```

Check which build is installed

`yamc.__version__` is the installed `yamc` version. `yamc.__commit__` is the full git hash of the core commit the extension was built from, and `yamc.__commit_dirty__` is `True` if tracked files differed from that commit when it was built. Both are `None` for a build made outside a git checkout.

```
python -c "import yamc; print(yamc.__version__, yamc.__commit__,
yamc.__commit_dirty__)"
```

A released wheel reports the version of its release. A build from a core source checkout (`pip install -e`, `maturin develop` or a `git+` install, such as the MPI builds below) reports `0.0.0.dev0`, and its commit identifies it instead.

MPI Installation (Advanced)

YAMC can be built with MPI support for distributed-memory parallelism across multiple nodes, for example on large clusters.

MPI is optional, and the standard installation above is enough for most users. Install MPI only if you need: - Multi-node parallelism (HPC clusters) - Scaling beyond a single machine - Hybrid MPI + threading parallelism

Prerequisites

First, install an MPI implementation on your system:

Ubuntu/Debian

```
sudo apt-get update && sudo apt-get install -y \
    libopenmpi-dev openmpi-bin \
    build-essential clang libclang-dev
```

Fedora/RHEL/CentOS

```
sudo dnf install openmpi-devel
# or
sudo yum install openmpi-devel

# Load MPI module (if using environment modules)
module load mpi/openmpi-x86_64
```

macOS (Homebrew)

```
brew install open-mpi
```

Windows (Microsoft MPI)

1. Download and install Microsoft MPI
2. Install both `msmpisetup.exe` (runtime) and `msmpisdk.msi` (SDK)

Verify MPI Installation

```
# Check MPI compiler is available
mpicc --version

# Test MPI works
mpirun -np 2 hostname
```

Install YAMC with MPI

Building with MPI requires the Rust toolchain. To install Rust:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

Then build `yamc` from source, with the `mpi` feature switched on. `MATURIN_PEP517_ARGS` passes the flag to maturin, which builds the wheel for pip:

```
MATURIN_PEP517_ARGS="--features mpi" pip install "yamc @
git+https://github.com/fusion-neutronics/yamc-and-
yani.git#subdirectory=packages/yamc"
```

This builds the main branch. To build a release instead, put its tag after `yamc-and-yani.git@`, as in `git+https://github.com/fusion-neutronics/yamc-and-yani.git@<tag>#subdirectory=packages/yamc`. The repository's tags are named after the versions they release. A build from source reports version `0.0.0.dev0`; `yamc.__commit__` gives the commit it was built from (see Check which build is installed).

Verify MPI Installation

```
# Check MPI is available in YAMC
python -c "import yamc; print(f'MPI enabled: {yamc.parallel.has_mpi()}')"
# Should print: MPI enabled: True

# Test with multiple ranks
mpirun -np 2 python -c "import yamc; print(f'Rank {yamc.parallel.mpi_rank()} of
{yamc.parallel.mpi_size()}')"
# Should print, the two lines in either order:
# Rank 0 of 2
# Rank 1 of 2
```

Docker Installation

Standard Docker Image

```
FROM python:3.11-slim

# cadquery, which the cad extra brings, needs the system OpenGL library to import
RUN apt-get update && apt-get install -y --no-install-recommends libgl1 \
    && rm -rf /var/lib/apt/lists/*

RUN pip install "yamc[all]"

WORKDIR /workspace
CMD ["python"]
```

Docker Image with MPI

```
FROM ubuntu:22.04

# Install system dependencies
RUN apt-get update && apt-get install -y \
    python3 python3-pip \
    libopenmpi-dev openmpi-bin \
    build-essential clang libclang-dev curl git \
    && rm -rf /var/lib/apt/lists/*

# Install Rust
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
ENV PATH="/root/.cargo/bin:${PATH}"

# Build yamc from source with MPI
RUN MATURIN_PEP517_ARGS="--features mpi" pip3 install \
    "yamc @ git+https://github.com/fusion-neutronics/yamc-and-
yani.git#subdirectory=packages/yamc"

# The container runs as root, and OpenMPI refuses to start as root without these
ENV OMPI_ALLOW_RUN_AS_ROOT=1 OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1

WORKDIR /workspace
CMD ["python3"]
```

Build and run:

```
docker build -t yamc-mpi -f Dockerfile .
docker run --rm yamc-mpi mpirun -np 4 python3 -c "import yamc;
print(yamc.parallel.mpi_rank())"
```

HPC/Cluster Installation

For HPC environments with environment modules:

```
# Load required modules
module load python/3.11
module load openmpi/4.1.4

# Install Rust in your home directory: yamc needs the latest stable
# release, which cluster modules rarely carry
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env

# Create virtual environment
python3 -m venv ~/.venvs/yamc-mpi
source ~/.venvs/yamc-mpi/bin/activate

# Build yamc with MPI
MATURIN_PEP517_ARGS="--features mpi" pip install "yamc @
git+https://github.com/fusion-neutronics/yamc-and-
yani.git#subdirectory=packages/yamc"

# Test installation
mpirun -np 4 python -c "import yamc; print(f'Rank {yamc.parallel.mpi_rank()}')"
```

See simulating for MPI usage, hybrid MPI+threading, and Slurm examples.

Quick start guide

A minimal simulation end to end: configure data, build a material, geometry and source, add a tally, run, and read the result. The example computes tritium production (the basis of the tritium breeding ratio, TBR) in a lithium sphere.

Import YAMC

```
import yamc
```

Configure the nuclear data

This example transports neutrons only, so it needs neutron data. A single library keyword downloads the required data automatically:

```
yamc.neutron_cross_section_data = "endf-b8.1"
```

A run that transports photons also sets `yamc.photon_cross_section_data`; see Neutron and photon cross sections.

Six libraries are available, and they differ in content as well as name; see Nuclear data libraries.

Create a material

Lithium breeds tritium, so it serves as the breeder material (`"Li"` expands to natural Li6 + Li7):

```
mat_lithium = yamc.Material(
    composition={"Li": 1.0},
    density=0.534,
    temperature=294,
)
```

Build the geometry

A lithium sphere bounded by a vacuum surface:

```
surface_sphere = yamc.Sphere(radius=100, boundary="vacuum")
cell = yamc.Cell(region=surface_sphere.below, material=mat_lithium)
geometry = yamc.Geometry(cells=[cell])
```

Build the source

A single float is a monoenergetic source (here the 14.06 MeV D-T neutron):

```
source = yamc.NeutronSource(energy=14.06e6)
```

Add a tally

Score H3 (tritium) production in the lithium sphere. With no spatial filter the tally scores over the whole geometry, which here is one cell. Per source neutron, this is the TBR:

```
tally = yamc.Tally(scores=["H3-production"])
```

With more than one cell, `cells=cell` bins the tally per cell instead; see Spatial binning.

Run the simulation

Pass the tally to the model and run the transport:

```
model = yamc.Model(
    geometry=geometry,
    source=source,
    tallies=[tally],
)
results = model.simulate_transport()  # runs until the statistical checks pass
```

Read the result

`results[tally]` is a labelled result. Its `.mean` is the H3-production rate per source neutron (the TBR for this geometry) and `.standard_deviation` is its statistical uncertainty. Both are lists with one entry per score:

```
result = results[tally]
print(f"TBR (H3 production per source neutron): {result.mean[0]} +/-
{result.standard_deviation[0]}")
```

For a pure-lithium sphere this prints a TBR of roughly:

```
TBR (H3 production per source neutron): 1.2169485390409323 +/- 0.002059927598397041
```

Usage guide

After installing and importing the package, the Python API lets you build models, run transport, and analyse the results.

```
import yamc
```

This guide is organised by topic:

- Nuclear data: configuring and inspecting cross sections
- Materials: composition, density, properties, mixing
- Geometry: CSG surfaces and cells, and CAD import
- Sources: spatial, energy, and angular distributions
- Tallies: scores, estimators, filters, and meshes
- Models: assembling a model and model function
- Weight windows: variance reduction for deep-penetration and streaming
- Simulating transport: run a particle transport simulation
- Simulating transmutation: run transmutation simulations
- Simulating shutdown dose: run shutdown (residual) dose simulations
- Results and post-processing: statistics, plotting, and export

Nuclear data libraries

Six libraries are available by keyword. Point a setting at one and the sections it needs are downloaded on first use from the library's current published release, verified against that release's manifest, and cached on disk under `yamc.cache_dir()`. Releases, the cache layout and offline use are described in Nuclear data.

```
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"
```

keyword	what it is
`endf-b8.1`	ENDF/B-VIII.1.
`jeff-4.0`	JEFF-4.0.
`jendl-5.0`	JENDL-5.
`tendl-2025`	TENDL-2025. The widest isotope coverage of the six, but it has no hydrogen or helium, and its lithium has a tritium production problem: see Hydrogen and helium in TENDL-2025 and Lithium in TENDL-2025.
`tendl-2017`	TENDL-2017, for comparing a result against the older release.
`fendl-3.2d`	FENDL-3.2d. Cross sections, plus `reactions` and `branching` subsections.

Every setting also takes a path instead of a keyword, including a folder that holds a whole library laid out as a release, which serves every setting at once (see A whole library in one folder), and `neutron_cross_section_data` and `photon_cross_section_data` additionally take a directory or a dict, keyed by nuclide and by element respectively; the four `transmutation_*` parts take one value each, since a network is assembled per subsection rather than per nuclide. Which setting does what, and what each does when left alone, is in Nuclear data settings. Each library can also be built locally from its evaluated sources with `yamc.generate_library`, into such a folder (see Building a library yourself).

Asking yamc what a library holds

`yamc.available_nuclear_data_libraries()` answers from each library's current release: a dict keyed by keyword, giving the nuclides with neutron cross sections, which of them carry each kind of covariance, the thermal scattering tables, the elements with photo-atomic data, the targets with photonuclear data, which transmutation subsections it publishes and the upstream files it was built from. It reads each release's manifest, so no data is downloaded, and the manifests are cached after the first call.

```
libraries = yamc.available_nuclear_data_libraries()
libraries["endf-b8.1"]["photon"][:3]          # ['Ac', 'Ag', 'Al']
libraries["tendl-2025"]["photon"]             # []: no photo-atomic data
len(libraries["tendl-2025"]["photonuclear"])  # photonuclear targets
"Fe56" in libraries["jeff-4.0"]["neutron_covariances"]["cross_section"]
libraries["irdff-ii"]["isomeric_branching"]   # True
```

Every list is complete, so an empty one means the library has none of that. A library that could not be looked up (no network and nothing cached) has `None` throughout and the reason under `"error"`. The tables below are the same information, summarised.

What each library publishes

Hover a column heading for what it holds; the list under the table says the same.

> **Cross sections or reactions?**
>
> There is one set of cross sections, and it is the first column: transport samples collisions from it, and transmutation folds the same neutron cross sections with the spectrum to get its reaction rates. The `reactions` subsection is not a second set: it holds no cross sections, only the network of which reaction on which nuclide makes which product. The cross sections decide **how fast** each reaction runs and the network decides **which** reactions and products exist, so an activation run needs both, and the two can come from different libraries. More in The two that both say reactions.

library	cross sections	`decay`	`reactions`	`fission_yields`	`branching`
`endf-b8.1`	neutron + photoatomic	yes	yes	yes	yes
`jeff-4.0`	neutron	yes	yes	yes	yes
`jendl-5.0`	neutron + photoatomic	yes	yes	yes	yes
`fendl-3.2d`	neutron + photoatomic	--	yes	--	yes
`tendl-2025`	neutron + photonuclear (next release, data only)	--	yes	--	yes
`tendl-2017`	neutron + photonuclear (next release, data only)	--	yes	--	yes

- **cross sections**: neutron data per nuclide, set by `neutron_cross_section_data`, and photoatomic data per element, set by `photon_cross_section_data`. Transport samples collisions from them, and the neutron ones also give transmutation its reaction rates.
- **`decay`**, set by `transmutation_decay_data`: half-lives, decay modes, decay energies and decay photon lines.
- **`reactions`**, set by `transmutation_reactions`: which reaction on which nuclide makes which product, and its Q. It decides which activation products can be made at all, and holds no cross sections.
- **`fission_yields`**, set by `transmutation_fission_yields`: which fission products a fission makes, and how many.
- **`branching`**, set by `transmutation_branch_ratios`: how a reaction's product splits between its ground and metastable states.

`yamc.neutron_cross_section_data` selects a library's neutron data and `yamc.photon_cross_section_data` its photoatomic data, independently. The photonuclear entries describe additional evaluated data, not another photon transport library. The neutron setting is not transport-only: `Material.transmute()` and `Model.simulate_transmutation()` fold the same cross sections against the spectrum to get their reaction rates, so an activation run needs it as much as a

transport run does. The four remaining columns are the transmutation subsections, each set independently through its own `yamc.transmutation_*` setting. A library with no subsections still works for transport, it just cannot supply a chain.

TENDL has neutron and photonuclear evaluations, but no decay or fission-yield evaluations. Take those from a library that has them and source `reactions` and `branching` from TENDL, whose parent coverage is far wider. FENDL-3.2d publishes the same two subsections, built against ENDF/B-VIII.1 decay data; its `branching` is the reason to use it, since 54 of its evaluations carry isomer production (MF=9 or MF=10), 15 of them with MF=40 covariances. Pointing a transmutation part at a library that does not provide it fails immediately with a message listing the subsections that library does provide.

`endf-b8.1`, `jeff-4.0` and `jendl-5.0` each supply a complete, single-library transmutation chain.

Photon data is selected separately

`yamc.photon_cross_section_data` selects photoatomic photon data, and `yamc.neutron_cross_section_data` selects neutron data. Both start unset and neither falls back on the other, so a coupled or D1S run sets both, a `PhotonSource` run needs only the photon setting, and a neutron-only run only the neutron one. Photon-only and coupled transport load the required elements automatically; an extra `material.read_nuclear_data(...)` call is not needed when the global setting names the intended library. Material construction itself downloads nothing: data is loaded when a calculation needs it.

```
import yamc

yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"
water = yamc.Material(composition={"H1": 2.0, "O16": 1.0}, density=1.0)
```

Only three libraries provide photoatomic data for photon transport:

- `endf-b8.1` and `jendl-5.0` publish the photoatomic and atomic-relaxation pair a coupled run needs. JENDL-5's is EPICS2017 adopted by JAEA, so it is the same data ENDF/B-VIII.1 carries.
- `fendl-3.2d` publishes photoatomic cross sections with no atomic relaxation, so absorption emits no fluorescence or Auger electrons; the loader warns about missing relaxation where it matters.
- `jeff-4.0` and both TENDL releases publish no photoatomic data at all. JEFF's photon sublibrary is photonuclear, which is a different quantity.

There is no implicit switch to another photon library. A photon run with `photon_cross_section_data = "tendl-2025"` or `"tendl-2017"` fails for missing photoatomic data, even after the new photonuclear files are published. A run that takes its neutron data from TENDL takes its photon data from one of the three libraries above, named in `photon_cross_section_data`. A single material can also be given an explicit per-element `photon_data` map on read_nuclear_data, which takes precedence over the global selection.

Photonuclear data in the next TENDL release

The next TENDL-2025 and TENDL-2017 regeneration adds the gamma evaluations from `TENDL-g.tgz`. These are per-nuclide photonuclear reactions, not the per-element photoatomic interactions above. Release `2026-10-12.1` does not contain this new output; it becomes available only after the regenerated release is validated and published.

The new folders are `photonuclear/{Nuclide}.arrow/`. Each contains:

- `photonuclear.arrow`: the MF=3 cross sections on their original energy grids, with Q values and ENDF interpolation, in eV and barns.
- `photonuclear_sections.arrow`: every original ENDF section, including outgoing-particle distributions.
- `version.json`: provenance identifying `particle="photonuclear"` and `scope="evaluated"`.

`photonuclear/index.json` lists the evaluations, and the release manifest lists and hashes the files. This is data processing, not photonuclear collision sampling: neither `PhotonSource` nor coupled photon transport consumes these tables yet. A new `photonuclear/` folder does not supply a photoatomic `photon/` folder.

How big each library is

Counted from each library's published release `2026-10-12.1`. These counts do not describe the forthcoming regeneration. `--` means that release does not publish the category, rather than publishing an empty one.

library	neutron	photoatomic elements	reaction edges	branching channels	decay nuclides	covariance
`endf-b8.1`	557	100	12,444	917	3,820	48%
`jeff-4.0`	593	--	81,045	8,362	3,851	99%
`jendl-5.0`	795	100	145,117	5,397	4,070	13%
`fendl-3.2d`	192	61	27,814	652	--	35%
`tendl-2025`	2,848	--	415,154	40,934	--	100%
`tendl-2017`	2,801	--	417,434	41,185	--	100%

`neutron` and `photoatomic elements` are evaluations, one per nuclide and one per element. `reaction edges` is how many `(parent, reaction, product)` triples the network holds, which is what decides whether a product is reachable at all. Most of them are the products of `(n,X)` (MT=5), its residuals and light particles, one triple each. `covariance` is the share of neutron evaluations carrying cross-section covariance.

The two ends of the table differ widely. TENDL carries five times the evaluations of ENDF/B-VIII.1 and over thirty times the reaction edges, and has none of the decay data. That is why a TENDL `reactions` network with ENDF/B-VIII.1 `decay` and `fission_yields` beside it is the usual arrangement.

Covariance is presence, not coverage

The percentage counts evaluations that carry cross-section covariance, which is not the same as covariance landing where a given run needs it. Depth varies as much as presence: JENDL-5 carries covariance for 13% of its evaluations, and its Fe56 covers 56 channels against ENDF/B-VIII.1's 7.

Six covariance families reach the converted data, and three of them are sampled. The cross sections' MF=33 is drawn by the `cross_sections` source. Resonance-parameter covariance (MF=32), where an evaluation often keeps its resolved-resonance uncertainty while leaving MF=33 at zero over that range, is used two ways: where a nuclide's folder carries the parameters themselves (`resonance_parameters.arrow`, in the published libraries), each sample draws them and rebuilds the resonance range from the draw; otherwise the converter's first-order MF=32 rows in `covariance.arrow` are folded like MF=33. Data with neither arrives with no capture, elastic or fission uncertainty below the top of its resolved resonances, and the uncertainty report counts rate from energies where the stated variance is zero as uncovered, so that gap shows in `rate_fraction_covered`. The radionuclide-

production covariance (MF=40, the covariance of each product state's production cross section) is in the branching subsection's `branching_covariance.arrow` and drawn by the `isomer_branching` source. ENDF/B-VIII.1, FENDL-3.2d and TENDL state no correlation in MF=40 between a product's states, or with the reaction's MF=33, so the split's uncertainty is only partly stated. The angular-distribution covariance (MF=34, `angular_covariance.arrow`), the fission multiplicity covariance (MF=31, `nubar_covariance.arrow`) and the fission spectrum covariance (MF=35, `spectrum_covariance.arrow`) are converted and published too, and `Model.data_uncertainty_coverage()` reports which nuclides carry them, but nothing samples them yet.

Beyond covariance, the decay data carries each nuclide's half-life with the standard deviation its evaluation states, and its decay energy with a standard deviation built from the evaluation's sigmas on the beta, gamma and alpha components. The evaluation states no correlation between them, so they are treated as independent and the total's sigma is their quadrature sum. The `half_life` and `decay_energy` uncertainty sources draw from these. The converter also stores the sigma each evaluation states on a decay mode's branching ratio (`branching_ratio_uncertainty` in `decay_modes.arrow`), which the `decay_branching` source draws from for parents with any number of modes. Decay data converted before that column existed carries no branching sigma. The converter also stores each evaluation's fission yields with their stated sigmas (`yield_uncertainties` in `fission_yields/evaluated_yields.arrow`), which the `fission_yield` source draws, and the sigmas stated on decay photon lines, which the `decay_photon_lines` source draws. What it does not cover lists what a nuclear-data uncertainty run holds at nominal.

Energy range

Every library reaches past 20 MeV for at least some evaluations, and only TENDL-2025 takes every evaluation to the same ceiling. Counting evaluations by the ceiling they were converted to:

library	ceiling
`endf-b8.1`	452 at 20 MeV, 34 at 150, 34 at 200, 28 at 30, rest at 28 to 65
`jeff-4.0`	491 at 200 MeV, 65 at 20, 16 at 150, 10 at 30, rest at 60 to 90
`jendl-5.0`	579 at 200 MeV, 216 at 20
`fendl-3.2d`	109 at 200 MeV, 72 at 150, 11 at 60 to 65, none at 20
`tendl-2025`	200 MeV throughout
`tendl-2017`	2,780 at 200 MeV, 14 at 20, 4 at 150, 2 at 30, 1 at 60

ENDF/B-VIII.1 is the only one that is mostly a 20 MeV library, and even its Fe56 and W186 reach 150 MeV. For a high-energy source, check the ceiling of the nuclides that carry the reaction rather than the library's headline figure.

Transport and transmutation treat the ceiling differently. In transport a particle above a nuclide's ceiling finds no cross section up there, and that reads as a rate too low rather than as an error. In a transmutation collapse, from yamc 0.11.0, it is refused: a group spectrum whose flux reaches past an evaluation's last energy point is rejected before the fold, naming the nuclide and the energy its evaluation stops at, because there is nothing up there to fold against and any answer would be invented. Earlier releases carried the last tabulated value forward and folded against that, which gives a wrong answer whenever the top groups are populated. Groups above the ceiling that carry no flux are accepted, so a padded spectrum is safe.

What a library can quietly lack

A library can lack two things that neither the tables above nor its cross sections reveal. Both matter for decay heat and shutdown dose rather than for transport.

The first is placeholder mean decay energies. A nuclide whose decay scheme nobody has evaluated still gets a decay file, with a third of each beta or electron-capture branch's Q booked to the light particles and a third to the photons. For an electron-capture emitter that is meaningless, since the neutrino carries almost all of Q: ENDF/B-VIII.1 books Sn111 at 1.63 MeV per decay against the 0.69 MeV JENDL-5 evaluates from its decay scheme. The counts are 1144 records for `endf-b8.1`, 1529 for `jeff-4.0` and 375 for `jendl-5.0`. They are exotic nuclides, but early decay heat is carried by exactly the short-lived products that tend to be unevaluated. Every one is named in the decay subsection's `provenance.json`, in data converted by yamc 0.10.0 or newer; libraries published before that carry the numbers without the record that says which of them are placeholders.

The second is reactions that do not list the isomer they make. A reaction that can leave its product in a metastable state has to say so, one entry per final state. One that lists the ground state alone puts the isomer outside the network altogether, so no code can make it. `Os190 (n,n')` to Os190m at 1706 keV and `Ir191 (n,2n)` to Ir190m2 at 377 keV are both carried by `jeff-4.0` and `tendl-2025` and both absent from `tendl-2017`, which is why an osmium foil's heat five minutes after a 14 MeV irradiation comes out at a third of the measured value on that library. Nothing about such a channel looks wrong from outside: it is present, its cross section is reasonable, and only the state list is short.

Lithium in TENDL-2025

TENDL-2025's Li6 and Li7 produce no tritium below 30 MeV. For a breeder blanket, a lithium-bearing shield, or anything else where lithium's reactions matter, use another library.

TENDL-2019 and TENDL-2023 carried the ENDF/B-VIII.0 lithium evaluations unchanged: Li6 from LANL (G.M. Hale, 2017), with its (n,t) reaction as MT=105, and Li7 from LANL (P.G. Young, 1988). TENDL-2025 replaces both with TALYS output, written by TEFAL. Its Li6 has MT=1 to 5, 51 to 55, 91 and 102 in MF=3

and no other cross section, and its Li7 has MT=1 to 5, 16, 51 to 54, 91 and 102. Neither has an (n,t) channel, so the only tritons left are those of MT=5, which starts at 30 MeV in both. Li6's resolved resonance range is a six-resonance MLBW fit with neutron and gamma widths only, so its thermal total is 0.485 b, where ENDF/B-VIII.1 has 938.9 b, 938.1 b of it (n,t).

A 100 cm sphere of natural lithium (0.534 g/cm3, 294 K) with a 14.06 MeV point source at its centre, 100,000 histories, yamc 0.16.0 on release `2026-10-12.1`, per source neutron:

library	tritium (TBR)	from Li6	from Li7	absorption
`endf-b8.1`	1.2169 +/- 0.0021	0.4624	0.7546	0.4788
`jeff-4.0`	1.2164 +/- 0.0021	0.4617	0.7547	0.4781
`tendl-2017`	1.2164 +/- 0.0021	0.4617	0.7547	0.4781
`fendl-3.2d`	1.2157 +/- 0.0021	0.4610	0.7548	0.4774
`jendl-5.0`	1.2120 +/- 0.0020	0.4750	0.7370	0.4925
`tendl-2025`	0	0	0	0.0000054

The tritium tally is not the only thing that changes. With no (n,t) and next to no capture (Li6's is 1.5e-10 b at thermal, against 0.0385 b in ENDF/B-VIII.1), lithium absorbs almost nothing on TENDL-2025: the sphere's flux comes out 3.7% higher and its neutron heating 43.5% lower than on ENDF/B-VIII.1, which counts the 4.78 MeV each Li6(n,t) releases. TENDL-2025 misses tritium production cross sections in Li6 and Li7. This is a known issue with the cross section library.

Transmutation is affected the same way, through either setting. A litre of natural lithium irradiated for 30 days at 1e14 n/cm2/s, in a spectrum with 30% of its flux between 13.5 and 14.5 MeV and the rest slowed down, makes 392 appm of tritium with `endf-b8.1` as both `neutron_cross_section_data` and `transmutation_reactions`, and none with `tendl-2025` as either.

The `tendl-2025` keyword serves the library as published; yamc does not patch or replace these two evaluations.

Hydrogen and helium in TENDL-2025

TENDL-2025 has no hydrogen or helium. Its neutron archive, `TENDL-n.tgz`, holds 2850 evaluations and none of them is H1, H2, H3, He3 or He4, so a material containing either element (water, polyethylene, a hydride, a helium coolant) cannot be modelled on `tendl-2025`. This is a gap in the library as published. Use another library for a model that contains them; `tendl-2017` carries all five.

Temperatures and size

All six are converted at the same six temperatures, 250, 293.6, 600, 900, 1200 and 2500 K, so a library swap never changes which temperature is available.

Size is per nuclide rather than per library, because only the nuclides in the model are fetched. Fe56 is 56 MB on `endf-b8.1` and 18 MB on `tendl-2025`; U235 is 171 MB and 43 MB. Transport pulls the `distributions` section on top of what an activation run needs, which is 15 MB of Fe56's 56 MB on ENDF/B-VIII.1.

Nuclear data

Config the nuclear data

This minimal example uses a single nuclear data library for every nuclide's neutron data and every element's photon data. Other configurations are covered under the nuclear data options below.

```
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"
```

A run that transports only neutrons needs only the first line, and one that transports only photons needs only the second.

Nuclear data settings

Six module-level settings, read and written like attributes. Each takes a library keyword or a path, and each can point somewhere different, so a run can mix libraries by particle and by subsection.

setting	supplies	decides	left unset
`neutron_cross_section_data`	neutron cross sections per nuclide, and the thermal scattering tables that go with them	reaction rates in neutron transport and transmutation	no default library; set it or provide explicit material data
`photon_cross_section_data`	photoatomic photon data per element	photon interactions in photon transport	no default library; set it or provide explicit material data
`transmutation_reactions`	which reaction on which nuclide gives which product, and its Q	which activation products exist to be made	no default: a calculation that reads it is refused
`transmutation_decay_data`	half-lives, decay modes, mean decay energies, decay photon lines	how the inventory decays, and the photon source a shutdown dose rate transports	no default: a calculation that reads it is refused
`transmutation_fission_yields`	fission product yields	what a fission makes	no default: a calculation that reads it is refused
`transmutation_branch_ratios`	isomeric branching against incident energy	which product is left in a metastable state	no overlay, so every product lands in its ground state

Only `transmutation_branch_ratios` has a default. The other five name the data a calculation reads, so nothing is downloaded from a library you did not choose: a calculation that needs a setting you have left unset is refused before it starts, with a message naming each unset setting and the line to copy. Every transmutation and shutdown-dose example on these pages sets what it reads. Which transmutation settings a calculation reads is listed under Transmutation data.

`neutron_cross_section_data` and `transmutation_reactions` are the pair most often confused. They are not alternatives: the cross sections decide how fast each reaction runs, and the reactions network, which holds no cross sections, decides which reactions and products exist. See The two that both say reactions.

Neutron and photon cross sections

`neutron_cross_section_data` and `photon_cross_section_data` are independent. Both start unset, and setting one never sets, reads or changes the other, so each particle's data comes from the library

you name for it. Each takes a library keyword, a local directory, or a dict: keyed by nuclide (`"Fe56"`) for neutrons and by element (`"Fe"`) for photons. Assigning a value replaces that particle's whole selection, and `None` or an empty dict clears it. What a calculation reads:

calculation	cross-section settings it reads
neutron transport	`neutron_cross_section_data`
transport from a `PhotonSource`	`photon_cross_section_data`
coupled neutron-photon transport (`transport_secondary_photons=True`)	both
a D1S shutdown dose run (`use_decay_photons=True`)	both, plus the decay data and reactions network
an irradiation: `Material.transmute()`, `yamc.transmute()` and `Model.simulate_transmutation()`	`neutron_cross_section_data`, plus the transmutation settings
activity, decay heat, decay photon lines and contact dose, and a schedule that only cools	neither, only `transmutation_decay_data`

In a coupled run the photons a neutron makes come from the neutron evaluation, and `photon_cross_section_data` decides how those photons then interact. The data are loaded automatically when a calculation needs them, and a calculation missing the data it needs is refused before it starts, naming the setting and the nuclides or elements without a source; no other library is substituted. Explicit per-material sources are available too. See Nuclear data libraries for which libraries publish photon data.

The two that both say reactions

`neutron_cross_section_data` and `transmutation_reactions` are halves of one calculation rather than alternatives. For each nuclide the network lists the channels it has, each channel's name is mapped to an MT number, and the cross sections are what that MT is priced with. So the network decides which channels are considered at all and the cross sections decide how fast each one runs. Swap the library behind `transmutation_reactions` and products appear or disappear; swap the one behind `neutron_cross_section_data` and the same products arrive at different rates.

They also fail differently, which is how to tell them apart in practice. A missing `neutron_cross_section_data` stops a neutron transport, an irradiation or a cross-section query before it starts, with an error naming the setting and a nuclide it has no data for; a decay-only schedule needs no cross sections and still runs. A network pointed at the wrong material is refused before the solve, naming what the material holds against what the network drives.

Nuclide properties and cross sections

Nuclides can be created and their properties inspected.

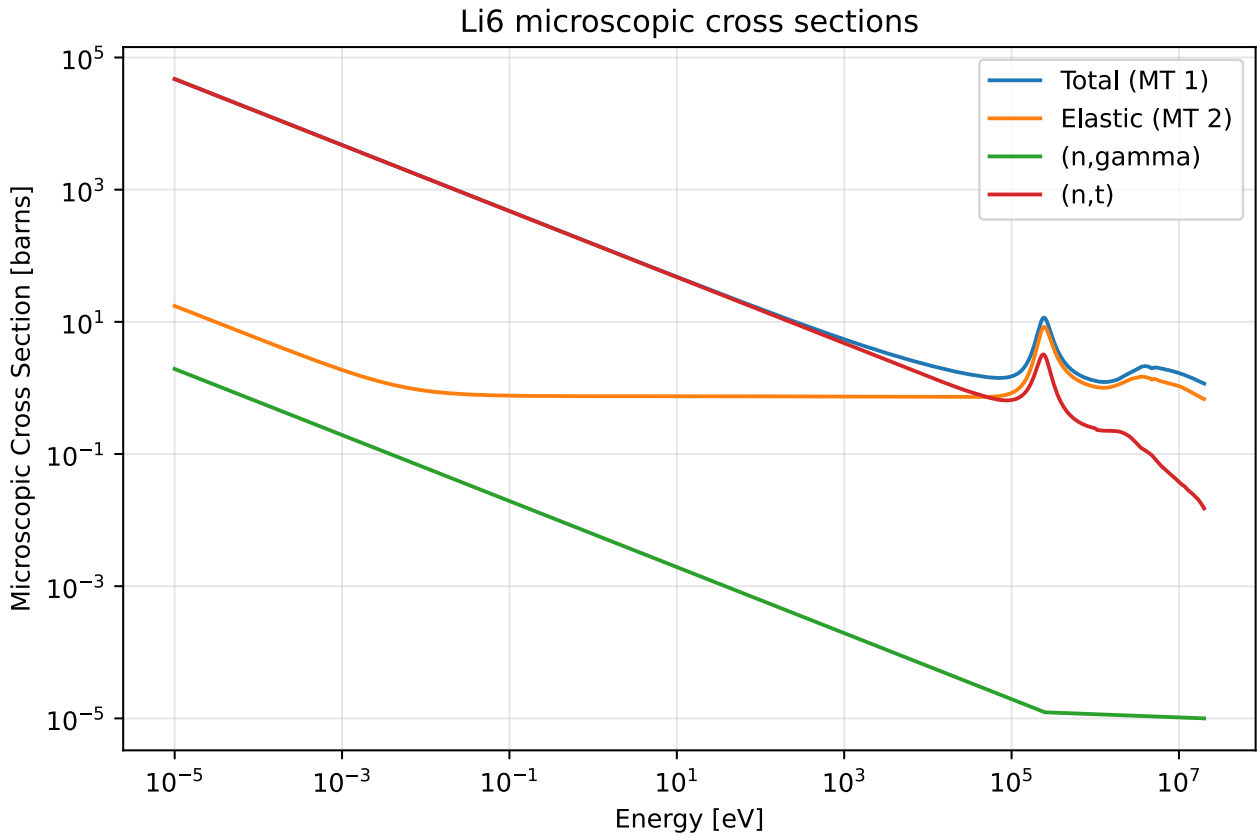
```
nuclide = yamc.Nuclide('Li6')

print(nuclide.name)                    # "Li6"
print(nuclide.element)                 # "lithium"
print(nuclide.atomic_number)           # 3
print(nuclide.mass_number)             # 6
print(nuclide.available_temperatures)  # ["250", "294", "600", "900", "1200", "2500"]
print(nuclide.reaction_mts)            # [1, 2, 3, 4, 27, 32, ..., 444, 901]
```

Microscopic cross sections are retrieved by reaction name or MT number. Each call returns `(xs_values, energy_grid)`. Different reactions may have different energy grids.

```
xs_total, energy = nuclide.microscopic_cross_section(reaction=1, temperature="294")
# total
xs_elastic, energy = nuclide.microscopic_cross_section(reaction=2, temperature="294")
# elastic
xs_ngamma, energy = nuclide.microscopic_cross_section(reaction="(n,gamma)",
temperature="294")
xs_nt, energy = nuclide.microscopic_cross_section(reaction="(n,t)",
temperature="294")
```

Plotting cross sections with matplotlib (the figure below is rendered live from this code when the docs are built):

```
import matplotlib.pyplot as plt
import yamc

yamc.neutron_cross_section_data = "endf-b8.1"

nuclide = yamc.Nuclide("Li6")
reactions = [
    (1, "Total (MT 1)"),
    (2, "Elastic (MT 2)"),
    ("(n,gamma)", "(n,gamma)"),
    ("(n,t)", "(n,t)"),
]

fig, ax = plt.subplots(figsize=(8, 5))
for reaction, label in reactions:
    xs, energy = nuclide.microscopic_cross_section(reaction=reaction,
temperature="294")
    ax.loglog(energy, xs, label=label)

ax.set_xlabel("Energy [eV]")
ax.set_ylabel("Microscopic Cross Section [barns]")
ax.set_title("Li6 microscopic cross sections")
ax.legend()
ax.grid(True, alpha=0.3)

from io import StringIO as _S
_b = _S(); plt.savefig(_b, format="svg", bbox_inches="tight"); print(_b.getvalue())
```



Elements and natural abundances

Element data is available via the `yamc.data` module.

```
# Natural abundances
abundances = yamc.data.natural_abundance()
print(abundances["Li6"])  # 0.07589
print(abundances["Li7"])  # 0.92411

# List nuclides for an element
nuclides = yamc.data.element_nuclides()
print(nuclides["Li"])  # ["Li6", "Li7"]
```

The abundances are the IUPAC TICE 2013 isotopic compositions (Meija et al., Pure Appl. Chem. 88 (2016) 293, doi:10.1515/pac-2015-0503). Each value is the column 9 representative isotopic composition. The 12 elements that TICE gives as an interval rather than a value (H, Li, B, C, N, O, Mg, Si, S, Cl, Br, Tl) use the column 6 best measurement instead, and mononuclidic elements are 1.

`natural_abundance_records()` returns the full TICE row behind each value: the column 9 value and its expanded uncertainty (or the interval), the column 4 observed range of natural variation, and the column 6 best measurement with its uncertainty, coverage factor and calibration flag. A field TICE leaves empty is `None`, which means "not stated", never zero.

```
records = yamc.data.natural_abundance_records()
fe58 = records["Fe58"]
print(fe58["representative_value"], fe58["representative_uncertainty"])  # 0.00282
0.00012
print(fe58["best_measurement"], fe58["best_measurement_uncertainty"])    # 0.002819
2.7e-05
print(fe58["best_measurement_coverage"])                                 # '2s'
print(records["Li6"]["representative_interval"])                         # (0.019,
0.078)
```

These uncertainties are reference data only. No calculation samples or propagates them: turning them into a distribution needs choices TICE does not make (a coverage factor for column 9, correlations between the isotopes of an element, how to treat an interval).

The element symbol table is bound too, so a script naming files or directories from ENDF data does not need its own copy of the periodic table.

```
yamc.data.atomic_symbol(26)         # 'Fe'
yamc.data.atomic_number("Fe")       # 26
yamc.data.split_nuclide("Fe56")     # ('Fe', 56, 0)
yamc.data.split_nuclide("Ta180_m1") # ('Ta', 180, 1)
```

`split_nuclide` returns `(symbol, mass number, isomeric state)`, the state being 0 for a ground state and 1 for the first metastable. Splitting a name by stripping its trailing digits does not do this: that

leaves `"Ta180_m"` with a mass number of 1, and natural tantalum really does contain Ta180m, so a composition built from `natural_abundance()` will hand you that name.

`atomic_symbol` takes Z = 0 and answers `'n'` for the neutron rather than raising, because ENDF libraries really do ship a neutron material (JENDL is one) and a caller walking a sublibrary will meet it. `atomic_number` is case sensitive and takes the symbol alone: `"fe"` and `"iron"` are both refused.

Element and reaction labels are available for building tables and selectors:

```
print(yamc.data.element_names()["Fe"])
print(yamc.data.reaction_names()[102])
print(yamc.cache_dir())
```

Photon attenuation and converted-data inspection

The NIST photon response tables used by contact-dose calculations are also public. They are bundled reference data, independent of the selected transport library:

```
iron_attenuation = yamc.data.mass_attenuation_coefficient(element="Fe")
air_absorption = yamc.data.mass_energy_absorption_coefficient(material="air")
print(iron_attenuation.interpolate(energy=1e6))  # cm2/g
print(air_absorption.interpolate(energy=1e6))   # cm2/g
```

After a conversion, inspect the individual nuclide or element directory without constructing a transport model:

```
nuclide_info = yamc.read_nuclide_from_arrow("converted/Fe56.arrow")
element_info = yamc.read_element_from_arrow("converted/Fe.arrow")
print(nuclide_info, element_info)
```

`yamc.data.clear_nuclide_cache()` clears the in-memory nuclide cache, not the downloaded files. A material holding data can release its own references with `material.release_nuclear_data()` before the next part of a large sweep.

Options for setting nuclear data source

The source of nuclear data can be set at the nuclide, material and program level in a few ways.

Cross sections from a library keyword

The simplest method is to name a library for each particle. The two need not be the same library: here neutrons use FENDL-3.2d and photons ENDF/B-VIII.1.

```
yamc.neutron_cross_section_data = "fendl-3.2d"
yamc.photon_cross_section_data = "endf-b8.1"
```

There are six recognized keywords: `endf-b8.1`, `jeff-4.0`, `jendl-5.0`, `tendl-2025`, `tendl-2017` and `fendl-3.2d`. Nuclear data libraries has what each one publishes, how big each one is, and which can serve as a photon source.

TENDL has no decay or fission-yield evaluations. Take `decay` / `fission_yields` from a library that has them while sourcing `reactions` and/or `branching` from TENDL. TENDL publishes no photoatomic data, so for coupled neutron/photon transport set `photon_cross_section_data` to a library that does. The next TENDL release adds photonuclear evaluated data, which does not replace photoatomic transport data. Pointing a transmutation part at a library that does not provide it still fails immediately.

TENDL-2025's Li6 and Li7 make no tritium below 30 MeV, whether used as cross sections or as a `reactions` network, so use another library wherever lithium matters: see Lithium in TENDL-2025.

A keyword resolves, once per process, to the library's current published release: a dated folder such as `2026-10-10.1` with a manifest that lists every file in it with its size and sha256. Whenever nuclear data is needed, yamc looks for the file in that release's folder of its cache. If the file is there it is used; otherwise it is downloaded first, checked against the manifest (a byte range against its own hash), and only then kept. A file that does not match is refused with an error naming it. A load of many nuclides downloads them in concurrent batches and prints a `Downloaded k of N nuclides` counter.

The cache is a `yamc` folder in the platform's per-user cache directory, and `yamc.cache_dir()` returns it:

platform	cache directory
Linux	`~/.cache/yamc`, or `$XDG_CACHE_HOME/yamc` when that is set
macOS	`~/Library/Caches/yamc`
Windows	`%LOCALAPPDATA%\yamc`

Each release has its own folder, laid out as the published tree:

```
<cache dir>/<keyword>/<release>/manifest.json
<cache dir>/<keyword>/<release>/neutron/<Nuclide>.arrow/...
<cache dir>/<keyword>/<release>/photon/<Element>.arrow/...
<cache dir>/<keyword>/<release>/transmutation/<subsection>.arrow/...
```

No setting or environment variable moves the cache (`YAMC_CACHE_DIR` is gone). To read data from somewhere else, point the setting at a local directory, as below, which bypasses the cache. When a newer release is published, the next run downloads what it needs from it, says so once, and leaves the older folder in place; every file one run reads from a library comes from a single release. Without a connection, yamc uses the newest cached release that already holds every file the run needs and says which; only a file missing from the cache is then an error. Results record the release of each library they used: `TransportResults.runs[i]["data_releases"]` and `TransmutationResults.data_releases`,

each mapping a keyword to its `release`, `manifest_sha256`, `format_version`, `offline` and `converters` (each build that wrote the release's files: the package, its version and the core commit it was compiled from). `yamc.combine_results` refuses runs that read different releases of one library.

Cross sections from a directory path

If all your Arrow data directories live in one directory you can point a setting at that directory. When a nuclide is needed (e.g. `Li6`), YAMC resolves it to `<dir>/Li6.arrow` in the neutron directory, and an element (e.g. `Fe`) to `<dir>/Fe.arrow` in the photon directory.

```
yamc.neutron_cross_section_data = "/path/to/neutron/"
yamc.photon_cross_section_data = "/path/to/photon/"
```

A whole library in one folder

A folder laid out as a published release, with its `manifest.json` beside the `neutron/`, `photon/` and `transmutation/` folders it describes, is a release folder, and its one path serves every setting. Neutron cross sections are read from `neutron/`, photon data from `photon/`, and each transmutation part from `transmutation/<part>.arrow/`:

```
yamc.neutron_cross_section_data = "/data/endf-b8.1"
yamc.photon_cross_section_data = "/data/endf-b8.1"
yamc.transmutation_decay_data = "/data/endf-b8.1"
yamc.transmutation_reactions = "/data/endf-b8.1"
yamc.transmutation_fission_yields = "/data/endf-b8.1"
```

A copy of every file a published release's `manifest.json` lists, from `https://yamc-data.xsplot.com/<keyword>/<release>/`, with the manifest itself at the top, is such a folder, and so is a release folder in the cache (`<cache dir>/<keyword>/<release>/`) for the files it holds.

Nothing is downloaded. The manifest is read as a published release's is: a `format_version` this build does not read is refused with the advice to upgrade, and each file a load reads is checked against the size the manifest gives it. A part the manifest does not list is refused, naming the parts the folder does carry, and so is a file the manifest lists that the folder does not hold. Results record the folder in `data_releases` under its absolute path, with its `release` and `manifest_sha256`, and `offline` set to `False`.

A load checks sizes only. `yamc.verify_library` is the full check: it re-hashes every file the manifest lists and looks for files it does not list, which is worth running on a library kept for offline use or copied between machines:

```
import pprint

report = yamc.verify_library("/data/irdff-ii")
pprint.pprint(report, sort_dicts=False)
```

```
{'keyword': 'irdff-ii',
 'release': '2026-10-12.1',
 'format_version': 2,
 'manifest_sha256':
'547003e2d63c167978be82915986206c581af7373c6f71ca0bb78732a1e182c5',
 'file_count': 3,
 'total_bytes': 6508798,
 'missing': [],
 'changed': [],
 'unlisted': [],
 'ok': True}
```

`ok` is `True` when `missing`, `changed` and `unlisted` are all empty. An unlisted file fails the check, because results identify the library by the manifest's sha256 and a file the manifest does not list is data that identity does not cover. A release folder in the cache holds only the files runs have downloaded, so the rest are reported missing.

A bare name that is both a keyword and a folder in the working directory (`endf-b8.1` with a folder `./endf-b8.1` beside it) is refused rather than guessed at. Write `./endf-b8.1` or an absolute path to mean the folder.

Building a library yourself

`yamc.generate_library` builds one of the published libraries from its evaluated source archives into a folder laid out exactly as the published release, with its `manifest.json` written last, so the folder serves every setting as above:

```
report = yamc.generate_library("endf-b8.1", "/data/endf-b8.1")
print(report["release"], report["file_count"], report["njoy"])

yamc.neutron_cross_section_data = "/data/endf-b8.1"
yamc.photon_cross_section_data = "/data/endf-b8.1"
yamc.transmutation_decay_data = "/data/endf-b8.1"
yamc.transmutation_reactions = "/data/endf-b8.1"
yamc.transmutation_fission_yields = "/data/endf-b8.1"
yamc.transmutation_branch_ratios = "/data/endf-b8.1"
```

It downloads each source archive and checks it against the size and sha256 it is pinned to, unpacks it, converts every evaluation (neutron data through NJOY with every covariance section the evaluation has, photon data with the Compton profile, density effect and bremsstrahlung tabulations, photonuclear data as evaluated), builds the transmutation chain, reads every folder back through the reader a simulation uses, and writes each particle's `index.json` and then the manifest.

The libraries are `"endf-b8.1"`, `"jeff-4.0"`, `"jendl-5.0"`, `"tendl-2025"`, `"tendl-2017"`, `"fendl-3.2d"` and `"irdff-ii"`. Each is built from its own evaluations only, never with another library's data:

keyword	particles	transmutation parts
`endf-b8.1`, `jendl-5.0`	neutron, photon	decay, reactions, fission_yields, branching
`jeff-4.0`	neutron	decay, reactions, fission_yields, branching
`tendl-2025`, `tendl-2017`	neutron, photonuclear	reactions, branching
`fendl-3.2d`	neutron, photon	reactions, branching
`irdff-ii`	none	branching

A library without decay data of its own (TENDL, FENDL, IRDFF-II) writes each excited product level of its branching as the evaluation gives it, and a run names the isomer against the decay data it is given: TENDL branching with `transmutation_decay_data = "endf-b8.1"` is matched to ENDF/B-VIII.1's isomers.

The keyword arguments narrow a build:

```
# The transmutation chain alone: no NJOY, minutes.
yamc.generate_library("endf-b8.1", "/data/endf-b8.1-chain",
                      parts=["decay", "reactions", "fission_yields", "branching"])

# A few nuclides and an element (every Fe evaluation, and Fe photon data).
yamc.generate_library("endf-b8.1", "/data/endf-b8.1-few",
                      nuclides=["Li6", "Li7", "Fe"], temperatures=[293.6, 600.0])
```

- `parts`: of `"neutron"`, `"photon"`, `"photonuclear"`, `"decay"`, `"reactions"`, `"fission_yields"` and `"branching"`; the default is every part the library's release carries.
- `nuclides`: nuclides and element symbols. The chain is always built whole.
- `temperatures`: in Kelvin, by default 250, 293.6, 600, 900, 1200 and 2500 K. The 0 K grid is always kept.
- `njoy_exec`: the NJOY to run. The default is the `njoy` the njoy2016 wheel (`pip install njoy2016 --index-url https://shimwell.github.io/wheels`) installs beside this Python, then the one on PATH. FENDL needs the IAEA-NDS fork, and the builder refuses any other for it.
- `downloads`: a folder of source archives fetched by hand, searched first.

Running it again resumes: a folder finished for this release is kept and only the rest are converted. The release stamp is the date the build started, kept on every resume. No manifest is written unless every evaluation converted (or is one NJOY is known not to process, declared with the reason), every folder reads back and the chain loads; a failed evaluation's reason is kept in `<path>/.build/failed/`, and running again retries only those. The returned dict has `keyword`, `release`, `path`, `parts`, `converted`

and `reused` (`"<particle>/<Name>"` keys), `not_converted` (the declared failures with their reasons), `stale_declarations`, `njoy`, `chain_nuclides`, `manifest_sha256`, `file_count` and `total_bytes`.

The release records the build that wrote it: `manifest.json` carries the package and version (`provenance.builder`), the core commit (`provenance.builder_commit`, as `yamc.__commit__`) and whether the tree had uncommitted changes (`provenance.builder_dirty`, as `yamc.__commit_dirty__`), and lists each distinct build as `converters`. A resume must be the same build, commit and dirty flag included, as the one that started the folder. An extension compiled from uncommitted changes builds only a partial library (one given `parts`, `nuclides` or `temperatures`), since no commit describes what it would write; commit the changes and rebuild to build a whole one.

A full build is long, most of it NJOY: on a 32-core machine about 50 minutes for ENDF/B-VIII.1, 9 hours for JENDL-5 and a day or more for each TENDL.

Converting single evaluations

The converters behind the builder take one evaluation at a time, for data of your own or a patched evaluation. Each writes the same folders a library holds:

```
# Transport data, through NJOY, with the evaluation's covariance sections.
yamc.convert_neutron_transport(
    input_path="n-026_Fe_056.endf", output_dir="out/neutron",
    njoy_exec="njoy", temperatures=[293.6, 600.0], covariance=True)

# Cross sections only, for a rate collapse; from ENDF, or from ACE without NJOY.
yamc.convert_neutron_xs(
    input_path="n-026_Fe_056.endf", output_dir="out/neutron-xs")

# Photon data, with the tabulations yamc carries.
yamc.convert_photon("photoat-026_Fe_000.endf", "out/photon",
                    relaxation_path="atom-026_Fe_000.endf")

# A photonuclear evaluation (TENDL's gamma sublibrary), kept as evaluated.
yamc.convert_photonuclear(input_path="g-Fe056.tendl", output_dir="out/photonuclear")

# The transmutation chain, from decay, fission yield and neutron evaluations,
# and its isomeric branching.
yamc.convert_transmutation(
    decay_files=["dec-027_Co_060.endf"], fpy_files=["nfy-092_U_235.endf"],
    neutron_files=["n-027_Co_059.endf"], output_path="out/chain")
yamc.convert_branching(
    neutron_files=["n-027_Co_059.endf"], decay_files=["dec-027_Co_060.endf"],
    output_path="out/chain")
```

A folder of `out/neutron/<Name>.arrow/` directories is a path `neutron_cross_section_data` takes, a folder of `out/photon/<Element>.arrow/` directories one `photon_cross_section_data` takes, and `out/chain` a path each transmutation setting takes. `convert_transmutation` and `convert_branching` read every material in each file, so a tape of many materials needs no splitting, and they name every product as the evaluation gives it: a product the decay data has no record for is counted at run time as produced with unknown fate, never booked onto a neighbouring nuclide.

`convert_branching` given no decay files writes each excited level as evaluated, to be matched at run time.

Cross sections from file paths

A nuclide's data is a folder of Arrow files, `<Nuclide>.arrow`, which a published release serves at `https://yamc-data.xsplot.com/<keyword>/<release>/neutron/<Nuclide>.arrow/` (every file its `manifest.json` lists for that folder). Once such folders are saved locally, their paths can be used to configure the nuclear data. Replace the `tests` dir with the path to the downloaded data files.

```
yamc.neutron_cross_section_data = {
    "Be9": "tests/Be9.arrow",
    "Fe54": "tests/Fe54.arrow",
    "Fe56": "tests/Fe56.arrow",
    "Fe57": "tests/Fe57.arrow",
    "Fe58": "tests/Fe58.arrow",
    "Li6": "tests/Li6.arrow",
    "Li7": "tests/Li7.arrow",
}
```

Photon data is keyed by element, and a published release serves each element at `https://yamc-data.xsplot.com/<keyword>/<release>/photon/<Element>.arrow/`:

```
yamc.photon_cross_section_data = {
    "Be": "tests/Be.arrow",
    "Fe": "tests/Fe.arrow",
    "Li": "tests/Li.arrow",
}
```

Cross sections from paths and specific libraries

Different sources can also be mixed. In this example we use the `endf-b8.1` and `fendl-3.2d` library keywords for some nuclides and local file paths for others. Replace the `tests` dir with the path to the downloaded data files.

```
yamc.neutron_cross_section_data = {
    "Be9": "fendl-3.2d",
    "Fe54": "endf-b8.1",
    "Fe56": "tests/Fe56.arrow",
    "Fe57": "tests/Fe57.arrow",
    "Fe58": "tests/Fe58.arrow",
    "Li6": "endf-b8.1",
    "Li7": "fendl-3.2d",
}
```

Specific nuclear data on the Nuclide

You can also avoid setting `yamc.neutron_cross_section_data` entirely and specify the nuclear data to use on the Nuclide object directly.

This can be done using a nuclear data library.

```
nuclide = yamc.Nuclide('Li6')
nuclide.read_nuclear_data('endf-b8.1')
```

Alternatively a specific nuclear data path

```
nuclide = yamc.Nuclide('Li6')
nuclide.read_nuclear_data('tests/Li6.arrow')
```

Or a directory path (resolves to `tests/Li6.arrow`)

```
nuclide = yamc.Nuclide('Li6')
nuclide.read_nuclear_data('tests')
```

Sample interacting reaction

Once the interacting nuclide is known then the reaction can be sampled using `Nuclide.sample_reaction()`.

```
import yamc
nuc = yamc.Nuclide('Li6')
nuc.read_nuclear_data('tests/Li6.arrow')
reaction = nuc.sample_reaction(energy=1.0, temperature='294', seed=42)
if reaction is not None:    # None where no reaction can be sampled
    print(f'interacting reaction is MT {reaction["mt_number"]}')
```

Transmutation data

For transmutation (`Material.transmute()`), coupled transport-transmutation (`Model.simulate_transmutation()`) and shutdown dose rate (D1S), YAMC also needs a transmutation network: the chain of reactions, decays and decay-photon sources used to evolve nuclide inventories.

The network is assembled from four independently sourced parts, each set globally to a library keyword (its subsection's files are downloaded from the library's current release on first use, verified and cached) or a path to a local chain directory:

```
# decay / reactions / fission_yields have no default: set each part the
# calculation reads (keyword or path). Parts may come from different libraries.
yamc.transmutation_decay_data     = "endf-b8.1"
yamc.transmutation_reactions      = "endf-b8.1"
yamc.transmutation_fission_yields = "endf-b8.1"

# Isomeric branching is a separate overlay part; sourced (when available) from a
# TENDL release. Unset means no branching overlay is applied.
yamc.transmutation_branch_ratios  = "tendl-2025"

# The reactions network topology + Q values can also come from a TENDL release,
# which covers far more parent nuclides than endf-b8.1:
yamc.transmutation_reactions      = "tendl-2017"

# A value can also be a path to a local chain directory that holds decay/,
# reactions/ and fission_yields/ subsection subdirectories, or to a whole
# library folder (see "A whole library in one folder" above):
yamc.transmutation_decay_data = "/path/to/transmutation-endf-b8.1-sfr.arrow"
```

A reactions or fission-yields folder must hold its `reactions.arrow` or `fission_yields.arrow`: one without it is refused, naming the folder, rather than read as a network with no reactions or no yields. Set the part to `False` (below) to leave it out.

See Nuclear data libraries for which library provides which subsection. Setting a part to a library that does not publish it fails fast with a message listing that library's available subsections.

A part can also be turned off, which is the correct setting for a decay-only calculation or for a material in which nothing fissions:

```
yamc.set_transmutation_reactions(False)       # decay only, no reaction topology
yamc.set_transmutation_fission_yields(False)  # nothing here fissions
```

`False` is not the same as leaving a part unset, which is refused by any calculation that reads it. Read back, each setting is what was assigned: the source, `False` when off, or `None` when unset. A rate that then needs the part that was switched off is refused when the burnup matrix is built, naming the nuclide and the reaction, rather than being solved as though the reaction produced nothing, or losing its fission products while still burning the nuclide.

The transmutation APIs and the decay-photon post-processing (`PulseSchedule.time_correct_tally`) read these settings; there is no per-call override. What each calculation reads:

calculation	reads
activity, decay heat, decay photon lines and contact dose of a result, and a schedule that only cools	`transmutation_decay_data`
a D1S run (`use_decay_photons=True`), `Model.radionuclides()` and `PulseSchedule.time_correct_tally`	`transmutation_decay_data` and `transmutation_reactions`
a schedule with an irradiating step: `Material.transmute()`, `yamc.transmute()` and `Model.simulate_transmutation()`	all three

Materials

Creating a material

A material is defined by a composition dictionary and a density. The simplest form uses nuclide names as keys with atom fractions as values.

```
iron = yamc.Material(
    composition={
        "Fe54": 0.0585,
        "Fe56": 0.9175,
        "Fe57": 0.0212,
        "Fe58": 0.0028,
    },
    density=7.874,
)
```

Composition keys

Composition keys can be nuclide names, element symbols, or chemical formulas. Elements are automatically expanded to their natural isotopic distribution and formulas are parsed into their constituent nuclides.

```
# Nuclide names (explicit isotopes)
mat = yamc.Material(
    composition={"Li6": 0.5, "Li7": 0.5},
    density=0.534,
)

# Element symbols (expanded to natural isotopes)
mat = yamc.Material(
    composition={"Fe": 1.0},
    density=7.874,
)

# Chemical formulas
mat = yamc.Material(
    composition={"H2O": 1.0},
    density=1.0,
)

# Mixed keys
mat = yamc.Material(
    composition={"Li4SiO4": 0.6, "Be": 0.4},
    density=2.0,
)
```

Density specification

Density is always a float; its meaning is set by the optional `density_units` keyword. With `density_units="atom/barn-cm"` the value is the total atom density and the composition values are relative atom fractions (requires `fraction_type="atom"`).

The keyword matches the `material.density_units` property. The former `units` keyword is not accepted.

```
# Mass density (default units g/cm3)
mat = yamc.Material(composition={"Fe": 1.0}, density=7.874)
mat = yamc.Material(composition={"Fe": 1.0}, density=7874.0, density_units="kg/m3")

# Total atom density + relative atom fractions
mat = yamc.Material(composition={"Li6": 0.625, "Li7": 0.375}, density=0.08,
                    density_units="atom/barn-cm")

# Absolute per-nuclide atom densities, stored verbatim (e.g. from a solver)
mat = yamc.Material.from_atom_densities({"Li6": 0.05, "Li7": 0.03})
```

`material.density` is always a float: the mass density for `g/cm3` / `kg/m3`, or the total atom density for `atom/barn-cm` (and for transmuted materials).

Atom vs weight percent

The `fraction_type` parameter controls how composition fractions are interpreted. The default is `"atom"` (atom fractions).

```
# Atom fractions (default)
mat = yamc.Material(
    composition={"U235": 0.05, "U238": 0.95},
    density=19.1,
    fraction_type="atom",
)

# Mass fractions
mat = yamc.Material(
    composition={"U235": 0.05, "U238": 0.95},
    density=19.1,
    fraction_type="mass",
)
```

Enrichment

The `enriched()` helper lets you specify isotopic enrichment for elements and formulas.

```
# Lithium enriched to 60% Li6
mat = yamc.Material(
    composition={
        "Li": yamc.enriched(1.0, target="Li6", percent=60.0),
    },
    density=0.534,
)

# Mixed: enriched element + natural element
mat = yamc.Material(
    composition={
        "Li": yamc.enriched(0.4, target="Li6", percent=90.0),
        "Fe": 0.6,
    },
    density=5.0,
)

# Mass percent enrichment
mat = yamc.Material(
    composition={
        "Li": yamc.enriched(1.0, target="Li6", percent=90.0, fraction_type="mass"),
    },
    density=0.534,
)
```

Material properties

The constructor accepts several optional keyword arguments for naming, identification, temperature selection, and transmutation.

```
mat = yamc.Material(
    composition={"Fe": 1.0},
    density=7.874,
    name="iron",
    temperature=294,    # Kelvin - selects temperature in nuclear data
    transmutable=True,   # required for Model.simulate_transmutation()
    volume=523.6,       # cm^3 - required for transmutation normalization
)
```

Properties are readable after construction.

```
print(mat.name)                 # "iron"
print(mat.nuclides)             # [("Fe54", 0.0585), ("Fe56", 0.9175), ...]
print(mat.density)              # 7.874
print(mat.density_units)        # "g/cm3"
print(mat.temperature)          # "294"
print(mat.average_molar_mass()) # g/mol
```

Thermal scattering

Bound nuclides (H in water, Be and Li in FLiBe, C in graphite) scatter thermal neutrons by the material's own thermal scattering tables, named with `s_alpha_beta`:

```
water = yamc.Material(composition={"H2O": 1.0}, density=1.0,
s_alpha_beta="c_H_in_H2O")
flibe = yamc.Material(
    composition={"Li7": 2, "Be9": 1, "F19": 4},
    density=1.94,
    temperature=900,
    s_alpha_beta=["c_Li_in_FLiBe", "c_Be_in_FLiBe", "c_F_in_FLiBe"],
)
# a dict gives the fraction of a table's nuclides that is bound
zrh = yamc.Material(
    composition={"Zr": 1, "H": 1.6}, density=5.6, temperature=600, s_alpha_beta=
{"c_H_in_ZrH": 0.6}
)
```

Which nuclides a table applies to is the table's own data (`material.thermal_bindings` lists them once the data is read). The tables come from the same library as the cross sections, `yamc.neutron_cross_section_data` included, at the material's temperature. A library without the table, or a temperature outside the table's range, raises an error naming the table rather than running without it.

Transmutable materials

Only materials marked `transmutable=True` are evolved by `Model.simulate_transmutation()`. Inert materials (the default) keep their composition fixed across timesteps, which avoids needlessly tracking nuclides in shielding or structural cells. A `volume` (cm³) is also required so reaction rates can be normalised to atom densities.

```
fuel = yamc.Material(
    composition={"U235": 0.05, "U238": 0.95},
    density=10.4,
    name="fuel",
    transmutable=True,    # evolved during transmutation
    volume=523.6,         # cm^3 - required when transmutable=True
)

shielding = yamc.Material(
    composition={"Fe": 1.0},
    density=7.874,
    name="shielding",
    # transmutable=False is the default; composition stays fixed
)
```

The flag is not needed for `Material.transmute()` (standalone transmutation with a pre-computed flux). That method always evolves the material it is called on.

Mixing materials

Existing materials can be combined with `Material.mix_materials()`. The `fraction_type` parameter controls whether fractions are by atom (`"atom"`), mass (`"mass"`), or volume (`"volume"`).

```
steel = yamc.Material(composition={"Fe": 1.0}, density=7.874)
water = yamc.Material(composition={"H2O": 1.0}, density=1.0)

# 80% steel, 20% water by volume
mixed = yamc.Material.mix_materials(
    materials=[steel, water],
    fractions=[0.8, 0.2],
    fraction_type="volume",
    name="steel_water_mix",
)
```

Material collections

A material can also be taken directly from a curated collection instead of typing in its composition. `yamc.materials.pnnl` holds 410 materials from the PNNL Compendium of Material Composition Data for Radiation Transport Modeling (PNNL-15870, Rev. 2): concretes, steels, tissues, air, soils, shielding, scintillators, fuels and more. The data is bundled in the wheel, so a lookup needs no network and cannot fail.

Materials are keyed by the report's own name, verbatim, so an entry can be quoted straight back to it.

```
steel = yamc.materials.pnnl["Steel, Stainless 304"]
```

The result is an ordinary `Material`, built fresh on every lookup, so it can be renamed and modified freely. `material()` sets fields in the same call, such as a name a mesh geometry expects and the temperature:

```
steel = yamc.materials.pnnl.material("Steel, Stainless 304",
name="firstwall_material", temperature=600)
```

See the full list of PNNL materials for every name, or search the collection from Python. `search()` is a case-insensitive substring match and returns names you can pass straight back in.

```
yamc.materials.pnnl.search("concrete")   # -> ['Concrete [Los Alamos (MCNP) Mix]',
...]
yamc.materials.pnnl.names()              # -> all 410 names
len(yamc.materials.pnnl)                 # -> 410
"Steel, Stainless 304" in yamc.materials.pnnl
yamc.materials.collections()             # -> ['pnnl']
```

An unrecognised name raises `KeyError` and suggests near matches.

```
yamc.materials.pnnl["Steel, Stainless 999"]
# KeyError: no material named 'Steel, Stainless 999' in the pnnl collection;
#           did you mean: Steel, Boron Stainless, Steel, HT9 Stainless, ...
```

Overriding fields at lookup

`temperature`, `volume`, `transmutable`, `name` and `id` are all settable on the returned material, or in the `material()` call. `density` is only settable in the call, because a `Material`'s per-nuclide atom densities are fixed at construction. Use it to build at a different density, for example a concrete at a different porosity or a steel diluted by a void fraction. `density_units` takes any `Material` density unit (`"g/cm3"` by default, `"kg/m3"`, `"atom/barn-cm"`), and a density above the collection's reference density raises a warning, since no void fraction or porosity can give one.

```
shield = yamc.materials.pnnl.material(
    "Concrete, Ordinary (NIST)",
    name="bioshield",
    density=1.8,        # g/cm3, replacing the collection's reference density
    temperature=600,
    volume=1.0e6,
    transmutable=True,
)
```

```
# 10% void in an 8000 kg/m3 steel
steel = yamc.materials.pnnl.material("Steel, Stainless 316", density=0.9 * 8000,
density_units="kg/m3")
```

Provenance

Every entry carries the report's own entry number so a composition can be cited.

```
yamc.materials.pnnl.entry("Steel, Stainless 304")
# {'name': 'Steel, Stainless 304', 'number': 331, 'density': 8.03,
#  'formula': None, 'composition': {'C': 0.003635, 'Mn': 0.01987, ...},
#  'citation': 'PNNL-15870, Rev. 2',
#  'url': 'https://www.pnnl.gov/main/publications/external/technical_reports/PNNL-
15870Rev2.pdf'}
```

How compositions are stored

Values are the report's own atom fractions, taken verbatim from its Elemental column, so no weight-to-atom conversion is applied on the way in.

Where an element is isotopically natural the element symbol is stored, and yamc expands it with its own natural abundances instead of the report's. Where a material is not natural (enriched lithium and boron, uranium, plutonium, heavy water, He-3) the individual nuclides are stored instead.

```
yamc.materials.pnnl.entry("Steel, Stainless 304")["composition"]  # {'Fe': 0.667971,
'Cr': 0.199443, ...}
yamc.materials.pnnl.entry("Water, Heavy")["composition"]          # {'H2': 0.666667,
'O': 0.333333}
```

One entry is omitted

The report has 411 entries; yamc bundles 410. Entry 173 "Iron Boride (Fe2B)" is omitted because it contradicts itself: its stated formula and molecular weight are those of Fe_2B, while its atom fractions, weight fractions and atom densities all describe FeB_2. Entry 174 "Iron Boride (FeB)" is unaffected and is included.

The compositions are reproduced from a US Department of Energy technical report (R.J. McConn Jr et al., Pacific Northwest National Laboratory, 2021). yamc bundles them for lookup convenience and claims no authorship of them.

Loading nuclear data

Set `yamc.neutron_cross_section_data` to select neutron data and `yamc.photon_cross_section_data` to select photoatomic photon data. The two are independent and both start unset. A calculation loads the required nuclides and elements automatically; material construction needs no explicit data read.

```
# Automatic: neutron and photoatomic photon data load as needed
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"

# Explicit: per-material file paths
mat.read_nuclear_data({
    "Fe54": "path/to/Fe54.arrow",
    "Fe56": "path/to/Fe56.arrow",
})

# Optional explicit photon override for local data or mixed libraries
mat.read_nuclear_data(
    {"Fe56": "path/to/Fe56.arrow"},
    photon_data={"Fe": "path/to/Fe.arrow"},
)
```

Arrow (`.arrow`) format is used for both neutron and photoatomic photon data. Explicit photon maps use element symbols, not nuclide names, and override the global selection. They are not required when the selected library provides the desired data. See Photon data is selected separately for available libraries and the distinction from TENDL photonuclear evaluations.

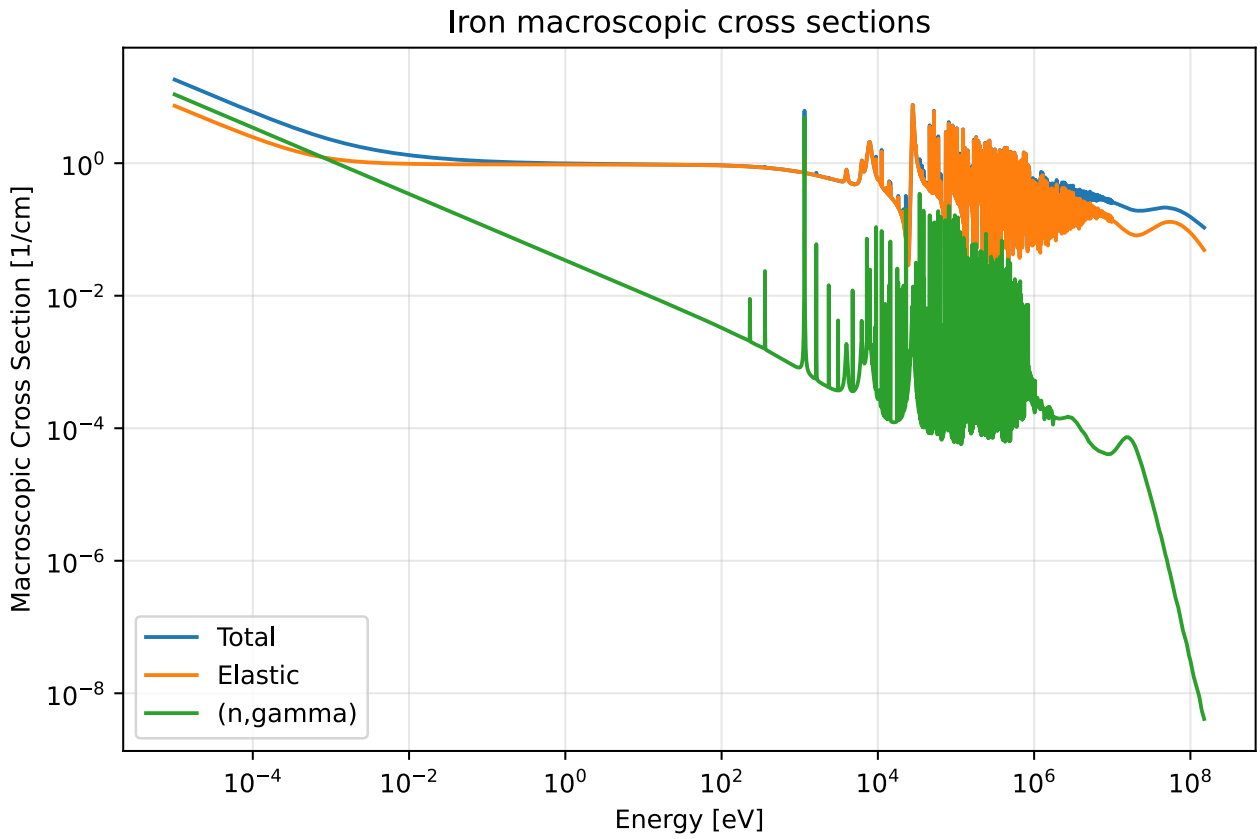
Macroscopic cross sections

The macroscopic cross section for a reaction can be retrieved by name or MT number. Returns `(xs_values, energy_grid)` on the material's unified energy grid.

```
# By reaction name
xs, energy = mat.macroscopic_cross_section(reaction="(n,total)")

# By MT number
xs, energy = mat.macroscopic_cross_section(reaction=1)   # total
xs, energy = mat.macroscopic_cross_section(reaction=102) # (n,gamma)
```

Plotting macroscopic cross sections (rendered live when the docs are built):

```
import matplotlib.pyplot as plt
import yamc

yamc.neutron_cross_section_data = "endf-b8.1"

mat = yamc.Material(composition={"Fe": 1.0}, density=7.874, temperature=294,
name="iron")

reactions = [(1, "Total"), (2, "Elastic"), ("(n,gamma)", "(n,gamma)")]
fig, ax = plt.subplots(figsize=(8, 5))
for reaction, label in reactions:
    xs, energy = mat.macroscopic_cross_section(reaction=reaction)
    ax.loglog(energy, xs, label=label)

ax.set_xlabel("Energy [eV]")
ax.set_ylabel("Macroscopic Cross Section [1/cm]")
ax.set_title("Iron macroscopic cross sections")
ax.legend()
ax.grid(True, alpha=0.3)

from io import StringIO as _S
_b = _S(); plt.savefig(_b, format="svg", bbox_inches="tight"); print(_b.getvalue())
```



Mean free path

The neutron mean free path at a given energy (in eV) is returned in cm.

```
mfp = mat.mean_free_path_neutron(14e6)  # 14 MeV
```

Sampling

Monte Carlo transport works by sampling random distances and interactions. These methods expose that process for inspection or educational use.

```
# Sample a distance to the next collision (cm)
distance = mat.sample_distance_to_collision(energy=14e6, seed=42)

# Sample which nuclide interacts
nuclide_name = mat.sample_interacting_nuclide(energy=14e6, seed=42)
```

Transmutation

Materials can be transmuted standalone using a pre-computed multigroup flux spectrum, without running transport. This is useful when the flux is known from a prior simulation.

The example below uses a simple hand-specified 3-group spectrum so it can be read in isolation. In a real workflow the spectrum is usually obtained from a transport tally. See Neutron and photon flux

spectra for how to score an energy-binned flux on a cell and feed `results[tally].mean` (scaled by the source strength) into `transmute()`.

```
# Every irradiation needs neutron cross sections, and a run without them stops
# with an error naming this setting.
yamc.neutron_cross_section_data = "endf-b8.1"

# The transmutation network is assembled from the per-subsection sources;
# point the three network-forming parts at the chain file.
yamc.transmutation_decay_data = "transmutation-endf-b8.1-sfr.arrow"
yamc.transmutation_reactions = "transmutation-endf-b8.1-sfr.arrow"
yamc.transmutation_fission_yields = "transmutation-endf-b8.1-sfr.arrow"

# The spectrum rides on each irradiation pulse: a NeutronSource whose energy is
# a Histogram over the energy groups. The Histogram normalizes the shape, so a
# multigroup flux from a tally can be passed straight in; see "Neutron and
# photon flux spectra" for the full workflow. The pulse rate is the total flux
# magnitude [n/cm^2/s]; Cooldown is decay-only.
energy_groups = [1e-5, 0.625, 1e5, 2e7]   # 3 groups (eV, ascending); or a name like
"CCFE-709"
multigroup_flux = [1e12, 5e12, 1e14]      # thermal, epithermal, fast

spectrum = yamc.NeutronSource(energy=yamc.sources.Histogram(energy_groups,
multigroup_flux))
total_flux = sum(multigroup_flux)
schedule = yamc.PulseSchedule(
    [yamc.Pulse(rate=total_flux, duration=(1, "h"), source=spectrum) for _ in
range(5)]  # 5h irradiation
    + [yamc.Cooldown(duration=(1, "h"))]
# 1h cooling
)
results = material.transmute(schedule=schedule)

# results is a TransmutationResults. step_materials() gives one Material per
# schedule step, in order.
for i, mat in enumerate(results.step_materials(material_id=material.id)):
    print(f"Step {i+1}: {len(mat.nuclides)} nuclides")

# The end state on its own, without walking the steps.
final = results.get_final_material(material_id=material.id)
if final is not None:
    print(f"Final: {len(final.nuclides)} nuclides")
```

Parameters: - `schedule`: a `PulseSchedule` of `Pulse` / `Cooldown` steps. Each irradiation `Pulse` carries a duration (seconds, or a `(value, unit)` tuple), a `rate` (the total flux magnitude [n/cm²/s]), and a `source`, which is a `NeutronSource` whose energy is a `Histogram` giving the multigroup spectrum (its boundaries may be a named group structure such as `"CCFE-709"`). The source's position/direction are ignored (no transport runs). `Cooldown` is decay-only. Different pulses may carry different spectra. The transmutation network is assembled from the configured per-subsection sources (`yamc.transmutation_decay_data`, `yamc.transmutation_reactions`, `yamc.transmutation_fission_yields`, and optionally `yamc.transmutation_branch_ratios`).

Clearance

A transmuted material can be checked against the regulatory clearance, exemption and disposal limits directly, from yamc 0.14.0. `clearance_index()` divides each radionuclide's activity by its limit in a set and sums the ratios; the material meets the limits when the sum is below the set's threshold, normally 1. The sets cover the UK (EPR 2016, IRR 2017), Germany (StrlSchV), the US (NRC 10 CFR 61, Fetter), the EU Basic Safety Standards and IAEA GSR Part 3, and come from the radiological-material-clearance-finder crate, whose documentation describes each one and where its numbers come from.

The half-lives are the configured transmutation chain's, the same ones `activity()` and `decay_heat()` use. The composition is enough for the Bq/g and Ci/m3 sets; only the total activity sets need `material.volume`. Stable isotopes have to stay in the material, since they are most of its mass and so most of the Bq/g denominator.

```
if final is not None:
    result = final.clearance_index("UK_EPR16_out_of_scope")
    result.index        # float, sum of activity-to-limit ratios
    result.clearable    # bool, index below the set's threshold
    result.by_nuclide   # dict[str, float], each nuclide's ratio, largest first
    result.uncovered    # dict[str, float], activity no limit in the set covers
    print(result)       # a short report of the dominant nuclides

    results = final.clearance_indices()   # dict[str, ClearanceResult], every set
    routes = [name for name, r in results.items() if r.clearable]
```

`metal=True` applies the NRC's activated metal rows, `apply_default_limit` and `exclude_daughters` switch off the catch-all limit and the secular equilibrium credit that some tables define, and `uncovered_fraction` is the number to check before trusting a comfortable index.

Activity, decay heat and decay photons

A transmuted material provides the usual decay quantities directly. Each takes `per=None` for the whole material, which needs `volume`, or `"cm3"` or `"g"` for a density, and the totals also take `by_nuclide=True`.

```
if final is not None:
    final.activity(per="g")                          # Bq/g
    final.decay_heat(per="cm3")                      # W/cm3
    final.decay_heat(per="cm3", component="gamma")   # the gamma part alone
    energies, rates = final.decay_photon_spectrum(per="cm3")   # lines, photons/s/cm3
    for continuum in final.decay_photon_continua(per="cm3"):
        print(continuum.nuclide, continuum.emission_rate)      # photons/s/cm3
```

`component` splits decay heat into the recoverable-heat components the decay data gives (`"beta"`, `"gamma"` or `"alpha"`): the gamma heat is the part that leaves a thin component, while the beta and alpha heat stay where they are made. It raises if a nuclide making decay heat carries no split, rather than understating the component by that nuclide's share.

`decay_photon_spectrum()` gives the discrete lines. Part of some decay spectra is given as a density over energy instead: the spontaneous-fission photons of an actinide, or the whole emission of a nuclide

far from stability whose lines were never measured. `decay_photon_continua()` returns those, one `PhotonContinuum` per nuclide and continuum, each with its own `energies`, `rates` per eV, `interpolation` law and `emission_rate` (the integral). Their rates are per eV, so they cannot be added to the line rates; a material with no continuum returns an empty list.

Repeated calls, and letting the data go

`transmute()` loads the cross sections it needs into the material and keeps them there, so calling it again on the same material does no file reading at all. On a steel against ENDF/B-VIII.1 that is 1.8 s of a 2.3 s call. The composition is not modified.

This helps when a material is transmuted more than once. It hurts in a sweep over thousands of distinct compositions, where every material would hold its nuclides' data, a few dozen for a pure metal and up to a couple of hundred for a steel, for as long as it lives. `release_nuclear_data()` drops it. The material stays usable, and the next call that needs the data loads it again.

```
for composition in many_compositions:
    scan_mat = yamc.Material(composition=composition, density=7.9)
    scan_mat.transmute(schedule=schedule)
    scan_mat.release_nuclear_data()
```

Many materials at once

`yamc.transmute()` is the plural of `Material.transmute()`, for the cells of a mesh from a transport run, a component broken into regions, or a sweep over compositions. Each material gets exactly the answer its own `transmute()` would give, and the result is one `TransmutationResults` keyed by each material's `id`. The chain is read once, each nuclide's cross sections are decoded once and shared, and a multigroup collapse runs once for every distinct combination of spectrum, composition, temperature, loaded nuclide data and shielding, before the materials are solved in parallel.

```
cells = [
    yamc.Material(composition={"Fe": 1.0}, density=7.874, id=i + 1, volume=1.0)
    for i in range(3)
]
cell_fluxes = [1e14, 5e13, 1e13]   # each cell's flux magnitude [n/cm^2/s]
schedules = [
    yamc.PulseSchedule([
        yamc.Pulse(rate=flux, duration=(1, "y"), source=spectrum),
        yamc.Cooldown(duration=(1, "d")),
    ])
    for flux in cell_fluxes
]
many = yamc.transmute(cells, schedules)

many.get_final_material(material_id=2)
many.get_source_rates(material_id=2)   # that cell's flux magnitude per step
many.collapse_reuse                    # {"performed": ..., "requested": 3}
```

Give one schedule for every material or one per material, in the same order. The schedules may differ in their spectra and rates but must share one timeline, the same durations with irradiation on the same

steps. The ids must be distinct. `data_uncertainty`, `self_shielding_shape` and `self_shielding_chord` apply to every material as `Material.transmute()` applies them to one. `collapse_reuse` reports the collapses actually run against one per distinct spectrum per material. Above, the three cells share one spectrum and one composition, so they share one collapse whenever they also load the same products: a product's cross sections are loaded only once it crosses `product_threshold` over the material's own schedule, so cells whose fluxes differ enough can load different products and need a collapse each.

Resonance self-shielding

The spectrum handed to `transmute()` is the flux outside the material. Inside a lump the flux is depressed at the energies where the material's own total cross section is large, so a resonance absorber shields itself and its reaction rates come out lower than a dilute fold predicts.

Give `transmute()` a lump size and that depression is computed and applied:

```
results = material.transmute(
    schedule=schedule,
    self_shielding_shape=yamc.shapes.FoilLump(thickness=0.1),   # cm
)
```

The five shapes live in `yamc.shapes`. The shape decides how much surface a volume hides behind, since the mean chord is `4V/S`. `SphereLump()` and `CubeLump()` take no arguments because `Material(volume=...)` already fixes them; `FoilLump(thickness=)`, `CylinderLump(radius=)` and `WireLump(radius=)` carry the one dimension a volume cannot imply. A sphere has the least surface for its volume, so it has the longest chord and shields more than any other shape of the same size. That makes it an upper bound, not a safe default, so there is no default shape.

`self_shielding_chord` takes the mean chord `4V/S` in cm directly, for a lump none of the five shapes describes. Give one or the other, not both.

Omit both and nothing is shielded: the rates are bit-identical to a run without the argument, and no shape is inferred from a geometry this material does not have.

The flux inside the lump comes from a slowing-down solve, which assumes nothing about resonances being narrow. The cheaper narrow-resonance approximation is deliberately not offered, because it over-shields strong elastic scatterers badly enough to be worse than applying no correction at all. W186(n,gamma) in FNG-tung measures 1.29 b, where the slowing-down solve gives C/E 0.80, no correction gives 0.86 and narrow resonance gives 0.46.

`get_self_shielding_info` reports what was done, and is present for every material a `transmute()` call solved, shielded or not:

```
shielding = results.get_self_shielding_info(material_id=material.id)
if shielding is not None:
    shielding["method"]            # how the depression was computed
    shielding["chord_cm"]          # None if the run was dilute
    shielding["shielded"]          # nuclides the correction reached
    shielding["not_shielded"]      # the ones it did not, and why
    shielding["strongest_factor"]  # smallest factor any group average was scaled by
```

A `strongest_factor` of `1.0` means the run shielded nothing in practice. That is not the same as not having tried.

A dilute run fills `would_shield` instead: the nuclides whose own resonances are structured enough to have suppressed a reaction, each mapped to how strongly. It is an indicator, not a correction and not a bound. It is computed from the one reaction and that nuclide's own density, with no geometry in it, so it can sit either side of the real factor: a lump thinner than a centimetre of chord shields less, and a material whose other nuclides dominate the total at the resonance dips the flux further. On the FNS tungsten foil it reads 0.634 for W186, where the slowing-down correction on the same foil and spectrum gives 0.730. It warns that the answer may be high and that the nuclide is a resonance absorber. Read the size of the effect off a shielded run.

Nuclear-data uncertainty

Pass a `DataUncertainty` and every nuclide density comes back with a standard deviation beside it:

```
results = material.transmute(
    schedule=schedule,
    data_uncertainty=yamc.DataUncertainty(seed=42),
)

mid = material.id
# Both answer None for a nuclide, material or step the run does not have, so a
# real script checks rather than formatting None into the output.
mean = results.get_nuclide_density(material_id=mid, nuclide="Mn56", step=1)
sigma = results.get_nuclide_uncertainty(material_id=mid, nuclide="Mn56", step=1)
if mean is not None and sigma is not None:
    print(f"Mn56: {mean:.3e} +/- {sigma:.3e}  ({100 * sigma / mean:.1f}%)")
```

Each sample redraws the inputs listed under Choosing what to perturb from their stated uncertainties: the activation cross sections from their ENDF MF=33 covariance and their resonance parameters where the library carries them; the half-lives, decay energies, decay photon lines and decay branching ratios from the decay data's own standard deviations; the fission yields from theirs; the isomeric splits from MF=40; the flux from the error you give it; and, on Model.simulate_transmutation(), the transport-tallied rates from their per-history covariance. On the spectrum calls the drawn cross sections are folded against this material's own spectrum. On `Model.simulate_transmutation` they are carried through the transport instead, so each sample's rates are the ones its own data would give, flux included. The schedule is re-solved with the drawn rates, branching, flux and half-lives, and the drawn decay energies are applied where decay heat is evaluated. That is exact to all orders in the matrix exponential: nothing is linearized, and the solver itself is untouched, only its input changes.

`DataUncertainty(seed=...)` is reproducible. A nuclide's cross-section, half-life, decay-branching, decay-energy, decay-photon, fission-yield and isomeric-branching draws in a given sample are a pure function of `(seed, sample, nuclide)`, shared by every material of the run. The exception is nuclides that an applied cross-material covariance links (see What it does not cover): they are drawn jointly, so their cross-section draws differ between a run with and without the partner. The flux draw is a function of `(seed, sample, spectrum)`, the spectrum's place among the schedule's spectra. The statistical draw is keyed on `(seed, sample, material)`, the material by its place in the geometry rather than its id, and made jointly over the material's tallied rates, so adding or removing a tallied rate changes the draws of the others, while different materials, whose tallies are separate estimates, draw independently. The flux draw carries no material key: it is keyed on the spectrum's place in each material's own schedule, so on a multi-material `transmute()` the materials whose schedules put a spectrum in the same place draw identical flux deviates. Their flux errors are then fully correlated, which is right for one shared tally and wrong for separately tallied cells. In every case the same seed gives the same answer whatever the sample count or iteration order.

`DataUncertainty(convergence=...)` sets how many samples run. It is the target for each sigma's standard error relative to the sigma, 0.05 by default, judged at every step on each nuclide density within 1e-6 of that step's largest and on the total activity, decay heat and decay photon line rate. The standard error carries the sample kurtosis, so a skewed output needs more samples than a Gaussian one. Leave `samples` unset and the solver adds samples 64 at a time until every one of those sigmas meets the target, running at least 128 and at most 1024. A run the cap stops sets `hit_cap` in the report, and `unconverged` lists every output, nuclide and step that missed. On `Model.simulate_transmutation(method="independent")` with the `cross_sections` source on, the count is fixed before the transport, which carries every sample: `samples`, or 256 at the default target. Every sample is solved, and whatever missed the target is reported the same way. Setting `samples` bounds cost or reproduces a specific run; `convergence` is the accuracy dial.

Omitting the argument costs nothing. No covariance is read from disk, no matrix is folded or factorized, and the inventories are bit-identical.

CHOOSING WHAT TO PERTURB

Nine sources can be perturbed, and by default every one that applies to the call is:

```
yamc.DataUncertainty.available_sources()
# ['cross_sections', 'flux_spectrum', 'half_life', 'decay_branching',
#  'statistical', 'decay_energy', 'decay_photon_lines', 'fission_yield',
#  'isomer_branching']

# isolate one contribution
xs_only = yamc.DataUncertainty(seed=42, sources=["cross_sections"])
```

- `cross_sections`: the activation cross sections, from the evaluation's MF=33 covariance. Where a nuclide's data folder carries its resonance parameters (MF=2 and MF=32, `resonance_parameters.arrow`, in the published libraries), each sample draws the parameters themselves, rebuilds the resonance range's cross sections from the draw and Doppler broadens

the change, so that range's uncertainty is exact in the parameters rather than first order. A nuclide whose parameters cannot be sampled keeps the first-order MF=32 rows of `covariance.arrow`, and the report's `resonance_parameters` says which and why. On the spectrum calls a sample's cross sections are folded against the spectrum the rates are collapsed with. On `Model.simulate_transmutation(method="independent")` nothing is folded: the one transport carries a likelihood-ratio weight per sample, as `simulate_transport(data_uncertainty=...)` does, and tallies every reaction rate, isomeric partial, fission-yield weight and flux bin once per sample under that sample's cross sections and resonance parameters. A sample's rates are then those its own data would give, flux included: a stronger resonance deepens its own self-shielding dip, and a perturbed scatterer moves every rate it shapes. The report's `flux_response` reads `"replica_weights"` there (`"supplied"` on the spectrum calls), and `transport_replicas` gives the count. With the coupled method each replica's transports read nuclear data with that replica's draw applied instead, and this source needs `statistical` beside it.

- `flux_spectrum`: the error on the spectrum you supplied, described below. The spectrum calls (`Material.transmute` and `yamc.transmute`) only. It moves every rate collapsed from the spectrum's groups, and every isomeric split and `(n,n')` channel with the groups its states are produced in. A rate it cannot move is named in `flux_rates_without_terms`, a gap.
- `half_life`: each reachable unstable nuclide's half-life, from the decay data's standard deviation on it. A sample's half-lives are used in its solve and again in the activity, decay heat and dose evaluated from its inventory. That keeps a saturated activity (`lambda N = R`) insensitive to its own half-life, where using the nominal value in either place would hand it the density's whole spread. A shutdown-dose `PulseSchedule.time_correct_tally` applies it, and `decay_photon_lines` (below), and no other source.
- `decay_branching`: the decay branching ratios of each reachable parent with two or more modes and a stated sigma, however many modes it has. Each draw keeps the parent's total and every ratio between zero and it, and each mode takes its evaluated ratio as its mean and its stated sigma as its standard deviation, so a small branch with a large relative sigma is drawn skewed rather than held. A mode is held only when its stated sigma is one no ratio with that mean can have, or it states none and is not the complement of the rest. The report names every parent held whole (`no_decay_branching_uncertainty`, `decay_branchings_too_wide`) or drawn short of its stated sigmas (`decay_branchings_partly_held`, `decay_branchings_sigma_adjusted`). Only the inventory moves: a drawn parent's lines and decay energy per decay follow its nominal branching.
- `statistical`: the Monte Carlo uncertainty of transport-tallied reaction rates, from their per-history covariance, so rates scored by the same histories move together. Model.simulate_transmutation() only. With the coupled method it gives each replica its own transport seed instead.
- `decay_energy`: each nuclide's mean decay energy, from the sigmas the decay data gives its recoverable-heat components (beta, gamma, alpha), or from the total's sigma where no component carries one. The evaluation states no correlation between the components, so they are drawn as independent and the total's sigma is taken as their quadrature sum. With `decay_photon_lines` on, as it is by default, the gamma part follows each sample's drawn photon lines instead, and the beta and alpha parts keep these draws. A decay energy never enters the solve, so this moves decay heat and nothing else.

- `decay_photon_lines`: each decay photon spectrum's normalisation (FD for lines, FC for a continuum), one draw per spectrum shared by all its lines, and each line's own intensity (dRI) and energy (dER), from the decay data's MT=457 sigmas. It moves the decay photon spectrum, the contact dose and, through the gamma decay energy, the decay heat; no photon enters the solve. The decay data does not state how a nuclide's photon intensities correlate beyond the shared normalisation, so each photon output is given at both ends: `std_dev` takes them independent and `std_dev_correlated` fully correlated.
- `fission_yield`: each fissioning parent's independent yields (MT=454), from the sigma the evaluation states on each, drawn on the tape's own products and summed onto the chain's the way the converter summed the nominal yields. No evaluation states a correlation between yields, so each is drawn independently and a draw's yields are not renormalised. It needs a chain carrying the evaluated yields (`fission_yields/evaluated_yields.arrow`); without them every parent is listed under `no_fission_yield_uncertainty`.
- `isomer_branching`: the isomeric split of every reachable reaction whose evaluation states the MF=40 covariance of its final states' production (the branching subsection's `branching_covariance.arrow`). Each state's production is drawn on the covariance's own grid, folded against the same spectrum as the nominal split, and the split remade from the draw. Where the list names its ground state only the shares move, and the reaction total stays the cross sections' own, so the total's uncertainty is not counted twice. An isomer listed alone, and every `(n,n')` isomer, is made at its drawn production, so its MF=40 sigma replaces the MF=33 sigma of its reaction that it carries under `cross_sections` alone, and can be the smaller. On `Model.simulate_transmutation(method="independent")` the transport scores each stated state's production on its covariance intervals, once per sample under the replica weights, and each sample re-splits the tallied partials from its draw. With the coupled method every step of a replica tallies its partials on production curves that carry the replica's draw.

A call ignores the sources that do not apply to it, and `sources` in the report below lists the ones that did.

Restricting `sources` isolates one contribution, so the growth of the inventory sigma as a source is added can be measured directly. Naming a source this build cannot perturb raises an error instead of being ignored, so a source that is not implemented yet cannot look like one that contributed nothing.

`flux_spectrum` propagates the error bar on the spectrum you supplied, which means the pulse has to carry one. `Pulse(flux_std_dev=...)` takes the per-bin standard deviation, in the same units and the same order as the `Histogram` values; from a yamc run that is `results[tally].standard_deviation`, scaled by whatever factor the mean was. The error is taken relative to the `Histogram` values as given, so a mean multiplied by the source strength beside an unscaled standard deviation reads a flux error smaller by that factor, and nothing flags it.

```
pulse = yamc.Pulse(
    rate=total_flux,
    duration=(1, "h"),
    source=spectrum,
    flux_std_dev=flux_sigma,    # per bin, same order as the Histogram values
)
```

A per-bin sigma treats the bins as independent, and the bins of a Monte Carlo tally are not: the same histories score them, so they move together, and their standard deviations alone understate the error of every rate that sums over a band. `Pulse(flux_covariance=...)` takes the full covariance of the `Histogram` values instead, in the square of their units, which a tally built with `Tally(covariance=True)` reports as `results[tally].covariance`. It scales with the square of the factor the mean was scaled by: a mean multiplied by the source strength `S` needs the covariance multiplied by `S**2`. Give one or the other.

Omitting both is the common case: a spectrum taken from a published reference set carries no stated error. The `flux_spectrum` source then contributes nothing and says so in the report's `spectra_without_flux_sigma`, rather than reading as a flux known exactly.

WHERE THE UNCERTAINTY COMES FROM

`DataUncertainty(attribution=True)` also says which source and which evaluation a nuclide's spread comes from, read with `get_uncertainty_breakdown`. It is off by default because it costs further solves, one ensemble per source and one deterministic solve per contributor, and it changes none of the numbers the run otherwise reports.

```
attributed = material.transmute(
    schedule=schedule,
    data_uncertainty=yamc.DataUncertainty(seed=42, attribution=True),
)
breakdown = attributed.get_uncertainty_breakdown(material_id=mid, nuclide="Mn56",
step=1)
if breakdown is not None:          # None unless attribution=True
    breakdown["variance"]          # the total, the square of get_nuclide_uncertainty
    breakdown["by_source"]         # each source alone, resampled; these sum to the
total
    breakdown["unattributed"]      # what that sum leaves: interaction and sampling
noise
    breakdown["contributors"]      # [(source, nuclide, reaction, variance)], largest
first
```

`by_source` resamples each source on its own, so it says how much of the spread is statistical and how much is each kind of nuclear data; the sources are independent and their variances add. `contributors` is first order and names the evaluation to look at: within the cross sections a nuclide's whole evaluation has a `reaction` of `None` and each channel alone names its reaction, and a half-life or a decay branching has `None` too. The total stays the resampled one. With the coupled method each `by_source` entry is a coupled ensemble of its own, and there are no `contributors`.

HOW A SAMPLE IS DRAWN

A nuclide's cross sections are drawn once per sample, over the cells of its covariance (each reaction's energy intervals), and every spectrum, step and channel reads its rate off that one draw. The relative cells are a multivariate lognormal with mean one and the evaluation's own covariance, drawn through the `ln(1 + C)` transform, so each cell keeps its stated variance and its stated correlations with the others, and every perturbed cross section is positive. A rate is linear in the cross section, so a channel's rate is its nominal rate plus its partial rates times the cells' changes, which is exact rather than a linearization. Two fully correlated cells with different sigmas, or an anticorrelation with `1 + C <= 0`, have no lognormal with that covariance; the nearest one is drawn and the nuclide is named in `lognormal_not_carried`, which describes the distribution rather than a defect of the data and is not a gap. Cells stated in barns (LB=0) and short-range ones (LB=8) are drawn as Gaussians, so a rate read off them, or off a channel that subtracts reactions, can come out negative; it is set to zero and counted in `rates_floored`, beside `rates_sampled`. For a cell whose relative sigma approaches or exceeds 1 the lognormal is heavy-tailed, so an ensemble of the size the run stops at (at most 1024) tends to report a sigma below the stated one. The standard error the stopping rule judges carries the sample kurtosis, which slows the run down on such a tail, but a tail no sample has reached yet does not show in it.

A half-life or decay energy is multiplied by a lognormal factor of the same kind, with mean one and variance equal to its squared relative sigma, drawn per nuclide (for decay energy, per beta, gamma and alpha component, or on the total where no component carries a sigma). The decay data states an expected value and a standard deviation and no correlation, so each draw keeps exactly what the evaluation states and cannot go negative, and there is no floor. A shutdown-dose `time_correct_tally` draws its half-lives the same way. A sigma no draw can carry, one stated on a decay energy of zero or one that is not finite, is held at nominal and listed in `half_life_uncertainty_not_carried` or `decay_energy_uncertainty_not_carried`.

A flux bin is scaled by a lognormal factor with mean one too: bin by bin from `flux_std_dev`, or through the same `ln(1 + C)` transform from `flux_covariance`, where a covariance no lognormal carries is drawn at the nearest one and named in `flux_lognormal_not_carried`. No flux bin is floored. The tallied rates of the `statistical` source are drawn from a Gaussian on their per-history covariance, since that carries the correlations between rates scored by the same histories. A tallied rate drawn below zero is set to zero and counted in `statistical_floored`, beside `statistical_sampled`. A count well above zero says the sigma is a large enough share of the value that a Gaussian no longer describes it.

A parent's decay branching ratios sum to a fixed total, so the deviations of a draw sum to zero. Of the zero-sum covariances that realise every stated sigma, the draw takes the one of largest entropy, which assumes no correlation the sum rule does not force (for two or three modes it is the only one); where the largest sigma exceeds the sum of the others', that mode is drawn at their sum. Every moving mode but one is a logit-normal on `[0, T]`, `T` the parent's total, fitted so that its mean and standard deviation are the evaluated ratio and its sigma, and the modes are coupled through a Gaussian copula that carries the target covariance. The last mode is their complement, which keeps the total. A draw whose complement would fall below zero is pulled back towards nominal and counted in `decay_branchings_floored`, beside `decay_branchings_sampled`.

Evaluated covariance matrices are not always positive semi-definite, and one that is not has no square root to sample from. The repair is on the correlation matrix of a nuclide's cells with a positive stated variance: it is replaced by the nearest correlation matrix and scaled back by the evaluated sigmas, so every cell is sampled at its evaluated sigma exactly and only the correlations move, by as little as a positive semi-definite matrix allows. A cell stated with a negative variance, or with a zero one and a covariance to another cell, has no correlation to keep and is held at nominal. A channel that folds several cells reads their correlations, so its sigma can still move, either way. A matrix counts as repaired when that correlation matrix has an eigenvalue below `-m * 1e-12` (`m` the number of cells with a positive stated variance), or when a cell is stated with a negative variance, or with a zero one and a covariance to another cell. Judged on the correlation matrix, the round-off of the eigendecomposition is on each cell's own scale, so a PSD matrix that comes back a few ulps negative is not a repair, and a cell with a huge variance cannot hide a real repair among the others. `covariance_repaired` lists the distinct nuclides, not a count over spectra, with a repaired channel a draw can move (a positive rate on a spectrum the schedule irradiates with), and any makes `has_gaps` true. The report covers the nuclides the material can populate: the fold takes every chain nuclide with data, which from almost any composition is the chain's whole closure, so a nuclide is reported only when an upper bound on its density over the schedule, at nominal rates, reaches the solver's floor of `1e-30` atoms/b-cm. One the bound leaves out cannot move any nominal density by as much as that floor, but a replica's rates on a wide channel can sit orders above nominal, so the repaired nuclides outside the bound with a channel a draw can move are named in `covariance_repaired_outside_bound`, and any of those makes `has_gaps` true as well, since the bound says nothing about a replica. `covariance_repairs` gives one entry per repaired populated nuclide and spectrum, including repairs no draw can move, with `lambda_min` (the most negative eigenvalue of the cells' correlation matrix before the repair), `largest_correlation_change` and `correlation_frobenius_change` (how far that correlation matrix moved), `cells`, `held_cells`, `converged` and, per channel, `evaluated_variance` (the folded diagonal as stated, which can be negative), `evaluated_sigma` (`None` for a negative variance) and `sampled_sigma`. `worst_sigma_change` is the largest `|sampled / evaluated sigma - 1|` over the repaired channels a draw can move, and `rate_weighted_sigma_change` is the weighted mean of each channel's own `|sampled / evaluated sigma - 1|` over every sampled channel of a populated nuclide, each weighted by its rate at unit flux times its spectrum's fluence in the schedule times its parent's initial density. The weight is on each channel's change, not on its sigma, so a wide channel with a small rate cannot drown out a repair on the channels that carry the reactions. The initial density makes it a measure of first-generation reactions: a nuclide the material starts without carries no weight, and its repairs show in `worst_sigma_change` and `covariance_repairs` instead. The sampled sigma is read off the factorization for every matrix, so on a matrix that needed no repair the decomposition's round-off shows here as it is, far below any real repair. Either headline can be `float('inf')` when a repair gives spread to a channel stated with none, and strict JSON does not accept that.

`sigma_at_least_one` and `sigma_at_least_ten` name the sampled channels, keyed `"Nuclide (n,x)"`, of populated nuclides with a positive rate on a spectrum the schedule irradiates with, whose folded relative sigma as evaluated is at least one or ten. At that width the spread depends on the

lognormal chosen to carry the evaluation's two moments, not on the evaluation alone. `sigma_at_least_one_outside_bound` does the same for the nuclides outside the populated bound, since a replica's draw on exactly such a channel can sit orders above nominal and populate the nuclide; the ten-or-more subset reads off its values. They are not a gap.

WHAT IT DOES NOT COVER

Every input below is held at its evaluated or nominal value in every sample, or drawn without a correlation the data states. Some carry published uncertainties the converted data does not hold yet, some have none to read, and some are a property of the model rather than of the data. Most leave a term out of the reported spread, so the sigma is too small by it. Some can go either way: self-shielding and, on the spectrum calls, the flux's response to the data, because holding them also removes a feedback that narrows the spread, and the dropped correlations, because a missing correlation can raise or lower the sigma of a sum. Each item says which way it goes, and where.

- **Decay branching ratios the draw holds.** A mode whose stated sigma no ratio with its mean can have, a mode stating no sigma that is not the complement of the rest, and a parent with no sigma at all are held at their evaluated ratios, and each is listed in the report under the reason. A drawn parent's per-decay emission, its lines and decay energy, stays that of the nominal scheme. The sigma is read from the decay data's `branching_ratio_uncertainty` column, so a library converted before that column existed carries none and lists its parents under `no_decay_branching_uncertainty`.
- **Fission yield correlations.** The `fission_yield` source draws each stated yield independently, since no evaluation publishes a correlation between yields, and does not renormalise a draw's yields to the nominal total. A parent whose tape yields do not sum back to the chain's (`fission_yields_mapping_mismatch`) is held. On the spectrum calls the weights that combine a nuclide's yield sets by incident energy come from the nominal spectrum in every sample; on `Model.simulate_transmutation` each sample's are tallied under its own data.
- **Isomeric branching without MF=40.** The `isomer_branching` source draws the split between a product's ground and metastable states where the evaluation states MF=40. A split with no MF=40 on any of its states is held and listed in `no_isomer_branching_uncertainty`, and MF=40 blocks that cannot be used are counted per parent in `isomer_branching_blocks_not_used`. ENDF/B-VIII.1, FENDL-3.2d and TENDL state no correlation in MF=40 between a product's states, or with the reaction's MF=33, so the split's uncertainty is only partly stated (Covariance is presence, not coverage). With `transmutation_branch_ratios` configured, the `(n,n')` channels the overlay adds to reach a metastable state, In115 to In115m among them, come from its MF=10 partials with no group-averaged cross section behind them. On the spectrum calls their cross-section covariance is MT=4's, weighted as MT=4 while their rate is the metastable production, so a relative MT=4 block lists them in `partials_above_rate`, where the sigma is overstated.
- **Resonance-parameter covariance (MF=32) neither sampled nor written.** Many evaluations keep the resolved-resonance uncertainty on capture, elastic and fission in MF=32 and leave MF=33 at zero over that range, which reaches down to thermal. Where the data folder carries the resonance parameters, the `cross_sections` source samples them exactly; otherwise the converter's first-order MF=32 rows in `covariance.arrow` are folded. What is left held is a range whose formalism

is not reconstructed, and data converted before either was written, on which a product made by capture at thermal or resonance energies draws little or none of its cross section's uncertainty. ENDF/B-VIII.1 W186 is the example: its resonance parameters carry about 1.5% on capture below 10 keV, where its MF=33 gives zero, so with MF=33 alone the nuclide is still listed under `perturbed` and the gap shows in that channel's `rate_fraction_covered`, which counts only rate from energies with a nonzero stated variance: W186 `(n,gamma)` reads about 0.07 on a fusion spectrum.

- **MF=33 blocks that are not used.** What is folded is each reaction's own MF=33 blocks, including those whose `MAT1` names the evaluation itself, and four more kinds. A lumped reaction (MT=851 to 870) with one component is that component; with several, a component with no covariance of its own takes the lump's relative perturbation, so the components move together and their sum moves as the lump states. Tungsten's `(n,2n)`, the channel behind most of a tungsten first wall's decay heat, is one: ENDF/B-VIII.1, JEFF-4.0 and FENDL-3.2d give its covariance only lumped with `(n,2np)`. A level sum with no covariance of its own is derived from its levels, each reading its own blocks or its lump's, so W186 `(n,p)` reads MT=855, the sum of MT=600 and MT=649. An LTY=0 NC block's reaction gets the covariance derived from the reactions it names (ENDF/B-VIII.1 O16 `(n,p)` is stated only that way). A block correlating two evaluations the run both holds is applied and counted in `applied_cross_material`, and its two nuclides are drawn jointly, one draw per sample shared by every material: the neutron standards, ENDF/B-VIII.1 U235 fission with U238 fission and capture, Pu239 fission, Au197 capture and Li6 `(n,t)`. Where the covariance stated between them is not admissible beside each one's own, the closest admissible one is drawn and the group is named in `cross_material_adjusted`, a gap. Four kinds are left out. A block correlating with an evaluation the run does not hold, or where one of the two reactions has no relative covariance to correlate, is skipped and counted in `skipped_cross_material`, and so is a block whose partner is not a cross section (`XMF1` other than 0 or 3, `skipped_other_file`). NC blocks that cannot be derived (LTY 1 to 4, or an LTY=0 block that fails, counted in `skipped_nc`) are skipped, so a reaction whose covariance is given only that way draws none. A lump stated only in barns (an absolute or short-range block) says nothing of how its sum splits, so it is not given to its components and is listed in `lumped_covariance_not_assignable`; no published library has one. Blocks on partial levels (MT=600 to 849 and 875 to 891) are folded where an NC block or a level sum names them, and are otherwise neither used nor counted, since the chain drives the total and has no rate for a level alone. A nuclide whose only blocks are of the skipped kinds is listed under `no_covariance_data`, and a channel missing its covariance shows in `rate_fraction_covered` and a lower `rate_fraction_covered_total`.
- **Self-shielding.** With `self_shielding_shape` or `self_shielding_chord` set, every sample uses the flux depression computed from the evaluated cross sections, and that pulls two ways. A perturbed capture cross section neither deepens nor relieves its own dip, and that feedback narrows the spread, since a larger capture then absorbs from a smaller flux. The elastic and total covariance never reaches the depression either, and that term widens the spread where the evaluation states a resonance-range elastic uncertainty. Which wins depends on the evaluation. For Au197 capture in a 0.1 mm foil under a 1/E spectrum on VITAMIN-J-175, ENDF/B-VIII.1 states no such elastic uncertainty, and the held sigma is too large: 1.70% held against 0.96% with the feedback. TENDL-2017 states one, and there the held sigma is too small: 5.34% held against 6.07% with the feedback

and the elastic term together. Those figures fold the shielded partial rates, as the covariance fold does: on a self-shielded run the partials come from the same shielded flux as the rate they are divided by.

- **The flux's response to the data, on the spectrum calls.** In `Material.transmute` and `yamc.transmute` the flux is your input, and a perturbed cross section does not change it. The term left out is the flux's response to each perturbed channel, which grows with how much that channel shapes the flux. It is negligible for a trace nuclide or a weak channel, and not for a material that shapes its own flux or for the main constituents of a shield. A larger Li6(n,t) cross section in a breeder depresses the flux it sees, the same narrowing feedback as self-shielding, so holding the flux overstates that rate's sigma. A shield's cross sections move the flux of every product behind it, however trace, and leaving that out understates those products' sigmas. Elastic scattering, which shapes the flux without transmuting anything, is not drawn on these calls at all. `Model.simulate_transmutation` carries all of this, elastic scattering included: with `method="independent"` its transport's replica weights move the flux with each sample's cross sections, and with `method="coupled"` each replica is transported on its own drawn data.
- **Decay photon spectrum covariance and continuum shape.** The `decay_photon_lines` source draws each spectrum's normalisation and each line's intensity and energy. MT=457 can also state a covariance between a spectrum's lines (LCOV), and no library ships one, so the lines of a spectrum are drawn independently apart from their shared normalisation, and the photon outputs are given at both ends of that range (`std_dev` and `std_dev_correlated`). A continuum's shape is held, and only its normalisation is drawn. A shutdown dose draws the normalisations alone.
- **Dose constants.** Contact dose is built from the material's photon mass attenuation coefficients, the air mass energy-absorption coefficients or the ICRP-116 conversion coefficients (by `dose_quantity`), and a constant build-up factor (`build_up` , 2.0 by default). A shutdown dose tally is scored with fluence-to-dose coefficients. None of these tables publishes a per-value uncertainty and the build-up factor is a model constant, so none is perturbed.
- **Natural isotopic abundances.** An element is expanded into its isotopes with the same abundances in every sample. The published isotopic-composition tables state an uncertainty or a range of natural variation on every isotope of a multi-isotope element, above 1% of the value for about a third of them (Li6's stated range, which includes commercially depleted lithium, spans a factor of four). The bundled table carries each isotope's stated uncertainty, or for the twelve elements the tables give as an interval, that interval, readable through `yamc.data.natural_abundance_records()` . For an interval element the tables state no distribution within it. Every sample uses the lookup value, so none of that spread is propagated: the tables' representative value, or for an interval element a single best measurement, which for Li6 (0.07589) sits near the top of its interval, [0.019, 0.078].
- **Atomic masses.** Each nuclide's mass, from AME2020, converts between mass and atoms the same way in every sample. AME2020 states an uncertainty on every mass, and the bundled table does not carry it.
- **Element fractions and density.** These are your input and start every sample at the values you gave. There is no argument to give their uncertainty, so it is not perturbed, and propagating it means

sampling compositions yourself, one run per composition. For dose from a trace impurity it can outweigh everything above: in a 316L-type steel with 0.1 wt% cobalt, irradiated for five years at 1e14 n/cm^2/s in an illustrative first-wall spectrum, the contact dose at ten years' cooling is almost all Co60, about 70% of it made from the cobalt itself, so a 30% uncertainty on the cobalt content is about 21% on the dose.

The report's `not_perturbed` names what the run held at nominal. Every run names the isomeric branching of reactions with no stated MF=40, the MT=5 product multiplicities (how the drawn `(n,X)` rate splits among its products), the MF=33 blocks that are not used (with an evaluation the run does not hold, a partner that is not a cross section, NC blocks that cannot be derived, and lumps stated in barns), MF=32 neither sampled nor in `covariance.arrow`, the decay photon spectrum covariance and continuum shape, the three dose tables and the build-up factor, material composition and density, natural isotopic abundance and atomic mass (AME2020). The rest depend on the run. Half-life, decay branching, decay energy, decay photon lines, fission yield, isomeric branching and the activation cross sections are named when `sources` leaves them out, and when decay branching is drawn for any parent, the per-branch decay emission (lines and decay energy following the nominal branching) is named instead. On a spectrum call the flux spectrum is named when no spectrum's flux is drawn, and named as held for spectra without a sigma only when some are drawn and some are not. On a transport call the tallied-rate statistics are named when the `statistical` source is off or there are no tallied rates to draw. The self-shielding correction is named whenever self-shielding is set.

Because a reported sigma of zero can mean "well known" or "nothing published", the report, `get_data_uncertainty_info`, keeps a nuclide with no usable MF=33 (`no_covariance_data`: none in the evaluation, or none this run could use, such as blocks all skipped as above or all for reactions the run does not drive) apart from one that states it. A channel whose stated covariance is zero over part of its range, as in the MF=32 case above, stays under `perturbed`, and `rate_fraction_covered` gives the share of its rate from energies with a nonzero stated variance:

```
info = results.get_data_uncertainty_info(material_id=mid)
if info is not None:            # None unless data_uncertainty was passed
    info["sources"]             # the sources that applied to this run
    info["perturbed"]           # nuclides that had usable MF=33 covariance
    info["no_covariance_data"]  # nuclides with none usable, absent or skipped
    info["rate_fraction_covered"]  # per channel, the share of the rate the fold
                                   # divides by (shielded on a self-shielded run)
                                   # from energies with a nonzero stated variance
    info["rate_fraction_covered_total"]  # ... over the run, by production; None on
                                         # a transport run or when a partials map
                                         # lists a channel
    info["skipped_cross_material"]    # blocks naming an evaluation not in the run
    info["applied_cross_material"]    # ... and those applied, drawn jointly
    info["skipped_other_file"]        # blocks whose partner is not a cross section
    info["skipped_nc"]                # NC blocks that could not be derived
    info["lumped_covariance_not_assignable"]  # lumps stated only in barns
    info["partials_above_rate"]  # channels whose partial rates add up to more
                                 # than the rate: sigma overstated
    info["partials_below_rate"]  # ... add up to less: sigma understated
    info["sigma_at_least_one"]   # channels with a relative sigma of 1 or more
    info["half_lives_perturbed"]      # nuclides whose half-life was drawn
    info["no_half_life_uncertainty"]  # ... and those whose data states no sigma
    info["decay_branchings_perturbed"]      # multi-mode parents whose split was
drawn
    info["no_decay_branching_uncertainty"]  # multi-mode parents with no sigma
    info["decay_branchings_partly_held"]    # drawn with some modes held
    info["decay_branchings_sigma_adjusted"] # drawn short of a stated sigma
    info["decay_photon_lines_perturbed"]    # nuclides whose photon data was drawn
    info["fission_yields_perturbed"]        # fissioning parents whose yields were
drawn
    info["isomer_branchings_perturbed"]     # reactions whose isomeric split was
drawn
    info["no_isomer_branching_uncertainty"] # splits held, no MF=40 on any state
    info["resonance_parameters"]  # per nuclide: parameters sampled, or
                                  # first-order rows and why
    info["decay_energies_perturbed"]  # nuclides whose decay energy was drawn
    info["no_decay_energy_uncertainty"]  # ... and those whose data states none
    info["half_life_uncertainty_not_carried"]     # a stated sigma no draw can
    info["decay_energy_uncertainty_not_carried"]  # carry, held at nominal
    info["flux_response"]       # "supplied", or "replica_weights" on a transport run
    info["samples"]             # how many samples ran
    info["converged"]           # True if every tracked sigma met the target
    info["unconverged"]         # ... and the ones that did not, worst first
    info["not_perturbed"]       # held inputs this run names, as above
    info["has_gaps"]            # True if a nuclide, covariance block or spectrum
                                # had no usable uncertainty, a block was skipped,
                                # malformed or of an unsupported layout, either
                                # partials check listed a channel, a covariance
                                # was repaired, a half-life or decay energy had
                                # no stated sigma or one no draw can carry, a
                                # parent's decay branching was held or drawn
                                # short, or an isomeric split had no MF=40
```

A `sigma` of `0.0` for a nuclide listed in `no_covariance_data` is not a claim of certainty. It means the evaluation says nothing this run could use, and the same goes for a half-life or decay energy listed as

having no stated sigma.

UNCERTAINTY ON ACTIVITY, DECAY HEAT AND DOSE

Activity, decay heat, contact dose and the decay-photon spectrum are all functions of a whole inventory, so their spread has to be taken over the ensemble rather than built from per-nuclide sigmas. Each has a getter that does this, evaluating the quantity once per replica and taking the spread of the result. On `Model.simulate_transmutation` with replica weights, every sample's rates come from the same transport histories, and the Monte Carlo noise those carry is measured (each sample is solved again from the even half of the histories) and removed from these spreads and from `get_nuclide_uncertainty`; the report's `noise_removed` says it was, and `noisy` names the tracked sigmas where the noise removed was large, which more histories would sharpen:

```
activity = results.get_activity_uncertainty(material_id=mid, step=1)   # Bq
heat = results.get_decay_heat_uncertainty(material_id=mid, step=1)     # W
dose = results.get_contact_dose_uncertainty(material_id=mid, step=1)   # Gy/h

if activity is not None:
    print(f"{activity.nominal:.4g} +/- {activity.std_dev:.2g} Bq "
          f"({activity.relative_std_dev:.1%}, {activity.replicas} replicas)")
```

Each returns an `Estimate`. `nominal` is the unperturbed run and is there whether or not uncertainty was asked for, while `mean`, `std_dev` and `relative_std_dev` are `None` below two replicas, because a spread over fewer than two samples is unmeasured rather than zero. Comparing `mean` against `nominal` is useful, but a gap between them needs interpreting. With the cross sections alone and little burnup, the inventory is close to linear in the drawn rates, whose mean is the nominal rate, so the two agree to within the sampling error. With half-lives perturbed they need not: a nuclide's activity after a long cooling is convex in its drawn half-life, so its mean sits above nominal although the draw itself is unbiased, by nearly 20% for a 3% half-life sigma thirty half-lives into cooling. A product reached through two or more perturbed steps that share a draw (the flux or the statistical draw) is nonlinear in that draw too: its mean moves from nominal by about the covariance of the shared draws, the product of the steps' relative sigmas times their correlation, with no bias in the draw. That is above nominal for positively correlated steps, such as two rates folded from one flux given by per-bin sigmas, and below it where a stated correlation is negative. Outside these cases, a gap says the perturbation is biased rather than merely wide, and the floored counters above say where a draw was cut.

`by_nuclide=True` returns a `dict[str, Estimate]` instead of one total. Those do not add up to the total in quadrature, and are not meant to.

Activity, decay heat and the photon spectrum need the material's `volume`, since a quantity in becquerel does. Contact dose does not, because the estimate takes the material for a half-space.

Contact dose is also the one quantity here that is not linear in the atom densities: the material attenuates its own decay photons, so the emitters sit in the numerator and the whole inventory in the denominator, and a replica that makes more of an emitter also absorbs more of it. Scaling the nominal dose by the density spread would report the activity's uncertainty, which is wrong by the whole value; evaluating per replica captures the cancellation.

The decay-photon spectrum comes back line by line:

```
lines = results.get_decay_photon_spectrum_uncertainty(material_id=mid, step=1)
for line in (lines or [])[:5]:
    print(f"{line.energy:10.1f} eV  {line.nominal:.3e} photons/s  "
          f"emitted in {line.emitting}/{line.replicas} replicas")
```

The set of lines is not the same in every replica: a nuclide that falls below the density floor in one draw takes its lines out of that draw. The spectrum is reported over the union, a line missing from a replica counts as a zero in it, and `emitting` says how many replicas emitted it at all, which separates a dim line from one that is sometimes absent.

For any other quantity, `get_uncertainty_inventories` returns every replica's full inventory to evaluate over, as raw replicas, with no noise removed:

```
inventories = results.get_uncertainty_inventories(material_id=mid, step=1)
# evaluate the quantity once per replica, then take its spread
```

The inventories are densities only: a replica's drawn half-lives and decay energies are not returned with them. Evaluating a quantity that reads a decay constant or a decay energy from them with the chain's nominal values gives it the wrong spread, the one the `half_life` item under Choosing what to perturb describes: a saturated activity then picks up the density's whole spread. Activity, decay heat, contact dose and the photon spectrum have the getters above. For any other such quantity, run with `sources` leaving out `half_life` and `decay_energy`, and read its spread as excluding those two.

Summing per-nuclide sigmas in quadrature instead would assume the nuclides vary independently, and they do not: a parent and its daughter move together, because every daughter atom came out of a parent atom.

Displacement damage

`displacement_damage=True` makes `transmute()` (and `yamc.transmute()`) also compute the radiation damage the schedule does: the damage energy deposited per atom and the NRT displacements per atom (dpa, ASTM E521). Each nuclide's MT=444 damage-energy cross section is folded against the pulse spectrum by the same collapse as the reaction rates, at each step's composition and flux, and cooldowns add nothing. Off by default, and then MT=444 is not fetched and nothing about the solve changes.

Pass `data_uncertainty` as well and the dpa comes back with its nuclear-data spread. Below, a year in the three-group spectrum of the example above, with every source that applies:

```
iron = yamc.Material(composition={"Fe": 1.0}, density=7.874, volume=1.0)
results = iron.transmute(
    schedule=yamc.PulseSchedule([
        yamc.Pulse(rate=total_flux, duration=(1, "a"), source=spectrum),
        yamc.Cooldown(duration=(1, "d")),
    ]),
    displacement_damage=True,
    data_uncertainty=yamc.DataUncertainty(seed=42),
)

results.get_dpa(material_id=iron.id)                # one value per state, cumulative
# [0.0, 6.4939, 6.4939]
results.get_dpa(material_id=iron.id, element="Fe")  # one element's dpa
results.get_damage_energy(material_id=iron.id)      # eV per atom, before the NRT
conversion

dpa = results.get_dpa_uncertainty(material_id=iron.id, step=1)
# Estimate(nominal=6.4939e0, std_dev=1.5820e-1 +/- 6.90e-3, replicas=256)
results.get_damage_energy_uncertainty(material_id=iron.id, step=1)
```

The values are cumulative, one per state aligned with `times`: entry 0 is the initial composition and is zero. The material total weights each element's dpa by its atom fraction, and `get_displacement_damage_info` gives the displacement threshold energy used for each element and its source, and lists the nuclides with no MT=444 (whose damage is not counted). `displacement_energies={"Fe": 40.0}` replaces a threshold.

The dpa above is 6.49 with a 2.4% sigma from the iron cross sections. Each replica's dpa is accumulated over its own inventory, with the damage-energy rate moved by its draw:

- `cross_sections`: MT=444 has no covariance of its own, but it is the sum over reactions of each cross section times the damage energy that reaction deposits, so a draw moves it by each reaction's deposit times its drawn change. Elastic and inelastic scattering, which carry most of it, are drawn with the activation channels as one evaluation. `get_data_uncertainty_info(...)["damage_energy"]` gives, per nuclide of the starting composition, the share of its MT=444 rate the reactions' deposits reproduce (`explained`, 1.00 for Fe56 here) and the relative sigma of its drawn rate (`sigma`, 2.8% for Fe56). The part they do not reproduce, fission's share above all, is held at nominal.
- `flux_spectrum`: the damage energy's own group terms under the replica's flux draw, where the pulse carries a `flux_std_dev` or `flux_covariance`.

The damage energy of transmutation products, short-range (LB=8) covariance blocks and the displacement threshold energies are held at nominal. Asking for damage widens each nuclide's drawn cross sections to the reactions its damage energy comes from, so for a given seed the other draws differ from those of a run without `displacement_damage`.

See also Model.simulate_transmutation() for coupled transport-transmutation.

PNNL material list

Every material in `yamc.materials.pnnl`, with the entry number from PNNL-15870, Rev. 2 so a composition can be cited back to the report.

Names are the report's own, verbatim. Pass one straight to the collection:

```
import yamc

steel = yamc.materials.pnnl["Steel, Stainless 304"]
```

See Material collections for search, overrides and how the compositions are stored. Composition values are atom fractions; an element symbol is expanded with yamc's natural abundances, while an explicit nuclide is used as given.

The table below is generated from the bundled data when these docs are built, so it always matches the shipped collection.

410 materials. Source: PNNL-15870, Rev. 2.

#	Name	Density (g/cm³)	Formula	Composition (atom fractions)
1	A-150 Tissue-Equivalent Plastic	1.127		H 0.58363, C 0.374869, N 0.014531, O 0.018984, F 0.005324, Ca 0.002662
2	Acetone	0.7899	C3H6O	H 0.6, C 0.3, O 0.1
3	Acetylene	0.0010967	C2H2	H 0.5, C 0.5
4	Air (dry, near sea level)	0.001205		C 0.00015, N 0.784429, O 0.21075, Ar 0.004671
5	Alanine	1.42		H 0.53844, C 0.230784, N 0.076925, O 0.153851
6	Aluminum	2.6989	Al	Al 1
7	Aluminum Oxide	3.97	Al2O3	O 0.6, Al 0.4
8	Aluminum, alloy 2024-O	2.78		Mg 0.017157, Al 0.955162, Si 0.002804, Ti 0.000494, Cr 0.000303, Mn 0.003036, Fe 0.00141, Cu 0.019031, Zn 0.000603
9	Aluminum, alloy 2090-T83	2.59		Li 0.083227, Mg 0.001729, Al 0.90171, Si 0.000598, Ti 0.000527, Cr 0.000162, Mn

				0.000153, Fe 0.000361, Cu 0.010951, Zn 0.000257, Zr 0.000325
10	Aluminum, alloy 3003	2.73		Al 0.987924, Si 0.003221, Mn 0.006198, Fe 0.00189, Cu 0.000536, Zn 0.000231
11	Aluminum, alloy 4043-O	2.69		Be 1.5e-05, Mg 0.000316, Al 0.944964, Si 0.050758, Ti 0.000643, Mn 0.00014, Fe 0.002203, Cu 0.000726, Zn 0.000235
12	Aluminum, alloy 5086-O	2.66		Mg 0.044472, Al 0.947942, Si 0.002062, Ti 0.000454, Cr 0.00078, Mn 0.002213, Fe 0.001296, Cu 0.000228, Zn 0.000553
13	Aluminum, alloy 6061-O	2.7		Mg 0.011162, Al 0.97733, Si 0.005796, Ti 0.000496, Cr 0.001017, Mn 0.000433, Fe 0.001986, Cu 0.001174, Zn 0.000606
14	Aluminum, alloy 7075-O	2.81		Mg 0.029013, Al 0.933052, Si 0.002353, Ti 0.00069, Cr 0.001248, Mn 0.000902, Fe 0.001479, Cu 0.007102, Zn 0.024161
15	Ammonia (liquid at T= -79 C)	0.771	NH3	H 0.75, N 0.25
16	Anthracene	1.28	C14H10	H 0.416667, C 0.583333

17	Argon	0.001662	Ar	Ar 1
18	Asbestos (Chrysotile)	2.53	Mg3Si2O9H4	H 0.222222, O 0.5, Mg 0.166667, Si 0.111111
19	Asphalt	1.3		H 0.586745, C 0.402599, N 0.002463, O 0.001443, S 0.006703, V 4.4e-05, Ni 3e-06
20	Asphalt pavement	2.5784		H 0.129089, C 0.153985, N 0.000449, O 0.459406, Na 0.00991, Mg 0.022142, Al 0.035271, Si 0.12081, P 0.000251, S 0.005816, K 0.007091, Ca 0.044214, Ti 0.001238, V 8e-06, Mn 0.000135, Fe 0.010184, Ni 1e-06
21	Bakelite	1.25		H 0.431804, C 0.488651, O 0.079545
22	Barium Fluoride	4.89	BaF2	F 0.666667, Ba 0.333333
23	Barium sulfate	4.5	BaSO4	O 0.666666, S 0.166667, Ba 0.166667
24	Benzene	0.8786	C6H6	H 0.5, C 0.5
25	Beryllium	1.848	Be	Be 1
26	Beryllium Carbide	1.9	Be2C	Be 0.666667, C 0.333333
27	Beryllium Oxide	3.01	BeO	Be 0.5, O 0.5
28	Bismuth	9.747	Bi	Bi 1

29	Bismuth Germanate (BGO)	7.13	Bi4Ge3O12	O 0.631579, Ge 0.157895, Bi 0.210526
30	Bismuth Iodide	5.778	BiI3	I 0.75, Bi 0.25
31	Blood (ICRP)	1.06		H 0.634596, C 0.052293, N 0.013288, O 0.298052, Na 0.000505, Mg 1e-05, Si 7e-06, P 7.1e-05, S 0.000362, Cl 0.000492, K 0.000262, Ca 9e-06, Fe 5.2e-05, Zn 1e-06
32	Bone Equivalent Plastic, B-100	1.45		H 0.513799, C 0.353638, N 0.012142, O 0.015863, F 0.069704, Ca 0.034854
33	Bone Equivalent Plastic, B-110	1.785		H 0.386395, C 0.335513, N 0.031096, O 0.031063, F 0.143965, Ca 0.071968
34	Bone, Compact (ICRU)	1.85		H 0.527877, C 0.192482, N 0.01603, O 0.213112, Mg 0.000684, P 0.018794, S 0.000519, Ca 0.030502
35	Bone, Cortical (ICRP)	1.85		H 0.475379, C 0.121907, N 0.030412, O 0.282853, Mg 0.000918, P 0.03438, S 0.000997, Ca 0.053138, Zn 1.6e-05
36	Boral (65% Al-35% B4C)	2.53		B 0.454449, C 0.113488, Al 0.432063
37	Boral (Aluminum 10% boron alloy)	2.6		B 0.217839, Na 0.005123, Al 0.767405, Si 0.002097, K

				0.006025, Ti 0.000246, Fe 0.001265
38	Boral (Aluminum 5% boron alloy)	2.6		B 0.116664, Na 0.005487, Al 0.86798, Si 0.001797, K 0.006453, Ti 0.000264, Fe 0.001355
39	Borax	1.73	Na2B4O17H20	Na 0.046512, B 0.093023, O 0.395349, H 0.465116
40	Boric Acid	1.5	BO3H3	H 0.428571, B 0.142857, O 0.428571
41	Boron	2.37	B	B 1
42	Boron Carbide	2.52	B4C	B 0.8, C 0.2
43	Boron Fluoride (B2F4)	0.004058	B2F4	B 0.333333, F 0.666667
44	Boron Fluoride (BF3)	0.002831	BF3	B 0.25, F 0.75
45	Boron Oxide	1.812	B2O3	B 0.4, O 0.6
46	Brain (ICRP)	1.03		H 0.654704, C 0.06227, N 0.005654, O 0.274958, Na 0.000477, Mg 3.7e-05, P 0.000682, S 0.000329, Cl 0.000397, K 0.000473, Ca 1.3e-05, Fe 5e-06, Zn 1e-06
47	Brass (typical composition)	8.07		Fe 0.000992, Cu 0.668462, Zn 0.318026, Sn 0.001437, P 0.011083

48	Brick, Common Silica	1.8		O 0.663427, Al 0.003747, Si 0.323229, Ca 0.007063, Fe 0.002534
49	Brick, Fire	2.1		O 0.636335, Mg 0.005057, Al 0.160954, Si 0.183806, Ca 0.003578, Ti 0.005135, Fe 0.005135
50	Brick, Kaolin (white)	2.1		O 0.635743, Mg 0.001008, Al 0.181264, Si 0.175774, Ca 0.000362, Ti 0.004323, Fe 0.001526
51	Bronze (typical composition)	8.4		Al 0.065612, Si 0.007378, Mn 0.004016, Fe 0.011343, Ni 0.007103, Cu 0.853653, Zn 0.034205, Sn 0.012809, Pb 0.003881
52	C-552 Air-Equivalent Plastic	1.76		H 0.268591, C 0.45814, O 0.003104, F 0.268613, Si 0.001552
53	CELOTEX (Lignocellulosic Fiberboard)	0.25	C6H10O5	C 0.285714, H 0.47619, O 0.238095
54	CLLB(Ce) - Cesium Lithium Lanthanum Bromide - 0.3 wt%	4.2		Cs 0.19994, Li 0.09997, La 0.09997, Br 0.599819, Ce 0.000301
55	Cadmium	8.65	Cd	Cd 1
56	Cadmium Nitrate Tetrahydrate	2.45	CdN2O10H8	H 0.380952, N 0.095238, O 0.47619, Cd 0.047619
57	Cadmium Telluride	6.2	CdTe	Cd 0.5, Te 0.5

58	Cadmium Tungstate (CWO)	7.9	CdWO4	O 0.666666, Cd 0.166667, W 0.166667
59	Cadmium Zinc Telluride (CZT)	5.78	CdZnTe	Cd 0.333333, Zn 0.333333, Te 0.333333
60	Calcium Carbonate	2.8	CaCO3	C 0.2, O 0.6, Ca 0.2
61	Calcium Fluoride	3.18	CaF2	F 0.666667, Ca 0.333333
62	Calcium Oxide	3.3	CaO	O 0.5, Ca 0.5
63	Calcium Sulfate	2.96	CaSO4	O 0.666666, S 0.166667, Ca 0.166667
64	Calcium Tungstate	7.9	CaWO4	Ca 0.166667, W 0.166667, O 0.666666
65	Carbon Dioxide	0.001842	CO2	C 0.333333, O 0.666667
66	Carbon Tetrachloride	1.594	CCl4	C 0.2, Cl 0.8
67	Carbon, Activated	0.45		B 1e-06, C 0.999999
68	Carbon, Amorphous	2		B 1e-06, C 0.999999
69	Carbon, Graphite (reactor grade)	1.7		B 1e-06, C 0.999999
70	Cat litter (clumping)	1.1		H 0.437432, O 0.437322, Na 0.003988, Al 0.039762, Si 0.079508, Ca 0.001988
71	Cat litter (non-clumping)	0.8		H 0.213278, O 0.528384, Na 0.02947, Mg 0.032513, Al 0.030251, Si 0.163456, K 0.00151, Ca 0.000524, Fe 0.000614

72	Cellulose	1.5	C6H10O5	H 0.47619, C 0.285714, O 0.238095
73	Cellulose Acetate	1.42		H 0.476179, C 0.285724, O 0.238097
74	Ceric Sulfate Dosimeter Solution	1.03		H 0.65908, N 0.000353, O 0.337663, S 0.002816, Ce 8.8e-05
75	Cerium Bromide	5.2	CeBr3(Ce)	Ce 0.25, Br 0.75
76	Cerium Fluoride	6.16	CeF3	F 0.75, Ce 0.25
77	Cesium Iodide - 1 wt% Sodium doped	4.51		Cs 0.497513, I 0.497512, Na 0.004975
78	Cesium Iodide - 1 wt% Thalium doped	4.51		Cs 0.497512, I 0.497513, Tl 0.004975
79	Cesium Lithium Yttrium Chloride (CLYC)	3.31	Cs2LiYCl6	Cs 0.2, Li 0.1, Y 0.1, Cl 0.6
80	Cesium Lithium Yttrium Chloride (CLYC) with 95% Li6	3.31	Cs2LiYCl6	Cs 0.2, Li6 0.095682, Li7 0.004318, Y 0.1, Cl 0.6
81	Chromium	7.18	Cr	Cr 1
82	Clay	2.2		O 0.633298, Na 0.006923, Mg 0.009202, Al 0.094688, Si 0.219137, P 7.6e-05, K 0.01093, Ca 0.009895, Ti 0.00204, Mn 2.4e-05, Fe 0.013787
83	Coal, Anthracite	0.84		H 0.228604, C 0.749029, N 0.006169, O 0.014402, S 0.001796

84	Coal, Bituminous	0.75	H 0.421415, C 0.533658, N 0.008665, O 0.033187, S 0.003075
85	Coal, Lignite	0.75	H 0.357496, C 0.519327, N 0.00735, O 0.114221, S 0.001605
86	Concrete [Los Alamos (MCNP) Mix]	2.25	H 0.084736, O 0.604078, Si 0.241925, Al 0.024842, Na 0.012523, Ca 0.027244, Fe 0.004652
87	Concrete, Barite (Type BA)	3.35	H 0.109599, O 0.600196, Mg 0.001515, Al 0.004777, Si 0.011474, S 0.103647, Ca 0.038594, Fe 0.026213, Ba 0.103985
88	Concrete, Barytes-Limonite	3.36	H 0.235412, O 0.548167, Na 0.000911, Mg 0.002201, Al 0.004311, Si 0.011183, S 0.05499, Ca 0.03117, Mn 0.000593, Fe 0.056904, Ba 0.054158
89	Concrete, Boron Frits-baryte	3.1	H 0.147509, B 0.025537, O 0.560956, F 0.003215, Na 0.013975, Mg 0.002513, Al 0.006298, Si 0.031294, S 0.075765, K 0.000679, Ca 0.041475, Mn 9.7e-05, Fe 0.010413, Zn 0.00268, Ba 0.077594
90	Concrete, Colemanite-baryte	3.2	H 0.208721, B 0.022432, O 0.539767,

			Na 0.001184, Mg 0.002241, Al 0.005596, Si 0.015511, S 0.074331, Ca 0.052248, Mn 4.5e-05, Fe 0.004565, Ba 0.073358
91	Concrete, Ferro-phosphorus	4.8	H 0.158638, O 0.207882, Mg 0.002632, Al 0.004741, Si 0.038716, P 0.203404, Ca 0.033514, Fe 0.350473
92	Concrete, Hanford Dry	2.18	H 0.078437, O 0.595591, Na 0.001864, Mg 0.011462, Al 0.05083, Si 0.195334, K 0.006577, Ca 0.039567, Fe 0.020338
93	Concrete, Hanford Wet	2.35	H 0.205169, O 0.539084, Na 0.001462, Mg 0.008992, Al 0.03988, Si 0.153253, K 0.00516, Ca 0.031043, Fe 0.015956
94	Concrete, Iron-Portland	5.9	H 0.135596, O 0.150642, Mg 0.002215, Al 0.005066, Si 0.013419, S 0.000646, Ca 0.040918, Mn 0.002638, Fe 0.64886
95	Concrete, Iron-limonite	4.4	H 0.018201, O 0.412586, Mg 0.003018, Al 0.006797, Si 0.018281, S 0.001144, Ca 0.055818, Mn 0.010681, Fe 0.473474

96	Concrete, Limonite and steel	4.54	H 0.209628, O 0.301635, Mg 0.001964, Al 0.007326, Si 0.016261, K 0.000698, Ca 0.04439, V 0.000535, Fe 0.417563
97	Concrete, Luminite-Portland-colemanite-baryte	3.1	H 0.248273, B 0.021458, O 0.525816, Na 0.001089, Mg 0.001872, Al 0.01113, Si 0.012069, S 0.063635, Ca 0.043537, Ti 0.000337, Mn 0.000166, Fe 0.007618, Ba 0.063
98	Concrete, Luminite-colemanite-baryte	3.1	H 0.247673, B 0.018639, O 0.528948, Na 0.001096, Mg 0.001319, Al 0.01494, Si 0.007829, S 0.06471, Ca 0.031317, Ti 0.006077, Mn 0.0005, Fe 0.012591, Ba 0.064361
99	Concrete, M-1	4.5	H 0.244682, B 0.025659, O 0.206177, Mg 0.054541, Cl 0.018262, Mn 0.001683, Ca 0.008462, Fe 0.440534
100	Concrete, MO	5.5	H 0.187553, O 0.141792, Mg 0.057557, Mn 0.002753, Cl 0.013865, Fe 0.59648
101	Concrete, Magnetite	3.53	H 0.082377, O 0.551, Mg 0.010248, Al 0.023218, Si 0.024456, S 0.001177, Ca 0.047269, Ti 0.030274,

			V 0.00163, Cr 0.000871, Mn 0.000962, Fe 0.226518
102	Concrete, Magnetite and steel	4.64	H 0.086074, O 0.314486, Mg 0.005517, Al 0.01403, Si 0.020499, Ca 0.050769, Ti 0.012192, V 0.000464, Fe 0.495969
103	Concrete, Magnuson's	2.147	H 0.059731, C 0.159072, O 0.566259, Na 0.001113, Mg 0.070306, Al 0.005284, Si 0.027193, S 0.001405, Cl 0.000268, K 0.004382, Ca 0.102437, Ti 0.000564, Mn 0.000169, Fe 0.001817
104	Concrete, Oak Ridge (ORNL)	2.2994	H 0.106687, C 0.253543, O 0.445629, Na 0.000205, Mg 0.023349, Al 0.006977, Si 0.021339, K 0.000506, Ca 0.139343, Fe 0.002423
105	Concrete, Ordinary (NBS 03)	2.35	H 0.149857, C 0.074206, O 0.526836, Mg 0.017713, Al 0.023794, Si 0.091975, S 0.001649, K 0.000773, Ca 0.109681, Fe 0.003516
106	Concrete, Ordinary (NBS 04)	2.35	H 0.103584, O 0.584808, Na 0.013974, Mg 0.001982, Al 0.03185, Si 0.210759, S 0.000752, K 0.009244,

				Ca 0.038876, Fe 0.004171
107	Concrete, Ordinary (NIST)	2.3		H 0.305322, C 0.00288, O 0.50041, Na 0.009212, Mg 0.000725, Al 0.010298, Si 0.151046, K 0.003578, Ca 0.014924, Fe 0.001605
108	Concrete, Portland	2.3		H 0.168753, C 0.001416, O 0.562526, Na 0.011838, Mg 0.0014, Al 0.021354, Si 0.204119, K 0.005656, Ca 0.018674, Fe 0.004264
109	Concrete, Regulatory Concrete (developed for U.S. NRC)	2.3		H 0.168032, O 0.563184, Na 0.021365, Al 0.021343, Si 0.203235, Ca 0.018595, Fe 0.004246
110	Concrete, Rocky Flats	2.321		H 0.133297, C 0.082335, N 0.000256, O 0.542949, Na 0.004909, Mg 0.009213, Al 0.014408, Si 0.098871, S 0.001061, K 0.006277, Ca 0.102809, Ti 0.000374, Fe 0.00324
111	Concrete, Serpentine	2.1		H 0.246195, C 0.001181, O 0.498976, Na 0.002776, Mg 0.086638, Al 0.011036, Si 0.116124, K 0.001632, Ca 0.026536, Cr 0.000273, Fe 0.008633
112	Copper	8.96	Cu	Cu 1

113	Diatomaceous Earth	0.2563		H 0.151307, O 0.581762, Na 0.00733, Mg 0.001944, Al 0.009834, Si 0.239362, K 0.004823, Ca 0.001676, Fe 0.001962
114	Diesel Fuel	0.849	C12H23	H 0.657143, C 0.342857
115	Earth, Typical Western U.S.	1.52		H 0.316843, O 0.501587, Al 0.039952, Si 0.141618
116	Earth, U.S. Average	1.52		O 0.670604, Na 0.005578, Mg 0.011432, Al 0.053073, Si 0.201665, K 0.007653, Ca 0.026664, Ti 0.002009, Mn 0.000272, Fe 0.02105
117	Ethane	0.001253	C2H6	H 0.75, C 0.25
118	Ethyl Acetate	0.901	C4H8O2	H 0.571429, C 0.285714, O 0.142857
119	Ethyl Alcohol	0.7893	C2H6O	H 0.666667, C 0.222222, O 0.111111
120	Ethylene	0.001174	C2H4	H 0.666667, C 0.333333
121	Ethylene Glycol	1.114	C2H6O2	H 0.6, C 0.2, O 0.2
122	Explosive Compound, AN	1.72	N2H4O3	H 0.444444, N 0.222222, O 0.333333
123	Explosive Compound, EGDN	1.49	C2H4N2O6	H 0.285714, C 0.142857, N 0.142857, O 0.428571

124	Explosive Compound, HMX	1.91	C4H8N8O8	H 0.285714, C 0.142857, N 0.285714, O 0.285714
125	Explosive Compound, NC	1.49		H 0.300764, C 0.234387, N 0.089844, O 0.375006
126	Explosive Compound, NG	1.6	C3H5N3O9	H 0.25, C 0.15, N 0.15, O 0.45
127	Explosive Compound, PETN	1.77	C5H8N4O12	H 0.275862, C 0.172414, N 0.137931, O 0.413793
128	Explosive Compound, RDX	1.858	C3H6N6O6	H 0.285714, C 0.142857, N 0.285714, O 0.285714
129	Explosive Compound, TNT	1.654	C7H5N3O6	H 0.238095, C 0.333333, N 0.142857, O 0.285714
130	Eye Lens (ICRP)	1.1		H 0.61832, C 0.10126, N 0.023878, O 0.256542
131	Felt	0.185		H 0.384044, C 0.316908, N 0.11036, O 0.188689
132	Ferric Oxide	5.2	Fe2O3	O 0.6, Fe 0.4
133	Ferrous Sulfate Dosimeter Solution	1.024		H 0.66001, N 1.2e-05, O 0.337475, Na 6e-06, S 0.002485, Cl 6e-06, Fe 6e-06
134	Fertilizer (Muriate of Potash)	1.121		H 0.001833, O 0.001658, Na 0.013642, Mg 0.000313, S 0.000183, Cl 0.49818, K

				0.483783, Ca 0.000254, Br 0.000153
135	Fiberglass, Type C	2.54		B 0.035032, O 0.609971, Na 0.052479, Mg 0.015131, Al 0.01595, Si 0.219923, S 0.000254, Ca 0.050751, Fe 0.000509
136	Fiberglass, Type E	2.57		B 0.043748, O 0.611969, F 0.005345, Na 0.006553, Mg 0.012598, Al 0.055773, Si 0.182532, K 0.004312, Ca 0.074245, Ti 0.001907, Fe 0.001018
137	Fiberglass, Type R	2.52		O 0.628478, Mg 0.030755, Al 0.101309, Si 0.206302, Ca 0.033156
138	Freon-12	1.12	CF2Cl2	C 0.2, F 0.4, Cl 0.4
139	Freon-12B2	1.8	CF2Br2	C 0.2, F 0.4, Br 0.4
140	Freon-13	0.95	CF3Cl	C 0.2, F 0.6, Cl 0.2
141	Freon-13B1	1.5	CF3Br	C 0.2, F 0.6, Br 0.2
142	Freon-13I1	1.8	CF3I	C 0.2, F 0.6, I 0.2
143	GAGG(CE)	6.63	Gd3Al2Ga3O12	Gd 0.15, Al 0.1, Ga 0.15, O 0.6
144	Gadolinium	7.9004	Gd	Gd 1
145	Gadolinium Aluminum Galium Oxide - 0.5 atom% Cerium doped	6.6		Gd 0.14925, Al 0.0995, Ga 0.14925, O 0.597, Ce 0.005

146	Gadolinium Oxysulfide	7.44	Gd2O2S	O 0.4, S 0.2, Gd 0.4
147	Gadolinium Silicate (GSO)	6.71	Gd2SiO5	O 0.625, Si 0.125, Gd 0.25
148	Gafchromic Sensor (GS)	1.3		H 0.558077, C 0.316313, N 0.050235, O 0.075375
149	Gallium Arsenide	5.31	GaAs	Ga 0.5, As 0.5
150	Gasoline	0.721	C8H18	H 0.692308, C 0.307692
151	Germanium, High Purity	5.323	Ge	Ge 1
152	Glass Scintillator, Li Doped (GS1, GS2, GS3)	2.66		Li 0.077858, O 0.581369, Mg 0.115886, Al 0.041992, Si 0.178152, Ce 0.004743
153	Glass Scintillator, Li Doped (GS10, GS20, GS30)	2.5		Li 0.209064, O 0.545546, Mg 0.017287, Al 0.061502, Si 0.162355, Ce 0.004246
154	Glass Scintillator, Li Doped (GSF1, GSF2, and GSF3)	2.42		Li 0.123884, O 0.586222, Na 0.014389, Al 0.0656, Si 0.207188, Ce 0.002717
155	Glass Scintillator, Li Doped (KG1, KG2, KG3)	2.42		Li 0.238376, O 0.546763, Si 0.209674, Ce 0.005187
156	Glass, Borosilicate (Pyrex Glass)	2.23		B 0.070433, O 0.641102, Na 0.023311, Al 0.008204, Si 0.255335, K 0.001615

157	Glass, Foam	0.128		H 0.018717, B 0.02617, O 0.629687, Na 0.132123, Si 0.18742, S 0.005883
158	Glass, Lead	6.22		O 0.592952, Si 0.174595, Ti 0.010251, As 0.002146, Pb 0.220056
159	Glass, Plate	2.4		O 0.603856, Na 0.088144, Si 0.251795, Ca 0.056205
160	Glycerol	1.2613	C3H8O3	H 0.571428, C 0.214286, O 0.214286
161	Gold	19.32	Au	Au 1
162	Gypsum (Plaster of Paris)	2.32	Ca1S1O6H4	H 0.333333, O 0.5, S 0.083333, Ca 0.083333
163	He-3 Proportional Gas	0.000125265	He	He3 1
164	He-4 Gas Detector	0.03	He	He 1
165	Helium, Natural	0.000166322	He	He 1
166	Hydrogen	8.3748e-05	H2	H 1
167	Incoloy Alloy 800	7.94		C 0.002984, Al 0.007663, Si 0.012761, S 0.000167, Ti 0.00432, Cr 0.222684, Mn 0.009785, Fe 0.430102, Ni 0.305305, Cu 0.00423
168	Inconel Alloy 600	8.47		C 0.004618, Si 0.006584, S 0.000172, Cr 0.169596, Mn 0.006731, Fe 0.081501, Ni 0.727888, Cu 0.00291

169	Inconel Alloy 625	8.44		C 0.004852, Al 0.008639, Si 0.010374, P 0.000281, S 0.000272, Ti 0.004869, Cr 0.24338, Mn 0.005303, Fe 0.052167, Co 0.009887, Ni 0.581642, Nb 0.023124, Mo 0.05521
170	Inconel Alloy 718	8.19		B 0.000294, C 0.003498, Al 0.010694, Si 0.006542, P 0.000253, S 0.000245, Ti 0.01085, Cr 0.210869, Mn 0.003344, Fe 0.175669, Ni 0.516178, Co 0.008909, Cu 0.002478, Nb 0.031833, Mo 0.018344
171	Indium	7.31	In	In 1
172	Iron	7.874	Fe	Fe 1
174	Iron Boride (FeB)	7.15	FeB	B 0.5, Fe 0.5
175	Iron, Armco Ingot	7.866		C 0.000556, O 0.003826, P 9e-05, S 0.000434, Mn 0.000172, Fe 0.994922
176	Iron, Cast (gray)	7.15		C 0.137105, Si 0.044837, P 0.004691, S 0.00151, Mn 0.00573, Fe 0.806127
177	Iron, Wrought (Byers No.1)	7.7		C 0.003747, Si 0.003163, P 0.001127, S 0.000175, Mn 0.000154, Fe 0.991634

178	Kaowool	0.096		B 0.000467, O 0.6361, Al 0.179644, Si 0.176545, Ca 0.000363, Ti 0.004332, Fe 0.002549
179	Kapton Polyimide Film	1.42		H 0.256392, C 0.564121, N 0.051282, O 0.128205
180	Kennertium	16.8		Ni 0.191007, Cu 0.294036, W 0.514957
181	Kernite	1.95	Na2B4O11H8	H 0.32, B 0.16, O 0.44, Na 0.08
182	Kerosene	0.819	C14H30	H 0.681818, C 0.318182
183	Krypton	0.00347832	Kr	Kr 1
184	Kynar	1.79	H2C2F2	H 0.333333, C 0.333333, F 0.333333
185	Lanthanum Bromide - 0.5 wt% Cerium doped	5.08		Br 0.755995, Ce 0.001249, La 0.242756
186	Lanthanum Bromide - 10 wt% Cerium and 0.10 wt% Strontium	5.08		Br 0.75, Ce 0.025, La 0.22475, Sr 0.00025
187	Lanthanum Bromide - 10 wt% Cerium doped	5.08		Br 0.748693, Ce 0.024741, La 0.226566
188	Lanthanum Bromide - 5 wt% Cerium doped	5.08		Br 0.748531, Ce 0.012368, La 0.239101
189	Lead	11.35	Pb	Pb 1
190	Lead Iodide	6.16	PbI2	Pb 0.333333, I 0.666667

191	Lead Tungstate (PWO)	8.24	PbWO4	O 0.666666, W 0.166667, Pb 0.166667
192	Lithium	0.534	Li	Li 1
193	Lithium Amide	1.178	LiNH2	H 0.5, Li 0.25, N 0.25
194	Lithium Fluoride	2.635	LiF	Li 0.5, F 0.5
195	Lithium Gadolinium Borate (LGB)	3.5	Li6GdB3O9	Li6 0.315789, B10 0.157895, O 0.473684, Gd 0.052632
196	Lithium Hydride	0.82	LiH	H 0.5, Li 0.5
197	Lithium Iodide (high density)	4.08	LiI	Li 0.5, I 0.5
198	Lithium Iodide (low density)	3.494	LiI	Li 0.5, I 0.5
199	Lithium Oxide	2.013	Li2O	Li 0.666667, O 0.333333
200	Lithium Tetraborate	2.44	Li2B4O7	Li 0.153846, B 0.307692, O 0.538462
201	Lucite	1.19	C5O2H8	H 0.533333, C 0.333333, O 0.133333
202	Lutetium Aluminum Garnet (LuAG)	6.73	Al5Lu3O12	O 0.6, Al 0.25, Lu 0.15
203	Lutetium Iodide	5.6	LuI3	Lu 0.25, I 0.75
204	Lutetium Orthoaluminate (LuAP)	8.4	LuAlO3	O 0.6, Al 0.2, Lu 0.2
205	Lutetium Oxyorthosilicate (LSO)	7.4	Lu2SiO5	O 0.625, Si 0.125, Lu 0.25
206	Lutetium Yttrium OxyorthoSilicate: 0.5			

	atom% Cerium (LYSO)	7.25		O 0.621875, Si 0.124375, Y 0.012438, Lu 0.236313, Ce 0.005
207	Magnesium	1.74	Mg	Mg 1
208	Magnesium Oxide	3.58	MgO	O 0.5, Mg 0.5
209	Magnesium Tetraborate	2.53	MgB4O7	B 0.333333, O 0.583334, Mg 0.083333
210	Masonite	1.3	C6H10O5	H 0.47619, C 0.285714, O 0.238095
211	Melamine	1.573	C3H6N6	H 0.4, C 0.2, N 0.4
212	Melamine Formaldehyde	1.35	C5H7N6	H 0.388889, C 0.277778, N 0.333333
213	Mercuric Iodide	6.36	HgI2	I 0.666667, Hg 0.333333
214	Mercury	13.546	Hg	Hg 1
215	Methane	0.000667151	CH4	H 0.8, C 0.2
216	Methanol	0.7914	CH4O	H 0.666666, C 0.166667, O 0.166667
217	Methylene Chloride	1.3266	CH2Cl2	H 0.4, C 0.2, Cl 0.4
218	Molybdenum	10.22	Mo	Mo 1
219	Monosodium Titanate, MST	1	NaTi2O5H	H 0.111111, O 0.555556, Na 0.111111, Ti 0.222222

220	Mortar	1.97		Al 0.176573, Ca 0.001118, O 0.625073, Fe 0.004185, K 0.003991, Na 0.018197, Si 0.167202, Ti 0.003661
221	Muscle Equivalent-Liquid, with sucrose	1.11		H 0.619257, C 0.082644, N 0.016082, O 0.282017
222	Muscle Equivalent-Liquid, without sucrose	1.07		H 0.631874, C 0.062437, N 0.015809, O 0.28988
223	Muscle, Skeletal	1.04		H 0.630924, C 0.056734, N 0.012488, O 0.298113, Na 0.000206, Mg 4.9e-05, P 0.000367, S 0.000475, Cl 0.000141, K 0.000488, Ca 5e-06, Fe 5e-06, Zn 5e-06
224	Muscle, Striated	1.04		H 0.633092, C 0.064072, N 0.015634, O 0.285073, Na 0.000218, Mg 5.1e-05, P 0.000404, S 0.000976, K 0.00048
225	NE-213 Equivalent	0.874	C8H10	C 0.444444, H 0.555556
226	Neon	0.000838505	Ne	Ne 1
227	Nickel	8.902	Ni	Ni 1
228	Niobium	8.57	Nb	Nb 1
229	Nitrogen	0.00116528	N2	N 1
230	Nylon, Dupont ELVAmide 8062	1.08		H 0.593353, C 0.311942, N 0.041061,

				O 0.053644
231	Nylon, Type 11 (Rilsan)	1.425	C11H21NO	H 0.617647, C 0.323529, N 0.029412, O 0.029412
232	Nylon, Type 6 and Type 6/6	1.14	C6H11NO	H 0.578947, C 0.315789, N 0.052632, O 0.052632
233	Nylon, Type 6/10	1.14	C8H15NO	H 0.6, C 0.32, N 0.04, O 0.04
234	Oil, Crude (Heavy, Cold Lake, Canada)	0.97		H 0.588874, C 0.397741, N 0.00163, O 0.003924, S 0.007831
235	Oil, Crude (Heavy, Mexican)	0.975		H 0.587741, C 0.40476, S 0.007499
236	Oil, Crude (Heavy, Qayarah, Iraq)	0.97		H 0.590036, C 0.391776, N 0.002914, S 0.015274
237	Oil, Crude (Light, Texas)	0.875		H 0.629377, C 0.36523, N 0.002578, S 0.002815
238	Oil, Fuel (Calif.)	0.955		H 0.633771, C 0.364359, S 0.00187
239	Oil, Hydraulic	0.871	C40H33O4Cl6P	H 0.392857, C 0.47619, O 0.047619, P 0.011905, Cl 0.071429
240	Oil, Lard	0.915	C10H18O	H 0.620689, C 0.344828, O 0.034483
241	Oxygen	0.00133151	O2	O 1
242	P-10 gas	0.001561		H 0.285714, C 0.071429, Ar 0.642857

243	P-5 gas	0.001611		H 0.166667, C 0.041667, Ar 0.791666
244	P-terphenyl	1.28	C18H14	H 0.4375, C 0.5625
245	Palladium	12.02	Pd	Pd 1
246	Paper, News print	0.65	C6H10O5	H 0.47619, C 0.285714, O 0.238095
247	Paper, glossy	1.135	C6H10O5	H 0.47619, C 0.285714, O 0.238095
248	Paper, printer	0.69	C6H10O5	H 0.47619, C 0.285714, O 0.238095
249	Photographic Emulsion, Gel in	1.2914		H 0.546943, C 0.235252, N 0.053934, O 0.161567, S 0.002304
250	Photographic Emulsion, Kodak Type AA	2.2		H 0.440284, C 0.25526, N 0.074899, O 0.148422, Br 0.040572, Ag 0.040562
251	Photographic Emulsion, Standard Nuclear	3.815		H 0.407073, C 0.175082, N 0.040139, O 0.120229, S 0.001715, Br 0.127142, Ag 0.127904, I 0.000715
252	Platinum	21.45	Pt	Pt 1
253	Plutonium Bromide	6.75	PuBr3	Br 0.75, Pu238 0.000126, Pu239 0.233817, Pu240 0.014942, Pu241 0.000992, Pu242 0.000123
254	Plutonium Carbide	13.6	PuC	C 0.5, Pu238 0.000251, Pu239 0.467635,

				Pu240 0.029883, Pu241 0.001984, Pu242 0.000247
255	Plutonium Chloride	5.71	PuCl3	Cl 0.75, Pu238 0.000126, Pu239 0.233817, Pu240 0.014942, Pu241 0.000992, Pu242 0.000123
256	Plutonium Dioxide	11.46	PuO2	O 0.666667, Pu238 0.000167, Pu239 0.311756, Pu240 0.019922, Pu241 0.001323, Pu242 0.000165
257	Plutonium Fluoride (PuF3)	9.33	PuF3	F 0.75, Pu238 0.000126, Pu239 0.233817, Pu240 0.014942, Pu241 0.000992, Pu242 0.000123
258	Plutonium Fluoride (PuF4)	7.1	PuF4	F 0.8, Pu238 0.0001, Pu239 0.187054, Pu240 0.011953, Pu241 0.000794, Pu242 9.9e-05
259	Plutonium Fluoride (PuF6)	5.08	PuF6	F 0.857143, Pu238 7.2e-05, Pu239 0.13361, Pu240 0.008538, Pu241 0.000567, Pu242 7.1e-05
260	Plutonium Iodide	6.92	PuI3	I 0.75, Pu238 0.000126, Pu239 0.233817, Pu240 0.014942, Pu241 0.000992, Pu242 0.000123

261	Plutonium Nitrate	2.447	PuN4O12	N 0.235294, O 0.705882, Pu238 3e-05, Pu239 0.055016, Pu240 0.003516, Pu241 0.000233, Pu242 2.9e-05
262	Plutonium Nitride	14.25	PuN	N 0.5, Pu238 0.000251, Pu239 0.467635, Pu240 0.029883, Pu241 0.001984, Pu242 0.000247
263	Plutonium Oxide (Pu2O3)	10.5	Pu2O3	O 0.6, Pu238 0.000201, Pu239 0.374108, Pu240 0.023907, Pu241 0.001587, Pu242 0.000198
264	Plutonium Oxide (PuO)	14	PuO	O 0.5, Pu238 0.000251, Pu239 0.467635, Pu240 0.029883, Pu241 0.001984, Pu242 0.000247
265	Plutonium, Aged WGPu (A: 4-7% Pu240)	19.84		Pu238 0.0001, Pu239 0.936557, Pu240 0.059677, Pu241 0.001981, Pu242 0.000296, Am241 0.001389
266	Plutonium, Aged WGPu (B: 10-13% Pu240)	19.84		Pu238 0.000897, Pu239 0.862406, Pu240 0.116661, Pu241 0.009837, Pu242 0.001665, Am241 0.008534
267	Plutonium, Aged WGPu (C: 16-19% Pu240)	19.84		Pu238 0.002345, Pu239 0.784809, Pu240 0.164523, Pu241 0.013894, Pu242 0.00663, Am241 0.027798

268	Plutonium, DOE 3013 WGPu	19.84	Pu	Pu238 0.000502, Pu239 0.935269, Pu240 0.059767, Pu241 0.003968, Pu242 0.000494
269	Plutonium, Fuel Grade	19.84	Pu	Pu238 0.001005, Pu239 0.861564, Pu240 0.119578, Pu241 0.015877, Pu242 0.001976
270	Plutonium, Power Grade	19.84	Pu	Pu238 0.009964, Pu239 0.62519, Pu240 0.217375, Pu241 0.118075, Pu242 0.029396
271	Plutonium, Shefelbine WGPu	19.84		Pu238 0.000301, Pu239 0.939451, Pu240 0.056777, Pu241 0.002976, Pu242 0.000296, Am241 0.000198
272	Polycarbonate	1.2	C16H14O3	H 0.424242, C 0.484848, O 0.090909
273	Polyethylene Terephthalate (PET)	1.38	C10H8O4	H 0.363636, C 0.454545, O 0.181818
274	Polyethylene, Borated	1		H 0.627756, B 0.04668, C 0.325564
275	Polyethylene, Non-borated	0.93	C2H4	H 0.666667, C 0.333333
276	Polyisocyanurate (PIR)	0.0482	C15H10N2O2	H 0.344828, C 0.51724, N 0.068966, O 0.068966
277	Polypropylene (PP)	0.9	C3H6	H 0.666667, C 0.333333

278	Polystyrene (PS)	1.06	C8H8	H 0.5, C 0.5
279	Polytetrafluoroethylene (PTFE)	2.25	C2F4	C 0.333333, F 0.666667
280	Polyurethane Foam (PUR)	0.021		H 0.360023, C 0.400878, N 0.076459, O 0.162639
281	Polyvinyl Acetate (PVA)	1.19	C4H6O2	H 0.5, C 0.333333, O 0.166667
282	Polyvinyl Chloride (PVC)	1.406	C2H3Cl	H 0.5, C 0.333333, Cl 0.166667
283	Polyvinyl Toluene (PVT)	1.032	C9H10	H 0.526316, C 0.473684
284	Polyvinylidene Chloride (PVDC)	1.7	C2H2Cl2	H 0.333333, C 0.333333, Cl 0.333333
285	Potassium Aluminum Silicate	1.1	KAlSi3O8	O 0.615385, Al 0.076923, Si 0.230769, K 0.076923
286	Potassium Iodide	3.13	KI	K 0.5, I 0.5
287	Potassium Oxide	2.32	K2O	O 0.333333, K 0.666667
288	Propane (gas)	0.00187939	C3H8	H 0.727273, C 0.272727
289	Propane (liquid)	0.43	C3H8	H 0.727273, C 0.272727
290	Quartz	2.62	SiO2	O 0.666667, Si 0.333333
291	Quartz Glass	2.2	SiO2	O 0.666667, Si 0.333333

292	Radiochromic Dye Film, Nylon Base (RDF: NB)	1.08		H 0.589064, C 0.317179, N 0.04111, O 0.052647
293	Rayon	1.16	C6H10O5	H 0.47619, C 0.285714, O 0.238095
294	Rock (Average of 5 types)	2.662		H 0.027052, C 0.098551, O 0.561516, Na 0.012119, Mg 0.027079, Al 0.043135, Si 0.147747, P 0.000307, S 0.005618, K 0.008671, Ca 0.054072, Ti 0.001514, Mn 0.000165, Fe 0.012454
295	Rock, Basalt	3.01		O 0.629838, Na 0.024716, Mg 0.003656, Al 0.058713, Si 0.24888, P 0.000351, K 0.018158, Ca 0.006737, Ti 0.00078, Mn 0.000146, Fe 0.008025
296	Rock, Granite	2.69		O 0.607744, Na 0.020743, Mg 0.037864, Al 0.068362, Si 0.182167, P 0.001047, K 0.005014, Ca 0.03782, Ti 0.005119, Mn 0.000616, Fe 0.033504
297	Rock, Limestone	2.6		C 0.190012, O 0.603322, Si 0.016659, Ca 0.190007
298	Rock, Sandstone	2.37		C 0.00148, O 0.654711, Na 0.006472, Mg 0.001355, Al 0.015468, Si 0.30491, P 3e-06, K 0.007027, Ca 0.004532, Ti 8.5e-05, Fe 0.003956, Mn 1e-06
299	Rock, Shale	2.6		H 0.029418, C 0.031447, O 0.607866, Na 0.003719, Mg 0.004942, Al 0.053784, Si 0.225978, P 4.1e-05, S 0.00611, K 0.008792, Ca

				0.014705, Ti 0.001096, Fe 0.012089, Mn 1.3e-05
300	Rubber, Butyl	0.92	C4H8	H 0.666667, C 0.333333
301	Rubber, Natural	0.92	C5H8	H 0.615385, C 0.384615
302	Rubber, Neoprene	1.23	C4H5Cl	H 0.5, C 0.4, Cl 0.1
303	Rubber, Silicon	1.0185		H 0.597028, C 0.199364, O 0.104171, Si 0.099437
304	Salt Water (T=0 C)	1.20986		H 0.621151, O 0.310588, Na 0.03413, Cl 0.034131
305	Salt Water (T=20 C)	1.02239		H 0.661897, O 0.330961, Na 0.003571, Cl 0.003571
306	Sand	1.7		H 0.135402, C 0.004874, O 0.583891, Na 0.012932, Al 0.022215, Si 0.226487, K 0.005179, Ca 0.004874, Fe 0.004146
307	Sea Water, Simple Artificial	1.02334		H 0.661583, O 0.331501, Na 0.002897, Mg 0.000326, S 0.000174, Cl 0.003392, K 6.3e-05, Ca 6.4e-05
308	Sea Water, Standard	1.02334		H 0.661559, B 3e-06, C 1.2e-05, O 0.331526, Na 0.002897, Mg 0.000326, S 0.000174, Cl 0.003371, K 6.3e-05, Ca 6.3e-05, Br 5e-06, Sr 1e-06
309	Sepiolite	2.14	Mg4Si6O23H14	H 0.297872, O 0.489362, Mg 0.085106, Si 0.12766
310	Silciate Yttrium - 0.5 atom% Cerium	4.5		Y 0.142173, S 0.214332, O 0.642995, Ce 0.0005
311	Silicon	2.33	Si	Si 1

312	Silicon Carbide (hexagonal)	3.21	SiC	C 0.5, Si 0.5
313	Silicon Dioxide (Alpha-quartz)	2.648	SiO2	O 0.666667, Si 0.333333
314	Silicon Dioxide (Silica)	2.32	SiO2	O 0.666667, Si 0.333333
315	Silver	10.5	Ag	Ag 1
316	Skin (ICRP)	1.1		H 0.619958, C 0.118062, N 0.020589, O 0.240355, Na 1.9e-05, Mg 1.5e-05, P 6.6e-05, S 0.000308, Cl 0.000468, K 0.000135, Ca 2.3e-05, Fe 1e-06, Zn 1e-06
317	Sodium	0.971	Na	Na 1
318	Sodium Bismuth Tungstate (NBWO)	7.57	NaBiW2O8	O 0.666667, Na 0.083333, W 0.166667, Bi 0.083333
319	Sodium Chloride	2.17	NaCl	Na 0.5, Cl 0.5
320	Sodium Iodide - 0.2 wt% Thalium Doped	3.667		Na 0.4995, I 0.499501, Tl 0.000999
321	Sodium Iodide with 8 wt% Lithium - 0.10 wt% Thalium doped	3.66		I 0.474211, Na 0.438678, Li6 0.028452, Li7 0.05822, Tl 0.000439
322	Sodium Nitrate	2.261	NaNO3	N 0.2, O 0.6, Na 0.2
323	Sodium Oxide	2.27	Na2O	O 0.333333, Na 0.666667
324	Steel, Boron Stainless	7.87		Cr 0.196075, Ni 0.118895, B 0.056685, C 0.002181, N 0.000374, P 3.4e-05, S 4.9e-05, Co 0.000267, Si 0.012871, Mn 0.018214, Fe 0.594355

325	Steel, HT9 Stainless	7.874	C 0.009637, N 0.000236, Al 0.000613, Si 0.004121, P 0.000142, S 5.2e-05, Ti 0.000115, V 0.00357, Cr 0.125077, Mn 0.005016, Fe 0.839996, Ni 0.004789, Mo 0.005916, W 0.00072
326	Steel, High Carbon (1095)	7.872	C 0.046122, Mn 0.004895, P 0.000695, S 0.000839, Fe 0.947449
327	Steel, Low Carbon (1008)	7.872	C 0.004629, Mn 0.00506, P 0.000718, S 0.000867, Fe 0.988726
328	Steel, Medium Carbon (1045)	7.872	C 0.022813, Mn 0.008977, P 0.000708, S 0.000854, Fe 0.966648
329	Steel, Stainless 202	7.86	C 0.006733, Mn 0.085864, P 0.001044, S 0.000504, Si 0.019196, Cr 0.186629, Ni 0.045926, N 0.009622, Fe 0.644482
330	Steel, Stainless 302	7.86	C 0.006783, Mn 0.019772, P 0.000789, S 0.000508, Si 0.019339, Cr 0.188019, Ni 0.083282, N 0.003878, Fe 0.67763
331	Steel, Stainless 304	8.03	C 0.003635, Mn 0.01987, P 0.000793, S 0.000511, Si 0.019434, Cr 0.199443, Ni 0.088343, Fe 0.667971
332	Steel, Stainless 304L	8	C 0.003635, Mn 0.01987, P 0.000793, S 0.000511, Si 0.019434, Cr 0.199443, Ni 0.088343, Fe 0.667971
333	Steel, Stainless 316	8	C 0.003683, Mn 0.020128, P 0.000803, S 0.000517, Si 0.019687, Cr 0.180771, Ni

				0.113043, Mo 0.014406, Fe 0.646962
334	Steel, Stainless 316L	8		C 0.001384, Mn 0.020165, P 0.000805, S 0.000518, Si 0.019722, Cr 0.181098, Ni 0.113247, Mo 0.014432, Fe 0.648629
335	Steel, Stainless 321	8		C 0.00364, Mn 0.019894, P 0.000794, S 0.000511, Si 0.019457, Cr 0.189172, Ni 0.097759, Fe 0.668773
336	Steel, Stainless 347	8		C 0.003653, Mn 0.019968, P 0.000797, S 0.000513, Si 0.01953, Cr 0.189877, Ni 0.098123, Nb 0.005609, Fe 0.66193
337	Steel, Stainless 409	7.8		C 0.001367, Mn 0.009961, P 0.000707, S 0.000341, Si 0.019484, Cr 0.116819, Ni 0.004662, N 0.001172, Ti 0.004916, Nb 0.001001, Fe 0.83957
338	Steel, Stainless 440A	7.8		C 0.030043, Mn 0.00973, P 0.00069, S 0.0005, Si 0.019034, Cr 0.174775, Mo 0.004178, Fe 0.76105
339	Steel, Stainless 440B	7.8		C 0.037602, Mn 0.009671, P 0.000686, S 0.000497, Si 0.018918, Cr 0.173713, Mo 0.004153, Fe 0.75476
340	Steel, Stainless 440C	7.8		C 0.047087, Mn 0.009576, P 0.000679, S 0.000492, Si 0.018732, Cr 0.172001, Mo 0.004112, Fe 0.747321
341	Sterotex	0.862	C57O6H110	H 0.635838, C 0.32948, O 0.034682

342	Stilbene (trans-stilbene isomer)	0.9707	C14H12	H 0.461538, C 0.538462
343	Strontium Iodide - 2.5 atom% Europium doped	4.59		Sr 0.330508, I 0.661017, Eu 0.008475
344	Sulfuric acid	1.84	H2SO4	H 0.285714, S 0.142857, O 0.571429
345	Sulphur	2	S	S 1
346	TLYC	4.58	Tl2LiYCl6	Tl 0.2, Li 0.1, Y 0.1, Cl 0.6
347	Tantalum	16.654	Ta	Ta 1
348	Thorium	11.72	Th	Th 1
349	Thorium Dioxide	9.7	ThO2	O 0.666667, Th 0.333333
350	Tin	7.31	Sn	Sn 1
351	Tissue Equivalent, MS20	1		H 0.546349, C 0.329488, N 0.008619, O 0.079014, Mg 0.036358, Cl 0.000172
352	Tissue Equivalent-Gas, methane based (TEG: MB)	0.001064		H 0.60524, C 0.22746, N 0.015038, O 0.152262
353	Tissue Equivalent-Gas, propane based (TEG: PB)	0.001826		H 0.598943, C 0.278538, N 0.014702, O 0.107817
354	Tissue, Adipose (ICRP)	0.92		H 0.634632, C 0.284071, N 0.003047, O 0.077749, Na 0.000116, Mg 4e-06, P 2.8e-05, S 0.000122, Cl 0.00018, K 4.4e-05, Ca 3e-06, Fe 2e-06, Zn 2e-06
355	Tissue, Breast	1.02		H 0.625772, C 0.164488, N 0.012745, O 0.196005, Na

				0.000259, P 0.000192, S 0.000371, Cl 0.000168
356	Tissue, Lung (ICRP)	1.05		H 0.633127, C 0.053676, N 0.012889, O 0.298167, Na 0.000504, Mg 0.000189, P 0.000163, S 0.000442, Cl 0.000473, K 0.000313, Ca 1.4e-05, Fe 4.2e-05, Zn 1e-06
357	Tissue, Ovary	1.05		H 0.643223, C 0.047812, N 0.01058, O 0.2964, Na 0.000537, P 0.000399, S 0.000385, Cl 0.000348, K 0.000316
358	Tissue, Soft (ICRP)	1		H 0.630445, C 0.117591, N 0.010805, O 0.239606, Na 0.000299, Mg 3.3e-05, P 0.000261, S 0.000377, Cl 0.00023, K 0.00031, Ca 3.5e-05, Fe 5e-06, Zn 3e-06
359	Tissue, Soft (ICRU four component)	1		H 0.630928, C 0.058093, N 0.011668, O 0.299311
360	Tissue, Testes (ICRP)	1.04		H 0.641353, C 0.047678, N 0.008835, O 0.300187, Na 0.00061, Mg 2.8e-05, P 0.00025, S 0.000283, Cl 0.000427, K 0.00033, Ca 1.5e-05, Fe 2e-06, Zn 2e-06
361	Tissue, Testis (ICRU)	1.04		H 0.64517, C 0.050569, N 0.00876, O 0.293726, Na 0.000534, P 0.000198, S 0.000383, Cl 0.000346, K 0.000314
362	Titanium	4.54	Ti	Ti 1
363	Titanium Dioxide	4.26	TiO2	O 0.666667, Ti 0.333333
364	Titanium Hydride	3.75	TiH2	H 0.666667, Ti 0.333333

365	Titanium alloy, grade 5	4.43		C 0.003011, O 0.00565, N 0.001614, H 0.006727, Fe 0.003238, Al 0.10261, V 0.035493, Ti 0.841657
366	Toluene	0.8669	C7H8	H 0.533333, C 0.466667
367	Tributyl Borate	0.8567	BO3C12H27	H 0.627907, B 0.023256, C 0.27907, O 0.069767
368	Tributyl Phosphate (TBP)	0.978	C12H27PO4	H 0.613637, C 0.272727, O 0.090909, P 0.022727
369	Tungsten	19.3	W	W 1
370	Uranium Carbide	13.63	UC	C 0.5, U234 0.000136, U235 0.015186, U236 7e-05, U238 0.484609
371	Uranium Dicarbide	11.28	UC2	C 0.666667, U234 9e-05, U235 0.010124, U236 4.6e-05, U238 0.323072
372	Uranium Dioxide	10.96	UO2	O 0.666667, U234 9e-05, U235 0.010124, U236 4.6e-05, U238 0.323072
373	Uranium Hexafluoride	5.09	UF6	F 0.857143, U234 3.9e-05, U235 0.004339, U236 2e-05, U238 0.13846
374	Uranium Hydride	11.1	UH3	H 0.75, U234 6.8e-05, U235 0.007593, U236 3.5e-05, U238 0.242304
375	Uranium Nitride	14.31	UN	N 0.5, U234 0.000136, U235 0.015186, U236 7e-05, U238 0.484609
376	Uranium Oxide	8.38	U3O8	O 0.727273, U234 7.4e-05, U235 0.008283, U236 3.8e-05, U238 0.264332

377	Uranium Tetrafluoride	6.7	UF4	F 0.8, U234 5.4e-05, U235 0.006074, U236 2.8e-05, U238 0.193843
378	Uranium Trioxide	7.29	UO3	O 0.75, U234 6.8e-05, U235 0.007593, U236 3.5e-05, U238 0.242304
379	Uranium, Depleted, Typical	18.95	U	U234 5e-06, U235 0.002532, U238 0.997463
380	Uranium, Enriched, Typical Commercial	18.944	U	U234 0.00031, U235 0.029967, U238 0.969723
381	Uranium, HEU, Health Physics Society	18.724	U	U234 0.010582, U235 0.932362, U236 0.002053, U238 0.055003
382	Uranium, HEU, Russian Average	18.732	U	U234 0.009722, U235 0.898982, U236 0.003798, U238 0.087498
383	Uranium, HEU, US Average	18.724	U	U234 0.009849, U235 0.932166, U236 0.004484, U238 0.053501
384	Uranium, Low Enriched (LEU)	18.944	U	U234 0.000271, U235 0.030372, U236 0.000139, U238 0.969217
385	Uranium, Natural (NU)	18.95	U	U 1
386	Uranium-Plutonium, Mixed Oxide (MOX)	11		O 0.666667, Pu238 0.00039, Pu239 0.008487, Pu240 0.004032, Pu241 0.001462, Pu242 0.001103, U234 4e-06, U235 0.000805, U238 0.317052
387	Uranyl Fluoride	6.37	UO2F2	O 0.4, F 0.4, U234 5.4e-05, U235 0.006074, U236 2.8e-05, U238 0.193843

388	Uranyl Nitrate	2.81	UN2O8	N 0.181818, O 0.727273, U234 2.5e-05, U235 0.002761, U236 1.3e-05, U238 0.088111
389	Vermiculite, Exfoliated	0.085		H 0.197174, O 0.520978, Mg 0.092157, Al 0.039305, Si 0.11341, K 0.009307, Ca 0.006852, Ti 0.003457, Fe 0.017361
390	Viton Fluoroelastomer	1.8		H 0.133322, C 0.333345, F 0.533333
391	Water, Heavy	1.1044	H2O	H2 0.666667, O 0.333333
392	Water, Liquid	0.997	H2O	H 0.666667, O 0.333333
393	Water, Vapor	0.000756182	H2O	H 0.666667, O 0.333333
394	Wax, M3	1.05		H 0.632195, C 0.304374, O 0.032117, Mg 0.030913, Ca 0.000401
395	Wax, Mix D	0.99		H 0.658871, C 0.320928, O 0.010845, Mg 0.007867, Ti 0.001489
396	Wax, Paraffin	0.812	C25H52	H 0.675325, C 0.324675
397	Wood (Southern Pine)	0.64		H 0.462413, C 0.323396, N 0.002773, O 0.208782, Mg 0.000639, S 0.001211, K 0.000397, Ca 0.000388
398	Xenon	0.00548536	Xe	Xe 1
399	Yttrium Aluminum Oxide - 1 atom% Cerium	4.56		O 0.4, Al 0.5, Y 0.1

400	Yttrium Aluminum Perovslite - 0.5 atom% Cerium	5.4		Y 0.199, Al 0.597, O 0.199, Ce 0.005
401	Zeolite (Natrolite)	2.25	Na2Al2Si3O12H4	H 0.173913, O 0.521738, Na 0.086957, Al 0.086957, Si 0.130435
402	Zinc	7.133	Zn	Zn 1
403	Zinc Selenide	5.27	ZnSe	Zn 0.5, Se 0.5
404	Zinc Sulfide	4.09	ZnS	S 0.5, Zn 0.5
405	Zircaloy-2	6.56		O 0.006797, Cr 0.001742, Fe 0.001622, Ni 0.000772, Zr 0.978381, Sn 0.010686
406	Zircaloy-4	6.56		O 0.00679, Cr 0.001741, Fe 0.003242, Zr 0.97755, Sn 0.010677
407	Zirconium	6.52	Zr	Zr 1
408	Zirconium Hydride (Zr5H8)	5.61	Zr5H8	H 0.615385, Zr 0.384615
409	Zirconium Hydride (ZrH2)	5.6	ZrH2	H 0.666667, Zr 0.333333
410	ZnS(Ag):LiF 95wt% 6Li PHOSPHOR POWDER Neutron Detectors	4.09		Zn 0.439541, S 0.439541, F 0.060459, Li6 0.057849, Li7 0.00261
411	ZnS:Ag PHOSPHOR POWDER EJ-600 for Neutron Detectors	4.09	ZnS	Zn 0.5, S 0.5

Geometry

CSG Geometry

YAMC uses constructive solid geometry (CSG) to define simulation domains. Surfaces partition space into half-spaces, which are combined with boolean operations to form regions, cells, and ultimately a geometry.

Surfaces

All surfaces take an optional `surface_id` and `boundary` (`"transmission"` or `"vacuum"`).

Planes are infinite planar surfaces:

```
# Axis-aligned planes
xp = yamc.Plane(axis="x", offset=5.0)
yp = yamc.Plane(axis="y", offset=0.0)
zp = yamc.Plane(axis="z", offset=-10.0)

# General plane: ax + by + cz = d
p = yamc.Plane(axis=(1.0, 1.0, 0.0), offset=5.0 / ((1.0**2 + 1.0**2 + 0.0**2) **
0.5))
```

A **sphere** is centered at `(x0, y0, z0)` with radius `r`:

```
sphere = yamc.Sphere(x0=0.0, y0=0.0, z0=0.0, radius=10.0)
```

Cylinders are infinite along their axis:

```
# Axis-aligned cylinders
zc = yamc.Cylinder(axis="z", x0=0.0, y0=0.0, radius=5.0)
xc = yamc.Cylinder(axis="x", y0=0.0, z0=0.0, radius=3.0)
yc = yamc.Cylinder(axis="y", x0=0.0, z0=0.0, radius=3.0)

# General cylinder (arbitrary axis)
cyl = yamc.Cylinder(axis=(1.0, 1.0, 0.0), x0=0.0, y0=0.0, z0=0.0, radius=5.0)
```

For **tori**, `r_major` is the major radius and `r_minor` the minor radius along the symmetry axis; pass `r_minor_2` for a different transverse minor radius (an elliptical cross-section). `axis` is `'x'`, `'y'`, or `'z'` (arbitrary directions are not yet supported):

```
zt = yamc.Torus(axis="z", x0=0.0, y0=0.0, z0=0.0, r_major=10.0, r_minor=3.0)
xt = yamc.Torus(axis="x", r_major=10.0, r_minor=3.0)
yt = yamc.Torus(axis="y", r_major=10.0, r_minor=3.0)
```

Cones are double-sheeted, with the apex at `(x0, y0, z0)` and `opening_angle` the half-opening angle in degrees. Cap with planes to keep one sheet:

```
import math

# Axis-aligned cones
zc = yamc.Cone(axis="z", x0=0.0, y0=0.0, z0=0.0,
opening_angle=math.degrees(math.atan(0.25**0.5)))
xc = yamc.Cone(axis="x", opening_angle=math.degrees(math.atan(1.0**0.5)))
yc = yamc.Cone(axis="y", opening_angle=math.degrees(math.atan(1.0**0.5)))

# General cone (arbitrary axis)
cone = yamc.Cone(axis=(0.0, 1.0, 0.0), x0=0.0, y0=0.0, z0=0.0,
opening_angle=math.degrees(math.atan(0.5**0.5)))
```

The **general quadric** is a catch-all primitive, `a x^2 + b y^2 + c z^2 + d xy + e yz + f xz + g x + h y + j z + k = 0`. The coefficient names skip `i`. It is useful for porting existing models and for surfaces with no dedicated variant. The dedicated variants are faster and give finite bounding boxes, so prefer them when one fits:

```
# A sphere of radius 10 written as a quadric
q = yamc.Quadric(a=1.0, b=1.0, c=1.0, k=-100.0)
```

Plasma and formula surfaces

`FluxSurface` builds a CSG surface from the same tokamak or stellarator shape used by a plasma source. `rho=1` is the plasma boundary; smaller positive values select an interior flux surface. `OffsetSurface` makes a layer at a signed distance in cm, positive outwards:

```
plasma_shape = yamc.sources.TokamakShape(
    major_radius=100.0, minor_radius=20.0, elongation=1.5,
    triangularity=0.2, shafranov_factor=0.0,
)
plasma_boundary = yamc.FluxSurface(plasma_shape, rho=1.0)
wall_boundary = yamc.OffsetSurface(plasma_boundary, distance=2.0)
plasma_region = plasma_boundary.below
first_wall_region = plasma_boundary.above & wall_boundary.below
```

Offsets compose by adding their distances. A large offset can have creases where fronts meet; construction warns rather than silently changing the distance. These are geometric surfaces, not a material or source assignment.

`ImplicitSurface` accepts a formula whose zero is the surface. Its negative side is `.below`; `^` is the formula language's power operator:

```
custom_surface = yamc.ImplicitSurface("sqrt(x^2 + y^2 + z^2) - 5")
custom_region = custom_surface.below
```

Use the dedicated primitives when they fit: they are faster and provide finite bounds directly. A formula value is not necessarily a signed distance; see the API reference for the permitted functions and domain restrictions.

Boundary types are set on the outermost surfaces:

- `"transmission"` (default): particles pass through freely
- `"vacuum"`: particles are killed on crossing

```
outer = yamc.Sphere(radius=50.0, boundary="vacuum")
```

Halfspaces and regions

Surfaces divide space into two halfspaces. The `.below` property gives the inside/below side (negative sense), `.above` gives the outside/above side (positive sense).

```
sphere = yamc.Sphere(radius=10.0)
inside = sphere.below   # r < 10
outside = sphere.above  # r > 10
```

Boolean operators

Regions are combined with `&` (intersection), `|` (union), and `~` (complement) to build more complicated shapes.

Intersection (`&`) is the space where both regions overlap:

```
# A cylinder capped by two planes gives a finite cylinder
cylinder = yamc.Cylinder(axis="z", radius=5.0)
bottom = yamc.Plane(axis="z", offset=-10.0)
top = yamc.Plane(axis="z", offset=10.0)
capped_cylinder = cylinder.below & bottom.above & top.below
```

Union (`|`) is the space belonging to either region:

```
# Two cylinders forming a cross
cyl_x = yamc.Cylinder(axis="x", radius=2.0)
cyl_z = yamc.Cylinder(axis="z", radius=2.0)
cross = cyl_x.below | cyl_z.below
```

Complement (`~`) is everything outside a region:

```
# Hollow cylinder (pipe)
inner_cyl = yamc.Cylinder(axis="z", radius=3.0)
outer_cyl = yamc.Cylinder(axis="z", radius=5.0)
bottom = yamc.Plane(axis="z", offset=-10.0)
top = yamc.Plane(axis="z", offset=10.0)
pipe = ~inner_cyl.below & outer_cyl.below & bottom.above & top.below
```

Combining them gives, for example, a box with a cylindrical hole:

```
# Box
xmin, xmax = yamc.Plane(axis="x", offset=-8.0), yamc.Plane(axis="x", offset=8.0)
ymin, ymax = yamc.Plane(axis="y", offset=-8.0), yamc.Plane(axis="y", offset=8.0)
zmin, zmax = yamc.Plane(axis="z", offset=-8.0), yamc.Plane(axis="z", offset=8.0)
box = xmin.above & xmax.below & ymin.above & ymax.below & zmin.above & zmax.below

# Cylindrical hole through the box
hole = yamc.Cylinder(axis="z", radius=3.0)
box_with_hole = box & ~hole.below  # box minus cylinder interior
```

Cells

A cell combines a region with an optional material fill. Cells with no fill are void.

```
steel = yamc.Material(composition={"Fe": 1.0}, density=7.874, name="steel")
water = yamc.Material(composition={"H2O": 1.0}, density=1.0, name="water")

# Steel pipe filled with water, inside a vacuum box
pipe_cell = yamc.Cell(region=pipe, material=steel, name="pipe")
water_cell = yamc.Cell(region=inner_cyl.below & bottom.above & top.below,
material=water, name="coolant")
void_cell = yamc.Cell(region=box & outer_cyl.above, name="void")
```

Geometry

A `Geometry` is constructed from a list of cells. Cell IDs are auto-assigned if not set. Material IDs are not assigned by the `Geometry`: every `Material` gets one when it is built (the `id=` given, or else a fresh one), and a `Geometry` refuses two different materials with the same ID.

```
geometry = yamc.Geometry([pipe_cell, water_cell, void_cell])
```

Point lookup and voxel data

Geometry can be inspected without transporting particles or loading nuclear data. Point lookup returns `None` outside all cells:

```python
import numpy as np

point_cell = geometry.find_cell(0.0, 0.0, 0.0)
if point_cell is not None:
    print(point_cell.id, point_cell.name)
voxels = geometry.voxelize(
    lower_left=(-10.0, -10.0, -10.0), spacing=(1.0, 1.0, 1.0),
    shape=(20, 20, 20),
)
cell_grid = np.asarray(voxels.cell_ids).reshape(20, 20, 20)
material_grid = np.asarray(voxels.material_ids).reshape(20, 20, 20)
```

The requested shape is `(nx, ny, nz)`, while flat voxel arrays reshape in `(nz, ny, nx)` C order. A cell ID of `-1` means no cell was found. These are sampled geometry labels, not transport tally values.

Plotting

The interactive plot renders in Jupyter notebooks and can be saved as HTML (with pan/zoom/slice controls) or PNG. `Cell` and `Region` objects also support `.plot()`.

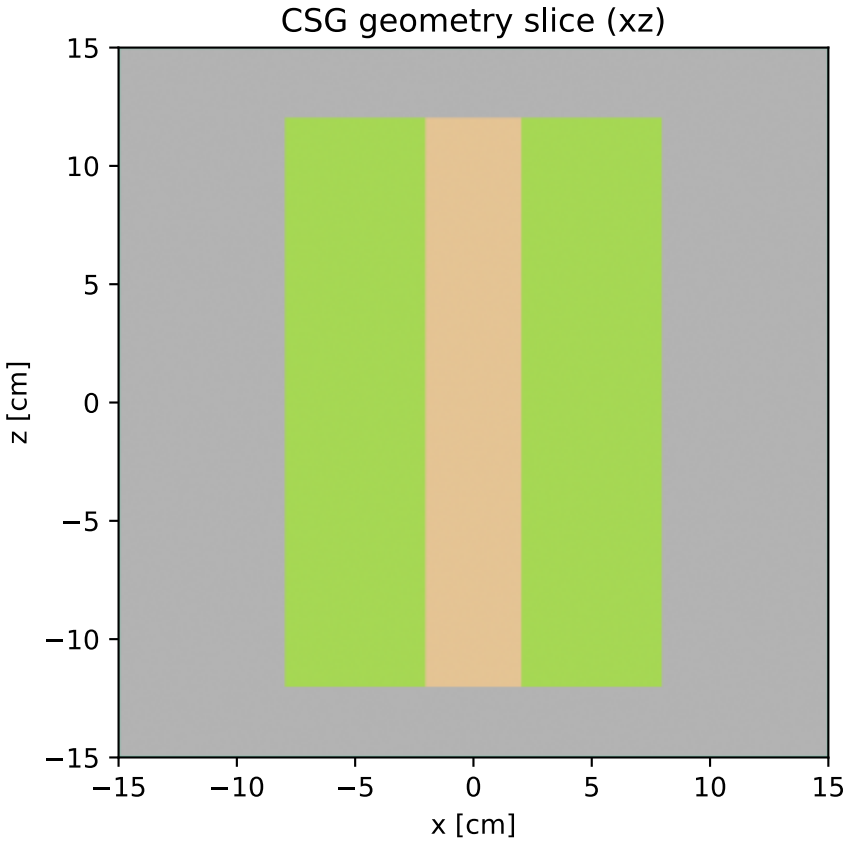
```python
plot = geometry.plot(basis="xz", color_by="material", outline="cell",
resolution=400000)
plot.to_html("geometry.html")   # interactive HTML
plot.to_png("geometry.png")     # static image
png = plot.to_png()             # or the bytes, with no file

# Also works on individual cells and regions
plot = cell.plot(basis="xz", resolution=400000)
plot = region.plot(basis="xz", resolution=400000)
```

The geometry viewer below is rendered live when the docs are built (drag to pan, scroll to zoom):



For manual control with matplotlib, use `sample_slice()` (the figure below is rendered live when the docs are built):

```
import numpy as np
import matplotlib.pyplot as plt
import yamc

# Steel pipe + water coolant inside a vacuum box
steel = yamc.Material(composition={"Fe": 1.0}, density=7.874, name="steel")
water = yamc.Material(composition={"H2O": 1.0}, density=1.0, name="water")
inner_cyl, outer_cyl = yamc.Cylinder(axis="z", radius=2.0), yamc.Cylinder(axis="z",
radius=8.0)
bottom, top = yamc.Plane(axis="z", offset=-12.0), yamc.Plane(axis="z", offset=12.0)
box = (yamc.Plane(axis="x", offset=-15.0, boundary="vacuum").above &
yamc.Plane(axis="x", offset=15.0, boundary="vacuum").below
       & yamc.Plane(axis="y", offset=-15.0, boundary="vacuum").above &
yamc.Plane(axis="y", offset=15.0, boundary="vacuum").below
       & yamc.Plane(axis="z", offset=-15.0, boundary="vacuum").above &
yamc.Plane(axis="z", offset=15.0, boundary="vacuum").below)
pipe = yamc.Cell(region=inner_cyl.above & outer_cyl.below & bottom.above & top.below,
material=steel, name="pipe")
coolant = yamc.Cell(region=inner_cyl.below & bottom.above & top.below,
material=water, name="coolant")
void = yamc.Cell(region=box & ~(outer_cyl.below & bottom.above & top.below),
name="void")
geometry = yamc.Geometry([pipe, coolant, void])

slice_data = geometry.sample_slice(basis="xz", resolution=(400, 400))
cell_ids = np.array(slice_data.cell_ids).reshape(400, 400)

fig, ax = plt.subplots()
ax.imshow(cell_ids, extent=slice_data.extent, origin="lower", cmap="Set2")
ax.set_xlabel("x [cm]")
ax.set_ylabel("z [cm]")
ax.set_title("CSG geometry slice (xz)")

from io import StringIO as _S
_b = _S(); plt.savefig(_b, format="svg", bbox_inches="tight"); print(_b.getvalue())
```

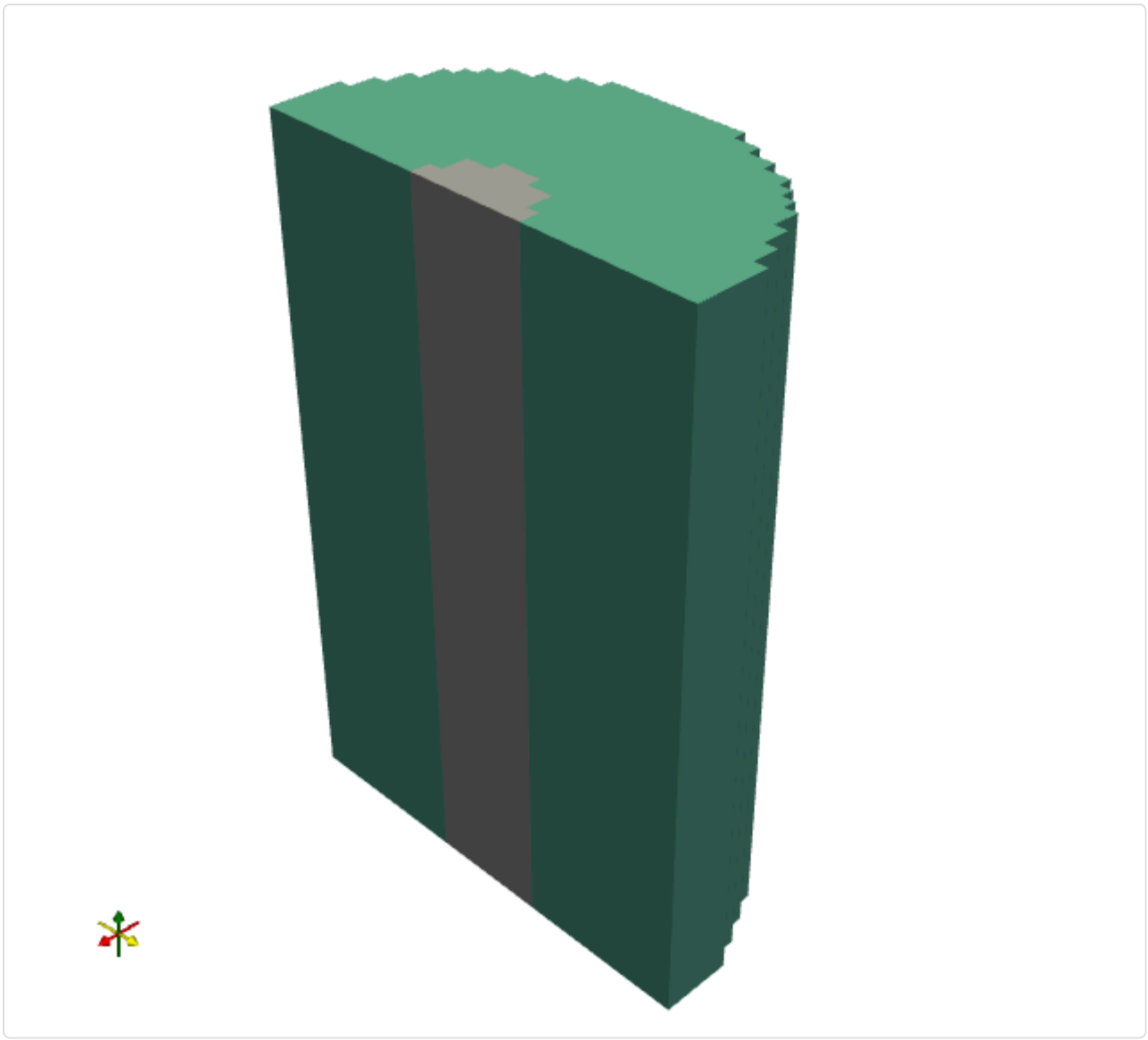


VTK-HDF geometry export

Geometry can be exported to VTK-HDF ImageData files for visualization in ParaView. `Cell` and `Region` objects also support `to_vtkhdf()`.

```
geometry.to_vtkhdf("geometry.vtkhdf", resolution=10000)
cell.to_vtkhdf("cell.vtkhdf")
region.to_vtkhdf("region.vtkhdf")
```

The viewer below is rendered live when these docs are built: it voxelizes the steel-pipe-and-water geometry to `geometry.vtkhdf`, reads it back, and shows it cut in half so the material regions are visible, coloured by `material_id` (drag to rotate, scroll to zoom).



Volume calculation

Stochastic volume calculation samples random points and checks which cell contains each one. Use it to find the volumes of CSG cells.

```
# Single cell
result = cell.calculate_volume(samples=1_000_000)
print(f"Volume: {result.volume:.2f} +/- {result.std_dev:.2f} cm^3")

# All cells in geometry
volumes = geometry.calculate_volume(samples=1_000_000)
for cell_id, result in volumes.items():
    print(f"Cell {cell_id}: {result.volume:.2f} cm^3")
```

Meshed geometries do not need stochastic volume calculation. Each volume's size is computed analytically (via the divergence theorem over its bounding triangles) when the mesh is loaded, and is available as `MeshGeometry.volume_measures`.

```
mesh_geom = yamc.MeshGeometry.from_arrow("geometry.arrow", materials={"fuel": fuel})

for name, vol in zip(mesh_geom.volume_names, mesh_geom.volume_measures):
    print(f"{name}: {vol:.2f} cm^3")
```

For consistency with the CSG API, `MeshGeometry.calculate_volume()` is also available. It returns the same dict-of-VolumeResult shape as `Geometry.calculate_volume()` but with exact values (`std_dev=0`) computed from the analytic measures, so the same calling code works for both CSG and mesh geometries.

```
volumes = mesh_geom.calculate_volume()  # samples= and bounding_box= are ignored
for vol_id, result in volumes.items():
    print(f"Volume {vol_id}: {result.volume:.2f} cm^3 (exact)")
```

CAD Geometry

YAMC meshes CAD itself, with `MeshGeometry.from_cadquery` (install with `pip install "yamc[cad]"`). CadQuery is a Python library for building parametric 3D CAD models with a script-based API. The pipeline lets CadQuery geometry be used directly for neutron transport.

CadQuery can also load externally created STEP files (e.g. exported from another CAD tool) with `cq.importers.importStep("part.step")`. The imported CAD should be clean and watertight so that it meshes correctly and works in particle transport. Avoid gaps, overlaps and self-intersections between solids; they cause meshing failures or lost or double-counted particles.

Defining geometry in CadQuery

```
import cadquery as cq

sphere = cq.Workplane("XY").sphere(10.0)
box = cq.Workplane("XY").box(30, 30, 30).cut(sphere)  # cut, so no point is in two
solids

assembly = cq.Assembly()
assembly.add(sphere, name="fuel", color=cq.Color("red"))
assembly.add(box, name="moderator", color=cq.Color("blue"))
```

Meshing

`MeshGeometry.from_cadquery` imprints the assembly by default, meshes its surfaces with the yamm scene mesher, optionally fills volumes with tetrahedra with the yamm tet mesher, and returns the geometry, ready for transport. The materials are keyed by the assembly's part names.

```python
import yamc

fuel = yamc.Material(composition={"U235": 0.05, "U238": 0.95}, density=19.1)
moderator = yamc.Material(composition={"H2O": 1.0}, density=1.0)
materials = {"fuel": fuel, "moderator": moderator}

# Surface-only mesh, at the default tolerances (0.5 cm and 0.2 rad)
geometry = yamc.MeshGeometry.from_cadquery(assembly, materials)

# Finer facets on curved surfaces
geometry = yamc.MeshGeometry.from_cadquery(
    assembly, materials, tolerance=0.1, angular_tolerance=0.1
)

# Per-solid tolerances, keyed by material tag
geometry = yamc.MeshGeometry.from_cadquery(
    assembly, materials, tolerance={"fuel": 0.01, "moderator": 0.5}
)

# Tets in selected volumes, about 2 cm on a side
geometry = yamc.MeshGeometry.from_cadquery(
    assembly, materials, tet_volumes=["fuel"], target_edge_length=2.0
)

# or tets added to a surface mesh afterwards (a new geometry; this one is unchanged)
geometry = geometry.add_tets(["moderator"], target_edge_length={"moderator": 4.0})
```

The tets of neighbouring volumes meet face to face on the surfaces they share, and their outer faces are exactly the surface triangles transport tracks against. `geometry.mesh_report` says what the mesher did, including what it repaired to make the tets conformal.

The volumes are named after the assembly's parts. `material_tags="assembly_materials"` names them by each part's `cq.Material` instead, and a list gives a tag per solid.

`tolerance` (chordal deflection, cm, default 0.5) and `angular_tolerance` (radians, default 0.2, about 11.5 degrees) each take one number for every solid, or a dict keyed by material tag (every solid with that tag) or by 1-based solid ID. Solids the dict does not name take the default. A surface two solids share is meshed at the finer of their two values, so the mesh stays watertight, and the `from_cadquery` docstring lists the other cases where a solid is meshed finer than it asks. On curved surfaces the angular tolerance is what limits how far the facets sag from the CAD, so it is the one to tighten for a thin curved wall.

Facets lie inside a curved surface, so a plain mesh of a curved solid is slightly smaller than the CAD. The mesher corrects this without adding triangles: it moves the vertices of each curved face and edge off the CAD, by less than the face's tolerance, until each face's mesh encloses the same volume as the CAD face. Shared vertices move once, so an imprinted mesh stays conformal, and planar faces are left as they are. A coarse mesh therefore keeps its solids' volumes, and the facets sit up to one tolerance either side of the CAD surface. Meshed at the defaults, the 10 cm sphere above has a `volume_measures` of 4188.790205 cm^3, against 4188.790205 cm^3 for the exact sphere.

Each solid is meshed from the faces that bound it. A face the solid lies on both sides of (an internal face, such as one left by splitting a solid with a face that stops short of its boundary) or on neither side bounds nothing, so it is left out of the mesh, as it is left out of the solid's CAD volume. The copies two faces hold of a point on their common edge are welded together within a millionth of the model's extent, but two vertices of one face only within a billionth of it, so the two sides of a thin plate stay apart and it keeps its volume. Only a solid thinner than about a billionth of the extent collapses, and the check below refuses it.

`from_cadquery` checks the result against the CAD (`fidelity_check=True`, the default) and raises `MeshFidelityError` for a mesh whose face areas or solid volumes are far from the CAD's, for a solid whose surface is open (an edge its triangles run along more often one way than the other, other than the hairline cracks two facetings of one CAD edge leave), for a solid that has collapsed (its mesh encloses less than 1% of its CAD volume, or has facets lying on each other), or for one whose own surface crosses itself. The last happens when a wall thinner than its facets' sag is meshed at a loose angular tolerance: its two faces cut through each other over a strip, the volume hardly changes, and particles crossing the strip meet the boundary in the wrong order. The error names each such solid, where the crossing is and the angular tolerance to tighten. `fidelity_check=False` skips the checks.

`mesh_threads` (`"all"` by default, or a positive integer) sets how many threads meshing uses, from the face meshing to the overlap checks and tet meshing. The mesh is the same on any number of threads; fewer threads lower the peak memory of a large assembly. `imprint_threads` does the same for the imprint.

Meshing without imprinting

Imprinting merges the surfaces where solids touch, so the mesh is conformal across them. It is also the slow part of meshing a large assembly, and it rejects CAD that will not imprint cleanly. `imprint=False` skips it and meshes every solid on its own:

```
geometry = yamc.MeshGeometry.from_cadquery(
    assembly, materials, imprint=False, contact_tolerance=0.05
)
```

Touching solids then meet along two separately faceted surfaces that may overlap or leave a thin gap. `MeshGeometry` tracks such a mesh discontinuously: a particle leaving one solid is placed in whichever touching solid contains it, instead of being read off a shared surface. Solids closer than `contact_tolerance` count as touching, so an overlap or a gap thinner than it is crossed straight into the next solid, with an overlap staying with the solid the particle was already in. A gap that is in the CAD stays a gap whatever the tolerance: the pair is checked against the CAD and recorded as separated. `contact_tolerance` defaults to twice `tolerance` (with per-solid values, twice the largest), which covers the slivers that two facetings of one CAD surface leave between them; raise it to accept sloppier CAD.

Two solids overlapping deeper than `contact_tolerance`, or without it twice the larger `tolerance` of the two, is an error by default. `overlaps="smaller_wins"` gives each such overlap to the smaller solid of the pair, wherever a particle arrives from, and takes the shared volume off the larger one. `overlaps="ignore"` skips the check.

The mesh file records how it was made, and `MeshGeometry` reports it:

```
print(geometry.discontinuous)      # True for a mesh made without imprinting
print(geometry.contact_tolerance)  # cm; None for a mesh tracked conformally
```

An imprinted mesh is tracked discontinuously too where the imprint left solids touching without a shared surface (two solids that meet only along a line, say, whose facetings then cross near it). Shared surfaces still hand over by topology, and a particle leaving a solid through a surface it does not share is placed in whichever touching solid contains it. Loading says when it does this, and uses the contact tolerance the file records or, without one, 1e-4 of the mesh's diagonal. A mesh from another writer that does not record imprinting is only warned about, since such solids could be a real overlap, which discontinuous tracking would hide; pass `contact_tolerance` to track it discontinuously.

Saving and loading

```
# Arrow IPC, for MeshGeometry.from_arrow (materials are not saved)
geometry.to_arrow("geometry.arrow")
data = geometry.to_arrow()  # no path: the file's bytes, e.g. in Pyodide

# VTKHDF, for ParaView: triangles and tets with each cell's volume_id
geometry.to_vtkhdf("geometry.vtkhdf")
```

A tet mesh from another code

A tet mesh from FEniCSx, gmsh or any other mesher can be the geometry itself. The volumes are the names in `tet_materials`, and their surfaces are built from the faces where the material changes, so a field on those tets (a temperature or a density from a thermal or CFD solve) needs no mapping:

```
geometry = yamc.MeshGeometry.from_tets(
    vertices,        # [[x, y, z], ...] in cm
    tets,            # [[i, j, k, l], ...]
    tet_materials,   # ["breeder", "breeder", ..., "steel", ...]
    {"breeder": breeder, "steel": steel},
    graveyard_offset=10.0,
)
```

MeshGeometry

`MeshGeometry.from_arrow` loads a saved mesh, from a path or bytes, and assigns materials to each volume by name.

```
mesh_geom = yamc.MeshGeometry.from_arrow(
    "geometry.arrow",
    materials={
        "fuel": fuel,
        "moderator": moderator,
    },
)
```

The mesh file is a YAMC Arrow IPC mesh (`.arrow`); other meshers are supported by writing that format. A surface with the same volume on both sides is an internal face of that volume: it bounds nothing, and particles pass through it as if it were not there. The `materials` keys are volume names as `mesh_geom.volume_names` lists them: the part names (or tags) the CAD was meshed with, without a `mat:` prefix. A key that matches no volume raises `ValueError`, which lists the volume names and suggests the nearest one:

```
ValueError: 'geometry.arrow': materials keys match no volume: 'fual'. Volume names
are: 'fuel', 'moderator'. Did you mean 'fuel' for 'fual'?
```

Volumes with no key are void, and are listed in one warning when the mesh is loaded:

```
Warning: 1 named volume(s) have no entry in materials and are void: 'moderator'
```

Optional parameters control the vacuum boundary:

```
mesh_geom = yamc.MeshGeometry.from_arrow(
    "geometry.arrow",
    materials={"fuel": fuel},
    implicit_complement_material=None,   # void outside mesh
    graveyard_offset=10.0,               # vacuum shell 10 cm outside mesh
)
```

`contact_tolerance` overrides the distance the file records and tracks the mesh discontinuously, even if it was written imprinted.

A whole-model `MeshGeometry` with no vacuum boundary (no `graveyard_offset` and no surface with a vacuum condition) is closed before transport: with a void implicit complement it gets a vacuum box 10 cm outside its bounding box, and with a material in the implicit complement the run is refused, since where the box sits would decide how much of that material surrounds the model. Pass `graveyard_offset` to choose the margin. A mesh used as the fill of a CSG cell is bounded by the CSG geometry and gets nothing added.

Properties:

```
print(mesh_geom.num_volumes)      # number of mesh volumes
print(mesh_geom.volume_names)     # material name per volume
print(mesh_geom.volume_measures)  # analytical volume per volume [cm^3]
```

A mesh with tet-meshed volumes also exposes the tets, for passing to another code, tallying (`Tally(mesh=mesh_geom)`, see Tallies) or setting per-tet data such as density factors (see Changing materials between runs):

```
vertices = mesh_geom.vertices                # [x, y, z] per vertex [cm]
tets = mesh_geom.tetrahedra                  # four vertex indices per tet
tet_names = mesh_geom.tet_volume_names       # volume name of each tet, None outside
every volume
tet_volumes = mesh_geom.tet_volumes          # cm3 per tet
mesh_geom.tet_density_factors = factors      # one per tet: the model starts from these
```

Plotting works the same as CSG geometry:

```
plot = mesh_geom.plot(basis="xz", color_by="material", outline="cell")
plot.to_html("mesh_geometry.html")
```

`MeshGeometry` can be passed directly to `Model()` in place of a CSG `Geometry`.

Sources

Source

A source defines where, in what direction, and at what energy particles are born. All parameters are keyword-only.

```
source = yamc.NeutronSource(
    position=(0, 0, 0),
    energy=14.06e6,
)
```

Defaults: origin at `[0, 0, 0]`, 14.06 MeV, isotropic, neutron, strength 1.0.

`position` accepts a plain `(x, y, z)` tuple and `energy` a single value in eV as shorthand for a fixed point and a monoenergetic source. Pass an explicit spatial or energy distribution (see below) when you need a ring, a spectrum, or any other multi-location / multi-energy source.

The spatial, energy and angular distributions below live in the `yamc.sources` submodule (`yamc.sources.Discrete`, `yamc.sources.Isotropic`, ...). The `NeutronSource` / `PhotonSource` builders themselves stay at the top level; you can also `from yamc import sources` and write `sources.Discrete(...)`.

Spatial distributions

```
# Fixed point, using the (x, y, z) tuple shorthand
source = yamc.NeutronSource(position=(0, 0, 0), energy=14.06e6)

# Cylindrical ring source (a bare ring). For a real tokamak plasma, with
# its shape and its density/temperature profiles, see
# "Parametric tokamak plasma source" below.
import math
source = yamc.NeutronSource(
    position=yamc.sources.CylindricalRing(
        radius=yamc.sources.Discrete([300], [1]),   # radius in cm
        phi=yamc.sources.Uniform(0, 2 * math.pi),   # full ring
        z=yamc.sources.Discrete([0], [1]),          # midplane
    ),
    energy=14.06e6,
)
```

Source on a mesh

`yamc.sources.MeshDistribution` spreads a source over the elements of a mesh: the voxels of a `RegularRectangularMesh`, the cells of a `RegularCylindricalMesh`, or the tetrahedra of a tet

`MeshGeometry`. Each element emits in proportion to its strength, from a point uniform within it. Pass it as `position=` to `NeutronSource` or `PhotonSource`.

```
mesh = yamc.RegularRectangularMesh([0, 0, 0], [10, 10, 10], [2, 1, 1])
space = yamc.sources.MeshDistribution(mesh, strengths=[3.0, 1.0])
source = yamc.PhotonSource(position=space, energy=1e6)

print(space.num_elements)    # 2
print(space.total_strength)  # 4.0
```

Here three quarters of the photons are born in the voxel with `x` from 0 to 5 cm and one quarter in the other.

`strengths` takes one non-negative value per element, in the order a tally on the same mesh numbers its bins, so a mesh tally's `mean` can be passed in directly. Omitted, each element's strength is its volume, which makes the source uniform over the mesh. Elements with zero strength never emit. A list of the wrong length, a NaN, infinite or negative value, or strengths that sum to zero raise `ValueError`. When the strengths are rates (particles per second), `total_strength` is the source rate to scale per-source-particle tallies by.

`cells=` (a `Cell` or a list of them) or `materials=` (a `Material` or a list) keeps each element's strength but has it emit only from the parts of it inside those cells, or inside the cells filled with those materials. A decay photon source on a voxel half filled with steel is then born in the steel, not in the void beside it. These need the geometry, so they apply in a `Model` run; `space.sample()` on its own draws from the whole element.

Energy distributions

```
# Monoenergetic
energy = 14.06e6

# Discrete line spectrum, e.g. Co-60's two main decay gammas,
# 1.17 MeV and 1.33 MeV, emitted in ~equal intensity
energy = yamc.sources.Discrete([1.1732e6, 1.3325e6], [0.5, 0.5])

# Histogram (piecewise-constant) spectrum: n+1 ascending bin edges (eV) and one
# per-bin weight. Weights are per-bin probability mass and are normalized
# internally, so a raw multigroup flux per group can be passed directly. Here
# 30% of neutrons are born uniformly in [0, 1] MeV and 70% in [1, 20] MeV.
energy = yamc.sources.Histogram([0.0, 1e6, 20e6], [0.3, 0.7])

# The bin edges can also be a built-in group structure name instead of a list
# (e.g. "CCFE-709", "UKAEA-1102"); pass one weight per group. yamc.group_structure()
# returns the edges of any of them, and yamc.group_structure_names() lists them;
# see the table under Tallies > Energy binning.
energy = yamc.sources.Histogram("CCFE-709", flux_per_group)   # 709 weights

# Uniform range
energy = yamc.sources.Uniform(1e6, 20e6)

# Gaussian
energy = yamc.sources.Normal(14.06e6, 3e5)

# DT fusion neutron spectrum (Ballabio et al.)
energy = yamc.sources.fusion_neutron_spectrum(ion_temp=20000.0)
# DD fusion neutron spectrum
energy = yamc.sources.fusion_neutron_spectrum(ion_temp=20000.0, reactants='DD')
```

Angular distributions

```
direction = yamc.sources.Isotropic()                # default, usually omitted
direction = yamc.sources.Monodirectional([0, 0, 1])  # beam in +z
```

Parametric tokamak plasma source

`yamc.sources.tokamak_source()` builds a tokamak's neutron source from the plasma itself (its shape, confinement mode, and ion density and temperature profiles) rather than from a hand-placed ring.

Bind the common parameters with `functools.partial`, then override the sector, fuel or mesh resolution when calling the source factory.

```
from functools import partial

import yamc

# Shape (cm, dimensionless)
shape = yamc.sources.TokamakShape(
    major_radius=906.0,
    minor_radius=292.258,
    elongation=1.557,
    triangularity=0.270,
    shafranov_factor=0.44789,
)

make_plasma_source = partial(
    yamc.sources.tokamak_source,
    shape=shape,
    # Confinement mode: "L", "H" or "A"
    mode="H",
    # Ion density profile (m^-3, cm for the pedestal radius)
    ion_density_centre=1.09e20,
    ion_density_peaking_factor=1,
    ion_density_pedestal=1.09e20,
    ion_density_separatrix=3e19,
    pedestal_radius=0.8 * 292.258,
    # Ion temperature profile (eV)
    ion_temperature_centre=45.9e3,
    ion_temperature_peaking_factor=8.06,
    ion_temperature_beta=6.0,
    ion_temperature_pedestal=6.09e3,
    ion_temperature_separatrix=0.1e3,
)

sources = make_plasma_source()
model = yamc.Model(geometry=geometry, source=sources, tallies=tallies)
```

Lengths are in cm, ion densities in m^-3, ion temperatures in eV and angles in radians. The full argument list is in the API reference.

The shape is its own object, `TokamakShape`, so it can be checked before any source is built. It refuses an unphysical shape (a minor radius not less than the major radius, a triangularity outside -1 to 1, a Shafranov factor of half the minor radius or more) when it is constructed, and gives the plasma volume and points on any flux surface:

```
shape.volume()                  # plasma volume inside the boundary, cm^3
r, z = shape.evaluate(1.0, 0.0) # outboard midplane of the boundary, cm
```

`evaluate(rho, theta)` takes the normalised radius `rho` (0 on the magnetic axis, 1 on the boundary) and the poloidal angle `theta` in radians. Either may be a sequence, so a whole flux surface comes back in one call.

It returns a list of `NeutronSource` ring sources. The plasma is discretised onto a cylindrical (R, phi, Z) mesh and each non-empty voxel becomes one source per reaction, carrying the Ballabio spectrum at

that voxel's emission-weighted ion temperature. Strengths are normalised to sum to 1, so the absolute neutron rate comes from the simulation's source rate. Pass the list straight to `Model(source=...)`, as under "Multiple sources and strength weighting" below.

The mesh defaults to `(100, 100)`, which is about 15000 sources for D-T fuel. Every non-empty voxel costs one source, so pass a coarser `mesh_resolution=(50, 50)` when a smoother source is not worth the memory.

Toroidal sectors

A wedge model only needs the matching wedge of plasma. `start_angle` and `rotation_angle` are in radians; a negative `rotation_angle` sweeps the other way.

```
import math

sources = make_plasma_source(
    start_angle=0.0,
    rotation_angle=math.pi / 4,    # a 45 degree sector
)
```

Fuel

`fuel` takes atom fractions that sum to 1, and defaults to 50:50 D-T. The supported species are `"D"` and `"T"`, giving D(d,n)3He and T(d,n)4He neutrons.

```
sources = make_plasma_source(fuel={"D": 1.0})          # D-D only
sources = make_plasma_source(fuel={"D": 0.4, "T": 0.6})
```

T(t,2n)4He is not included because yamc has no T-T neutron spectrum model, so D-D and D-T carry the whole yield. For 50:50 D-T fuel at reactor temperatures the T-T channel is well under a percent of the total (0.7% for the parameters above), and pure-tritium fuel is rejected rather than silently returning no neutrons.

Plasma profiles

The profiles the source is built from are exposed on their own, for plotting and for sanity-checking a parameter set. They depend only on the minor radius, and each accepts a float or a sequence of floats.

```
densities = yamc.sources.tokamak_ion_density(
    mode="H",
    ion_density_centre=1.09e20,
    ion_density_peaking_factor=1,
    ion_density_pedestal=1.09e20,
    ion_density_separatrix=3e19,
    minor_radius=292.258,
    pedestal_radius=0.8 * 292.258,
    r=[0.0, 100.0, 292.258],           # cm
)

temperatures = yamc.sources.tokamak_ion_temperature(
    r=[0.0, 100.0, 292.258],
    mode="H",
    ion_temperature_centre=45.9e3,
    ion_temperature_peaking_factor=8.06,
    ion_temperature_beta=6.0,
    ion_temperature_pedestal=6.09e3,
    ion_temperature_separatrix=0.1e3,
    minor_radius=292.258,
    pedestal_radius=0.8 * 292.258,
)
```

`tokamak_neutron_source_density()` completes the set: it gives the Bosch-Hale reaction rate density in neutrons/s/m^3. Mapping flux coordinates onto `(R, Z)`, with the elongation, triangularity and Shafranov shift applied, is `TokamakShape.evaluate()` above.

Plotting the two profiles across the plasma:

```
import matplotlib.pyplot as plt
import yamc

minor_radius = 292.258
pedestal_radius = 0.8 * minor_radius
radii = [i * minor_radius / 200 for i in range(201)]

densities = yamc.sources.tokamak_ion_density(
    mode="H",
    minor_radius=minor_radius,
    pedestal_radius=pedestal_radius,
    ion_density_centre=1.09e20,
    ion_density_peaking_factor=1,
    ion_density_pedestal=1.09e20,
    ion_density_separatrix=3e19,
    r=radii,
)
temperatures = yamc.sources.tokamak_ion_temperature(
    mode="H",
    minor_radius=minor_radius,
    pedestal_radius=pedestal_radius,
    r=radii,
    ion_temperature_centre=45.9e3,
    ion_temperature_peaking_factor=8.06,
    ion_temperature_beta=6.0,
    ion_temperature_pedestal=6.09e3,
    ion_temperature_separatrix=0.1e3,
)

fig, (top, bottom) = plt.subplots(2, 1, sharex=True, figsize=(8, 6))
top.plot(radii, densities)
top.set_ylabel("Ion density [m$^{-3}$]")
bottom.plot(radii, temperatures)
bottom.set_xlabel("Minor radius [cm]")
bottom.set_ylabel("Ion temperature [eV]")
for axis in (top, bottom):
    axis.axvline(0.8 * minor_radius, ls="--", c="grey", lw=1, label="pedestal")
    axis.grid(True, alpha=0.3)
top.legend()
plt.show()
```

Plotting the emission

Sampling the sources shows the plasma's poloidal cross-section: the D-shape from the triangularity, the elongation, and the emission concentrated near the magnetic axis.

```python
import math

import matplotlib.pyplot as plt
import numpy as np
import yamc

sources = make_plasma_source(mesh_resolution=(60, 60))

# Share the points out between the sources by strength, the way the transport
# picks them. Sampling every source equally would instead give each voxel the
# same weight, hiding both the peaked emission and the true D-D fraction.
n_points = 40_000
strengths = np.array([source.strength for source in sources])
counts = np.random.default_rng(1).multinomial(n_points, strengths / strengths.sum())

radii, heights, energies = [], [], []
for source, n in zip(sources, counts):
    if n == 0:
        continue
    positions, sampled = source.sample_n(int(n))
    for (x, y, z), energy in zip(positions, sampled):
        radii.append(math.hypot(x, y))
        heights.append(z)
        energies.append(energy / 1e6)

fig, ax = plt.subplots(figsize=(7, 7))
scatter = ax.scatter(radii, heights, c=energies, s=2, alpha=0.4, cmap="plasma")
ax.set_xlabel("R [cm]")
ax.set_ylabel("Z [cm]")
ax.set_aspect("equal")
ax.set_title("Neutron birth sites, coloured by energy")
fig.colorbar(scatter, ax=ax, label="Birth energy [MeV]")
plt.show()
```

Emission crowds towards the magnetic axis, where the plasma is hottest, and the sparse 2.5 MeV points are the D-D component, about 0.5% of the yield for 50:50 D-T fuel at these temperatures.

Stellarator plasma source

`yamc.sources.stellarator_source()` builds a stellarator's neutron source from its shape and the same ion density and temperature profiles as the tokamak source.

The shape is a `StellaratorShape`: R and Z as Fourier-Zernike series, in the conventions of DESC,

$$R(\rho,\theta,\zeta) = \sum_{lmn} R_{lmn}\, \mathcal{R}_l^{|m|}(\rho)\, \Theta_m(\theta)\, \Phi_n(\zeta)$$

and likewise for Z. $\mathcal{R}_l^{|m|}$ is the Zernike radial polynomial ($\mathcal{R}_l^{|m|}(1) = 1, l \geq |m|, l - |m|$ even), Θ_m is $\cos(m\theta)$ for $m \geq 0$ and $\sin(|m|\theta)$ for $m < 0$, and Φ_n is $\cos(nN\zeta)$ for $n \geq 0$ and $\sin(|n|N\zeta)$ for $n < 0$, with N the number of field periods. ρ is the square root of the normalised toroidal flux and ζ

the cylindrical toroidal angle. Each mode is an `(l, m, n)` triple and coefficients are in cm. A rotating ellipse with five field periods, $R = R_0 + a\rho\cos\theta + e\rho\cos(\theta - N\zeta)$ and $Z = a\rho\sin\theta - e\rho\sin(\theta - N\zeta)$, is seven coefficients:

```
import yamc

shape = yamc.sources.StellaratorShape(
    num_field_periods=5,
    r_modes=[(0, 0, 0), (1, 1, 0), (1, 1, 1), (1, -1, -1)],
    r_lmn=[600.0, 150.0, 50.0, 50.0],
    z_modes=[(1, -1, 0), (1, -1, 1), (1, 1, -1)],
    z_lmn=[150.0, -50.0, 50.0],
)

sources = yamc.sources.stellarator_source(
    shape=shape,
    mode="H",
    ion_density_centre=1.09e20,
    ion_density_peaking_factor=1,
    ion_density_pedestal=1.09e20,
    ion_density_separatrix=3e19,
    ion_temperature_centre=45.9e3,
    ion_temperature_peaking_factor=8.06,
    ion_temperature_beta=6.0,
    ion_temperature_pedestal=6.09e3,
    ion_temperature_separatrix=0.1e3,
    pedestal_rho=0.8,
)
model = yamc.Model(geometry=geometry, source=sources, tallies=tallies)
```

The constructor refuses a shape whose surfaces reach $R \le 0$, self-intersect or are not nested. `shape.evaluate(rho, theta, zeta)` gives (R, Z) for floats or sequences, and `shape.volume()` the plasma volume.

The plasma is split into 50 radial shells in ρ, each one source per reaction. A shell samples positions with density $S(\rho)\,R\,|\tau|$, the neutron source density times the volume element of the shape, and emits the Ballabio spectrum at its emission-weighted ion temperature. Strengths are normalised to sum to 1, and `fuel` works as for the tokamak source.

The profiles are the tokamak ones with the minor radius replaced by ρ, so `pedestal_rho` (between 0 and 1) places the pedestal. To plot them, call `tokamak_ion_density()` / `tokamak_ion_temperature()` with `minor_radius=1`, `pedestal_radius=pedestal_rho` and `r` set to the ρ values.

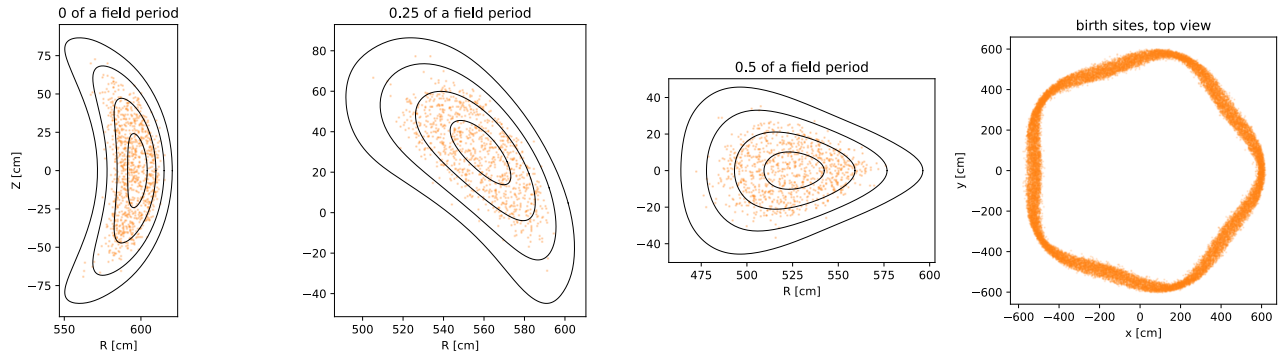
From DESC

A DESC equilibrium already is this series. Its coefficients are in m:

```
import desc.io

eq = desc.io.load("equilibrium.h5")
shape = yamc.sources.StellaratorShape(
    num_field_periods=int(eq.NFP),
    r_modes=eq.R_basis.modes.tolist(),
    r_lmn=(100 * eq.R_lmn).tolist(),
    z_modes=eq.Z_basis.modes.tolist(),
    z_lmn=(100 * eq.Z_lmn).tolist(),
)
```

The W7-X equilibrium that ships with DESC, converted this way, is in `docs/data/w7x_fourier_zernike.json`. Its flux surfaces at the start, a quarter and half way through a field period, and where its neutrons are born:

```
import json
import math

import matplotlib.pyplot as plt
import numpy as np
import yamc

shape =
yamc.sources.StellaratorShape(**json.load(open("docs/data/w7x_fourier_zernike.json"))
)
sources = yamc.sources.stellarator_source(
    shape=shape,
    mode="H",
    ion_density_centre=1.09e20,
    ion_density_peaking_factor=1,
    ion_density_pedestal=1.09e20,
    ion_density_separatrix=3e19,
    ion_temperature_centre=45.9e3,
    ion_temperature_peaking_factor=8.06,
    ion_temperature_beta=6.0,
    ion_temperature_pedestal=6.09e3,
    ion_temperature_separatrix=0.1e3,
    pedestal_rho=0.8,
)

# Birth sites, shared out between the sources by strength as the transport does.
strengths = np.array([source.strength for source in sources])
counts = np.random.default_rng(1).multinomial(300_000, strengths / strengths.sum())
points = np.concatenate(
    [np.array(source.sample_n(int(n))[0]) for source, n in zip(sources, counts) if n]
)
phi = np.arctan2(points[:, 1], points[:, 0])
major = np.hypot(points[:, 0], points[:, 1])

period = 2 * math.pi / shape.num_field_periods
thetas = np.linspace(0, 2 * math.pi, 200)
fig, axes = plt.subplots(1, 4, figsize=(16, 4.5))
for ax, fraction in zip(axes, (0.0, 0.25, 0.5)):
    zeta = fraction * period
    near = np.abs((phi - zeta + math.pi) % (2 * math.pi) - math.pi) < 0.01
    ax.scatter(major[near], points[near, 2], s=1, alpha=0.3, color="tab:orange",
rasterized=True)
    for rho in (0.25, 0.5, 0.75, 1.0):
        r, z = shape.evaluate(rho, thetas, zeta)
        ax.plot(r, z, color="k", lw=0.8)
    ax.set_aspect("equal")
    ax.set_xlabel("R [cm]")
    ax.set_title(f"{fraction:g} of a field period")
axes[0].set_ylabel("Z [cm]")
axes[3].scatter(points[::10, 0], points[::10, 1], s=0.5, alpha=0.2,
color="tab:orange", rasterized=True)
axes[3].set_aspect("equal")
axes[3].set_xlabel("x [cm]")
axes[3].set_ylabel("y [cm]")
axes[3].set_title("birth sites, top view")
fig.tight_layout()

from io import StringIO as _S
_b = _S(); plt.savefig(_b, format="svg", bbox_inches="tight"); print(_b.getvalue())
```



Emission is concentrated on the magnetic axis, where the plasma is hottest.

From VMEC and VMEC++

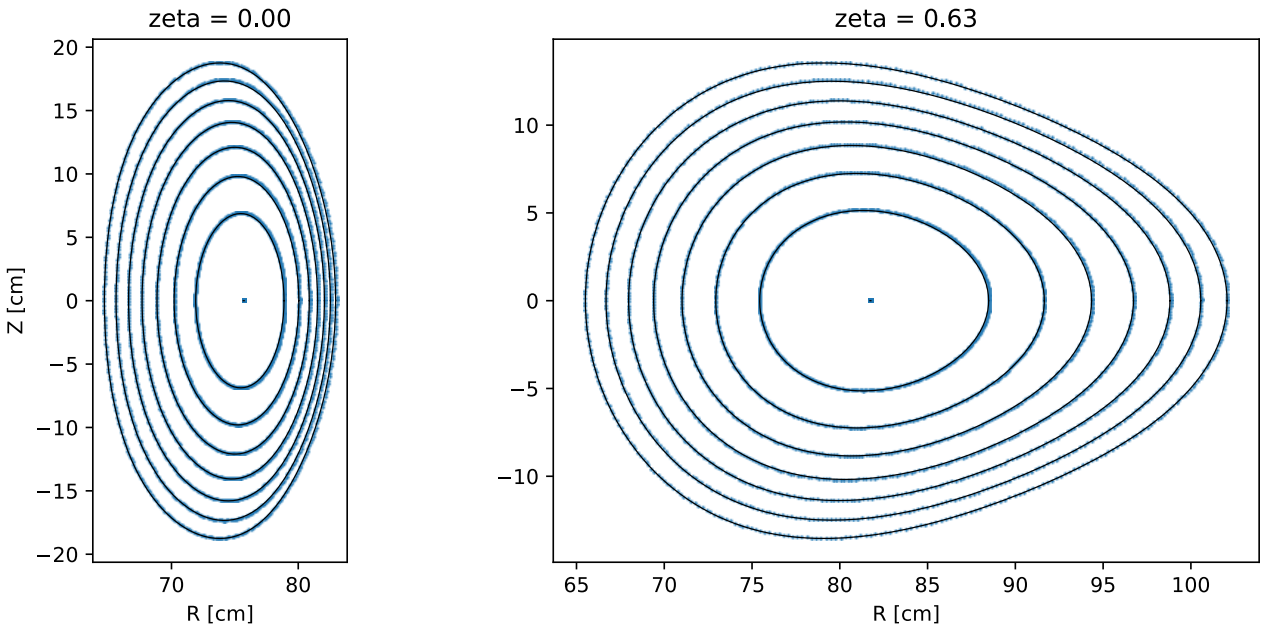
A wout file stores R and Z on a set of radial surfaces in the basis $\cos(m\theta - n\zeta)$ / $\sin(m\theta - n\zeta)$. VMEC2000, VMEC++ and DESC's VMEC export all write it as classic NetCDF, which `scipy.io.netcdf_file` reads:

```
from scipy.io import netcdf_file

with netcdf_file("wout_example.nc", mmap=False) as wout:
    arrays = {name: wout.variables[name].data.copy() for name in ("nfp", "xm", "xn",
"rmnc", "zmns")}
```

The function below turns those arrays into a `StellaratorShape`. It splits each term into the product basis, then fits every (m, n) coefficient across the surfaces with Zernike radial polynomials, choosing the degree by cross-validation on alternate surfaces. It needs only numpy; copy it as it stands. Here it runs on the arrays of VMEC++'s CTH-like example (`docs/data/cth_like_wout.json`), and the fitted surfaces (lines) are drawn over the wout's own (dots):

```
import json
import math

import matplotlib.pyplot as plt
import numpy as np
import yamc


def zernike_radial(rho, ell, m):
    """R_l^m(rho) = (-1)^k rho^m P_k^(m, 0)(1 - 2 rho^2), k = (l - m) / 2."""
    x, p_prev, p = 1.0 - 2.0 * rho**2, np.zeros_like(rho), np.ones_like(rho)
    for j in range(1, (ell - m) // 2 + 1):
        if j == 1:
            p_prev, p = p, (m + 1) + 0.5 * (m + 2) * (x - 1.0)
            continue
        s = 2 * j + m
        p_prev, p = p, (
            (s - 1) * (s * (s - 2) * x + m**2) * p - 2 * (j + m - 1) * (j - 1) * s *
p_prev
        ) / (2 * j * (j + m) * (s - 2))
    return (-1) ** ((ell - m) // 2) * rho**m * p


def shape_from_wout(nfp, xm, xn, rmnc, zmns, rmns=None, zmnc=None, max_degree=20):
    """A StellaratorShape from wout arrays (m, shape (ns, mn_mode))."""
    rho = np.sqrt(np.linspace(0.0, 1.0, len(rmnc)))
    folds = (np.arange(len(rho))[0::2], np.arange(len(rho))[1::2])

    def fit(values, m):
        def design(r, degree):
            return np.column_stack([zernike_radial(r, m + 2 * i, m) for i in
range(degree + 1)])

        def held_out_error(degree):
            return max(
                np.abs(design(rho[b], degree) @ np.linalg.lstsq(design(rho[a],
degree), values[a], rcond=None)[0] - values[b]).max()
                for a, b in (folds, folds[::-1])
            )

        degree = min(range(min(max_degree, len(folds[1]) - 2) + 1),
key=held_out_error)
        return [m + 2 * i for i in range(degree + 1)], np.linalg.lstsq(design(rho,
degree), values, rcond=None)[0]

    shape = {"num_field_periods": int(nfp)}
    for name, cos, sin in (("r", rmnc, rmns), ("z", zmnc, zmns)):
        columns = {}
        for j, (m, n) in enumerate(zip(np.asarray(xm, int), np.asarray(xn, int) //
int(nfp))):
            c = 0.0 if cos is None else np.asarray(cos)[:, j]
            s = 0.0 if sin is None else np.asarray(sin)[:, j]
            m, sign, n = int(m), int(np.sign(n)), int(abs(n))
            # cos(a - b) = cos a cos b + sin a sin b, sin(a - b) = sin a cos b - cos
a sin b
            terms = [((m, n), c), ((m, -n), -sign * s)]
            if m > 0:
                terms += [((-m, -n), sign * c), ((-m, n), s)]
            for mode, value in terms:
```

```
                columns[mode] = columns.get(mode, 0.0) + value
        modes, lmn = [], []
        for (m, n), values in columns.items():
            if np.all(np.asarray(values) == 0.0):
                continue
            ells, coefficients = fit(values, abs(m))
            modes += [(ell, m, n) for ell in ells]
            lmn += (100.0 * coefficients).tolist()
        shape[f"{name}_modes"], shape[f"{name}_lmn"] = modes, lmn
    return yamc.sources.StellaratorShape(**shape)


wout = json.load(open("docs/data/cth_like_wout.json"))
shape = shape_from_wout(**wout)

# The wout's own surfaces, summed from its Fourier coefficients (m to cm).
xm, xn = np.array(wout["xm"]), np.array(wout["xn"])
rmnc, zmns = 100 * np.array(wout["rmnc"]), 100 * np.array(wout["zmns"])
thetas = np.linspace(0, 2 * math.pi, 400)
fig, axes = plt.subplots(1, 2, figsize=(10, 4.5))
for ax, zeta in zip(axes, (0.0, math.pi / wout["nfp"])):
    phase = np.outer(thetas, xm) - xn * zeta
    for j in range(0, len(rmnc), 2):
        rho = math.sqrt(j / (len(rmnc) - 1))
        ax.plot(np.cos(phase) @ rmnc[j], np.sin(phase) @ zmns[j], ".", ms=1.5,
color="tab:blue", rasterized=True)
        r, z = shape.evaluate(rho, thetas, zeta)
        ax.plot(r, z, color="k", lw=0.6)
    ax.set_aspect("equal")
    ax.set_xlabel("R [cm]")
    ax.set_title(f"zeta = {zeta:.2f}")
axes[0].set_ylabel("Z [cm]")
fig.tight_layout()

from io import StringIO as _S
_b = _S(); plt.savefig(_b, format="svg", bbox_inches="tight"); print(_b.getvalue())
```



The fit reproduces the stored surfaces to well under a millimetre, and unlike interpolation between surfaces it follows the $\rho^{|m|}$ behaviour of the coefficients near the magnetic axis, where emission peaks. For equilibria with `lasym` set, pass `rmns` and `zmnc` too.

Multiple sources and strength weighting

Pass a list of sources with relative strengths to `Model()`.

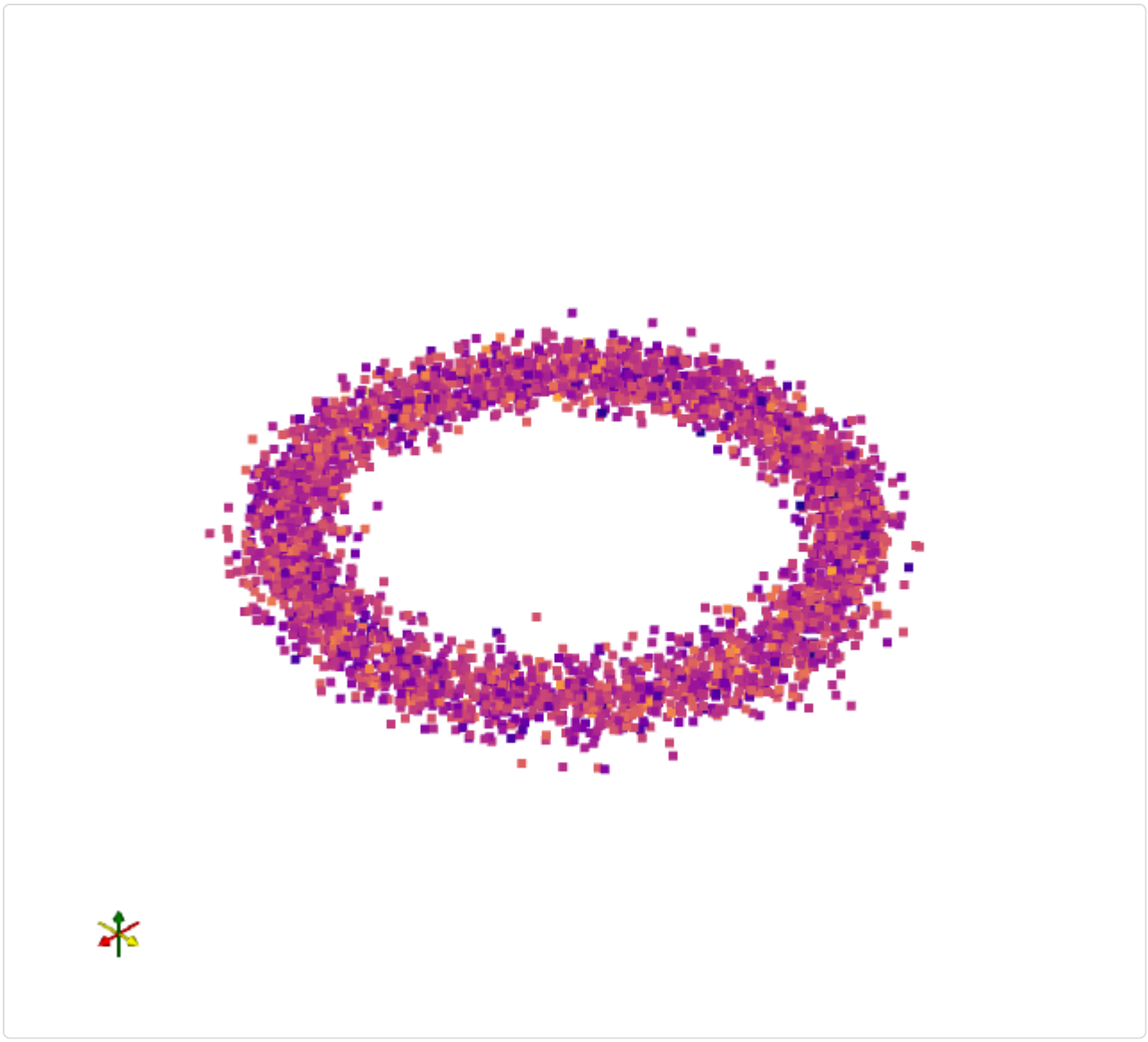
```
source_a = yamc.NeutronSource(
    position=(0, 0, -7),
    energy=14.06e6,
    strength=2.0,
)
source_b = yamc.NeutronSource(
    position=(0, 0, 7),
    energy=2.5e6,
    strength=1.0,
)
model = yamc.Model(geometry=geometry, source=[source_a, source_b])
```

VTK-HDF source export

Sampled source points can be exported for ParaView visualization.

```
source.to_vtkhdf("source_points.vtkhdf", samples=1000)
```

The viewer below is rendered live when these docs are built: it samples a tokamak-like cylindrical-ring plasma source, writes `source_points.vtkhdf`, reads it back, and shows the sampled emission sites as a point cloud coloured by energy (drag to rotate, scroll to zoom).



Tallies

Scores

Tallies accumulate results during particle transport. Scores can be specified by name or ENDF MT number.

```
# Named scores
tally = yamc.Tally(scores=["flux"], cells=cell)
tally = yamc.Tally(scores=["heating"], cells=cell)
tally = yamc.Tally(scores=["heating-local"], cells=cell)     # no secondary photon
contribution
tally = yamc.Tally(scores=["absorption"], cells=cell)
tally = yamc.Tally(scores=["fission"], cells=cell)
tally = yamc.Tally(scores=["damage-energy"], cells=cell)     # DPA (MT 444)
tally = yamc.Tally(scores=["H3-production"], cells=cell)     # TBR

# MT numbers
tally = yamc.Tally(scores=[2], cells=cell)    # elastic
tally = yamc.Tally(scores=[18], cells=cell)   # fission
tally = yamc.Tally(scores=[102], cells=cell)  # (n,gamma)
tally = yamc.Tally(scores=[105], cells=cell)  # (n,t) tritium production

# Multiple scores in one tally
tally = yamc.Tally(scores=["flux", "heating", 105], cells=cell)
```

Virtual response overlays (`response=`)

By default a tally is macroscopic: cross sections are multiplied by the cell's atom density, so it only scores where the responding material physically exists. Passing `response=` scores a virtual response everywhere instead, even in void, as if a detector that is not in the geometry were present. `response` and `nuclides` are mutually exclusive (`nuclides` remains the macroscopic per-nuclide breakdown at the real in-material density).

The type of `response` selects the regime:

- `response="Si28"` or `response=["Li6", "Li7"]` : microscopic cross sections (barns) at unit density, with one result bin per nuclide. Weighting by atom density is left to post-processing.
- `response=<Material>` : the material's macroscopic response, weighting each nuclide's microscopic XS by its real atom density and collapsing to a single combined bin (e.g. a SiO2 dose map). No post-processing is needed.

```
# Nuclide overlay (unit density), one bin per nuclide:
tally = yamc.Tally(
    scores=["heating"],
    response="Si28",
    mesh=mesh,
    name="silicon_dose_overlay",
)

# Material overlay, one combined macroscopic bin weighted by atom density:
silicon = yamc.Material(composition={"Si28": 1.0}, density=2.33)
tally = yamc.Tally(
    scores=["heating"],
    response=silicon,
    mesh=mesh,
    name="silicon_dose",
)
```

Estimator

Each tally has an `estimator` controlling how a particle history is turned into score contributions:

- `"track-length"` (default): every cell crossing contributes `X · weight · Δs`, where `X` is the relevant cross section (for `flux` the XS drops out, leaving a pure length integral; `heating_xs` for `heating`; `σ_r` for a reaction rate; and so on). It has lower variance in optically thin regions and touches more bins per history.
- `"collision"`: scores at every collision site with contribution `(X / Σ_t) · weight`, where `Σ_t` is the URR-modified macroscopic total cross section the collision was sampled against. It has lower variance in optically thick regions and touches one bin per collision rather than one per crossing. Both estimators converge to the same expected value, so identical seeds must agree to within statistical noise.

```
# Default: track-length flux
tally = yamc.Tally(scores=["flux"], cells=cell)

# Collision-estimator flux, set explicitly
tally = yamc.Tally(scores=["flux"], cells=cell, estimator="collision")

# Collision-estimator reaction rate (URR-aware for total / elastic /
# fission / absorption / capture; smooth XS otherwise)
tally = yamc.Tally(scores=["(n,gamma)"], cells=cell, estimator="collision")
```

Score / estimator compatibility:

Score	Track-length	Collision
`flux`	✓ (default)	✓
`heating` / `heating-local`	✓: neutron KERMA (MT 301/901) and photon collision KERMA, both scored along tracks	✓: neutrons use `(heating_xs / Σ_t) · weight`; photons use the analog absorbed energy `(E_in − E_out − Σ E_banked_photons) · weight` at each collision
`reaction-rate` (`total` / `elastic` / `fission` / `absorption` / arbitrary MT)	✓	✓, URR-aware for MTs 1 / 2 / 18 / 27 / 102
`H1-production` / `H2-production` / `H3-production` / `He3-production` / `He4-production`	✓	✓
`damage-energy` (MT 444)	✓	✓: `(damage_energy_xs / Σ_t) · weight` per neutron collision
`coherent-scatter` / `incoherent-scatter` / `photoelectric` / `pair-production` (photon XS)	✓	✓: `(σ_component / Σ_t) · weight` per photon collision, where `Σ_t` is the macroscopic photon total

Every score type supports both estimators, with the following caveats. In each case the unsupported combination is refused or documented, never silently substituted.

- Photon heating depends on the estimator, because the two follow different physical conventions. `estimator="collision"` scores the analog absorbed energy at each photon collision: `E_in − E_out − Σ E_banked_photons`, which is the electron/positron kinetic energy deposited locally minus the banked TTB bremsstrahlung (the coupled neutron-photon local-deposition convention). This matches a reference code's unfiltered heating to 0.01% on an absorbing Fe sphere, but not its photon-filtered share, which that code books on its electron pseudo-particles. `estimator="track-length"` scores collision KERMA along tracks, which has lower variance and suits mesh tallies. It counts the same electron energy, but the TTB-radiated part is deposited locally instead of where those photons land. On thick absorbing bodies the two agree within ~1%. In thin or streaming geometries the track-length value exceeds the absorbed energy by the escaping radiative fraction, which is the usual distinction between kerma and dose (4.7% on a 5 MeV broomstick). Photon `heating-local` reports the same values as `heating`, since with no electron transport there is nothing to separate.

- Unstructured-mesh tallies support both estimators in every mode. The collision estimator resolves the containing tetrahedron with a BVH point query; under `woodcock` / `hybrid`, flux-score track-length unstructured tallies score true track length along each flight segment (like regular meshes) and other scores use the delta-collision estimator with per-tet attribution.

After running, the chosen estimator is reported on the tally:

```
print(tally.estimator)  # "track-length" or "collision"
```

Variance estimator

Each tally reports a per-bin standard deviation alongside its mean. Variance is estimated with per-history Welford accumulation: each source-particle history is one variance sample, each rayon worker updates its private scratch online with Welford's algorithm, and a pairwise reduction then combines the workers. There are no user settings for this (no batch size and no choice of method).

Spatial binning

Each tally scores in one spatial domain family: cells, materials, mesh, or unstructured mesh. `cells` and `materials` are mutually exclusive: a single tally cannot bin over both.

```python
# Cell binning, single cell (one result bin)
tally = yamc.Tally(scores=["flux"], cells=cell)

# Cell binning, list of cells (one result bin per cell)
tally = yamc.Tally(scores=["flux"], cells=[cell1, cell2, cell3])
# After results = model.simulate_transport():
#   results[tally].mean[0] is the flux in cell1
#   results[tally].mean[1] is the flux in cell2
#   results[tally].mean[2] is the flux in cell3
# A bare single cell is also accepted: cells=cell1 gives one bin.

# Material binning, single material
tally = yamc.Tally(scores=["flux"], materials=material)

# Material binning, list of materials (one result bin per material)
tally = yamc.Tally(scores=["flux"], materials=[m1, m2])

# Regular mesh binning. `shape` is either per-axis voxel counts [nx, ny, nz]
# or a single int (total voxel count, split into roughly cubic bins over the box).
mesh = yamc.RegularRectangularMesh(
    lower_left=[-10, -10, -10],
    upper_right=[10, 10, 10],
    shape=[20, 20, 20],           # or shape=8000 for ~8000 voxels
)
tally = yamc.Tally(scores=["flux"], mesh=mesh)

# Mesh from geometry bounding box (same shape rules: int total or [nx, ny, nz])
mesh = yamc.RegularRectangularMesh.from_domain(geometry, shape=1_000_000)
tally = yamc.Tally(scores=["flux"], mesh=mesh)

# Regular cylindrical (r, phi, z) mesh, suited to tokamak and cylindrical
# geometries. phi_bounds defaults to the full circle (0, 2*pi); pass a
# sub-interval (e.g. (0, math.pi)) for a sector.
import math
mesh = yamc.RegularCylindricalMesh(
    r_bounds=(0.0, 10.0),
    z_bounds=(-5.0, 5.0),
    shape=(10, 16, 20),                # (nr, nphi, nz)
    phi_bounds=(0.0, 2 * math.pi),
    origin=(0.0, 0.0, 0.0),
)
tally = yamc.Tally(scores=["flux"], mesh=mesh)
# Cylindrical cells are not equal in volume: outer radial rings are larger.
# Normalise per cell when comparing flux across radii:
#   results[tally].mean[b] / mesh.element_volume(b)

# Unstructured mesh binning: a bin per tet of a MeshGeometry, in the
# order of mesh_geom.tetrahedra
tally = yamc.Tally(scores=["flux"], mesh=mesh_geom)
tally = yamc.Tally(scores=["flux"], mesh=(mesh_geom, "fuel"))  # one volume's tets

# A tally-only mesh, from any mesher and at any resolution: it need not match
# the geometry, and a bin per cell, in the order given
thermal = yamc.UnstructuredMesh(vertices, cells)
tally = yamc.Tally(scores=["heating"], mesh=thermal)
```

`mesh=` takes every kind of mesh. A regular mesh bins on its grid. A `MeshGeometry` or an `UnstructuredMesh` has a bin per cell, and with a region name, `(mesh_geom, "fuel")`, only that volume's cells score (a volume ID is accepted for a `MeshGeometry` too). `UnstructuredMesh` takes `cell_regions=` to name regions, and has `cells`, `cell_volumes`, `cell_regions` and `to_vtkhdf`.

The `mean` / `std_dev` / `relative_error` arrays are laid out flat. When multiple binning axes are present, the bin order from slowest to fastest varying is `score`, `surface`, `cosine`, `cell`, `material`, `nuclide`, `parent_nuclide`, `energy`, `mesh`. The `surface` and `cosine` axes only appear on surface current tallies.

Energy binning

```
# Custom bin boundaries (eV)
tally = yamc.Tally(scores=["flux"], cells=cell, energy_bins=[0, 0.625, 1e5, 20e6])

# Named group structure
tally = yamc.Tally(scores=["flux"], cells=cell, energy_group_structure="VITAMIN-J-
175")
```

The built-in group structures

name	groups	range	bin holding 14.1 MeV
`XMAS-172`	172	1.00001e-5 eV to 19.6403 MeV	13.84 to 14.92 MeV
`VITAMIN-J-175`	175	1e-5 eV to 19.64 MeV	13.84 to 14.19 MeV
`SCALE-252`	252	0 eV to 20 MeV	13.84 to 14.55 MeV
`TRIPOLI-315`	315	1e-5 eV to 19.64 MeV	13.84 to 14.19 MeV
`SHEM-361`	361	0 eV to 19.6403 MeV	13.84 to 14.92 MeV
`LLNL-616`	616	1e-5 eV to 20 MeV	13.80 to 14.45 MeV
`CCFE-709`	709	1e-5 eV to 1 GeV	14.00 to 14.20 MeV
`SCALE-999`	999	1e-5 eV to 20 MeV	13.84 to 14.19 MeV
`UKAEA-1102`	1102	1e-5 eV to 1 GeV	14.00 to 14.20 MeV
`ECCO-1968`	1968	1.00001e-5 eV to 19.64033 MeV	14.07 to 14.19 MeV
`CCFE-24-PHOTON`	24	1 keV to 20 MeV	photon structure
`VITAMIN-J-42`	42	1 keV to 50 MeV	photon structure

Two of them, `SCALE-252` and `SHEM-361`, have a lowest boundary of exactly 0 eV. That is OpenMC's value and is kept, but it sits below the 1e-5 eV floor of the transport data, so treat their bottom bin as open-ended rather than as a real energy range.

Passing a name that is not in the table raises an error listing the valid names, so a typo cannot silently select a different structure. Every name except `CCFE-24-PHOTON` carries the same boundaries as OpenMC's `openmc.mgxs.GROUP_STRUCTURES` entry of that name, so a spectrum tabulated for one code can be handed to the other without re-binning.

The structures can also be read as data, so a multigroup result can be plotted or folded against the energies it was scored on:

```
yamc.group_structure_names()          # every name in the table above
edges = yamc.group_structure("UKAEA-1102")
len(edges)                            # 1103, one more than the group count
edges[0], edges[-1]                   # 1e-05, 1000000000.0
```

`Tally.energy_bins` returns the same list once a group structure has been set on a tally.

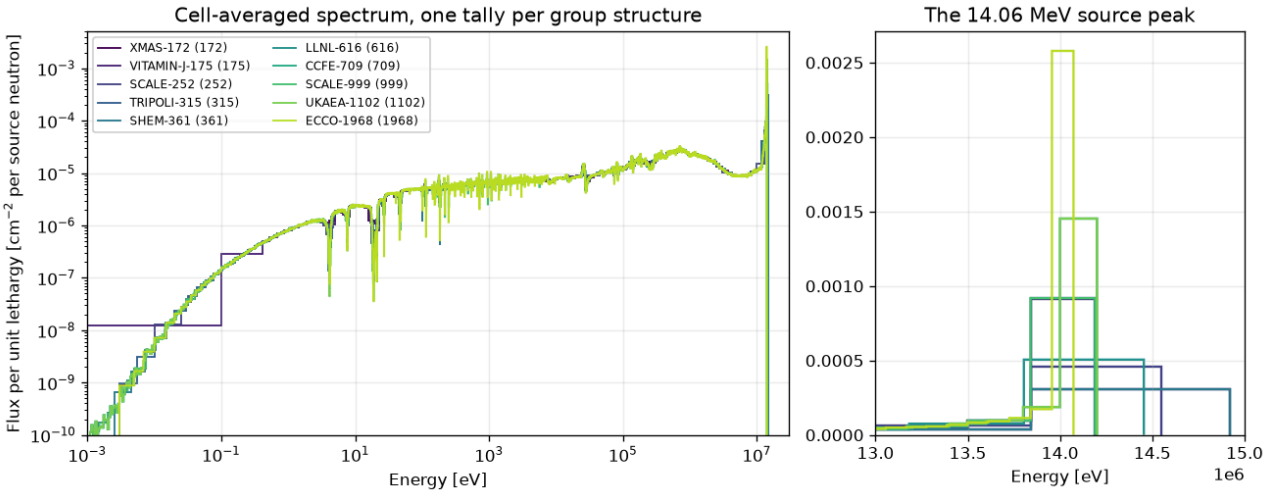
What the choice of structure costs

Choosing a structure trades resolution against bin count, and the effect is easiest to show with an example. Below, a 14.06 MeV point source sits at the centre of a 50 cm sphere of one homogenized mixture of tungsten, water, steel, lithium and beryllium, and the same cell flux is tallied ten times over, once per neutron structure in the table. Flux is divided by bin width in lethargy, so that structures with different bin widths land on one axis.

The left panel shows the whole slowing-down spectrum. All ten agree on it, but the coarse structures step over the resonance dips and the thermal end that the fine ones resolve. The right panel shows the source line itself, which each structure smears across whichever bin contains 14.06 MeV. The wider the bin, the lower and flatter the peak, from `XMAS-172` at the bottom to `ECCO-1968` at the top. This figure is rendered live when these docs are built.

```
import base64
from io import BytesIO

import numpy as np
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt

import yamc

yamc.neutron_cross_section_data = "endf-b8.1"

RADIUS = 50.0
mixture = yamc.Material(
    composition={"W": 0.10, "Fe": 0.20, "Cr": 0.05, "Ni": 0.03,
                 "H2O": 0.30, "Li": 0.20, "Be": 0.12},
    density=4.0,
)
sphere = yamc.Sphere(radius=RADIUS, boundary="vacuum")
cell = yamc.Cell(region=sphere.below, material=mixture)
geometry = yamc.Geometry([cell])            # before the tallies: filtering needs the
cell id
source = yamc.NeutronSource()

names = [n for n in yamc.group_structure_names() if "PHOTON" not in n and n !=
"VITAMIN-J-42"]
tallies = [yamc.Tally(scores=["flux"], cells=cell, energy_group_structure=n, name=n)
           for n in names]
results = yamc.Model(
    geometry=geometry, source=source, tallies=tallies, verbose=[]
).simulate_transport(total_particles=500_000)

volume = 4 / 3 * np.pi * RADIUS**3
fig, (ax, axz) = plt.subplots(1, 2, figsize=(11, 4.4), gridspec_kw={"width_ratios":
[1.9, 1]})
for color, name, tally in zip(plt.cm.viridis(np.linspace(0, 0.9, len(names))), names,
tallies):
    edges = np.array(tally.energy_bins)
    # SCALE-252 and SHEM-361 start at 0 eV, which has no lethargy; widen that
    # bottom bin to the 1e-5 eV transport floor instead.
    lethargy = np.diff(np.log(np.maximum(edges, 1e-5)))
    flux = np.array(results[tally].mean) / volume / lethargy
    flux[flux <= 0] = np.nan                # bins no particle reached, not zero flux
    ax.stairs(flux, edges, color=color, lw=1.1, label=f"{name} ({len(edges) - 1})")
    axz.stairs(flux, edges, color=color, lw=1.4)

ax.set(xscale="log", yscale="log", xlim=(1e-3, 3e7), ylim=(1e-10, 5e-3),
       xlabel="Energy [eV]",
       ylabel=r"Flux per unit lethargy [cm$^{-2}$ per source neutron]",
       title="Cell-averaged spectrum, one tally per group structure")
ax.legend(fontsize=7, ncol=2, loc="upper left", framealpha=0.9)
ax.grid(alpha=0.25)
axz.set(xlim=(1.30e7, 1.50e7), xlabel="Energy [eV]", title="The 14.06 MeV source
peak")
axz.ticklabel_format(axis="x", style="sci", scilimits=(6, 6))
axz.grid(alpha=0.25)
fig.tight_layout()
```

```
buf = BytesIO()
fig.savefig(buf, format="png", dpi=120, bbox_inches="tight")
print(
    '<img alt="the same spectrum tallied on ten group structures" '
    f'src="{base64.b64encode(buf.getvalue()).decode()}" '
    'style="max-width:100%">'
)
```



Nuclide binning

Score per-nuclide contributions separately.

```
tally = yamc.Tally(
    scores=["H3-production"],
    cells=cell,
    nuclides=["Li6", "Li7", "Be9"],
)
# results[tally].mean returns [Li6_score, Li7_score, Be9_score]
```

Particle binning

Restrict scoring to neutrons or photons. This is required for separate neutron and photon spectra.

```
tally_n = yamc.Tally(scores=["flux"], cells=cell, particle="neutron")
tally_p = yamc.Tally(scores=["flux"], cells=cell, particle="photon")
```

Surface currents

A tally with `surfaces=` counts particles crossing those surfaces, instead of scoring along tracks or at collisions. This gives leakage through a boundary directly, and the current through any surface inside a model. There is one result bin per surface, in the order given, and the results are particles per source particle.

The scores count crossings by the side of the surface they cross to:

Score	Counts
`"current-above"`	crossings towards the surface's `.above` side, each adding the particle weight
`"current-below"`	crossings towards `.below`, each adding the particle weight
`"current-net"`	above minus below

Which side is `.above` is the same as when building regions: outside a `Sphere`, `Cylinder`, `Cone` or `Torus`, and the positive side of a `Plane` (`x > offset` for `axis="x"`). On a sphere, `"current-above"` is the outward current and `"current-below"` the inward one. On an x plane they are the currents towards +x and -x. A particle that crosses the same surface several times is counted at every crossing.

```
# Leakage out of a sphere in vacuum. Nothing comes back through the outer
# surface, so above and net are both the leakage and below is zero.
outer = yamc.Sphere(radius=30.0, boundary="vacuum")
leakage = yamc.Tally(
    scores=["current-above", "current-below", "current-net"],
    surfaces=outer,
)
# results[leakage].mean is [above, below, net] per source particle

# Current through a plane inside a model, towards +x and towards -x
wall = yamc.Plane(axis="x", offset=50.0)
through_wall = yamc.Tally(scores=["current-above", "current-below"], surfaces=wall)

# Several surfaces in one tally: one bin per surface
layers = yamc.Tally(scores=["current-above"], surfaces=[inner_sphere, outer_sphere])
```

A surface tally also takes `energy_bins` (the energy of the particle as it crosses), `particle`, and an energy function or dose coefficients. `cosine_bins` bins the crossings by the cosine of the angle between the particle's direction and the surface normal, which points to `.above`. A cosine of 1 is straight along the normal and 0 is grazing; negative cosines are crossings towards `.below`.

```python
import math

# Outward leakage spectrum
leakage_spectrum = yamc.Tally(
    scores=["current-above"],
    surfaces=outer,
    energy_group_structure="VITAMIN-J-175",
    particle="neutron",
)

# Only particles leaving within 17.28 degrees of the normal, as a detector
# viewing part of the surface sees them
detector_view = yamc.Tally(
    scores=["current-above"],
    surfaces=outer,
    cosine_bins=[math.cos(math.radians(17.28)), 1.0],
)
```

A surface tally cannot also take `cells`, `materials`, `mesh`, `nuclides`, `response` or `estimator`. It needs `tracking_mode="surface"` (the default) and CSG surfaces, and runs on the CPU. Each surface needs an id; one is assigned automatically when a surface is created without `surface_id=`.

Example: leakage spectrum from an iron sphere

```python
import matplotlib.pyplot as plt
import numpy as np
import yamc

yamc.neutron_cross_section_data = "endf-b8.1"

outer = yamc.Sphere(radius=30.0, boundary="vacuum")
iron = yamc.Material(composition={"Fe": 1.0}, density=7.874)
cell = yamc.Cell(region=outer.below, material=iron)
geometry = yamc.Geometry([cell])

source = yamc.NeutronSource()

leakage = yamc.Tally(
    scores=["current-above"],
    surfaces=outer,
    energy_group_structure="VITAMIN-J-175",
    name="neutron_leakage",
)

model = yamc.Model(geometry=geometry, tallies=[leakage], source=source)
results = model.simulate_transport(total_particles=200000)

bins = np.array(leakage.energy_bins)
current = np.array(results[leakage].mean)
lethargy = np.log(bins[1:] / bins[:-1])
print(f"neutrons leaking per source neutron: {current.sum():.3f}")

fig, ax = plt.subplots()
ax.stairs(current / lethargy, bins)
ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel("Energy [eV]")
ax.set_ylabel("Leakage per unit lethargy per source neutron")
fig.savefig("leakage_spectrum.png", dpi=150)
```

Energy function weighting

Apply an energy-dependent weighting function to the tally (e.g. response functions, DPA cross sections).

`interpolation` says how the values are read between points: `"cubic"`, a natural cubic spline in linear energy and the default, which needs at least four points; `"linear"`; or `"log-log"`, the convention for tabulated coefficients, which falls back to linear over an interval with a zero value. The last two need two points. An `energy_function` given as `(energies, values, units, interpolation)`, the tuple `as_energy_function()` returns, carries its own. A particle whose energy falls outside the tabulated range scores nothing into that tally, so make the grid span the whole spectrum you expect.

```
tally = yamc.Tally(
    scores=["flux"],
    cells=cell,
    # (energies, values); at least 4 points, energies strictly increasing
    energy_function=([1e-5, 1e3, 1e6, 20e6], [1.0, 1.5, 2.0, 0.5]),
)
```

Dose coefficients

Apply effective dose conversion coefficients directly. They are ICRP-116 by default; add `"icrp74"` as a third element, `("neutron", "AP", "icrp74")`, for ICRP-74. They are interpolated log-log, as contact dose reads the same tables.

```
tally = yamc.Tally(
    scores=["flux"],
    mesh=mesh,
    dose_coefficients=("neutron", "AP"),  # (particle, geometry[, data_source])
)
```

A dose tally scores only the particle its coefficients are for, so a neutron dose tally in a coupled neutron-photon run leaves the photons out. A `particle=` naming the other particle is refused; use `energy_function=` to weight a particle by some other curve.

For the total dose from neutrons and photons, use `"neutron_photon"` as the particle. Each particle is weighted by its own coefficients and both go into the same bin, so the standard deviation is that of the total:

```
total_dose = yamc.Tally(scores=["flux"], mesh=mesh, dose_coefficients=
("neutron_photon", "AP"))
```

Adding a neutron dose tally and a photon dose tally gives the same mean, but not the error of the sum: the photons of a history are correlated with the neutrons that made them, so adding the two errors in quadrature can be wrong in either direction. Add separate neutron and photon tallies as well if the breakdown is wanted.

For ambient dose equivalent H*(10), the quantity most regulations still reference, fetch the ICRP-74 coefficients with `yamc.data.dose_coefficients(..., dose_quantity="ambient")` and fold them in as an energy function, whose tuple names the log-log interpolation itself. H*(10) is defined in an aligned and expanded field, so it takes no geometry, and ICRP-116 does not tabulate it.

```
h10 = yamc.data.dose_coefficients("neutron", dose_quantity="ambient")
tally = yamc.Tally(
    scores=["flux"],
    mesh=mesh,
    energy_function=h10.as_energy_function(),
)
```

The tables cover 1 meV to 20 MeV for neutrons and 10 keV to 10 MeV for photons and are not extrapolated, so a particle outside that range scores nothing. H*(10) is designed to over-estimate effective dose for area monitoring, but for high-energy neutrons it under-reads it.

Bin covariance

`standard_deviation` treats each bin on its own, but the bins of one tally are scored by the same histories and move together, so anything summed over them, a reaction rate over a spectrum for instance, inherits an understated error from the standard deviations alone. `covariance=True` also accumulates the covariance of the bin means, history by history:

```
tally = yamc.Tally(
    scores=["flux"],
    cells=cell,
    energy_group_structure="VITAMIN-J-175",
    covariance=True,
)
# after the run: results[tally].covariance, a bins x bins nested list
```

It is meant for spectra: it is limited to 2048 bins and runs on the CPU only. It is what `Pulse(flux_covariance=...)` takes, as Nuclear-data uncertainty describes.

Neutron and photon flux spectra

A complete example scoring energy-resolved neutron and photon flux with coupled transport.

```
import yamc

# Coupled transport reads neutron and photon data.
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"

# Geometry: iron sphere
sphere = yamc.Sphere(radius=30.0, boundary="vacuum")
iron = yamc.Material(composition={"Fe": 1.0}, density=7.874)
cell = yamc.Cell(region=sphere.below, material=iron)
geometry = yamc.Geometry([cell])

# Source: 14 MeV neutrons
source = yamc.NeutronSource()

# Tallies: energy-binned neutron and photon flux
tally_n = yamc.Tally(
    scores=["flux"],
    cells=cell,
    energy_group_structure="VITAMIN-J-175",
    particle="neutron",
    name="neutron_spectrum",
)
tally_p = yamc.Tally(
    scores=["flux"],
    cells=cell,
    energy_group_structure="CCFE-24-PHOTON",
    particle="photon",
    name="photon_spectrum",
)

model = yamc.Model(
    geometry=geometry,
    tallies=[tally_n, tally_p],
    source=source,
    transport_secondary_photons=True,
)
results = model.simulate_transport(total_particles=500000)

# Plot spectra
import numpy as np
import matplotlib.pyplot as plt

n_bins = np.array(tally_n.energy_bins)
n_flux = np.array(results[tally_n].mean)
n_std = np.array(results[tally_n].standard_deviation)

p_bins = np.array(tally_p.energy_bins)
p_flux = np.array(results[tally_p].mean)
p_std = np.array(results[tally_p].standard_deviation)

fig, ax = plt.subplots()
for bins, flux, std, label in [
    (n_bins, n_flux, n_std, "neutron"),
    (p_bins, p_flux, p_std, "photon"),
]:
    centers = np.sqrt(bins[:-1] * bins[1:])      # geometric bin centres
    ax.step(bins[:-1], flux, where="post", label=label)
    ax.errorbar(centers, flux, yerr=std, fmt="none", capsize=2, alpha=0.5)
```

```
ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel("Energy [eV]")
ax.set_ylabel("Flux [cm / source particle]")
ax.legend()
ax.grid(True, alpha=0.3)
fig.savefig("flux_spectra.png", dpi=150)
```

Models and simulating

Model

Creating a Model

The `Model` brings together geometry, sources, tallies, and simulation settings.

```
model = yamc.Model(
    geometry=geometry,
    tallies=[tally1, tally2],
    source=source,
)
results = model.simulate_transport()
```

The particle count and random seed are per-run arguments to `simulate_transport` (and `simulate_transmutation`), not `Model` constructor kwargs, so one model can be re-run at different counts/seeds without rebuilding it. `seed` defaults to 1 and `total_particles` to `None`, no cap, so a run needs it, `max_runtime` or convergence targets: one with no stop condition raises `ValueError`.

Both CSG `Geometry` and `MeshGeometry` are accepted.

Before a run, inspect which data the assembled model needs:

```
print(model.required_nuclides())
print(model.required_temperatures())
print(model.required_elements())
```

These are requirements, not a download. Temperatures map each nuclide to the requested temperature strings, or `None` if a material loads every available temperature. Elements are needed for photon transport and are empty when the model will transport no photons.

Photon transport

Enable coupled neutron-photon transport with `transport_secondary_photons=True`. A coupled run reads both `yamc.neutron_cross_section_data` and `yamc.photon_cross_section_data`; set the photon one to a library with photoatomic data. The model loads the required element data automatically. No explicit material data read is needed (see Loading nuclear data). A primary `PhotonSource` transports its own photons without this coupling flag.

```
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"

model = yamc.Model(
    geometry=geometry,
    source=source,
    transport_secondary_photons=True,
    photon_cutoff_energy=1000.0,      # eV (default)
    electron_treatment="ttb",          # thick-target bremsstrahlung (default); or
"local"
)
```

For D1S shutdown dose rate calculations, enable decay photon production. A D1S run reads both cross-section settings, set above, and also the decay data and the reactions network, which have no default (see Transmutation data):

```
yamc.transmutation_decay_data = "endf-b8.1"
yamc.transmutation_reactions = "endf-b8.1"

model = yamc.Model(
    geometry=geometry,
    source=source,
    transport_secondary_photons=True,
    use_decay_photons=True,
)
```

Simulation knobs

These `Model` constructor kwargs are changed less often. The defaults are usually fine; adjust them when the simulation aborts or when benchmarking the GPU path.

- `free_gas_threshold` (default `400.0`): the threshold, in multiples of `k_B·T`, above which elastic scattering treats the target nuclide as a stationary scatterer rather than sampling its thermal velocity from the free-gas Maxwell-Boltzmann distribution. The default of 400 kT is the standard convention; raising it widens the energy range where the thermal motion of the target is sampled (slightly slower); lowering it treats the target as stationary from a lower energy (slightly faster, slightly less accurate near the threshold).
- `max_lost_particles` (default `10`): abort `simulate_transport` after this many particles have been lost (no valid cell found at a step, or no surface ahead of a particle in an unbounded cell). A lost particle indicates a geometry problem (overlapping cells, an uncovered region, a numerical-precision edge case); aborting early avoids running a long simulation against a broken model. See Lost particles and geometry debugging for diagnosing them.
- `gpu_max_steps_per_particle` (default `100000`): hard cap on transport steps per particle on the GPU path. CPU transport ignores it, since histories there end through absorption, leakage, and the lost-particle diagnostics; the GPU kernel has no concept of "lost" and needs an explicit cap to bound shader runtime. Setting it and then running on the CPU emits a warning, since the request has no effect there. The default is deliberately high so the GPU matches the CPU on strong

scatterers (a `1000` cap cost ~10% of the integral flux on a pure-H2 sphere). A cap that binds is an error: if any history in a GPU launch is still transporting at the cap, `simulate_transport` raises `ValueError` naming the cap rather than returning under-counted tallies, so raise it for a deep-penetration GPU run, or run on the CPU. Lower it only to bound worst-case kernel time on a pathological geometry you expect to leak or absorb well within it.

- `gpu_fission_bank` (default `True`): whether the GPU banks fission progeny on the device. Inert without fissile material and on the CPU. Turn it off to run a fissile model with a mesh tally on the GPU, which the dispatch otherwise refuses: the combination of per-source accumulation across fission generations and direct mesh scoring is not wired up. With it off, fission progeny are not banked on the device and the run takes the non-fissile loop, which is what lets the mesh path work.

```
model = yamc.Model(
    geometry=geometry,
    source=source,
    free_gas_threshold=400.0,         # default
    max_lost_particles=10,            # default
    gpu_max_steps_per_particle=100000,   # default; GPU path only
    gpu_fission_bank=True,            # default; GPU path only
)

model.gpu_fission_bank = False        # also settable after construction
```

Tracking mode (Woodcock / delta tracking)

`tracking_mode` selects the particle-transport algorithm. `"surface"` (default) is standard ray-tracing: at each step it computes the distance to the nearest cell boundary, advances the particle, and samples a collision against the current material's `Σ_t`. Delta tracking instead samples a free flight against a *global majorant* `Σ_maj` (the per-energy max of `Σ_t` across every material in the model), moves the particle the full sampled distance ignoring cell boundaries, and runs a rejection loop at the sampled point that decides whether the event is real (`Σ_t / Σ_maj`) or virtual (no physics, continue). The speed-up comes from skipping the per-step `closest_boundary` calculation.

There are two delta-tracking modes. For nearly all models, use `"hybrid"`:

```
model = yamc.Model(
    geometry=geometry,
    source=source,
    tracking_mode="hybrid",   # delta tracking, safe in voids
)
```

When to use which:

- `"surface"`: the default, and always safe. It is a good general choice and the fastest option for typical cell-sparse geometries (few cells per mean free path), where delta tracking has few boundaries to skip.
- `"hybrid"`: the recommended Woodcock mode. Delta tracking with an automatic per-cell fallback to surface tracking in cells where delta tracking would be inefficient (voids, large low-density

regions such as a vacuum vessel or air). It keeps the boundary-skipping gain in dense regions and does not slow down in voids, so it is the safe way to get delta-tracking speed on a realistic fusion model. Use it whenever the geometry contains any void or low-density region.

- `"woodcock"`: pure delta tracking with no fallback. Use it only for geometrically dense, void-free models (many cells per mean free path with similar `Σ_t`, e.g. a finely diced or CAD-tessellated geometry). On such models it can be several times faster than surface tracking. Do not use it on a model with large voids: `Σ_maj` then far exceeds the local `Σ_t`, the rejection loop spends most of its time on virtual collisions, and it becomes much slower than `"surface"`. This is the case `"hybrid"` handles. When in doubt, use `"hybrid"`.

All three give the same answer within statistics; they differ only in speed.

Survival biasing (implicit capture)

`variance_reduction` takes a list of technique objects; an empty list (the default) is fully analog transport. Techniques compose in the same list (currently survival biasing, below, and weight windows), and the list order carries no meaning.

`yamc.SurvivalBiasing` enables the first technique, implicit capture: instead of killing the particle when absorption is sampled (analog capture), every collision multiplies the particle weight by the scattering probability `Σ_s / Σ_t` and the particle always scatters. Histories are never cut short by capture, so they keep contributing track length to deep tallies. Low-weight particles are culled by weight-cutoff Russian roulette: below `weight_cutoff` a particle survives with probability `weight / weight_survive` (continuing at `weight_survive`) or is killed, which conserves weight in expectation. The parameters and defaults match the standard `survival_biasing` setting.

```
model = yamc.Model(
    geometry=geometry,
    source=source,
    variance_reduction=[
        yamc.SurvivalBiasing(
            weight_cutoff=0.25,    # default; roulette trigger
            weight_survive=1.0,    # default; weight of roulette survivors
        ),
    ],
)
```

Notes:

- Works in all three tracking modes; the means are unchanged within statistics, only the variance (and history length) differ.
- Judge the benefit by the per-tally FOM printed at the end of the run rather than by the particles/s line. Histories no longer end at capture, so raw throughput often drops (in strongly absorbing media roulette can instead shorten histories and raise it), while the error bars shrink faster than the throughput changes. On absorption-dominated problems a FOM gain of roughly 2x is typical, which means the same error bars in about half the CPU time.

- Most effective on absorption-dominated problems and deep tallies; for leakage-dominated problems it changes little.
- At most one `SurvivalBiasing` entry is allowed in the list (duplicates are rejected). `model.variance_reduction` returns copies: configure technique objects and assign a whole list rather than mutating entries in place.
- Composes with everything weight-aware: tallies, secondary-photon production, D1S decay photons, and transmutation reaction rates all score with the particle weight (photons inherit the parent neutron's pre-discount weight). `compute='gpu'` refuses `variance_reduction` (the kernel runs analog transport) rather than silently ignoring it.

Weight windows

The second variance-reduction technique, weight windows (mesh-based splitting and roulette), targets deep-penetration and streaming tallies that survival biasing alone cannot reach. It has its own generation workflow and is documented separately in Weight windows.

Changing materials between runs

For feedback loops, such as a thermal solve that returns temperatures, a model can change its materials between runs without being rebuilt. The geometry, loaded nuclear data, tallies and weight windows are all kept.

```
model.set_material_temperature(coolant, 650.0)  # K
model.set_material_density(coolant, 0.98)       # in coolant.density_units
results = model.simulate_transport()
```

Only the model's copy of the material changes; `coolant` itself keeps its old values. A density change reuses the microscopic cross sections, so it costs far less than a new model. A temperature change loads the data at the new temperature.

On a tet-meshed `MeshGeometry` the density can also vary tet by tet, as for a liquid breeder whose density follows its temperature. Each factor scales every macroscopic cross section in its tet, so no extra nuclear data is needed:

```
import numpy as np

in_breeder = np.array(geometry.tet_volume_names) == "breeder"
geometry.tet_density_factors = np.where(in_breeder, density_at_each_tet /
breeder.density, 1.0)

# the model starts from the geometry's factors, as from copies of its materials
model = yamc.Model(geometry=geometry, source=source, tallies=tallies,
tracking_mode="hybrid")
results = model.simulate_transport()

# then change the model's copy between runs, as set_material_density does
model.set_tet_density_factors(new_factors)  # one per tet, in the order of
geometry.tetrahedra
model.set_tet_density_factors(None)         # back to the materials' own densities
```

Per-tet factors need `tracking_mode="hybrid"` or `"woodcock"`. Heating and reaction-rate scores follow the scaled density. Coupling to a heat solve runs this loop with FEniCSx.

Saving and loading results

A `TransportResults` object can be written to a single Arrow file and read back in any later session. The full per-bin statistics (mean, variance, history counts) and run provenance are preserved, so a loaded result behaves exactly like a fresh one for plotting, inspection, or combining.

```
results = model.simulate_transport()
results.to_arrow("run1.arrow")                          # save

# Later, or in another session / on another machine:
loaded = yamc.TransportResults.from_arrow("run1.arrow")  # load
print(loaded["flux"].mean, loaded["flux"].relative_error)

# With no path, to_arrow() returns the file's bytes, and from_arrow takes
# them back: no file system needed, as in a browser or Pyodide session.
data = results.to_arrow()
loaded = yamc.TransportResults.from_arrow(data)
```

Combining results from separate runs

`yamc.combine_results` pools the results of independent runs of the same model into one result that is statistically identical to a single longer run. It merges the raw per-bin Welford state (mean, M2, history count) of every tally, the same exact operation the transport loop uses to combine per-thread partials. Means, error bars and figures of merit come out as if all histories had been one simulation (FOM uses the summed wall-clock time).

Every run needs its own seed. yamc derives per-particle random streams from the base seed, so two runs with the same seed re-trace the same histories; `combine_results` refuses them with an error rather than silently producing wrong error bars.

```
# Two runs of the same model, each with its own seed; particle counts can
# differ. Here a 100k run and a 200k run.
model_a = yamc.Model(geometry=geometry, source=source, tallies=tallies)
model_b = yamc.Model(geometry=geometry, source=source, tallies=tallies)

results_a = model_a.simulate_transport(total_particles=100_000, seed=1)
results_b = model_b.simulate_transport(total_particles=200_000, seed=2)

combined = yamc.combine_results(results_a, results_b)   # pooled, as if 300k
histories
print(combined["flux"].relative_error)

# Works across sessions too: save each run with results.to_arrow(...) (see
# "Saving and loading results" above) and combine the Arrow files later.
r1 = yamc.TransportResults.from_arrow("run1.arrow")
r2 = yamc.TransportResults.from_arrow("run2.arrow")
combined = yamc.combine_results(r1, r2)
```

This also allows cluster runs without MPI: launch N jobs with seeds 1..N, save each with `results.to_arrow(...)`, and combine the files afterwards. The combination is statistically exact.

The following are checked before combining, and any failure is a hard error:

- distinct seeds across all contributing runs;
- same model: geometry, materials, source, physics settings and per-nuclide nuclear-data libraries (a library mismatch is reported per nuclide, e.g. `n:Co58: "endf-b8.1" vs "fendl-3.2d"`);
- same-name tallies must have identical configurations. A material filter's bins are matched by the material each id names, not by the id, so a model built again in the same process, whose materials get new ids, still combines;
- GPU-produced results are refused (the GPU path does not record the exact merge state).

Results saved by yamc 0.16.2 still load and combine with each other. They do not combine with runs from 0.16.3 if the model is a CSG model with materials (the fingerprint now labels each material by its place in the geometry, not its id), or if a tally both runs have filters on materials (0.16.2 did not record which material each id names).

Tallies present in only some runs are carried through unchanged with a `UserWarning`. Their statistics come from the runs that scored them, and their FOM uses those runs' time only. `results.runs` lists the provenance (seed, history count, elapsed time, fingerprint, data libraries) of every contributing run.

Plotting

`model.plot()` renders the geometry with source points overlaid.

```
plot = model.plot(
    basis="xz",
    color_by="material",
    outline="cell",
    resolution=400000,
    n_samples=1000,  # number of source points to overlay
)
plot.to_html("model.html")
plot.to_png("model.png")
```

The same plot rendered live (the dots are sampled source points):



For manual matplotlib plotting:

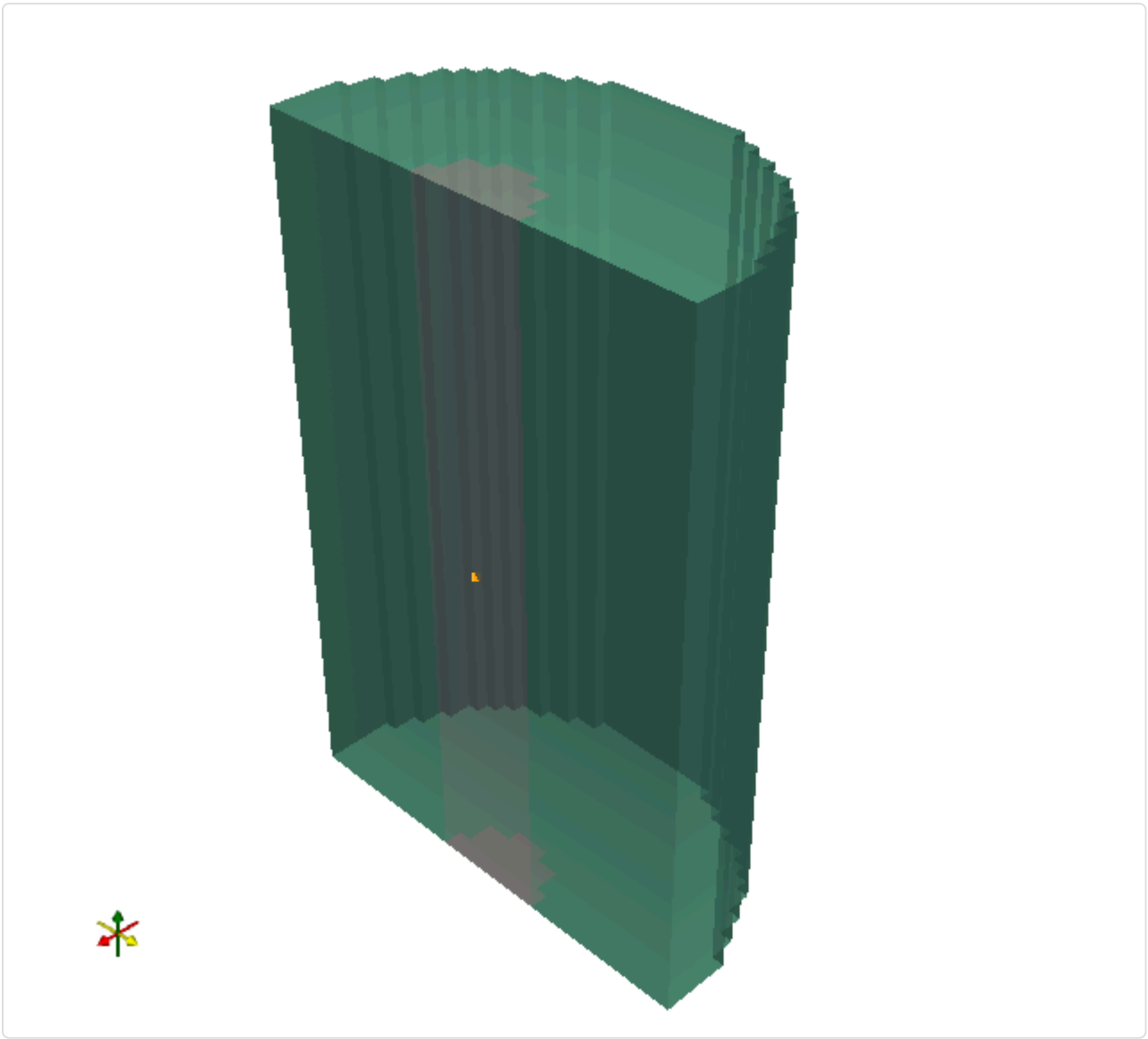
```
slice_data = model.sample_slice(basis="xy", resolution=(400, 400))
```

VTK-HDF model export

`model.to_vtkhdf` writes the geometry and the sampled source together (as a VTK-HDF partitioned collection), so a single file shows both in ParaView.

```
model.to_vtkhdf("model.vtkhdf", resolution=10000, samples=1000)  # geometry + source
model.to_vtkhdf("source.vtkhdf", include=["source"], samples=1000)
```

The viewer below is rendered live when these docs are built: it writes `model.vtkhdf`, reads it back, and shows the voxelized geometry (cut in half, coloured by material) together with the sampled source points (drag to rotate, scroll to zoom).



Save and load (JSON)

A model can be serialised to and from JSON. The JSON captures the specification: geometry, materials, source, tallies and run parameters. Nuclide cross-section data is not included, so a loaded model still needs to load nuclear data before `simulate_transport` (the same way a model built from scratch does).

```
# Write a pretty-printed JSON file (git-diff-friendly).
model.to_json("blanket_v3.json")

# Round-trip from disk.
model = yamc.Model.from_json("blanket_v3.json")

# Or without a file: with no path, to_json() returns the JSON as bytes,
# and from_json() takes bytes as well as a path.
spec = model.to_json()                # pretty=False for a single line
model2 = yamc.Model.from_json(spec)
```

Typical uses:

- Check models into git so reviewers can `diff` the spec without running anything.
- Build a model in a notebook, serialise it, and re-load it in an HPC job script. The particle count and seed are not part of the saved model, so pass a different `total_particles` or `seed` to `simulate_transport` when you run it there.
- Ship a model spec alongside a publication.

Export to a runnable HTML page (`to_html`)

`model.to_html(path)` writes a self-contained HTML file (with no path, `to_html()` returns the page as bytes) that opens in any browser and runs the simulation inside the page, using a WASM build of the transport engine. It can be used to share a working example with someone who does not have Python or yamc installed, to embed a runnable demo in a paper or course, or to lock down a model for educational use.

```python
# Minimal: writes geometry / source / tallies; XS data fetched at
# runtime from yamc-data.xsplot.com (requires network), one temperature
# per nuclide: the byte ranges of the batches each material's
# temperature needs, not the whole file.
model.to_html("demo.html")

# Fully offline: embed every nuclide the current materials use.
# Recipient gets a portable file that runs with no network.
model.to_html("portable.html", embed_cross_sections=True)

# Embed a family of nuclides to anticipate the recipient swapping
# Li6 for Li7 without breaking the offline behaviour.
model.to_html("portable.html", embed_cross_sections=["Li"])

# Lock-down for teaching: the recipient can re-simulate but cannot edit
# geometry or tally definitions.
model.to_html(
    "locked.html",
    show_surfaces=False,
    show_materials=True,    # let students change density / composition
    show_tallies=False,
)

# Bake in a static snapshot of an interactive mesh-tally plot so the
# recipient sees the spatial distribution on first load.
plot = tally.plot(geometry=geometry, basis="xz")
model.to_html(
    "with_flux_map.html",
    embed_cross_sections=True,
    tally_plot_html=plot._repr_html_(),
)
```

Embedded XS data inflates the HTML size by roughly the raw byte count × 1.33 (base64), and embeds every temperature. ENDF/B-VIII.1 nuclides range from ~1.2 MB (Li6) to ~56 MB (Fe56) whole, so embedding everything in a heavy-element material can produce a multi-hundred-MB HTML; pick `embed_cross_sections` accordingly. The fetch-at-runtime path is lighter: for a material with a temperature set it fetches that temperature of each nuclide's reactions and energy grids (about 22 MB for Fe56 rather than 56), because the published data is written one Arrow batch per (MT, temperature) with a byte-range index in each nuclide's `version.json`.

The `show_*` flags are presentation-only: the recipient can still inspect (or open devtools and edit) the embedded model JSON. Use them to declutter the UI, not as a security boundary.

Weight windows

Weight windows are a variance-reduction technique for deep-penetration and streaming problems: places where an analog simulation almost never sends a particle, so the tally there is dominated by a handful of lucky histories (or stays zero). A weight window defines, per mesh voxel and energy group, a target weight band. A particle above the band is split into several lower-weight copies (more histories where the tally needs them); a particle below the band is rouletted (killed or promoted); a particle inside the band is left alone. Weight is conserved in expectation, so the tally means are unchanged and only the variance is reduced.

Weight windows are applied at collisions and run on the CPU only (`compute='gpu'` refuses a model carrying them). They compose with the rest of the `variance_reduction` list; see Models for that list and for survival biasing.

Generating windows automatically (DeGVR)

The usual workflow is to generate the windows from a cheap pre-pass rather than write bounds by hand. `model.generate_weight_windows` runs the density-extrapolation method (DeGVR): two short transport passes at reduced material density, then a per-(voxel, group) extrapolation to full density. It returns a `WeightWindowBounds` you pass into a production run.

```
import yamc

mesh = yamc.RegularRectangularMesh(
    lower_left=[-1100, -1100, -1100], upper_right=[1100, 1100, 1100], shape=[50, 50,
50]
)
tally = yamc.Tally(scores=["flux"], name="flux", mesh=mesh)
model = yamc.Model(geometry=geometry, source=source, tallies=[tally])

# 1. Generate windows on the same mesh you want to resolve. Each pass stops
#    once the window mesh is converged, so no particle budget is needed.
windows = model.generate_weight_windows(
    yamc.WeightWindowGeneratorDeGVR(mesh=mesh)
)

# 2. Run production transport with the windows applied.
prod = yamc.Model(
    geometry=geometry, source=source, tallies=[tally], variance_reduction=[windows]
)
results = prod.simulate_transport(total_particles=2_000_000)
```

Each pass stops at the first of three conditions: the window mesh converged (`relative_error`, default 0.1), `total_particles` or `max_runtime`. The mesh counts as converged once 95% of the voxels that have scored, in every energy group and particle field, have a relative error at or below `relative_error`.

A voxel reached for the first time has an error near 1, so the pass keeps running while the scored region is still growing into the shield; voxels that never score (outside the geometry) do not count. The same target applies to both passes. Pass `relative_error=None` to stop on the budgets alone. The model's own `convergence_targets` do not apply: they watch the model's tallies, which the generation passes do not carry.

The same per-voxel metric is available to any run as a convergence target, `yamc.ConvergenceTarget("bin_relative_error", 0.05, tally="flux")`, which ends a mesh-tally run once 95% of its scored voxels reach 5% (CPU only).

Generation has a cost, so for a fair efficiency comparison charge its wall-clock time against the weight-window budget: time the generation and run the production pass for the remaining time (see `max_runtime`).

```
import time

t0 = time.time()
windows = model.generate_weight_windows(
    yamc.WeightWindowGeneratorDeGVR(mesh=mesh)
)
t_gen = time.time() - t0

prod = yamc.Model(geometry=geometry, source=source, tallies=[tally],
variance_reduction=[windows])
results = prod.simulate_transport(total_particles=10_000_000_000, max_runtime=(20,
"min") )
# analog baseline for the same total wall-clock, including generation:
analog = model.simulate_transport(total_particles=10_000_000_000,
                                  max_runtime=t_gen + results.elapsed)
```

`WeightWindowGeneratorDeGVR` parameters (all optional except `mesh`):

parameter	default	meaning
`mesh`	--	the regular mesh the windows are defined on
`particle`	`"neutron"`	particle type the windows act on
`energy_bins`	`None`	energy group edges (eV); `None` is a single group
`density_reduction`	`None` (auto)	reduced-density factor `N` for the extrapolation; `None` computes it from a particle-free optical-depth ray-trace (source to deepest voxel), `N = tau / 7` (continuous, clamped `>= 2`). Pass a float to override.
`ratio`	`5`	upper/lower bound ratio (window width)
`survival_factor`	`3`	sets the roulette survival weight
`max_split`	`10`	most copies one split may create
`weight_floor`	`1e-38`	absolute weight below which a particle is killed

By default `density_reduction` is auto: `N` is derived, before the two generation passes, from the problem's optical thickness `tau` (a particle-free ray-trace from the source to the deepest mesh voxel at the source energy) as `N = tau / 7` (continuous, clamped to `>= 2`; `tau` is taken as a high percentile of the per-voxel optical depth so a single oblique corner voxel does not skew it). The empirical sweet spot `tau / N ~ 7` mean free paths balances fiducial penetration against extrapolation error: a smaller `N` leaves the fiducial too thick to resolve the deep region, a larger `N` over-splits on a noise-amplified `G = F*(H/F)^(N-1)` extrapolation. `model.estimate_density_reduction(gen)` returns the `(N, tau)` it would pick without running generation. Pass an explicit float to `density_reduction` to override.

Applying windows directly

If you already have bounds (e.g. from a previous generation, via `windows.lower_bounds` / `windows.upper_bounds`, or an analytic importance map) you can build a `WeightWindowBounds` yourself. Bounds are flat, indexed `group * num_voxels + voxel`; pass `lower_bounds` and either `upper_bounds` or a `ratio` that fills them in. A negative lower bound marks a "no window" voxel that is skipped.

```
windows = yamc.WeightWindowBounds(
    mesh=mesh,
  lower_bounds=[1.0] * mesh.num_bins,  # one energy group, one bound per voxel
    upper_bounds=None,          # or omit and pass ratio=
    ratio=5.0,
    particle="neutron",
    survival_factor=3.0,        # > 1
    max_split=10,
    weight_floor=1e-38,
)
```

Photon sources

Weight windows are per particle type: set `particle=` on both the generator and the bounds, and the collision handler applies each window only to matching particles. For a photon source (`PhotonSource`), build the window with `particle="photon"` and run as above. The photon transport path splits and roulettes photons at their collisions and the tally means stay unbiased.

```
src = yamc.PhotonSource(energy=2e6)
model = yamc.Model(geometry=geom, source=src, tallies=[flux],
                   transport_secondary_photons=True)
photon_ww = model.generate_weight_windows(
    yamc.WeightWindowGeneratorDeGVR(mesh=mesh, particle="photon"))
prod = yamc.Model(geometry=geom, source=src, tallies=[flux],
                  transport_secondary_photons=True, variance_reduction=[photon_ww])
results = prod.simulate_transport(total_particles=200_000)
```

Secondary and decay photons (coupled)

Photons produced inside the run (secondary photons from neutron collisions with `transport_secondary_photons=True`, and D1S decay photons with `use_decay_photons=True`) travel the same photon transport path, so a `particle="photon"` window acts on them at their collisions too.

To generate a neutron window and a photon window for a coupled neutron-gamma problem in a single call, pass a list of particle types. One coupled pair of reduced-density passes tallies both fields and returns one window per particle, in the requested order; apply them together:

```
mesh = yamc.RegularRectangularMesh(lower_left=[-3, -3, 0], upper_right=[3, 3, 40],
                                   shape=[1, 1, 8])
src = yamc.NeutronSource(position=(0, 0, 0.5), energy=14.1e6)
model = yamc.Model(geometry=geom, source=src, tallies=[n_flux, p_flux],
                   transport_secondary_photons=True)

neutron_ww, photon_ww = model.generate_weight_windows(
    yamc.WeightWindowGeneratorDeGVR(mesh=mesh, particle=["neutron", "photon"]))

prod = yamc.Model(geometry=geom, source=src, tallies=[n_flux, p_flux],
                  transport_secondary_photons=True,
                  variance_reduction=[neutron_ww, photon_ww])
results = prod.simulate_transport(total_particles=200_000)
```

A single-string `particle` still returns one `WeightWindowBounds` (unchanged); a list returns a `list` in the same order. See `examples/python/coupled_np_degvr.py`.

For a photon window, the auto `density_reduction` ray-trace uses a photon energy rather than the driving 14 MeV neutron energy (photon attenuation at 14 MeV is nothing like at ~1 MeV). For neutron-driven secondary / decay photons it defaults to ~1 MeV, a typical gamma energy; override with `photon_energy=` on the generator if your spectrum is harder or softer. This only affects generation efficiency, never bias.

D1S decay photons

A shutdown-dose (D1S, `use_decay_photons=True`) run is a coupled run whose photon field is the decay photons emitted at activation sites, so the same `particle="photon"` (or coupled `["neutron", "photon"]`) generation produces a window that resolves the deep decay-photon dose. Tally the decay dose as usual with `parent_nuclides=model.radionuclides()` (which, like the D1S run itself, needs `yamc.transmutation_decay_data` and `yamc.transmutation_reactions` set, see Transmutation data); the window shapes transport, and the pulse-schedule time-correction is applied afterwards unchanged (it is a linear post-scaling and does not interact with the window). See `examples/python/decay_photon_degvr.py`.

One caveat: reducing material density for the DeGVR passes lowers the activation rate as well as the attenuation, so the decay-photon source scales with density and the log-linear-in-density extrapolation is more approximate than for pure attenuation. The window stays unbiased regardless (windows are unbiased for any bounds); only its efficiency is affected. Use a modest generation budget (the D1S activation path runs twice at reduced density).

Notes

- With or without windows the tally means agree within statistics; only the variance (and the run cost) differ. Judge the benefit by coverage and the per-tally relative error in the deep region, or the per-tally FOM.

- Windows help most under exponential attenuation (thick shields, many decades of flux drop), where analog cannot reach at all. For pure void streaming (an air duct), analog already fills the domain and the gain is smaller.
- A weight-window history can be expensive (it can process thousands of split copies), so a timed run stops on `max_runtime` at the end of a chunk and may overrun the budget by up to one chunk; read the run's measured `elapsed`.
- Both applying and generating windows are CPU only. `compute='gpu'` rejects a model carrying weight windows, and `generate_weight_windows` takes a `compute` argument that accepts only `'cpu'` (the default). Generation is ordinary transport under a window, so it cannot move to the GPU before the kernels honour windows at all; asking for it raises `ValueError` saying so.

Simulate transport

Running the Simulation

model.simulate_transport()

```
results = model.simulate_transport()
```

`simulate_transport()` returns a `TransportResults` object containing a `TallyResult` for every tally attached to the model, plus run metadata (`elapsed`, `transport_time`, `particles_per_second`, and `runs`: per run the seed, `n_histories`, the data libraries and the published release of each, `data_releases`).

A run stops on a particle budget (`total_particles`), a wall-time budget (`max_runtime`), or convergence targets on the model. They are OR-combined, so the run ends at the first one satisfied. `total_particles` defaults to `None` (no particle cap).

With none of them set, the run goes on until it has converged (see Running until converged):

```
results = model.simulate_transport()  # stops once every bin passes the statistical
checks
```

Accessing tally results

A `TallyResult` can be looked up four ways: by the `Tally` object, by name, by id, or by iteration. Names and ids are optional; set them on the `Tally` constructor to look results up that way.

```
flux_tally = yamc.Tally(scores=["flux"], cells=cell, name="flux", id=1)
heating_tally = yamc.Tally(scores=["heating"], cells=cell)

model = yamc.Model(
    geometry=geometry,
    source=source,
    tallies=[flux_tally, heating_tally],
)
results = model.simulate_transport()

# By Tally object (always works)
flux_result = results[flux_tally]

# By name (requires name= on the Tally)
flux_result = results["flux"]

# By id (requires id= on the Tally)
flux_result = results[1]

# Iterate over (tally, result) pairs
for tally, result in results.items():
    print(tally.name, result.mean)

# Membership test
assert flux_tally in results
assert "flux" in results
```

A `TallyResult` exposes `mean`, `standard_deviation`, `relative_error`, `variance`, `shape`, `dim_labels`, and `to_numpy()`, plus a set of statistical-reliability metrics and a PASS/FAIL verdict that print whenever you display a result (see Statistical reliability and Results below).

Running until converged

A run with no `total_particles`, `max_runtime` or convergence targets stops once **every scored bin of every tally** passes the statistical checks at two checkpoints in a row. Each bin is judged as if it were a single-bin tally: from its own mean, error and history, so a mesh or spectrum tally converges bin by bin, not just in its total. Bins that never scored (a region no particle reaches) are not waited for.

`convergence=` chooses how many checks must pass. Each level includes every check of the looser levels:

Check	`"loose"`	`"standard"` (default)	`"strict"`
relative error < 0.10	✓	✓	✓
mean stable (no drift)	✓	✓	✓
relative error decreasing	✓	✓	✓
relative error falls as 1/√N	✓	✓	✓
figure of merit stable	✓	✓	✓
figure of merit has no trend	✓	✓	✓
variance of variance < 0.10		✓	✓
variance of variance decreasing		✓	✓
variance of variance falls as 1/N		✓	✓
large-score tail slope > 3			✓
checks	**6**	**9**	**10**

```
results = model.simulate_transport()                        # "standard"
results = model.simulate_transport(convergence="loose")     # quickest; can stop early
on heavy-tailed scores
results = model.simulate_transport(convergence="strict")    # every check, every bin
```

Choosing a level:

- `"loose"` uses only each bin's mean and error. It is the quickest, but a rarely sampled large score can leave the relative error looking settled while it is not.
- `"standard"` adds the variance of the variance, which catches exactly that: an error estimate that is itself unreliable. It keeps two more numbers per bin per thread (16 bytes).
- `"strict"` adds the tail slope of each bin's score distribution, which needs each bin's 201 largest scores: about 1.6 KiB per scored bin per thread. That is fine for cell and spectrum tallies; for a mesh of a million voxels on 16 threads it is about 26 GB.

`convergence=` only applies when no other stop condition is set, and combining it with `total_particles`, `max_runtime` or convergence targets raises `ValueError`.

The checks are made at checkpoints every ~25% more histories, and the progress output names the level and how many checks the slowest bin passes, with an ETA projected from the slowest bin:

```
No stop condition set: running until every scored bin passes the standard statistical
checks (9).
Progress: 12500 particles  elapsed 1.2s, ETA ~3.4s (statistical checks 7/9 passing)
Every scored bin passed the standard statistical checks (9) after 51046 particles;
stopping.
```

Converging every bin can take a long time, for example deep in a shield without variance reduction. When, after a minute, the ETA is more than ten times the time already spent, a warning names the slowest tally bin and its relative error. Cancel the run, add weight windows, or set `total_particles`, `max_runtime` or a `ConvergenceTarget` to stop sooner.

Running until converged is CPU only and single process. On the GPU or under MPI, set a stop condition.

Progress output (`verbose`)

The `verbose` argument on `Model` is a list of independent flags selecting which categories of progress are printed during transport. Progress is reported in source particles (the run is chunked internally, but you never deal with batches). The flags are:

- `"progress"` : periodic `Progress: N/total particles (P%)` lines.
- `"eta"` : progress lines that also show elapsed wall-clock and ETA.
- `"tally"` : each tally's running mean ± std (rel. err) as it converges.
- `"summary"` : the end-of-run completion line and per-tally final statistics.
- `"nuclear_data"` : one line per nuclear-data file as it loads (nuclide and path), plus the transmutation chain file.

Any combination is allowed. The default (pass nothing, or `None`) is `["eta", "tally", "summary"]` . An empty list `[]` prints nothing, not even the end-of-run summary.

```
# Default output
model = yamc.Model(geometry=geometry, source=source, tallies=[tally])

# Fully silent, for scripts that post-process results themselves
model = yamc.Model(geometry=geometry, source=source, tallies=[tally], verbose=[])

# Just the final summary, no per-particle progress
model = yamc.Model(geometry=geometry, source=source, tallies=[tally], verbose=
["summary"])

# Read or change after construction
print(model.verbose)                       # ['eta', 'tally', 'summary']
model.verbose = ["progress", "summary"]
```

Threading

By default all available CPU cores are used. Specify a thread count explicitly:

```
model.simulate_transport(threads=8)
```

GPU transport (`compute="gpu"`)

The transport loop has a GPU implementation that runs on Vulkan via cubecl. Select it with `compute="gpu"` on `simulate_transport`:

```
# Check that a usable GPU is present at module level.
if yamc.parallel.gpu_available():
    results = model.simulate_transport(total_particles=1_000_000, compute="gpu")
else:
    results = model.simulate_transport(total_particles=1_000_000)
```

On a machine with more than one GPU (e.g. an integrated GPU plus a dedicated one), `compute="gpu"` auto-selects an adapter (preferring a discrete GPU via the high-performance power preference). To pin a specific adapter, list the available ones with `yamc.parallel.list_gpu_adapters()` and pass its name directly as `compute=<name>`:

```
adapters = yamc.parallel.list_gpu_adapters()
print(adapters)
# e.g. ['NVIDIA GeForce RTX 4070', 'Intel(R) UHD Graphics']

# Pin a specific GPU by name (e.g. the dedicated one):
results = model.simulate_transport(total_particles=1_000_000, compute=adapters[0])
```

`yamc.parallel.gpu_available()` returns `True` only when a GPU with f64 compute support is reachable. The kernel uses double-precision throughout (cross-section interpolation, sampling, and the variance accumulator), so single-precision-only consumer GPUs are rejected.

The GPU path supports a defined subset of the options, verified to match the CPU path on real discrete hardware (NVIDIA RTX A500): `cpu_gpu_equivalence_two_materials` is bit-exact and the `gpu_*.rs` GPU-vs-CPU suite passes. It currently supports:

- Neutron transport, primary-photon transport (a `PhotonSource`), coupled neutron-to-photon secondary production (`transport_secondary_photons=True`) and D1S decay photons (`use_decay_photons=True`). D1S on a mixed neutron and photon source is rejected.
- Tallies with either estimator: flux, heating / heating-local, damage-energy, reaction rates (total / elastic / absorption / (n,γ) / inelastic / fission), gas-production, and photon cross-section scores. Tally covariance (`covariance=True`) and virtual-overlay tallies (`response=`) are rejected.
- Tally filters: `cells=` , `materials=` , `mesh=` , `energy_bins=` , `particle=` , `nuclides=` , `parent_nuclides=` , and `energy_function=` / `dose_coefficients=` . The energy function weights

rather than bins, so it still needs a spatial binner alongside it; its interpolation table (cubic, linear or log-log) is built on the host and evaluated in the kernel, so cubic and linear match the CPU bit for bit and log-log to about an ulp.

- Full neutron physics: elastic (tabulated angular), inelastic (Kalbach-Mann, n-body, evaporation), free-gas thermal, URR, and the photon interactions on the photon path.
- CSG geometry only (mesh is rejected), with every CSG surface type.
- Survival biasing is supported. Weight windows are rejected, not silently ignored.
- The kernels always surface-track: a `tracking_mode` of `"woodcock"` or `"hybrid"` gives a warning and runs with surface tracking.

If any unsupported option is used, `compute="gpu"` raises a `ValueError` at simulate-transport start with a specific message. If no GPU with f64 compute is available, it raises `RuntimeError`. Passing `threads`, `capture_tracks` or `data_uncertainty` with `compute="gpu"` raises `ValueError` (the kernel parallelises across all particles in one launch, and tracking and nuclear-data uncertainty are CPU-only). Repeated `compute="gpu"` calls in one process are safe; each resets the shared GPU client.

For the full, always-current per-option breakdown of which CPU options the GPU path supports (and which it rejects), see the CPU-vs-GPU option matrix harness `crates/yamc/tests/gpu_cpu_comparison_matrix.rs`, which is the source of truth.

MPI (distributed memory)

For multi-node parallelism, build YAMC with MPI support (see installation).

```
# Check MPI support
if yamc.parallel.has_mpi():
    print(f"Rank {yamc.parallel.mpi_rank()} of {yamc.parallel.mpi_size()}")
```

Run with `mpirun`:

```
mpirun -np 4 python simulation.py
```

Hybrid MPI + threading:

```
model.simulate_transport(total_particles=10000, threads=8)  # 8 threads per MPI rank
```

```
# 4 ranks x 8 threads = 32 cores
mpirun -np 4 python simulation.py
```

yamc also runs inside an MPI job that another code started, such as a coupled solve with mpi4py, FEniCSx or preCICE. If MPI is already initialised when yamc needs it, yamc joins it on the world communicator and leaves finalising it to the host:

```
from mpi4py import MPI  # the host initialises MPI

import yamc

results = model.simulate_transport(total_particles=1_000_000)  # every rank of
MPI.COMM_WORLD
```

Tallies are combined across ranks once, at the end of the run: rank 0 folds every rank's statistics in rank order and broadcasts them, so every rank returns the same result. Read and save results on rank 0 (check `yamc.parallel.mpi_rank() == 0`); `yamc.combine_results` refuses a result from any other rank, since combining two ranks' copies would count the run twice. The tally-level reliability statistics (the `aggregate_*` values, the score PDF and the checks built on them) are not gathered across ranks yet, so they are reported for single-process runs only.

Each history's random numbers depend only on the base seed and its global index, not on the thread, chunk or rank that runs it, so the same histories run whatever the rank count; merged statistics can differ in the last bits, since the fold order changes with it.

Slurm example:

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=32
srun python simulation.py  # In script:
model.simulate_transport(total_particles=1_000_000, threads=32)
```

Particle tracking

Track particle histories for debugging or visualization. Tracks are off by default (zero overhead) and are only captured when `capture_tracks=` is set on `simulate_transport()`. Histories are addressed by their global 0-based index, so the captured set is identical regardless of thread count:

- `capture_tracks=N`: the first `N` histories (recommended for debugging)
- `capture_tracks=range(20, 30)`: histories 20..29 (half-open); strides work too, e.g. `range(0, 1000, 10)` for every 10th of the first 1000
- `capture_tracks=[20, 30, 45]`: exactly those history indices
- `capture_tracks='all'`: every history (warns; high memory on large runs)
- `capture_tracks=None` / omitted: no tracking

When tracking is enabled the captured `Tracks` are attached to the returned `TransportResults` as `results.tracks`; otherwise `results.tracks` is `None`.

A captured history holds one track per particle transported in it: the source particle and every secondary it produced (neutrons from (n,2n) and fission, and photons with

`transport_secondary_photons=True`), each with a `particle_type` of `"neutron"` or `"photon"`. A track is identified by its `history` and its `particle_id`, which is 0 for the source particle and numbers the secondaries within the history the same way on any thread count. `parent_id` and `generation` give each secondary's lineage. `len(tracks)` is the number of particle tracks, which is more than the number of histories captured once secondaries are made: capturing 5 histories of 14 MeV neutrons in an iron sphere with secondary photons on gives 71 tracks.

```
results = model.simulate_transport(total_particles=10000, capture_tracks=100)
tracks = results.tracks  # None if capture_tracks= was not set
assert tracks is not None

print(len(tracks))            # number of particle tracks (source and secondaries)
print(tracks.total_events())  # total events across all tracks

# Iterate over individual particle tracks
for track in tracks.tracks[:3]:  # first 3 particles
    print(f"History {track.history}, {track.particle_type} {track.particle_id} "
          f"(parent {track.parent_id}, generation {track.generation}):")
    for event in track.events[:5]:  # first 5 events
        print(f"  {event.event_type:12s}  E={event.energy_in:.4e} ->
{event.energy_out:.4e} eV"
              f"  pos=({event.x:.2f}, {event.y:.2f}, {event.z:.2f})"
              f"  cell={event.cell_id}  nuclide={event.nuclide}  MT=
{event.reaction_mt}")

# Convert to records for pandas
import pandas as pd
df = pd.DataFrame(tracks.to_dataframe_records())
print(df[["particle_type", "event_type", "energy_in", "energy_out", "x", "y", "z",
"nuclide"]].head(10))
photons = df[df.particle_type == "photon"]
```

Each `TrackEvent` includes: `particle_type`, `event_type`, `position`, `direction`, `energy_in`, `energy_out`, `weight`, `cell_id`, `reaction_mt`, `nuclide`, and `birth_reaction`. A track starts with a `source_birth` or a `secondary_birth` event, and the other event types are `collision`, `surface_crossing`, `absorption`, `russian_roulette` and `leak`.

Viewing tracks in ParaView

`tracks.to_vtkhdf()` writes one polyline per particle track to a VTK-HDF file for ParaView (needs `h5py`, from `pip install "yamc[viz]"`). The coordinates are in cm, as in `model.to_vtkhdf()`, so the tracks overlay the geometry when both files are opened together.

```
tracks.to_vtkhdf("tracks.vtkhdf")
model.to_vtkhdf("model.vtkhdf")  # the geometry and source to show them in
```

Each point carries `energy` (eV, after the event), `weight`, `event_type` and `cell_id` (-1 where unknown), and each line carries `particle_type`, `particle_id`, `parent_id` (-1 for a source particle), `generation` and `history`, so the tracks can be coloured by any of them. `event_type` and `particle_type` are stored as integer codes, which ParaView can colour by; the names they stand for

are in the file's root attributes `event_type_names` and `particle_type_names`. `tracks.polylines()` returns the same arrays as a dict, and `yamc.vtkhdf.polylines_to_vtkhdf(filename, points, offsets, point_data=..., cell_data=...)` writes any set of polylines in the same format.

Lost particles and geometry debugging

If a particle crosses into a region not covered by any cell, or reaches an unbounded cell with no surface ahead of it (a missing vacuum boundary), it is "lost". YAMC prints diagnostics, kills the particle and counts it towards `max_lost_particles`.

On a triangulated `MeshGeometry` a particle is also lost at a point the mesh cannot place (a stray or flipped facet makes the test rays disagree), and where it leaves a volume through a crack in the volume's surface (see Mesh geometry). A defective mesh therefore shows up as particles lost at the defect, rather than as particles carried on as void and leaking out uncounted.

```
model = yamc.Model(
    geometry=geometry,
    source=source,
    max_lost_particles=10,  # abort after this many (default 10)
)
model.simulate_transport(total_particles=1000)

# Inspect lost particles
for lp in model.lost_particles:
    print(f"Position: {lp.position}, Surface: {lp.surface_id}")
```

Transmutation

Model.simulate_transmutation()

Coupled transport-transmutation runs transport at each timestep with updated material compositions. Materials must have `transmutable=True` and a `volume` set.

```
yamc.neutron_cross_section_data = "endf-b8.1"

_chain = "transmutation-endf-b8.1-sfr.arrow"
yamc.transmutation_decay_data = _chain
yamc.transmutation_reactions = _chain
yamc.transmutation_fission_yields = _chain

model = yamc.Model(geometry=geometry, source=source)

schedule = yamc.PulseSchedule([
    yamc.Pulse(rate=1e14, duration=(1, "d"), source=source),   # irradiation, rate in
n/s
    yamc.Pulse(rate=1e14, duration=(1, "d"), source=source),   # irradiation
    yamc.Cooldown(duration=(1, "d")),                          # cooling
])
results = model.simulate_transmutation(
    method="coupled",                    # or "independent"
    schedule=schedule,
    total_particles=5000,
)
```

The irradiation/cooling timeline is one `PulseSchedule` of `Pulse` / `Cooldown` steps, the same type used for the shutdown-dose-rate post-processing below. Each `Pulse` carries a duration (seconds, or a `(value, unit)` tuple) and a source `rate` in n/s; each `Cooldown` is decay-only. Transport uses the source configured on the `Model`, so a schedule with more than one distinct `Pulse` source is rejected.

Methods: - `"coupled"` : transport runs every step with updated compositions - `"independent"` : transport runs once; reaction rates scaled by source rate per step (faster for low-burnup / fusion)

`simulate_transmutation` accepts a `compute` argument for symmetry with `simulate_transport`, but `"cpu"` (the default) is the only value it takes. The transmutation solve has no GPU backend, and the per-step transport would have to hand its rates back to a CPU solve every step even if it ran on the device. Any other value raises a `ValueError` explaining this (not a `TypeError` about an unexpected keyword).

Continuous feeds, extraction and transfer

Pass a `Processing` object to `simulate_transmutation()` to change inventories continuously during irradiation and cooling. Every material it names must be a transmutable material in the model, with its

own volume. This example transfers tritium from a lithium region to a surrounding collection region and feeds new lithium during the first schedule step:

```
import math

feed_material = yamc.Material(
    composition={"Li": 1.0}, density=0.534,
    volume=4 * math.pi * 10**3 / 3, transmutable=True,
)
collection_material = yamc.Material(
    composition={"Fe": 1.0}, density=7.87,
    volume=4 * math.pi * (20**3 - 10**3) / 3, transmutable=True,
)
inner_surface = yamc.Sphere(radius=10)
outer_surface = yamc.Sphere(radius=20, boundary="vacuum")
processing_geometry = yamc.Geometry([
    yamc.Cell(region=inner_surface.below, material=feed_material),
    yamc.Cell(region=inner_surface.above & outer_surface.below,
              material=collection_material),
])
processing_model = yamc.Model(geometry=processing_geometry, source=source)
processing = yamc.Processing()
processing.add_transfer(
    material=feed_material, component="H3", rate=1e-4,
    rate_units="1/s", destination=collection_material,
)
processing.add_feed(
    material=feed_material, composition={"Li6": 0.6, "Li7": 0.4},
    rate=1e-3, rate_units="g/s", steps=[0],
)
processed = processing_model.simulate_transmutation(
    method="independent", schedule=schedule, total_particles=5000,
    processing=processing,
)
```

A transfer rate is a first-order removal rate, not a mass flow. Omit `destination` to extract material from the system, omit `component` to transfer the whole composition, or name an element to select all its isotopes. Feed compositions are normalized atom fractions, with absolute units such as `g/s`, `kg/d`, `mol/h` or `atoms/s`. `steps` selects zero-based schedule steps; without it, a term also applies during cooldowns. The same object works with `method="coupled"` and with the standalone batch function `yamc.transmute()`.

Cooling curves

A decay curve needs points at cumulative times, but `Cooldown` takes the duration of its own step, so writing one by hand means differencing the times first. `cooldown_steps` does that:

```
schedule = yamc.PulseSchedule(
    [yamc.Pulse(rate=1e14, duration=(1, "d"), source=source)]
    + yamc.cooldown_steps((1, "h"), (10, "a"), 12)
)
```

Twelve cooling points from one hour to ten years after shutdown. The step durations sum to `stop` exactly: the first one runs from the end of the irradiation to `start`, rather than the steps spanning `stop - start`.

Log spacing is the default because decay is exponential. A linear sweep wastes steps in the flat tail and misses the early fall. Activity and decay heat are plotted on log-log axes, so points evenly spaced on the plot are log-spaced in time. `spacing="linear"` gives evenly spaced times, as in a fixed-interval measurement campaign.

`start` and `stop` take the same forms as a `Cooldown` duration: seconds, or a `(value, unit)` tuple. `n` is the number of points and must be at least 2. Log spacing needs a positive `start`, since the first point is a ratio from it and zero has no logarithm.

TransmutationResults

Results are keyed by each transmutable material's `id`. Below, `material` is the transmutable material in `geometry`.

```
# Evolution of a nuclide over all timesteps
evolution = results.get_nuclide_evolution(material_id=material.id, nuclide="Co60")
print(results.times)  # cumulative times including t=0

# Density at a specific step
density = results.get_nuclide_density(material_id=material.id, nuclide="Co60",
step=2)

# Full material at a step. The getters answer None for a material_id or step
# the run does not have, so a real script checks before using the result.
mat = results.get_material(material_id=material.id, step=2)
if mat is not None:
    print(mat.nuclides)

# All nuclides at a step
nuclides = results.get_material_nuclides(material_id=material.id, step=2)
```

Gas production and data-coverage diagnostics are available on the same results:

```
gas = results.get_gas_production(material_id=material.id)
if gas is not None:
    print(list(zip(results.times, gas["H"], gas["He"])))
screening = results.get_product_screening_info(material_id=material.id)
if screening is not None:
    print(screening["loaded"], screening["below_threshold"],
screening["unavailable"])
print(results.data_gaps)
```

Gas is in appm relative to the initial number of atoms, with the initial gas subtracted by default; `produced=False` instead reports all gas present. Unknown gas contributions are not silently zero-filled when the chain lacks a gas nuclide. Screening leaves a small product's production and decay in the

inventory, but may omit its neutron reactions; inspect `unavailable` and `data_gaps` before treating a small result as a complete prediction.

Saving results

`to_arrow()` writes `TransmutationResults` to an Arrow IPC file, and `yamc.TransmutationResults.from_arrow()` reads it back with every getter answering as before, to the bit. Without a path `to_arrow()` returns the file's bytes, for a browser or Pyodide session with no file system to write to.

```
results.to_arrow("steel.arrow")
results = yamc.TransmutationResults.from_arrow("steel.arrow")

data = results.to_arrow()                       # bytes
results = yamc.TransmutationResults.from_arrow(data)
```

A calculation too big to hold at once can be split into parts, each saved and dropped before the next. Every file has the same columns, one row per stored number (`material_id`, `material_name`, `step`, `quantity`, `nuclide`, `reaction`, `product`, `value`), so the parts read as one table without yamc, with `polars.scan_ipc("part_*.arrow")` for example.

The file holds the results, not the nuclear data behind them. Three getters read that data again: `get_production_routes` and `get_isomeric_branching` walk the transmutation chain, and `get_reaction_rate_spectrum` reads the initial composition's cross sections. The file pins both, the chain by a hash of its contents and each nuclide by the file it was read from, and `from_arrow()` reattaches them from the configured `yamc.transmutation_*` sources and those files. Where that data is not the same or is no longer there, it warns once, and those three getters raise rather than answer from other data.

Tallies at each step

The model's own tallies (`Model(tallies=[...])`) are scored by every step's transport, so anything that changes as the materials burn (TBR as Li6 depletes, heating, flux, spectra, mesh tallies) comes out of the one run, with no need to rebuild the model from `get_material` and run `simulate_transport` again:

```
tbr = yamc.Tally(name="tbr", cells=breeder, scores=["H3-production"])
model = yamc.Model(geometry=geometry, source=source, tallies=[tbr])
results = model.simulate_transmutation(method="coupled", schedule=schedule,
total_particles=5000)

for step, t in enumerate(results.timesteps):
    step_results = results.tally_results(step)   # a TransportResults, or None
    if step_results is not None:
        print(t, step_results[tbr].mean, step_results[tbr].standard_deviation)

means, std_devs = results.tally_evolution(tbr)   # one list of bins per transport
step
```

`tally_results(step)` returns the whole `TransportResults` of that step's transport, read exactly as `simulate_transport`'s are (by tally or name, `to_numpy`, mesh tallies included). `step` indexes `timesteps`, as `get_source_rates` and `get_reaction_rates` do: step `i` is the transport that drove the materials from composition `i` to composition `i + 1`. Each step's tallies are reset before its transport, so they reflect that step's compositions only.

- Values are per source particle, as `simulate_transport` returns them; multiply by `get_source_rates(material_id)[step]` for absolute rates.
- A decay-only step (a `Cooldown`, or a `Pulse` with rate 0) runs no transport and returns `None`; `tally_evolution` skips it.
- Only the model's own tallies come back. The transmutation's internal reaction rate tallies and its scouting transport are not among them.
- With `method="independent"` transport runs once, on the initial compositions, and every irradiation step returns that one run's results: they do not follow the burnup. Use `method="coupled"` for a TBR that falls as the Li6 burns.
- Every step keeps its own results, so a large mesh tally costs its size once per transport step.

Production routes from the chain

`TransmutationChain` carries the topology a solve steps through: which nuclide becomes which, and how. `reactions` holds the neutron-induced edges, `decays` the radioactive ones, both keyed by parent and both giving `(kind, target, branching)`:

```
chain = yamc.TransmutationChain.from_arrow("transmutation-endf-b8.1-sfr.arrow")

chain.reactions["W186"]   # [('(n,2n)', 'W185', 1.0), ..., ('(n,a)', 'Hf183', 1.0)]
chain.decays["Hf183"]     # [('beta-', 'Ta183', 1.0)]
chain.decays["W185_m1"]   # [('IT', 'W185', 1.0)]
```

Stable nuclides have no decay modes and are absent from `decays`, as they are from `half_lives`. A `target` of `None` means the channel names no single product: fission on the reaction side, a product outside the chain on the decay side. Decay modes carry the evaluation's own spellings (`"beta-"`,

`"ec/beta+"`, `"alpha"`, `"IT"`, `"sf"`, and emissions written as `"beta-,n"`), and the branchings for one parent sum to 1.

Because the two share a shape they can go in one edge list, which is what a route walk needs. With reactions alone a walk stops at the first product, one step short of most real routes:

```
edges = {}
for parent, es in chain.reactions.items():
    edges.setdefault(parent, []).extend(es)
for parent, es in chain.decays.items():
    edges.setdefault(parent, []).extend(es)

for kind1, mid, _ in edges["W186"]:              # two-step routes from W186 to Ta183
    for kind2, end, _ in edges.get(mid, []):
        if end == "Ta183":
            print(f"W186{kind1}{mid}({kind2}){end}")

# W186(n,3n)W184((n,np))Ta183
# W186(n,4n)W183((n,p))Ta183
# W186(n,a)Hf183(beta-)Ta183      <- the one that needs the decay edge
```

The branching on an edge is a share of its channel. It says how a channel splits and nothing about how often the channel fires, so these numbers alone cannot weight two routes into the same product against each other.

Per-edge reaction rates

The solve computes the rate of every individual production edge to build its burnup matrix, whose entries are sums over the edges into each product. `get_reaction_rates` returns those edges, so routes can be weighted as well as listed:

```
# None for a material_id or step the run does not have; `or {}` keeps the rest
# of the page working on a plain dict.
rates = results.get_reaction_rates(material_id=material.id, step=0) or {}   # step
indexes timesteps
rates["Fe58"]["(n,2n)"]        # [('Fe57', 1.205e-09)]            # rate in 1/s
```

`parent -> reaction kind -> [(target, rate [1/s])]`. A `target` of `None` is a channel naming no single product, which in practice means fission, whose products come from the yields rather than from an edge. A decay-only step drove no reaction and reports an empty map.

Here `step` is the schedule step, the same index as `timesteps` and `get_source_rates(material_id)`, which is one less than the `step` the composition getters take (where 0 is the initial composition).

Both methods report the same quantity in the same shape. `method="coupled"` takes the rates from that step's own transport tallies and `method="independent"` from the single transport's tallies, scaled by the step's source rate. Either way an edge rate is that rate times the branching of the chain the step was solved with, isomeric overlay included, so a consumer does not have to know which method produced the result.

The share of a product arriving down each route is the main column of a decay heat pathway analysis, and it is what makes a C/E useful for diagnosis. A foil reading 2.5x the measurement says something is wrong; knowing that 73% of the dominant product arrives down one channel says which evaluated quantity to check.

```
from collections import defaultdict

into = defaultdict(list)
for parent, kinds in rates.items():
    for kind, edges in kinds.items():
        for target, rate in edges:
            if target is not None:
                into[target].append((rate, parent, kind))

routes = sorted(into["Fe57"], reverse=True)
total = sum(rate for rate, _, _ in routes)
for rate, parent, kind in routes:
    print(f"{rate / total:6.1%}  {parent}{kind}Fe57   {rate:.4g} /s")
```

For a 14.06 MeV point source in a 10 cm iron sphere, one hour at 1e18 n/s:

```
 73.3%  Fe58(n,2n)Fe57       1.205e-09 /s
 18.3%  Co58(n,np)Fe57       3.016e-10 /s
  6.1%  Ni60(n,a)Fe57        1.009e-10 /s
  1.3%  Co58(n,d)Fe57        2.176e-11 /s
  0.6%  Fe56(n,gamma)Fe57    9.747e-12 /s
  0.3%  Ni61(n,na)Fe57       4.809e-12 /s
  0.0%  Co59(n,t)Fe57        6.359e-13 /s
  0.0%  Ni59(n,3He)Fe57      8.149e-15 /s
```

A rate is per atom of the parent, so weighting a route by inventory means multiplying by the parent's density at that step (`get_nuclide_density`). Over one short step the two together reproduce the step's own burnup: the Fe57 produced is the summed edge rate times the parent densities times the step length.

Uncertainty with the coupled method

`data_uncertainty` works with both methods, and the inventories, activity, decay heat and dose come back with a standard deviation as they do from Material.transmute(). With `method="independent"` the one transport's rates are resampled and each replica's schedule is solved again. With `method="coupled"` that would not describe the spread: each step's rates were tallied in compositions that the earlier steps' noise and data had already moved. Each replica therefore runs the whole coupled calculation again, every step's transport included, on its own draw:

- `"statistical"` : the replica has its own transport seed. Without this source every replica keeps the nominal seed, and the spread is the nuclear data's.
- `"cross_sections"` : every transport of the replica, in every material, reads nuclear data with the replica's MF=33 and resonance-parameter draw applied, so the draw moves the flux, the rates, the

compositions and every later step's flux. A product that enters transport part way through the run keeps its draw.

- `"isomer_branching"`: every step of the replica tallies its isomeric partials on production curves that carry the replica's MF=40 draw, and splits them at each collision energy by the nominal rule.
- `"half_life"`, `"decay_branching"` and `"fission_yield"` are used in each of the replica's solves, and `"decay_energy"` and `"decay_photon_lines"` move what is evaluated from its inventories.

Replica `k` draws the same data as replica `k` of the independent method, so the two can be compared replica by replica. Nothing is linearised, and the nominal results are bit-identical to a run without `data_uncertainty`. The default request, every source, runs. `"cross_sections"` is carried only beside `"statistical"`: a replica on its own cross sections follows its own histories whatever its seed, so its spread holds the transport's noise as well, and naming `"cross_sections"` alone raises. The independent method's replica weights do separate the two.

```
iron = yamc.Material(composition={"Fe": 1.0}, density=7.87, volume=4188.79,
                     transmutable=True)
dt = yamc.NeutronSource(energy=14.1e6)
sphere = yamc.Sphere(radius=10.0, boundary="vacuum")
model = yamc.Model(
    geometry=yamc.Geometry([yamc.Cell(region=sphere.below, material=iron)]),
    source=dt,
)
schedule = yamc.PulseSchedule([
    yamc.Pulse(rate=1e18, duration=(1, "h"), source=dt),
    yamc.Pulse(rate=1e18, duration=(1, "h"), source=dt),
    yamc.Cooldown(duration=(1, "h")),
])
results = model.simulate_transmutation(
    method="coupled",
    schedule=schedule,
    total_particles=2000,
    data_uncertainty=yamc.DataUncertainty(samples=16,
                                          sources=["statistical", "half_life"]),
)
mean = results.get_nuclide_density(material_id=iron.id, nuclide="Mn56", step=3)
sigma = results.get_nuclide_uncertainty(material_id=iron.id, nuclide="Mn56", step=3)
if mean is not None and sigma is not None:
    print(f"Mn56: {mean:.3e} +/- {sigma:.3e}  ({100 * sigma / mean:.1f}%)")
# Mn56: 5.681e-08 +/- 7.772e-10  (1.4%)
info = results.get_data_uncertainty_info(material_id=iron.id)
if info is not None:
    info["flux_response"]   # "coupled_replicas"
```

The cost is the transport of every irradiation step once per replica, the 16 replicas above taking about 7 seconds on 32 cores. Leave `samples` unset and replicas are added 64 at a time by the convergence rule, at least 128 of them, so ten irradiation steps of ten minutes' transport each come to about 210 hours. Each nuclide whose resonance parameters are sampled adds a few seconds per replica to rebuild its resonance range. `samples` bounds the cost.

`DataUncertainty(attribution=True)` reruns the ensemble once per applied source, each source drawn alone on the same streams, so it costs one more coupled ensemble per source. The `"cross_sections"` entry of `by_source` is the variance with `"statistical"` and `"cross_sections"` less the variance with `"statistical"` alone, the two sharing their transport seeds, and comes out negative where the cross sections move a nuclide by less than the transport's noise. There are no first-order `contributors`, and every `linearity` entry is `None`. On the sphere above:

```
attributed = model.simulate_transmutation(
    method="coupled",
    schedule=schedule,
    total_particles=2000,
    data_uncertainty=yamc.DataUncertainty(
        samples=16, sources=["statistical", "cross_sections"], attribution=True,
    ),
)
breakdown = attributed.get_uncertainty_breakdown(material_id=iron.id, nuclide="Mn56",
step=3)
if breakdown is not None:
    breakdown["variance"]    # 3.44e-18, Mn56 at 3.3%
    breakdown["by_source"]   # {"cross_sections": 2.84e-18, "statistical": 6.04e-19}
```

The statistical entry is the square of the first example's Mn56 sigma (7.77e-10), whose replicas ran on the same transport seeds, and the iron cross sections carry the rest. With the cross sections each replica rebuilds the resonance ranges of the iron isotopes for its draw, so the two ensembles of 16 replicas took about 200 seconds.

Statistical uncertainty of the tallied rates

Run `method="independent"` with `data_uncertainty` including the `"statistical"` source and `get_reaction_rate_uncertainty` gives each tallied reaction rate with its Monte Carlo standard deviation at one step:

```
uncertain = model.simulate_transmutation(
    method="independent",
    schedule=schedule,
    total_particles=5000,
    data_uncertainty=yamc.DataUncertainty(sources=["statistical"]),
)
for nuclide, reaction, target, rate, std_dev in (
    uncertain.get_reaction_rate_uncertainty(material_id=material.id, step=0) or []
):
    print(nuclide, reaction, target, rate, std_dev)   # 1/s per atom
```

A reaction total has a `target` of `None` and an isomeric partial names its final state. The rates are the tally's, scaled by the step's source rate, as the step's solve used them. They were scored by the same histories and so are correlated: the inventory sigmas carry those correlations, and these standard deviations alone do not.

Isomeric branching report

`get_branching_report(material_id, step)` says, per channel, how the isomeric-branching evaluation's split was applied over that step's spectrum: whether its list was read as shares of the total, as absolute productions or, for the light particles of `(n,X)` (MT=5), as multiplicities, and how much of the parent's removal rate rests on anything the evaluation does not give.

```
report = results.get_branching_report(material_id=material.id, step=0)
if report is not None:
    report["channels"]          # one dict per channel: parent, reaction, mt, file,
                                # representation ("share", "absolute" or
                                # "multiplicity"), states, ...
    report["dropped"]           # channels that were not folded, with the reason
    report["unmodelled_mt5"]    # [(nuclide, share, reason)]: parents whose MT=5
                                # residuals the chain does not model
```

A run refuses when the clipped or held production of its channels, each parent weighted by its density, is more than 0.1% of the material's neutron removal rate, and likewise when the material's own nuclides lose more than that to MT=5 residuals the chain does not model. Each channel's own shares are of its parent's removal rate, so a trace parent can read above 0.1% in a run that was not refused. The step is the schedule step, as `get_reaction_rates` takes it, and a decay-only step gives empty lists.

Material.transmute()

For standalone transmutation with a pre-computed flux spectrum (no transport), see Material transmutation in the Materials section.

Shutdown Dose Rate

After irradiation, activated materials keep emitting decay gammas. The dose rate once the neutron source is switched off is the shutdown dose rate. YAMC computes it from a single transport run. This puts it among the single-transport methods, which also include UKAEA's D1S (Direct-1-Step) and N1S (Novel-1-Step), as opposed to R2S (Rigorous-2-Step), which runs a separate activation step. Decay photons are coupled directly into the neutron transport run, replacing the prompt photons. YAMC's method derives from D1S but computes time correction factors (TCFs) automatically for every decay path that reaches a photon emitter and applies them in post-processing, so the irradiation and cooling schedule can change without re-running transport. R2S is available too, through `yamc.simulate_shutdown_dose(method="r2s")` : see Rigorous two-step (R2S).

Overview

1. Bin a photon tally by decay-photon emitter (`parent_nuclides=` the model's `radionuclides()`)
2. Run transport with `use_decay_photons=True`
3. Describe the irradiation/cooling timeline as a `PulseSchedule`
4. Call `schedule.time_correct_tally(...)` to get the dose rate(s)

Workflow

```python
import yamc

# A D1S run transports neutrons and photons, so it needs both cross-section settings.
yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"

_chain = "transmutation-endf-b8.1-sfr.arrow"
yamc.transmutation_decay_data = _chain
yamc.transmutation_reactions = _chain
yamc.transmutation_fission_yields = _chain

# 1. Bin a photon tally by decay-photon emitter.
#    parent_nuclides=... makes the tally score one bin per emitter, so each
#    emitter's decay photons can be scaled independently in post-processing.
#    The emitters come from the model's materials, so no schedule is needed
#    yet. Any spatial filter works: a mesh here, but
#    cells and materials work too.
mesh = yamc.RegularRectangularMesh.from_domain(geometry, shape=1_000_000)
radionuclides = yamc.Model(geometry=geometry, source=source).radionuclides()
dose_tally = yamc.Tally(
    scores=["flux"],
    particle="photon",
    parent_nuclides=radionuclides,
    mesh=mesh,
)

# 2. Run transport with decay photons
model = yamc.Model(
    geometry=geometry,
    tallies=[dose_tally],
    source=source,
    transport_secondary_photons=True,
    use_decay_photons=True,
)
results = model.simulate_transport(total_particles=1000000)

# 3. Describe the irradiation/cooling timeline. Each Pulse carries its own
#    source + rate (n/s) + (value, unit) duration; a Cooldown has no source.
schedule = yamc.PulseSchedule([
    yamc.Pulse(source=source, rate=1e14, duration=(1, "a")),   # 1 year irradiation
    yamc.Cooldown(duration=(30, "d")),                          # 30 days cooling
])

# 4. Time-correct the tally into dose rate(s). steps defaults to all schedule
#    steps; an int selects one (negative counts from the end). There is no
#    pre-irradiation baseline row to slice off.
dose = schedule.time_correct_tally(results[dose_tally], steps=-1)  # end of cooling
print(f"Dose rate at end of cooling: {dose.mean}")   # one value per mesh cell
print(dose.std_dev)                                  # matching uncertainty
print(dose.by_nuclide)                               # per-emitter breakdown
```

`radionuclides()` names an emitter a reaction makes directly as itself (`"Co60"`), and one made through a decaying parent by its decay path (`"Cs137>Ba137m"`), since the parent's half-life sets when its

photons arrive. An emitter made both ways is two bins, each with its own time correction, and `dose.by_nuclide` is keyed the same way.

For a single `int` `steps` the `DoseResult` arrays are 1-D; for the default (all steps) or a list of steps they are 2-D (one row per step), and `dose.times` gives the elapsed time at the end of each selected step. The flat bin layout per row is score then mesh (the general order is in Spatial binning), so reshape with `np.array(dose.mean).reshape(mesh.shape)`.

Nuclear-data uncertainty on the dose

`dose.std_dev` is the transport's statistical error alone. Pass `data_uncertainty` and the time correction's nuclear-data uncertainty comes back beside it:

```
dose = schedule.time_correct_tally(
    results[dose_tally],
    steps=-1,
    data_uncertainty=yamc.DataUncertainty(),
)
dose.data_std_dev            # from the nuclear data, same shape as dose.mean
dose.total_std_dev           # std_dev and data_std_dev in quadrature
dose.data_uncertainty_info   # what was sampled, what states no sigma, the
                             # replica count, the convergence target and
                             # whether every factor's sigma met it
```

Two sources act here. `"half_life"` acts through the time correction: a time-correction factor is an activity over the schedule, and the tally's photon yield is per decay, so the half-lives enter there and nowhere else. Each replica draws every half-life feeding an emitter once and uses it for every campaign. `"decay_photon_lines"` acts through each photon spectrum's normalisation (FD for lines, FC for a continuum), which scales an emitter's whole spectrum and so its tally, and is applied after the fact. The draws are those a transmutation with the same seed makes.
`dose.data_std_dev_correlated` and `dose.total_std_dev_correlated` give the upper end of the range left by the unstated correlation between an emitter's gamma and x-ray normalisations. The line intensities and energies change the spectrum's shape, which would need line-resolved tallies, and are held; ENDF/B-VIII.1 folds the normalisation's sigma into the line intensities, so under it almost nothing is drawn. Decay branching ratios also shape the dose, through the share of a parent's decays that feeds an emitter (which the photon yield carries, so a time correction is the time dependence alone), but are held at nominal here, and `data_uncertainty_info` lists everything held under `not_perturbed`. The two sigmas are independent (one comes from transport sampling, the other from the evaluated half-lives), so they add in quadrature. Without `data_uncertainty` the three attributes are `None`.

Mixed-spectrum (DD + DT) campaigns

A schedule can mix sources, for example a DT pulse, cooling, then a DD pulse. Each distinct spectrum activates the materials differently, so run transport once per source and pass `time_correct_tally` a dict mapping each source in `schedule.sources` to its run's tally result; the per-source corrections are summed exactly.

```
dt = yamc.NeutronSource(energy=yamc.sources.fusion_neutron_spectrum(ion_temp=20000.0,
reactants="DT"))
dd = yamc.NeutronSource(energy=yamc.sources.fusion_neutron_spectrum(ion_temp=20000.0,
reactants="DD"))

schedule = yamc.PulseSchedule([
    yamc.Pulse(source=dt, rate=1e18, duration=(1, "a")),
    yamc.Cooldown(duration=(1, "d")),
    yamc.Pulse(source=dd, rate=5e17, duration=(30, "d")),
    yamc.Cooldown(duration=(7, "d")),
])

# One transport run per source, the same model otherwise.
model_dt = yamc.Model(geometry=geometry, tallies=[dose_tally], source=dt,
                      transport_secondary_photons=True, use_decay_photons=True)
model_dd = yamc.Model(geometry=geometry, tallies=[dose_tally], source=dd,
                      transport_secondary_photons=True, use_decay_photons=True)
results_dt = model_dt.simulate_transport(total_particles=1_000_000)
results_dd = model_dd.simulate_transport(total_particles=1_000_000)
dose = schedule.time_correct_tally({dt: results_dt[dose_tally],
                                    dd: results_dd[dose_tally]})
```

A machine that changes between steps

The workflow above activates and transports in one geometry. A real machine changes: it is irradiated with DD neutrons in one configuration and with DT in another, components are added between campaigns, and during shutdown, maintenance and decommissioning they are removed or moved. `yamc.simulate_shutdown_dose` takes a schedule whose steps name the `Model` the machine is in, and does the transport itself.

- A `Pulse(model=...)` irradiates that model with its own neutron source (give `model` instead of `source`).
- A `Cooldown(model=...)` is the machine the decay photons are transported through while it cools. It needs no source.
- A step without a `model` stays in the previous step's.
- Cells are the same component across models when they have the same **name**, so name every cell that activates.
- A cell missing from a model has been removed: it emits nothing there, but its activation keeps decaying, and counts again if it comes back.
- A cell somewhere else is described by the step's `moves`, `{cell or name: (dx, dy, dz)}` in cm or a 4x4 rigid transformation matrix, relative to where the cell was built. The model must hold the cell at its moved position; each moved decay photon is checked to land inside it.

```
dd_phase = yamc.Model(geometry=reactor_without_plug, source=dd_source)
dt_phase = yamc.Model(geometry=reactor_with_plug, source=dt_source)
maintenance = yamc.Model(geometry=reactor_with_plug_pulled_out)  # no source needed
decommissioned = yamc.Model(geometry=reactor_without_plug)

schedule = yamc.PulseSchedule([
    yamc.Pulse(model=dd_phase, rate=1e16, duration=(30, "d")),
    yamc.Pulse(model=dt_phase, rate=1e17, duration=(30, "d")),
    yamc.Cooldown(duration=(1, "d")),                    # still the DT layout
    yamc.Cooldown(model=maintenance, duration=(7, "d"),
                  moves={"port plug": (150, 0, 0)}),       # plug pulled 150 cm out
    yamc.Cooldown(model=decommissioned, duration=(30, "d")),
])

# A photon tally, without parent_nuclides: it is binned by every emitter
# the irradiations make. Each tally needs a name; results are keyed by it.
worker = yamc.Tally(scores=["flux"], name="worker", mesh=mesh, particle="photon")
dose = yamc.simulate_shutdown_dose(schedule, tallies=[worker],
total_particles=1_000_000)
dose["worker"].mean        # one row per schedule step
dose["worker"].std_dev     # the whole calculation's standard error
dose["worker"].by_nuclide
dose["worker"].times
```

For every model a pulse irradiates in, the neutrons are transported once with decay photons on, and each decay photon is recorded where it is born instead of being tracked. Then, for each configuration the machine is observed in, each neutron history's decay photons, from every irradiated model, are moved with their cells (or dropped, for removed cells) and transported through the observing model together, as one photon history. That history's dose at the end of each step is formed from the time correction factors of each irradiating configuration's own pulses. A schedule with two irradiating and four observing configurations takes two neutron runs and four photon runs, however many steps it has, and every photon is transported once.

`std_dev` is the standard error of the whole calculation. It is the spread of the per-history doses, so it covers the neutron transport as well as the photon transport, and the correlation one neutron history carries between radionuclides and between irradiations. For a machine that never changes it equals the error of the single-model run above. There is nothing to tune: the photon runs have one history per neutron history. Runs on the CPU, on one process.

Rigorous two-step (R2S)

`yamc.simulate_shutdown_dose(method="r2s")` computes the dose with the rigorous two-step method (Chen and Fischer 2002) instead of the one-step method above. It takes the same schedule, whose steps name the `Model` the machine is in (see A machine that changes between steps), and does the transport itself:

1. One neutron transport of the irradiation model tallies the neutron spectrum of every material-filled cell.

2. Each cell's inventory is solved through the whole schedule with its own spectrum, so activation products that are themselves activated, and burnup, are included.
3. At the end of each step every cell emits its decay photons uniformly over its volume, in that step's model, and one photon transport per step scores the tallies. `by_nuclide` breaks the dose down by emitting radionuclide.

Cells are matched across models by **name**. A cell missing from a step's model has been removed and emits nothing there; a cell the model holds somewhere else emits where it now is. Every pulse must irradiate in the same model. A cell's volume is estimated by sampling unless `Cell(volume=...)` gives it.

A whole cell emitting uniformly is the method's weak point in a thick component, where the activation falls with depth. Pass a `RegularRectangularMesh` over the activated components as `mesh` and the source follows the flux instead (cell-under-mesh R2S): every part of a material-filled cell under a voxel is a region of its own, solved with that voxel's neutron spectrum (tallied in VITAMIN-J-175 groups) and emitting uniformly over that part. Cells do not have to be split. Below, a 40 cm iron bar is irradiated by 14 MeV neutrons from one end, and the photon flux is scored in a box beside each end:

```
iron = yamc.Material(composition={"Fe": 1.0}, density=7.874)
world = yamc.Sphere(radius=400.0, boundary="vacuum")
bar = (yamc.Plane("x", 0.0).above & yamc.Plane("x", 40.0).below
       & yamc.Cylinder("x", radius=2.0).below)
irradiation = yamc.Model(
    geometry=yamc.Geometry([
        yamc.Cell(region=bar, material=iron),
        yamc.Cell(region=~bar & world.below),
    ]),
    source=yamc.NeutronSource(position=(-0.5, 0, 0)),
)
schedule = yamc.PulseSchedule([
    yamc.Pulse(model=irradiation, rate=1e12, duration=(1, "h")),
    yamc.Cooldown(duration=(1, "h")),
])

def box(x):
    return yamc.RegularRectangularMesh(lower_left=[x - 5, -5, 5],
                                       upper_right=[x + 5, 5, 15], shape=[1, 1, 1])

tallies = [yamc.Tally(scores=["flux"], name="near", mesh=box(0), particle="photon"),
           yamc.Tally(scores=["flux"], name="far", mesh=box(40), particle="photon")]

cells = yamc.simulate_shutdown_dose(schedule, tallies=tallies,
                                    total_particles=4000, method="r2s")
voxels = yamc.simulate_shutdown_dose(
    schedule, tallies=tallies, total_particles=4000, method="r2s",
    mesh=yamc.RegularRectangularMesh(lower_left=[0, -2, -2],
                                     upper_right=[40, 2, 2], shape=[8, 1, 1]),
)
for name, dose in (("cells", cells), ("mesh", voxels)):
    near, far = dose["near"].mean[1][0], dose["far"].mean[1][0]
    print(f"{name}: near / far = {near / far:.2f}")
# cells: near / far = 0.79
# mesh: near / far = 41.60
```

With whole cells the bar emits evenly along its length, so the two ends read about the same. With eight 5 cm voxels along it, the source sits where the neutrons activated the iron, and the end facing the neutron source reads forty times the other. Fe56(n,p)Mn56 dominates both.

The parts' volumes are estimated by sampling each voxel, so a part smaller than about 0.2% of its voxel can be missed. Voxels no neutron reaches are not activated. With a mesh the pulses must irradiate with one set of `moves`, and a cell moved after irradiation needs its `moves` (`{cell or name: (dx, dy, dz)}` in cm or a 4x4 rigid transformation matrix, relative to where the cell was built), so that each voxel part goes with its cell; a part that does not land in its cell is an error. `mesh` is for R2S only and is refused with `method="d1s"`.

With R2S, `std_dev` is the photon transport's statistical error given the solved inventories, and does not include the neutron transport's. `total_particles` sets the neutron histories and each step's photon histories, so R2S runs one neutron transport and one photon transport per step.

Reference

- `model.radionuclides()` : the decay-photon emitters reachable from a Model's materials, each named as itself or, where made through a decaying parent, by its decay path (`"Cs137>Ba137m"`); pass to `Tally(parent_nuclides=...)` .
- `yamc.Pulse(rate, duration, source=None, model=None, moves=None)` : an irradiation campaign. `duration` is seconds or a `(value, unit)` tuple (units: s, min, h, d, a). `source` is the NeutronSource for transport and SDR. `Model.simulate_transmutation` may omit it and transports the model's source; `Material.transmute` needs one whose energy is a `Histogram` , and there `rate` is the total flux in n/cm^2/s.
- `yamc.Cooldown(duration, model=None, moves=None)` : a cooling step (no source). `model` and `moves` describe the machine during the step, for `simulate_shutdown_dose` .
- `yamc.simulate_shutdown_dose(schedule, tallies, total_particles, seed=1, threads=None, method="d1s", mesh=None)` : the dose over a schedule whose steps name models; returns `{tally name: DoseResult}` with one row per step. `method="d1s"` (the default) is the one-step method of A machine that changes between steps; `method="r2s"` , with `mesh` optional, is Rigorous two-step (R2S).
- `yamc.PulseSchedule([steps...])` : the timeline; `.sources` lists its distinct sources, `len(schedule)` its step count.
- `schedule.time_correct_tally(results, steps=None, data_uncertainty=None)` : returns a `DoseResult` with `.mean` / `.std_dev` / `.by_nuclide` / `.times` , and `.data_std_dev` / `.total_std_dev` / `.data_uncertainty_info` when `data_uncertainty` is given.

Parent-nuclide binning is set by the `parent_nuclides=` argument on `Tally` (step 1). No separate setup call is needed.

Results and post-processing

Tally results

`model.simulate_transport()` returns a `TransportResults` indexed by tally:

```
results = model.simulate_transport()
result = results[tally]
print(result.mean)                # list of mean values
print(result.standard_deviation)  # list of standard deviations
print(result.relative_error)      # list of relative errors
```

Statistical reliability

Every `TallyResult` also carries reliability statistics. They are computed automatically from a small per-history aggregate, so the runtime cost is negligible. Printing a result shows all of them: the mean ± std, the figure of merit, the higher-moment metrics, and a PASS/FAIL verdict:

```
result = results["flux"]
print(result)
# TallyResult 'flux'   shape=[1]   particles=1000000   elapsed=1.025 s
#   mean . . . . . . . 1.7667e1 +/- 1.21e-2   (0.07% rel err)
#   figure of merit .. 2.080e6
#   variance of var. . 7.277e-6   skewness 1.875   kurtosis 5.277
#   large-score tail slope .. 7.78
#   statistical checks: PASSED (11/11)
#     mean stable..................... pass
#     relative error < 0.10........... pass
#     per-bin relative error < 0.10..... pass
#     relative error decreasing......... pass
#     relative error falls as 1/sqrt(N). pass
#     variance of variance < 0.10....... pass
#     variance of variance decreasing... pass
#     variance of variance falls as 1/N. pass
#     figure of merit stable............ pass
#     figure of merit has no trend...... pass
#     large-score tail slope > 3........ pass
```

Per-bin array (same shape as `mean`):

Attribute	Meaning
`variance`	variance of the mean (`standard_deviation ** 2`)

Tally-level reliability scalars, computed from the per-history *total* score:

Attribute	Meaning
`aggregate_mean` / `aggregate_std_dev` / `aggregate_relative_error`	statistics of the integrated tally total
`aggregate_variance_of_variance`	relative variance of the variance; large values flag an unreliable error estimate (rare large contributions are undersampled)
`aggregate_skewness` / `aggregate_kurtosis`	shape of the history-score distribution
`aggregate_tail_slope`	slope of the large-score PDF tail; the variance is well-defined only when this exceeds 3

The `statistical_checks` object gives the pass/fail verdict, and `convergence_history` records the statistics versus number of histories:

```
checks = result.statistical_checks
checks.passed                 # bool: overall verdict, False if the tail slope could
not be estimated
checks.relative_error_ok      # individual checks: True / False / None (not
evaluated)
print(checks.summary())       # the pass/fail table and verdict: PASSED, FAILED or
INCOMPLETE

for point in result.convergence_history:
    print(point.n_histories, point.relative_error, point.figure_of_merit)
```

The raw empirical history-score PDF is available for inspection or plotting:

```
pdf = result.score_pdf
pdf.counts        # histogram counts per log-spaced magnitude bin
pdf.bin_centers   # bin centers (len == counts); bin_edges has one more
pdf.zero          # histories whose score was exactly zero
pdf.tail_slope    # the fitted large-score tail slope
```

Stopping on a precision target

A `ConvergenceTarget` ends a run early once a tally is precise enough. It works on the CPU (under MPI the decision is made on the rank-combined statistics) and on the GPU, where it is decided between kernel launches, so the run overshoots the target by at most one launch chunk:

```
model.convergence_targets = [yamc.ConvergenceTarget("relative_error", 0.02,
tally="flux")]
# the run stops once flux relative error <= 2% (or total_particles / max_runtime is
reached)
```

The metric is one of `"relative_error"`, `"std_dev"`, or `"variance_of_variance"`, all on the tally's total, or `"bin_relative_error"`, the relative error that 95% of the tally's scored bins meet (CPU only). Use `"bin_relative_error"` for a mesh tally, whose total converges long before its deep voxels do. Omit `tally=` to require the threshold for every tally.

Nuclear-data uncertainty diagnostics

When transport was requested with `data_uncertainty`, a tally exposes the nuclear-data spread separately from ordinary transport statistics. The attributes below are `None` on a run without that request:

```
sigma = result.nuclear_data_standard_deviation
if sigma is not None:
    print(sigma, result.nuclear_data_replica_spread, result.nuclear_data_noise)
    print(result.nuclear_data_noise_dominated)
    print(result.replica_mean, result.replica_standard_error)
    print(result.nuclear_data_linear_variance)
    print(result.nuclear_data_variance_by_nuclide)
```

`mean` remains the nominal calculation; `replica_mean` is the ensemble mean. Replica spread includes Monte Carlo noise, so it is not interchangeable with the noise-corrected nuclear-data sigma. The noise flag identifies bins where that separation is poorly resolved. Per-nuclide contributions describe the first-order variance, not every nonlinear contribution to the final spread. See Nuclear-data uncertainty for the sampling controls and coverage limitations.

Structured results

Each `TallyResult` exposes `shape` and `dim_labels` for reshaping into numpy arrays.

```
result = results[tally]
print(result.shape)       # e.g. (1, 2, 20, 20, 20)
print(result.dim_labels)  # e.g. ('score', 'energy', 'mesh_z', 'mesh_y', 'mesh_x')

import numpy as np
data = np.array(result.mean).reshape(result.shape)
```

xarray integration

With `xarray` installed (`pip install "yamc[xarray]"`), the built-in exporters retain the tally's dimensions and coordinates:

```
result = results[tally]
da = result.to_xarray(field="mean")
std_dev = result.to_xarray(field="std_dev")
dataset = result.to_dataset()

# Select a single energy bin
flux_high = da.isel(energy=1)
```

`to_dataset()` exports mean, standard deviation and relative error together. For manual array handling, `to_numpy()` and the flat arrays remain available.

Plotting mesh tallies

`tally.plot()` returns an interactive `InteractiveTallyPlot` that renders inline in Jupyter and saves as a self-contained HTML page with `.to_html("file.html")`. Without a path `to_html()` returns the page as bytes.

```
plot = tally.plot(geometry=geometry, basis="xz")
plot.to_html("flux_map.html")
```

Parameters:

Parameter	Type	Default	Description
`geometry`	Geometry or MeshGeometry	`None`	Optional outline overlay
`basis`	str	`"xy"`	Initial slice plane: `"xy"`, `"xz"`, or `"yz"`
`slice_coord`	float	mesh center	Initial slice coordinate (cm)
`slices`	None, list[int], or `"all"`	None	Which slices to embed for the in-browser slider. `None` = just the initial slice; `"all"` = every bin (larger HTML, full slider)
`score_index`	int	0	Which score (when the tally has multiple)
`energy_index`	int or None	None	Energy bin; `None` sums over energy
`value`	str	`"mean"`	`"mean"`, `"standard_deviation"`, or `"relative_error"`
`colorscale`	str	`"Viridis"`	Plotly colorscale name (`Viridis`, `Cividis`, `Hot`, etc.)
`log_scale`	bool	`True`	Logarithmic color scale
`outline`	str or None	`"material"`	Outline by `"material"`, `"cell"`, or `None`
`resolution`	int	400000	Total pixel budget for geometry-outline sampling
`axis_units`	str	`"cm"`	Axis units: `"mm"`, `"cm"`, `"m"`, or `"km"`
`contour_kwargs`	dict	None	Outline appearance: `{"colors": [...], "linewidths": int}`
`title`	str or None	None	Plot title (auto-generated if None)
`colorbar_title`	str or None	None	Colorbar label (auto-generated if None)

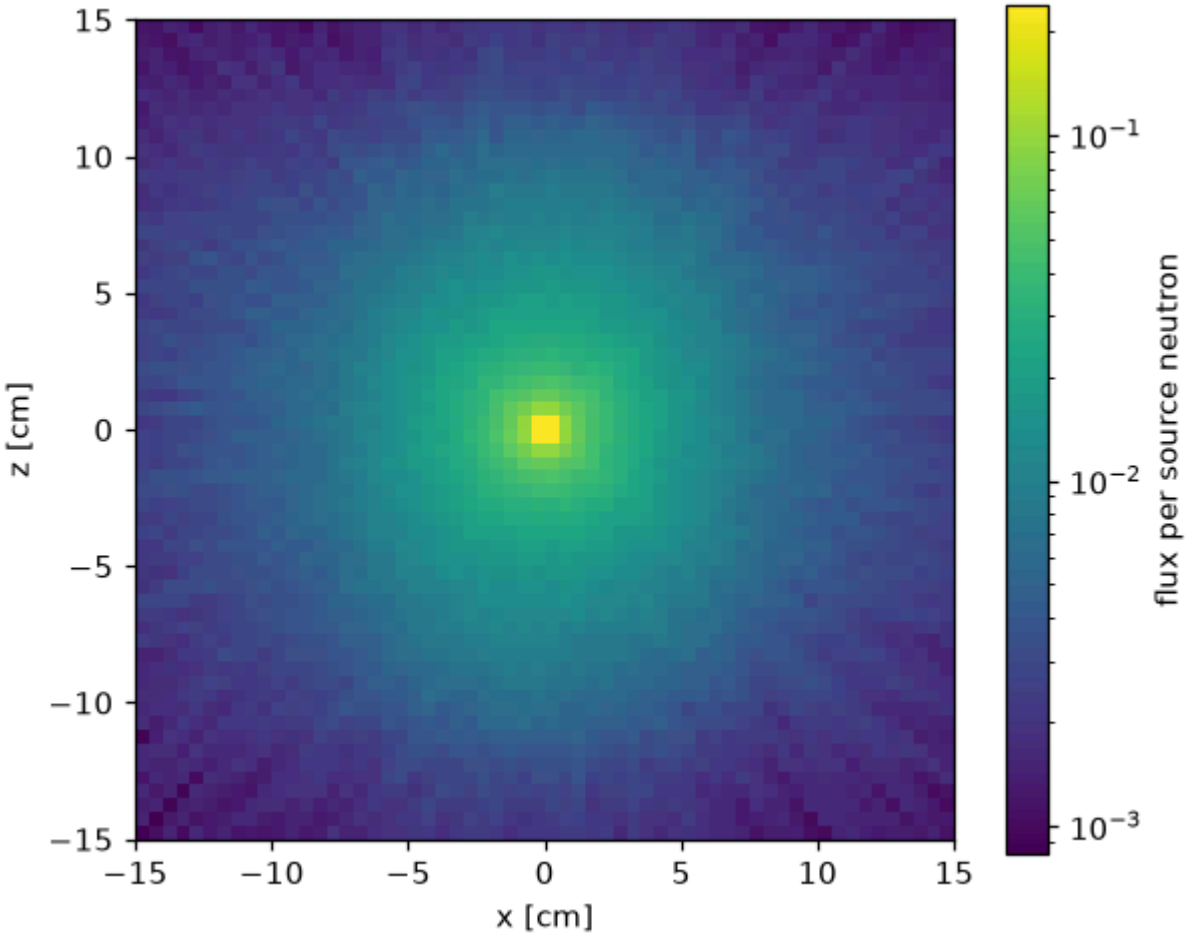
`scaling_factor`	float	1.0	Multiply all tally values by this factor (unit conversion)
`font_size`	int	18	Base font size in pixels for labels and ticks
`show_colorbar`	bool	`True`	Show the colorbar alongside the plot

```
# Per-energy-bin slider over every bin, log scale, source-strength scaled
plot = tally.plot(
    geometry=geometry,
    basis="xz",
    slices="all",
    energy_index=0,
    value="mean",
    colorscale="Cividis",
    log_scale=True,
    axis_units="m",
    scaling_factor=1e14,           # multiply by source strength
    title="Fast neutron flux",
    colorbar_title="Flux [n/cm²/s]",
)
plot.to_html("flux_fast.html")
```

For manual matplotlib plotting, use `extract_mesh_slice()` with optional geometry outline overlay.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

# Extract tally slice
mean = tally.extract_mesh_slice(basis="xy", value="mean")
mean_2d = np.array(mean)

# Sample geometry for outline overlay
geom_slice = geometry.sample_slice(basis="xy", resolution=(400, 400))
edge_mask = np.array(geom_slice.edges("material"))

# Plot
fig, ax = plt.subplots()
vmin = mean_2d[mean_2d > 0].min()
im = ax.pcolormesh(mean.h_edges, mean.v_edges, mean_2d,
                   cmap="viridis", norm=LogNorm(vmin=vmin))
outline_rgba = np.zeros((*edge_mask.shape, 4))
outline_rgba[edge_mask] = [1, 1, 1, 0.8]
ax.imshow(outline_rgba, extent=geom_slice.extent, origin="lower",
          aspect="auto", interpolation="nearest", zorder=2)
fig.colorbar(im, ax=ax, label=f"Flux [{mean.h_label}]")
plt.savefig("manual_tally_plot.png")
```

The flux map below is rendered when these docs are built. The model is a small steel pipe carrying water with a 14.06 MeV point source on the axis, sliced in the `xz` plane:



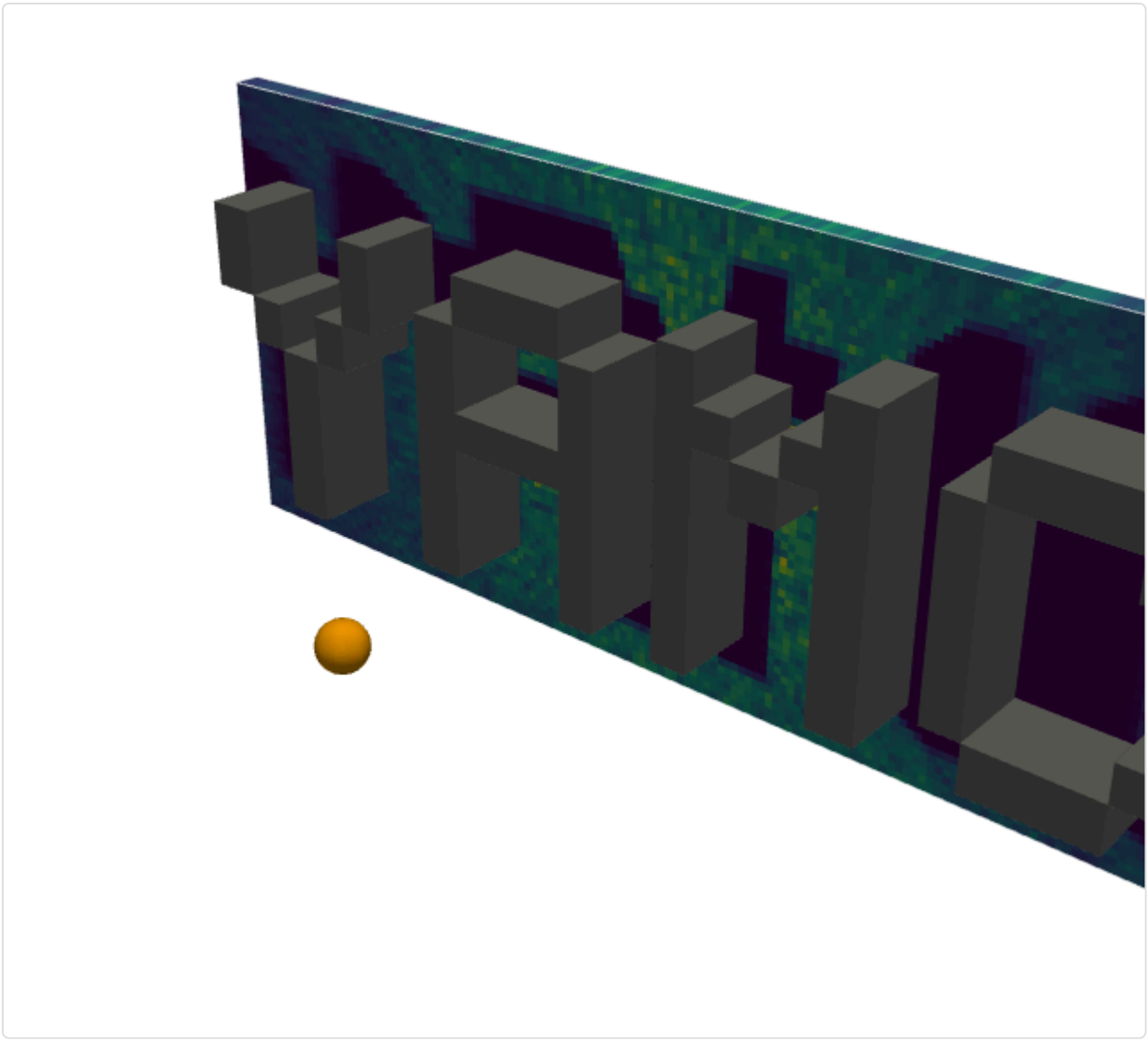
VTK-HDF export and interactive 3D view

Mesh tally results export to VTK-HDF with `Tally.to_vtkhdf()`, called after `model.simulate_transport()` on a tally that has a `mesh` set. The file opens in ParaView and, as shown below, can also be viewed in the browser.

```
tally.to_vtkhdf("flux.vtkhdf")

# With options
tally.to_vtkhdf(
    "flux.vtkhdf",
    scaling_factor=1.6e-19 * 1e21,  # eV to W conversion
    volume_normalization=True,      # divide by voxel volume (default)
    sum_energy=True,                # sum over energy bins (default)
    sum_nuclides=True,              # sum over parent nuclide bins (default)
)

# Per-energy-bin export
tally.to_vtkhdf("flux_per_energy.vtkhdf", sum_energy=False)
```

The viewer below is rendered live when these docs are built. It is a neutron radiograph: the word `YAMC` is built from boron-10 blocks (a strong thermal-neutron absorber), a thermal point source (orange marker) sits in front of them, and the letters cast a shadow onto a flux screen behind them. Both the

flux tally and the geometry are written to VTK-HDF; the grey `YAMC` blocks are the geometry (read back from `geometry.vtkhdf`), and the coloured panel is the flux screen (read back from `flux.vtkhdf`), where the letters appear as dark shadows. Drag to rotate, scroll to zoom, right-drag to pan. The same files open in ParaView for further analysis.



Performance stats

Timing information is stored on the results returned by `simulate_transport`:

```
results = model.simulate_transport()
print(f"Total time:     {results.elapsed:.2f} s")
print(f"Particles/sec:  {results.particles_per_second}")
print(f"Data load time: {results.data_load_time:.2f} s")
print(f"Transport time: {results.transport_time:.2f} s")
```

From an ENDF file to a transport simulation

Nuclear data evaluations are published as ENDF files, and yamc reads Arrow. This example takes one evaluation, aluminium-27 from TENDL-2025, converts it to Arrow on your own machine and uses it in a transport simulation: a 10 cm radius sphere of aluminium with a 14.06 MeV point source at its centre, tallying the spectrum of the neutrons that leak out of it. Finally the same model is run on yamc's published TENDL-2025 data, so you can check the conversion against it.

The same steps work for any neutron evaluation: one from another library, a patched one, or a new one produced by a code such as TALYS.

Installing

yamc, plus NJOY to reconstruct the resonances and Doppler broaden the evaluation. NJOY2016 is not on PyPI; this wheel puts an `njoy` executable in the environment's `bin/`:

```
pip install yamc matplotlib
pip install --extra-index-url https://shimwell.github.io/wheels njoy2016
```

Any NJOY build works too. Pass its path to the converter as `njoy_exec`.

Download the evaluation

TENDL serves each evaluation as its own file, about 4 MB for Al27:

```
import urllib.request

url = "https://tendl.imperial.ac.uk/tendl_2025/neutron_files/Al/Al027/lib/endf/n-Al027.tendl"
request = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
with urllib.request.urlopen(request) as response, open("n-Al027.tendl", "wb") as f:
    f.write(response.read())
```

Convert it to Arrow

`convert_neutron_transport` runs NJOY on the evaluation and writes everything a transport simulation reads: the cross sections, the secondary angle and energy distributions, and the lookup tables. Ask for the temperature the material will be simulated at, here room temperature.

```
import yamc

yamc.convert_neutron_transport(
    "n-Al027.tendl",
    "my_data/neutron",
    temperatures=[294.0],
)
```

This takes about 20 seconds and writes `my_data/neutron/Al27.arrow/`. The nuclide's name comes from the evaluation itself, not from the file name.

Simulate the leakage spectrum

Point yamc at the folder of converted data. Any nuclide the model uses is then read from `<folder>/<Nuclide>.arrow`.

```
yamc.neutron_cross_section_data = "my_data/neutron"

sphere = yamc.Sphere(radius=10.0, boundary="vacuum")
aluminium = yamc.Material(composition={"Al27": 1.0}, density=2.7)
geometry = yamc.Geometry([yamc.Cell(region=sphere.below, material=aluminium)])

leakage = yamc.Tally(
    scores=["current-above"],
    surfaces=sphere,
    energy_group_structure="VITAMIN-J-175",
)

model = yamc.Model(geometry=geometry, tallies=[leakage], source=yamc.NeutronSource())
converted = model.simulate_transport(total_particles=1_000_000)
```

`current-above` on the outer sphere counts every neutron crossing it outwards, and with a vacuum boundary nothing comes back in, so this is the leakage. One million neutrons take about a second.

Compare with the published data

yamc's published TENDL-2025 data is processed from the same evaluation, so the two spectra should lie on top of each other. A model keeps the nuclear data it first ran with, so build a new one on the same geometry for the published data, run it, and plot both spectra per unit lethargy:

```
import matplotlib.pyplot as plt
import numpy as np

yamc.neutron_cross_section_data = "tendl-2025"
model = yamc.Model(geometry=geometry, tallies=[leakage], source=yamc.NeutronSource())
published = model.simulate_transport(total_particles=1_000_000)

bins = np.array(leakage.energy_bins)
lethargy = np.log(bins[1:] / bins[:-1])

fig, ax = plt.subplots()
for label, run in [("converted here", converted), ("published tendl-2025",
published)]:
    current = np.array(run[leakage].mean)
    print(f"{label}: {current.sum():.4f} neutrons leak per source neutron")
    ax.stairs(current / lethargy, bins, label=label)
ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel("Energy [eV]")
ax.set_ylabel("Leakage per unit lethargy per source neutron")
ax.legend()
fig.savefig("leakage_spectrum.png", dpi=150)
```

On the published data about 0.89 neutrons leak per source neutron. The peak at 14 MeV is the neutrons that crossed the sphere without a collision; below it are those that scattered on the way out. If your own conversion differs from the published one by more than the Monte Carlo noise, the two were processed differently, for example at another temperature.

Coupling to a heat solve

This example couples yamc's nuclear heating to a steady heat conduction solve in FEniCSx, and feeds the temperature back into yamc as the density of a liquid breeder. A sphere of PbLi sits in a steel box with a 14.06 MeV point source at its centre:

1. yamc tallies the heating in every tet of the mesh.
2. FEniCSx solves for the temperature on the same tets, with the box's outer faces held at 600 K.
3. The PbLi's density at each tet's temperature goes back into yamc with `Model.set_tet_density_factors`, and the loop repeats until the temperature stops changing.

Both codes run on one tet mesh, made from the CAD by `MeshGeometry.from_cadquery`, so nothing is interpolated between them: each tet carries one heating value, one temperature and one density factor. Lengths are in cm throughout, including in the heat solve.

Installing

yamc with its CAD extra, and FEniCSx with PETSc. FEniCSx is not on PyPI yet; these wheels install it with pip on Linux, for Python 3.12 or newer and glibc 2.34 or newer (Ubuntu 22.04 on):

```
pip install "yamc[cad]"
pip install --extra-index-url https://shimwell.github.io/wheels mpich "fenics-dolfinx[petsc4py]"
```

Materials

The PbLi density follows a linear fit in temperature. The material is built at the wall temperature, and the density factors are relative to that.

```
import time

import basix.ufl
import cadquery as cq
import numpy as np
import ufl
from dolfinx import fem, mesh
from dolfinx.fem.petsc import LinearProblem
from mpi4py import MPI

import yamc

yamc.neutron_cross_section_data = "endf-b8.1"
yamc.photon_cross_section_data = "endf-b8.1"

T_WALL = 600.0  # K


def pbli_density(T):
    """PbLi density in g/cm3 at T in K (Mas de les Valls et al. 2008)."""
    return 10.52035 - 1.19051e-3 * T


breeder = yamc.Material(
    composition={"Pb": 0.83, "Li": 0.17}, density=pbli_density(T_WALL),
temperature=T_WALL
)
steel = yamc.materials.pnnl.material("Steel, Stainless 316", temperature=T_WALL)
```

Geometry and mesh

`from_cadquery` imprints the assembly, meshes its surfaces and fills the listed volumes with tets of about `target_edge_length`, all in one call. The materials are keyed by the assembly's part names. Both volumes are tet-meshed, the breeder for its density feedback and the steel because the heat solve needs every region the heat flows through.

```
sphere = cq.Workplane("XY").sphere(10.0)
box = cq.Workplane("XY").box(30, 30, 30).cut(sphere)
assembly = cq.Assembly()
assembly.add(sphere, name="breeder")
assembly.add(box, name="structure")

geometry = yamc.MeshGeometry.from_cadquery(
    assembly,
    materials={"breeder": breeder, "structure": steel},
    tet_volumes=["breeder", "structure"],
    target_edge_length=2.5,  # cm
)
tets = np.array(geometry.tetrahedra)
tet_names = np.array(geometry.tet_volume_names)
in_breeder = tet_names == "breeder"
tet_volume = np.array(geometry.tet_volumes)  # cm3
```

The tets of the two volumes meet face to face on the sphere, and their outer faces are exactly the surface triangles transport tracks against. `geometry.to_arrow("blanket.arrow")` would save the mesh, for `MeshGeometry.from_arrow` to load later, but nothing here needs the file.

Neutronics model

One heating tally covers every tet: given the geometry, a tet-mesh tally has a bin for each of its tets, in the order of `geometry.tetrahedra`.

```
heating_tally = yamc.Tally(scores=["heating"], mesh=geometry)
model = yamc.Model(
    geometry=geometry,
    source=yamc.NeutronSource(),
    tallies=[heating_tally],
    transport_secondary_photons=True,
    tracking_mode="hybrid",  # per-tet densities need delta tracking
)


def heating_density(seed):
    """Nuclear heating in W/cm3 in every tet."""
    results = model.simulate_transport(total_particles=100_000, seed=seed)
    ev_per_source = np.array(results[heating_tally].mean)
    source_rate = 5e15  # neutrons per second
    return ev_per_source * source_rate * 1.602176634e-19 / tet_volume  # eV to J
```

A tet-mesh tally scores eV per source neutron in each tet, integrated over the tet. Multiplying by the source rate and dividing by the tet volume gives W/cm3.

Thermal model on the same tets

`dolfinx.mesh.create_mesh` builds the FEniCSx mesh from yamc's vertices and tets. FEniCSx reorders the cells, and `original_cell_index` maps each of its cells back to a yamc tet. Heating and conductivity are constant per tet (DG0), the temperature is continuous (CG1), and each tet's temperature for the feedback is the CG1 field at its centre.

```
CONDUCTIVITY = {"breeder": 0.15, "structure": 0.25}  # W/(cm K)

used, cells = np.unique(tets, return_inverse=True)
coordinates = np.array(geometry.vertices)[used]
element = basix.ufl.element("Lagrange", "tetrahedron", 1, shape=(3,))
domain = mesh.create_mesh(MPI.COMM_WORLD, cells.reshape(-1, 4), ufl.Mesh(element),
coordinates)
yamc_tet = domain.topology.original_cell_index

DG0 = fem.functionspace(domain, ("DG", 0))
V = fem.functionspace(domain, ("Lagrange", 1))
kappa = fem.Function(DG0)
kappa.x.array[:] = [CONDUCTIVITY[name] for name in tet_names[yamc_tet]]
q = fem.Function(DG0)

# the mesh is conformal, so its boundary is just the box's outer faces
domain.topology.create_connectivity(2, 3)
outer = mesh.exterior_facet_indices(domain.topology)
bc = fem.dirichletbc(T_WALL, fem.locate_dofs_topological(V, 2, outer), V)
u, v = ufl.TrialFunction(V), ufl.TestFunction(V)
problem = LinearProblem(
    kappa * ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx,
    q * v * ufl.dx,
    bcs=[bc],
    petsc_options_prefix="heat_",
    petsc_options={"ksp_type": "cg", "pc_type": "gamg", "ksp_rtol": 1e-10},
)
temperature_per_tet = fem.Function(DG0)


def tet_temperatures(heating):
    """Steady temperature in K at the centre of every tet."""
    q.x.array[:] = heating[yamc_tet]
    temperature_per_tet.interpolate(problem.solve())
    T = np.empty(len(tets))
    T[yamc_tet] = temperature_per_tet.x.array
    return T
```

Every facet of the breeder/steel interface is shared by a breeder tet and a steel tet, so FEniCSx's exterior facets are exactly the box's outer faces and the boundary condition goes on all of them, with no geometric test. Heat crosses the whole interface.

Iterating

Each transport run is noisy, so the heating is a running mean over the iterations rather than the latest run alone. That averages the noise down as the temperature converges. Only the breeder's tets get density factors; the steel's stay at 1.

```
heating = None
T_old = np.full(len(tets), T_WALL)
for iteration in range(1, 7):
    start = time.perf_counter()
    factors = np.where(in_breeder, pbli_density(T_old) / pbli_density(T_WALL), 1.0)
    model.set_tet_density_factors(factors)
    new = heating_density(seed=iteration)
    heating = new if heating is None else heating + (new - heating) / iteration
    T = tet_temperatures(heating)
    print(
        f"iteration {iteration}: power {np.sum(heating * tet_volume) / 1e3:.2f} kW, "
        f"peak T {T.max():.1f} K, breeder density {factors[in_breeder].min():.4f}"
        f"-{factors[in_breeder].max():.4f} of 600 K, max |dT| {np.abs(T -
T_old).max():.2f} K, "
        f"{time.perf_counter() - start:.0f} s"
    )
    T_old = T
```

```
iteration 1: power 4.39 kW, peak T 784.8 K, breeder density 1.0000-1.0000 of 600 K,
max |dT| 184.82 K, 44 s
iteration 2: power 4.37 kW, peak T 782.8 K, breeder density 0.9776-0.9954 of 600 K,
max |dT| 2.24 K, 43 s
iteration 3: power 4.38 kW, peak T 782.4 K, breeder density 0.9778-0.9954 of 600 K,
max |dT| 0.54 K, 42 s
iteration 4: power 4.38 kW, peak T 781.8 K, breeder density 0.9779-0.9954 of 600 K,
max |dT| 0.62 K, 44 s
iteration 5: power 4.38 kW, peak T 781.7 K, breeder density 0.9779-0.9954 of 600 K,
max |dT| 0.23 K, 40 s
iteration 6: power 4.38 kW, peak T 781.5 K, breeder density 0.9779-0.9954 of 600 K,
max |dT| 0.17 K, 41 s
```

The first iteration has no feedback, as every factor is 1. From the second on the hottest PbLi is 2.2% less dense than at the wall, and the temperature has settled to within the transport noise by the third. The feedback lowers the peak temperature by about 3 K here: PbLi's density changes little over a few hundred kelvin, so density feedback is a small correction for this breeder.

Method

This page describes what yamc computes and in what order. The usage guide covers how to call it.

yamc is a continuous-energy Monte Carlo code for fixed-source neutron and photon transport. It has no eigenvalue (criticality) mode: fission progeny are followed as part of the history that created them, and a model that is supercritical is stopped with an error rather than run.

Overview

A run has these stages:

1. **Load.** Each material's nuclides are read from the nuclear data library and the cross sections are prepared at the material temperature (Nuclear data).
2. **Source.** Each history starts with one particle drawn from the sources (Sources).
3. **Transport.** The particle is moved from collision to collision through the geometry (Geometry, Tracking). At each collision a nuclide and a reaction are sampled (Neutron collisions, Photon transport), and secondary particles are put on a per-history bank.
4. **Scoring.** Tallies are scored along tracks or at collisions (Tallies).
5. **History end.** When the bank is empty, the history's contribution to each tally bin is folded into the running statistics (Statistics).
6. **Stop.** The run ends when the particle count is reached, every convergence target is met, or the runtime budget is spent, whichever comes first.

Transmutation and shutdown dose runs build on this loop (Transmutation coupling, Shutdown dose rate).

Nuclear data

Each nuclide is a directory of Apache Arrow IPC files, one per section: metadata and temperatures, energy grid, reactions, secondary products and distributions, unresolved resonance probability tables, total ν, fission photon data and (opt-in) covariances. Only the sections a run needs are loaded: transport loads every section but the covariances, which only a nuclear-data uncertainty run reads, and for a material with a temperature set only that temperature or the two published ones that bracket it; activation loads only the reaction cross sections.

Cross sections

Cross sections are interpolated linearly in energy (lin-lin) everywhere. Threshold reactions are zero below threshold.

The summed reactions are rebuilt from the partials at load, using the ENDF sum rules, rather than taken from the file:

- total is elastic plus non-elastic (MT 2 + MT 3);
- absorption is disappearance (MT 27) minus fission, so it never includes fission;
- scattering is elastic plus every neutron-emitting non-elastic partial;
- fission is MT 18, or the sum of MT 19/20/21/38 where only the partials are given.

A redundant total is skipped when its own levels are present (for example MT 16 next to MT 875 to 890), so nothing is counted twice.

There are two lookup levels:

- **Material.** Each material has a union grid of its nuclides' grids, with the macroscopic total Σ_t and each nuclide's macroscopic total on it. These give the flight distance and the nuclide choice. The macroscopic sum is taken in a fixed nuclide order so that it is reproducible.
- **Nuclide.** Each nuclide keeps its own grid, holding total, absorption, scattering and fission, plus the per-reaction partials. These give the reaction choice and the secondary channel.

Both use a logarithmic index (8000 to 8192 bins) to bracket the energy, followed by a short search. Energies off the end of a grid take the end value.

Temperature

If a material's temperature matches a library temperature, that data is used. If it falls between two library temperatures, a new temperature is made once at load by blending the two data sets pointwise, linearly in T, on the union of their grids. A temperature outside the library range is an error.

The blend is deterministic. Picking one of the two bracketing temperatures at random for each lookup would bias deep-penetration results, because the attenuation through a thick shield is an average of exponentials, not the exponential of the average cross section. Probability tables are not blended; the new temperature takes the nearer neighbour's tables.

There is no on-the-fly Doppler broadening. The default temperature is 294 K.

Sources

A history starts by choosing a source in proportion to its strength, then drawing its position, direction and energy.

- **Position:** a point, or a cylindrical ring with independent r, φ and z distributions about an axis, or a mesh. On a mesh the element is chosen by a binary search over the cumulative strengths, and the point is uniform within it: directly in a box, uniform in r^2 (and in φ and z) in a cylindrical cell, and from sorted uniform barycentric weights in a tetrahedron. With `cells` or `materials` given, points outside them are rejected and redrawn in the same element.
- **Direction:** isotropic or monodirectional.
- **Energy:** discrete lines and histogram bins are drawn by the alias method. Uniform, normal and the D-D and D-T fusion spectra of Ballabio et al. 1998 are also available. The default neutron energy is 14.06 MeV.

The tokamak source discretises the plasma profiles of Fausser et al. 2012 into one ring source per (voxel, reaction), each with a Ballabio spectrum at the voxel's ion temperature.

The stellarator source takes the plasma shape as a Fourier-Zernike series in the conventions of DESC, R and Z as functions of ρ, θ and ζ, evaluated with a Jacobi recurrence for the Zernike radial polynomials. The volume element $R\,\tau$, with $\tau = R_\rho Z_\theta - R_\theta Z_\rho$, is evaluated analytically, and a shape whose volume element changes sign (self-intersecting or non-nested surfaces) is refused. The plasma is split into radial shells; shell strengths are Gauss-Legendre integrals in ρ of the emission density times $\mathrm{d}V/\mathrm{d}\rho$, the angular integral taken by a trapezoid rule fine enough to be exact for the trigonometric polynomial $R\,\tau$. Within a shell, ρ is proposed with density $\propto \rho$ and the angles uniformly over one field period, accepted against a majorant of $S(\rho)\,R\,|\tau|/\rho$, and the field period is then chosen at random.

Source strength is a relative selection probability only. Results are per source particle and are scaled to an absolute rate afterwards.

Geometry

CSG

Surfaces are planes, spheres, cylinders and cones on any axis, elliptical tori about x, y or z, and general quadrics. Distances to tori come from a closed-form quartic solve. A cell's region is a tree of half-spaces combined by intersection, union and complement. At setup it is compiled into a flat reverse-Polish program, so a point-in-cell test needs no recursion and no allocation.

To find the cell containing a point, yamc first tries the particle's current cell, then the neighbours learned for that cell during the run (Harper et al. 2020), then a bounding-volume hierarchy over the cells'

bounding boxes. Bounding boxes are computed recursively from the region tree, so ring and frame cells built from unions and complements still have finite boxes.

The nearest surface along the flight is the smallest positive distance over the cell's surfaces. After a crossing the particle is moved 10^{-8} cm past the surface and its cell is found again.

Mesh geometry

Triangulated surface meshes are tracked directly, with one surface BVH per volume: a binned SAH build, collapsed to a 4-wide tree tested with SIMD. Tetrahedra, when present, are used for unstructured mesh tallies, not for locating particles. The space between meshed volumes is an implicit complement, void unless given a material. After a crossing the next volume comes from the surface-to-volume topology, so no point location is needed.

Before a crossing is accepted, the segment is checked for any other surface it passes through. A segment that skips a surface is recorded as lost rather than continuing in the wrong volume.

A defect in the mesh loses the particle where the defect is, rather than letting it fly on in the wrong volume:

- **A point the mesh cannot place.** Point location (at a source site, or after a Woodcock flight) casts test rays and takes the parity of their crossings. A ray whose parity disagrees with the way the nearest facet along it faces (a stray facet, a flipped facet or a crack) is passed over. A point that no ray places consistently is lost there, instead of being put in the implicit complement and carried on as void.
- **A crack in a volume's surface.** A particle that slips out through a missing facet next meets the volume's far side from outside. That facet is checked just short of the crossing: if the particle is still inside the volume, the facet is only wound the wrong way and the particle leaves through it; if it is outside, the particle left through the crack and is lost where it left.

A sound mesh takes neither path, so these checks cost it nothing.

Mesh inside CSG

A mesh can fill a CSG cell, with a translation and rotation. Its volumes become cells inside the host, and the host's material fills the space between them. The next boundary is the nearer of the CSG exit and the mesh crossing. The mesh crossing wins only if it is at least 10^{-8} cm nearer, so a mesh face lying on the host surface does not trap the particle. A mesh that sticks out of its host is refused unless clipping is explicitly allowed.

Boundaries and lost particles

A surface is either transmitting (the default) or vacuum. A particle that crosses a vacuum surface leaks and is killed. There are no reflective, white or periodic boundaries.

A particle whose cell cannot be found is recorded as lost (position, direction, energy, last cell and surface) and killed. The run stops once more than `max_lost_particles` (default 10) are lost, because a lost particle almost always means a geometry error rather than a rare event.

Tracking

The distance to the next collision is

$$d = -\frac{\ln \xi}{\Sigma_t(E)},$$

with ξ uniform on $(0, 1]$. Three tracking modes are available.

Surface tracking (the default) compares d with the distance to the nearest surface. If the surface is nearer, the particle crosses it and d is redrawn in the next cell; otherwise it collides.

Woodcock (delta) tracking samples d against a majorant $\Sigma_{\mathrm{maj}}(E) \geq \Sigma_t(E)$ that holds in every material, moves the particle without looking at surfaces, then accepts a real collision with probability $\Sigma_t / \Sigma_{\mathrm{maj}}$. Otherwise the collision is virtual and the flight continues at the same energy. A landing point in void is always virtual. The majorant is global: the union of all the materials' energy grids, with the largest macroscopic total at each point, using the worst-case probability-table band inside the unresolved range. A per-material majorant would not be a bound once the flight crosses into another material. Vacuum exits are still found exactly by testing the flight against the vacuum surfaces only.

Hybrid tracking chooses per cell. A cell is surface-tracked when the expected number of virtual collisions across it, $(\Sigma_{\mathrm{maj}} - \Sigma_t)\,\bar{\ell}$, exceeds 1, where $\bar{\ell} = 4V/S$ is the mean chord of its bounding box. Thin, dense cells in a mostly light model are then tracked by surfaces, and everything else by Woodcock.

Woodcock and hybrid tracking are CPU only and do not support mesh fills.

Neutron collisions

Choosing the nuclide and the reaction

The nuclide is chosen in proportion to its macroscopic total. The reaction is then chosen from four partials: elastic, inelastic (every neutron-emitting non-elastic reaction), fission and absorption. Within inelastic or absorption, the specific reaction is chosen in proportion to its cross section.

Elastic scattering

Below $400\,kT$ (the `free_gas_threshold`), the target's thermal motion is sampled from a free gas with the constant cross section approximation. For hydrogen (atomic weight ratio $A \leq 1$) it is always sampled. Above the threshold the target is at rest. The centre-of-mass angle comes from the tabulated distribution, and the outgoing energy and lab angle from two-body kinematics:

$$E' = E\,\frac{A^2 + 2A\mu + 1}{(A+1)^2}, \qquad \mu_{\mathrm{lab}} = \frac{1 + A\mu}{\sqrt{A^2 + 2A\mu + 1}}.$$

Thermal scattering laws ($S(\alpha, \beta)$) and resonance-corrected elastic kernels (DBRC, RVS) are not used.

Unresolved resonances

In the unresolved resonance range, cross sections come from probability tables. One random number is drawn when the particle reaches a new energy and is held until the energy changes, including across surfaces and through void, so the flight, the nuclide choice, the reaction choice, the secondary photon production, the tallies and the transmutation reaction rates all see the same band. Each nuclide derives its own band from that number with a hash of its ZA, so the bands of different nuclides are independent.

The band is read at the two bracketing table energies with the same random number and interpolated between them. The band total is always the sum of its partials (elastic, inelastic, capture, fission), not the table's own total column. Where the tables are relative to the smooth cross sections, capture is scaled from the smooth absorption, which in yamc already excludes fission, so fission is not subtracted twice.

Secondary neutrons

Outgoing angle and energy come from the product distributions in the data: discrete levels, tabulated and continuous tabular energy laws, Maxwell, Watt and evaporation spectra, Kalbach-Mann, correlated angle-energy and N-body phase space. Tabulated distributions pick an incident-energy bracket at random and scale the sampled energy to the interpolated bracket limits. Laws given in the centre-of-mass frame are transformed to the lab with

$$E_{\mathrm{lab}} = E_{\mathrm{cm}} + \frac{E + 2\mu_{\mathrm{cm}}(A+1)\sqrt{EE_{\mathrm{cm}}}}{(A+1)^2}, \qquad \mu_{\mathrm{lab}} = \mu_{\mathrm{cm}}\sqrt{\frac{E_{\mathrm{cm}}}{E_{\mathrm{lab}}}} + \frac{1}{A+1}\sqrt{\frac{E}{E_{\mathrm{lab}}}}.$$

For an integer neutron yield, such as (n,2n), the extra neutrons are banked at the particle's weight, which is analog. A non-integer yield multiplies the weight instead. Secondaries go on a last-in, first-out bank and are followed to completion within the same history.

Fission

The number of neutrons is $\bar{\nu}_{\text{total}}(E)$, rounded stochastically. Each neutron is delayed with probability $\beta(E) = \bar{\nu}_d(E)/\bar{\nu}_{\text{total}}(E)$. Delayed neutrons are emitted at once (there is no precursor time) from a single delayed spectrum: the six group spectra summed with their group yields. Prompt neutrons are drawn from the prompt spectrum of the sampled fission channel. Emission is isotropic in the lab. Using the prompt spectrum for every neutron would make the fission spectrum too hard.

A history whose live fission bank passes 20,000 neutrons is stopped with an error, because the model is then supercritical or close to it.

Energy limits

There is no neutron energy cutoff. Neutrons are followed until they are absorbed or leak.

Photon transport

The element is chosen in proportion to its macroscopic total, then the interaction from its cross sections (interpolated log-log):

- **Coherent (Rayleigh):** angle from the integrated form factor, then rejection on $(1 + \mu^2)/2$. The energy is unchanged.
- **Incoherent (Compton):** Klein-Nishina with rejection on the incoherent scattering function. Doppler broadening is applied using the relativistic impulse approximation of Kaltiaisenaho 2016, with the Compton profiles of Biggs et al. 1975. The vacancy left in the struck shell is relaxed.
- **Photoelectric:** the subshell is chosen from the subshell cross sections, the photoelectron gets $E - E_b$ and leaves at a Sauter angle about the photon's direction, and the vacancy is relaxed.
- **Pair production:** the electron and positron energies are sampled with the PENELOPE method (Salvat 2011). The positron annihilates at rest into two back-to-back 511 keV photons. Triplet production is included in the pair cross section.

Atomic relaxation follows the full cascade of radiative and non-radiative transitions. Fluorescence photons are banked and Auger electrons are treated as electrons, below.

Electrons and positrons are not transported. By default their energy is converted to bremsstrahlung photons using thick-target bremsstrahlung (TTB) yields from Seltzer and Berger 1986; the photons start at the electron's position and direction and the rest of the energy is deposited locally. With `electron_treatment="local"` all of it is deposited locally.

Photons below `photon_cutoff_energy` (default 1 keV) are killed.

The photon data are read directly from the ENDF photo-atomic and atomic relaxation sublibraries, with no NJOY step.

Coupled neutron-photon transport

With secondary photons on, photons are sampled at every neutron collision, before the reaction is processed. The mean number is $\sigma_{\gamma,\mathrm{prod}}/\sigma_t$, made an integer by a Bernoulli draw on its fractional part, and each photon carries the neutron's weight. Here σ_t is the total the collision was sampled from; in the unresolved range that is the probability-table band's, and each reaction's photon production is moved by its own channel's band factor (capture by the capture factor, and so on). The producing reaction is chosen in proportion to its photon production, and the photon's energy and angle come from that reaction's distribution, relative to the incoming neutron direction. Fission photon yields are scaled up by $(\bar{\nu}_p + \bar{\nu}_d)/\bar{\nu}_p$ to include delayed fission photons.

Variance reduction

The default is fully analog. Two methods can be switched on. Survival biasing runs on the CPU and the GPU, and weight windows on the CPU only.

Survival biasing

At a neutron collision absorption is not sampled. The weight is multiplied by the scattering probability, $(\sigma_e + \sigma_i)/\sigma_t$, and fission neutrons are banked from the weight before that discount, in proportion to σ_f/σ_t. Secondary photons are sampled before the discount. Below `weight_cutoff` (default 0.25) the particle plays Russian roulette and survives with probability w/w_s at weight w_s (`weight_survive`, default 1). Photons stay analog.

Weight windows

A weight window gives lower and upper weight bounds per mesh voxel and energy group, for each particle type. Windows are applied after each collision:

- above the upper bound, the particle is split into $n = \min(\lceil w/w_u \rceil, n_{\max})$ copies of weight w/n, with $n_{\max}$ = `max_split` (default 10);
- at or below the lower bound, it plays roulette with survival weight $\min(w\,n_{\max},\ c\,w_l)$, with c = `survival_factor` (default 3).

Splitting stops for a history once it has made 10,000 extra particles, which caps the cost of a badly placed window without changing the expected result.

Generating windows (DeGVR)

Windows are generated by density-extrapolation global variance reduction (Pan et al. 2023). The model is first run analog with every density divided by N, which is cheap because particles go further. A

second run at density divided by $N/2$ uses the first run's windows. With F and H the two flux maps, the flux at full density is extrapolated voxel by voxel as

$$G = F\left(\frac{H}{F}\right)^{N-1}$$

in log space. The lower bound is the extrapolated flux divided by the survival factor and the group maximum; the upper bound is 5 times the lower.

If `density_reduction` is not given, N is chosen without transport: a ray trace from the strongest source to the voxel centres gives the optical depth τ at the 90th percentile, and $N = \max(\tau/7,\ 2)$. Coupled neutron-photon windows are generated from one pair of runs with a flux mesh per particle type.

Tallies

Estimators

A tally score has the form $R(E)$ times a flux contribution, where R is 1 for flux, a macroscopic cross section for a reaction rate, or a KERMA factor for heating. Two estimators are available.

Track-length (the default) scores every track segment:

$$\text{score} = w\,\ell\,R(E),$$

with w the weight and ℓ the length of the segment in the bin.

Collision scores every real collision:

$$\text{score} = w\,\frac{R(E)}{\Sigma_t(E)},$$

with Σ_t the total the collision was sampled from, including the probability-table band in the unresolved range.

Under Woodcock tracking there is no track in a cell to score, so track-length tallies switch to the collision-density estimator, scored at every collision, real or virtual:

$$\text{score} = w\,\frac{R(E)}{\Sigma_{\text{maj}}(E)}.$$

The exception is a flux tally on a mesh with no cell or material filter. Its true track length can be found by walking the flight through the mesh, so it keeps the track-length estimator.

Mesh tallies

A track crossing a mesh is split between voxels in proportion to the length in each. Regular meshes are walked with the Amanatides-Woo traversal (Amanatides and Woo 1987); cylindrical meshes are intersected analytically; unstructured tetrahedral meshes are walked element to element.

Heating

Neutron heating uses the KERMA cross sections (MT 301, or MT 901 for local heating). Photon heating depends on the estimator:

- **Collision:** the energy absorbed at each collision, $w\left(E_{\text{in}} - E_{\text{out}} - \sum E_{\text{banked photons}}\right)$. This is analog.
- **Track-length:** the photon KERMA, built from the cross sections: for Compton, the mean energy transfer of Klein-Nishina scattering weighted by the incoherent scattering function (the distribution Compton scattering samples), less the fluorescence energy; for the photoelectric effect, E less the expected fluorescence energy; for pair production, $E - 2m_e c^2$. Coherent scattering deposits nothing.

Both include the energy of the electrons. They differ in where the bremsstrahlung energy ends up: the collision estimator follows the TTB photons, the KERMA counts that energy where the electron was made. In a thick body they agree to about 1%. In a thin or streaming one the KERMA is higher by the fraction that escapes, which is the usual difference between kerma and absorbed dose.

Dose

Dose is a flux tally weighted by a dose coefficient $f(E)$: ICRP 74 or ICRP 116 effective dose per fluence, for a chosen irradiation geometry. The coefficients are interpolated log-log, the way the tables are read everywhere else; events outside the table are not scored.

Statistics

Mean and standard deviation

One history is one sample. During a history, its contributions to each bin are summed in a scratch buffer, and at the end of the history that sum x is folded into the bin's running mean μ and sum of squared deviations M_2 with Welford's update:

$$\delta = x - \mu, \qquad \mu \leftarrow \mu + \frac{\delta}{n}, \qquad M_2 \leftarrow M_2 + \delta\,(x - \mu).$$

Histories that do not touch a bin are folded in as zeros at the end, in one step. Each thread keeps its own accumulators, which are merged pairwise with the parallel form of the update (Chan et al. 1979), so the result is exact in real arithmetic whatever the split.

The reported standard deviation is that of the mean,

$$s_{\bar{x}} = \sqrt{\frac{M_2}{N(N-1)}},$$

and the relative error is $s_{\bar{x}}/\bar{x}$. There are no batches and no batch size to choose.

This sample-per-history estimate is correct when secondaries are correlated with their parent, as fission and (n,xn) progeny are, because all of a history's particles add to the same sample.

The GPU accumulates $\sum x$ and $\sum x^2$ per bin instead and forms $M_2 = \sum x^2 - (\sum x)^2/N$.

Convergence checks

For each tally, the per-history total over all bins is also tracked, with its third and fourth moments. From it yamc reports:

- the **variance of the variance**, $\mathrm{VoV} = M_4/M_2^2 - 1/N$;
- the **figure of merit**, $\mathrm{FOM} = 1/(R^2 t)$, with R the relative error and t the wall-clock time;
- the **tail slope** of the score distribution, from a Hill estimate over the 201 largest scores.

The checks passed are: relative error below 0.1, in total and in every populated bin, VoV below 0.1, tail slope above 3, and, over the second half of the run, a mean that is not trending (a 95% Student-t test on the slope of the means of the blocks of histories between snapshots), relative error and VoV that do not grow, a FOM that stays within a factor of 2, a relative error falling as $1/\sqrt{N}$ (a log-log slope between -0.75 and -0.25), a VoV falling as $1/N$ (a slope between -1.5 and -0.5) and a FOM with no trend in N (a slope within ± 0.5). The three rate checks read as not evaluated with fewer than four snapshots in that half, and both FOM checks also when the half spans less than 0.1 s. A tally passes only if its tail slope could be estimated.

A run can stop early on convergence targets: a relative error, standard deviation or VoV threshold, on all tallies or a named one. Targets are checked between chunks of histories, as is the runtime budget, so a run can go over either by up to one chunk.

Bin covariance

On request, a tally also accumulates $P_{ij} = \sum_h x_{hi} x_{hj}$ over histories, giving

$$\mathrm{Cov}(\bar{x}_i, \bar{x}_j) = \frac{P_{ij} - N \bar{x}_i \bar{x}_j}{N(N-1)},$$

for up to 2048 bins.

Combining runs

Separate runs of the same model can be combined exactly: the means, M_2, higher moments, covariance products and score distributions are merged with the same pairwise update. Runs are refused if they could share random number streams, if the model, data library or tally definitions differ, if they read different published releases of one library (each run records its releases in `runs[i]["data_releases"]`), or if they came from the GPU.

Random numbers and reproducibility

Every history has its own random number stream, seeded from the base seed and the history's index in the run. The thread, chunk and MPI rank it runs on do not enter, so a history is the same whatever the parallel layout. Collisions use a 64-bit PCG generator shared with the GPU kernels. Each secondary gets its own seed from its parent's seed and its birth order, so the order in which the bank is emptied does not change it. The CPU empties the bank last in, first out and the GPU first in, first out, and both follow the same histories.

Histories run in parallel on a rayon thread pool and, with MPI, across ranks. Per-thread statistics are merged at the end. Because floating-point addition is not associative, results are statistically identical across thread counts but may differ in the last bits.

Transmutation coupling

Reaction rates for transmutation are scored directly from continuous-energy cross sections, $w\,\ell\,\sigma(E)$ per nuclide and reaction on every track, not by folding a group flux. In the unresolved range σ is the probability-table band the flight drew, as for the other tallies. They are normalised per source particle, then by the step's source rate and the material volume. Fission yields are weighted by the continuous-energy fission rate. Isomeric branching ratios are applied at the collision energies.

A short scouting run first finds the nuclides the material can populate, so only those are tallied. Each step is then solved with the yani CRAM-48 solver (Pusa 2016). In "coupled" mode transport is re-run

every step with the updated compositions, using the rates at the start of the step. In "independent" mode transport runs once and its rates are scaled by each step's source rate.

Shutdown dose rate

Shutdown dose uses the direct one-step method (D1S). During transport, activation reactions that lead (through any decay chain) to a photon emitter are treated as prompt photon sources. One decay photon is emitted per collision, isotropically, with weight $w\,\sigma_{\mathrm{D1S}}/\sigma_t$, and tagged with its decay path: the emitter alone when the reaction makes it, the chain from the reaction product to the emitter (`Cs137>Ba137m`) when it is made through decay. The path's branching ratios are in the photon yield. After the run, each path's bin is multiplied by its time correction factor, the Bateman activity along that path over the irradiation and cooling schedule, which carries the time dependence alone. One transport run therefore gives the dose at any number of cooling times.

When the machine changes between steps (`simulate_shutdown_dose`), the run is split in two. A neutron run per irradiating model records each decay photon at birth (history, position, direction, energy, weight, decay path and cell) instead of tracking it. A photon run per observing configuration o then makes photon history i out of every photon that neutron history i recorded, in every irradiating configuration c, each moved by its cell's rigid motion $T_o T_c^{-1}$ (direction by the rotation alone; dropped if the cell is absent) and transported once at its own weight. The history's dose at the end of each step s that o observes is formed at the end of the history,

$$D_i(s) = \sum_c \sum_p \mathrm{TCF}_{c,p}(s)\, x_{i,c,p},$$

where $x_{i,c,p}$ is the history's tally of the photons of decay path p recorded in configuration c, and $\mathrm{TCF}_{c,p}$ uses only that configuration's pulses. The histories are independent, so the mean and standard error of $D_i(s)$ over them are those of the whole calculation, the neutron stage and every correlation within a history included. The photon runs use random-number streams independent of the neutron runs', since transporting history i's photons on the stream that made them would correlate the two.

The rigorous two-step method (R2S, Chen and Fischer 2002) is offered beside the one-step method. One neutron transport tallies the spectrum of every activated region, each region's inventory is solved through the schedule with the transmutation solver, and a photon transport per step samples the decay photons uniformly over each region at that step's activity, with the line and continuum spectra of the decay data. A region is a material-filled cell, or with a mesh each part of such a cell under a voxel (cell-under-mesh), its spectrum then the voxel's in VITAMIN-J-175 groups and its volume estimated from 8^3 jittered points per voxel. Products that are themselves activated, and burnup, enter through the inventory solve.

Nuclear-data uncertainty

Transport can estimate the uncertainty from the cross-section covariances in one run. Each history carries a likelihood ratio for each of a set of perturbed data replicas (Rief 1984): with surface tracking a flight of length d multiplies it by $e^{-\Delta\Sigma_t d}$ and a collision by $\sigma'_{\mathrm{MT}}/\sigma_{\mathrm{MT}}$. Under delta tracking the majorant is raised to cover every replica's total, so a flight moves no ratio, a real collision takes the same $\sigma'_{\mathrm{MT}}/\sigma_{\mathrm{MT}}$ and a virtual one multiplies it by $(\Sigma_{\mathrm{maj}} - \Sigma'_t)/(\Sigma_{\mathrm{maj}} - \Sigma_t)$. Track-length scores are weighted by the mean survival over the track. Collision-estimator scores take the ratio a history arrives with, before the sampled reaction changes it: the collision density's change in Σ_t cancels the estimator's $1/\Sigma_t$. With either estimator, a neutron's reaction-rate, production, heating or damage-energy score is also multiplied by the change to its own response, Σ'_x/Σ_x. The spread of the replica results includes the Monte Carlo noise of the shared histories, which is estimated and subtracted (Clements et al. 2024).

The MF=33 cross-section covariances are sampled, and where a nuclide's data carries its resonance parameters (MF=2 and MF=32), each replica draws those and rebuilds the resonance cross sections from the draw. Heating and damage energy move with the reactions that deposit them, apart from fission's share, which stays nominal. Secondary distributions, $\bar{\nu}$ and fission spectra are not perturbed, and photon data carries no covariance: a photon inherits its neutron's ratios times the change to the cross section of the reaction that made it. The ratios are carried through every tracking mode, survival biasing, weight windows and MPI. GPU runs, and overlay, mesh and per-nuclide tallies, are refused.

A transmutation with the "independent" method takes its uncertainty from its one transport: the replica weights above give each replica's rates, the per-history covariance of the tallies gives their statistical spread, and each replica's schedule is solved again from them. The "coupled" method cannot do this, because each step's rates were tallied in compositions that the earlier steps' noise and data had already moved. Each replica instead repeats the whole coupled calculation, as in Total Monte Carlo (Koning and Rochman 2008): its own transport seed, its own cross sections and isomeric production curves in every transport, and its own half-lives, decay branching ratios and fission yields in every solve. The spread over the replicas then carries the feedback through the compositions exactly, with nothing linearised. A replica transported on its own cross sections follows its own histories whatever its seed, so its spread holds the transport's noise as well, which is why the coupled method carries the cross sections only beside the statistical source.

References

- J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing", Eurographics 1987 Technical Papers (1987), doi:10.2312/egtp.19871000.
- L. Ballabio, J. Källne and G. Gorini, "Relativistic calculation of fusion product spectra for thermonuclear plasmas", Nucl. Fusion 38 (1998) 1723-1735.

- F. Biggs, L. B. Mendelsohn and J. B. Mann, "Hartree-Fock Compton profiles for the elements", At. Data Nucl. Data Tables 16 (1975) 201-309.
- T. F. Chan, G. H. Golub and R. J. LeVeque, "Updating formulae and a pairwise algorithm for computing sample variances", Stanford University report STAN-CS-79-773 (1979).
- Y. Chen and U. Fischer, "Rigorous MCNP based shutdown dose rate calculations: computational scheme, verification calculations and application to ITER", Fusion Eng. Des. 63-64 (2002) 107-114.
- K. B. Clements, G. Geraci, A. J. Olson and T. S. Palmer, "A variance deconvolution estimator for efficient uncertainty quantification in Monte Carlo radiation transport applications", J. Quant. Spectrosc. Radiat. Transf. 319 (2024) 108958.
- D. W. Dudt and E. Kolemen, "DESC: A stellarator equilibrium solver", Phys. Plasmas 27 (2020) 102513, doi:10.1063/5.0020743.
- C. Fausser et al., "Tokamak D-T neutron source models for different plasma physics confinement modes", Fusion Eng. Des. 87 (2012) 787, doi:10.1016/j.fusengdes.2012.02.025.
- S. Harper, P. Romano, B. Forget and K. S. Smith, "Threadsafe dynamic neighbor lists for Monte Carlo ray tracing", Nucl. Sci. Eng. 194 (2020) 1009-1015.
- T. Kaltiaisenaho, "Implementing a photon physics model in Serpent 2", M.Sc. thesis, Aalto University (2016).
- A. J. Koning and D. Rochman, "Towards sustainable nuclear energy: Putting nuclear physics to work", Ann. Nucl. Energy 35 (2008) 2024-2030, doi:10.1016/j.anucene.2008.06.004.
- Q. Pan, L. Wang, Y. Cai, X. Liu and J. Xiong, "Density-extrapolation Global Variance Reduction (DeGVR) method for large-scale radiation field calculation", Comput. Math. Appl. 143 (2023) 10-22.
- M. Pusa, "Higher-order Chebyshev rational approximation method and application to burnup equations", Nucl. Sci. Eng. 182 (2016) 297-318, doi:10.13182/NSE15-26.
- H. Rief, "Generalized Monte Carlo perturbation algorithms for correlated sampling and a second-order Taylor series approach", Ann. Nucl. Energy 11 (1984) 455-476.
- F. Salvat, J. M. Fernández-Varea and J. Sempau, "PENELOPE-2011: a code system for Monte Carlo simulation of electron and photon transport", OECD/NEA (2011).
- S. M. Seltzer and M. J. Berger, "Bremsstrahlung energy spectra from electrons with kinetic energy 1 keV-10 GeV incident on screened nuclei and orbital electrons of neutral atoms with Z = 1-100", At. Data Nucl. Data Tables 35 (1986) 345-418, doi:10.1016/0092-640X(86)90014-8.

Python API

The reference below is generated by mkdocstrings from the package's type stubs. Classes appear under their public names (`yamc.Material`, `yamc.Tally`, ...) with their typed constructors. The private `yamc._core` module that holds the compiled extension is not listed.

yamc -- Yet Another Monte Carlo code.

---

class AngleDistribution

Energy-dependent secondary-angle distribution; `sample(energy)` draws a scattering cosine (mu) for a given incoming energy.

attr energy property

```
energy: list[float]
```

Get energy grid

attr num_energy_points property

```
num_energy_points: int
```

Get number of energy points

meth sample

```
sample(incoming_energy: float) -> float
```

Sample scattering cosine (mu) for a given incoming energy

Source code in `yamc/_core/__init__.pyi`

```
157  def sample(self, incoming_energy: builtins.float) -> builtins.float:
158      r"""
159      Sample scattering cosine (mu) for a given incoming energy
160      """
```

class BoundingBox

```
BoundingBox(lower_left: Sequence[float], upper_right: Sequence[float])
```

Axis-aligned bounding box in 3D space.

This class is typically returned by geometry methods rather than constructed directly.

Create a bounding box from corner coordinates.

Parameters:

Name	Type	Description	Default
`lower_left`	`Sequence[float]`	Lower-left corner [x, y, z] in cm.	*required*
`upper_right`	`Sequence[float]`	Upper-right corner [x, y, z] in cm.	*required*

Returns:

Name	Type	Description
`BoundingBox`	`BoundingBox`	An axis-aligned bounding box.

Source code in `yamc/_core/__init__.pyi`

```
200  def __new__(cls, lower_left: typing.Sequence[builtins.float], upper_right:
201  typing.Sequence[builtins.float]) -> BoundingBox:
202      r"""
203      Create a bounding box from corner coordinates.
204
205      Args:
206          lower_left: Lower-left corner [x, y, z] in cm.
207          upper_right: Upper-right corner [x, y, z] in cm.
208
209      Returns:
210          BoundingBox: An axis-aligned bounding box.
         """
```

attr lower_left property

```
lower_left: list[float]
```

Lower-left corner coordinates [x, y, z] in cm.

attr upper_right property

```
upper_right: list[float]
```

Upper-right corner coordinates [x, y, z] in cm.

attr center property

```
center: list[float]
```

Center coordinates [x, y, z] in cm.

attr width property

```
width: list[float]
```

Width along each axis [dx, dy, dz] in cm.

meth is_finite

```
is_finite() -> bool
```

True when every corner coordinate is finite.

An unbounded region (a half-space, or a cell with no outer surface) produces infinite extents, which cannot be voxelized. Mirrors [`yamc_geo::BoundingBox::is_finite`].

Returns:

Type	Description
`bool`	bool

Source code in `yamc/_core/__init__.pyi`

```
189  def is_finite(self) -> builtins.bool:
190      r"""
191      True when every corner coordinate is finite.
192
193      An unbounded region (a half-space, or a cell with no outer surface)
194      produces infinite extents, which cannot be voxelized. Mirrors
195      [`yamc_geo::BoundingBox::is_finite`].
196
197      Returns:
198          bool
199      """
```

class ClearanceResult

The outcome of assessing a material against one clearance limit set.

Every regulation here uses the same arithmetic: each radionuclide's activity divided by its tabulated limit, summed. The material meets the limits when that sum, `index`, is below `threshold`. The per-nuclide dicts are ordered largest first.

Examples:

```
>>> result = material.clearance_index("UK_EPR16_out_of_scope")
>>> result.clearable
False
>>> result.dominant(2)
[('Co60', 11.2...), ('Cs137', 0.029...)]
>>> print(result)
```

attr limit_set property

```
limit_set: str
```

Name of the limit set assessed against.

attr index property

```
index: float
```

The sum of activity-to-limit ratios.

attr threshold property

```
threshold: float
```

The value the index must stay below, normally 1.

attr units property

```
units: str
```

Activity units the comparison was made in: `"Bq/g"`, `"Ci/m3"` or `"Bq"`.

attr clearable property

```
clearable: bool
```

Whether the material meets this set's limits.

attr out_of_scope property

```
out_of_scope: bool
```

Whether the regulation excludes this material outright, which for the UK sets means every radionuclide present is shorter lived than 100 s.

attr by_nuclide property

```
by_nuclide: dict[str, float]
```

Each nuclide's contribution to the index, largest first.

attr activities property

```
activities: dict[str, float]
```

Each nuclide's activity in `units`, largest first.

attr limits_used property

```
limits_used: dict[str, float]
```

The limit applied to each nuclide, after any metal, per-gram, dynamic or secular equilibrium adjustment.

attr defaulted property

```
defaulted: list[str]
```

Nuclides that took the set's catch-all limit because the table does not list them.

attr excluded property

```
excluded: dict[str, str]
```

Nuclide to the reason its activity was left out, which is always that a parent's limit already accounts for it in full.

attr credited property

```
credited: dict[str, float]
```

Nuclide to the activity a parent accounted for, where the parent could only support part of it. The remainder was assessed against the nuclide's own limit.

attr uncovered property

```
uncovered: dict[str, float]
```

Activity present with no limit and no catch-all, so absent from the index entirely. The number to check before trusting a comfortable index.

attr unlimited property

```
unlimited: list[str]
```

Nuclides present that the source explicitly places no limit on.

attr uncovered_activity property

```
uncovered_activity: float
```

Total activity with no limit, in `units`.

attr uncovered_fraction property

```
uncovered_fraction: float
```

Share of total activity that falls outside the sum, from 0 to 1.

attr material_name property

```
material_name: str
```

The material's name, carried through for reporting.

meth assessed_activity

```
assessed_activity(nuclide: str) -> float
```

The activity actually charged against the limit for one nuclide: its total activity, less any part a parent accounted for.

Parameters:

Name	Type	Description	Default
`nuclide`	`str`	Nuclide name, such as `"Co60"`.	*required*

Source code in `yamc/_core/__init__.pyi`

```
529  def assessed_activity(self, nuclide: builtins.str) -> builtins.float:
530      r"""
531      The activity actually charged against the limit for one nuclide: its
532      total activity, less any part a parent accounted for.
533
534      Args:
535          nuclide (str): Nuclide name, such as ``"Co60"``.
536      """
```

meth dominant

```
dominant(count: int = 10) -> list[tuple[str, float]]
```

The nuclides contributing most to the index, largest first.

Parameters:

Name	Type	Description	Default
`count`	`int`	How many to return.	`10`

Source code in `yamc/_core/__init__.pyi`

```
537  def dominant(self, count: builtins.int = 10) ->
538  builtins.list[tuple[builtins.str, builtins.float]]:
539      r"""
540      The nuclides contributing most to the index, largest first.
541
542      Args:
543          count (int): How many to return.
         """
```

`meth` to_json

```
to_json(path: Optional[str | PathLike | Path] = None) -> Optional[bytes]
```

Save the result as JSON: the limit set, material, index, threshold and units, whether it is clearable or out of scope, and per nuclide the index contributions, activities and limits used, with the defaulted, excluded, credited, uncovered and unlimited nuclides.

Parameters:

Name	Type	Description	Default
`path`	`str | PathLike | None`	Where to write. None returns the JSON as bytes instead, for a browser or Pyodide session.	`None`

Returns:

Type	Description
`Optional[bytes]`	bytes | None: The JSON when `path` is None.

Examples:

```
>>> result.to_json("clearance.json")
>>> json.loads(result.to_json())["clearable"]
True
```

Source code in `yamc/_core/__init__.pyi`

```
544  def to_json(self, path: typing.Optional[builtins.str | os.PathLike |
545  pathlib.Path] = None) -> typing.Optional[bytes]:
546      r"""
547      Save the result as JSON: the limit set, material, index, threshold
548      and units, whether it is clearable or out of scope, and per nuclide
549      the index contributions, activities and limits used, with the
550      defaulted, excluded, credited, uncovered and unlimited nuclides.
551
552      Args:
553          path (str | os.PathLike | None): Where to write. None returns the
554              JSON as bytes instead, for a browser or Pyodide session.
555
556      Returns:
557          bytes | None: The JSON when ``path`` is None.
558
559      Examples:
560          >>> result.to_json("clearance.json")
561          >>> json.loads(result.to_json())["clearable"]
562          True
        """
```

class ConvergencePoint

One snapshot of a tally's aggregate statistics versus history count.

attr elapsed_secs `property`

```
elapsed_secs: float
```

Transport wall-clock time elapsed at this snapshot, in seconds.

class ConvergenceTarget

```
ConvergenceTarget(metric: str, threshold: float, tally: Optional[str] = None,
tally_id: Optional[int] = None)
```

A precision-based stopping criterion. Attach to `model.convergence_targets` to stop a (single-process) run early once a tally is precise enough.

```
model.convergence_targets = [ConvergenceTarget("relative_error", 0.02,
tally="flux")]
```

`metric` is one of `"relative_error"`, `"std_dev"`, `"variance_of_variance"` (all on the tally's total), or `"bin_relative_error"`: the relative error that 95% of the tally's scored bins meet, so a mesh tally converges voxel by voxel rather than in total (CPU only). Restrict to one tally with `tally` (name) or `tally_id`; otherwise the threshold must hold for every tally.

Source code in `yamc/_core/__init__.pyi`

```
604  def __new__(cls, metric: builtins.str, threshold: builtins.float, tally:
605  typing.Optional[builtins.str] = None, tally_id: typing.Optional[builtins.int] =
606  None) -> ConvergenceTarget:
607      r"""
608      ``metric`` is one of ``"relative_error"``, ``"std_dev"``,
609      ``"variance_of_variance"`` (all on the tally's total), or
610      ``"bin_relative_error"``: the relative error that 95% of the tally's
611      scored bins meet, so a mesh tally converges voxel by voxel rather than
612      in total (CPU only). Restrict to one tally with ``tally`` (name) or
          ``tally_id``; otherwise the threshold must hold for every tally.
          """
```

`class` Cooldown

```
Cooldown(duration: Any, model: Optional[Any] = None, moves: Optional[Any] = None)
```

A cooling step: no source, the materials just decay for `duration`.

Create a cooling step.

Parameters:

Name	Type	Description	Default
`duration`	`Any`	A number of seconds, or a `(value, unit)` tuple such as `(7, "d")`. Units: s, min, h, d, a (year).	*required*
`model`	`Optional[Any]`	The yamc `Model` the machine is in while it cools, for `yamc.simulate_shutdown_dose`: the decay photons of every earlier irradiation are transported through its geometry. It needs no source. Cells absent from it have been removed and no longer emit; a cell of the same name is the same component. A later step without a `model` stays in this one.	`None`
`moves`	`Optional[Any]`	Cells of `model` that sit away from their reference position (where they were built in the model they were irradiated in, unless that step moved them too), as `{cell or cell name: (dx, dy, dz)}` in cm or a 4x4 rigid transformation matrix. The activation moves with the cell; `model` must already hold the cell at its moved position.	`None`

Source code in `yamc/_core/__init__.pyi`

```
635 def __new__(cls, duration: typing.Any, model: typing.Optional[typing.Any] = None,
636 moves: typing.Optional[typing.Any] = None) -> Cooldown:
637     r"""
638     Create a cooling step.
639 
640     Args:
641         duration: A number of seconds, or a ``(value, unit)`` tuple such as
642             ``(7, "d")``. Units: s, min, h, d, a (year).
643         model: The yamc ``Model`` the machine is in while it cools, for
644             ``yamc.simulate_shutdown_dose``: the decay photons of every
645             earlier irradiation are transported through its geometry. It
646             needs no source. Cells absent from it have been removed and no
647             longer emit; a cell of the same name is the same component. A
648             later step without a ``model`` stays in this one.
649         moves: Cells of ``model`` that sit away from their reference
650             position (where they were built in the model they were
651             irradiated in, unless that step moved them too), as
652             ``{cell or cell name: (dx, dy, dz)}`` in cm or a 4x4 rigid
653             transformation matrix. The activation moves with the cell;
654             ``model`` must already hold the cell at its moved position.
    """
```

attr model property

```
model: Optional[Any]
```

The Model the machine cools in, or `None`.

attr moves property

```
moves: dict[str, list[list[float]]]
```

The moved cells, `{name: 4x4 matrix}`.

attr duration property

```
duration: float
```

Cooldown duration in seconds.

---

class DoseCoefficients

Fluence-to-dose conversion coefficients (effective dose or ambient dose equivalent H*(10)), returned by :func:`yamc.data.dose_coefficients`.

Carries the energy grid, the coefficients, and their units, and can repackage itself for a tally via :meth:`as_energy_function`.

attr energy property

```
energy: list[float]
```

Incident energies in eV.

attr coefficients property

```
coefficients: list[float]
```

Fluence-to-dose coefficients, one per energy.

attr units property

```
units: str
```

Units of the coefficients (`"pSv cm2"`).

meth as_energy_function

```
as_energy_function() -> tuple[list[float], list[float], str, str]
```

Repackage as the `(energy, coefficients, units, interpolation)` tuple accepted by `Tally(energy_function=...)`. The interpolation is `"log-log"`, how the tabulation is read everywhere else.

Source code in `yamc/_core/__init__.pyi`

```
937  def as_energy_function(self) -> tuple[builtins.list[builtins.float],
938  builtins.list[builtins.float], builtins.str, builtins.str]:
939      r"""
940      Repackage as the ``(energy, coefficients, units, interpolation)`` tuple
941      accepted by ``Tally(energy_function=...)``. The interpolation is
942      ``"log-log"``, how the tabulation is read everywhere else.
         """
```

class DoseResult

The result of time-correcting a tally: dose at the selected cooling step(s).

`mean` / `std_dev` are 1-D (one selected step, when `steps` was a single int) or 2-D (one row per selected step). `by_nuclide` maps each parent radionuclide to its mean contribution, in the same shape. `times` is the elapsed time (seconds) at the end of each selected step.

attr mean property

```
mean: Any
```

Mean dose rate, 1-D for a single selected step or 2-D (one row per step).

attr std_dev property

```
std_dev: Any
```

Standard error on `mean`, same shape: the transport's statistical uncertainty alone.

attr data_std_dev property

```
data_std_dev: Optional[Any]
```

The nuclear-data uncertainty on `mean`, same shape, when `time_correct_tally` was given `data_uncertainty`; `None` otherwise. From the half-lives behind the time-correction factors and the decay photon spectrum normalisations that scale each emitter's tally, with an emitter's gamma and x-ray normalisations drawn independently: the lower end of the range `data_std_dev_correlated` closes.

attr data_std_dev_standard_error property

```
data_std_dev_standard_error: Optional[Any]
```

The standard error of `data_std_dev`, same shape: how far another ensemble of the same size could put it. It allows for a heavy tail (it carries the sample kurtosis). `None` when `data_std_dev` is, and NaN in a bin below four replicas.

attr data_std_dev_correlated property

```
data_std_dev_correlated: Optional[Any]
```

`data_std_dev` with each emitter's gamma and x-ray normalisations drawn fully correlated, the upper end of the range the decay data leave by not stating their correlation. Equal to `data_std_dev` when no normalisation is drawn. Negative correlations are not considered: what the spectra share (a decay scheme's normalisation, its conversion coefficients) moves them the same way.

attr total_std_dev property

```
total_std_dev: Optional[Any]
```

`std_dev` and `data_std_dev` in quadrature, when both exist. They are independent: one is the transport's sampling, the other the evaluated decay data.

attr total_std_dev_correlated property

```
total_std_dev_correlated: Optional[Any]
```

`std_dev` and `data_std_dev_correlated` in quadrature, when both exist.

attr data_uncertainty_info property

```
data_uncertainty_info: Optional[Any]
```

What the nuclear-data uncertainty covered, when asked for: the half-lives sampled, those with no stated sigma, the emitters whose photon normalisation was drawn (`decay_photon_normalisations_perturbed`) and those with a spectrum whose normalisation is folded into its line sigmas (`decay_photon_spectra_folded`), the replica count, the `convergence` target, whether every time-correction factor's sigma reached it (`converged`), whether the replica cap stopped the run (`hit_cap`), the ones that missed (`unconverged`), and what was held at nominal (`not_perturbed`).

attr by_nuclide property

```
by_nuclide: Any
```

Per-radionuclide mean contribution: `{nuclide: values}` in the same shape as `mean`.

attr times property

```
times: Any
```

Elapsed time (seconds) at the end of each selected step.

---

class Element

```
Element(name: str)
```

Chemical element container providing isotope helper methods.

Parameters:

Name	Type	Description	Default
`name`	`str`	Element symbol (e.g. "Fe", "U", "H").	*required*

Create a new Element.

Parameters:

Name	Type	Description	Default
`name`	`str`	Element symbol (e.g. "Fe", "U", "H").	*required*

Source code in `yamc/_core/__init__.pyi`

```
1046  def __new__(cls, name: builtins.str) -> Element:
1047      r"""
1048      Create a new Element.
1049
1050      Args:
1051          name (str): Element symbol (e.g. "Fe", "U", "H").
1052      """
```

attr name `property`

```
name: str
```

Element symbol.

meth get_nuclides

```
get_nuclides() -> list[str]
```

Return list of isotope (nuclide) identifiers for this element.

Returns:

Type	Description
`list[str]`	List[str]: Isotope names sorted by mass number (e.g. ["Fe54", "Fe56"]).

Source code in `yamc/_core/__init__.pyi`

```
1054  def get_nuclides(self) -> builtins.list[builtins.str]:
1055      r"""
1056      Return list of isotope (nuclide) identifiers for this element.
1057
1058      Returns:
1059          List[str]: Isotope names sorted by mass number (e.g. ["Fe54",
1060  "Fe56"]).
          """
```

`class` Enriched

```
Enriched(fraction: float, *, target: str, percent: float, fraction_type: str =
'atom')
```

Enrichment specification for use in Material composition dicts.

Build one with `enriched()` rather than by hand; the constructor is the same either way.

Create an enrichment specification.

Parameters:

Name	Type	Description	Default
`fraction`	`float`	Fraction of this component in the material.	*required*
`target`	`str`	Target nuclide for enrichment (e.g. "Li6").	*required*
`percent`	`float`	Enrichment percentage of the target isotope.	*required*
`fraction_type`	`str`	"atom" for atom percent or "mass" for weight percent.	`'atom'`

Returns:

Name	Type	Description
`Enriched`	`Enriched`	An enrichment specification.

Source code in `yamc/_core/__init__.pyi`

```
1091  def __new__(cls, fraction: builtins.float, *, target: builtins.str, percent:
1092  builtins.float, fraction_type: builtins.str = 'atom') -> Enriched:
1093      r"""
1094      Create an enrichment specification.
1095
1096      Args:
1097          fraction (float): Fraction of this component in the material.
1098          target (str): Target nuclide for enrichment (e.g. "Li6").
1099          percent (float): Enrichment percentage of the target isotope.
1100          fraction_type (str): "atom" for atom percent or "mass" for weight percent.
1101
1102      Returns:
1103          Enriched: An enrichment specification.
          """
```

attr fraction property

```
fraction: float
```

Fraction of this component in the material.

attr target property

```
target: str
```

Target nuclide for enrichment (e.g. "Li6").

attr percent property

```
percent: float
```

Enrichment percentage of the target isotope.

attr fraction_type property

```
fraction_type: str
```

"atom" for atom percent or "mass" for weight percent.

class Estimate

A quantity's value, and the spread the nuclear-data ensemble puts on it.

`nominal` is the unperturbed run, and is present whether or not uncertainty was asked for. `mean` and `std_dev` are `None` below two replicas: a spread over fewer than two samples is unmeasured, not zero, and reporting it as zero would read as a quantity known exactly.

A quantity a decay photon intensity enters (contact dose, the photon spectrum, and the decay heat through its gamma part) has a range rather than one spread when the `"decay_photon_lines"` source is on, because the decay data do not state how a nuclide's photon intensities are correlated. `std_dev` is the lower end, every unstated correlation taken as zero, and `std_dev_correlated` the upper end, every one taken as one; `std_dev_range` gives both. For any other quantity, or with the source off, the two are equal.

attr nominal `property`

```
nominal: float
```

The quantity from the unperturbed inventory.

attr mean `property`

```
mean: Optional[float]
```

The ensemble mean, or None below two replicas.

Worth comparing against `nominal`: activity and decay heat are linear in the atom densities, so the two agree to within the sampling error, and a gap between them says the perturbation is biased rather than merely wide.

attr std_dev `property`

```
std_dev: Optional[float]
```

The ensemble's sample standard deviation, or None below two replicas.

The lower end of `std_dev_range`: where the `"decay_photon_lines"` source is drawn, the correlations the decay data leave unstated between a nuclide's photon intensities are taken as zero, which is the evaluation read literally.

attr std_dev_correlated property

```
std_dev_correlated: Optional[float]
```

The ensemble's sample standard deviation with those correlations taken as one, or None below two replicas: within each nuclide the lines of a spectrum, the spectrum's normalisation and its lines, and its gamma and x-ray spectra all move together. The upper end of `std_dev_range`.

Evaluated on the same inventories as `std_dev`, so the two differ by the line data alone. Equal to `std_dev` for activity, or when the `"decay_photon_lines"` source is not drawn.

attr std_dev_range property

```
std_dev_range: Optional[tuple[float, float]]
```

`(std_dev, std_dev_correlated)`, the range every non-negative correlation between a nuclide's photon intensities gives, or None below two replicas.

Negative correlations are not considered: what the intensities leave unstated is a shared normalisation, which moves every line it scales the same way and cannot anticorrelate them.

attr relative_std_dev property

```
relative_std_dev: Optional[float]
```

`std_dev` as a fraction of `nominal`, or None if either is absent.

attr relative_std_dev_correlated property

```
relative_std_dev_correlated: Optional[float]
```

`std_dev_correlated` as a fraction of `nominal`, or None if either is absent.

attr std_dev_standard_error property

```
std_dev_standard_error: Optional[float]
```

The standard error of `std_dev`: how far another ensemble of the same size could put it, or None below four replicas.

It allows for a heavy tail, `Var(s^2) = s^4 (2/(n-1) + kappa/n)` with `kappa` the sample excess kurtosis, so a lognormal-tailed quantity reads as less settled than a Gaussian one at the same replica count.

attr replicas property

```
replicas: int
```

How many replicas the ensemble held.

attr replica_spread property

```
replica_spread: Optional[float]
```

The replicas' standard deviation as they were solved, before any Monte Carlo noise was removed, or None below two replicas. Equal to `std_dev` where no noise was measured.

attr noise property

```
noise: Optional[float]
```

The standard deviation of the Monte Carlo noise removed from `replica_spread`, so `std_dev**2 = replica_spread**2 - noise**2` (floored at zero), or None where it was not measured.

With transport replica weights each replica's tallied shift from the nominal is resolved only to the transport's history count. Each replica is solved a second time from the even half of the histories, and the spread of the difference measures that noise.

attr noise_dominated property

```
noise_dominated: bool
```

Whether the noise removed exceeded a quarter of the variance left: the sigma is then mostly what more transport histories would sharpen.

class FaceOutput

Python-visible face output.

class FaceVolumeMesh

```
FaceVolumeMesh(vertices: Sequence[Sequence[float]], triangles:
Sequence[Sequence[int]], face_ids: Sequence[int])
```

A welded surface mesh being corrected so each curved CAD face encloses its CAD volume (the compiled core of :mod: `yamc.cad._face_volumes`).

It holds the mesh and its adjacency and does the volume sums, the root finding and the vertex moves; the caller evaluates the CAD (curve points and normals, a face's exact volume) and hands it in.

Parameters:

Name	Type	Description	Default
`vertices`	`Sequence[Sequence[float]]`	List of `[x, y, z]` coordinates.	*required*
`triangles`	`Sequence[Sequence[int]]`	List of `[i, j, k]` vertex indices.	*required*
`face_ids`	`Sequence[int]`	The CAD face of each triangle.	*required*

Raises:

Type	Description
`ValueError`	If `face_ids` is not one per triangle or a triangle indexes past the vertices.

Source code in `yamc/_core/__init__.pyi`

```
1260    def __new__(cls, vertices: typing.Sequence[typing.Sequence[builtins.float]],
        triangles: typing.Sequence[typing.Sequence[builtins.int]], face_ids:
        typing.Sequence[builtins.int]) -> FaceVolumeMesh: ...
```

meth faces

```
faces() -> list[int]
```

The face ids, in order of first appearance among the triangles.

Source code in `yamc/_core/__init__.pyi`

```
1261  def faces(self) -> builtins.list[builtins.int]:
1262      r"""
1263      The face ids, in order of first appearance among the triangles.
1264      """
```

meth vertices

```
vertices() -> list[list[float]]
```

The vertices, as moved so far.

Source code in `yamc/_core/__init__.pyi`

```
1265  def vertices(self) -> builtins.list[builtins.list[builtins.float]]:
1266      r"""
1267      The vertices, as moved so far.
1268      """
```

meth neighbours

```
neighbours(faces: Sequence[int]) -> list[int]
```

`faces` and every face sharing a mesh edge of their boundaries, sorted.

Source code in `yamc/_core/__init__.pyi`

```
1269  def neighbours(self, faces: typing.Sequence[builtins.int]) ->
1270  builtins.list[builtins.int]:
1271      r"""
1272      ``faces`` and every face sharing a mesh edge of their boundaries,
1273      sorted.
      """
```

meth boundary_points

```
boundary_points(faces: Sequence[int], lo: Sequence[float], hi:
Sequence[float]) -> list[tuple[int, list[float]]]
```

`[(vertex, [x, y, z]), ...]`: the vertices on the boundaries of `faces` inside the box from `lo` to `hi` (bounds included), by vertex index.

Source code in `yamc/_core/__init__.pyi`

```
1274  def boundary_points(self, faces: typing.Sequence[builtins.int], lo:
1275  typing.Sequence[builtins.float], hi: typing.Sequence[builtins.float]) ->
1276  builtins.list[tuple[builtins.int, builtins.list[builtins.float]]]:
1277      r"""
1278      ``[(vertex, [x, y, z]), ...]``: the vertices on the boundaries of
1279      ``faces`` inside the box from ``lo`` to ``hi`` (bounds included), by
          vertex index.
          """
```

meth is_chain

```
is_chain(chain: Sequence[int]) -> bool
```

Whether every two consecutive vertices of `chain` share a mesh edge.

Source code in `yamc/_core/__init__.pyi`

```
1280  def is_chain(self, chain: typing.Sequence[builtins.int]) -> builtins.bool:
1281      r"""
1282      Whether every two consecutive vertices of ``chain`` share a mesh edge.
1283      """
```

meth add_edge

```
add_edge(chain: Sequence[int], arcs: Sequence[Sequence[Sequence[float]]]) ->
int
```

Record a matched CAD edge: its mesh vertices `chain` in curve order and `arcs`, per segment, points of the edge from the segment's first vertex to its second, ends included. Returns the edge's index.

Raises:

Type	Description
`ValueError`	If there is not one arc of at least two points per segment, or a vertex is not in the mesh.

Source code in `yamc/_core/__init__.pyi`

```
1284    def add_edge(self, chain: typing.Sequence[builtins.int], arcs:
1285    typing.Sequence[typing.Sequence[typing.Sequence[builtins.float]]]) ->
1286    builtins.int:
1287        r"""
1288        Record a matched CAD edge: its mesh vertices ``chain`` in curve order
1289        and ``arcs``, per segment, points of the edge from the segment's first
1290        vertex to its second, ends included. Returns the edge's index.
1291
1292        Raises:
1293            ValueError: If there is not one arc of at least two points per
                    segment, or a vertex is not in the mesh.
        """
```

`meth` move_edge

```
move_edge(edge: int, faces: Sequence[int], outward:
Sequence[Optional[Sequence[float]]], cap: float) -> Optional[bool]
```

Move the vertices inside edge `edge` along `outward` (per interior vertex of its chain, the unit vector away from the centre of curvature, or None), all by one distance of at most `cap`, so the polyline cuts off the arc's area. Only vertices on no face but `faces` move. Returns whether the move hit the cap, or None when nothing moved.

Raises:

Type	Description
`ValueError`	If there is no such edge or `outward` is not one per interior vertex.

Source code in `yamc/_core/__init__.pyi`

```
def move_edge(self, edge: builtins.int, faces: typing.Sequence[builtins.int],
outward: typing.Sequence[typing.Optional[typing.Sequence[builtins.float]]],
cap: builtins.float) -> typing.Optional[builtins.bool]:
    r"""
    Move the vertices inside edge ``edge`` along ``outward`` (per interior
    vertex of its chain, the unit vector away from the centre of
    curvature, or None), all by one distance of at most ``cap``, so the
    polyline cuts off the arc's area. Only vertices on no face but
    ``faces`` move. Returns whether the move hit the cap, or None when
    nothing moved.

    Raises:
        ValueError: If there is no such edge or ``outward`` is not one per
            interior vertex.
    """
```

`meth` face_apex

```
face_apex(face_id: int) -> Optional[list[float]]
```

Start correcting face `face_id`: the point to take the CAD face's volume from, or None (with the reason counted) when the face has no vertex it may move or its boundary is not matched to added edges.

Raises:

Type	Description
`ValueError`	If the mesh has no such face.

Source code in `yamc/_core/__init__.pyi`

```
def face_apex(self, face_id: builtins.int) ->
typing.Optional[builtins.list[builtins.float]]:
    r"""
    Start correcting face ``face_id``: the point to take the CAD face's
    volume from, or None (with the reason counted) when the face has no
    vertex it may move or its boundary is not matched to added edges.

    Raises:
        ValueError: If the mesh has no such face.
    """
```

meth move_face

```
move_face(cad_volume: float, cap: float) -> bool
```

Finish the face `face_apex` started: move its own vertices along their normals, all by one distance of at most `cap`, so its triangles enclose `cad_volume` (the CAD face's volume from the apex) once closed to the CAD edges. Returns whether the move hit the cap.

Raises:

Type	Description
`ValueError`	If no face was started since the last move.

Source code in `yamc/_core/__init__.pyi`

```
1316  def move_face(self, cad_volume: builtins.float, cap: builtins.float) ->
1317  builtins.bool:
1318      r"""
1319      Finish the face ``face_apex`` started: move its own vertices along
1320      their normals, all by one distance of at most ``cap``, so its
1321      triangles enclose ``cad_volume`` (the CAD face's volume from the apex)
1322      once closed to the CAD edges. Returns whether the move hit the cap.
1323
1324      Raises:
1325          ValueError: If no face was started since the last move.
      """
```

meth stats

```
stats() -> list[tuple[str, int]]
```

Counts of what the moves did: `[(name, count), ...]`.

Source code in `yamc/_core/__init__.pyi`

```
1326  def stats(self) -> builtins.list[tuple[builtins.str, builtins.int]]:
1327      r"""
1328      Counts of what the moves did: ``[(name, count), ...]``.
1329      """
```

class GeometrySliceData

Rich return type for `sample_slice()` on Geometry, Cell, and Model.

Behaves like a `(cell_ids, material_ids)` tuple for backward compatibility (supports tuple unpacking), while also exposing coordinate metadata and an `edges()` method for matplotlib outline overlays.

meth edges

```
edges(color_by: str = 'material') -> list[list[bool]]
```

Compute edge-detection mask for cell or material boundaries.

Parameters:

Name	Type	Description	Default
`color_by`	`str`	`"cell"` or `"material"`. Defaults to `"material"`.	`'material'`

Returns:

Type	Description
`list[list[bool]]`	2D list of bools -- `True` at boundary pixels.

Source code in `yamc/_core/__init__.pyi`

```
1463  def edges(self, color_by: builtins.str = 'material') ->
1464  builtins.list[builtins.list[builtins.bool]]:
1465      r"""
1466      Compute edge-detection mask for cell or material boundaries.
1467
1468      Args:
1469          color_by: ``"cell"`` or ``"material"``. Defaults to ``"material"``.
1470
1471      Returns:
1472          2D list of bools -- ``True`` at boundary pixels.
      """
```

class InteractivePlot

Result of `.plot()` -- renders inline in Jupyter and supports saving to `.html` or `.png`.

In a Jupyter notebook the plot renders automatically via `_repr_html_()`. Use :meth:`save` to write the result to disk.

Examples:

```
>>> plot = geometry.plot(basis="xy")
>>> plot.to_html("geometry.html")   # interactive HTML
>>> plot.to_png("geometry.png")     # static PNG snapshot
```

attr html `property`

```
html: str
```

The raw HTML string.

meth to_html

```
to_html(path: Optional[str | PathLike | Path] = None) -> Optional[bytes]
```

Save the plot as a self-contained interactive HTML page, or return its bytes when `path` is None.

Parameters:

Name	Type	Description	Default
`path`	`str | PathLike | None`	Where to write.	`None`

Returns:

Type	Description
`Optional[bytes]`	bytes | None: The page when `path` is None.

Source code in `yamc/_core/__init__.pyi`

```
1542  def to_html(self, path: typing.Optional[builtins.str | os.PathLike |
1543  pathlib.Path] = None) -> typing.Optional[bytes]:
1544      r"""
1545      Save the plot as a self-contained interactive HTML page, or return
1546      its bytes when ``path`` is None.
1547
1548      Args:
1549          path (str | os.PathLike | None): Where to write.
1550
1551      Returns:
1552          bytes | None: The page when ``path`` is None.
          """
```

`meth` to_png

```
to_png(path: Optional[str | PathLike | Path] = None) -> Optional[bytes]
```

Save the plot as a static PNG image with axes, border and tick marks, or return its bytes when `path` is None.

Parameters:

Name	Type	Description	Default
`path`	`str | PathLike | None`	Where to write.	`None`

Returns:

Type	Description
`Optional[bytes]`	bytes | None: The image when `path` is None.

Source code in `yamc/_core/__init__.pyi`

```
1553  def to_png(self, path: typing.Optional[builtins.str | os.PathLike |
1554  pathlib.Path] = None) -> typing.Optional[bytes]:
1555      r"""
1556      Save the plot as a static PNG image with axes, border and tick
1557      marks, or return its bytes when ``path`` is None.
1558
1559      Args:
1560          path (str | os.PathLike | None): Where to write.
1561
1562      Returns:
1563          bytes | None: The image when ``path`` is None.
          """
```

class InteractiveTallyPlot

Result object for interactive tally plots.

Renders inline in Jupyter notebooks via `_repr_html_()`. Use :meth:`save` to write the result to disk.

Examples:

```
>>> plot = tally.interactive_plot(geometry=geo, basis="xy")
>>> plot.to_html("tally.html")   # interactive HTML
```

attr html `property`

```
html: str
```

The raw HTML string.

meth to_html

```
to_html(path: Optional[str | PathLike | Path] = None) -> Optional[bytes]
```

Save the plot as a self-contained interactive HTML page, or return its bytes when `path` is None.

Parameters:

Name	Type	Description	Default
`path`	`str | PathLike | None`	Where to write.	`None`

Returns:

Type	Description
`Optional[bytes]`	bytes | None: The page when `path` is None.

Source code in `yamc/_core/__init__.pyi`

```
1596  def to_html(self, path: typing.Optional[builtins.str | os.PathLike |
1597  pathlib.Path] = None) -> typing.Optional[bytes]:
1598      r"""
1599      Save the plot as a self-contained interactive HTML page, or return
1600      its bytes when ``path`` is None.
1601
1602      Args:
1603          path (str | os.PathLike | None): Where to write.
1604
1605      Returns:
1606          bytes | None: The page when ``path`` is None.
          """
```

class LineEstimate

One decay-photon line, with the ensemble's spread on its emission rate and on the energy it is emitted at.

The set of lines is not the same in every replica: a nuclide that falls below the density floor in one draw takes its lines out of that draw. The spectrum is reported over the union, a line missing from a replica counts as a zero in it, and `emitting` says how many replicas emitted it at all -- which is the difference between a line that is dim and a line that is sometimes not there.

attr energy `property`

```
energy: float
```

Nominal line energy [eV], the one the evaluation states. Lines are matched across replicas on it.

attr energy_mean property

```
energy_mean: Optional[float]
```

The mean energy the emitting replicas drew for this line [eV], or None below two of them. Equal to `energy` unless the `"decay_photon_lines"` source drew line energies.

attr energy_std_dev property

```
energy_std_dev: Optional[float]
```

The sample standard deviation of the drawn energy over the emitting replicas [eV], or None below two of them. Zero unless the `"decay_photon_lines"` source drew line energies.

attr nominal property

```
nominal: float
```

Emission rate from the unperturbed inventory [photons/s].

attr mean property

```
mean: Optional[float]
```

The ensemble mean [photons/s], or None below two replicas.

attr std_dev property

```
std_dev: Optional[float]
```

The ensemble's sample standard deviation [photons/s], or None below two replicas.

The lower end of `std_dev_range`: where the `"decay_photon_lines"` source is drawn, the correlations the decay data leave unstated between a nuclide's photon intensities are taken as zero, which is the evaluation read literally.

attr std_dev_correlated property

```
std_dev_correlated: Optional[float]
```

The ensemble's sample standard deviation with those correlations taken as one, or None below two replicas: within each nuclide the lines of a spectrum, the spectrum's normalisation and its lines, and its gamma and x-ray spectra all move together. The upper end of `std_dev_range`.

Evaluated on the same inventories as `std_dev`, so the two differ by the line data alone. Equal to `std_dev` when the `"decay_photon_lines"` source is not drawn.

attr std_dev_range property

```
std_dev_range: Optional[tuple[float, float]]
```

`(std_dev, std_dev_correlated)`, the range every non-negative correlation between a nuclide's photon intensities gives, or None below two replicas.

Negative correlations are not considered: what the intensities leave unstated is a shared normalisation, which moves every line it scales the same way and cannot anticorrelate them.

attr relative_std_dev property

```
relative_std_dev: Optional[float]
```

`std_dev` as a fraction of `nominal`, or None if either is absent.

attr relative_std_dev_correlated property

```
relative_std_dev_correlated: Optional[float]
```

`std_dev_correlated` as a fraction of `nominal`, or None if either is absent.

attr std_dev_standard_error property

```
std_dev_standard_error: Optional[float]
```

The standard error of `std_dev`, or None below four replicas (see `Estimate.std_dev_standard_error`).

attr replicas property

```
replicas: int
```

How many replicas the ensemble held.

attr emitting property

```
emitting: int
```

Replicas that emitted this line at a positive rate.

Below `replicas` when the emitter dropped out of some draws. A line with `emitting` far below `replicas` has a mean that is mostly zeros, and its spread says more about whether the line is there than about how bright it is.

---

class LostParticle

A particle that was lost during transport due to a geometry gap.

Contains diagnostic information about the particle's state at the time it was lost, including position, direction, energy, and the last known cell and surface.

Examples:

```
>>> model.simulate_transport()
>>> for lp in model.lost_particles:
...     print(f"Lost {lp.particle_type} at {lp.position}, E={lp.energy} eV")
...     print(f"  Last cell: {lp.last_cell_name} (ID: {lp.last_cell_id})")
...     print(f"  Surface crossed: {lp.surface_id}")
```

attr particle_type property

```
particle_type: str
```

Particle type ("neutron" or "photon").

attr position property

```
position: list[float]
```

Position where the particle was lost [x, y, z] in cm.

attr direction property

```
direction: list[float]
```

Direction the particle was traveling [u, v, w].

attr energy property

```
energy: float
```

Energy of the particle in eV.

attr last_cell_id property

```
last_cell_id: Optional[int]
```

Cell ID of the last cell the particle was in before getting lost.

attr last_cell_name property

```
last_cell_name: Optional[str]
```

Name of the last cell the particle was in before getting lost.

attr surface_id property

```
surface_id: Optional[int]
```

Surface ID of the surface that was crossed when the particle got lost.

---

class Material

```
Material(composition: dict[str | Nuclide, float | Enriched], density: float, *,
density_units: str = 'g/cm3', name: Optional[str] = None, id: Optional[int] =
None, fraction_type: str = 'atom', transmutable: bool = False, volume:
Optional[float] = None, temperature: Optional[float] = None, s_alpha_beta: str |
list[str] | dict[str, float] | None = None)
```

A homogeneous mixture of nuclides at a fixed density and temperature.

A Material is built from a composition dictionary whose keys may be nuclide names (`"Fe56"`), element symbols (`"Fe"`), chemical formulas (`"H2O"`), or :class: `Nuclide` objects, and whose values are atom or weight fractions (or :func: `enriched` specifications). Density is always a float;

`density_units` sets its meaning (`g/cm3` by default, or `atom/barn-cm` for a total atom density). For absolute per-nuclide atom densities use :meth:`Material.from_atom_densities`.

Every Material has an `id` from construction: the one passed to the constructor, or else a fresh one that no earlier material in the process was given, so a `yamc.Tally` can filter on it before any geometry exists. Materials may also carry a name, volume (cm³), temperature (K), and a `transmutable` flag that marks them for depletion by `Model.simulate_transmutation` (not needed for `Material.transmute()`).

Examples:

```
>>> import yamc
>>> water = yamc.Material(composition={"H2O": 1.0}, density=1.0)
>>> iron = yamc.Material(
...     composition={"Fe54": 0.0585, "Fe56": 0.9175,
...                  "Fe57": 0.0212, "Fe58": 0.0028},
...     density=7.874, name="iron")
```

Create a new material with a declarative composition and density.

Parameters:

Name	Type	Description	Default
`composition`	`dict[str | Nuclide, float | Enriched]`	Material composition. Keys are element symbols ("Fe"), nuclide names ("Fe56"), chemical formulas ("H2O"), or `yamc.Nuclide` objects. Values are fractions (float) or enriched() objects.	*required*
`density`	`float`	Density value, interpreted per `density_units`. For `density_units="atom/barn-cm"` it is the total atom density and the composition values are relative atom fractions.	*required*
`density_units`	`str`	Density units: "g/cm3", "g/cc", "kg/m3", or "atom/barn-cm" (total atom density; requires fraction_type="atom").	`'g/cm3'`
`name`	`Optional[str]`	Name for the material.	`None`
`id`	`Optional[int]`	ID for the material. Defaults to a fresh one, which no earlier material was given.	`None`
`fraction_type`	`str`	"atom" for atom fractions or "mass" for mass fractions.	`'atom'`
`transmutable`	`bool`	Mark this material for depletion by `Model.simulate_transmutation`. Not needed for `Material.transmute()`, which depletes whatever it is called on.	`False`
`volume`	`Optional[float]`	Material volume in cm³.	`None`
`temperature`	`Optional[float]`	Temperature in Kelvin. If omitted, 294 K is used, or the only temperature the nuclear data carries when it carries just one.	`None`

`s_alpha_beta`	`str | list[str] | dict[str, float] | None`	Thermal scattering tables for bound nuclides, by name, such as `"c_H_in_H2O"` or `["c_Li_in_FLiBe", "c_Be_in_FLiBe", "c_F_in_FLiBe"]`. A dict gives the fraction of each table's nuclides that is bound (`{"c_H_in_ZrH": 0.6}`); otherwise all of them are. Which nuclides a table applies to is the table's own data. The tables come from the library the cross sections are read from, at the material's temperature (the table it sits on, or the two it falls between); a library without thermal tables, or a temperature outside a table's range, is refused.	`None`

Returns:

Name	Type	Description
`Material`	`Material`	A new material.

Source code in `yamc/_core/__init__.pyi`

```
def __new__(cls, composition: dict[str | Nuclide, float | Enriched], density:
builtins.float, *, density_units: builtins.str = 'g/cm3', name:
typing.Optional[builtins.str] = None, id: typing.Optional[builtins.int] = None,
fraction_type: builtins.str = 'atom', transmutable: builtins.bool = False, volume:
typing.Optional[builtins.float] = None, temperature:
typing.Optional[builtins.float] = None, s_alpha_beta: str | list[str] | dict[str,
float] | None = None) -> Material:
    r"""
    Create a new material with a declarative composition and density.

    Args:
        composition (dict[str | Nuclide, float | Enriched]): Material composition.
            Keys are element symbols ("Fe"), nuclide names ("Fe56"),
            chemical formulas ("H2O"), or ``yamc.Nuclide`` objects.
            Values are fractions (float) or enriched() objects.
        density (float): Density value, interpreted per ``density_units``. For
            ``density_units="atom/barn-cm"`` it is the total atom density and the
            composition values are relative atom fractions.
        density_units (str): Density units: "g/cm3", "g/cc", "kg/m3", or
            "atom/barn-cm" (total atom density; requires fraction_type="atom").
        name (Optional[str]): Name for the material.
        id (Optional[int]): ID for the material. Defaults to a fresh one,
            which no earlier material was given.
        fraction_type (str): "atom" for atom fractions or "mass" for mass
fractions.
        transmutable (bool): Mark this material for depletion by
            ``Model.simulate_transmutation``. Not needed for
            ``Material.transmute()``, which depletes whatever it is called on.
        volume (Optional[float]): Material volume in cm³.
        temperature (Optional[float]): Temperature in Kelvin. If omitted,
            294 K is used, or the only temperature the nuclear data carries
            when it carries just one.
        s_alpha_beta (str | list[str] | dict[str, float] | None): Thermal
            scattering tables for bound nuclides, by name, such as
            ``"c_H_in_H2O"`` or ``["c_Li_in_FLiBe", "c_Be_in_FLiBe",
            "c_F_in_FLiBe"]``. A dict gives the fraction of each table's
            nuclides that is bound (``{"c_H_in_ZrH": 0.6}``); otherwise all
            of them are. Which nuclides a table applies to is the table's
            own data. The tables come from the library the cross sections
            are read from, at the material's temperature (the table it sits
            on, or the two it falls between); a library without thermal
            tables, or a temperature outside a table's range, is refused.

    Returns:
        Material: A new material.
    """
```

attr name `property` `writable`

```
name: Optional[str]
```

Get the name of the material

attr s_alpha_beta `property` `writable`

```
s_alpha_beta: dict[str, float]
```

The thermal scattering tables this material uses, as a dict of table name to the fraction of its nuclides that is bound.

attr thermal_bindings `property`

```
thermal_bindings: dict[str, list[str]]
```

The nuclides each loaded thermal scattering table applies to, as a dict of table name to nuclide names. Empty until the nuclear data is read.

attr id `property` `writable`

```
id: int
```

The material's ID.

Assigned when the material is built: the `id` given to the constructor, or else a fresh one that no earlier material in the process was given. Copies keep it, as do the materials a transmutation result returns, which keep their original's. Setting it replaces it, but a tally keeps the IDs it was built with. Two different Material objects with one ID are refused where they meet, in a `yamc.Geometry` (its cells and their mesh fills) or a `yamc.MeshGeometry`, and a `yamc.Tally` refuses one ID listed twice.

attr transmutable `property` `writable`

```
transmutable: bool
```

Whether this material is marked for depletion by `Model.simulate_transmutation`.

Only the coupled model driver reads this: in a geometry of many materials it selects which ones to deplete. `Material.transmute()` ignores it, so you do not need to set it for a standalone transmutation.

attr fraction_type `property`

```
fraction_type: str
```

Get the percent type ("atom" or "mass") for this material's nuclide fractions.

attr nuclides property

```
nuclides: list[tuple[str, float]]
```

Get the material nuclides as a tuple of (name, fraction) pairs.

When the Material was constructed via the Python API, the order matches the user's composition dict (with each input key's expanded nuclides sorted alphabetically). Nuclides added later (e.g. by transmutation) appear alphabetically at the end. When no input order is tracked, all nuclides are alphabetical.

attr volume property writable

```
volume: Optional[float]
```

Material volume in cm^3, if set.

attr density property

```
density: float
```

Density value, in the material's `density_units`. Always a float: for `"atom/barn-cm"` (and transmuted materials) it is the total atom density, otherwise the mass density.

attr density_units property

```
density_units: str
```

Density units string ("g/cm3", "kg/m3", or "atom/barn-cm").

attr temperature property writable

```
temperature: str
```

Temperature label in Kelvin (e.g. "293").

attr reaction_mts property

```
reaction_mts: list[int]
```

Sorted list of all unique MT reaction numbers present.

meth sample_distance_to_collision

```
sample_distance_to_collision(energy: float, seed: Optional[int]) ->
Optional[float]
```

Sample a distance to the next neutron collision.

Uses the macroscopic total cross section (calculating it first if missing) and an exponential distribution to sample a path length. A deterministic RNG seed can be supplied for reproducibility.

Parameters:

Name	Type	Description	Default
`energy`	`float`	Neutron energy in eV.	*required*
`seed`	`Optional[int]`	RNG seed; if omitted a fixed internal seed is used.	*required*

Returns:

Type	Description
`Optional[float]`	Optional[float]: Sampled distance in cm, or None if total XS unavailable.

Source code in `yamc/_core/__init__.pyi`

```
1923  def sample_distance_to_collision(self, energy: builtins.float, seed:
1924  typing.Optional[builtins.int]) -> typing.Optional[builtins.float]:
1925      r"""
1926      Sample a distance to the next neutron collision.
1927
1928      Uses the macroscopic total cross section (calculating it first if missing)
1929      and an exponential distribution to sample a path length. A deterministic
1930      RNG seed can be supplied for reproducibility.
1931
1932      Args:
1933          energy (float): Neutron energy in eV.
1934          seed (Optional[int]): RNG seed; if omitted a fixed internal seed is
1935  used.
1936
1937      Returns:
          Optional[float]: Sampled distance in cm, or None if total XS
      unavailable.
      """
```

meth from_atom_densities staticmethod

```
from_atom_densities(atom_densities: dict, *, name: Optional[str] = None, id:
Optional[int] = None, transmutable: bool = False, volume: Optional[float] =
None, temperature: Optional[float] = None) -> Material
```

Build a material directly from absolute per-nuclide atom densities.

Stores the given atoms/barn-cm values verbatim (no normalize-then-recombine round-trip), so callers that already hold raw number densities -- e.g. the transmutation stepper -- keep full precision. `density` then reports the sum of the values. This replaces the old `Material(comp, density="sum")`.

Parameters:

Name	Type	Description	Default
`atom_densities`	`dict[str, float]`	nuclide name -> atoms/barn-cm.	*required*
`name`	`Optional[str]`	Name for the material.	`None`
`id`	`Optional[int]`	ID for the material. Defaults to a fresh one, which no earlier material was given.	`None`
`transmutable`	`bool`	Mark this material for depletion by `Model.simulate_transmutation`. Not needed for `Material.transmute()`, which depletes whatever it is called on.	`False`
`volume`	`Optional[float]`	Material volume in cm³.	`None`
`temperature`	`Optional[float]`	Temperature in Kelvin. If omitted, 294 K is used, or the only temperature the nuclear data carries when it carries just one.	`None`

Returns:

Name	Type	Description
`Material`	`Material`	A material whose composition is absolute atom densities.

Source code in `yamc/_core/__init__.pyi`

```
@staticmethod
def from_atom_densities(atom_densities: dict, *, name:
typing.Optional[builtins.str] = None, id: typing.Optional[builtins.int] =
None, transmutable: builtins.bool = False, volume:
typing.Optional[builtins.float] = None, temperature:
typing.Optional[builtins.float] = None) -> Material:
    r"""
    Build a material directly from absolute per-nuclide atom densities.

    Stores the given atoms/barn-cm values verbatim (no normalize-then-
recombine
    round-trip), so callers that already hold raw number densities -- e.g. the
    transmutation stepper -- keep full precision. ``density`` then reports the
    sum of the values. This replaces the old ``Material(comp,
density="sum")``.

    Args:
        atom_densities (dict[str, float]): nuclide name -> atoms/barn-cm.
        name (Optional[str]): Name for the material.
        id (Optional[int]): ID for the material. Defaults to a fresh one,
            which no earlier material was given.
        transmutable (bool): Mark this material for depletion by
            ``Model.simulate_transmutation``. Not needed for
            ``Material.transmute()``, which depletes whatever it is called on.
        volume (Optional[float]): Material volume in cm³.
        temperature (Optional[float]): Temperature in Kelvin. If omitted,
            294 K is used, or the only temperature the nuclear data carries
            when it carries just one.

    Returns:
        Material: A material whose composition is absolute atom densities.
    """
```

meth get_nuclide_names

```
get_nuclide_names() -> list[str]
```

Return a list of nuclide names currently present in the material.

Source code in `yamc/_core/__init__.pyi`

```
def get_nuclide_names(self) -> builtins.list[builtins.str]:
    r"""
    Return a list of nuclide names currently present in the material.
    """
```

meth read_nuclear_data

```
read_nuclear_data(nuclide_path_map: Optional[Any] = None, photon_data:
Optional[Mapping[str, str]] = None) -> None
```

Bulk load nuclear data from a mapping of nuclide -> file path or a keyword string.

Parameters:

Name	Type	Description	Default
`nuclide_path_map`	`Optional[Dict[str, str] | str]`	Mapping of nuclide names to file paths or a keyword string.	`None`
`photon_data`	`Optional[Dict[str, str]]`	Mapping of element symbols to photon Arrow data paths.	`None`

Raises:

Type	Description
`ValueError`	If any file cannot be read / parsed.

Source code in `yamc/_core/__init__.pyi`

```
2020  def read_nuclear_data(self, nuclide_path_map: typing.Optional[typing.Any] =
2021  None, photon_data: typing.Optional[typing.Mapping[builtins.str, builtins.str]]
2022  = None) -> None:
2023      r"""
2024      Bulk load nuclear data from a mapping of nuclide -> file path or a keyword
2025  string.
2026
2027      Args:
2028          nuclide_path_map (Optional[Dict[str,str] | str]): Mapping of nuclide
2029  names to file paths or a keyword string.
2030          photon_data (Optional[Dict[str,str]]): Mapping of element symbols to
      photon Arrow data paths.

          Raises:
              ValueError: If any file cannot be read / parsed.
          """
```

meth nuclide_ptr_addr

```
nuclide_ptr_addr(nuclide: str) -> Optional[int]
```

Return raw pointer address of an internal shared Nuclide (debug only).

Parameters:

Name	Type	Description	Default
`nuclide`	`str`	Nuclide name.	*required*

Returns:

Type	Description
`Optional[int]`	Optional[int]: Address value (process-local) or None if not present.

Source code in `yamc/_core/__init__.pyi`

```
2031  def nuclide_ptr_addr(self, nuclide: builtins.str) ->
2032  typing.Optional[builtins.int]:
2033      r"""
2034      Return raw pointer address of an internal shared Nuclide (debug only).
2035
2036      Args:
2037          nuclide (str): Nuclide name.
2038
2039      Returns:
2040          Optional[int]: Address value (process-local) or None if not present.
      """
```

meth unified_energy_grid_neutron

```
unified_energy_grid_neutron() -> list[float]
```

Return (and build if needed) the unified neutron energy grid.

Returns:

Type	Description
`list[float]`	List[float]: Energy grid in eV.

Source code in `yamc/_core/__init__.pyi`

```
2041    def unified_energy_grid_neutron(self) -> builtins.list[builtins.float]:
2042        r"""
2043        Return (and build if needed) the unified neutron energy grid.
2044
2045        Returns:
2046            List[float]: Energy grid in eV.
2047        """
```

`meth` calculate_microscopic_xs_neutron

```
calculate_microscopic_xs_neutron(mt_filter: Optional[Sequence[int]] = None) -
> dict[str, dict[int, list[float]]]
```

Calculate microscopic neutron cross sections on the unified energy grid.

Parameters:

Name	Type	Description	Default
`mt_filter`	`Optional[List[int]]`	Restrict to these MT numbers.	`None`

Returns:

Type	Description
`dict[str, dict[int, list[float]]]`	Dict[str, Dict[int, List[float]]]: nuclide -> MT -> xs array

Source code in `yamc/_core/__init__.pyi`

```
2048  def calculate_microscopic_xs_neutron(self, mt_filter:
2049  typing.Optional[typing.Sequence[builtins.int]] = None) ->
2050  builtins.dict[builtins.str, builtins.dict[builtins.int,
2051  builtins.list[builtins.float]]]:
2052      r"""
2053      Calculate microscopic neutron cross sections on the unified energy grid.
2054
2055      Args:
2056          mt_filter (Optional[List[int]]): Restrict to these MT numbers.
2057
          Returns:
              Dict[str, Dict[int, List[float]]]: nuclide -> MT -> xs array
          """
```

meth get_atoms_per_barn_cm

```
get_atoms_per_barn_cm() -> dict[str, float]
```

Number density (atoms / barn-cm) per nuclide.

Returns:

Type	Description
`dict[str, float]`	Dict[str, float]: nuclide -> atoms / b-cm

Source code in `yamc/_core/__init__.pyi`

```
2058  def get_atoms_per_barn_cm(self) -> builtins.dict[builtins.str,
2059  builtins.float]:
2060      r"""
2061      Number density (atoms / barn-cm) per nuclide.
2062
2063      Returns:
2064          Dict[str, float]: nuclide -> atoms / b-cm
          """
```

meth activity

```
activity(*, by_nuclide: bool = False, per: Optional[str] = None) -> Any
```

Calculate the radioactive activity of the current material inventory.

The material must have a `volume` in cm³. Atom densities are converted to total atoms using `atoms/barn-cm * 1e24 * volume`. The decay data (`transmutation_decay_data`, which has no default) provides half-lives; each nuclide's activity is `N * ln(2) / half_life`. Stable nuclides are omitted.

Parameters:

Name	Type	Description	Default
`by_nuclide`	`bool`	Return a `dict[str, float]` by nuclide instead of the total.	`False`
`per`	`str | None`	`None` (default) for the total, which needs `volume`; `"cm3"` for Bq/cm³, which needs nothing; `"g"` for Bq/g, which needs only `density`.	`None`

Returns:

Type	Description
`Any`	float | dict[str, float]: Activity, in Bq when `per` is `None`,
`Any`	Bq/cm³ when it is `"cm3"` and Bq/g when it is `"g"`. The unit of
`Any`	a `by_nuclide` dict's values is the same.

Source code in `yamc/_core/__init__.pyi`

```
2065  def activity(self, *, by_nuclide: builtins.bool = False, per:
2066  typing.Optional[builtins.str] = None) -> typing.Any:
2067      r"""
2068      Calculate the radioactive activity of the current material inventory.
2069
2070      The material must have a ``volume`` in cm³. Atom densities are converted
2071      to total atoms using ``atoms/barn-cm * 1e24 * volume``. The decay data
2072      (``transmutation_decay_data``, which has no default) provides
2073      half-lives; each nuclide's activity is ``N * ln(2) / half_life``.
2074      Stable nuclides are omitted.
2075
2076      Args:
2077          by_nuclide (bool): Return a ``dict[str, float]`` by nuclide instead
2078              of the total.
2079          per (str | None): ``None`` (default) for the total, which needs
2080              ``volume``; ``"cm3"`` for Bq/cm³, which needs nothing; ``"g"``
2081              for Bq/g, which needs only ``density``.
2082
2083      Returns:
2084          float | dict[str, float]: Activity, in Bq when ``per`` is ``None``,
2085          Bq/cm³ when it is ``"cm3"`` and Bq/g when it is ``"g"``. The unit of
2086          a ``by_nuclide`` dict's values is the same.
      """
```

`meth` decay_heat

```
decay_heat(*, by_nuclide: bool = False, per: Optional[str] = None, component:
Optional[str] = None) -> Any
```

Calculate decay heat from the current material inventory.

The material must have a `volume` in cm³. Atom densities are converted to total atoms using `atoms/barn-cm * 1e24 * volume`. The decay data (`transmutation_decay_data`, which has no default) provides half-lives and mean decay energies.

Parameters:

Name	Type	Description	Default
`by_nuclide`	`bool`	Return a `dict[str, float]` by nuclide instead of the total.	`False`
`per`	`str | None`	`None` (default) for the total, which needs `volume`; `"cm3"` for W/cm³, which needs nothing; `"g"` for W/g, which needs only `density`.	`None`
`component`	`str | None`	`None` (default) for the whole decay heat; `"beta"`, `"gamma"` or `"alpha"` for one recoverable-heat component alone (the ENDF MT=457 light-particle, electromagnetic and heavy-particle energies). The gamma heat is the part that leaves a thin component; the beta and alpha heat stays put. Raises if a nuclide making decay heat carries no split, which data converted before the split does, rather than understating the component by its share.	`None`

Returns:

Type	Description
`Any`	float | dict[str, float]: Decay heat, in W when `per` is `None`,
`Any`	W/cm³ when it is `"cm3"` and W/g when it is `"g"`. The unit of a
`Any`	`by_nuclide` dict's values is the same.

Source code in `yamc/_core/__init__.pyi`

```
2087  def decay_heat(self, *, by_nuclide: builtins.bool = False, per:
2088  typing.Optional[builtins.str] = None, component: typing.Optional[builtins.str]
2089  = None) -> typing.Any:
2090      r"""
2091      Calculate decay heat from the current material inventory.
2092
2093      The material must have a ``volume`` in cm³. Atom densities are converted
2094      to total atoms using ``atoms/barn-cm * 1e24 * volume``. The decay data
2095      (``transmutation_decay_data``, which has no default) provides
2096      half-lives and mean decay energies.
2097
2098      Args:
2099          by_nuclide (bool): Return a ``dict[str, float]`` by nuclide instead
2100              of the total.
2101          per (str | None): ``None`` (default) for the total, which needs
2102              ``volume``; ``"cm3"`` for W/cm³, which needs nothing; ``"g"``
2103              for W/g, which needs only ``density``.
2104          component (str | None): ``None`` (default) for the whole decay heat;
2105              ``"beta"``, ``"gamma"`` or ``"alpha"`` for one recoverable-heat
2106              component alone (the ENDF MT=457 light-particle, electromagnetic
2107              and heavy-particle energies). The gamma heat is the part that
2108              leaves a thin component; the beta and alpha heat stays put.
2109              Raises if a nuclide making decay heat carries no split, which
2110              data converted before the split does, rather than understating
2111              the component by its share.
2112
2113      Returns:
2114          float | dict[str, float]: Decay heat, in W when ``per`` is ``None``,
2115          W/cm³ when it is ``"cm3"`` and W/g when it is ``"g"``. The unit of a
              ``by_nuclide`` dict's values is the same.
          """
```

`meth` decay_photon_spectrum

```
decay_photon_spectrum(*, per: Optional[str] = None) -> tuple[list[float],
list[float]]
```

The decay photon line spectrum of the current inventory.

The material must have a `volume` in cm³. Each nuclide's atom count multiplies the line intensities the transmutation chain records for it, which are per atom per second (the emission probability already times the decay constant), so the values are photons per second rather than probabilities. Lines at the same energy are summed and the result is ascending in energy. Stable nuclides, nuclides the chain does not know and non-photon sources contribute nothing.

Lines only. Part of some decay spectra is a continuum, a density per eV rather than a set of lines, and `decay_photon_continua()` returns it.

The two lists are the `(x, p)` pair the source distributions take, so the lines round-trip straight into a photon transport source:

```
>>> energies, rates = activated.decay_photon_spectrum()
>>> source = PhotonSource(energy=sources.Discrete(energies, rates))
```

`Discrete` normalizes the weights, so the shape is what transport samples. `sum(rates)` is the line emission rate (photons/s) only, the rate that scales tallies from this source. That source leaves out every continuum, which for a continuum emitter can be most of its photons (all of them for Sm158 in ENDF/B-VIII.1); the continua emit `sum(c.emission_rate for c in activated.decay_photon_continua())` more. No `sources` distribution yet holds a linear-linear continuum exactly, so a transport source cannot yet carry every continuum.

Returns:

Type	Description
`list[float]`	tuple[list[float], list[float]]: Line energies (eV) and their
`list[float]`	emission rates (photons/s).

Source code in `yamc/_core/__init__.pyi`

```
2116  def decay_photon_spectrum(self, *, per: typing.Optional[builtins.str] = None)
2117  -> tuple[builtins.list[builtins.float], builtins.list[builtins.float]]:
2118      r"""
2119      The decay photon line spectrum of the current inventory.
2120
2121      The material must have a ``volume`` in cm³. Each nuclide's atom count
2122      multiplies the line intensities the transmutation chain records for it,
2123      which are per atom per second (the emission probability already times
2124      the decay constant), so the values are photons per second rather than
2125      probabilities. Lines at the same energy are summed and the result is
2126      ascending in energy. Stable nuclides, nuclides the chain does not know
2127      and non-photon sources contribute nothing.
2128
2129      Lines only. Part of some decay spectra is a continuum, a density per eV
2130      rather than a set of lines, and ``decay_photon_continua()`` returns it.
2131
2132      The two lists are the ``(x, p)`` pair the source distributions take, so
2133      the lines round-trip straight into a photon transport source:
2134
2135          >>> energies, rates = activated.decay_photon_spectrum()
2136          >>> source = PhotonSource(energy=sources.Discrete(energies, rates))
2137
2138      ``Discrete`` normalizes the weights, so the shape is what transport
2139      samples. ``sum(rates)`` is the line emission rate (photons/s) only,
2140      the rate that scales tallies from this source. That source leaves out
2141      every continuum, which for a continuum emitter can be most of its
2142      photons (all of them for Sm158 in ENDF/B-VIII.1); the continua emit
2143      ``sum(c.emission_rate for c in activated.decay_photon_continua())``
2144      more. No ``sources`` distribution yet holds a linear-linear continuum
2145      exactly, so a transport source cannot yet carry every continuum.
2146
2147      Returns:
2148          tuple[list[float], list[float]]: Line energies (eV) and their
2149          emission rates (photons/s).
          """
```

`meth` decay_photon_continua

```
decay_photon_continua(*, per: Optional[str] = None) -> list[PhotonContinuum]
```

The decay photon continua of the current inventory, one per nuclide and continuum, in nuclide-name order.

The part of a decay spectrum ENDF gives as a density over energy rather than as lines, which `decay_photon_spectrum()` does not include. Each continuum keeps its own energy grid and interpolation law, and its rates are photons/s/eV (or per cm³ or per g, following `per`), so they are not line rates; `emission_rate` is the integral. A material whose nuclides have no continuum returns an empty list.

Parameters:

Name	Type	Description	Default
`per`	`str | None`	None for the whole material, which needs a `volume`, or `'cm3'` or `'g'`, as for `decay_photon_spectrum()`.	`None`

Returns:

Type	Description
`list[PhotonContinuum]`	list[PhotonContinuum]: The continua, each with its nuclide,
`list[PhotonContinuum]`	energies, rates, interpolation law and emission rate.

Examples:

```
>>> for continuum in activated.decay_photon_continua():
...     print(continuum.nuclide, continuum.emission_rate)
```

Source code in `yamc/_core/__init__.pyi`

```
2150  def decay_photon_continua(self, *, per: typing.Optional[builtins.str] = None)
2151  -> builtins.list[PhotonContinuum]:
2152      r"""
2153      The decay photon continua of the current inventory, one per nuclide
2154      and continuum, in nuclide-name order.
2155
2156      The part of a decay spectrum ENDF gives as a density over energy rather
2157      than as lines, which ``decay_photon_spectrum()`` does not include. Each
2158      continuum keeps its own energy grid and interpolation law, and its
2159      rates are photons/s/eV (or per cm³ or per g, following ``per``), so
2160      they are not line rates; ``emission_rate`` is the integral. A material
2161      whose nuclides have no continuum returns an empty list.
2162
2163      Args:
2164          per (str | None): None for the whole material, which needs a
2165              ``volume``, or ``'cm3'`` or ``'g'``, as for
2166              ``decay_photon_spectrum()``.
2167
2168      Returns:
2169          list[PhotonContinuum]: The continua, each with its nuclide,
2170          energies, rates, interpolation law and emission rate.
2171
2172      Examples:
2173          >>> for continuum in activated.decay_photon_continua():
2174          ...     print(continuum.nuclide, continuum.emission_rate)
      """
```

meth contact_dose

```
contact_dose(*, dose_quantity: str = 'absorbed-air', build_up: float = 2.0,
by_nuclide: bool = False) -> Any
```

Contact dose rate from the material's own decay photons.

The dose someone receives with a hand on the material. It is a slab estimate rather than a transport result: the material is taken to be a half-space, so half the photons emitted at any depth head for the surface and the ones that arrive are the ones the material did not attenuate itself. That leaves no distance and no volume in the answer -- unlike `activity` and `decay_heat`, this needs no `material.volume` and a bigger lump of the same material reads the same.

Per photon line of energy `E` and per-atom emission rate `S` the estimate is `(build_up / 2) * (response(E) / mu_material(E)) * S * E` for the absorbed dose in air, and the same without the trailing `E` for the effective dose. `mu_material` is the material's own linear attenuation coefficient, built from the NIST XCOM mass attenuation coefficients of the elements present; the response is the NIST-126 mass energy-absorption coefficient of air, or the ICRP-116 photon effective-dose coefficient for anterior-posterior irradiation.

Follows the FISPACT-II manual (UKAEA-CCFE-RE(21)02, Appendix C.7.1) for the absorbed-air quantity. For photon lines it agrees with OpenMC's `Material.get_photon_contact_dose_rate`. A photon continuum is integrated exactly under its evaluated interpolation law.

Bremsstrahlung from decay electrons is not modelled, and nuclides whose radiation the chain file does not describe contribute nothing. Photon lines outside the tabulated range (1 keV to 20 MeV for the absorbed-air quantity, 10 keV to 20 MeV for the effective dose) are dropped, and a continuum is integrated over its part of that range.

Parameters:

Name	Type	Description	Default
`dose_quantity`	`str`	`'absorbed-air'` for the absorbed dose in air in Gy/h (the default, following FISPACT-II), or `'effective'` for the ICRP-116 effective dose in Sv/h.	`'absorbed-air'`
`build_up`	`float`	Build-up factor standing in for the photons that scatter in the slab and still arrive. Default 2.0, as the FISPACT-II manual suggests.	`2.0`
`by_nuclide`	`bool`	Return a `dict[str, float]` by nuclide instead of the total. Nuclides that contribute nothing are left out.	`False`

Returns:

Type	Description
`Any`	float | dict[str, float]: Contact dose rate in Gy/h
`Any`	(`'absorbed-air'`) or Sv/h (`'effective'`).

Raises:

Type	Description
`ValueError`	If a nuclide in the material has a photon continuum in the dose tables' range that this build cannot integrate: one whose data states no interpolation law, as transmutation data written before the law was stored does, one tabulated under a law other than histogram or linear-linear, or one whose energy and rate lists are unpaired, whose energies are not finite or descend, or whose rates are negative or not finite. Its integral is unknown, and leaving it out would understate the dose.

Examples:

```
>>> activated.contact_dose()
380.088...
>>> activated.contact_dose(by_nuclide=True)
{'Co60': 380.088...}
>>> activated.contact_dose(dose_quantity='effective')
380.728...
```

Source code in `yamc/_core/__init__.pyi`

▼ Details