YANI

YANI (Yet Another Nuclide Inventory) computes transmutation and activation. A material, an irradiation schedule and a neutron spectrum go in; inventories, activities, decay heat, decay photon spectra and displacement damage (dpa) come out.

```
pip install yani
```

Try it without installing anything

yani-online runs YANI in the browser: build a material, build an irradiation and cooling schedule, run a real transmutation, and read back inventory, activity, decay heat, contact dose rate and decay photon lines.

It is the same Rust solver compiled to WebAssembly, against the same continuous-energy cross sections, so it answers what the wheel answers rather than a reduced version of it. The solve runs on your machine and nothing is uploaded. The page will write your setup out as a Python script when you want to carry it over here.

What it solves

The nuclide inventory of a material under irradiation is a system of coupled linear ODEs, the Bateman equations with reaction terms:

$$\frac{\mathrm{d}N_i}{\mathrm{d}t} = \underbrace{\sum_j (\lambda_{j\to i} + \sigma_{j\to i}\,\phi) N_j}_{\text{production}} - \underbrace{(\lambda_i + \sigma_i\,\phi) N_i}_{\text{loss}}$$

where N_i is the atom density of nuclide i, $\lambda_{j\to i}$ the decay constant of each parent j that decays into it, $\sigma_{j\to i}$ the microscopic cross section of each reaction on j that produces it, and ϕ the scalar flux. Every nuclide is produced by both routes, decay and neutron reaction, and lost by both.

Assembled over every nuclide in the network this is a matrix A, and one timestep is a matrix exponential:

$$\mathbf{N}(t) = e^{At}\,\mathbf{N}(0)$$

which YANI evaluates with CRAM (the Chebyshev Rational Approximation Method) at order 48, following Pusa 2010 and Pusa 2015, using a sparse LU factorization whose symbolic factorization is reused across the poles. Reaction rates are held at their beginning-of-step values, so a schedule of n steps is n matrix exponentials per material.

The coefficients of A come from a transmutation network (decay constants, branching, reaction products, fission yields) and from folding the spectrum against continuous-energy cross sections to get each $\sigma\phi$.

Nuclear data libraries describes the six available libraries and what each contains. Method describes how the network, the collapse, self-shielding, schedules, derived quantities (activity, decay heat, photon emission, contact dose, clearance, displacement damage) and uncertainty propagation work.

Whether the answers are right

YANI is run against 497 measured cases from two IAEA CoNDERC archives, both of which publish another inventory code's results beside the measurement, so a disagreement can be attributed rather than merely noted. Every case has a figure, every number has the script that produced it, and what is *not* covered is listed too. See Validation.

Why YANI

Four things, in order of how often they decide the question: it has to be easy to run, the numbers have to be right, it has to be fast enough to use in a loop, and it has to bend to the problem you actually have rather than the one a tool author imagined.

Easy

`pip install yani`, on Linux, macOS or Windows. The wheel carries the compiled solver. There is no compiler to invoke, no transport stack to build first, and no system package to hunt down.

Nuclear data downloads on demand. Point a setting at a library keyword and the sections your calculation actually needs are fetched and cached on first use, not a whole evaluation up front. See Where the data comes from.

The license stack is fully permissive. YANI is dual-licensed under MIT or Apache-2.0, at your option, which is the usual arrangement in the Rust ecosystem and means a review board that wants Apache-2.0's explicit patent grant can have it without asking anyone. Nothing in the dependency chain forces a copyleft license onto whatever you build with it, so using it in a closed pipeline or a commercial tool is not a licensing question, just an engineering one.

It runs in a browser too, with nothing installed at all. The solver is Rust and compiles to `wasm32` cleanly, so the browser build is that same solver against the same continuous-energy data, not a reduced version of it. There is no server, no account and no upload: the calculation runs on your own machine, and the page is a single file you can save and open offline. It is the fastest way to try YANI before installing anything, and the easiest way to hand a colleague a worked example, since a link carries the whole setup.

A citable material library ships in the wheel. The PNNL Compendium (PNNL-15870 Rev. 2), 410 named materials with their compositions and densities, is bundled rather than a separate download, so a standard shielding or structural material is `yani.materials.pnnl.material(key="Steel, Stainless 316", volume=...)` away instead of a retyping exercise from someone else's PDF. See Materials.

Accurate

Continuous-energy cross sections, not a multigroup collapse. A multigroup library has already averaged the cross section over each group using someone else's assumed spectrum shape, and resonance structure inside a group is gone before you ever see it. YANI folds your actual spectrum against the pointwise cross section directly, at the energy resolution the evaluation was measured at, so the only averaging is the one your own spectrum implies. This matters most exactly where it is usually skipped: narrow resonances that a coarse group structure smears across a bin.

An uncertainty on the answer, not just the answer. Pass `data_uncertainty=` and every nuclide density comes back with a standard deviation beside it. The activation cross sections are resampled from the evaluation's own ENDF MF=33 covariance, folded against your spectrum, and their resonance parameters from MF=32 where the library carries them; the half-lives, decay energies and decay branching ratios from the decay data's own sigmas; the isomeric splits from MF=40; and the schedule is re-solved for each sample. That is exact to all orders in the matrix exponential: nothing is linearized, and the sandwich rule is not used, so correlations between nuclides survive. When a replica moves a channel, the channel's product moves against its target and together with the nuclides the product decays to, and a quantity summed over them keeps both: the spreads of the first pair partly cancel and those of the second add. If your spectrum arrives from a Monte Carlo run with an error on it, hand it to the pulse as `flux_std_dev`, or as a full `flux_covariance`, and that propagates too, as its own source, so you can see which source dominates.

Every quantity you would put in a report carries a band from the same ensemble. Activity, decay heat, contact dose and every decay photon line come back as an `Estimate`, a nominal value with the ensemble's spread beside it. Each is evaluated once per replica, with that replica's own half-lives, and summed inside that replica: adding the per-nuclide sigmas in quadrature double-counts a variance that partly cancels, and contact dose is not even linear in the densities, since a replica that makes more of an emitter also absorbs more of it. Each line's emission per decay and energy are drawn from the decay data's own sigmas, so a photon line's band is its emitters' activity and its own intensity together. The attenuation, response and build-up behind contact dose are the same in every replica. A calculated band beside a measured one says whether the disagreement is larger than the propagated data allows.

A sigma of zero says which kind of zero it is. The hard part of an uncertainty is not producing one, it is knowing what it left out. A nuclide whose evaluation publishes no covariance and a nuclide whose covariance is genuinely small would otherwise both report `0.0`, and only one of those is reassuring. `get_data_uncertainty_info` keeps them apart: which nuclides were perturbed, which have no usable published covariance, what share of each reaction rate carries a nonzero stated variance, which evaluated matrices were not positive semi-definite and had to be repaired (a repair that keeps every evaluated sigma and moves only correlations, with each channel's evaluated sigma beside the one sampled), and which sources were held at nominal (What is not propagated explains each one). On ENDF/B-VIII.1 the second point is not academic (48% of evaluations carry cross-section covariance, against 100% of TENDL-2025), so the same run on two libraries can give two very different sigmas, and the report is what tells you why. What it separates is an evaluation with no MF=33 from one whose MF=33 is small, and a capture whose MF=33 is zero over the resonance range, because the evaluation keeps that uncertainty in resonance-parameter covariance (MF=32), reads as uncovered there when the data carries no resonance parameters (ENDF/B-VIII.1 W186 about 0.07 on the FNS spectrum from MF=33 alone, and 1 on the published library, which samples them). `rate_fraction_covered_total` weights coverage by reaction rate and by parent density instead of counting evaluations.

Resonance self-shielding from a slowing-down solve, with a warning when you skip it. Give a lump its shape, `yani.shapes.FoilLump(thickness=0.1)`, or its mean chord, and the flux inside it is depressed where the total cross section is large. The solve assumes nothing about resonances being narrow: the cheaper narrow-resonance approximation is not offered, because it over-shields strong elastic scatterers badly enough to be worse than applying no correction at all. Leave the correction out, which

is the default since a material carries no geometry, and the run still reports what it skipped: `would_shield` in `get_self_shielding_info` names the resonance absorbers it left uncorrected and how strongly their own resonances could bite. That is a screen rather than a bound, since it carries no geometry, so it says which answers to distrust rather than by how much. See Self-shielding.

Gamma lines, not a binned photon response. `decay_photon_spectrum()` returns discrete energy/intensity pairs, not a spectrum pre-collapsed onto a group structure. A binned response answers "how much energy in this window"; lines answer "which nuclide is that", which is what you need for identifying an isotope from an emitted spectrum or feeding a photon transport run without inheriting someone else's binning choice.

Displacement damage from the same solve, with an uncertainty. Pass `displacement_damage=True` and the run also returns damage energy and NRT dpa, per element and for the material, cumulative over the schedule. The MT=444 damage-energy cross section is folded against your spectrum by the same continuous-energy collapse as the reaction rates, self-shielding included, and at each step's composition, so transmutation products count as they build up. With `data_uncertainty=` the dpa gets a band from the same ensemble as everything else, and the report says how much of each nuclide's damage rate the sampled reactions account for. See Displacement damage.

Clearance and disposal limits from the same inventory. `clearance_index()` checks a material against the UK (EPR 2016, IRR 2017), German (StrlSchV), US (NRC 10 CFR 61, Fetter), EU Basic Safety Standards and IAEA GSR Part 3 limit sets, with the transmutation chain's own half-lives, and reports the activity no limit in the set covers so a low index cannot hide a gap. See Clearance and disposal limits.

Production routes come back with a share on each one. `get_production_routes()` lists the routes into a product and weights each by what the material's own reactions drove over the step. A chain on its own can do neither: it offers `Os190(n,a)` for W187 as readily as `W186(n,gamma)`, and nothing in a tungsten foil is osmium, and the branching on one of its edges is a share of that channel alone, so it cannot rank two routes. The result has the shape of a published pathway table. See Production routes.

Isomeric branching is tracked, not folded into the ground state, and it is flux-weighted rather than a fixed number. A reaction that leaves a fraction of its product in a metastable state is modeled as a branch to that state, with its own half-life and its own decay chain, rather than lumped into the ground-state inventory. The branching fraction itself is stored as a curve against incident energy, not a single evaluation-average ratio, and is folded against your actual spectrum at rate-compute time, so two calculations with different spectra but the same target nuclide can legitimately get different branching fractions, because they should. `Ag110_m1` dominating the activity of a silver foil for years after `Ag110` itself has decayed away, as in the getting started example, is the kind of result that disappears if branching is not tracked. `get_isomeric_branching()` reports the split each step was solved with, which separates a disagreement caused by a cross section from one caused by a branching ratio. See Weighted, from a solve.

A reaction rate can be read back per energy group, not just as one number. `get_reaction_rate_spectrum()` resolves one channel's rate onto the groups of the spectrum that drove it, and the entries sum to the collapsed rate because both come from the same walk of the same cross sections. A capture cross section spans decades, so an effective one-group value of tens of

millibarns against a 14 MeV-dominated spectrum is either fast capture or resonance capture, and only the breakdown says which. That is the difference between a disagreement that belongs to the resonance processing and one that belongs to the fast cross section. Set against where a covariance states a nonzero variance, it also shows, to group resolution, where in energy the share that `rate_fraction_covered` reports as one number comes from, and on a shielded run it says which groups the depression moved. See Where in energy a rate came from.

Verification and validation runs against every open benchmark we have found, CoNDERC among them. The rest is a todo, honestly labeled: if you know of an open, reproducible benchmark this list is missing, tell us and we will add a reproducible V&V script for it.

Fast

No transport solve stands between your spectrum and an answer. Point a `Material` at whatever spectrum you already have and call `transmute()`; there is no geometry to build and no particle transport run to get through first. YANI folds continuous-energy cross sections against your spectrum directly, as one function call.

Parsed nuclear data stays loaded in RAM, not reloaded from disk. Cross section libraries and transmutation chains are parsed once per source, a library keyword or a file path, into a process-wide cache keyed on exactly that, and reused from then on. Looping over many materials, spectra or schedule variants in one script pays the parse cost once, not once per `transmute()` call, and a later call that needs a wider slice of the same source (say, more reactions) tops the cache up rather than re-parsing it from scratch. The cross sections a `transmute()` needs are kept on the material itself as well, so a second call on the same material does no file reading at all. A sweep over thousands of distinct compositions hands them back with `release_nuclear_data()`. Downloads are cached separately, to disk in the platform cache directory (`yani.cache_dir()`), so a second run of the script does not refetch.

The chain that gets built is exactly the one your material can reach, no manual depth setting required. Rather than truncating the transmutation network at a fixed number of reaction or decay steps from the starting nuclides, a value that silently drops real nuclides if set too low and loads data no calculation needs if set too high, YANI finds the products that matter by solving: it loads the composition's cross sections, solves the schedule, loads every product that reached `product_threshold` (1e-12 by default) as an atom fraction or as a share of the activity, decay heat or decay photon power, and repeats until nothing new crosses it. A year's irradiation of a 316 steel against ENDF/B-VIII.1 loads 99 nuclides that way, of the 465 its network can reach. A product left out is still made, decays and is reported, and only its own reactions are dropped, by an amount `get_product_screening_info` bounds; `product_threshold=0` loads every product the solve populates, which is exact.

Nuclear data is a binary columnar format, not text to parse. Cross sections and transmutation chains are stored as Arrow, so loading them is reading typed columns straight into memory rather than parsing fixed-width text records the way an ENDF tape has to be. Converting from ENDF or ACE is a one-time step; every calculation after that reads the fast format.

Many materials in one call. `yani.transmute(materials=..., schedules=...)` solves a whole mesh, a component split into regions or a sweep over compositions together. The chain is read once, each nuclide's cross sections are decoded once for every material that needs them, a collapse runs once per distinct spectrum, composition, temperature and shielding, and the solves run in parallel. Each material gets exactly what its own `transmute()` would give it. See Several materials at once.

Flexible

The same solver targets native and the browser. The Rust core builds for `wasm32` as a first-class target, not a native tool with a JS wrapper bolted on afterward, so the identical solver behind `pip install yani` can also run client-side with no backend to stand up. Where a calculation runs is a deployment choice, not a rewrite.

Four independently sourced transmutation subsections, mixed at will. `transmutation_reactions`, `transmutation_decay_data`, `transmutation_fission_yields` and `transmutation_branch_ratios` are each their own setting, so a network can mix libraries by subsection instead of committing to one library for everything. A TENDL reactions network borrowing decay data and fission yields from ENDF/B-VIII.1, the usual arrangement since TENDL publishes neither, is a first-class configuration, not a data file hack. See Nuclear data settings.

A schedule can mix spectra, not just magnitudes. Every `Pulse` carries its own `source`, so a campaign whose neutron spectrum changes shape between phases, not just intensity, is expressed directly instead of approximated with one averaged spectrum. A fusion reactor's operating life is the obvious case: DD-phase pulses and DT-phase pulses each get their own spectrum in the same `PulseSchedule`, back to back, and the network sees the real spectrum shape for each phase. See Schedules.

Any spectrum group structure works, not just a preset list. A `Histogram` takes whatever ascending list of energy boundaries your spectrum happens to use, of whatever length, and named structures like `CCFE-709` are a convenience for a handful of common cases, never a requirement. See Named group structures.

Local ENDF files, local Arrow files and premade Arrow files, in any combination. Convert your own ENDF evaluations, point settings at your own converted directories, or use a downloaded keyword, and mix all three within one calculation: `neutron_cross_section_data` alone can be set per-nuclide, so one network can pull cross sections from a local file for the isotope you care about and from a downloaded library for everything else. See Making your own data and Nuclear data settings.

Getting started

Install

```
pip install yani
```

The wheel carries the compiled solver, so there is nothing to build and no transport stack to pull in.

`yani.__version__` gives the installed version, and `yani.__commit__` the full git hash the compiled extension was built from (`None` if that was not recorded). `yani.__commit_dirty__` is `True` when the build had uncommitted changes to tracked files. Quote all three when reporting a problem.

If you would rather not install yet, yani-online runs the same solver in your browser and will write the setup you build there back out as a Python script.

Point it at nuclear data

Two kinds of data are needed, and they are configured separately.

Neutron cross sections, for the `sigma * phi` reaction rates, come from a continuous-energy library:

```
import yani

yani.neutron_cross_section_data = "tendl-2025"
```

The transmutation network, for the decay constants, reaction products, fission yields and isomeric branching, is assembled from four independently sourced subsections, so each is set on its own and a network can mix libraries:

```
yani.transmutation_reactions = "tendl-2025"
yani.transmutation_branch_ratios = "tendl-2025"
yani.transmutation_decay_data = "endf-b8.1"
yani.transmutation_fission_yields = "endf-b8.1"
```

The split above is not arbitrary. TENDL has no decay or fission-yield sublibrary, so those two come from a library that does. Its gamma evaluations are photonuclear data, separate from the neutron data used for activation; see the next TENDL release. It is still what you want for `reactions`, because it covers far more parent nuclides, and that is what decides whether an activation product appears in your inventory at all.

Six libraries are available, and each publishes only some of what a calculation needs. Which does which, and how much is in each, is in Nuclear data libraries.

Pointing a subsection at a library that does not publish it fails immediately, with a message listing what that library does have, rather than failing partway through a download. Forgetting

`neutron_cross_section_data` fails too: it has no default, so an irradiation stops with a message naming the setting rather than running with no rates.

A keyword is all this page needs. Paths, per-nuclide dicts and where the download lands are in Where the data comes from.

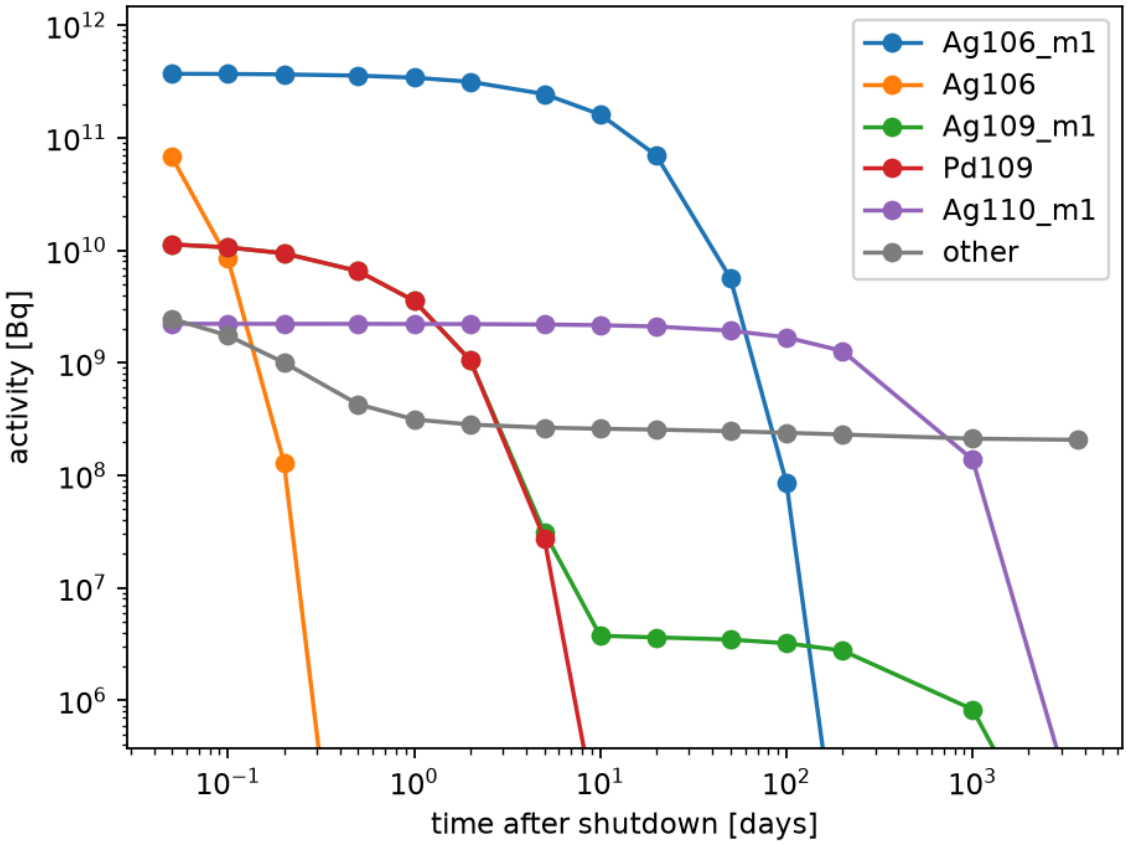
A first calculation

An irradiation schedule, a spectrum, and one call:

```
import yani

yani.neutron_cross_section_data = "tendl-2025"
yani.transmutation_reactions = "tendl-2025"
yani.transmutation_branch_ratios = "tendl-2025"
yani.transmutation_decay_data = "endf-b8.1"      # TENDL has no decay data
yani.transmutation_fission_yields = "endf-b8.1"  # nor fission yields

# A 1 cm3 silver foil. Giving the element rather than the nuclides expands it
# over natural abundance, here Ag107 (51.8%) and Ag109 (48.2%).
#
# A volume is required for anything extensive (activity, decay heat, photon
# lines): the solver works in atoms/barn-cm and the volume turns that into atoms.
foil = yani.Material(composition={"Ag": 1.0}, density=10.49, volume=1.0)

# The spectrum rides on the pulse. The Histogram normalizes the shape, so a
# multigroup flux from a tally can be passed straight in; the pulse `rate` is the
# total flux magnitude in n/cm2/s.
spectrum = yani.NeutronSource(
    energy=yani.sources.Histogram(
        boundaries=[1e-5, 1e5, 1e6, 1.5e7], probabilities=[1e12, 1e13, 1e14]
    )
)
# Sample the decay at these times after shutdown, in days. A Cooldown takes the
# duration OF THAT STEP, so the schedule needs the gaps between them.
days = [0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 1000, 3650]
gaps = [0.05, 0.05, 0.1, 0.3, 0.5, 1, 3, 5, 10, 30, 50, 100, 800, 2650]

schedule = yani.PulseSchedule(
    steps=[yani.Pulse(rate=1.11e14, duration=(100, "d"), source=spectrum)]  # 100 days
on
    + [yani.Cooldown(duration=(g, "d")) for g in gaps]                      # then
cooling
)

results = foil.transmute(schedule=schedule)    # TransmutationResults
steps = results.step_materials(material_id=foil.id)   # one Material per step
final = steps[-1]

print(final.activity(), "Bq")
print(final.decay_heat(), "W")
print(final.contact_dose(), "Gy/h")
```

What comes out

Ask for the results by nuclide and plot them and the shape of an activation problem appears: which product dominates depends entirely on how long you wait.

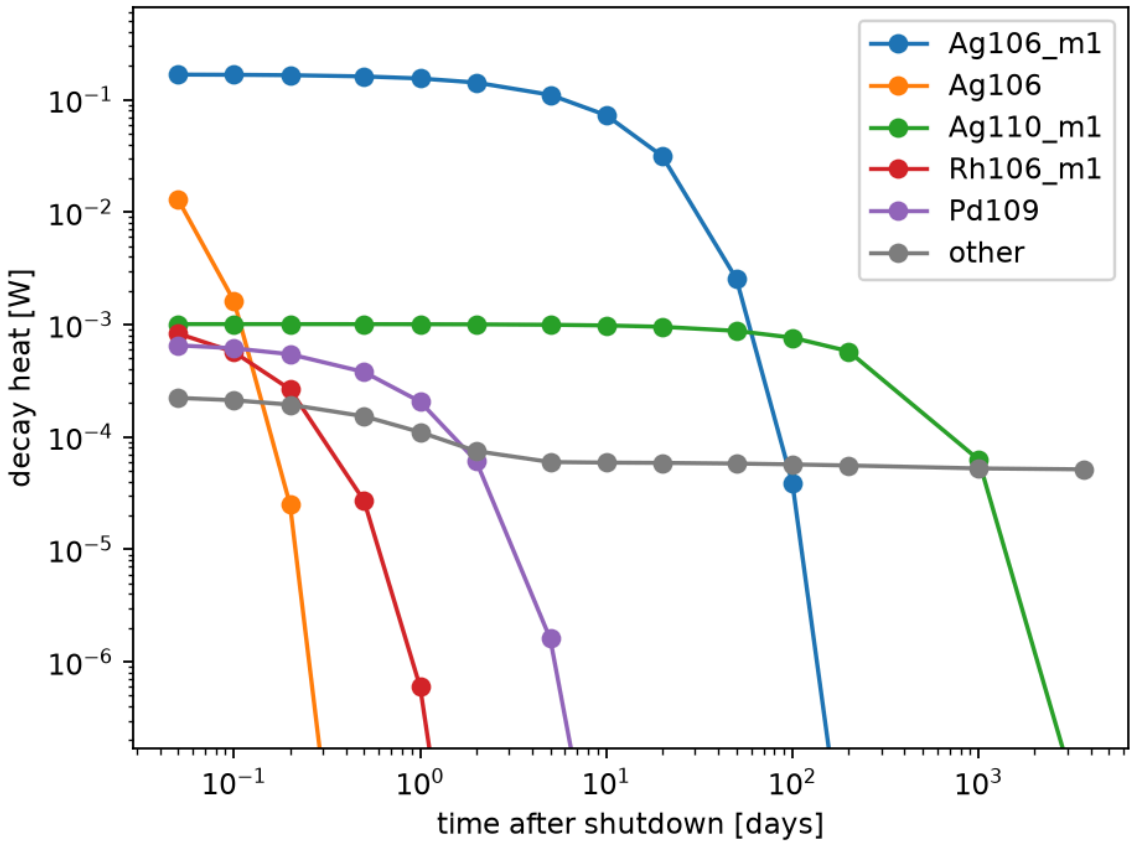


`Ag106_m1` carries the activity for the first couple of months, then `Ag110_m1` takes over for a few hundred days, so what dominates depends on when you look. `Ag106` and `Pd109` are gone within days. Past a few years `Ag110_m1` has decayed too and `Ag108_m1` is what is left: it never peaks high enough to make the five largest, so the grey "other" line is the one still carrying the activity at the right-hand edge.

Plotting code

```
import matplotlib.pyplot as plt

cooled = steps[1:]                          # drop the irradiation step
per_step = [m.activity(by_nuclide=True) for m in cooled]

# The five largest by peak value, everything else summed into "other".
keys = {k for d in per_step for k in d}
peak = {k: max(d.get(k, 0.0) for d in per_step) for k in keys}
top = [k for k, _ in sorted(peak.items(), key=lambda kv: -kv[1])[:5]]

fig, ax = plt.subplots()
for name in top:
    ax.plot(days, [d.get(name, 0.0) for d in per_step], marker="o", label=name)
ax.plot(days, [sum(v for k, v in d.items() if k not in top) for d in per_step],
        marker="o", color="grey", label="other")

ax.set_xscale("log"); ax.set_yscale("log")
# Show six decades. Anything below that has decayed away, and letting it set
# the y limits squashes the rest of the plot.
ax.set_ylim(max(peak.values()) / 1e6, max(peak.values()) * 4)
ax.set_xlabel("time after shutdown [days]"); ax.set_ylabel("activity [Bq]")
ax.legend()
```

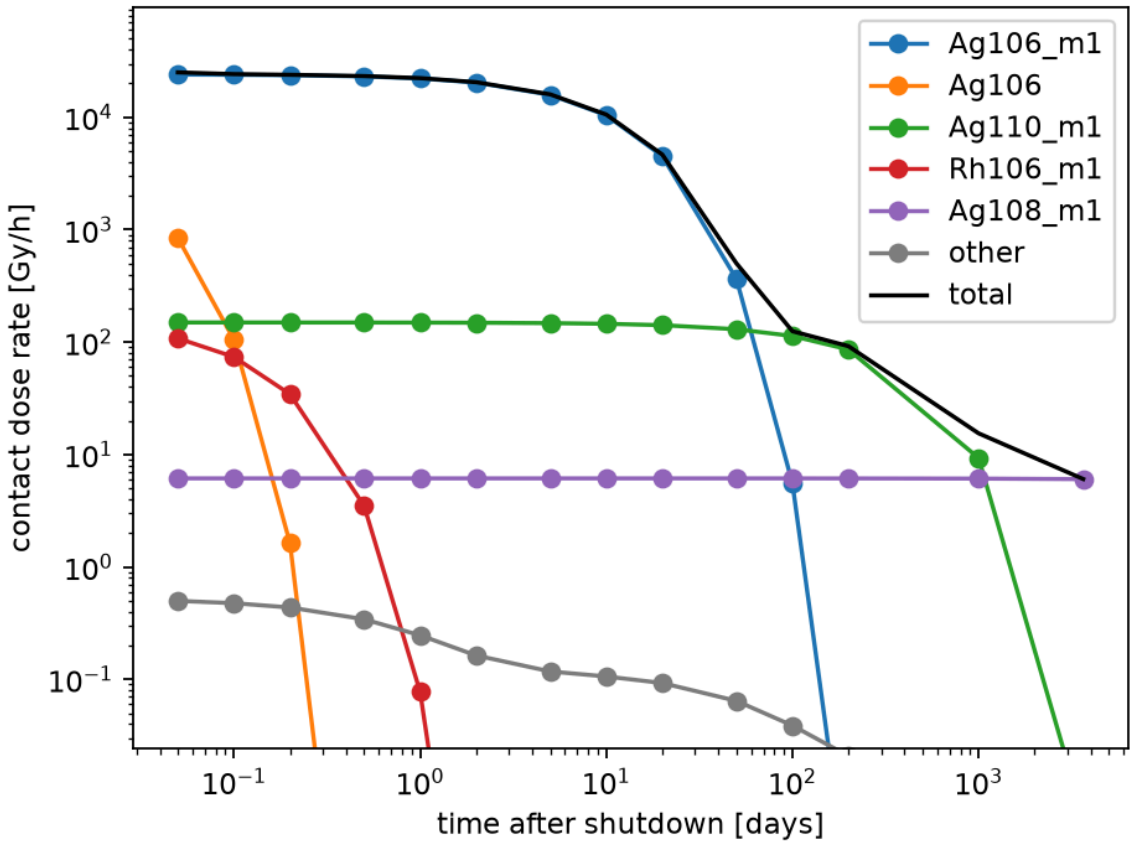
Decay heat is the same call with a different observable, and tells a different story: `Ag110_m1` climbs from fifth place to third and `Ag109_m1` leaves the five altogether, replaced by `Rh106_m1`. What heats the material is energy per decay, not decays per second, and an isomeric transition is a cheap decay.



Plotting code

```
import matplotlib.pyplot as plt

cooled = steps[1:]                          # drop the irradiation step
per_step = [m.decay_heat(by_nuclide=True) for m in cooled]

# The five largest by peak value, everything else summed into "other".
keys = {k for d in per_step for k in d}
peak = {k: max(d.get(k, 0.0) for d in per_step) for k in keys}
top = [k for k, _ in sorted(peak.items(), key=lambda kv: -kv[1])[:5]]

fig, ax = plt.subplots()
for name in top:
    ax.plot(days, [d.get(name, 0.0) for d in per_step], marker="o", label=name)
ax.plot(days, [sum(v for k, v in d.items() if k not in top) for d in per_step],
        marker="o", color="grey", label="other")

ax.set_xscale("log"); ax.set_yscale("log")
# Show six decades. Anything below that has decayed away, and letting it set
# the y limits squashes the rest of the plot.
ax.set_ylim(max(peak.values()) / 1e6, max(peak.values()) * 4)
ax.set_xlabel("time after shutdown [days]"); ax.set_ylabel("decay heat [W]")
ax.legend()
```

Contact dose rate is a third reading of the same inventory, and the only one of the three that needs no `volume`: it is a slab estimate, so the geometry cancels and a bigger piece of the same foil reads the same with a hand on it. Its breakdown is always a subset of the activity's, because only the gamma emitters reach it -- a nuclide that carries the activity without emitting photons drops out of this plot entirely, which is what makes the two curves worth reading side by side.



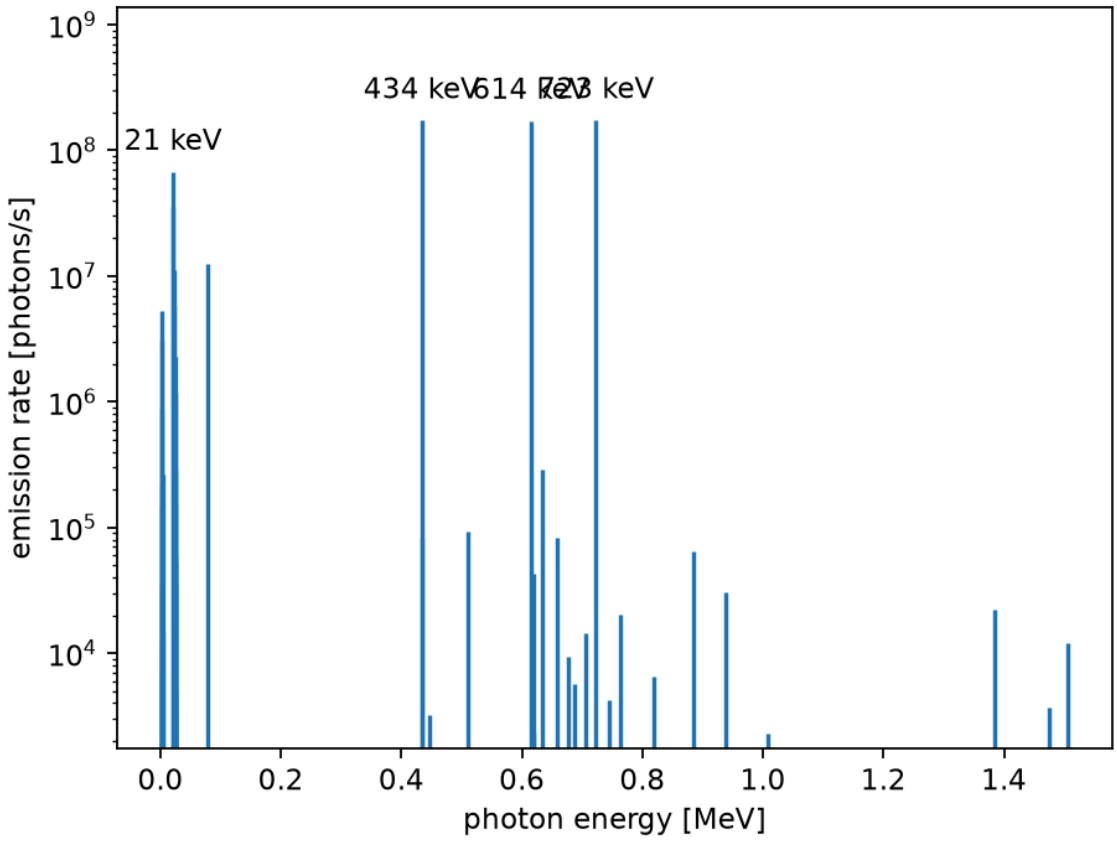
Here that costs `Ag109_m1` and `Pd109` their places: third and fourth in the activity, neither makes the five largest by dose, and `Rh106_m1` and `Ag108_m1` take their seats. Nine of the seventy-one nuclides carrying activity are simply absent: six emit no photons at all, `H3`, `Ru106` and `Pd107` among them, and three, `Ag104_m1`, `Rh100_m1` and `Rh108_m1`, are short-lived isomers whose decay records carry a gamma energy but no photon lines for the dose to use.

The dose itself is `Ag106_m1`'s out to twenty days, 96 to 99% of it, and `Ag110_m1`'s by a hundred. What the log axis then shows, and the other two plots have no equivalent of, is the flat line the total settles onto: `Ag108_m1` has a 438 year half-life, so across the whole ten years it neither grows nor decays, and at the right-hand edge it is the entire dose. Three and a half decades below the peak and last in the five for most of the plot, it is nonetheless what decides whether the foil can be handled once the rest has gone.

Plotting code

```
import matplotlib.pyplot as plt

cooled = steps[1:]                          # drop the irradiation step
per_step = [m.contact_dose(by_nuclide=True) for m in cooled]

# The five largest by peak value, everything else summed into "other".
keys = {k for d in per_step for k in d}
peak = {k: max(d.get(k, 0.0) for d in per_step) for k in keys}
top = [k for k, _ in sorted(peak.items(), key=lambda kv: -kv[1])[:5]]

fig, ax = plt.subplots()
for name in top:
    ax.plot(days, [d.get(name, 0.0) for d in per_step], marker="o", label=name)
ax.plot(days, [sum(v for k, v in d.items() if k not in top) for d in per_step],
        marker="o", color="grey", label="other")

# The total is a line of its own here, since "can this be handled" is a question
# about the sum rather than about which nuclide is responsible for it.
ax.plot(days, [m.contact_dose() for m in cooled], color="black", label="total")

ax.set_xscale("log"); ax.set_yscale("log")
ax.set_ylim(max(peak.values()) / 1e6, max(peak.values()) * 4)
ax.set_xlabel("time after shutdown [days]")
ax.set_ylabel("contact dose rate [Gy/h]")
ax.legend()
```

The decay photon spectrum is a set of discrete lines rather than a curve, so it wants stems. `steps[-1]` is ten years after shutdown, where the strongest lines are at 723, 434 and 614 keV, which are `Ag108_m1`'s gammas: this is the spectrum a detector outside the foil would see, and it identifies the nuclide.



Plotting code

```
import matplotlib.pyplot as plt

energies, intensities = steps[-1].decay_photon_spectrum()

# Several hundred lines come back. Keep the ones within five decades of the
# strongest; the weaker ones are too faint to plot.
pairs = sorted(zip(energies, intensities), key=lambda p: -p[1])
floor = pairs[0][1] / 1e5
keep = [(e / 1e6, i) for e, i in pairs if i >= floor]

fig, ax = plt.subplots()
ax.vlines([e for e, _ in keep], floor, [i for _, i in keep])
ax.set_yscale("log"); ax.set_ylim(floor, pairs[0][1] * 8)
ax.set_xlabel("photon energy [MeV]")
ax.set_ylabel("emission rate [photons/s]")

for e, i in keep[:4]:
    ax.annotate(f"{e * 1000:.0f} keV", xy=(e, i), xytext=(0, 8),
                textcoords="offset points", ha="center")
```

`days` above is the cumulative cooling time of each step, which is what the plots use for their x axis.

`transmute()` returns a `TransmutationResults`, keyed by material id. `step_materials()` unpacks it into one `Material` per timestep, in order, each carrying the inventory at the end of that step.

`get_final_material()` hands back the last of them directly, returning `None` rather than raising if the id is not one it stepped. They are ordinary materials, so anything you can ask a material you can ask a result:

```
print(final.activity(by_nuclide=True))     # {"Ag108_m1": ..., "Ag110_m1": ..., ...}
print(final.decay_heat(by_nuclide=True))   # W per nuclide
print(final.contact_dose(by_nuclide=True)) # Gy/h per nuclide
print(len(final.nuclides))                 # how much the network grew

energies, intensities = final.decay_photon_spectrum()   # photons/s per line
```

Next steps

- Usage for schedules, spectra, results and converting your own data.
- API reference for the full typed surface.

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 (`yani.cache_dir()` says where); only the sections activation reads are fetched, never a whole library. Releases, the cache layout and offline use are in Where the data comes from.

```
import yani

yani.neutron_cross_section_data = "tendl-2025"
yani.transmutation_reactions = "tendl-2025"
yani.transmutation_decay_data = "endf-b8.1"
yani.transmutation_fission_yields = "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 its lithium breeds no tritium.
`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 to a converted directory 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` takes a dict keyed by nuclide. Which setting does what, and what each one does when left alone, is in Nuclear data settings. Only `transmutation_branch_ratios` has a default; the others are refused when a calculation reads them unset.

Asking yani what a library holds

`yani.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 = yani.available_nuclear_data_libraries()
libraries["tendl-2025"]["decay"]              # False: take decay from another library
"Fe56" in libraries["tendl-2025"]["photonuclear"]
libraries["irdff-ii"]["isomeric_branching"]   # True
sorted(libraries["endf-b8.1"]["neutron_covariances"])
# ['angular_distribution', 'cross_section', 'fission_spectrum', 'nubar',
'resonance_parameters']
```

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

A calculation needs cross sections and four transmutation subsections, and each is set on its own, so a network can take each part from wherever publishes it. 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. 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 a calculation 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**, set by `neutron_cross_section_data`: cross sections against energy for each nuclide, folded with the spectrum into reaction rates. They decide how fast each reaction runs.
- **`decay`**, set by `transmutation_decay_data`: half-lives, decay modes, decay energies and decay photon lines. They decide how the inventory decays and what it emits.
- **`reactions`**, set by `transmutation_reactions`: which reaction on which nuclide makes which product, and its Q. It decides which 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.

TENDL has neutron and photonuclear evaluations, but no decay or fission-yield sublibrary, so those two must come from one of the three that do. It is still what you want for `reactions`, because it covers far more parent nuclides, and that is what decides whether an activation product appears in your inventory at all. `endf-b8.1`, `jeff-4.0` and `jendl-5.0` each supply a complete network from one library.

FENDL-3.2d publishes the same two subsections as TENDL, both built against ENDF/B-VIII.1 decay data, so it too takes its `decay` and `fission_yields` from elsewhere. Its `branching` subsection is the

main reason to use it: 54 of its evaluations carry isomer production (MF=9 or MF=10), 15 of them with MF=40 covariances.

Pointing a subsection at a library that does not publish it fails immediately, naming what that library does have, rather than solving to an inventory that is quietly missing a route.

The `cross sections` column lists different kinds of evaluated data. YANI's activation rates use neutron cross sections. Photoatomic data matters when a decay photon spectrum is transported in yamc; it is not the photonuclear data in TENDL. JEFF's photonuclear sublibrary is not processed as photoatomic data, and FENDL's photoatomic data carries no atomic relaxation.

Photonuclear data in the next TENDL release

The next TENDL-2025 and TENDL-2017 regeneration processes `TENDL-g.tgz` as well as the neutron archive. These gamma evaluations describe photonuclear reactions on nuclides, not photoatomic interactions on elements. The published `2026-10-12.1` release does not contain the new files; they become available only after the regenerated release is validated and published.

Output lives under `photonuclear/{Nuclide}.arrow/`:

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

The particle index and release manifest list the available evaluations and files. These are evaluated data, not transport-ready secondary distributions. `Material.transmute()` remains neutron-flux driven and does not consume these tables, and yamc does not yet sample photonuclear collisions. In particular, TENDL gamma data cannot replace a photoatomic `photon/` library.

For a local evaluation, the converter and readback are available on this wheel:

```
folder = yani.convert_photonuclear(
    input_path="g-Fe056.tendl", output_dir="out/photonuclear",
    library="tendl-2025",
)
summary = yani.read_photonuclear_from_arrow(folder)
print(summary["name"], summary["mts"])
```

The readback checks the Arrow cross sections against the preserved MF=3 source. Neither neutron NJOY processing nor auxiliary atomic tabulations are needed.

How big each library is

Counted from published release `2026-10-12.1`, not from the forthcoming regeneration. `--` means that release does not publish the category, rather than publishing an empty one. Photonuclear evaluation counts will come from the new release's index; the photoatomic counts below do not include them.

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

`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 the number that decides whether a product is reachable at all; the isomeric overlay adds more at solve time, 12,817 rather than 12,444 on `endf-b8.1`. Most of each count is the residual production of MT=5, the `(n,X)` edges described under Weighted, from a solve: 7,269 of ENDF/B-VIII.1's 12,444 and 347,357 of TENDL-2025's 415,154. `covariance` is the share of neutron evaluations carrying cross-section covariance.

The spread between the two ends is the thing worth noticing: TENDL carries five times the evaluations of ENDF/B-VIII.1 and thirty-three times the reaction edges, and ENDF/B-VIII.1 carries the decay data that TENDL has none of. That is why the usual arrangement is a TENDL `reactions` network with ENDF/B-VIII.1 `decay` and `fission_yields` beside it.

Mixing that way asks the decay subsection to cover everything the reactions network can reach, and the published pairings do: the TENDL-2025 network puts 3,366 nuclides in play, and both ENDF/B-VIII.1 and JENDL-5 carry decay data for every one of them. It is worth knowing what the miss would look like, because a hand-built or scoped chain can produce it. A product with no decay entry gets no decay term in the burnup matrix, so it accumulates and never decays, reports zero activity and zero heat, and is indistinguishable from a genuinely stable product.

Covariance is presence, not coverage

The percentage counts evaluations that carry cross-section covariance, which is not the same as covariance landing where your run needs it. Read `rate_fraction_covered_total` before any sigma: it weights each channel by the production it drove, so a channel stated with no covariance, or only on partial levels, reads as uncovered however many of the element's isotopes carry MF=33. Nuclear-data uncertainty has the cases.

Depth varies as much as presence. JENDL-5 carries covariance for only 13% of its evaluations, and its Fe56 covers 56 channels against ENDF/B-VIII.1's 7.

Three covariance families reach the data. The cross sections' MF=33 is folded against your spectrum. Resonance-parameter covariance (MF=32), where many evaluations keep their whole resonance-range uncertainty, is used where the library carries the parameters themselves (`resonance_parameters.arrow`): each replica draws them and rebuilds the resonance range from the draw. A nuclide whose parameters cannot be sampled falls back to the first-order MF=32 blocks the converter writes into `covariance.arrow`. On data with neither, a capture rate driven by resonance flux can carry a sigma near zero on a library whose MF=33 is zero there, and the coverage report shows that case: a zero-valued MF=33 interval counts as uncovered, so such a capture reads a `rate_fraction_covered` well below 1 (about 0.07 for ENDF/B-VIII.1 W186 on the FNS spectrum, with MF=33 alone). The isomeric splits' MF=40 covariance, in the branching subsection's `branching_covariance.arrow`, is drawn by the `isomer_branching` source. The ENDF parser also reads angular-distribution covariance (MF=34), and nothing consumes it yet. The decay data's own sigmas on half-lives, decay energies, decay branching ratios (any number of modes), decay photon lines and fission yields are carried as well, and propagated. Everything held at nominal is listed under What is not propagated.

Energy range

Every library reaches past 20 MeV for at least some evaluations, and only TENDL-2025 is uniform. 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, rest at 30 to 60

So ENDF/B-VIII.1 is the only one that is mostly a 20 MeV library, and even its Fe56 and W186 reach 150 MeV. A spectrum whose flux reaches above a nuclide's ceiling is **refused before the collapse**, naming the nuclide and the energy its evaluation stops at, because there is no cross section up there to fold against and any answer would be an invention. So check the ceiling of the nuclides that carry your reaction rather than the library's headline.

Groups above the ceiling that carry no flux cost nothing, which is what makes a padded spectrum safe: the fold stops at the last tabulated point and contributes zero above it, rather than carrying the last value forward to 1 GeV. Those are different answers whenever the top groups are populated, and the second one is wrong: a threshold reaction whose cross section is still rising at 20 MeV would otherwise be priced at its 20 MeV value across two more decades.

Some decay records are placeholders

A nuclide whose decay scheme nobody has evaluated still gets a decay file. The conversions from the Nuclear Wallet Cards and from NUBASE carry a half-life and the decay modes and, in place of measured average energies, book a third of each beta or electron-capture branch's Q to the light particles, a third to the photons and the last third to the neutrino. That is not a rough estimate: electron capture hands most of Q to the neutrino, so for an EC emitter the placeholder can be several times the recoverable energy. ENDF/B-VIII.1 books Sn111 at 1.63 MeV per decay against the 0.69 MeV JENDL-5 evaluates from its decay scheme.

library	placeholder decay records
`endf-b8.1`	1144
`jeff-4.0`	1529
`jendl-5.0`	377

They are exotic nuclides, so most runs never touch one, but decay heat at short cooling times is carried by exactly the short-lived products that tend to be unevaluated: on the FNS tin, cadmium, palladium and nickel foils these records carry 12 to 35% of the heat in the first minutes. The converter labels them rather than hiding them, and can substitute another library's evaluation for them; see Filling placeholder decay energies.

Some reactions do not list the isomer

A reaction that can leave its product in a metastable state has to say so in the evaluation, one entry per final state. One that lists the ground state alone is not merely less accurate: the isomer is absent from any network built from it, so no code can make it, and a measurement that sees its decay heat cannot be reproduced by any means. Nothing about such a channel looks wrong from outside. It is present, its cross section is reasonable, and only the state list is short.

Two channels where that decides an answer, both from the FNS decay-heat benchmark:

channel	`jeff-4.0`	`endf-b8.1`	`jendl-5.0`	`tendl-2017`	`tendl-2025`
`Os190 (n,n')` to Os190m, 1706 keV	yes	no file	yes	**no**	yes
`Ir191 (n,2n)` to Ir190m2, 377 keV	yes	yes	yes	**no**	yes

The heat from an osmium foil five minutes after a 14 MeV irradiation is Os190m, so on `tendl-2017` the calculated heat is a third of the measurement and no solver setting recovers it. On `tendl-2025` it is within about 20%. That is a larger effect than any of the differences above, and it is invisible in the cross sections.

So a library can be asked what states it lists, before a run rather than after:

```
channels = yani.radionuclide_production(neutron)
for channel in channels:
    if channel["reaction"] == "(n,2n)":
        print(channel["parent"], [s["excitation_energy_eV"] for s in
channel["states"]])
```

The excitation energy is what compares between libraries. The level index beside it is the evaluation's own numbering and is not comparable: Ir190's 377 keV isomer is level 3 in ENDF/B-VIII.1 and level 37 in JEFF-4.0 and TENDL-2025.

TENDL-2025's lithium breeds no tritium

TENDL-2019 and TENDL-2023 carried ENDF/B-VIII.0's lithium unchanged, two Los Alamos evaluations: Li6 by G.M. Hale (2017), which gives `(n,p)` and `(n,t)` as MT=103 and MT=105, and Li7 by P.G. Young (1988). TENDL-2025 replaces both with TALYS output, written as ENDF by TEFAL. Its Li6 file has cross sections for MT 1 to 5, 51 to 55, 91 and 102 only, and its Li7 file for MT 1 to 5, 16, 51 to 54, 91 and 102, so neither has a triton channel: the tritons MT=5 produces, and the MT=205 derived from them, begin at 30 MeV. Li6's resonance range is a six-resonance multi-level Breit-Wigner fit with neutron and gamma widths only, and its thermal total cross section is 0.485 b, where ENDF/B-VIII.1's is 938.9 b, almost all of it `(n,t)`.

So lithium on `tendl-2025` makes no tritium below 30 MeV, which is every energy a fusion or fission system reaches. ENDF/B-VIII.1 at 294 K has Li6 `(n,t)` at 938.1 b at 0.0253 eV and 0.0259 b at 14.06 MeV, and Li7 triton production (MT=205) at 0.301 b at 14.06 MeV; TENDL-2025 has zero for all three. In yani, a Li6 foil's H3 production averaged over a 0.02 to 0.03 eV group is 949 b on `endf-b8.1` and 0 on `tendl-2025`, and over a 14.0 to 14.1 MeV group 0.0259 b and 0. Transport shows it at the scale of a blanket: a 100 cm sphere of natural lithium around a 14.06 MeV source has a tritium breeding ratio of 1.2169 +/- 0.0021 on `endf-b8.1` and exactly 0 on `tendl-2025` in yamc (100,000 histories).

For a lithium-bearing breeder, blanket or shield, take both the cross sections and the `reactions` network from another library. The network is what holds the `(n,t)` channel: with `transmutation_reactions = "tendl-2025"`, Li6 makes no tritium even when its cross sections come from `endf-b8.1`.

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 a run loads are fetched: those in your material and the products its irradiation populates. Fe56 is 56 MB on `endf-b8.1` and 19 MB on `tendl-2025`; U235 is 179 MB and 387 MB, 346 MB of the latter its cross-section covariance, which only an uncertainty run reads. The chains are not small: 88 MB for a TENDL network against 67 MB for JENDL-5's complete one, most of it the `(n,X)` residual production in `reactions`.

TENDL chains are not standalone

`TransmutationChain` cannot open a TENDL or FENDL chain on its own, because it has no `decay` subsection and the loader wants a complete one. Set the subsections individually instead, which is the arrangement the tables above describe:

```
yani.transmutation_reactions = "tendl-2025"
yani.transmutation_branch_ratios = "tendl-2025"
yani.transmutation_decay_data = "endf-b8.1"
yani.transmutation_fission_yields = "endf-b8.1"
```

Usage

The examples on this page assume the nuclear data settings are made, as in Getting started: `neutron_cross_section_data` and the three `transmutation_*` subsections have no default, and a calculation that reads an unset one is refused (see Nuclear data settings).

Schedules

An irradiation history is one `PulseSchedule` built from `Pulse` (irradiation) and `Cooldown` (decay only) steps, in order. Every step carries a duration, either seconds as a plain number or a `(value, unit)` tuple:

unit	accepted spellings
seconds	`s`, `sec`, `second`, `seconds`
minutes	`min`, `minute`, `minutes`
hours	`h`, `hr`, `hour`, `hours`
days	`d`, `day`, `days`
years	`a`, `y`, `yr`, `year`, `years` (365.25 days)

```
schedule = yani.PulseSchedule(steps=[
    yani.Pulse(rate=1e14, duration=(2, "h"), source=spectrum),
    yani.Cooldown(duration=(30, "min")),
    yani.Pulse(rate=5e13, duration=(2, "h"), source=spectrum),   # different rate
    yani.Cooldown(duration=(1, "a")),
])
```

Each pulse gets its own `rate` and its own `source`, so a campaign whose spectrum or magnitude changes between phases is expressed directly. The results carry one material per step, so the number of results is the number of steps.

A decay curve wants points at cumulative times, while `Cooldown` takes the duration of its own step, so a sweep means differencing the times by hand. `cooldown_steps()` does that for you:

```
schedule = yani.PulseSchedule(steps=[
    yani.Pulse(rate=1e14, duration=(2, "h"), source=spectrum),
    *yani.cooldown_steps(start=(1, "h"), stop=(10, "a"), n=40),
])
```

The steps land at cumulative times from `start` to `stop`, log-spaced by default: decay is exponential, so a linear sweep spends its points on the flat tail and misses the early fall. `spacing="linear"` is the other one, and anything else raises.

> **What `rate` means depends on the entry point**
>
> For standalone `Material.transmute()`, `rate` is the **total flux magnitude** in n/cm²/s: the spectrum supplies the shape and `rate` supplies the size. For the coupled `Model.simulate_transmutation()` in yamc, the same field is a **source rate** in n/s, because there a transport solve turns emission into flux. The `Pulse` docstring gives the latter.

Spectra

The spectrum rides on the pulse as a `NeutronSource` whose energy distribution is a `Histogram` of group boundaries and per-group values:

```
energy_groups = [1e-5, 0.625, 1e5, 2e7]   # eV, ascending, n+1 boundaries
multigroup_flux = [1e12, 5e12, 1e14]      # n groups: thermal, epithermal, fast

spectrum = yani.NeutronSource(
    energy=yani.sources.Histogram(boundaries=energy_groups,
probabilities=multigroup_flux)
)
pulse = yani.Pulse(rate=sum(multigroup_flux), duration=(1, "h"), source=spectrum)
```

The `Histogram` normalizes the shape, so a flux spectrum can be passed in with its own magnitudes and the total handed to `rate`. The source's position and direction are ignored, since nothing is transported.

Named group structures

Any group structure works. The boundaries argument is just an ascending list of energies in eV, of whatever length your spectrum happens to have, and nothing about YANI prefers one structure over another. The example above uses three groups.

What the names below add is convenience for a handful of common structures, so that if your spectrum is already on one of them you do not type out its boundaries. They are not privileged, and using one is not required:

```
spectrum = yani.NeutronSource(
    energy=yani.sources.Histogram(boundaries="CCFE-709", probabilities=flux_709)   #
709 values
)
```

name	groups	range	typical use
`XMAS-172`	172	1.00001e-5 eV to 19.6403 MeV	the European (XMAS/JEF) lattice set
`VITAMIN-J-175`	175	1e-5 eV to 19.64 MeV	the long-standing neutron transport standard
`SCALE-252`	252	0 eV to 20 MeV	SCALE/AMPX
`TRIPOLI-315`	315	1e-5 eV to 19.64 MeV	TRIPOLI
`SHEM-361`	361	0 eV to 19.6403 MeV	resonance self-shielding in thermal lattices
`LLNL-616`	616	1e-5 eV to 20 MeV	LLNL
`CCFE-709`	709	1e-5 eV to 1 GeV	fine-group fusion activation spectra
`SCALE-999`	999	1e-5 eV to 20 MeV	fine-group SCALE/AMPX
`UKAEA-1102`	1102	1e-5 eV to 1 GeV	the finer companion to `CCFE-709`, same 1 GeV top
`ECCO-1968`	1968	1.00001e-5 eV to 19.64033 MeV	ECCO/ERANOS fine group
`VITAMIN-J-42`	42	1 keV to 50 MeV	photon, with resolution at the 511 keV and Co60 lines
`CCFE-24-PHOTON`	24	1 keV to 20 MeV	coarse photon

Two of them reach far past the evaluations. `CCFE-709` and `UKAEA-1102` top out at 1 GeV, while most ENDF/B-VIII.1 evaluations stop at 20 MeV, so a spectrum on one of those structures with flux in its top groups is refused, naming the nuclide and the energy its evaluation stops at. Leaving those groups empty is the normal case and costs nothing. Which library reaches how far is in Energy range.

Passing a name that is not one of these raises with the list of the ones that are, so a typo never silently becomes something else. Note that `SCALE-252` and `SHEM-361` start at exactly 0 eV; treat their bottom bin as open-ended.

If you need the edges themselves, to bin your own data, to plot a spectrum against them, or to fold a cross section over them, ask for them by name:

```
edges = yani.group_structure(name="CCFE-709")
len(edges)              # 710, one more than the group count
edges[0], edges[-1]     # 1e-05 eV, 1e+09 eV

yani.group_structure_names()   # every name in the table above
```

The list is a copy, and the same edges a `Histogram` built on that name bins on.

Cross sections are collapsed against each distinct spectrum once and then scaled by each step's `rate`, rather than recollapsed every step.

Materials

A composition dict plus a density. Element symbols expand over natural abundance, nuclide names are taken literally, and chemical formulas are parsed:

```
foil = yani.Material(composition={"Ag": 1.0}, density=10.49, volume=1.0)
enriched_li = yani.Material(
    composition={"Li": yani.enriched(fraction=1.0, target="Li6", percent=60.0)},
    density=0.534,
    volume=10.0,
)
breeder = yani.Material(composition={"Li4SiO4": 1.0}, density=2.4, volume=100.0)
```

For a standard shielding or structural material, the PNNL Compendium (PNNL-15870 Rev. 2) ships with the package, 410 materials keyed by the report's own names, so the composition and density come from a citable source rather than from your own retyping:

```
steel = yani.materials.pnnl.material(key="Steel, Stainless 316", volume=1000.0)
concrete = yani.materials.pnnl.material(key="Concrete, Ordinary (NIST)", volume=1e6)

list(yani.materials.pnnl)          # every name in the compendium
```

Pass `volume` at construction for either route: it is what turns the solver's atom densities into the atom counts that activity and decay heat need.

Two other constructors cover compositions you assemble rather than look up. `Material.mix_materials` blends existing materials by atom, mass or volume fraction and works out the mixed density; `Material.from_atom_densities` takes absolute atoms/barn-cm per nuclide and stores them verbatim, for an inventory you already hold as number densities:

```
eurofer = yani.Material(composition={"Fe": 0.9, "Cr": 0.09, "W": 0.01}, density=7.8)
water = yani.Material(composition={"H2O": 1.0}, density=1.0)
blanket = yani.Material.mix_materials(
    materials=[eurofer, water], fractions=[0.8, 0.2], fraction_type="volume"
)
blanket.volume = 5e4

measured = yani.Material.from_atom_densities(
    {"Fe56": 7.7e-2, "Co59": 5.0e-6}, volume=1.0
)
```

`density` on a `from_atom_densities` material is the sum of the values given.

`transmute()` loads the cross sections it needs into the material and leaves them there, so calling it again on the same material does no file reading at all. That is the right trade for a material you transmute more than once and the wrong one for a sweep over thousands of distinct compositions, each of which would otherwise hold its nuclides' data for as long as it lives: those of the composition and of the products the irradiation populates, a few dozen for a pure metal and up to a couple of hundred for a steel. Call `material.release_nuclear_data()` when you are done with one; the material stays usable, and the next call that needs the data loads it again.

Results

`transmute()` returns a `TransmutationResults` -- the same object the coupled `Model.simulate_transmutation()` returns in yamc -- keyed by the material's `id`. Every material has one from construction: the `id` passed to it, or else a fresh one that no earlier yani material in the process was given (yamc numbers its own materials separately). `step_materials()` unpacks the results into `list[Material]`, one per step, each holding the inventory at the end of that step and keeping the original's `id`. Extensive quantities need the material's `volume` in cm³:

```
results = material.transmute(schedule=schedule)
steps = results.step_materials(material_id=material.id)

for step, mat in enumerate(steps, start=1):
    print(step, mat.activity(), "Bq", mat.decay_heat(), "W")

final = steps[-1]
final.activity(by_nuclide=True)      # dict[str, float], Bq per nuclide
final.decay_heat(by_nuclide=True)    # dict[str, float], W per nuclide
final.nuclides                       # list[(name, fraction)]
final.get_atoms_per_barn_cm()        # dict[str, float], the solver's own units
```

Stable nuclides contribute nothing to activity or decay heat and are omitted from the by-nuclide dictionaries.

`per` is the way out of needing a size at all. `per="cm3"` gives Bq/cm³ or W/cm³ and needs neither `volume` nor `density`, and `per="g"` gives Bq/g or W/g and needs only `density`. The decay photon spectrum takes the same argument:

```
final.activity(per="cm3")                  # Bq/cm3, no volume needed
final.decay_heat(per="g")                  # W/g, needs only density
final.decay_photon_spectrum(per="cm3")     # photons/s/cm3
```

`component=` splits decay heat into the recoverable parts 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 is deposited where the decay happens:

```
final.decay_heat(component="gamma")                  # W, the photons alone
final.decay_heat(component="beta", by_nuclide=True)  # dict[str, float], W
```

A nuclide making decay heat whose data carries no split, which data converted before the split was recorded does not, raises rather than understating the component.

`contact_dose()` is the third of these, and the one that needs no `volume`: it is the dose someone receives with a hand on the material, from the material's own decay photons, and a bigger lump of the same material reads the same at contact.

```
final.contact_dose()                          # Gy/h, absorbed dose in air
final.contact_dose(by_nuclide=True)           # dict[str, float], Gy/h per nuclide
final.contact_dose(dose_quantity="effective") # Sv/h, ICRP-116 effective dose
```

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. Integrating over depth cancels the geometry, which is why no distance and no volume appear in the answer. A `build_up` factor, 2.0 by default, stands in for the photons that scatter on the way out and still arrive.

The two NIST tabulations it folds against are public, so a response function of your own can be built from the same data: `mu/rho` for any element from Z = 1 to 100, and `mu_en/rho` for air. Both read log-log between their tabulated points.

```
iron = yani.data.mass_attenuation_coefficient(element="Fe")   # or by atomic number,
26
iron.interpolate(energy=1.0e6)                                # cm2/g at 1 MeV
air = yani.data.mass_energy_absorption_coefficient(material="air")
```

Fluence-to-dose coefficients for folding with a transport tally come from `yani.data.dose_coefficients()`. Effective dose is the default, from ICRP-116 or ICRP-74 for an irradiation geometry; `dose_quantity="ambient"` gives the ambient dose equivalent H*(10) from ICRP-74, the quantity most area-monitoring regulations still reference.* H(10) is defined in an aligned and expanded field, so it takes no `geometry`:

```
effective = yani.data.dose_coefficients("neutron", geometry="AP")
h10 = yani.data.dose_coefficients("neutron", dose_quantity="ambient")
```

The H*(10) 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 meant to over-estimate effective dose, but for high-energy neutrons it reads under it.

The default quantity follows UKAEA-CCFE-RE(21)02, Appendix C.7.1. Bremsstrahlung from decay electrons is not modelled, so a strong beta emitter reads low at contact.

The decay photon line spectrum comes back as the `(x, p)` pair the source distributions take, in photons per second, so it feeds straight into a photon transport run in yamc:

```
energies, intensities = final.decay_photon_spectrum()
```

Part of some decay spectra is given in the data as a density over energy rather than as lines: the spontaneous-fission photons of an actinide, or the whole photon emission of a nuclide far from stability. `decay_photon_spectrum()` holds the lines only, and `decay_photon_continua()` returns the rest, one `PhotonContinuum` per nuclide and continuum. Its rates are per eV, so they are not line rates and cannot be added to the spectrum above; `emission_rate` is their integral. It takes the same `per` argument:

```
for continuum in final.decay_photon_continua(per="cm3"):
    continuum.nuclide          # e.g. "Cf252"
    continuum.energies         # eV, the continuum's own grid
    continuum.rates            # photons/s/eV/cm3 on that grid
    continuum.interpolation    # the ENDF law between points, e.g. "linear-linear"
    continuum.emission_rate    # photons/s/cm3, the integral
```

Inventory and gas-production diagnostics

Composition getters include the initial state at index zero; `times` is the matching cumulative-time array. Unknown material IDs or states return `None`:

```
print(results.material_ids, results.num_steps, results.times)
co60 = results.get_nuclide_evolution(material_id=material.id, nuclide="Co60")
state = results.get_material(material_id=material.id, step=1)
if state is not None:
  print(state.get_nuclide_names())

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 production is in appm, relative to the material's initial atom count. By default the initial gas is subtracted; `produced=False` returns the gas present, including what was there initially. A consumed gas can therefore have negative net production. A chain missing one of the five gas nuclides raises rather than reporting its unknown contribution as zero.

With `data_uncertainty=` the gas comes with a nuclear-data spread as well, one `Estimate` per gas nuclide and total at a single step:

```
gas = results.get_gas_production_uncertainty(material_id=material.id, step=1)
if gas is not None:                  # None unless data_uncertainty was given
    gas["He"].nominal, gas["He"].std_dev
```

Each replica's gas is taken from its own inventory against the one initial inventory, and the `"H"` and `"He"` totals are summed within a replica before the spread is taken, so they are not the quadrature sum of their parts.

Product screening omits the neutron reactions of sufficiently small products, not their production or decay. `data_gaps` reports missing-data assumptions separately, so a small or zero activity is not a substitute for checking it.

Displacement damage

`displacement_damage=True` also computes the damage the irradiation does to the material's lattice: the damage energy deposited per atom, and NRT displacements per atom (dpa), per element and for the material. Each getter returns one cumulative value per state, aligned with `times`, starting from zero:

```
iron = yani.Material(composition={"Fe": 1.0}, density=7.874, volume=1.0)
schedule = yani.PulseSchedule(steps=[
    yani.Pulse(rate=sum(multigroup_flux), duration=(1, "a"), source=spectrum),
    yani.Cooldown(duration=(1, "d")),
])
results = iron.transmute(schedule=schedule, displacement_damage=True)

results.get_dpa(material_id=iron.id)                  # [0.0, 6.49, 6.49]
results.get_dpa(material_id=iron.id, element="Fe")    # the iron's own dpa
results.get_damage_energy(material_id=iron.id)        # eV per atom: [0.0, 649.4,
649.4]
results.get_displacement_damage_info(material_id=iron.id)
```

Each nuclide's MT=444 damage-energy cross section is folded against the pulse spectrum by the same collapse as the reaction rates, self-shielding included, at each step's composition; cooldowns add nothing. An element's dpa is `0.8 * E_damage / (2 * E_d)` (the NRT model, ASTM E521), with the displacement threshold energy `E_d` from ASTM E521, or from the OECD-NEA 2015 report "Primary Radiation Damage in Materials" for an element E521 does not cover. `displacement_energies={"Fe": 40.0}` replaces them, and an element in neither must be given there. The material total weights the elements by atom fraction. `get_displacement_damage_info` names the `E_d` used for each element and its source, the transmutation products whose data has no MT=444 (`without_damage_energy`) and the product elements with no `E_d`, hydrogen and helium above all (`without_displacement_energy`), each with the largest atom fraction it reached. Without the flag, MT=444 is not fetched and nothing about the solve changes.

Pass a `DataUncertainty` as well and the dpa and damage energy come with a nuclear-data spread, one `Estimate` per step:

```
results = iron.transmute(
    schedule=schedule,
    displacement_damage=True,
    data_uncertainty=yani.DataUncertainty(seed=42),
)
dpa = results.get_dpa_uncertainty(material_id=iron.id, step=1)
if dpa is not None:                # None unless both were asked for
    dpa.nominal, dpa.std_dev       # 6.49, 0.158: 2.4%
results.get_damage_energy_uncertainty(material_id=iron.id, step=1, element="Fe")
```

Each replica's dpa is accumulated over its own inventory. Under `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. What no reaction's deposit reproduces, fission's share above all, is held, and `get_data_uncertainty_info(...)["damage_energy"]` gives, per nuclide of the starting composition, the share of its MT=444 rate the deposits reproduce (`explained`) and the relative sigma of its drawn rate (`sigma`); for this iron, Fe56 is explained in full with a sigma of 2.8%. Under `flux_spectrum`, the damage energy moves with the replica's flux draw. The damage energy of transmutation products, short-range (LB=8) covariance and the displacement threshold energies are held at nominal. Asking for damage widens what is drawn for each nuclide of the starting composition, so for one seed the draws differ from those of a run without `displacement_damage`.

Clearance and disposal limits

A material can also be checked against the regulatory clearance, exemption and disposal limits directly, from yani 0.19.0. `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 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 `volume`. Stable isotopes have to stay in the material, since they are most of its mass and so most of the Bq/g denominator.

```
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.

The rest of a `ClearanceResult` says how the index was reached:

- `limits_used` is the limit applied to each nuclide after any metal, per-gram, dynamic or secular-equilibrium adjustment, and `assessed_activity(nuclide)` the activity charged against it: the nuclide's total less any part a parent already accounted for.
- `defaulted` lists the nuclides that took the set's catch-all limit, `unlimited` those the source explicitly places no limit on, and `uncovered_activity` the total activity no limit covers, in the set's units.
- `out_of_scope` is true when the regulation excludes the material outright; for the UK sets that means every radionuclide present is shorter lived than 100 s.
- `limit_set` and `material_name` are carried through for reporting, and `to_json("clearance.json")` saves the whole result as JSON; with no path, `to_json()` returns the JSON as bytes, so `json.loads(result.to_json())` gives it as plain Python.

Several materials at once

`yani.transmute()` is the plural of `Material.transmute()`, for the cells of a mesh, a component broken into regions, or a sweep over compositions. Each material gets exactly the answer `material.transmute(schedule)` would give it, and the result is one `TransmutationResults` keyed by each material's `id`, so the ids must be distinct. Every material is built with its own, so two share one only when one was given it explicitly or came out of a transmutation result: the materials a result returns keep their original's `id`, so they share it with the original and with each other.

```
inner = yani.Material(composition={"Fe": 1.0}, density=7.87, volume=10.0)
outer = yani.Material(composition={"Fe": 1.0}, density=7.87, volume=10.0)

results = yani.transmute(materials=[inner, outer], schedules=[schedule, schedule])
results.get_source_rates(material_id=outer.id)   # list[float], each step's flux,
n/cm2/s
results.collapse_reuse                           # {"performed": ..., "requested":
...}
```

Pass one schedule for every material, or one per material in the same order. Each schedule carries that material's own spectra and flux magnitudes, but all of them must share one timeline: the same durations, irradiating on the same steps. What a Python loop would repeat is done once: the chain is read once, each nuclide's cross sections are decoded once for every material that needs them, and a collapse runs once per distinct spectrum, composition, temperature and shielding, so cells of one steel that saw the same spectrum share it whatever their flux magnitudes. `collapse_reuse` says how many collapses that saved, and the solves then run in parallel. `data_uncertainty`, `self_shielding_chord` and `self_shielding_shape` apply to every material as `Material.transmute()` applies them to one, and the same seed perturbs a given evaluation the same way in every material.

`get_source_rates()` gives the rate each step drove a material at, zero for a cooldown. On a spectrum solve that is the material's own flux magnitude in n/cm²/s; on a yamc transport run it is the source strength in n/s.

Saving results

`to_arrow()` writes a `TransmutationResults` to an Arrow IPC file, and `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.

A calculation too big to hold at once (thousands of mesh cells, a long schedule, an uncertainty run) can be split into parts, each saved and dropped before the next:

```
for part, cells in enumerate([cells[:500], cells[500:]]):
    results = yani.transmute(materials=cells, schedules=schedule)
    results.to_arrow(f"part_{part}.arrow")

results = yani.TransmutationResults.from_arrow("part_0.arrow")
data = results.to_arrow()                       # bytes
yani.TransmutationResults.from_arrow(data)
```

Every file has the same columns, one row per stored number: `material_id`, `material_name`, `step`, `quantity` (`"density"`, `"reaction_rate"` or `"source_rate"`), `nuclide`, `reaction`, `product` and `value`. So the parts read as one table without yani:

```
import polars as pl

pl.scan_ipc("part_*.arrow").filter(pl.col("quantity") == "density").collect()
```

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 `yani.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. Everything else answers from the file.

Continuous processing

`Processing` describes continuous feeds, extraction and transfers between materials during a step. Pass it to the batch `transmute()` function, not to `Material.transmute()`, so every source and destination is part of one solve:

```
processing = yani.Processing()
processing.add_transfer(
  material=inner,
  component="H3",
  rate=1e-4,
  rate_units="1/s",
  destination=outer,
)
processing.add_feed(
  material=inner,
  composition={"Li6": 0.6, "Li7": 0.4},
  rate=1e-3,
  rate_units="g/s",
  steps=[0],
)
processed = yani.transmute(
  materials=[inner, outer], schedules=schedule, processing=processing,
)
print(processing.material_ids)
```

The transfer removes a fraction of the named component per second; omitting `destination` removes it from the system. An element name selects every isotope, and omitting `component` transfers the whole composition. Feed compositions are atom fractions, normalized internally. Absolute feed units such as `g/s`, `kg/d`, `mol/h` and `atoms/s` need the receiving material's volume.

`steps` selects zero-based schedule steps. Omitted, a term applies to every step, including cooldowns. Rates can also be tables of `(times, rates)`, and `proportional_to="Li6"` makes a feed follow a nuclide's current inventory with rate units such as `1/s`; the full forms are in the API reference.

Self-shielding

A lump of a resonance absorber shields itself: the flux inside it is depressed exactly where the total cross section is large, so the reaction rate is lower than the same material spread thin. Nothing is corrected unless you ask, because a `Material` carries no geometry to infer a size from. Give the lump a shape, or state its mean chord `4V/S` in cm yourself:

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

`SphereLump()` and `CubeLump()` are fixed by the material's `volume` and take no arguments, while `FoilLump(thickness=)`, `CylinderLump(radius=)` and `WireLump(radius=)` carry the one dimension a volume cannot imply. Give a shape or `self_shielding_chord=`, not both. 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: an upper bound rather than a safe default, which is why there is no default at all.

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: on W186(n,gamma) in FNG-tung, measured at 1.29 b, the slowing-down solve gives C/E 0.80, no correction gives 0.86 and narrow resonance gives 0.46.

What was done comes back on the results, and a run that shielded nothing says so rather than staying silent:

```
shielding = results.get_self_shielding_info(material_id=foil.id)
if shielding is not None:              # None for a coupled solve or unknown id
    shielding["method"], shielding["chord_cm"]   # how, and the chord used
    shielding["shielded"]              # nuclides the correction reached
    shielding["not_shielded"]          # and why each of the others was left
    shielding["strongest_factor"]      # smallest factor any group average took
```

A `chord_cm` of `None` means the run was dilute and nothing was corrected. A `strongest_factor` of `1.0` means the correction ran and changed nothing in practice. A dilute run fills `would_shield` instead: nuclides whose own resonances are structured enough to have suppressed a reaction, each mapped to how strongly.

`would_shield` is an indicator, not a correction and not a bound. There is no geometry in it. The weight is `1 / (1 + N * sigma_x)` on that one reaction and that nuclide's own density, which fixes the background at 1/cm, while the correction proper uses `1 / chord_cm` against the material's total with in-scattering. 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 across a resonance dips the flux further than one reaction can express. On the FNS tungsten foil it reads 0.634 for W186, against a slowing-down correction that gives 0.750 at a 1 mm chord and saturates at 0.730.

Read it as "this answer may be high, and this is a resonance absorber", which is the warning a dilute run should carry rather than silence. For the size of the effect, ask for a shielded run.

Production routes

`TransmutationChain` carries the topology behind the solve: which nuclide becomes which, and how. `reactions` holds the neutron-induced edges and `decays` the radioactive ones, both keyed by parent and both giving `(kind, target, branching)`:

```
chain = yani.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.

The two share a tuple shape, so a route walk can put them in one edge list. With reactions alone the 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, via, _ in edges["W186"]:              # two-step routes from W186 to Ta183
    for kind2, end, _ in edges.get(via, []):
        if end == "Ta183":
            print(f"W186{kind1}{via}({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 own channel, not a rate: it says how a channel splits, not how often the channel fires. Two routes into the same product are not weighted against each other by these numbers alone.

Note that the chain this object holds is the base three-part merge. The isomeric branching overlay is applied inside the solve, so a `(n,2n)` edge here names the ground state where the overlay would split it between ground state and isomer.

Weighted, from a solve

The walk above can enumerate routes but not rank them, and the branching it reads is the file's, not the one the solve used. `get_production_routes()` answers the same question from a solve that has happened. Asked for the Ta183 of the example above:

```
routes = results.get_production_routes(material_id=mid, product="Ta183", step=0)
for route in routes or []:          # None if the material or the step is unknown
    print(f"{route['route']:<32} {route['share']:.1%}")

# W183(n,p)Ta183                   72.4%
# W186(n,a)Hf183(BETA-)Ta183       16.1%   <- the one that needs the decay edge
# W184(n,np)Ta183                   7.1%
# W184(n,d)Ta183                    4.4%
```

Each entry also carries `steps`, the same route as `(parent, kind, target)` triples with the chain's own spellings (`('Hf183', 'beta-', 'Ta183')`), where the `route` string uppercases decay kinds the way the published pathway tables print them.

The walk starts from the nuclides the material began with, which a bare chain cannot do. Asked what makes W187, a chain lists `Os190(n,a)` and `Ir192(n,npa)` as readily as `W186(n,gamma)`, and nothing in a tungsten foil is osmium.

`share` is the fraction of that product's production arriving down the route: the atoms it starts from, times what its reaction drove per atom of its parent over the step, times the branching of every decay it passes through, so a route through a 1% branch delivers 1% of what the reaction made. Reaction steps carry the step duration too, so `reaction_depth=2` is in the same units as the default 1 and comes out smaller by roughly a factor of the fluence. `production` is the same number unnormalised, in atoms per barn-cm, because 100% of almost nothing and 100% of the inventory read alike otherwise.

The flux-weighted isomeric branching comes from the same place:

```
branching = results.get_isomeric_branching(material_id=mid, step=0) or []
branching[0]
# {'parent': 'W186', 'reaction': '(n,2n)', 'production': 9.35e-14,
#  'split': [('W185_m1', 0.535), ('W185', 0.465)]}
```

Channels come back ordered by what they made, the channel's rate times its parent's atom density at the start of the step, rather than by rate alone. A rate is per atom of its parent, so ordering on that promotes whatever sits on a trace isotope: on the FNS tungsten foil `W180 (n,2n)` has the highest per-atom rate of any channel in the foil, and W180 is 0.12% of it, so weighted by what it actually made the channel falls to fifth, two orders of magnitude below the `W186 (n,2n)` carrying most of that foil's decay heat. A parent the step did not start with has a production of `0.0` and sorts last rather than being dropped.

Which state a reaction leaves its product in is energy dependent, so the one number describing a spectrum is the branching collapsed against it, and that exists only inside a solve. In the chain file that same channel reads `W186 (n,2n) -> W185 1.000000` and `-> W185_m1 0.000000`, a placeholder the overlay replaces at solve time. Only channels landing in more than one final state are returned, since a single-product channel has no branching to report and listing it at 1.0 buries the ones that do.

This is what separates a disagreement caused by a cross section from one caused by a branching ratio. On a foil whose decay heat comes from an isomer, getting it wrong is a factor of several.

`get_branching_report()` says how each channel's split was read from the evaluation over the step's spectrum: as shares of the reaction's total, or as absolute productions with the ground state taking the rest, and how much of the parent's removal rate rests on anything the evaluation does not give:

```
report = results.get_branching_report(material_id=mid, step=0)
if report is not None:
    channel = report["channels"][0]
    channel["parent"], channel["reaction"], channel["representation"]
    channel["states"]            # each final state and its share of the reaction
    channel["clipped_share"]     # of the parent's removal rate
    report["dropped"]            # channels that could not be folded, and why
    report["unmodelled_mt5"]     # (nuclide, share, reason) where MT=5's residuals are
not modelled
```

A run refuses when the clipped or held production of its channels, each parent weighted by its density in the material, is more than 0.1% of the material's neutron removal rate; each channel's `clipped_share` and `extrapolated_share` are of its own parent's removal. Weighting by the material keeps one bad list

on a product present at a trace from refusing every irradiation whose network reaches it, while a material made of that parent is held to the same 0.1%.

MT=5, `(n,anything)`, is the `(n,X)` reaction. The reactions subsection carries its products from the evaluation's MF=6 MT=5: each residual with its isomeric state, folded as a share of the MT=5 total at each energy (`file` 6, `representation` `"share"`), and each light particle H1, H2, H3, He3 and He4 as its multiplicity times that total (`representation` `"multiplicity"`, whose `share` is the multiplicity folded over the spectrum and can exceed one). Their gas is therefore in the inventory. In `TransmutationChain.reactions` an `(n,X)` edge's branching is NaN, since its split depends on the spectrum.

`unmodelled_mt5` lists what is left: a parent whose evaluation gives MT=5's light particles but not its residuals (ENDF/B-VIII.1's Fe58 and zirconium isotopes, among others), each with its share of the parent's removal rate and the reason. A run refuses when the material's own nuclides lose more than 0.1% of the material's removal rate that way. A reactions subsection converted before MT=5 was carried has no `(n,X)` at all, and its MT=5 is reported, as it was, rather than refused.

Where in energy a rate came from

A one-group rate cannot say which part of the spectrum made it. `get_reaction_rate_spectrum()` resolves one channel's rate onto the groups of the spectrum that drove it:

```
capture = results.get_reaction_rate_spectrum(
    material_id=mid, nuclide="W186", kind="(n,gamma)", step=0
) or {}

edges = capture["boundaries"]     # group boundaries in eV, one longer than rates
per_group = capture["rates"]      # each group's contribution, 1/s

sum(per_group)                    # the rate get_reaction_rates() reports
```

The two agree because they come from the same walk of the same cross sections, self-shielding included, so this is a decomposition of the rate the solve used rather than a second estimate of it.

The attribution is what the single number hides. A capture cross section spans decades, and an effective one-group value of tens of millibarns against a spectrum that is almost all fast is either a fast-capture rate or a resonance-region rate. Which one it is decides whether a disagreement belongs to the resonance processing or to the fast cross section, and those are different data and different fixes:

```
below_100_keV = sum(r for lo, r in zip(edges, per_group) if lo < 1.0e5)
print(f"{below_100_keV / sum(per_group):.1%} of the capture rate is resonance region")
```

It is also what to read beside `rate_fraction_covered`: set against the energies where a channel's covariance states a nonzero variance, it shows, to the resolution of the groups, where in energy the covered and uncovered parts of the rate the run actually used sit, which the share gives only as one number. On a shielded run it also shows which groups the flux depression moved, which `strongest_factor` gives only as a worst case.

Nothing is stored for it. One reaction over a 709-group structure is cheap to walk when asked, and keeping the breakdown for every channel would be tens of megabytes, so a run that never asks pays nothing. `None` comes back for a step that drove no flux, for a nuclide or channel the data does not carry, for `(n,n')`, whose rate comes from the branching overlay rather than from a group average, and for a coupled yamc run, which scores its rates at the collision energy and keeps no group structure to resolve them onto.

Inspecting nuclear data

The bundled names, natural abundances and nuclide-name parser need no nuclear data download:

```
print(yani.data.element_names()["Fe"])
print(yani.data.reaction_names()[102])
print(yani.data.atomic_number("Fe"), yani.data.atomic_symbol(26))
print(yani.data.split_nuclide("Ta180_m1"))
print(yani.data.element_nuclides()["Li"])
print(yani.data.natural_abundance()["Li6"])
print(yani.data.natural_abundance_records()["Li6"])
print(yani.cache_dir())
```

Natural-abundance records retain the reference's stated uncertainty or interval; an absent value is `None`, not zero. These reference uncertainties are not automatically propagated by a transmutation calculation.

Converted data can be inspected without running a solve. Pass the individual nuclide or element directory written by the converter, not the library root:

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

For a long-running inspection process, `yani.data.clear_nuclide_cache()` clears the in-memory nuclide cache, not the downloaded files. Release data held by a material with `material.release_nuclear_data()` when the material is no longer needed for the current sweep.

Nuclear data settings

Five module-level settings, read and written like attributes:

setting	supplies	decides	left unset
`neutron_cross_section_data`	continuous-energy cross sections per nuclide	how fast each channel runs	an irradiation is refused, naming the setting
`transmutation_reactions`	which reaction on which nuclide gives which product, and its Q	which products exist to be made	none: a calculation that reads it is refused, naming the setting
`transmutation_decay_data`	half-lives, decay modes, mean decay energies, photon lines	how the inventory decays and what it emits	none: a calculation that reads it is refused, naming the setting
`transmutation_fission_yields`	fission product yields	what a fission makes	none: a calculation that reads it is refused, naming the setting
`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

Each takes a library keyword or a path, and each can point somewhere different, so a network can mix libraries by subsection: a TENDL reactions network borrowing ENDF/B-VIII.1 decay data is the usual arrangement. The first two are easily confused, and 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. They are global process state, not per-call arguments. See Nuclear data libraries for which library publishes which subsection, and how much is in each.

There is also a `photon_cross_section_data` setting, the same one yamc has, which selects photon interaction data for photon transport. No yani calculation reads it: the photon attenuation behind `contact_dose()` is the built-in NIST XCOM tabulation, so it can stay unset. It is independent of `neutron_cross_section_data`, and neither falls back to the other.

Note which way the last column goes. Only `transmutation_branch_ratios` has a default. The other four name the data a result is made from, so none is chosen for you and nothing is downloaded from a library you did not pick: leave one alone and a calculation that reads it stops with a `ValueError` naming each unset setting, the hosted keywords and a line to copy. What each calculation reads:

calculation	reads
a schedule with an irradiating step (`Material.transmute()`, `yani.transmute()`)	`neutron_cross_section_data` and all three `transmutation_*` subsections above
a schedule that only cools, and `activity()`, `decay_heat()`, `contact_dose()`, `clearance_index()` and the other decay quantities	`transmutation_decay_data`

A decay-only schedule folds no cross sections and runs without `neutron_cross_section_data`.

The two that both say reactions

`neutron_cross_section_data` and `transmutation_reactions` are halves of one calculation rather than alternatives, and the rate collapse is where they meet. 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**. Neither substitutes for the other: 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.

Two consequences follow from that loop, one quiet and one not:

- A channel the network names whose MT the cross sections do not carry is skipped. `(n,n')` is the standing example: it has no transport MT, so its rate comes from the branching overlay's partials rather than from a group average, which is why it is absent from a `get_reaction_rate_spectrum()` answer.
- A missing `neutron_cross_section_data` is refused, per the column above, and so is a network pointed at the wrong material: before the solve, naming what the material holds against what the network drives.

`transmutation_reactions` and `transmutation_fission_yields` additionally accept `False`, which turns that subsection off. A decay-only calculation carries no reaction rates, so it needs no reaction topology; a material nothing in which fissions needs no yields, and neither does a library that publishes none of its own. Off is not the same as unset: `None` leaves the setting unset, so a calculation that reads it is refused, whereas `False` says the subsection is absent on purpose. Each reads back as it was assigned: the source, `False` or `None`. A rate that then needs it is refused when the burnup matrix is built, naming the nuclide and the reaction, rather than being solved as though the reaction produced nothing.

A reactions network built for one material and pointed at another is refused on the same terms. Networks are often scoped to the nuclides they were built from, because walking a whole library to activate one foil is wasted work. The cost is that two scoped networks look alike from the outside: two

conversions writing to the same output path leave the second network under a filename that names the first. Solving against the wrong one does not fail. Every step runs, no rate is negative, and the composition comes back as it went in, so the mistake surfaces as a decay heat of exactly zero much later with nothing to point at. An irradiated schedule whose material has no drivable nuclide is therefore refused up front, naming both what the material holds and what the network has reactions for. A decay-only schedule is exempt, since driving nothing is correct for it.

Where the data comes from

A keyword is resolved on first use, once per process, to the library's current published release: a dated folder such as `2026-10-10.1` with a manifest listing every file in it with its size and sha256. The sections needed are downloaded from that release, each checked against the manifest (a byte range against its own hash) before it is kept, then reused. A file that does not match is refused with an error naming it, never cached. Only the sections activation reads are fetched, never a whole library, so this is a small fraction of a transport data set. A load of many nuclides fetches them in concurrent batches and prints a `Downloaded k of N nuclides` counter as it goes.

The cache is a `yamc` folder in the platform's per-user cache directory, shared with yamc, and `yani.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 keeps a folder of its own, laid out as the published tree:

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

There is no setting that moves the cache. To read data from somewhere else, point the setting at a local directory instead (below), which bypasses the cache. When a newer release is published, the next run downloads what it needs from the new release, says so once, and leaves the older folder in place; every file of one library a run reads comes from a single release. With no connection, yani 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.

The release each keyword resolved to is recorded on the results, as `TransmutationResults.data_releases`: keyword to `release`, `manifest_sha256`, `format_version`, `offline`, `converters`, `locally_processed` and `library`, so a result says exactly which data

produced it. For a published release `locally_processed` is `False` and `library` is empty. `converters` is every build of the converters that wrote the release's files, as its manifest lists them, such as `"yani-python 0.29.0 (core 1db838a3...)"`. Each entry names the wheel that did the converting, so data a yamc converter wrote reads `"yamc-python ..."`, and the version is `unknown` when that wheel's distribution metadata was not installed. It is empty for a release published before manifests listed them.

Every setting also accepts a **path** to a local converted directory instead of a keyword:

```
yani.transmutation_reactions = "/data/my-network.arrow"
```

`neutron_cross_section_data` additionally accepts a **dict** keyed by nuclide, to mix sources nuclide by nuclide. The four `transmutation_*` settings take a single value each, since a network is assembled per subsection rather than per nuclide:

```
yani.neutron_cross_section_data = {
    "Fe56": "tendl-2025",
    "Li6": "endf-b8.1",
    "Be9": "/data/Be9.arrow",
}
```

A whole library in one folder

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

```
yani.neutron_cross_section_data = "/data/endf-b8.1"
yani.transmutation_decay_data = "/data/endf-b8.1"
yani.transmutation_reactions = "/data/endf-b8.1"
yani.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 subsection the manifest does not list is refused, naming the subsections the folder does carry, and so is a file the manifest lists that the folder does not hold. `TransmutationResults.data_releases` records the folder under its absolute path, with its `release` and `manifest_sha256`, and `offline` set to `False`.

A load checks sizes only. `yani.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 = yani.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.

Wherever a reactions or fission-yields source resolves, the folder must hold `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, which would make every rate through it zero. Set the subsection to `False` to leave it out. 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.

To build the data yourself from ENDF tapes rather than download it, the converters ship on this wheel: see Making your own data.

Making your own data

The wheel carries the library builder and the converters, so producing the Arrow data is a function call rather than a separate toolchain.

A whole library

`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:

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

It downloads each source archive and checks it against the size and sha256 it is pinned to, unpacks it, converts every evaluation, builds the transmutation chain, reads every folder back, and writes each

particle's `index.json` and then the manifest. The folder then serves every setting, as a downloaded release does (see A whole library in one folder):

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

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, so a library publishes the parts it can build from what it evaluates:

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

`thermal` is the thermal scattering tables (`thermal/c_H_in_H2O.arrow/` and so on). Like `photon` and `photonuclear`, it is there for transport in yamc: no yani calculation reads it, so a build for activation alone can leave it out with `parts`.

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, by level and excitation energy, and a run names the isomer against the decay data it is given: TENDL branching run with `transmutation_decay_data = "endf-b8.1"` is matched to ENDF/B-VIII.1's isomers, and the branching report says so.

The keyword arguments narrow a build, for testing or for the part you need:

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

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

- `parts`: of `"neutron"`, `"photon"`, `"photonuclear"`, `"thermal"`, `"decay"`, `"reactions"`, `"fission_yields"` and `"branching"`; the default is every part the library's release carries, and a part it does not carry is refused.
- `nuclides`: nuclides and element symbols, and thermal tables by name (`"c_H_in_H2O"`) or by the element they bind (`"H"` is every table binding hydrogen). A name the library does not have is refused. 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 installs beside this Python, then the one on PATH. Only the neutron and thermal parts need NJOY; 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, so an interrupted build continues where it stopped. The release stamp is the date the build started, kept on every resume. A folder holds one library built by one build of the package: the same version compiled from the same core commit, with the same dirty flag. A resume by any other build is refused rather than mixed.

The build is recorded in the release. The manifest's `provenance` carries `builder` (`"yani-python 0.28.0"`), `builder_commit` (the core commit the wheel was compiled from, `yani.__commit__`) and `builder_dirty` (`yani.__commit_dirty__`), both `null` for a wheel built outside a git checkout, beside `njoy`. Every `version.json` and `provenance.json` carries the same as one `converter` stamp, such as `"yani-python 0.28.0 (core 1db838a3...)"`, with `, dirty` after the hash for a modified tree, and the manifest lists each distinct stamp as `converters`. The single-evaluation converters below write the same stamp.

A wheel compiled from a tree with uncommitted changes (`yani.__commit_dirty__` is `True`) writes data that no commit describes, so it builds only a partial library: one given `parts`, `nuclides` or `temperatures`. A whole library from such a wheel is refused before anything is written. Commit the changes and rebuild the wheel to build a whole library. Wheels installed from PyPI are built from a clean commit and are never refused for this.

No manifest is written unless every evaluation converted (or is one NJOY is known not to process, declared in the builder with the reason), every folder reads back, and the chain loads. An evaluation that fails is reported with its reason, which is also kept in `<path>/.build/failed/`, and running again retries only those. The returned dict says what the build did:

- `keyword`, `release`, `path`, `parts`;
- `converted` and `reused`: `"<particle>/<Name>"` keys;
- `not_converted`: the declared failures, with NJOY's reason for each;
- `stale_declarations`: declared failures that converted after all;
- `njoy`: the NJOY version used, or `None`;

- `chain_nuclides`, `manifest_sha256`, `file_count` and `total_bytes`.

`verify_library` re-hashes the finished folder against its manifest, as for any library folder.

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. The builder checks the free disk a full build needs before it starts.

Single evaluations

The converters behind the builder take one evaluation, or one set of tapes, at a time:

```
# Cross sections for the rate collapse, from ENDF (needs NJOY) or from ACE.
yani.convert_neutron_xs(
    input_path="n-026_Fe_056.endf",
    output_dir="out/", njoy_exec="njoy")

# The transmutation network, from decay, fission-yield and neutron evaluations.
yani.convert_transmutation(
    decay_files=decay, fpy_files=fpy, neutron_files=neutron,
    output_path="out/transmutation_tendl-2025.arrow",
)

# Isomeric branching, as an overlay on the network.
yani.convert_branching(
    neutron_files=neutron, decay_files=decay,
    output_path="out/branching.arrow",
)
```

`convert_transmutation`, `convert_branching` and `radionuclide_production` read every material in each file, so a tape can be passed as it is published: JEFF-4.0's decay and fission yield tapes, or IRDFF-II's neutron tape of 70 materials, need no splitting first. `convert_neutron_xs` runs NJOY in a hidden folder inside `output_dir`, removed when the conversion ends, so that disk, not the system's temporary folder, needs room for NJOY's tapes.

A run that reads a transmutation subsection `convert_transmutation` wrote, given by its directory rather than a keyword, records it in `TransmutationResults.data_releases` under the directory's absolute path, with `locally_processed` set to `True`, `library` and `release` the `library` and `data_version` the converter recorded, and `manifest_sha256` the sha256 of its files. Data processed locally, such as the UKDD and UKFY libraries FISPACT-II pairs with TENDL, is then named in the result as what it is, not passed off as a published release.

Both network converters record which nuclides carry reactions of their own, as `parents` on each subsection of the manifest they write. A nuclide that appears only as somebody else's product is not a parent. That distinction is what lets an irradiated solve refuse a network scoped to a different material instead of returning the starting composition unchanged.

What the converters report about the library

Neither converter corrects its input. What they do is say what the input claims that cannot be right, so that a number carrying your decay heat can be looked up before it is believed.

The `decay` subsection's `provenance.json` carries two records:

- `decay_energy_placeholders` names every nuclide whose average decay energies are the Q/3 stand-in rather than an evaluated scheme, and the rule used to detect them. Listed by name rather than counted, because the question a reader asks is whether a nuclide carrying heat in their inventory is one of them.
- `decay_inconsistencies` lists, per kind, the records that cannot all be true: flagged unstable with a half-life of zero, branching ratios that do not sum to one, and isomeric transitions whose light and electromagnetic averages do not add up to the transition's Q. That last one matters most for decay heat, since an isomeric transition emits no neutrino and so must pay out its whole Q. ENDF/B-VIII.1 books Hf177m1 at 1.52 MeV against a 1.32 MeV transition.

`convert_branching` returns a dict rather than writing one, since a build script is what decides whether to publish:

- `level_routes` counts how each excited production level was matched to an isomeric state: by energy, by energy within a tenth, by level index, as the only candidate, or not at all. Given no decay files, nothing is matched: each level is written as the evaluation gives it (route `as_evaluated`) and a run matches it against the decay data it is given. A level matched by index is right only while two level schemes happen to agree, so a rebuild against another decay library moving levels out of `energy` is the regression the plain counts hide.
- `flagged_levels` is one line per level worth reading before publishing.
- `partial_sum_mismatches` is one line per reaction whose MF=10 partial cross sections do not reconstruct its MF=3 total, or whose MF=9 yields do not sum to one, by more than 2% of the nonelastic cross section. The rates are shared out in the partials' proportions either way, so this does not move a yani answer; it moves a code that folds the partials as they stand, and it points at the evaluation. TENDL-2017's Ir191 (n,2n) partials sum to 95% of MF=3 at 14 MeV because the file lists two of Ir190's three states.

Filling placeholder decay energies

Where the decay library has no evaluated scheme for a nuclide and another library does, the second one's average energies can be substituted:

```
yani.convert_transmutation(
    decay_files=decay, fpy_files=fpy, neutron_files=neutron,
    output_path="out/transmutation_endf-b8.1.arrow",
    library="endf-b8.1",
    decay_fill_files=jendl_decay, decay_fill_library="jendl-5.0",
)
```

Only a placeholder is replaced, only from a record that is not itself a placeholder, and only when the two half-lives agree within 25%, which is what keeps a differently-assigned isomer out. Only the mean energy and its uncertainty move: half-lives, decay modes and spectra stay as the decay library has them, so the network's topology is untouched. Every substitution is written to `provenance.json` as `decay_energy_fill`, with the value before and after, and each nuclide's `decay_energy_source` reads `evaluation`, `placeholder` or `filled:<library>`.

Without `decay_fill_files` nothing is substituted and the numbers are byte-identical to what the decay library published, so the placeholder list is worth reading even when you do not act on it.

`convert_neutron_transport` and `convert_photon` are present too, since all the wheels share one bindings crate, but they write sections only transport reads. `convert_photon` carries the Compton profile, density effect and bremsstrahlung tabulations it adds to every element (they are in no evaluation), so it takes the photo-atomic file and an output folder, and the same call gives the same data from yamc:

```
yani.convert_photon("photoat-026_Fe_000.endf", "out/photon",
                    relaxation_path="atom-026_Fe_000.endf")
```

`convert_photonuclear` keeps a photonuclear evaluation (TENDL's gamma sublibrary) as evaluated, its MF=3 cross sections on their own grids and every original section, without NJOY.

NJOY is needed for the ENDF route and nothing avoids it: an ENDF evaluation describes the resonance region with resonance parameters rather than pointwise cross sections, so something has to reconstruct and Doppler broaden it. `source_format="ace"` reads an already-processed table and needs no NJOY, at the cost of one temperature and no heating sections.

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=yani.DataUncertainty(seed=42),
)

mid = material.id
mean = results.get_nuclide_density(material_id=mid, nuclide="Mn56", step=1)
sigma = results.get_nuclide_uncertainty(material_id=mid, nuclide="Mn56", step=1)
```

The activation cross sections are resampled from their ENDF MF=33 covariance, folded against your own spectrum, and the schedule is re-solved per sample. That is exact to all orders in the matrix exponential; the solver is untouched and only its input changes. Omitting the argument costs nothing at all: no covariance is read, nothing is folded, and the inventories are bit-identical.

Every draw is a pure function of the seed, the sample and what is drawn: a nuclide for the cross sections (the linked group, where a cross-material block joins two evaluations), half-lives, decay branching, decay

energies, decay photons and isomeric splits, an evaluation for the fission yields, a spectrum for the flux, and a material's tallied rates, jointly, for the statistical error. So a seed reproduces a run regardless of sample count or iteration order. Leave `samples` unset and the driver adds samples, 64 at a time, until every tracked standard deviation is known to `convergence`, the standard error of a sigma relative to that sigma (0.05 unless set). It is 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, leaving out any sigma under 0.01% of its value. At least 128 samples run whatever the target and at most 1024; a run the cap stops sets `hit_cap`, and `get_data_uncertainty_info` lists every sigma that missed, worst first, under `unconverged`. A fixed `samples` runs exactly that many and lists what missed all the same, and a transport run fixes its count before the transport, 256 at the default target. The photon spectrum line by line and the contact dose are not judged, since their dose quantity and build-up are chosen when they are read; every `Estimate` carries its own `std_dev_standard_error`.

Nine sources can be perturbed, and `DataUncertainty.available_sources()` names them:

- `cross_sections`: the activation cross sections, from their ENDF MF=33 covariance folded against your spectrum. Where the library carries the evaluation's resonance parameters (MF=2 and MF=32, in `resonance_parameters.arrow`), each replica draws the parameters themselves from their covariance and rebuilds the resonance range's cross sections from the draw, so the resonance-range 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 says which and why.
- `flux_spectrum`: the spectrum itself, from the error you hand the pulse (below). A spectrum with no stated error contributes nothing.
- `half_life`: the half-life of every unstable nuclide the material can reach, from the decay data's own sigma on it. A replica's half-lives are used in its solve and in the activity, decay heat and dose evaluated from it, so a saturated activity stays as insensitive to its own half-life as it physically is.
- `decay_branching`: the decay branching ratios of every 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 spread, so a small branch with a large relative sigma is drawn skewed rather than held. A mode is held at its ratio 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 short of its stated sigmas, and why.
- `decay_energy`: each nuclide's mean decay energy, from the sigma on each beta, gamma and alpha component that states one, or on the total when no component states a sigma. A decay energy never enters the solve, so this moves decay heat and nothing else.
- `decay_photon_lines`: each decay photon spectrum's normalisation, one draw per spectrum shared by all its lines, and each line's own intensity and energy, from the decay data's sigmas on them. No photon enters the solve, so this moves the photon spectrum, the contact dose and, through the gamma decay energy, which follows each replica's drawn lines in place of a `decay_energy` draw, the decay heat. Each of these has a range rather than one spread, `Estimate.std_dev` to

`Estimate.std_dev_correlated`, because the decay data do not state how a nuclide's photon intensities are correlated.

- `fission_yield`: the independent yields of each parent the run gives a fission rate, from the DY the evaluation states on each, drawn independently since no evaluation states a correlation. A parent with yields that never fissions in the run (an actinide reached by capture from a structural material, say) is not drawn and is not counted as a gap; the transport-coupled path still lists every reachable parent with yields. It needs the evaluated yields, which the published fission yield libraries carry; a chain without them lists every parent 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. It applies on `Material.transmute` and on a transport run, where the tally scores each state's production on the covariance's intervals.
- `statistical`: the Monte Carlo error of transport-tallied reaction rates, from their per-history covariance, on a yamc `Model.simulate_transmutation` run with the independent method. A spectrum run's rates are a deterministic collapse, so on `Material.transmute` it has nothing to act on and is left out of the report's `sources`, as `flux_spectrum` is on a transport run.

`sources=` restricts a run to some of them, which is how a contribution is measured. Naming a source this build cannot perturb raises. The decay branching ratios that source does not sample, the isomeric splits of reactions with no MF=40 covariance, the material composition and the other inputs listed under What is not propagated stay at their evaluated or nominal values.

A spectrum that came from a Monte Carlo run carries a statistical error of its own. Hand it to the pulse, per bin, in the same order and units as the histogram values, and it is sampled alongside the cross sections:

```
flux_sigma = [2e10, 8e10, 1e12]          # one per group, as the histogram has

pulse = yani.Pulse(
    rate=sum(multigroup_flux),
    duration=(1, "h"),
    source=spectrum,
    flux_std_dev=flux_sigma,
)
```

Omitting it is the common case, since a spectrum taken from a published reference set carries no stated error. A run then reports the omission in `spectra_without_flux_sigma` of `get_data_uncertainty_info` rather than letting the flux read as known exactly.

A per-bin sigma treats the bins as independent, and a tally's bins are not: the same histories score them, so they move together, and a per-bin sigma understates the error on any rate that sums over a band of them. Where the correlations are known, give the pulse the full covariance of the histogram values as `flux_covariance` instead of `flux_std_dev`. It has to be symmetric and positive semi-definite, and is checked.

Because a zero sigma could mean either "well known" or "nothing published", the two are separated in `get_data_uncertainty_info` for a nuclide with no usable MF=33 block for any channel the chain drives (none on the tape, or only underivable NC blocks, blocks to an evaluation the run does not hold, lumped blocks stated in barns, partial-level or unsupported ones):

```
info = results.get_data_uncertainty_info(material_id=mid)
if info is not None:               # None unless data_uncertainty was passed
    info["perturbed"]              # had usable MF=33 covariance
    info["no_covariance_data"]     # no usable MF=33 block for a driven channel
    info["rate_fraction_covered"]  # share of each rate with a nonzero stated variance
    info["rate_fraction_covered_total"]  # ... and over the run, weighted by
production
    info["partials_above_rate"]    # channels whose sigma is overstated
    info["partials_below_rate"]    # channels whose sigma is understated
    info["half_lives_perturbed"]   # unstable nuclides whose half-life was sampled
    info["no_half_life_uncertainty"]     # ... and those whose data states no sigma
    info["decay_energies_perturbed"]     # nuclides whose decay energy was sampled
    info["no_decay_energy_uncertainty"]  # decay energies held for want of a sigma
    info["decay_branchings_perturbed"]   # multi-mode parents whose split was drawn
    info["half_life_uncertainty_not_carried"]     # a stated sigma no draw can
    info["decay_energy_uncertainty_not_carried"]  # carry, held; see the method page
    info["decay_photon_lines_perturbed"]  # nuclides whose photon data was drawn
    info["no_decay_photon_line_uncertainty"]  # photon data held for want of a sigma
    info["fission_yields_perturbed"]      # fissioning parents whose yields were drawn
    info["fission_yields_mapping_mismatch"]  # yields that did not map back, held
    info["isomer_branchings_perturbed"]   # reactions whose isomeric split was drawn
    info["no_isomer_branching_uncertainty"]  # splits held, no MF=40 on any state
    info["isomer_production_sigma"]       # MF=40's folded sigma per product
    info["resonance_parameters"]   # per nuclide: parameters sampled, or first-order
rows and why
    info["rates_floored"]          # rate draws floored at zero, see the method page;
                                   # also statistical_floored
    info["skipped_cross_material"] # {nuclide: blocks to an evaluation not in the run}
    info["skipped_other_file"]     # {nuclide: blocks whose partner is not a cross
section}
    info["mirrored_disagree"]      # pairs stored both ways whose copies differ
    info["not_perturbed"]          # every input held at nominal, see below
    info["sources"]                # the sources that applied to this run
    info["has_gaps"]               # True if something was left out or is inconsistent
```

`not_perturbed` names every input held at nominal on every run, from any resonance-parameter covariance (MF=32) neither sampled nor in `covariance.arrow` and the MT=5 product multiplicities to the photon spectrum covariance, dose coefficients and the material composition, plus the run-dependent entries: a source the run switched off, spectra without a flux sigma, tallied-rate statistics a transport run did not draw, the self-shielding correction, and the per-branch decay emission of a parent whose branching was drawn. What is not propagated explains each one, and names the few held inputs with no entry of their own, among them MF=33 blocks on partial levels (MT=600-849, 875-891), which

are neither listed nor counted unless an LTY=0 NC block names them, because the chain drives no rate for them. `skipped_cross_material`, `skipped_nc` and `lumped_covariance_not_assignable` count the cross-material blocks naming an evaluation the run does not hold, the underivable NC blocks and the lumped blocks stated in barns, none of which is used. A multi-mode parent whose branching was held whole is named in `no_decay_branching_uncertainty` or `decay_branchings_too_wide`, and one that was drawn short of its stated sigmas in `decay_branchings_partly_held` (some modes held) or `decay_branchings_sigma_adjusted` (a mode drawn more than 0.1% from its stated sigma, because the sum rule or non-negative ratios cannot carry it). MF=40 blocks that could not be used are counted per parent in `isomer_branching_blocks_not_used`. `has_gaps` looks only at the sources the run perturbs, and is True when one of them met a nuclide with no usable MF=33 block, a skipped or unassignable MF=33 block, a block whose layout is unsupported or malformed, a channel whose partial rates add up to more, or less, than its rate (`partials_above_rate`, `partials_below_rate`, below), a derived channel resting on an unstated covariance between opposing terms, a covariance repaired past round-off on a channel a draw can move (inside the populated bound or outside it), a cross-material covariance drawn at the closest admissible one (`cross_material_adjusted`), an evaluation its own library warns about (`covariance_warnings`), a spectrum with no flux sigma or a rate under one that the flux draw cannot move, a reachable unstable nuclide with no stated half-life sigma, or with a decay energy or decay photon data but no stated sigma on it, a half-life, decay-energy or photon sigma no draw can carry, a fissioning parent with no yield sigma, a yield sigma no draw can carry or yields that do not map back, a multi-mode parent whose branching was held or drawn short of its stated sigmas, or an isomeric split with no MF=40 to draw, an MF=40 block not used or an MF=40 matrix repaired. Partial-level blocks set it only through `no_covariance_data`, when they are all a nuclide has.

Read `rate_fraction_covered_total` before any sigma above it. A count of nuclides with MF=33 measures how much covariance exists. This measures how much of it lands on the reactions the run drove, weighted by rate and by the parent's own density.

Tungsten is a case where the coverage rests on a lump. ENDF/B-VIII.1, JEFF-4.0 and FENDL-3.2d state the `(n,2n)` that makes 98% of a tungsten foil's decay heat only lumped with `(n,2np)`, as the uncertainty of their sum (MT 852). Each component takes the lump's relative perturbation, so the two move together and their sum moves as the lump states. On ENDF/B-VIII.1 and the FNS spectrum a tungsten foil's `rate_fraction_covered_total` is 1, and `sources=["cross_sections"]` gives its decay heat a spread of about 3% at the first measured cooling time, nearly all of it W185m's.

Covered means the evaluation states a nonzero variance there, not that a covariance grid spans it. A grid can run across the whole range with a variance of zero on some intervals, and rate from those counts as uncovered, the same as rate from outside the grid. ENDF/B-VIII.1 W186 `(n,gamma)` is the case: its MF=33 block states zero from 1e-5 eV to 10 keV, where nearly all of a capture rate is, because the evaluation keeps that uncertainty in resonance-parameter covariance (MF=32). The published libraries carry the resonance parameters, which are sampled and cover that range, so on the FNS spectrum W186 `(n,gamma)` reads 1; on data with MF=33 alone it reads 0.07, and W186 stays under `perturbed` for the intervals above 10 keV.

The share is of the rate the fold divides by: the rate from energies with a nonzero stated variance, over the rate across the flux range, both with the dilute cross section on a dilute run and the shielded one on

a self-shielded run. Shielding depresses the resonance range, which is where capture blocks often state zero, so a shielded run's shares can sit well away from the dilute ones. The total weights each channel's share by the production the run actually drove, so it is the share of that production coming from covered energies. A transport run is the exception: its shares are of the dilute rate over the tally spectrum, the covered share of the tallied production is not computed, and `rate_fraction_covered_total` is `None` there rather than a figure weighting that production by dilute shares. The per-channel shares are still given, and the relative sigma is diluted by a different amount than they say.

`partials_above_rate` lists, with their ratio, the channels whose partial rates the covariance was weighted with add up to more than the rate it was divided by. The two were then computed different ways, and the relative sigma is overstated. A grafted `(n,n')`, whose rate is the metastables' MF=10 production while its partials are MT 4's, is one cause. The `1/E` within-group weight is another: there the part of a group a covariance edge cuts off is weighted by its share of the group's energy width, so the parts need not add up to the group's rate even on a dilute run, and by a lot: Fe56 `(n,p)` on a three-group spectrum whose fast group holds its 4.3 MeV covariance edge sums to about ten times its rate. A tallied rate on a transport run is a third. The coverage share is of the fold's own rate, not of the listed rate, and a channel in either map makes `rate_fraction_covered_total` `None`.

`partials_below_rate` is the same check the other way. A covariance grid that spans the whole flux range leaves no rate outside it, so its partial rates must add up to the rate, and a shortfall understates the sigma. The `1/E` weight gives one for a reaction falling with energy when a covariance edge cuts a group. A grid that stops short of the flux range cannot be checked this way, since rate from outside it rightly leaves its partials short. Both maps count towards `has_gaps`.

Three maps say which blocks were read but not folded. `skipped_cross_material` counts, per nuclide, the blocks correlating one of its reactions with a reaction of another evaluation that were not applied: the evaluation they name is not in the run, or one of the two reactions has no relative covariance to correlate. Where the run holds both evaluations the block is applied, and the nuclides it links are drawn jointly, one draw per replica shared by every material, counted in `applied_cross_material`; where the stated covariance is not admissible beside each evaluation's own, the closest admissible one is drawn and `cross_material_adjusted` says by how much (ENDF/B-VIII.1 U235 and U238 fission read about 0.31), which counts towards `has_gaps`. A block naming its partner by the evaluation's own MAT is not one of them; ENDF-102 allows that spelling alongside `MAT1 = 0`, and both are folded. `skipped_other_file` counts the blocks whose partner is not a cross section (`XMF1` other than 0 or 3, or a final state). Both count only blocks on a reaction the run drives, once per nuclide however many spectra the run has. `mirrored_disagree` is keyed `"Nuclide (n,a) (n,b)"`: a pair stored in both orientations is folded once, from the lower MT's section, and the other copy is checked against its transpose. A difference above 1e-5 of the pair's largest entry is listed with its size. JEFF-4.0 Be9 stores 130 pairs both ways, and they agree exactly. All three count towards `has_gaps`.

Both weights are needed. Rate alone, without the density of the parent each rate belongs to, counts a channel on a trace isotope the same as one on the bulk: on an iron foil that is 32% against 96%.

For a decay-only schedule there is no production to weight by, so the value is `None`, not zero. It is `None` on a transport run too, as above, and when either partials map lists a channel.

Activity, decay heat and contact dose come back with a band from the same ensemble, each as an `Estimate` holding the unperturbed value and the ensemble's spread on it:

```
activity = results.get_activity_uncertainty(material_id=mid, step=1)
activity.nominal                    # the unperturbed run, always present
activity.mean, activity.std_dev     # None below two replicas
activity.relative_std_dev

results.get_decay_heat_uncertainty(material_id=mid, step=1)
results.get_contact_dose_uncertainty(material_id=mid, step=1,
dose_quantity="effective")
results.get_activity_uncertainty(material_id=mid, step=1, by_nuclide=True)   #
dict[str, Estimate]
```

Each is evaluated once per replica and summed within that replica, so the correlations between nuclides survive. Building the same number out of the per-nuclide sigmas is wrong in a specific direction: evaluating from the mean inventory gives no spread at all, and adding sigmas in quadrature double-counts a variance that partly cancels, since every Mn56 atom in an irradiated iron foil came out of an Fe56 atom. `mean` and `std_dev` are `None` below two replicas, because a spread over one sample is unmeasured rather than zero.

With `displacement_damage=True`, the dpa and damage energy come with a band too, from `get_dpa_uncertainty` and `get_damage_energy_uncertainty`: see Displacement damage.

The decay photon spectrum comes back line by line, as `LineEstimate`s over the union of the lines the unperturbed run and every replica emit:

```
lines = results.get_decay_photon_spectrum_uncertainty(material_id=mid, step=1)
for line in lines or []:
    line.energy, line.nominal, line.std_dev
    line.energy_mean, line.energy_std_dev  # the energy each replica drew
    line.emitting              # replicas that emitted it at all
```

A line a replica does not emit counts as a zero in it, which is the only rule under which two lines' spreads are taken over the same sample. `emitting` keeps that zero-fill visible, so a dim line and an intermittent one stay distinguishable.

Every one of these bands is evaluated with each replica's own half-lives and, for decay heat, its own decay energies. With `decay_photon_lines` on, each replica also draws every line's emission per decay and energy, so a line's band is its emitters' activity and its own intensity together, and lines are matched across replicas on their nominal `energy`. Contact dose holds its attenuation coefficients, response and build-up factor the same in every replica.

`get_uncertainty_inventories(material_id=mid, step=step)` is still there for a quantity these four do not cover, and returns every replica's full inventory to take the spread over yourself.

Where an uncertainty comes from

`DataUncertainty(attribution=True)` also says where each nuclide's sigma comes from. The run then resamples each source alone as well as all of them together, so it costs more than one without:

```
results = material.transmute(
    schedule=schedule,
    data_uncertainty=yani.DataUncertainty(seed=42, attribution=True),
)
breakdown = results.get_uncertainty_breakdown(
    material_id=mid, nuclide="Mn56", step=1
)
if breakdown is not None:          # None unless attribution was asked for
    breakdown["variance"]          # the total, the square of the sigma above
    breakdown["by_source"]         # each source alone; these sum to the total
    breakdown["unattributed"]      # what the sum leaves, small when it holds
    breakdown["contributors"]      # (source, nuclide, reaction, variance), largest
first
```

The sources are independent, so `by_source` is how much of the variance is cross sections, half-lives, flux and so on, and `unattributed` is interaction and sampling noise. `contributors` is first order and says which evaluation to look at: within the cross sections a nuclide's whole evaluation has a `reaction` of `None` and each channel names its own, and a half-life or a parent's decay branching (its whole split) has `None`. The total stays the resampled one.

On a yamc transport run with the `statistical` source, `get_reaction_rate_uncertainty(material_id=mid, step=step)` lists each tallied rate with its standard deviation, as `(nuclide, reaction, target, rate, std_dev)` in 1/s per atom. The rates were scored by the same histories and are correlated; the inventory sigmas carry those correlations, and these standard deviations alone do not.

Reading ensemble diagnostics

The density sigma, its sampling error and the replica noise are different quantities. After a run with `data_uncertainty`, inspect them explicitly:

```
sigma_evolution = results.get_nuclide_uncertainty_evolution(
  material_id=material.id, nuclide="Co60",
)
sigma_error = results.get_nuclide_uncertainty_standard_error(
  material_id=material.id, nuclide="Co60", step=1,
)
replica_noise = results.get_nuclide_uncertainty_noise(
  material_id=material.id, nuclide="Co60", step=1,
)
activity_estimate = results.get_activity_uncertainty(material_id=material.id, step=1)
if activity_estimate is not None:
  print(activity_estimate.std_dev, activity_estimate.std_dev_range)
  print(activity_estimate.replica_spread, activity_estimate.noise_dominated)
```

The evolution includes the initial state, matching `get_nuclide_evolution()`. Unknown IDs or unavailable uncertainty results return `None`; they do not establish zero uncertainty. The noise tuple holds the replica spread and the Monte Carlo noise removed from it. `std_dev_range` describes how well the estimated sigma is settled, not a confidence interval on the physical value.

Limits worth knowing

- **A step is exact at any length, for a fixed spectrum.** The burnup matrix is built from decay constants and `sigma*phi` rates and does not depend on the composition, so it is constant over a step and the matrix exponential is solved exactly. One year in a single step and one year in 365 agree to round-off. Shortening steps to chase accuracy buys nothing here.

What a long step **does** miss is the spectrum changing as the composition does, which `transmute()` cannot see because the flux is your input rather than something it solves for. If the field would harden or soften appreciably over the campaign, split the schedule and give each pulse its own spectrum. That is a statement about the physics you are feeding it, not about the solver. - `data_uncertainty` covers the activation cross sections (with their resonance parameters where the library carries them), the half-lives, the decay branching ratios, the decay energies, the decay photon lines and continuum normalisations, the fission yields, the isomeric branching where an evaluation states MF=40, and the flux spectrum when a pulse carries `flux_std_dev` or `flux_covariance`. Without one the flux is taken as exact, since nothing is transported here. The decay branching ratios the draw holds, isomeric splits with no MF=40, resonance-parameter covariance (MF=32) neither sampled nor in `covariance.arrow`, cross-material blocks to an evaluation the run does not hold, underivable NC, barn-stated lumped and partial-level MF=33 blocks, the self-shielding correction, the photon spectrum covariance, the dose constants and the material composition are held at their evaluated or nominal values; What is not propagated says how much each can matter. - Nothing is self-shielded unless you give a shape or a chord, so a dilute run of a resonance absorber reads high. `would_shield` in `get_self_shielding_info` flags which nuclides on the run that skipped it, and how strongly their own resonances could bite. It is a screen, not a bound: only a shielded run says by how much. - The network is only as complete as the chain you configure. A product whose parent reaction is missing from `transmutation_reactions` never appears, and neither does any route through it. - A route table covers one step, to the depth you ask for. Each route is weighted by what its own reactions drove over that step, which is right while the intermediates barely burn. A route needing more reactions than `reaction_depth` or more decays than `decay_depth` does not appear, and the shares are shares of the routes that did. Raise the depths before reading an absence as impossible. - Nuclides many decades below the largest inventory carry no significant figures. Treat a deep trace as a bound, not a number.

Method

This page describes what YANI computes and in what order. The overview gives the governing equation. Where a cheaper option was tried and measured, the result is recorded here as well.

Overview

A run has six stages:

1. **Network.** The transmutation chain gives each nuclide's decay modes, reaction products and fission yields. YANI finds the products the material populates over the schedule by solving it, and loads cross sections for those and the composition only (Which nuclides get loaded).
2. **Rates.** Each reaction rate per atom is the pointwise cross section folded against the spectrum (The collapse), with an optional self-shielding correction (Self-shielding). Fission yields and isomeric splits are weighted by the same spectrum.
3. **Matrix.** Decay constants and reaction rates are assembled into the sparse matrix A (Building the matrix).
4. **Solve.** Each step of the schedule is one matrix exponential (The solve, Schedules).
5. **Derived quantities.** Activity, decay heat, photon emission, contact dose and clearance indices are calculated from each step's inventory and the decay data (Derived quantities).
6. **Uncertainty.** If requested, stages 2 to 5 are repeated with perturbed inputs (Nuclear-data uncertainty).

The solve

One timestep is a matrix exponential, $\mathbf{N}(t) = e^{At}\mathbf{N}(0)$, evaluated by CRAM at order 48 following Pusa 2010 and Pusa 2015, in the incomplete partial fraction form.

A is built from decay constants and $\sigma\phi$ rates and does not depend on the composition, so it is constant over a step and the exponential is solved exactly. Step length therefore does not affect accuracy for a fixed spectrum: one year in a single step and one year in 365 steps agree to round-off.

What a long step does not capture is a change in the spectrum caused by the changing composition. `transmute()` takes the flux as an input and does not solve for it. If the spectrum would harden or soften appreciably over a campaign, split the schedule into several pulses, each with its own `source`.

The 24 poles are evaluated sequentially, because the incomplete partial fraction form is a recurrence. The sparse LU runs single-threaded so that the order of floating point additions, and therefore the result, does not depend on the thread count. The symbolic factorization is computed once and reused for every pole.

Most of a transmutation network is acyclic, since decay chains only run one way. The linear solve at each pole is therefore done block by block over the strongly connected components of the network, in topological order. Blocks without a cycle are solved by substitution, and only blocks with a cycle (for example a capture followed by a decay back to the parent) use the sparse LU.

After the exponential, two adjustments are made:

- A nuclide with no parent present in the step is set to its closed-form value $N_i(0)\, e^{A_{ii}t}$ instead of the CRAM value. Both describe the same equation, and the closed form is exact, so a stable nuclide that is only being burned does not accumulate round-off.
- Densities at or below 10^{-30} atoms/b-cm are dropped.

Building the matrix

The coefficients of A come from two sources:

- **The transmutation network** gives decay constants, decay branching, reaction products and fission yields. Each of these can come from a different library through four independent settings. See Nuclear data settings.
- **The collapse** gives each $\sigma\phi$ (see The collapse).

A has one column per parent nuclide j:

- **Decay.** $\lambda_j = \ln 2 / T_{1/2}$ is added to the diagonal as a loss, and each decay mode adds $b\,\lambda_j$ to its daughter's row, where b is the mode's branching ratio. Emitted light particles are also produced, so an alpha decay produces He4. A mode with no daughter in the chain, such as spontaneous fission, removes atoms without producing any.
- **Reactions.** Each reaction rate R is added to the diagonal as a loss, and $b\,R$ is added to each product's row. The light ejectiles named by the reaction (H1, H2, H3, He3, He4) are also produced, so gas production is part of the inventory. A reaction whose product is the target itself cancels against its own loss term.
- **Fission.** The fission rate R_f is a loss, and each fission product p gains $R_f\, Y(p)$, where Y is the spectrum-weighted yield described in Fission yields.

Stable nuclides have no decay term and are removed only by reactions.

The collapse

Each $\sigma\phi$ is the pointwise cross section folded against the spectrum:

$$\sigma_{\text{eff}} = \frac{\sum_g \sigma_g \phi_g}{\sum_g \phi_g}, \qquad R = 10^{-24}\,\sigma_{\text{eff}} \sum_g \phi_g = 10^{-24} \int \sigma(E)\,\psi(E)\,\mathrm{d}E$$

where ψ is the piecewise-constant flux density given by the group fluxes. The total flux cancels, so the rate is the integral of the cross section over the spectrum. The cross section is used at the energy resolution of the evaluation, so the group structure of the spectrum is the only averaging applied.

Each σ_g is the integral of the cross section over the group divided by the group width. The integral uses the trapezoid rule over the group edges and every evaluated point inside the group. Cross sections are interpolated linearly between points, so the trapezoid rule is exact.

At the ends of the evaluated energy range:

- Below the first energy, a threshold reaction is zero and any other reaction takes its first value.
- Above the last energy, the cross section is zero, while the group keeps its full width. This would make the rate too low, so a run raises an error if more than 0.1% of the flux lies above the top of a nuclide's evaluation. See Energy range.

Cross sections are taken at the material's temperature. A temperature between two in the data is served by blending the two that bracket it, linearly in temperature, and one outside the data's range is refused, naming the range. If the material has no temperature set, the lowest temperature in the data is used, 250 K on the published libraries.

Within each group the flux is assumed flat in energy. This assumption has an error of its own. Compared with a pointwise reference for the FNS foil reactions on VITAMIN-J-175, flat within-group flux gives a 3.7% mean error on the hard FNS spectrum (21% with $1/E$ within each group). With a 10% $1/E$ tail added to the spectrum, the error is 28% for flat and 3.2% for $1/E$. Since neither shape is better in both cases, YANI uses the flat shape. Finer groups reduce this error for both shapes.

Fission yields

Fission yields are tabulated at a few incident energies, typically thermal, 500 keV and 14 MeV, and the yield depends on the energy of the neutron causing the fission. YANI uses the independent yields (MT=454) and combines the tabulated sets as

$$Y(p) = \sum_k c_k \, Y_k(p), \qquad c_k \propto \int \sigma_f(E)\, \psi(E)\, h_k(E)\, \mathrm{d}E$$

where h_k is a hat function equal to 1 at the k-th tabulated energy and falling linearly to 0 at the neighbouring tabulated energies, held constant beyond the first and last. Each set is weighted by the fraction of fissions near its energy, and the weights are normalised to sum to one. The interpolation between tabulated energies is always linear, regardless of the interpolation law given in the evaluation.

The set of reachable fission products is the union over all tabulated sets. A nuclide with yields and a nonzero fission rate but no weights raises an error.

Which nuclides get loaded

YANI does not truncate the network at a fixed number of reaction or decay steps. A fixed depth drops real nuclides when set too low and loads unneeded data when set too high. Nor does it load cross sections for everything the composition can reach: from almost any composition that is one large connected network, 465 nuclides with neutron data on ENDF/B-VIII.1 from Fe56, Al27 or anything from Be9 up, and hundreds of MB, where a year's irradiation of a 316 steel needs 99 of them.

Instead, YANI finds the products that matter by solving. It loads the composition's cross sections, solves the schedule, and loads every product with neutron reactions that, at any state of the schedule, reached `product_threshold` (default 10^{-12}) as its atom fraction or as its share of the activity, the decay heat or the decay photon line power. It then solves again with those products reacting, and repeats until a solve populates nothing new above the threshold. Decay shares are tested as well as atom fractions because a rare product can carry a decay quantity: Na22 in an irradiated Li4SiO4, at an atom fraction near 10^{-15}, carries nearly all of the contact dose a month after shutdown, and its own 28 kb thermal (n,p) burns it out twelvefold.

A product left out is still produced, decays and is reported; only its own neutron reactions are dropped. Its atom fraction and its share of each decay quantity are then off by less than the threshold, and what its reactions would have made is bounded by the summed peak atom fraction of the products left out. `get_product_screening_info` reports that sum as `dropped_fraction`, beside what was loaded, what was left out and how many solves the screening took. A threshold of 0 loads every product the solve populates at all, which is exact.

Isomeric branching

When a reaction leaves part of its product in a metastable state, the metastable state is a separate nuclide with its own half-life and decay chain. The branching fraction is stored as a function of incident energy and folded against the spectrum when rates are calculated, so runs with different spectra on the same target can have different branching fractions. `get_isomeric_branching()` returns the split used in each step. See Weighted, from a solve.

Self-shielding

The collapse assumes the given spectrum is the flux inside the material. Often the spectrum is the flux in the surrounding field instead. Inside a lump of material, the flux is depressed across resonances where the total cross section is large, so the true reaction rate is lower than the unshielded rate.

The correction changes the shape of the flux within each group and preserves the group totals, since those come from a measurement or a transport calculation. It is applied as a per-group, per-reaction weighting. A nuclide with no resonances in a group is unchanged.

The flux depression and the escape probability both depend on how easily a neutron leaves the lump, which depends on its geometry. A `Material` has no geometry, so the user gives a shape or a mean

chord length $4V/S$. YANI never infers a shape. Without one, no shielding is applied and the result is bit-identical to a run without self-shielding.

The flux inside the lump is found by solving

$$(\Sigma_t(u) + \Sigma_e)\phi(u) = \int_{u-\Delta u}^{u} \Sigma_s(u')\,\phi(u')\,\frac{e^{-(u-u')}}{1-\alpha}\,\mathrm{d}u' + \Sigma_e$$

from high to low energy in lethargy, so the in-scattering source uses the flux already calculated at higher energies. It makes no assumption that resonances are narrow.

The narrow-resonance approximation, which uses the asymptotic $1/E$ form for the in-scattering source, was implemented and tested but is not included. It over-shields nuclides with strong elastic resonances, where in-scattering fills in the capture dips. On a $1/E$ field at 50 μm it is 41% low on Co59, 63% low on W186 and 16% low on Mn55, and within 1% on Rh103, Fe56, Eu151 and Nb93. Against measurement, W186(n,γ) in FNG-tung is 1.29 b. The slowing-down solve gives C/E 0.80 and the narrow-resonance approximation gives 0.46, which is twice as far from the measurement as no correction. Whether the approximation is accurate for a given run depends on its spectrum and composition, which a user cannot easily check in advance, so it is not offered. See Self-shielding.

Schedules

A schedule is a list of steps, each with a duration and a rate, and each step is one matrix exponential. A `Cooldown` is a step with a rate of zero. Its matrix contains only decay terms, and it uses the same solver as an irradiation step. The inventory is reported at the start and at the end of every step, so the output times are the cumulative sums of the step durations. To get a decay curve, use many short cooldown steps, which are cheap; `cooldown_steps()` generates them.

On `Material.transmute`, rates are calculated once for each distinct spectrum. The collapse gives each reaction rate per unit flux from the shape of the spectrum, and each pulse multiplies these by its own `rate`. Pulses that share a `source` share one collapse. With self-shielding on, the flux depression is calculated from the starting composition. Within one spectrum, rates do not change as the composition changes.

Coupled to transport

In yamc, `Model.simulate_transmutation` uses a transport calculation instead of the collapse. Each material's reaction rates are tallied directly as continuous-energy track-length estimates of $\sigma(E)\,\ell$, scored at the energy of each track, so there is no group structure. In the unresolved resonance range, σ is the probability-table band the transport sampled for that flight (Levitt, Nucl. Sci. Eng. 49, 450 (1972)), not the smooth average: inside a band the flux and the cross section are anti-correlated, so scoring the average would over-count the rate in a material that shields itself there. Capture and the other

disappearance channels take the band's capture factor, fission channels its fission factor and elastic its elastic factor. The rate per target atom is

$$R = \frac{\langle \sigma \ell \rangle}{10^{24}\, V} \times S$$

where $\langle \sigma \ell \rangle$ is the mean per source particle in b-cm, V is the material volume in cm³ and S is the pulse's source rate in n/s. For this reason, `rate` is a source rate in yamc and a total flux in `Material.transmute`. Fission-yield weights are tallied in the same way, with the hat functions applied on each track.

There are two methods:

- **independent** runs transport once, with the starting composition, and scales the per-source-particle rates by each step's source rate. It is the faster method and the only one that supports nuclear-data uncertainty.
- **coupled** runs transport at the start of every irradiation step with the current compositions, and updates the materials in the geometry after each step. Rates are held at their beginning-of-step values for the whole step, with no predictor-corrector, so split any step long enough for burnup to change the flux.

Both methods start with a short scouting transport that gives each material's flux spectrum. The products whose cross sections are loaded and tallied are chosen on it by the same screening as Which nuclides get loaded.

Derived quantities

Each derived quantity is a sum over nuclides of atom density times a per-atom value from the decay data. With N_i in atoms/b-cm and V in cm³, the number of atoms is $10^{24} N_i V$.

Activity is $A_i = \lambda_i \, 10^{24} N_i V$ in Bq, with $\lambda_i = \ln 2 / T_{1/2}$ from the same chain used in the solve. Stable nuclides are omitted.

Decay heat is $P_i = A_i \, \bar{E}_i$, where $\bar{E}_i$ is the mean energy released per decay, the sum of three MT=457 average energies:

- light particles: beta, conversion and Auger electrons
- electromagnetic: gamma and X-rays
- heavy particles: alpha, protons, neutrons and fragments

Neutrino energy is excluded because it is not deposited. `component=` returns the beta, gamma or alpha part on its own. If a nuclide that produces heat has no component split in its data, an error is raised. Some decay records contain $Q/3$ placeholder energies instead of evaluated ones; see Some decay records are placeholders.

Specific values replace V: `per="cm3"` sets it to 1 and `per="g"` sets it to $1/\rho$, so neither needs the material volume.

Decay photon lines come from the MF=8 discrete spectra of the gamma and X-ray radiation types. Each line's emission rate per atom is λ_i times its intensity per decay, and the line spectrum is that rate times the number of atoms, summed over nuclides. Lines at the same energy are merged. The lines are not binned, so the spectrum can be used directly as a photon source.

Decay photon continua are the parts of decay photon spectra given in the evaluation as a density over energy instead of as lines, for example the photons from spontaneous fission or the emission of nuclides far from stability. Each continuum keeps its own energy grid and ENDF interpolation law and is not resampled or merged. Its emission rate is the exact integral under that law. These are evaluated decay spectra. Bremsstrahlung is not modelled.

Clearance index

A clearance, exemption or disposal limit set gives a limit L_i for each radionuclide, in Bq/g, Ci/m³ or Bq depending on the regulation. The index is

$$I = \sum_i \frac{a_i}{L_i}$$

where a_i is the nuclide's activity in the same units. The material meets the set's limits when I is below the set's threshold, normally 1. The activities use the half-lives of the transmutation chain, the same ones as `activity()`. Specific activities are divided by the mass of the whole material, so stable nuclides must remain in the inventory.

The limit tables and the arithmetic come from the radiological-material-clearance-finder crate. Depending on the set, the limit used for a nuclide can be adjusted:

- a nuclide the table does not list takes the set's catch-all limit, if it has one, and is listed in `defaulted`;
- where a parent's limit already includes a daughter in secular equilibrium, the daughter's activity is left out (`excluded`), or the part the parent accounts for is credited and only the rest is assessed (`credited`);
- activity with no limit and no catch-all is not in the index and is reported in `uncovered` and `uncovered_fraction`.

A low index with a large `uncovered_fraction` does not show that the material meets the limits. See Clearance and disposal limits.

Contact dose

The contact dose rate treats the material as a half-space that emits photons uniformly. At the surface, the uncollided flux at energy E from a uniform isotropic source of q photons/cm³/s is $q/2\mu(E)$, where μ is the material's linear attenuation coefficient. The size of the material cancels, so the result needs no volume or distance. Summed over all lines of all nuclides:

$$\dot{D} = \frac{B}{2} \sum_i 10^{24} N_i \sum_\ell S_{i\ell} \, \frac{E_\ell \, (\mu_{\mathrm{en}}/\rho)_{\mathrm{air}}(E_\ell)}{\mu(E_\ell)}$$

for absorbed dose in air, where $S_{i\ell}$ is the emission rate of line ℓ per atom and $\mu = \sum_e \rho_e \, (\mu/\rho)_e$ is calculated from each element's partial density and its NIST XCOM mass attenuation coefficient. $(\mu_{\mathrm{en}}/\rho)_{\mathrm{air}}$ is from NIST SRD 126. For effective dose, $E\,(\mu_{\mathrm{en}}/\rho)_{\mathrm{air}}$ is replaced by the ICRP-116 photon fluence-to-effective-dose coefficient for the AP geometry. Both tables are interpolated log-log, and lines outside the energy range of both tables are not counted. Continua are integrated exactly under their interpolation law, using Gauss-Legendre moments of the response.

The build-up factor B (default 2) accounts for photons that scatter and still reach the surface. It is the largest source of error: compared with a photon transport calculation of the same half-space, it gives a result 16 to 17% high for Co60 in steel. The method follows UKAEA-CCFE-RE(21)02, Appendix C.7.1.

Dose rates away from the material need a photon transport calculation. The photon lines can be used as its source, and `dose_coefficients()` provides the ICRP-116 and ICRP-74 coefficients, including H*(10), to fold with its flux tally.

Displacement damage

Unlike the quantities above, displacement damage is a reaction quantity, not a decay one: it accrues during pulses and cooldowns add nothing. With `displacement_damage=True`, each nuclide's damage-energy cross section (MT=444) is folded against the pulse spectrum by the same collapse as the reaction rates, self-shielding included. Each irradiation step adds, per element e, that rate averaged over the element's own nuclides, times the step's flux and duration:

$$\Delta E_{\mathrm{dam},e} = \frac{\sum_{i \in e} \left(N_i^0 + N_i^1\right) \bar{\sigma}_{444,i}}{\sum_{i \in e} \left(N_i^0 + N_i^1\right)} \, \phi \, \Delta t$$

where N^0 and N^1 are the atom densities at the start and end of the step. Averaging the two ends is the trapezoidal rule on the damage deposited, so a nuclide that burns out over a step contributes less to it and nothing after, and a product element that appears within the step is still defined. Displacements per atom follow the NRT model (Norgett, Robinson and Torrens 1975; ASTM E521):

$$\mathrm{dpa}_e = \frac{0.8 \, E_{\mathrm{dam},e}}{2 \, E_{d,e}}$$

Only the linear branch is applied. MT=444 is already integrated over the recoil spectrum, so the per-recoil threshold steps (no displacement below E_d, one up to $2E_d/0.8$) cannot be applied to it, and in a fast spectrum the recoils they would change carry a negligible share of the damage energy.

The material total is the atom-fraction-weighted sum over elements, $\sum_e f_e \,\mathrm{dpa}_e$, so each element's damage energy is converted with its own E_d before the elements are combined, and the damage-energy total is weighted the same way. This is the elemental superposition of NRT, which is defined for a monatomic target: each element's recoils are treated as slowing down among atoms of their own kind. It reduces to the elemental value for a pure element, and leaves out the transfer of recoil energy between elements in a cascade, which the polyatomic displacement functions of Parkin and Coulter (1981) describe and a single E_d per element cannot.

E_d is the spatially averaged displacement threshold energy: the ASTM E521 recommended value where the standard gives one, otherwise the average from Table 2.4 of the OECD-NEA report *Primary Radiation Damage in Materials* (NEA/NSC/DOC(2015)9). An element in neither has no default, since published values spread too widely to pick one silently, and must be given through `displacement_energies=`.

Nothing missing is counted as zero silently. A nuclide of the starting composition with no MT=444, or an element of it with no E_d, is refused. A product with no MT=444 (often a short-lived one the library does not publish) is left out and listed in `without_damage_energy`. A product element with no E_d (hydrogen and helium from (n,p) and (n,α), rhenium from tungsten) still adds to the damage-energy total but gets no dpa, and is listed in `without_displacement_energy`. Both lists carry the largest atom fraction each reached on an irradiated step, and `get_displacement_damage_info` also reports the E_d and its source for every element. The same conversion is used for the damage-energy tally of a transport run, so the two paths cannot drift apart. See Displacement damage.

Nuclear-data uncertainty

The solve is deterministic and every entry of the matrix is a rate, so uncertainty is propagated by perturbing the inputs to those rates and repeating the solve. This is exact to all orders in the matrix exponential: nothing is linearised, the sandwich rule is not used, and the solver is unchanged.

Folding the covariance

A reaction rate is linear in the cross section, and MF=33 gives a covariance that is constant on each interval of the evaluation's energy grid. Together these give

$$\mathrm{Cov}(R_i, R_j) = \sum_{k,l} C_{ij}[k,l]\; r_i[k]\; r_j[l]$$

where $r_i[k]$ is reaction i's rate restricted to covariance interval k. This is exact and does not project the covariance onto the flux group structure, since the integration is done on the covariance grid. Each block is folded on its own grid and the contributions are summed.

LB=8 blocks are the exception. An LB=8 block is a short-range variance in barns squared, not relative to the cross section. ENDF-102 section 33.2.2.2 states that the average over a subinterval ΔE_j of its interval ΔE_k has variance $F_k\,\Delta E_k/\Delta E_j$ and is uncorrelated with any other subinterval. Its contribution to a rate therefore depends on how the flux varies within ΔE_k, which is the only place the flux groups enter. The fold splits ΔE_k at the group boundaries, where the flux density ψ is constant, so the rule applies exactly to each piece and the variance of the rate is $F_k\,\Delta E_k \sum_j \psi_j^2\,\Delta E_j$. A flux that is flat over the whole interval gives the plain absolute diagonal, which is the smallest this term can be for a given flux in the interval. The more the flux is concentrated within an LB=8 interval, the larger the term: ENDF/B-VIII.1 Cr52 (n,p) is 17.7% for a flux flat over the tape's [14, 16] MeV interval and 22.5% for a flux in a single 0.2 MeV group at 14.1 MeV.

A covariance grid does not have to cover the whole flux range. The part of a rate from outside the grid has no stated uncertainty, so it is included in the denominator but not the numerator, and the relative uncertainty is smaller than the covariance grid alone would suggest. `rate_fraction_covered` records the fraction of each rate that has a stated uncertainty: the rate from energies where the reaction's diagonal variance, summed over its blocks, is nonzero, divided by the rate over the whole flux range. An interval with a variance of zero counts as uncovered. The rate used is the rate in the fold, dilute on a dilute run and shielded on a self-shielded run. On a transport run, the tallied rate is different: the fraction there is of the dilute rate over the tally spectrum, the covered fraction of the tallied rate is not calculated, and the production-weighted `rate_fraction_covered_total` is `None`.

Drawing a replica

For the cross sections, a replica draws each nuclide's evaluation once, on its covariance cells (the energy intervals of its blocks), and every spectrum, step and channel reads its rate off that one draw, so one evaluation is one uncertainty wherever it is used. The relative cells are drawn as a multivariate lognormal with mean one and the evaluation's own covariance C:

$$\Sigma = \ln(1 + C), \qquad LL^T = \Sigma, \qquad z \sim N(0, I), \qquad m_k = \exp\left((Lz)_k - \tfrac{1}{2}\Sigma_{kk}\right)$$

with the logarithm taken element by element, so each multiplier m_k has mean one, $\mathrm{Cov}(m_k, m_l) = C_{kl}$, and no perturbed cross section is negative. A rate is linear in the cross section, so the perturbed rate is exact rather than linearised:

$$R_i' = R_i + \sum_k r_i[k]\;(m_k - 1)$$

with $r_i[k]$ the partial rate of the fold above. Absolute blocks (LB=0) are drawn as normal shifts in barns, and LB=8 short-range variances as independent normal pieces over the flux groups that cut their intervals; a rate that reads one of these, or that subtracts reactions, can come out negative, and is then set to zero and counted in `rates_floored`. A normal draw of the relative cells, $R_i(1 + \delta_i)$, can give negative rates: on the FNS decay-heat benchmark it sent 8.3% of all sampled rates below zero (13% on iron), and setting them to zero biases the mean upward.

A covariance no lognormal can carry, two fully correlated cells with different sigmas or an anticorrelation with $1 + C_{kl} \leq 0$, is drawn at the nearest one that keeps every sigma: where Σ is not positive semi-definite its correlation matrix is replaced by the nearest correlation matrix, and the nuclide is listed in `lognormal_not_carried` with how far its correlations moved. That is a property of the distribution rather than a defect of the data, so it does not set `has_gaps`.

The evaluation gives a covariance but no distribution, so the lognormal shape is an assumption. For a channel with sigma near 1 or above, this assumption determines the tails. Such channels are heavy-tailed and their sampled spread converges slowly: at 1024 replicas, the maximum the driver adds automatically, the median sample standard deviation is 0.98 of sigma at $\sigma = 1$, 0.92 at 2, 0.83 at 3 and 0.50 at 9. TENDL-2017 has channels at 10^4% and above. The report lists every sampled channel of a populated nuclide with an evaluated relative sigma of at least 1 in `sigma_at_least_one` and at least 10 in `sigma_at_least_ten`. Channels with sigma of 1 or more on nuclides outside the populated bound (described below) are listed in `sigma_at_least_one_outside_bound`, since a draw on such a channel can be orders of magnitude above nominal and populate the nuclide.

Each factor comes from an eigendecomposition of a correlation matrix, scaled back by the standard deviations, so every cell is factorised as accurately as its own variance allows: cyclic Jacobi for small blocks, and a blocked tridiagonalisation with divide and conquer for large ones, whose eigenvectors are signed by a fixed rule. Each runs in a fixed order, so a factor depends only on the matrix and is the same on any thread count. Cholesky would require positive definite matrices, and MF=33 matrices are often not even positive semi-definite as evaluated.

Such a matrix has no square root, so it is repaired before it is sampled. Clipping its negative eigenvalues would only ever add variance. Instead, the correlation matrix of the cells with a positive stated variance, $R = D^{-1/2} C D^{-1/2}$, is replaced by the nearest correlation matrix and rescaled by the evaluated sigmas, so every such cell is sampled at its evaluated sigma exactly and only 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 sampled sigma can still move, either way.

A matrix counts as repaired when any of the following holds:

- R, over the cells with a positive variance, has an eigenvalue below $-m \cdot 10^{-12}$, where m is the number of those cells;
- a cell has a negative variance;
- a cell has zero variance and a nonzero covariance with another cell.

R is congruent to that block of C, so one is positive semi-definite exactly when the other is. On R the round-off of the decomposition is about m times machine epsilon regardless of the cell sigmas, so a positive semi-definite matrix whose smallest eigenvalue is a few ulps below zero does not count as repaired. The test does not compare $\lambda_{\min}$ of C with its $\lambda_{\max}$, because TENDL-2017 has channels with relative sigma up to 5.6×10^8 (variance near 3×10^{17}), and one such channel would set the threshold far above a real repair among the other channels.

The repair is reported in these fields:

- `covariance_repaired` lists the nuclides with a repaired channel that a draw can move (one with a positive rate on a spectrum used in the schedule). Any entry sets `has_gaps` to true.
- The report covers the nuclides the material can populate. The fold uses every nuclide whose data the run loaded, the composition and the products that reach `product_threshold`, so a nuclide is reported only when an upper bound on its density over the schedule, at nominal rates, reaches the solver floor of 10^{-30} atoms/b-cm. A nuclide outside the bound cannot change any nominal density by that much, but a replica's rates on a wide channel can be orders of magnitude above nominal. Repaired nuclides outside the bound with a channel a draw can move are listed in `covariance_repaired_outside_bound`, and any entry also sets `has_gaps` to true.
- `covariance_repairs` records, for each repaired populated nuclide and spectrum, $\lambda_{\min}$ of R before the repair, `largest_correlation_change` and `correlation_frobenius_change` (the largest single change and the Frobenius change of R), the `cells` repaired and the `held_cells`, whether the repair `converged`, and each channel's evaluated variance (as stated, even when negative) and evaluated sigma beside the sigma it is sampled at.
- `worst_sigma_change` is the largest $|\sigma_{\text{sampled}}/\sigma_{\text{evaluated}} - 1|$ over the repaired channels a draw can move, and is infinite where a repair gave a spread to a channel whose stated variance is zero or negative. Every cell with a positive stated variance keeps its sigma, so a change here comes from a channel that folds several cells, whose correlations moved, or from a held cell.
- `rate_weighted_sigma_change` is the weighted mean of each channel's $|\sigma_{\text{sampled}}/\sigma_{\text{evaluated}} - 1|$. Each channel is weighted by its rate at unit flux, its spectrum's fluence in the schedule and its parent's initial density. Weighting the change instead of the sigma stops a wide channel with a small rate from dominating. Because of the initial density, this is a first-generation measure: produced nuclides have no weight, and their repairs appear in the other fields.

The sampled sigma is read from the factorization for every matrix, so matrices below the repair threshold show their round-off there.

The other sources are sampled using their own stated sigma:

- **Flux.** Drawn once per replica and group, from the pulse's `flux_std_dev` or through the factor of its `flux_covariance`, and shared by every rate collapsed against that spectrum, the isomeric splits and the `(n,n')` rates the branching overlay folds from it included. A rate is linear in the flux, so the perturbed rate is the collapse's per-group terms reweighted exactly, without repeating the collapse.
- **Half-lives.** Drawn per nuclide from the sigma in the decay data.
- **Decay energies.** Drawn from the sigma on each beta, gamma and alpha component that has one, or on the total if no component has a sigma.
- **Decay photons.** Per spectrum, one draw from the sigma on its normalisation (FD for lines, FC for a continuum) scales all of its lines or continuum points, so the lines of one spectrum move together. Each line then takes its own draw from its intensity sigma (dRI, which MT=457 states relative to FD and without FD's sigma, so nothing is counted twice) and one from its energy sigma (dER). A nuclide's gamma decay energy follows the draw, by the drawn change in its photon energy per

decay (each line's energy times its intensity, plus each continuum's energy integral), in place of a `decay_energy` draw of it; the part of it the tabulated spectra do not carry is held, and the beta and alpha parts keep their own draws.

- **Fission yields.** Each independent yield (MT=454) the tape states a DY for is drawn on the tape's own product, and the draws are summed onto the chain's products the way the converter summed the nominal yields: a product with no decay data lands on its stand-in. The mapping is not stored, so it is rebuilt with the converter's rule and checked against the nominal yields. An actinide that borrows another's yields shares that evaluation's draw.
- **Isomeric branching.** Where the branching subsection carries the MF=40 covariance of a reaction's final-state productions (`branching_covariance.arrow`), each state's production is drawn as a lognormal multiplier on the covariance's own energy intervals, folded with the same spectrum and weight as the nominal split, and the split is remade from the drawn productions by the nominal branching rule. Where the list names its ground state, the shares are the drawn productions over their drawn sum and the reaction total stays the one the cross-section draw gives, so a factor common to every state cancels and the uncertainty of the total is not counted twice. An isomer listed alone, and every `(n,n')` isomer, is an absolute production: it is made at its drawn value and the ground state takes the rest of the total, so its own MF=40 sigma replaces the MF=33 sigma of its reaction that it carries with the cross sections alone, and adding this source can lower its sigma. A draw above the drawn total is scaled back to it and counted in `isomer_branchings_clipped`. On a transport run the tally also scores each stated state's production on the MF=40 intervals, so the same draw re-splits the tallied partials; a tally that scored no intervals holds every split.
- **Resonance parameters.** Where a nuclide's folder carries its resonance parameters and their covariance (MF=2 and MF=32, `resonance_parameters.arrow`, in the published libraries), each replica draws the parameters themselves, rebuilds the resonance range's cross sections from them and Doppler broadens the change to the temperature in use, so that range's uncertainty is exact in the parameters rather than first order. A nuclide whose parameters cannot be sampled falls back to the first-order MF=32 rows in `covariance.arrow`; `resonance_parameters` in the report says which was used, and why.
- **Tallied rates.** On a transport run, drawn jointly from their per-history covariance.

The decay data gives an expected value and a standard deviation for each value, with no distribution or correlation (ENDF-102 section 29.1), and each nuclide is sampled independently. A half-life, decay energy, photon value or fission yield is its nominal value times a lognormal factor with mean one and variance equal to the squared relative sigma, so every draw is positive and the ensemble has the stated mean and sigma exactly, with no floor. The independence between a nuclide's decay-energy components and between different nuclides' half-lives is an assumption, since the data gives no correlation. A sigma that cannot be sampled (one on a value of zero, such as a decay energy or a yield, or one that is not finite) is held at nominal and listed in `half_life_uncertainty_not_carried`, `decay_energy_uncertainty_not_carried`, `decay_photon_line_uncertainty_not_carried` or `fission_yield_uncertainty_not_carried`, all counted as gaps.

Decay branching ratios are drawn per parent, for every parent with two or more modes and a stated sigma. MT=457 gives each mode a ratio and a sigma with no covariance between modes, and the ratios

sum to a fixed total T, so the deviations of a draw must sum to zero. Such a zero-sum covariance exists only when no mode's sigma exceeds the sum of the others'. Of those 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. When the largest sigma exceeds the sum of the others', that mode is drawn at their sum, the most the sum rule allows.

Every moving mode but one is a logit-normal on $[0, T]$ whose mean and standard deviation are fitted to the evaluated ratio and its sigma, so a small branch with a large relative sigma is drawn skewed and never below zero. The modes are coupled through a Gaussian copula so that they carry the target covariance, and the last mode is their complement, which keeps the total. A single mode stating no sigma is that complement when it is the largest mode or when only one other mode states a sigma, as an evaluator writing a dominant mode as one minus the others intends. A mode is held at its ratio only when its stated sigma is one no ratio with its mean can have, $\sigma^2 \geq b(T - b)$ for a ratio b, or when it states none and is not the complement; the other modes share what is left. A draw whose complement would fall below zero is pulled back towards nominal and counted in `decay_branchings_floored`.

A parent held whole is listed in `no_decay_branching_uncertainty` (no mode states a sigma, or one does beside two or more that do not) or `decay_branchings_too_wide` (every mode that could move states a sigma no ratio can have). A parent drawn short of its stated sigmas is listed in `decay_branchings_partly_held` (some modes held) or `decay_branchings_sigma_adjusted` (a mode drawn more than 0.1% from its stated sigma, because the largest exceeds the sum of the others' or the complement needs a correlation non-negative ratios cannot carry). Each of the four counts as a gap. Only the inventory changes: a sampled parent's lines and decay energy per decay use its nominal branching.

The flux draw is a lognormal too, by the cross sections' transform: each bin's flux is scaled by $\exp(y_g - \Sigma_{gg}/2)$ with $y \sim N(0, \Sigma)$ and $\Sigma = \ln(1 + C)$ element by element, C the relative covariance of the pulse's `flux_std_dev` or `flux_covariance`. No sampled flux is negative, even for a bin whose relative error exceeds 100%, and nothing is floored; a `flux_covariance` no lognormal carries is drawn at the nearest one and listed in `flux_lognormal_not_carried`. The tallied-rate draw stays normal, because it carries the per-history correlations between rates: a tallied rate drawn below zero is set to zero and counted in `statistical_floored`, which biases the mean of that source upward, so a nonzero count means the normal distribution is a poor description of the tally.

Seeds depend only on their arguments. A cross-section, half-life, decay-branching, decay-energy, decay-photon or isomeric-branching draw depends only on `(seed, replica, nuclide)`, the nuclide being the parent for an isomeric split: not on the number of replicas, the order they ran in, or the other nuclides in the material. Resonance parameters add the range, `(seed, replica, nuclide, range)`, and fission yields are keyed on the evaluation. Nuclides linked by an applied cross-material block are drawn jointly from `(seed, replica, group)`, the group being every linked nuclide of the run, so their draw depends on which partners the run holds, though their distribution does not. The flux draw depends on `(seed, replica, spectrum)`, where spectrum is the spectrum's index among the schedule's spectra. The statistical draw depends on `(seed, replica, material)`, the material being its slot in the yamc geometry's material store rather than its id: the model fixes the slot, while an id the user did not set

depends on what else the process built. It is made jointly over that material's tallied rates, so adding or removing a tallied rate changes the other draws of that material. A given seed always reproduces a run.

Derived quantities under uncertainty

Activity, decay heat, contact dose and every photon line are calculated once per replica, with that replica's half-lives, and the spread is taken over the results. They are not combined from per-nuclide sigmas.

Adding in quadrature would be wrong. Every Mn56 atom in an irradiated iron foil came from an Fe56 atom, so the two densities are anticorrelated and their spreads partly cancel. Quadrature would count variance that is not there. Calculating once from the mean inventory would also be wrong: activity and decay heat are linear in N, so this gives the correct mean but a spread of zero. Contact dose is not linear in the densities at all, since a replica that produces more of an emitter also absorbs more of its photons.

The replica's half-lives must be used here as well as in the solve. A saturated activity is $\lambda N = R$, which depends very little on its own half-life. Using the nominal λ would give it the full spread of $N = R/\lambda$. A line's intensity per atom is its emission per decay times λ, so it is also rescaled with the replica's λ, which keeps the emission per decay at the value the decay photon draw then starts from. Decay energies and decay photons are only used here: they are sampled where decay heat, the photon spectrum and contact dose are calculated and affect nothing else. The photon draw moves line energies as well, so the photon spectrum matches a line across replicas on its nominal energy.

What is not propagated

Nine sources are propagated: activation cross sections (with resonance parameters where the library carries them), the flux spectrum (on `Material.transmute`), half-lives, decay branching ratios as described above, decay energies, decay photon lines and continua, independent fission yields, isomeric branching from MF=40, and the statistical error of tallies (on a transport run). Everything listed below is held at its evaluated or nominal value in every replica. Most of these have their own uncertainty or model error, which is then missing from the sigma, so the sigma is underestimated, and for some results this is the largest term. Two can go either way, because holding them also removes a feedback that reduces the spread: self-shielding, and, on `Material.transmute`, the response of the flux to the cross sections of a material that shapes its own flux. Their entries below say which way each goes. The covariance blocks that are skipped because they correlate an evaluation with one the run does not hold can also go either way: dropping a covariance term lowers the variance of a sum when the term is positive and raises it when the term is negative.

- **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. The draw cannot reproduce a stated sigma the sum rule or non-negative ratios cannot carry, and reports where it fell short.

- **Fission yield correlations and energy weights.** No evaluation publishes a correlation between yields, so each is drawn independently and a draw's yields are not renormalised to the nominal total. A parent whose tape yields do not sum back to the chain's (`fission_yields_mapping_mismatch`) is held. On `Material.transmute` the weights that combine a nuclide's yield sets by incident energy come from the nominal spectrum in every replica, so a flux draw does not change them; on a transport run with `cross_sections` on, each replica tallies its own.
- **Isomeric branching without MF=40.** The split of a reaction product between the ground state and an isomer comes from MF=9 or MF=10. Where an evaluation gives its uncertainty, in MF=40, the `isomer_branching` source draws it, on `Material.transmute` and on a transport run alike. A split whose evaluation states 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 (another evaluation's partner, a partner that is not an MF=10 production, NC, absolute and short-range layouts, a state the branching rule folds no curve for) are counted in `isomer_branching_blocks_not_used`. With the branching overlay configured (`transmutation_branch_ratios`), a flux draw on `Material.transmute` moves each split with the groups its states are produced in, and re-makes the `(n,n')` rates the overlay adds to reach a metastable state, including In115 to In115m. Those channels have no group-averaged cross section and no cross-section covariance of their own.
- **Resonance-parameter covariance (MF=32) neither sampled nor written.** ENDF-6 gives the cross-section covariance in the resonance range as an MF=32 part plus the MF=33 part, and many evaluations put the whole resonance-range uncertainty in MF=32 and leave MF=33 at zero there. Where the library carries the resonance parameters, the draw samples them exactly (above); otherwise the converter's first-order MF=32 rows in `covariance.arrow` are folded like any MF=33 block. What is left held is a range whose formalism `endf` does not reconstruct, and data converted before either was written, on which the fold sees MF=33 only and a capture rate driven by resonance flux can have a sigma near zero. For capture in a $1/E$ field, NJOY ERRORR gives ENDF/B-VIII.1 W186 1.53% with MF=32 and 0.00% from MF=33 alone. On such data the coverage report shows this: W186's capture block states zero variance from 10^{-5} eV to 10 keV, where nearly all the capture rate is, so on the FNS spectrum its `rate_fraction_covered` is about 0.07, against 1 on the published library, which samples the resonance parameters. W186 is still listed under `perturbed`, since the block's other intervals are used. The same check gives 6.32% against 0.01% on the JEFF-4.0 Ag109 tape and 4.71% against 0.00% on the TENDL-2025 Co59 tape.
- **MF=33 blocks that are not used.** These are:
  - blocks whose `MAT1` names an evaluation the run does not hold, or that correlate a reaction with no relative covariance of its own; a block naming an evaluation the run does hold is applied, and the two nuclides are drawn jointly;
  - blocks whose partner is not a cross section (`XMF1` other than 0 or 3);
  - NC blocks that cannot be derived (LTY 1 to 4, and the LTY=0 blocks counted in `skipped_nc`);
  - lumped reactions (MT=851 to 870) whose block is stated in barns rather than relative, which says nothing of how the sum splits;
  - blocks on partial levels (MT=600 to 849 and 875 to 891) where the chain uses the total and no LTY=0 NC block or lump names them.

A block naming the evaluation's own MAT is folded, as are an LTY=0 NC block's derivation from the reactions it names, a lump with a single component, and a relative lump of several: each 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. `skipped_cross_material`, `skipped_other_file`, `skipped_nc` and `lumped_covariance_not_assignable` count the first four kinds, on reactions the fold reaches, and all set `has_gaps`; no library yani reads states a partner that is not a cross section, an NC block with LTY 1 to 4, or a lump in barns. Partial levels are not counted, and the rates they would cover count as uncovered in `rate_fraction_covered_total`. A nuclide whose blocks are all of these kinds, or all on reactions the chain does not use, is listed under `no_covariance_data`, which sets `has_gaps`. Tungsten's `(n,2n)` is lumped: ENDF/B-VIII.1, JEFF-4.0 and FENDL-3.2d give it only together with `(n,2np)`, as MT 852, and it is sampled through that lump. JEFF-4.0 Be9 gives its `(n,2n)` only on the levels MT=875 to 890, and ENDF/B-VIII.1 Ca40 gives its `(n,p)` and `(n,a)` only on MT=600 and 800, so Ca40 is listed under `perturbed` through its capture alone. - **Self-shielding.** With a shape or a chord, every replica uses the flux depression calculated from the evaluated cross sections. Holding it fixed drops two effects with opposite signs. A larger capture cross section would deepen its own dip, which narrows the spread, and this is not included. The elastic and total covariance does not reach the correction either, which would widen the spread where MF=33 gives a resonance-range uncertainty for elastic scattering. Which effect is larger depends on the evaluation. For a 0.1 mm Au197 foil under $1/E$ on VITAMIN-J-175, ENDF/B-VIII.1 gives no resonance-range elastic uncertainty, and the held sigma is 1.8 times too large (1.70% held against 0.96% with the feedback). TENDL-2017 does give one, and there the held sigma is too small: 5.34% held against 6.07% with the feedback and the elastic term together (2.23% with the feedback alone). These figures fold the shielded partials, as the fold does. - **Response of the flux to the cross sections, on `Material.transmute`.** There the flux is the one the pulse is given, so a replica's cross sections move the rates and leave the flux as it is. For a trace activation product this is exact to first order in its own cross section. For a material that shapes its own flux it is not: a larger Li6(n,t) cross section in a breeder depresses its own flux, the same narrowing feedback as self-shielding, so holding the flux overstates that rate's sigma. Nor do the cross sections of the materials the neutrons passed through (a shield, a multiplier, a breeder) change the flux, which understates the sigma of a product behind many mean free paths of steel. The spectrum's own uncertainty is only what the pulse is given. yamc's `Model.simulate_transmutation` carries both: with `cross_sections` on, its one transport gives every history a likelihood-ratio weight per replica, so each replica's rates, isomeric partials, fission-yield weights and flux are what a transport with that replica's data would tally, every material's cross sections included (`flux_response` is `"replica_weights"` in the report, and `transport_replicas` the count). The count is fixed before the transport, `samples` or 256 at the default `convergence`, and the Monte Carlo noise each replica's shift carries is measured from the even half of the histories and removed from every sigma (`noise_removed`, `noisy`). Only the independent method supports `data_uncertainty`. - **Decay photon spectrum covariance and continuum shape.** MT=457 can state a covariance between a spectrum's lines (LCOV), and no library ships one. A continuum's shape is held, and only its normalisation is drawn. The decay data also leave unstated whether a spectrum's line intensities share a part (ENDF/B-VIII.1 folds each spectrum's normalisation sigma into every line's), whether a normalisation and its lines are correlated, and whether a nuclide's gamma and x-

ray spectra are. None of these is assumed: every photon output, and the gamma part of decay heat, is evaluated with all of them taken as zero (`Estimate.std_dev`) and as one (`Estimate.std_dev_correlated`), which bounds every non-negative correlation, and `decay_photon_spectra_folded` names the spectra written the ENDF/B way, where most of that range comes from. The decay data give a sigma on 99.6% of ENDF/B-VIII.1 gamma lines, and for one emitter as a trace in iron the contact dose from its line data alone has a relative sigma of 0.014% for Co60, 1.09 to 1.75% for Mn56 and 0.82 to 1.70% for W187 on ENDF/B-VIII.1, from the independent end to the fully correlated one. - **Dose constants and build-up.** Contact dose uses the same photon attenuation coefficients, the same response (air energy absorption, or the ICRP-116 coefficients for effective dose) and the same constant build-up factor in every replica. None of these tables gives a per-value uncertainty. The build-up factor is a modelling choice with a larger error: the default of 2 is 16 to 17% high for Co60 in steel compared with a photon transport calculation of the same half-space. - **Material composition and natural abundances.** Element and impurity fractions, density and natural isotopic abundances are the same in every replica. For activation driven by a trace impurity this is often the largest missing term: in a 316L-like steel with 0.1 wt% cobalt, the contact dose at 10 years is 99.6% Co60 and changes by 0.69% per 1% change in the cobalt fraction. A material contains only the elements it is given, and the bundled PNNL compendium's `Steel, Stainless 316L` has no cobalt, niobium, tantalum or silver. - **Displacement damage beyond the starting composition's reaction deposits.** MT=444 has no covariance of its own, so a replica moves it through the drawn cross sections of the reactions that deposit it. The share no reaction's deposit reproduces (fission's above all), the MT=444 of transmutation products, short-range (LB=8) covariance and the displacement threshold energies E_d are held at nominal. The NRT model itself is a model choice with no uncertainty attached. `get_data_uncertainty_info(...)["damage_energy"]` reports the share explained per starting nuclide.

`not_perturbed` in `get_data_uncertainty_info` lists these on every run: isomeric branching (MF=9/MF=10) of a reaction with no stated MF=40 covariance, the MT=5 product multiplicities (MF=6 MT=5), cross-material blocks naming an evaluation the run does not hold, non-cross-section and underivable NC MF=33 blocks, lumped MF=33 blocks stated in barns, resonance-parameter covariance (MF=32) neither sampled nor in `covariance.arrow`, the decay photon spectrum covariance and continuum shape, the photon attenuation, energy-absorption and fluence-to-dose coefficients, the contact-dose build-up factor, and the material composition, density, natural abundances and atomic masses. It also adds an entry for each source a run turns off (half-life, decay energy, decay photon lines, fission yield, decay branching, isomeric branching, activation cross section), the per-branch decay emission of a parent whose branching was sampled, the flux spectrum on `Material.transmute` when that source is off or some or all spectra have no sigma, the tallied-rate statistics when a transport run does not sample them, and the self-shielding correction when shielding is on. The MT=5 multiplicities are shares of the MT=5 rate: the rate is drawn with the cross sections, and its split among the residuals and light particles is held.

Some held inputs on this page have no entry of their own: MF=33 blocks on partial levels (MT=600-849, 875-891), for which the chain has no rate, so they are neither listed nor counted; the cross-section covariance of the overlay's `(n,n')` channels; the fission-yield weights held at the nominal spectrum

under a flux draw; and, on `Material.transmute`, the response of the flux to the cross sections of the materials the neutrons passed through.

`has_gaps` only considers the sources the run perturbs. It is true when any of these occurs:

- a nuclide with no usable MF=33 block;
- a skipped cross-material, non-cross-section or NC block;
- a mirrored pair whose copies disagree;
- an unassignable lump;
- a block with an unsupported or malformed layout;
- a channel whose partial rates add up to more or less than its rate (`partials_above_rate`, `partials_below_rate`);
- a derived channel whose opposing terms have no stated covariance (`derived_opposing_uncorrelated`);
- a covariance repaired beyond round-off on a channel a draw can move, inside or outside the populated bound;
- a cross-material covariance drawn at the closest admissible one (`cross_material_adjusted`);
- a perturbed evaluation its own library documents a problem with (`covariance_warnings`);
- a spectrum with no flux sigma, or a rate under a spectrum with one that the flux draw cannot move (`flux_rates_without_terms`);
- a reachable unstable nuclide with no half-life sigma, or with a decay energy or decay photon data but no sigma on it;
- a half-life, decay-energy or decay photon sigma that cannot be sampled;
- a fissioning parent with no yield sigma, a yield sigma that cannot be sampled, or yields that do not map back (`fission_yields_mapping_mismatch`);
- a multi-mode parent whose branching was held, wholly or in part, or drawn short of its stated sigmas;
- a reachable isomeric split with no MF=40 to draw, an MF=40 block not used, or an MF=40 matrix repaired to the nearest correlation matrix.

Of the inputs in the not-propagated list, the skipped and unassignable MF=33 blocks set `has_gaps`, and the partial-level blocks set it only through `no_covariance_data`, when they are all a nuclide has. A reported sigma is the spread from the propagated sources, with everything in this list held fixed. See Nuclear-data uncertainty.

Reproducibility

These three properties are checked by tests:

- **Thread count does not change the result.** A 1-thread and a 7-thread run agree to the last bit, including the per-replica ensemble, the sigmas and the truncation counters.
- **A seed reproduces a run.** See the seeding rules above.
- **Optimisations do not change the result.** Every performance change to the collapse is checked to be bit-identical to the previous implementation, by hex-float comparison of inventories and against a reference of the per-group terms, one-group rates and fission-yield weights.

Without `data_uncertainty`, no covariance is read from disk, no matrix is folded or factorised, and no replica is solved. The means are bit-identical to a build without uncertainty support.

Validation

YANI is run against every open, reproducible benchmark we have found, and the results, the scripts that produced them and a figure for every single case are published separately:

The verification and validation site, and the repository it is built from.

It is a separate repository because it is a different kind of thing. It holds gigabytes of result JSON and hundreds of figures, it takes hours of solving to regenerate, and it should be runnable by someone who has never cloned this one. Everything there reproduces from a clean checkout with the benchmark data committed, so nothing is fetched from the IAEA to check a number.

What is covered

benchmark	what is measured	cases	data behind another code's published results
FNS decay heat	specific decay heat after a 14 MeV irradiation	132	TENDL-2017
Effective cross sections	spectrum-averaged cross sections in 23 neutron fields	365	TENDL-2017

FNS decay heat. JAEA's Fusion Neutronics Source held a 1 g foil in a 14 MeV field of about 1.1e10 n/cm²/s, withdrew it, and measured the heat its activation products gave off as it cooled. 73 foils across three campaigns: two five minute irradiations followed for about an hour, and one 7.6 hour irradiation followed out to 400 days.

Effective cross sections. A foil is irradiated in a known field and the product counted, giving the production rate per target atom per unit flux. That is one fold of a cross section against a spectrum with nothing else in the way, which makes it the most direct available test of the step YANI takes before any solving happens. 89 of the 365 reactions measure an isomeric state directly, so for those the branching fraction is not a correction to the answer, it is the answer. See isomeric branching.

Both come from the IAEA's CoNDERC collection. What makes these archives worth building on is that each carries three things rather than one: the measurement with its experimental uncertainty, the conditions needed to reproduce it (the neutron field as group fluxes, and for the decay heat cases the foil composition and mass and the schedule), and another code's calculated results for the same cases. So a disagreement can be attributed rather than merely noted.

How to read a comparison there

The like-for-like line is a verification result. The other code's published results used TENDL-2017 cross sections and UKAEA's `decay_2012` decay library, so that is what YANI is run with on that line. Two independently written codes given identical evaluations should land on the same answer, and when they do, that is evidence both implementations are right. A large win on identical data would be evidence that one of them is wrong, and it should be read that way rather than celebrated.

The other lines are a different question. YANI can be pointed at TENDL-2025, JEFF-4.0, ENDF/B-VIII.1, FENDL-3.2d or a mixture assembled subsection by subsection, and where one of those tracks the measurement better, that is a statement about data rather than about code. Both are worth knowing, they are not the same claim, and every page keeps them apart.

Where a deviation is shared, it belongs to neither code. Several foils sit tens or hundreds of percent from the measurement in both codes at once. That is the evaluation, and no amount of work on either code will move it.

What is not covered

Stated here so it cannot be mistaken for coverage that exists.

- **Every field in both benchmarks is a DT field**, at JAEA-FNS, FNG-Frascati or TU Dresden. There is no fission-spectrum or thermal validation on the site yet.
- **No foil in either set is fissionable**, so the fission-yield path these benchmarks exercise is none of it, however the yield settings are configured.
- **Self-shielding** is compared against measurement on a single case.
- **Displacement damage (dpa)** is not compared against measurement or against another code on the site yet.
- Coverage of the nuclear-data uncertainty machinery is reported per run by `get_data_uncertainty_info`, and the site's methodology page states what the benchmark runs there did and did not propagate. See Nuclear-data uncertainty.

If you know of an open, reproducible benchmark that is not covered, raise an issue and it will be added with a reproducible script.

Reproducing it

```
git clone https://github.com/fusion-neutronics/yani-verification-and-validation
cd yani-verification-and-validation
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt

python benchmarks/fns_decay_heat/run.py --case Fe
```

`python tools/run_all.py --plan` says what a full run would do and roughly how long it would take, and checks that YANI is installed, the benchmark data is present and the disk can hold the nuclear data a full run caches, before running any of it.

API reference

The reference below is generated by mkdocstrings from the type stubs shipped inside the installed `yani` wheel. Classes appear under their public names (`yani.Material`, `yani.PulseSchedule`, ...) rather than the private `yani._core` extension module they live in.

The members are grouped by what they are for. The `get_*` / `set_*` accessors behind nuclear-data settings are documented as the settings themselves under Nuclear data settings. Public submodules are listed below under the names used in examples, such as `yani.sources.Histogram` and `yani.shapes.FoilLump`.

Materials

---

class yani.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 yani
>>> water = yani.Material(composition={"H2O": 1.0}, density=1.0)
>>> iron = yani.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 `yani.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,`	Thermal scattering tables for bound nuclides, by name, such as	`None`

Name	Type	Description	Default
	`float] | None`	`"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:

Name	Type	Description
`Material`	`Material`	A new material.

Source code in `yani/_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 ``yani.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 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 `yani/_core/__init__.pyi`

```
1326    @staticmethod
1327    def from_atom_densities(atom_densities: dict, *, name:
1328    typing.Optional[builtins.str] = None, id: typing.Optional[builtins.int] = None,
1329    transmutable: builtins.bool = False, volume: typing.Optional[builtins.float] =
1330    None, temperature: typing.Optional[builtins.float] = None) -> Material:
1331        r"""
1332        Build a material directly from absolute per-nuclide atom densities.
1333
1334        Stores the given atoms/barn-cm values verbatim (no normalize-then-recombine
1335        round-trip), so callers that already hold raw number densities -- e.g. the
1336        transmutation stepper -- keep full precision. ``density`` then reports the
1337        sum of the values. This replaces the old ``Material(comp, density="sum")``.
1338
1339        Args:
1340            atom_densities (dict[str, float]): nuclide name -> atoms/barn-cm.
1341            name (Optional[str]): Name for the material.
1342            id (Optional[int]): ID for the material. Defaults to a fresh one,
1343                which no earlier material was given.
1344            transmutable (bool): Mark this material for depletion by
1345                ``Model.simulate_transmutation``. Not needed for
1346                ``Material.transmute()``, which depletes whatever it is called on.
1347            volume (Optional[float]): Material volume in cm³.
1348            temperature (Optional[float]): Temperature in Kelvin. If omitted,
1349                294 K is used, or the only temperature the nuclear data carries
1350                when it carries just one.
1351
            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 `yani/_core/__init__.pyi`

```
1352    def get_nuclide_names(self) -> builtins.list[builtins.str]:
1353        r"""
1354        Return a list of nuclide names currently present in the material.
1355        """
```

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 `yani/_core/__init__.pyi`

```
1369  def read_nuclear_data(self, nuclide_path_map: typing.Optional[typing.Any] = None,
1370  photon_data: typing.Optional[typing.Mapping[builtins.str, builtins.str]] = None) ->
1371  None:
1372      r"""
1373      Bulk load nuclear data from a mapping of nuclide -> file path or a keyword
1374  string.
1375
1376      Args:
1377          nuclide_path_map (Optional[Dict[str,str] | str]): Mapping of nuclide names
1378  to file paths or a keyword string.
1379          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 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 `yani/_core/__init__.pyi`

```
1380  def calculate_microscopic_xs_neutron(self, mt_filter:
1381  typing.Optional[typing.Sequence[builtins.int]] = None) ->
1382  builtins.dict[builtins.str, builtins.dict[builtins.int,
1383  builtins.list[builtins.float]]]:
1384      r"""
1385      Calculate microscopic neutron cross sections on the unified energy grid.
1386
1387      Args:
1388          mt_filter (Optional[List[int]]): Restrict to these MT numbers.
1389
          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 `yani/_core/__init__.pyi`

```
1390  def get_atoms_per_barn_cm(self) -> builtins.dict[builtins.str, builtins.float]:
1391      r"""
1392      Number density (atoms / barn-cm) per nuclide.
1393
1394      Returns:
1395          Dict[str, float]: nuclide -> atoms / b-cm
1396      """
```

`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 `yani/_core/__init__.pyi`

```
1397  def activity(self, *, by_nuclide: builtins.bool = False, per:
1398  typing.Optional[builtins.str] = None) -> typing.Any:
1399      r"""
1400      Calculate the radioactive activity of the current material inventory.
1401
1402      The material must have a ``volume`` in cm³. Atom densities are converted
1403      to total atoms using ``atoms/barn-cm * 1e24 * volume``. The decay data
1404      (``transmutation_decay_data``, which has no default) provides
1405      half-lives; each nuclide's activity is ``N * ln(2) / half_life``.
1406      Stable nuclides are omitted.
1407
1408      Args:
1409          by_nuclide (bool): Return a ``dict[str, float]`` by nuclide instead
1410              of the total.
1411          per (str | None): ``None`` (default) for the total, which needs
1412              ``volume``; ``"cm3"`` for Bq/cm³, which needs nothing; ``"g"``
1413              for Bq/g, which needs only ``density``.
1414
1415      Returns:
1416          float | dict[str, float]: Activity, in Bq when ``per`` is ``None``,
1417          Bq/cm³ when it is ``"cm3"`` and Bq/g when it is ``"g"``. The unit of
1418          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 `yani/_core/__init__.pyi`

```
def decay_heat(self, *, by_nuclide: builtins.bool = False, per:
typing.Optional[builtins.str] = None, component: typing.Optional[builtins.str] =
None) -> typing.Any:
    r"""
    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.

    Args:
        by_nuclide (bool): Return a ``dict[str, float]`` by nuclide instead
            of the total.
        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``.
        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.

    Returns:
        float | dict[str, float]: Decay heat, in W when ``per`` is ``None``,
        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 `yani/_core/__init__.pyi`

```
def decay_photon_spectrum(self, *, per: typing.Optional[builtins.str] = None) ->
tuple[builtins.list[builtins.float], builtins.list[builtins.float]]:
    r"""
    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:
        tuple[list[float], list[float]]: Line energies (eV) and their
        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 `yani/_core/__init__.pyi`

```
1482  def decay_photon_continua(self, *, per: typing.Optional[builtins.str] = None) ->
1483  builtins.list[PhotonContinuum]:
1484      r"""
1485      The decay photon continua of the current inventory, one per nuclide
1486      and continuum, in nuclide-name order.
1487
1488      The part of a decay spectrum ENDF gives as a density over energy rather
1489      than as lines, which ``decay_photon_spectrum()`` does not include. Each
1490      continuum keeps its own energy grid and interpolation law, and its
1491      rates are photons/s/eV (or per cm³ or per g, following ``per``), so
1492      they are not line rates; ``emission_rate`` is the integral. A material
1493      whose nuclides have no continuum returns an empty list.
1494
1495      Args:
1496          per (str | None): None for the whole material, which needs a
1497              ``volume``, or ``'cm3'`` or ``'g'``, as for
1498              ``decay_photon_spectrum()``.
1499
1500      Returns:
1501          list[PhotonContinuum]: The continua, each with its nuclide,
1502          energies, rates, interpolation law and emission rate.
1503
1504      Examples:
1505          >>> for continuum in activated.decay_photon_continua():
1506          ...     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 `yani/_core/__init__.pyi`

```
def contact_dose(self, *, dose_quantity: builtins.str = 'absorbed-air', build_up:
builtins.float = 2.0, by_nuclide: builtins.bool = False) -> typing.Any:
    r"""
    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.

    Args:
        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.
        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.
        by_nuclide (bool): Return a ``dict[str, float]`` by nuclide instead
            of the total. Nuclides that contribute nothing are left out.

    Returns:
        float | dict[str, float]: Contact dose rate in Gy/h
        (```'absorbed-air'```) or Sv/h (```'effective'```).

    Raises:
        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...
    """
```

`meth` clearance_index

```
clearance_index(limit_set: str, *, metal: bool = False, apply_default_limit: bool
= True, exclude_daughters: bool = True) -> ClearanceResult
```

Clearance index of the current inventory against one regulatory limit set.

Each radionuclide's activity is divided by its limit in the set and the ratios are summed; the material meets the limits when the sum is below the set's threshold, normally 1. The limit sets cover UK, German, US, EU and IAEA clearance, exemption and disposal tables, and come from the `radiological-material-clearance-finder` crate, whose documentation describes each one and where its numbers come from.

Half-lives come from the configured transmutation chain, the same ones `activity()` and `decay_heat()` use, so a nuclide the chain calls stable contributes nothing. Specific activities (Bq/g) need nothing beyond the composition, and the volumetric US sets (Ci/m3) use the mass density the atom densities imply. Only the total activity sets (Bq) need `material.volume`.

Stable isotopes have to stay in the material: they are most of its mass, and so most of the Bq/g denominator.

Parameters:

Name	Type	Description	Default
`limit_set`	`str`	Name of the limit set, such as `"UK_EPR16_out_of_scope"` or `"StrlSchV_unrestricted"`. `clearance_indices()` assesses against every set at once.	*required*
`metal`	`bool`	Whether the material is activated metal, which changes some NRC limits and adds others.	`False`
`apply_default_limit`	`bool`	Whether to apply the set's catch-all limit to nuclides its table does not list. The UK regulations define one, so leaving this on is what they say.	`True`
`exclude_daughters`	`bool`	Whether to leave out daughters whose parent's limit already covers them, as the regulation's own table directs. Turning it off double counts them.	`True`

Returns:

Name	Type	Description
`ClearanceResult`	`ClearanceResult`	The index, whether it clears, each nuclide's
	`ClearanceResult`	contribution, and any activity no limit in the set covers.

Raises:

Type	Description
`KeyError`	If no limit set has that name.
`ValueError`	If the set is in Bq and the material has no volume.

Examples:

```
>>> result = material.clearance_index("UK_EPR16_out_of_scope")
>>> result.index
11.24...
>>> result.clearable
False
```

Source code in `yani/_core/__init__.pyi`

```
def clearance_index(self, limit_set: builtins.str, *, metal: builtins.bool = False,
apply_default_limit: builtins.bool = True, exclude_daughters: builtins.bool = True)
-> ClearanceResult:
    r"""
    Clearance index of the current inventory against one regulatory limit set.

    Each radionuclide's activity is divided by its limit in the set and the
    ratios are summed; the material meets the limits when the sum is below
    the set's threshold, normally 1. The limit sets cover UK, German, US, EU
    and IAEA clearance, exemption and disposal tables, and come from the
    ``radiological-material-clearance-finder`` crate, whose documentation
    describes each one and where its numbers come from.

    Half-lives come from the configured transmutation chain, the same ones
    ``activity()`` and ``decay_heat()`` use, so a nuclide the chain calls
    stable contributes nothing. Specific activities (Bq/g) need nothing
    beyond the composition, and the volumetric US sets (Ci/m3) use the mass
    density the atom densities imply. Only the total activity sets (Bq)
    need ``material.volume``.

    Stable isotopes have to stay in the material: they are most of its
    mass, and so most of the Bq/g denominator.

    Args:
        limit_set (str): Name of the limit set, such as
            ``"UK_EPR16_out_of_scope"`` or ``"StrlSchV_unrestricted"``.
            ``clearance_indices()`` assesses against every set at once.
        metal (bool): Whether the material is activated metal, which
            changes some NRC limits and adds others.
        apply_default_limit (bool): Whether to apply the set's catch-all
            limit to nuclides its table does not list. The UK regulations
            define one, so leaving this on is what they say.
        exclude_daughters (bool): Whether to leave out daughters whose
            parent's limit already covers them, as the regulation's own
            table directs. Turning it off double counts them.

    Returns:
        ClearanceResult: The index, whether it clears, each nuclide's
        contribution, and any activity no limit in the set covers.

    Raises:
        KeyError: If no limit set has that name.
        ValueError: If the set is in Bq and the material has no volume.

    Examples:
        >>> result = material.clearance_index("UK_EPR16_out_of_scope")
        >>> result.index
        11.24...
        >>> result.clearable
        False
    """
```

`meth` clearance_indices

```
clearance_indices(limit_sets: Optional[Sequence[str]] = None, *, metal: bool =
False, apply_default_limit: bool = True, exclude_daughters: bool = True) ->
dict[str, ClearanceResult]
```

Clearance indexes of the current inventory against many limit sets.

The same assessment as `clearance_index()`, once per set. Sets the material cannot be assessed against are skipped rather than raising, which in practice means the total activity sets (Bq) when the material has no `volume`.

Parameters:

Name	Type	Description	Default
`limit_sets`	`list[str] | None`	Names of the limit sets. `None` (the default) assesses against every available set, which is also the way to see their names.	`None`
`metal`	`bool`	As for `clearance_index()`.	`False`
`apply_default_limit`	`bool`	As for `clearance_index()`.	`True`
`exclude_daughters`	`bool`	As for `clearance_index()`.	`True`

Returns:

Type	Description
`dict[str, ClearanceResult]`	dict[str, ClearanceResult]: Results keyed by limit set name.

Raises:

Type	Description
`KeyError`	If a named limit set does not exist.

Examples:

```
>>> results = material.clearance_indices()
>>> [name for name, r in results.items() if r.clearable]
['Fetter', 'NRC_long', ...]
```

Source code in `yani/_core/__init__.pyi`

```
1621  def clearance_indices(self, limit_sets:
1622  typing.Optional[typing.Sequence[builtins.str]] = None, *, metal: builtins.bool =
1623  False, apply_default_limit: builtins.bool = True, exclude_daughters: builtins.bool
1624  = True) -> builtins.dict[builtins.str, ClearanceResult]:
1625      r"""
1626      Clearance indexes of the current inventory against many limit sets.
1627
1628      The same assessment as ``clearance_index()``, once per set. Sets the
1629      material cannot be assessed against are skipped rather than raising,
1630      which in practice means the total activity sets (Bq) when the material
1631      has no ``volume``.
1632
1633      Args:
1634          limit_sets (list[str] | None): Names of the limit sets. ``None``
1635              (the default) assesses against every available set, which is
1636              also the way to see their names.
1637          metal (bool): As for ``clearance_index()``.
1638          apply_default_limit (bool): As for ``clearance_index()``.
1639          exclude_daughters (bool): As for ``clearance_index()``.
1640
1641      Returns:
1642          dict[str, ClearanceResult]: Results keyed by limit set name.
1643
1644      Raises:
1645          KeyError: If a named limit set does not exist.
1646
1647      Examples:
1648          >>> results = material.clearance_indices()
          >>> [name for name, r in results.items() if r.clearable]
          ['Fetter', 'NRC_long', ...]
      """
```

`meth` macroscopic_cross_section

```
macroscopic_cross_section(reaction: int | str, temperature: Optional[str] = None)
-> tuple[list[float], list[float]]
```

Calculate macroscopic cross section for a specific reaction.

This method accepts either an integer MT number or a string reaction name (like "(n,gamma)", "fission", etc.) and returns the macroscopic cross section for that reaction.

Parameters:

Name	Type	Description	Default
`reaction`	`Union[int, str]`	Either an MT number or reaction name.	*required*
`temperature`	`Optional[str]`	Temperature key (e.g., "294"). If not provided, uses material.temperature, else the only available temperature if there is just one, else 294 K.	`None`

Returns:

Type	Description
`tuple[list[float], list[float]]`	Tuple[List[float], List[float]]: (cross_section_values, energy_grid)

Source code in `yani/_core/__init__.pyi`

```
1649  def macroscopic_cross_section(self, reaction: builtins.int | builtins.str,
1650  temperature: typing.Optional[builtins.str] = None) ->
1651  tuple[builtins.list[builtins.float], builtins.list[builtins.float]]:
1652      r"""
1653      Calculate macroscopic cross section for a specific reaction.
1654
1655      This method accepts either an integer MT number or a string reaction name
1656      (like "(n,gamma)", "fission", etc.) and returns the macroscopic cross section
1657      for that reaction.
1658
1659      Args:
1660          reaction (Union[int, str]): Either an MT number or reaction name.
1661          temperature (Optional[str]): Temperature key (e.g., "294"). If not
1662  provided,
1663              uses material.temperature, else the only available temperature if
1664              there is just one, else 294 K.
1665
          Returns:
              Tuple[List[float], List[float]]: (cross_section_values, energy_grid)
          """
```

`meth` transmute

```
transmute(schedule: Any, data_uncertainty: Optional[DataUncertainty] = None,
self_shielding_chord: Optional[float] = None, self_shielding_shape: Optional[Any]
= None, displacement_damage: bool = False, displacement_energies:
Optional[Mapping[str, float]] = None, product_threshold: float = 1e-12) ->
TransmutationResults
```

Transmute this material over an irradiation/cooling schedule, without re-running transport.

Each irradiation `Pulse` carries a `NeutronSource` whose energy is a `Histogram` (the multigroup spectrum); the pulse `rate` is the total flux magnitude [n/cm²/s]. The spectrum collapses cross sections into one-group reaction rates, scaled by `rate`. A `Cooldown` is a decay-only step. Different pulses may carry different spectra (e.g. a DD and a DT campaign).

Parameters:

Name	Type	Description	Default
`schedule`	`PulseSchedule`	The irradiation/cooling timeline. Every irradiation `Pulse` must carry a `NeutronSource` whose energy is a `Histogram`; its `rate` is the flux magnitude [n/cm²/s]. The source's position/direction are ignored (no transport runs). Use `Cooldown` for decay-only steps.	*required*

The transmutation network is assembled from the configured per-subsection sources (`yani.transmutation_decay_data` / `_reactions` / `_fission_yields`); decay-only schedules need only the decay source. There is no default library: a source the schedule reads that is unset raises `ValueError` before any work, naming each such setting. `_reactions` and `_fission_yields` can be set to `False` instead, for a material with no reaction rates or one in which nothing fissions.

The material does **not** need `transmutable=True`. That flag exists so `Model.simulate_transmutation` can pick which materials in a geometry to deplete; here you have already named the one you mean.

The cross sections this needs are loaded **into the material** and kept, so calling it again on the same material does no file reading at all. Only the composition's and those of the products the schedule populates are loaded (see `product_threshold`), not every nuclide the chain can reach. The composition is not modified. A sweep over thousands of *distinct* compositions should call `release_nuclear_data()` on each material when it is done with it, since otherwise every one of them holds its nuclides' data for as long as it lives.

Returns:

Name	Type	Description
`TransmutationResults`	`TransmutationResults`	The same object `Model.simulate_transmutation` returns, keyed by this material's `id`. `get_material(id, step)` takes step 0 as the initial composition and step `i` as the state after schedule step `i - 1`; `get_reaction_rates(id, step)` indexes the schedule steps directly, so its step 0 is the first step.
`data_uncertainty`	`(DataUncertainty, optional)`	Ask for nuclear-data uncertainty on the result. Each source `DataUncertainty` names is sampled where it applies (`statistical` needs a transport run, `flux_spectrum` a supplied flux sigma), and the schedule is re-solved until every tracked standard deviation is known to `DataUncertainty.convergence` (5% by default). Omit it (the default) and nothing is read, folded or sampled: the inventories are bit-identical either way. Read the sigmas with `get_nuclide_uncertainty`, and what was and was not covered with `get_data_uncertainty_info(id)`.
`self_shielding_chord`	`(float, optional)`	Mean chord length `4V/S` of this material's lump, in cm, which is twice the thickness for a thin slab and `4R/3` for a sphere. Given one, the flux inside each energy group is depressed where the total cross section is large, and the reaction rates come back shielded. Omit it (the default) and nothing is shielded: the rates are bit-identical to a run without it, and no shape is inferred from a geometry this material does not have. Read what was done with

Name	Type	Description
		`get_self_shielding_info(id)`. 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: 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 this gives C/E 0.80, no correction gives 0.86 and narrow resonance gives 0.46).
`self_shielding_shape`	`(SphereLump | CubeLump | FoilLump | CylinderLump | WireLump, optional)`	The lump's form, from `shapes`, as an alternative to stating the chord yourself. The chord is `4V/S`, so the shape decides how much surface a volume hides behind: `SphereLump()` and `CubeLump()` are fixed by `Material(volume=...)` and take no arguments, while `FoilLump(thickness=)`, `CylinderLump(radius=)` and `WireLump(radius=)` carry the one dimension a volume cannot imply. Give this or `self_shielding_chord`, not both. 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. It is an upper bound rather than a safe default, which is why there is no default here at all.
`displacement_damage`	`bool`	Also compute displacement damage over the schedule: the damage energy deposited per atom (eV) and NRT displacements per atom, per element and for the material, read with `get_dpa`,

Name	Type	Description
		`get_damage_energy` and `get_displacement_damage_info`. Each nuclide's MT=444 damage-energy cross section is folded against the pulse spectrum by the same collapse as the reaction rates (self-shielding included), using each step's composition and flux; cooldowns add nothing. dpa is `0.8 * E_damage / (2 * E_d)` per element, and the material total weights the elements by atom fraction. Off by default, and then MT=444 is not fetched and nothing about the solve changes. No uncertainty is given for it yet.
`displacement_energies`	`(dict[str, float], optional)`	Displacement threshold energies `E_d` in eV by element symbol, e.g. `{"Fe": 40.0}`, replacing the defaults (ASTM E521, and the OECD-NEA 2015 report "Primary Radiation Damage in Materials" for elements it does not cover). An element of the material in neither source must be given here; it is never guessed. Needs `displacement_damage=True`.
`product_threshold`	`float`	The atom fraction, or share of the activity, decay heat or decay photon power, a transmutation product must reach at some state of the schedule for its cross sections to be downloaded and its own neutron reactions solved. Default 1e-12. The schedule is solved with the composition's cross sections, every product crossing this is loaded, and the solve is repeated until nothing new crosses it, so a run fetches the few dozen nuclides an irradiation populates rather than the hundreds its chain can reach. A product below it is still produced, decays

Name	Type	Description
		and is reported; only its reactions are dropped, which moves its own atom fraction and decay-quantity shares by less than the threshold and what it would have made by less than the summed peak fraction of the products left out. What was loaded and left out is in `get_product_screening_info(id)`. 0 loads every product the solve populates, which is exact.

Raises:

Type	Description
`ValueError`	If an irradiation pulse lacks a NeutronSource, its energy is not a Histogram, a transmutation source the schedule reads is unset, or on other invalid inputs.

Examples:

```
>>> spectrum = yani.NeutronSource(energy=yani.sources.Histogram("CCFE-709",
flux_709))
>>> sched = yani.PulseSchedule([
...     yani.Pulse(rate=1e14, duration=(1, "h"), source=spectrum),
...     yani.Cooldown(duration=(7, "d")),
... ])
>>> results = iron.transmute(schedule=sched)
>>> final = results.get_final_material(iron.id)
>>> # the isomeric split of the dominant channel, as two edges of one
>>> # reaction: the sum is sigma * phi * N and the ratio is f_m
>>> results.get_reaction_rates(iron.id, 0)["W186"]["(n,2n)"]
[('W185', 1.1e-08), ('W185_m1', 4.6e-07)]
```

Source code in `yani/_core/__init__.pyi`

```
def transmute(self, schedule: typing.Any, data_uncertainty:
typing.Optional[DataUncertainty] = None, self_shielding_chord:
typing.Optional[builtins.float] = None, self_shielding_shape:
typing.Optional[typing.Any] = None, displacement_damage: builtins.bool = False,
displacement_energies: typing.Optional[typing.Mapping[builtins.str,
builtins.float]] = None, product_threshold: builtins.float = 1e-12) ->
TransmutationResults:
    r"""
    Transmute this material over an irradiation/cooling schedule, without
    re-running transport.

    Each irradiation ``Pulse`` carries a ``NeutronSource`` whose energy is a
    ``Histogram`` (the multigroup spectrum); the pulse ``rate`` is the total
    flux magnitude [n/cm²/s]. The spectrum collapses cross sections into
    one-group reaction rates, scaled by ``rate``. A ``Cooldown`` is a
    decay-only step. Different pulses may carry different spectra (e.g. a DD
    and a DT campaign).

    Args:
        schedule (PulseSchedule): The irradiation/cooling timeline. Every
            irradiation ``Pulse`` must carry a ``NeutronSource`` whose energy
            is a ``Histogram``; its ``rate`` is the flux magnitude [n/cm²/s].
            The source's position/direction are ignored (no transport runs).
            Use ``Cooldown`` for decay-only steps.

    The transmutation network is assembled from the configured per-subsection
    sources (``yani.transmutation_decay_data`` / ``_reactions`` /
    ``_fission_yields``); decay-only schedules need only the decay source.
    There is no default library: a source the schedule reads that is unset
    raises ``ValueError`` before any work, naming each such setting.
    ``_reactions`` and ``_fission_yields`` can be set to ``False`` instead,
    for a material with no reaction rates or one in which nothing fissions.

    The material does **not** need ``transmutable=True``. That flag exists so
    ``Model.simulate_transmutation`` can pick which materials in a geometry to
    deplete; here you have already named the one you mean.

    The cross sections this needs are loaded **into the material** and kept,
    so calling it again on the same material does no file reading at all.
    Only the composition's and those of the products the schedule populates
    are loaded (see ``product_threshold``), not every nuclide the chain can
    reach. The composition is not modified. A sweep over thousands of
    *distinct* compositions should call ``release_nuclear_data()`` on each
    material when it is done with it, since otherwise every one of them
    holds its nuclides' data for as long as it lives.

    Returns:
        TransmutationResults: The same object ``Model.simulate_transmutation``
            returns, keyed by this material's ``id``.
            ``get_material(id, step)`` takes step 0 as the initial
            composition and step ``i`` as the state after schedule step
            ``i - 1``; ``get_reaction_rates(id, step)`` indexes the schedule
            steps directly, so its step 0 is the first step.

        data_uncertainty (DataUncertainty, optional): Ask for nuclear-data
            uncertainty on the result. Each source ``DataUncertainty`` names
            is sampled where it applies (``statistical`` needs a transport
            run, ``flux_spectrum`` a supplied flux sigma), and the schedule
            is re-solved until every tracked standard deviation is known to
            ``DataUncertainty.convergence`` (5% by default).
            Omit it (the default) and nothing is read, folded or sampled:
            the inventories are bit-identical either way. Read the sigmas
            with ``get_nuclide_uncertainty``, and what was and was not
            covered with ``get_data_uncertainty_info(id)``.
```

```
1730
1731            self_shielding_chord (float, optional): Mean chord length ``4V/S`` of
1732                this material's lump, in cm, which is twice the thickness for a
1733                thin slab and ``4R/3`` for a sphere. Given one, the flux inside
1734                each energy group is depressed where the total cross section is
1735                large, and the reaction rates come back shielded. Omit it (the
1736                default) and nothing is shielded: the rates are bit-identical to
1737                a run without it, and no shape is inferred from a geometry this
1738                material does not have. Read what was done with
1739                ``get_self_shielding_info(id)``.
1740
1741                The flux inside the lump comes from a slowing-down solve, which
1742                assumes nothing about resonances being narrow. The cheaper
1743                narrow-resonance approximation is deliberately not offered: it
1744                over-shields strong elastic scatterers badly enough to be worse
1745                than applying no correction at all (W186(n,gamma) in FNG-tung
1746                measures 1.29 b, where this gives C/E 0.80, no correction gives
1747                0.86 and narrow resonance gives 0.46).
1748
1749            self_shielding_shape (SphereLump | CubeLump | FoilLump | CylinderLump |
1750    WireLump, optional):
1751                The lump's form, from ``shapes``, as an alternative to stating the
1752                chord yourself. The chord is ``4V/S``, so the shape decides how
1753                much surface a volume hides behind: ``SphereLump()`` and ``CubeLump()``
1754                are fixed by ``Material(volume=...)`` and take no arguments,
1755                while ``FoilLump(thickness=)``, ``CylinderLump(radius=)``
1756                and ``WireLump(radius=)`` carry the one dimension a volume cannot
1757    imply.
1758                Give this or ``self_shielding_chord``, not both.
1759
1760                A sphere has the least surface for its volume, so it has the
1761                longest chord and shields more than any other shape of the same
1762                size. It is an upper bound rather than a safe default, which is
1763                why there is no default here at all.
1764
1765            displacement_damage (bool): Also compute displacement damage over the
1766                schedule: the damage energy deposited per atom (eV) and NRT
1767                displacements per atom, per element and for the material, read
1768                with ``get_dpa``, ``get_damage_energy`` and
1769                ``get_displacement_damage_info``. Each nuclide's MT=444
1770                damage-energy cross section is folded against the pulse spectrum
1771                by the same collapse as the reaction rates (self-shielding
1772                included), using each step's composition and flux; cooldowns
1773                add nothing. dpa is ``0.8 * E_damage / (2 * E_d)`` per element,
1774                and the material total weights the elements by atom fraction.
1775                Off by default, and then MT=444 is not fetched and nothing about
1776                the solve changes. No uncertainty is given for it yet.
1777
1778            displacement_energies (dict[str, float], optional): Displacement
1779                threshold energies ``E_d`` in eV by element symbol, e.g.
1780                ``{"Fe": 40.0}``, replacing the defaults (ASTM E521, and the
1781                OECD-NEA 2015 report "Primary Radiation Damage in Materials"
1782                for elements it does not cover). An element of the material in
1783                neither source must be given here; it is never guessed. Needs
1784                ``displacement_damage=True``.
1785
1786            product_threshold (float): The atom fraction, or share of the
1787                activity, decay heat or decay photon power, a transmutation
1788                product must reach at some state of the schedule for its cross
1789                sections to be downloaded and its own neutron reactions solved.
1790                Default 1e-12. The schedule is solved with the composition's
1791                cross sections, every product crossing this is loaded, and the
1792                solve is repeated until nothing new crosses it, so a run fetches
1793                the few dozen nuclides an irradiation populates rather than the
```

```
            hundreds its chain can reach. A product below it is still
            produced, decays and is reported; only its reactions are
            dropped, which moves its own atom fraction and decay-quantity
            shares by less than the threshold and what it would have made
            by less than the summed peak fraction of the products left out.
            What was loaded and left out is in
            ``get_product_screening_info(id)``. 0 loads every product the
            solve populates, which is exact.

    Raises:
        ValueError: If an irradiation pulse lacks a NeutronSource, its energy
            is not a Histogram, a transmutation source the schedule reads is
            unset, or on other invalid inputs.

    Examples:
        >>> spectrum = yani.NeutronSource(energy=yani.sources.Histogram("CCFE-709",
flux_709))
        >>> sched = yani.PulseSchedule([
        ...     yani.Pulse(rate=1e14, duration=(1, "h"), source=spectrum),
        ...     yani.Cooldown(duration=(7, "d")),
        ... ])
        >>> results = iron.transmute(schedule=sched)
        >>> final = results.get_final_material(iron.id)
        >>> # the isomeric split of the dominant channel, as two edges of one
        >>> # reaction: the sum is sigma * phi * N and the ratio is f_m
        >>> results.get_reaction_rates(iron.id, 0)["W186"]["(n,2n)"]
        [('W185', 1.1e-08), ('W185_m1', 4.6e-07)]
    """
```

meth release_nuclear_data

```
release_nuclear_data() -> None
```

Drop the nuclear data this material has loaded.

`transmute()` loads the cross sections it needs into the material and leaves them there, so calling it again on the same material does no file reading at all. That is the composition and the products the irradiation populates, a few dozen nuclides for a pure metal and up to a couple of hundred for a steel, held for as long as the material is.

That is the right trade for a material you transmute more than once, and the wrong one for a sweep over thousands of *distinct* compositions -- so releasing is explicit rather than automatic. The material stays usable: the next call that needs the data loads it again.

Examples:

```
>>> for composition in many_compositions:
...     mat = yani.Material(composition=composition, density=7.9)
...     results = mat.transmute(schedule=sched)
...     mat.release_nuclear_data()
```

Source code in `yani/_core/__init__.pyi`

```
1813  def release_nuclear_data(self) -> None:
1814      r"""
1815      Drop the nuclear data this material has loaded.
1816
1817      ``transmute()`` loads the cross sections it needs into the material and
1818      leaves them there, so calling it again on the same material does no file
1819      reading at all. That is the composition and the products the
1820      irradiation populates, a few dozen nuclides for a pure metal and up to a
1821      couple of hundred for a steel, held for as long as the material is.
1822
1823      That is the right trade for a material you transmute more than once, and
1824      the wrong one for a sweep over thousands of *distinct* compositions --
1825      so releasing is explicit rather than automatic. The material stays
1826      usable: the next call that needs the data loads it again.
1827
1828      Examples:
1829          >>> for composition in many_compositions:
1830          ...     mat = yani.Material(composition=composition, density=7.9)
1831          ...     results = mat.transmute(schedule=sched)
1832          ...     mat.release_nuclear_data()
1833      """
```

meth average_molar_mass

```
average_molar_mass() -> float
```

Return the average molar mass of this material (g/mol).

Source code in `yani/_core/__init__.pyi`

```
1834  def average_molar_mass(self) -> builtins.float:
1835      r"""
1836      Return the average molar mass of this material (g/mol).
1837      """
```

meth get_mass_density

```
get_mass_density() -> float
```

Return the mass density in g/cm³ regardless of stored units.

Source code in `yani/_core/__init__.pyi`

```
1838  def get_mass_density(self) -> builtins.float:
1839      r"""
1840      Return the mass density in g/cm³ regardless of stored units.
1841      """
```

meth mix_materials classmethod

```
mix_materials(materials: Sequence[Material], fractions: Sequence[float],
fraction_type: str = 'atom', *, name: Optional[str] = None, id: Optional[int] =
None) -> Material
```

Create a new material by mixing multiple materials.

Parameters:

Name	Type	Description	Default
`materials`	`List[Material]`	Materials to mix.	*required*
`fractions`	`List[float]`	Mixing fractions (one per material).	*required*
`fraction_type`	`str`	"atom", "mass", or "volume".	`'atom'`
`name`	`Optional[str]`	Name for the new material. If omitted, one is built from the components' names and fractions, such as `"steel(0.9000)_unnamed(0.1000)"`.	`None`
`id`	`Optional[int]`	ID for the new material. Defaults to a fresh one, which no earlier material was given.	`None`

Returns:

Name	Type	Description
`Material`	`Material`	A new material with the mixed composition and density.

Source code in `yani/_core/__init__.pyi`

```
1842  @classmethod
1843  def mix_materials(cls, materials: typing.Sequence[Material], fractions:
1844  typing.Sequence[builtins.float], fraction_type: builtins.str = 'atom', *, name:
1845  typing.Optional[builtins.str] = None, id: typing.Optional[builtins.int] = None) ->
1846  Material:
1847      r"""
1848      Create a new material by mixing multiple materials.
1849
1850      Args:
1851          materials (List[Material]): Materials to mix.
1852          fractions (List[float]): Mixing fractions (one per material).
1853          fraction_type (str): "atom", "mass", or "volume".
1854          name (Optional[str]): Name for the new material. If omitted, one
1855              is built from the components' names and fractions, such as
1856              ``"steel(0.9000)_unnamed(0.1000)"``.
1857          id (Optional[int]): ID for the new material. Defaults to a fresh
1858              one, which no earlier material was given.
1859
          Returns:
              Material: A new material with the mixed composition and density.
          """
```

`class` yani.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 `yani/_core/__init__.pyi`

```
843  def __new__(cls, fraction: builtins.float, *, target: builtins.str, percent:
844  builtins.float, fraction_type: builtins.str = 'atom') -> Enriched:
845      r"""
846      Create an enrichment specification.
847
848      Args:
849          fraction (float): Fraction of this component in the material.
850          target (str): Target nuclide for enrichment (e.g. "Li6").
851          percent (float): Enrichment percentage of the target isotope.
852          fraction_type (str): "atom" for atom percent or "mass" for weight percent.
853
854      Returns:
855          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.

func yani.enriched

```
enriched(fraction: float, *, target: str, percent: float, fraction_type: str = 'atom')
-> Enriched
```

Create an enrichment specification for a Material composition entry.

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 `yani/_core/__init__.pyi`

```
5343  def enriched(fraction: builtins.float, *, target: builtins.str, percent:
5344  builtins.float, fraction_type: builtins.str = 'atom') -> Enriched:
5345      r"""
5346      Create an enrichment specification for a Material composition entry.
5347
5348      Args:
5349          fraction (float): Fraction of this component in the material.
5350          target (str): Target nuclide for enrichment (e.g. "Li6").
5351          percent (float): Enrichment percentage of the target isotope.
5352          fraction_type (str): "atom" for atom percent or "mass" for weight percent.
5353
5354      Returns:
5355          Enriched: An enrichment specification.
      """
```

func yani.transmute

```
transmute(materials: Sequence[Material], schedules: PulseSchedule |
Sequence[PulseSchedule], data_uncertainty: Optional[DataUncertainty] = None,
self_shielding_chord: Optional[float] = None, self_shielding_shape: SphereLump |
CubeLump | FoilLump | CylinderLump | WireLump | None = None, displacement_damage: bool
= False, displacement_energies: Optional[Mapping[str, float]] = None,
product_threshold: float = 1e-12, processing: Optional[Processing] = None) ->
TransmutationResults
```

Transmute several materials over one timeline in one call.

The plural of `Material.transmute`, for a mesh from a transport run, a component broken into regions, or a sweep over compositions. The answer for each material is exactly what `material.transmute(schedule)` gives it, and the result is one `TransmutationResults` keyed by each material's `id`, the same shape `Model.simulate_transmutation` returns.

What a Python loop would repeat is done once instead: the chain is read once, each nuclide's cross sections are decoded once and shared by every material that needs them, and a multigroup collapse runs once for every distinct combination of spectrum, composition, temperature and shielding. Cells of one steel that saw the same spectrum share a collapse, whatever their flux magnitudes. `results.collapse_reuse` says how many collapses were shared. The materials' solves then run in parallel.

Parameters:

Name	Type	Description	Default
`materials`	`list[Material]`	The materials, each with a distinct `id`, since the results are keyed by it. Every material has one from construction; two share one only when one was given the other's explicitly or is a copy of it, as a material a transmutation result returns is of its original. As with `Material.transmute`, the cross sections each needs are loaded into it and kept, and the composition is not modified.	*required*
`schedules`	`PulseSchedule | list[PulseSchedule]`	One schedule for every material, or one per material in the same order. Each material's schedule carries its own spectra (the pulse sources) and flux magnitudes (the pulse rates), which is how each cell of a mesh gets its own flux. They must share one timeline, the same durations and irradiation on the same steps, because the result has one series of times.	*required*
`data_uncertainty`	`DataUncertainty`	Nuclear-data uncertainty, applied to every material as `Material.transmute` applies it to one. The same seed perturbs a given evaluation the same way in every material.	`None`
`self_shielding_chord`	`float`	One chord length `4V/S` in cm, for every material. See `Material.transmute`.	`None`

Name	Type	Description	Default
`self_shielding_shape`	`SphereLump | CubeLump | FoilLump | CylinderLump | WireLump`	One lump shape for every material, turned into a chord through each material's own `volume`. Give this or `self_shielding_chord`, not both.	`None`
`displacement_damage`	`bool`	Also compute each material's displacement damage, damage energy per atom and NRT dpa. See `Material.transmute`. Off by default, and then nothing about the solve or the data it loads changes.	`False`
`displacement_energies`	`dict[str, float]`	Displacement threshold energies in eV by element symbol, replacing the defaults, for every material. Needs `displacement_damage=True`.	`None`
`product_threshold`	`float`	The atom fraction, or share of a decay quantity, a product must reach for its cross sections to be loaded and its reactions solved, applied to each material's own schedule. Default 1e-12. See `Material.transmute`.	`1e-12`
`processing`	`Processing`	Removals, transfers between materials and feeds. The materials a transfer links are solved together each step. A material it names that is not in `materials`, a storage tank say, is carried unirradiated and reported under its own id. Not yet combined with `data_uncertainty`.	`None`

Returns:

Name	Type	Description
`TransmutationResults`	`TransmutationResults`	Keyed by each material's `id`. Per material, `get_source_rates(id)` gives its flux magnitudes and `get_self_shielding_info(id)` its shielding report.

Raises:

Type	Description
`ValueError`	If two materials share an id, the timelines differ, the number of schedules does not match the number of materials, or any single material would fail `Material.transmute`, in which case the message names the material.
`TypeError`	If `schedules` is neither a PulseSchedule nor a list of them, or the same Material object is given twice.

Examples:

```
>>> schedules = [
...     yani.PulseSchedule([
...         yani.Pulse(rate=flux[i], duration=(1, "y"), source=spectra[i]),
...         yani.Cooldown(duration=(1, "d")),
...     ])
...     for i in range(len(cells))
... ]
>>> results = yani.transmute(cells, schedules)
>>> results.get_final_material(cells[3].id)
```

Source code in `yani/_core/__init__.pyi`

```
def transmute(materials: typing.Sequence[Material], schedules: PulseSchedule |
typing.Sequence[PulseSchedule], data_uncertainty: typing.Optional[DataUncertainty] =
None, self_shielding_chord: typing.Optional[builtins.float] = None,
self_shielding_shape: shapes.SphereLump | shapes.CubeLump | shapes.FoilLump |
shapes.CylinderLump | shapes.WireLump | None = None, displacement_damage: builtins.bool
= False, displacement_energies: typing.Optional[typing.Mapping[builtins.str,
builtins.float]] = None, product_threshold: builtins.float = 1e-12, processing:
typing.Optional[Processing] = None) -> TransmutationResults:
    r"""
    Transmute several materials over one timeline in one call.

    The plural of ``Material.transmute``, for a mesh from a transport run, a
    component broken into regions, or a sweep over compositions. The answer for
    each material is exactly what ``material.transmute(schedule)`` gives it, and
    the result is one ``TransmutationResults`` keyed by each material's ``id``,
    the same shape ``Model.simulate_transmutation`` returns.

    What a Python loop would repeat is done once instead: the chain is read
    once, each nuclide's cross sections are decoded once and shared by every
    material that needs them, and a multigroup collapse runs once for every
    distinct combination of spectrum, composition, temperature and shielding.
    Cells of one steel that saw the same spectrum share a collapse, whatever
    their flux magnitudes. ``results.collapse_reuse`` says how many collapses
    were shared. The materials' solves then run in parallel.

    Args:
        materials (list[Material]): The materials, each with a distinct
            ``id``, since the results are keyed by it. Every material has one
            from construction; two share one only when one was given the
            other's explicitly or is a copy of it, as a material a
            transmutation result returns is of its original. As with
            ``Material.transmute``, the cross sections each needs are loaded
            into it and kept, and the composition is not modified.
        schedules (PulseSchedule | list[PulseSchedule]): One schedule for every
            material, or one per material in the same order. Each material's
            schedule carries its own spectra (the pulse sources) and flux
            magnitudes (the pulse rates), which is how each cell of a mesh gets
            its own flux. They must share one timeline, the same durations and
            irradiation on the same steps, because the result has one series of
            times.
        data_uncertainty (DataUncertainty, optional): Nuclear-data uncertainty,
            applied to every material as ``Material.transmute`` applies it to
            one. The same seed perturbs a given evaluation the same way in every
            material.
        self_shielding_chord (float, optional): One chord length ``4V/S`` in cm,
            for every material. See ``Material.transmute``.
        self_shielding_shape (SphereLump | CubeLump | FoilLump | CylinderLump |
WireLump, optional):
            One lump shape for every material, turned into a chord through each
            material's own ``volume``. Give this or ``self_shielding_chord``,
            not both.
        displacement_damage (bool): Also compute each material's displacement
            damage, damage energy per atom and NRT dpa. See
            ``Material.transmute``. Off by default, and then nothing about the
            solve or the data it loads changes.
        displacement_energies (dict[str, float], optional): Displacement
            threshold energies in eV by element symbol, replacing the defaults,
            for every material. Needs ``displacement_damage=True``.
        product_threshold (float): The atom fraction, or share of a decay
            quantity, a product must reach for its cross sections to be loaded
            and its reactions solved, applied to each material's own schedule.
            Default 1e-12. See ``Material.transmute``.
        processing (Processing, optional): Removals, transfers between
            materials and feeds. The materials a transfer links are solved
```

```
            together each step. A material it names that is not in
            ``materials``, a storage tank say, is carried unirradiated and
            reported under its own id. Not yet combined with
            ``data_uncertainty``.

    Returns:
        TransmutationResults: Keyed by each material's ``id``. Per material,
            ``get_source_rates(id)`` gives its flux magnitudes and
            ``get_self_shielding_info(id)`` its shielding report.

    Raises:
        ValueError: If two materials share an id, the timelines differ, the
            number of schedules does not match the number of materials, or
            any single material would fail ``Material.transmute``, in which
            case the message names the material.
        TypeError: If ``schedules`` is neither a PulseSchedule nor a list of
            them, or the same Material object is given twice.

    Examples:
        >>> schedules = [
        ...     yani.PulseSchedule([
        ...         yani.Pulse(rate=flux[i], duration=(1, "y"), source=spectra[i]),
        ...         yani.Cooldown(duration=(1, "d")),
        ...     ])
        ...     for i in range(len(cells))
        ... ]
        >>> results = yani.transmute(cells, schedules)
        >>> results.get_final_material(cells[3].id)
    """
```

Irradiation schedules

---

class yani.Pulse

```
Pulse(rate: float, duration: Any, source: Optional[Any] = None, flux_std_dev:
Optional[Sequence[float]] = None, flux_covariance: Optional[Sequence[Sequence[float]]]
= None, model: Optional[Any] = None, moves: Optional[Any] = None)
```

One irradiation campaign: a neutron source held at a constant `rate` (n/s) for `duration`.

Create an irradiation pulse.

Parameters:

Name	Type	Description
`rate`	`float`	For `Model.simulate_transmutation` rate in particles/second (n/s). For `Ma` the total flux magnitude [n/cm^2/s] (t `Histogram` energy gives the spectrur
`duration`	`Any`	A number of seconds, or a `(value, u` `(1, "a")` or `(30, "d")`. Units: s, mi
`source`	`Optional[Any]`	The NeutronSource for this campaigr which `Model.simulate_transmutati` uses the model's source); `Material.` source whose energy is a `Histogram` position/direction.
`flux_std_dev`	`Optional[Sequence[float]]`	Per-bin standard deviation of the flux the same order as the `Histogram` va omitting it is the common case: a spe published reference set carries no sta run it is `results[tally].standard_d` Used only by `Material.transmute(data_uncertai` `flux_spectrum` source enabled. Omi contributes nothing and says so in `data_uncertainty_info["spectra_w` rather than reading as a flux known e Treats the bins as independent. A spe Carlo tally is not: its bins are scored b and move together, and a per-bin sigr the error of every rate that sums over `flux_covariance` instead when the known.
`flux_covariance`	`Optional[Sequence[Sequence[float]]]`	The full covariance of the `Histogram` matrix (nested lists or a 2-D array) wi per bin, in the square of their units. Tl `flux_std_dev`, whose diagonal it co other. It must be symmetric and posit is checked. Each replica's flux perturb through its factor, so correlated bins
`model`	`Optional[Any]`	The yani `Model` this pulse irradiates, `yamc.simulate_shutdown_dose`: its g

Name	Type	Description
		and neutron source are the machine it instead of `source`. A later step wit this one.
`moves`	`Optional[Any]`	Cells of `model` that sit away from the as `{cell or cell name: (dx, dy,` rigid transformation matrix. Only with `Cooldown` for how moves work.

Source code in `yani/_core/__init__.pyi`

```
def __new__(cls, rate: builtins.float, duration: typing.Any, source:
typing.Optional[typing.Any] = None, flux_std_dev:
typing.Optional[typing.Sequence[builtins.float]] = None, flux_covariance:
typing.Optional[typing.Sequence[typing.Sequence[builtins.float]]] = None, model:
typing.Optional[typing.Any] = None, moves: typing.Optional[typing.Any] = None) ->
Pulse:
    r"""
    Create an irradiation pulse.

    Args:
        rate: For ``Model.simulate_transmutation``, the source emission rate
            in particles/second (n/s). For ``Material.transmute``, the total
            flux magnitude [n/cm^2/s] (the source's ``Histogram`` energy gives
            the spectrum shape).
        duration: A number of seconds, or a ``(value, unit)`` tuple such as
            ``(1, "a")`` or ``(30, "d")``. Units: s, min, h, d, a (year).
        source: The NeutronSource for this campaign. Defaults to ``None``,
            which ``Model.simulate_transmutation`` accepts (transport uses the
            model's source); ``Material.transmute`` requires a source whose
            energy is a ``Histogram`` and ignores its position/direction.
        flux_std_dev: Per-bin standard deviation of the flux, in the same
            units and the same order as the ``Histogram`` values. Optional,
            and omitting it is the common case: a spectrum taken from a
            published reference set carries no stated error. From a yani
            run it is ``results[tally].standard_deviation``.

            Used only by ``Material.transmute(data_uncertainty=...)`` with
            the ``flux_spectrum`` source enabled. Omitted, that source
            contributes nothing and says so in
            ``data_uncertainty_info["spectra_without_flux_sigma"]`` rather
            than reading as a flux known exactly.

            Treats the bins as independent. A spectrum from a Monte Carlo
            tally is not: its bins are scored by the same histories and
            move together, and a per-bin sigma then understates the error
            of every rate that sums over a band. Give ``flux_covariance``
            instead when the correlations are known.
        flux_covariance: The full covariance of the ``Histogram`` values, a
            square matrix (nested lists or a 2-D array) with one row and
            column per bin, in the square of their units. The alternative
            to ``flux_std_dev``, whose diagonal it contains: give one or
            the other. It must be symmetric and positive semi-definite, and
            is checked. Each replica's flux perturbation is drawn through
            its factor, so correlated bins move together.
        model: The yani ``Model`` this pulse irradiates, for
            ``yamc.simulate_shutdown_dose``: its geometry, materials and
            neutron source are the machine during this pulse. Give it
            instead of ``source``. A later step without a ``model`` stays in
            this one.
        moves: Cells of ``model`` that sit away from their reference
            position, as ``{cell or cell name: (dx, dy, dz)}`` in cm or a
            4x4 rigid transformation matrix. Only with ``model``; see
            ``Cooldown`` for how moves work.
    """
```

attr model property

```
model: Optional[Any]
```

The Model this pulse irradiates, or `None`.

attr moves property

```
moves: dict[str, list[list[float]]]
```

The moved cells, `{name: 4x4 matrix}`.

attr source property

```
source: Optional[Any]
```

The neutron source active during this pulse, or `None` if sourceless.

attr flux_std_dev property

```
flux_std_dev: Optional[list[float]]
```

The per-bin flux standard deviation, or `None` if none was given.

attr flux_covariance property

```
flux_covariance: Optional[list[list[float]]]
```

The flux covariance, or `None` if none was given.

attr rate property

```
rate: float
```

Source emission rate in particles/second.

attr duration property

```
duration: float
```

Pulse duration in seconds.

`class` yani.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 yani `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 `yani/_core/__init__.pyi`

```
259  def __new__(cls, duration: typing.Any, model: typing.Optional[typing.Any] = None, moves:
260  typing.Optional[typing.Any] = None) -> Cooldown:
261      r"""
262      Create a cooling step.
263
264      Args:
265          duration: A number of seconds, or a ``(value, unit)`` tuple such as
266              ``(7, "d")``. Units: s, min, h, d, a (year).
267          model: The yani ``Model`` the machine is in while it cools, for
268              ``yamc.simulate_shutdown_dose``: the decay photons of every
269              earlier irradiation are transported through its geometry. It
270              needs no source. Cells absent from it have been removed and no
271              longer emit; a cell of the same name is the same component. A
272              later step without a ``model`` stays in this one.
273          moves: Cells of ``model`` that sit away from their reference
274              position (where they were built in the model they were
275              irradiated in, unless that step moved them too), as
276              ``{cell or cell name: (dx, dy, dz)}`` in cm or a 4x4 rigid
277              transformation matrix. The activation moves with the cell;
278              ``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.

func yani.cooldown_steps

```
cooldown_steps(start: Any, stop: Any, n: int, spacing: str = 'log') -> list[Cooldown]
```

Build a run of cooling steps at log-spaced *cumulative* times.

`Cooldown` takes the duration OF THAT STEP, which is the right primitive and the wrong thing to type. Anyone plotting a decay curve wants points at cumulative times, and would otherwise have to difference them by hand:

```
HOUR, YEAR = 3600.0, 365.25 * 86400.0
cum  = [HOUR * (10 * YEAR / HOUR) ** (k / (n - 1)) for k in range(n)]
gaps = [cum[0]] + [cum[k] - cum[k - 1] for k in range(1, n)]
```

Log spacing is the default because decay is exponential, so a linear sweep is wrong twice over: it wastes steps in the flat tail and misses the early fall entirely. Activity and decay heat are plotted on log-log axes, so evenly spaced PLOTTED points means log-spaced times.

Source code in `yani/_core/__init__.pyi`

```
5318  def cooldown_steps(start: typing.Any, stop: typing.Any, n: builtins.int, spacing:
5319  builtins.str = 'log') -> builtins.list[Cooldown]:
5320      r"""
5321      Build a run of cooling steps at log-spaced *cumulative* times.
5322
5323      `Cooldown` takes the duration OF THAT STEP, which is the right primitive and
5324      the wrong thing to type. Anyone plotting a decay curve wants points at
5325      cumulative times, and would otherwise have to difference them by hand:
5326
5327      ```text
5328      HOUR, YEAR = 3600.0, 365.25 * 86400.0
5329      cum  = [HOUR * (10 * YEAR / HOUR) ** (k / (n - 1)) for k in range(n)]
5330      gaps = [cum[0]] + [cum[k] - cum[k - 1] for k in range(1, n)]
5331      ```
5332
5333      Log spacing is the default because decay is exponential, so a linear sweep
5334      is wrong twice over: it wastes steps in the flat tail and misses the early
5335      fall entirely. Activity and decay heat are plotted on log-log axes, so
5336      evenly spaced PLOTTED points means log-spaced times.
         """
```

`class` yani.PulseSchedule

```
PulseSchedule(steps: Any)
```

An irradiation/cooling timeline built from `Pulse` and `Cooldown` steps.

`time_correct_tally` post-processes a decay-photon tally into shutdown dose rates over the schedule.

Build a schedule from a list of `Pulse` / `Cooldown` steps.

Source code in `yani/_core/__init__.pyi`

```
2804  def __new__(cls, steps: typing.Any) -> PulseSchedule:
2805      r"""
2806      Build a schedule from a list of `Pulse` / `Cooldown` steps.
2807      """
```

attr sources property

```
sources: list[Any]
```

The distinct neutron sources used by the pulses, in first-appearance order. For a mixed-spectrum schedule, pass `time_correct_tally` a dict mapping each of these to its transport run's tally result.

---

class yani.NeutronSource

```
NeutronSource(*, position: Optional[Any] = None, energy: Optional[Any] = None,
direction: Optional[Any] = None, strength: Optional[float] = None)
```

A neutron particle source with spatial, energy, and angular distributions.

Create a neutron source.

Parameters:

Name	Type	Description	Default
`position`	`Optional[Any]`	Birth location. Pass a tuple/list `(x, y, z)` in cm for a fixed point (recommended), or a spatial distribution object (yani.sources.CylindricalRing, yani.sources.MeshDistribution) for finer control. Defaults to the origin (0, 0, 0).	`None`
`energy`	`Optional[Any]`	Birth energy. Pass a float in eV for a single energy (recommended), or an energy distribution (yamc.Discrete, yamc.Uniform, yamc.Normal) for a spectrum. Defaults to 14.06 MeV (D-T fusion).	`None`
`direction`	`Optional[Any]`	Angular distribution (e.g., yamc.Isotropic()). Defaults to Isotropic.	`None`
`strength`	`Optional[float]`	Source strength for multi-source weighting. Defaults to 1.0.	`None`

Returns:

Name	Type	Description
`NeutronSource`	`NeutronSource`	A neutron source.

Raises:

Type	Description
`ValueError`	If `position` has a NaN or infinite coordinate, `energy` is a NaN or infinite number, or `strength` is NaN, infinite or negative. The message names the parameter and its value.

Source code in `yani/_core/__init__.pyi`

```
2032  def __new__(cls, *, position: typing.Optional[typing.Any] = None, energy:
2033  typing.Optional[typing.Any] = None, direction: typing.Optional[typing.Any] = None,
2034  strength: typing.Optional[builtins.float] = None) -> NeutronSource:
2035      r"""
2036      Create a neutron source.
2037
2038      Args:
2039          position: Birth location. Pass a tuple/list ``(x, y, z)`` in cm for a fixed
2040                 point (recommended), or a spatial distribution object
2041                 (yani.sources.CylindricalRing, yani.sources.MeshDistribution) for finer
2042  control.
2043                 Defaults to the origin (0, 0, 0).
2044          energy: Birth energy. Pass a float in eV for a single energy (recommended),
2045                 or an energy distribution (yamc.Discrete, yamc.Uniform, yamc.Normal)
2046                 for a spectrum. Defaults to 14.06 MeV (D-T fusion).
2047          direction: Angular distribution (e.g., yamc.Isotropic()).
2048                 Defaults to Isotropic.
2049          strength: Source strength for multi-source weighting. Defaults to 1.0.
2050
2051      Returns:
2052          NeutronSource: A neutron source.
2053
2054      Raises:
2055          ValueError: If ``position`` has a NaN or infinite coordinate, ``energy``
              is a NaN or infinite number, or ``strength`` is NaN, infinite or
              negative. The message names the parameter and its value.
      """
```

attr position `property` `writable`

```
position: Any
```

Spatial distribution describing where particles are sampled. Returns an `(x, y, z)` list for a fixed point, or the spatial distribution object.

attr energy `property` `writable`

```
energy: Any
```

Energy distribution describing source particle energies in eV. Returns a `Discrete`, `Uniform`, or `Normal` distribution.

attr direction `property` `writable`

```
direction: Any
```

Angular distribution describing source particle directions. Returns an `Isotropic` or `Monodirectional` distribution.

attr strength property writable

```
strength: float
```

Relative strength of this source for multi-source weighting. Default is 1.0; only meaningful when multiple sources are passed to a Model.

Continuous processing

Feeds, extraction and transfers are configured on one `Processing` and passed to the batch solver; see Continuous processing.

---

class yani.Processing

```
Processing()
```

Removals, transfers between materials and feeds, for a transmutation run.

Each term adds to the Bateman equations of the materials it names, and every one is solved exactly: a first-order removal is a loss rate, a transfer is that loss arriving in the destination scaled by the ratio of volumes, and a feed is a source held constant over each interval. Materials a transfer links are solved together each step.

A destination need not be irradiated. One that `transmute` or `simulate_transmutation` is not otherwise given (a storage tank outside the geometry, say) is carried unirradiated and reported in the results under its own id, so its activity and contact dose read like any other material's.

A rate is a number, or a table `(times, rates)`: piecewise constant, each rate holding from its time (seconds from the start of the schedule, or `(value, unit)`) to the next, and zero before the first. Steps are split where a table changes, so a table is solved exactly. `steps` restricts a term to the listed schedule steps (0-based).

Examples:

```
>>> loop = yani.Processing()
>>> # tritium out of the breeder with a 2 h half-time, into a tank
>>> loop.add_transfer(pbli, "H3", rate=(2, "h"), rate_units="half-time",
...                   destination=tank)
>>> loop.add_transfer(pbli, "He", rate=6, rate_units="1/h")  # vented
>>> loop.add_feed(pbli, {"Li6": 1.0}, rate=0.05, rate_units="kg/d", steps=[0, 2])
>>> # a whole liquid circulating between a core and a pool
>>> loop.add_transfer(water, rate=flow / water.volume, destination=pool)
>>> loop.add_transfer(pool, rate=flow / pool.volume, destination=water)
>>> results = yani.transmute([pbli, water], schedules, processing=loop)
```

Source code in `yani/_core/__init__.pyi`

```
2657    def __new__(cls) -> Processing: ...
```

attr material_ids property

```
material_ids: list[int]
```

The ids of every material a term names.

meth add_transfer

```
add_transfer(material: Material, component: Optional[str] = None, *, rate: Any,
rate_units: str = '1/s', destination: Optional[Material] = None, steps:
Optional[Sequence[int]] = None) -> None
```

Remove `component` from `material` continuously, at a first-order rate, into `destination` when given and out of the system otherwise.

Parameters:

Name	Type	Description	Default
`material`	`Material`	Where it is removed from.	*required*
`component`	`str`	An element (`"H"`, every isotope) or a nuclide (`"H3"`). `None` (the default) removes everything, as a flowing liquid does.	`None`
`rate`	`float | tuple`	The rate, or a `(times, rates)` table.	*required*
`rate_units`	`str`	`"1/s"` (or `1/min`, `1/h`, `1/d`, `1/a`), or `"half-time"` for the time to remove half, in seconds or as `(value, unit)`.	`'1/s'`
`destination`	`Material`	Where it goes. Both materials need a volume: what leaves is diluted by their ratio.	`None`
`steps`	`list[int]`	The schedule steps it acts on; every step by default.	`None`

Source code in `yani/_core/__init__.pyi`

```
2658  def add_transfer(self, material: Material, component: typing.Optional[builtins.str]
2659  = None, *, rate: typing.Any, rate_units: builtins.str = '1/s', destination:
2660  typing.Optional[Material] = None, steps:
2661  typing.Optional[typing.Sequence[builtins.int]] = None) -> None:
2662      r"""
2663      Remove ``component`` from ``material`` continuously, at a first-order
2664      rate, into ``destination`` when given and out of the system otherwise.
2665
2666      Args:
2667          material (Material): Where it is removed from.
2668          component (str, optional): An element (``"H"``, every isotope) or a
2669              nuclide (``"H3"``). ``None`` (the default) removes everything,
2670              as a flowing liquid does.
2671          rate (float | tuple): The rate, or a ``(times, rates)`` table.
2672          rate_units (str): ``"1/s"`` (or ``1/min``, ``1/h``, ``1/d``,
2673              ``1/a``), or ``"half-time"`` for the time to remove half, in
2674              seconds or as ``(value, unit)``.
2675          destination (Material, optional): Where it goes. Both materials need
2676              a volume: what leaves is diluted by their ratio.
              steps (list[int], optional): The schedule steps it acts on; every
                  step by default.
          """
```

meth add_feed

```
add_feed(material: Material, composition: Mapping[str, float], *, rate: Any,
rate_units: str = 'g/s', proportional_to: Optional[str] = None, steps:
Optional[Sequence[int]] = None) -> None
```

Feed `composition` into `material` continuously.

Parameters:

Name	Type	Description	Default
`material`	`Material`	What it is fed into.	*required*
`composition`	`dict[str, float]`	Nuclides or natural elements, by atom fraction (normalized).	*required*
`rate`	`float | tuple`	The rate, or a `(times, rates)` table.	*required*
`rate_units`	`str`	An amount per time, `"g/s"`, `"kg/d"`, `"mol/h"` or `"atoms/s"`, for which `material` needs a volume; or with `proportional_to` a rate constant `"1/s"`.	`'g/s'`
`proportional_to`	`str`	A nuclide of `material`: the feed is then `rate` times its inventory, which holds an enrichment (Li6 fed in proportion to Li7, say).	`None`
`steps`	`list[int]`	The schedule steps it acts on; every step by default.	`None`

Source code in `yani/_core/__init__.pyi`

```
def add_feed(self, material: Material, composition: typing.Mapping[builtins.str,
builtins.float], *, rate: typing.Any, rate_units: builtins.str = 'g/s',
proportional_to: typing.Optional[builtins.str] = None, steps:
typing.Optional[typing.Sequence[builtins.int]] = None) -> None:
    r"""
    Feed ``composition`` into ``material`` continuously.

    Args:
        material (Material): What it is fed into.
        composition (dict[str, float]): Nuclides or natural elements, by
            atom fraction (normalized).
        rate (float | tuple): The rate, or a ``(times, rates)`` table.
        rate_units (str): An amount per time, ``"g/s"``, ``"kg/d"``,
            ``"mol/h"`` or ``"atoms/s"``, for which ``material`` needs a
            volume; or with ``proportional_to`` a rate constant ``"1/s"``.
        proportional_to (str, optional): A nuclide of ``material``: the feed
            is then ``rate`` times its inventory, which holds an enrichment
            (Li6 fed in proportion to Li7, say).
        steps (list[int], optional): The schedule steps it acts on; every
            step by default.
    """
```

Results

`class` yani.TransmutationResults

Results from a transmutation calculation.

Contains material compositions at each timestep for all transmuted materials. Index 0 is the initial composition, index i is after timestep[i-1].

`attr` data_gaps `property`

```
data_gaps: dict
```

What the result rests on that the nuclear data does not give, per material ID.

A product whose decay the decay library has no record for is never booked onto another nuclide: its fate is unknown, so it is counted as produced and kept out of the inventory (`get_material` does not hold it). Fission in a nuclide the library gives no yields for has unknown products. Each material's entry is a dict:

- `decay_library` : the decay data the run read, as configured;

- `volume`: the material's volume in cm^3, or `None`;
- `fate_unknown`: per nuclide, `atoms_per_barn_cm` at each state (index 0 is the initial composition), `production_rate` over each step and `activity_bound` at each state, both per barn-cm per second, and `produced_by` (`"<parent> <reaction>"`, `"<parent> decay"` or `"<parent> fission"`). The bound holds whatever the half-life: activity is never more than the largest production rate so far, taken here over each step's average;
- `fission_without_yields`: per nuclide, its `fission_rate` in each step, per barn-cm per second;
- `unknown_fission_share`: per step, the share of the whole fission rate whose products are unknown.

Every entry names a gap of the library the run read, which is what its evaluators would need to fill it. A material whose answer rests on nothing missing has empty `fate_unknown` atoms and no `fission_without_yields`.

attr times property

```
times: list[float]
```

Cumulative times [s] including t=0.

attr timesteps property

```
timesteps: list[float]
```

Timesteps used [s].

attr collapse_reuse property

```
collapse_reuse: Optional[dict]
```

How much of the multigroup collapse work was shared, or `None` for a transport-coupled solve, which does none.

A dict with `performed`, the collapses actually run, and `requested`, one per distinct spectrum per material. Materials with the same spectrum, composition, temperature and shielding collapse to the same rates and share one, so `performed` below `requested` is the saving `transmute` made over solving them one at a time.

attr data_releases property

```
data_releases: dict
```

The nuclear-data release each library keyword came from, and the release of each local release folder data was read from, as a dict keyed by keyword or by the folder's absolute path. Each value is a dict with `release` (the release identifier), `manifest_sha256` (the hash of that release's manifest, which pins every file in it), `format_version`, `offline` (True when the data origin was unreachable and the newest complete cached release was used, always False for a local folder), `converters` (every build that wrote the release's files, as its manifest lists them, such as `"yani-python 0.22.1 (core 250c439e...)"`; empty for a release published before releases listed them), `locally_processed` and `library`. A transmutation subsection read from a local directory that is not a release folder (one `convert_transmutation` wrote) is keyed by that directory's absolute path, with `locally_processed` True, `library` and `release` the library and `data_version` its converter recorded, and `manifest_sha256` the sha256 of its files. For a release `locally_processed` is False and `library` empty.

attr num_steps property

```
num_steps: int
```

Number of transmutation steps.

attr material_ids property

```
material_ids: list[int]
```

List of material IDs that were transmuted.

meth get_source_rates

```
get_source_rates(material_id: int) -> Optional[list[float]]
```

The rate each step drove a material at, one per timestep, zero for a cooldown.

A spectrum solve (`Material.transmute`, `transmute`) gives each material's own flux magnitude in n/cm^2/s, which differs between materials given their own schedules.
`Model.simulate_transmutation` gives the source strength in n/s, the same for every material.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Returns:

Type	Description
`Optional[list[float]]`	List of rates, or None if the material is not in the results.

Source code in `yani/_core/__init__.pyi`

```
3470  def get_source_rates(self, material_id: builtins.int) ->
3471  typing.Optional[builtins.list[builtins.float]]:
3472      r"""
3473      The rate each step drove a material at, one per timestep, zero for a
3474      cooldown.
3475
3476      A spectrum solve (``Material.transmute``, ``transmute``) gives each
3477      material's own flux magnitude in n/cm^2/s, which differs between
3478      materials given their own schedules. ``Model.simulate_transmutation``
3479      gives the source strength in n/s, the same for every material.
3480
3481      Args:
3482          material_id: Material ID number.
3483
3484      Returns:
3485          List of rates, or None if the material is not in the results.
      """
```

`meth` get_uncertainty_breakdown

```
get_uncertainty_breakdown(material_id: int, nuclide: str, step: int) ->
Optional[dict]
```

Where a nuclide's uncertainty at one step comes from.

Present when the run was asked for it with `DataUncertainty(attribution=True)`, and `None` otherwise. A dict:

- `variance`: the total, resampled, the square of `get_nuclide_uncertainty`;
- `by_source`: each source alone, resampled the same way, so this says how much is statistical and how much is each kind of nuclear data. The sources are independent and these sum to the total;
- `unattributed`: what that sum leaves, interaction and sampling noise, small when the attribution holds;
- `contributors`: first order, a list of `(source, nuclide, reaction, variance)`, largest reach first. Within the cross sections a nuclide's whole evaluation has `reaction` of `None` and each channel alone names it; a half-life has `None`. A decay branching contributor is a two-mode parent's one degree of freedom, with `reaction` of `None`. It says which evaluation to look at; the total is the resampled one.

- `linearity`: how well that first order explains the replicas, per source and as `"all"` for every source together. Each replica's first-order prediction is the nominal plus every sensitivity times that replica's own change in its input, so no extra solve is made. Per entry: `r2` (squared correlation between replicas and predictions), `residual_share` (the share of the variance first order does not account for), `by_contributor` (`{source: {nuclide: r2}}`, each contributor's term alone), `ranking_agrees` (whether the first-order top contributor is also the best correlated), and `flagged` (`residual_share` above 0.1, or the ranking disagrees: read the contributors with care). First order has terms only for `cross_sections`, `half_life` and `decay_branching`. `None` for every other source (`flux_spectrum`, `statistical`, `decay_energy`, `decay_photon_lines`, `fission_yield`), and for `"all"` when any applied source is one of those; absent for a nuclide with no spread. The default source set applies several of them, so with the defaults `"all"` is `None`: restrict `DataUncertainty(sources=...)` to `cross_sections`, `half_life` and `decay_branching` (or a subset) for an `"all"` entry.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name.	*required*
`step`	`int`	As in `get_nuclide_uncertainty`: 0 is the initial composition, which carries none.	*required*

Source code in `yani/_core/__init__.pyi`

```
def get_uncertainty_breakdown(self, material_id: builtins.int, nuclide:
builtins.str, step: builtins.int) -> typing.Optional[dict]:
    r"""
    Where a nuclide's uncertainty at one step comes from.

    Present when the run was asked for it with
    ``DataUncertainty(attribution=True)``, and ``None`` otherwise. A dict:

    - ``variance``: the total, resampled, the square of
      ``get_nuclide_uncertainty``;
    - ``by_source``: each source alone, resampled the same way, so this
      says how much is statistical and how much is each kind of nuclear
      data. The sources are independent and these sum to the total;
    - ``unattributed``: what that sum leaves, interaction and sampling
      noise, small when the attribution holds;
    - ``contributors``: first order, a list of ``(source, nuclide,
      reaction, variance)``, largest reach first. Within the cross sections
      a nuclide's whole evaluation has ``reaction`` of ``None`` and each
      channel alone names it; a half-life has ``None``. A decay branching
      contributor is a two-mode parent's one degree of freedom, with
      ``reaction`` of ``None``. It says which evaluation to look at; the
      total is the resampled one.
    - ``linearity``: how well that first order explains the replicas, per
      source and as ``"all"`` for every source together. Each replica's
      first-order prediction is the nominal plus every sensitivity times
      that replica's own change in its input, so no extra solve is made.
      Per entry: ``r2`` (squared correlation between replicas and
      predictions), ``residual_share`` (the share of the variance first
      order does not account for), ``by_contributor`` (``{source:
      {nuclide: r2}}``, each contributor's term alone), ``ranking_agrees``
      (whether the first-order top contributor is also the best
      correlated), and ``flagged`` (``residual_share`` above 0.1, or the
      ranking disagrees: read the contributors with care). First order has
      terms only for ``cross_sections``, ``half_life`` and
      ``decay_branching``. ``None`` for every other source
      (``flux_spectrum``, ``statistical``, ``decay_energy``,
      ``decay_photon_lines``, ``fission_yield``), and for ``"all"`` when
      any applied source is one of those; absent for a nuclide with no
      spread. The default source set applies several of them, so with the
      defaults ``"all"`` is ``None``: restrict ``DataUncertainty(sources=...)``
      to ``cross_sections``, ``half_life`` and ``decay_branching`` (or a
      subset) for an ``"all"`` entry.

    Args:
        material_id: Material ID number.
        nuclide: Nuclide name.
        step: As in ``get_nuclide_uncertainty``: 0 is the initial
            composition, which carries none.
    """
```

`meth` get_reaction_rate_uncertainty

```
get_reaction_rate_uncertainty(material_id: int, step: int) ->
Optional[list[tuple[str, str, Optional[str], float, float]]]
```

The statistical uncertainty of each transport-tallied reaction rate at one step.

Present for `Model.simulate_transmutation` run with `data_uncertainty` including the `"statistical"` source, and `None` otherwise. Each entry is `(nuclide, reaction, target, rate, std_dev)` in 1/s per atom: a reaction total has `target` of `None`, an isomeric partial names its final state. The rates are the tally's, scaled by the step's source rate, exactly as the step's solve used them before the branching fold.

The rates are correlated, having been scored by the same histories, and the inventory sigmas are computed with those correlations. These standard deviations alone do not carry them.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Schedule step index, as in `get_reaction_rates`.	*required*

Source code in `yani/_core/__init__.pyi`

```
3534  def get_reaction_rate_uncertainty(self, material_id: builtins.int, step:
3535  builtins.int) -> typing.Optional[builtins.list[tuple[builtins.str, builtins.str,
3536  typing.Optional[builtins.str], builtins.float, builtins.float]]]:
3537      r"""
3538      The statistical uncertainty of each transport-tallied reaction rate
3539      at one step.
3540
3541      Present for ``Model.simulate_transmutation`` run with
3542      ``data_uncertainty`` including the ``"statistical"`` source, and
3543      ``None`` otherwise. Each entry is ``(nuclide, reaction, target, rate,
3544      std_dev)`` in 1/s per atom: a reaction total has ``target`` of ``None``,
3545      an isomeric partial names its final state. The rates are the tally's,
3546      scaled by the step's source rate, exactly as the step's solve used them
3547      before the branching fold.
3548
3549      The rates are correlated, having been scored by the same histories,
3550      and the inventory sigmas are computed with those correlations. These
3551      standard deviations alone do not carry them.
3552
3553      Args:
3554          material_id: Material ID number.
          step: Schedule step index, as in ``get_reaction_rates``.
      """
```

`meth` get_nuclide_evolution

```
get_nuclide_evolution(material_id: int, nuclide: str) -> Optional[list[float]]
```

Get the evolution of a specific nuclide over all timesteps.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name (e.g., "Co60").	*required*

Returns:

Type	Description
`Optional[list[float]]`	List of atom densities [atoms/barn-cm] at each time point
`Optional[list[float]]`	(index 0 = initial, index i = after step i).
`Optional[list[float]]`	Returns None if material_id not found.

Source code in `yani/_core/__init__.pyi`

```
3555  def get_nuclide_evolution(self, material_id: builtins.int, nuclide: builtins.str) -
3556  > typing.Optional[builtins.list[builtins.float]]:
3557      r"""
3558      Get the evolution of a specific nuclide over all timesteps.
3559
3560      Args:
3561          material_id: Material ID number.
3562          nuclide: Nuclide name (e.g., "Co60").
3563
3564      Returns:
3565          List of atom densities [atoms/barn-cm] at each time point
3566          (index 0 = initial, index i = after step i).
3567          Returns None if material_id not found.
          """
```

`meth` get_nuclide_density

```
get_nuclide_density(material_id: int, nuclide: str, step: int) -> Optional[float]
```

Get nuclide density at a specific timestep.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name (e.g., "U235").	*required*
`step`	`int`	Timestep index (0 = initial).	*required*

Returns:

Type	Description
`Optional[float]`	Atom density [atoms/barn-cm] or None if not found.

Source code in `yani/_core/__init__.pyi`

```
3568  def get_nuclide_density(self, material_id: builtins.int, nuclide: builtins.str,
3569  step: builtins.int) -> typing.Optional[builtins.float]:
3570      r"""
3571      Get nuclide density at a specific timestep.
3572
3573      Args:
3574          material_id: Material ID number.
3575          nuclide: Nuclide name (e.g., "U235").
3576          step: Timestep index (0 = initial).
3577
3578      Returns:
3579          Atom density [atoms/barn-cm] or None if not found.
      """
```

`meth` get_nuclide_uncertainty

```
get_nuclide_uncertainty(material_id: int, nuclide: str, step: int) ->
Optional[float]
```

Get the nuclear-data standard deviation on a nuclide density.

The spread over an ensemble of solves with the activation cross sections resampled from their MF=33 covariance. In the same units as `get_nuclide_density`, so the pair reads as `mean ± sigma`. With transport replica weights the Monte Carlo noise of the replicas' tallies is removed from it (see `get_nuclide_uncertainty_noise`).

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name (e.g., "Mn56").	*required*
`step`	`int`	Timestep index (0 = initial).	*required*

Returns:

Type	Description
`Optional[float]`	Standard deviation [atoms/barn-cm], or None if the transmutation was
`Optional[float]`	run without `data_uncertainty`. Step 0 is the initial composition
`Optional[float]`	and always reports 0.0: it is an input, and perturbing cross sections
`Optional[float]`	does not move it.
`Optional[float]`	A nuclide whose evaluation carries no covariance also reports 0.0.
`Optional[float]`	That is not a claim of certainty -- check
`Optional[float]`	`get_data_uncertainty_info(material_id)["no_covariance_data"]`,
`Optional[float]`	which lists exactly
`Optional[float]`	those nuclides.

Source code in `yani/_core/__init__.pyi`

```
3580  def get_nuclide_uncertainty(self, material_id: builtins.int, nuclide: builtins.str,
3581  step: builtins.int) -> typing.Optional[builtins.float]:
3582      r"""
3583      Get the nuclear-data standard deviation on a nuclide density.
3584
3585      The spread over an ensemble of solves with the activation cross sections
3586      resampled from their MF=33 covariance. In the same units as
3587      ``get_nuclide_density``, so the pair reads as ``mean ± sigma``. With
3588      transport replica weights the Monte Carlo noise of the replicas'
3589      tallies is removed from it (see ``get_nuclide_uncertainty_noise``).
3590
3591      Args:
3592          material_id: Material ID number.
3593          nuclide: Nuclide name (e.g., "Mn56").
3594          step: Timestep index (0 = initial).
3595
3596      Returns:
3597          Standard deviation [atoms/barn-cm], or None if the transmutation was
3598          run without ``data_uncertainty``. Step 0 is the initial composition
3599          and always reports 0.0: it is an input, and perturbing cross sections
3600          does not move it.
3601
3602          A nuclide whose evaluation carries no covariance also reports 0.0.
3603          That is not a claim of certainty -- check
3604          ``get_data_uncertainty_info(material_id)["no_covariance_data"]``,
3605          which lists exactly
3606          those nuclides.
      """
```

`meth` get_nuclide_uncertainty_standard_error

```
get_nuclide_uncertainty_standard_error(material_id: int, nuclide: str, step: int)
-> Optional[float]
```

Get the standard error of `get_nuclide_uncertainty`: how far another ensemble of the same size could put that sigma.

It allows for a heavy tail, through the replicas' sample kurtosis, so a lognormal-tailed density reads as less settled than a Gaussian one at the same replica count.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name.	*required*
`step`	`int`	As in `get_nuclide_uncertainty`.	*required*

Returns:

Type	Description
`Optional[float]`	The standard error [atoms/barn-cm], or None if the transmutation was
`Optional[float]`	run without `data_uncertainty` or with fewer than four replicas.
`Optional[float]`	Step 0 reports 0.0.

Source code in `yani/_core/__init__.pyi`

```
3607  def get_nuclide_uncertainty_standard_error(self, material_id: builtins.int,
3608  nuclide: builtins.str, step: builtins.int) -> typing.Optional[builtins.float]:
3609      r"""
3610      Get the standard error of ``get_nuclide_uncertainty``: how far another
3611      ensemble of the same size could put that sigma.
3612
3613      It allows for a heavy tail, through the replicas' sample kurtosis, so a
3614      lognormal-tailed density reads as less settled than a Gaussian one at
3615      the same replica count.
3616
3617      Args:
3618          material_id: Material ID number.
3619          nuclide: Nuclide name.
3620          step: As in ``get_nuclide_uncertainty``.
3621
3622      Returns:
3623          The standard error [atoms/barn-cm], or None if the transmutation was
3624          run without ``data_uncertainty`` or with fewer than four replicas.
3625          Step 0 reports 0.0.
      """
```

`meth` get_nuclide_uncertainty_noise

```
get_nuclide_uncertainty_noise(material_id: int, nuclide: str, step: int) ->
Optional[tuple[float, float]]
```

Get what `get_nuclide_uncertainty` is made of: the replicas' spread as solved, and the Monte Carlo noise removed from it.

With transport replica weights, each replica's tallied shift from the nominal carries Monte Carlo noise, which inflates the replicas' spread at low history counts. Each replica is then solved a second time from the even half of the transport's histories, and the spread of the difference between the two solves measures that noise, independent of every data draw. `get_nuclide_uncertainty` is `sqrt(replica_spread**2 - noise**2)`, zero where the noise is the larger.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name.	*required*
`step`	`int`	As in `get_nuclide_uncertainty`.	*required*

Returns:

Type	Description
`Optional[tuple[float, float]]`	`(replica_spread, noise)` [atoms/barn-cm], noise 0.0 where it was
`Optional[tuple[float, float]]`	not measured, or None if the transmutation was run without
`Optional[tuple[float, float]]`	`data_uncertainty`. Step 0 reports zeros.

Source code in `yani/_core/__init__.pyi`

```
3626  def get_nuclide_uncertainty_noise(self, material_id: builtins.int, nuclide:
3627  builtins.str, step: builtins.int) -> typing.Optional[tuple[builtins.float,
3628  builtins.float]]:
3629      r"""
3630      Get what ``get_nuclide_uncertainty`` is made of: the replicas' spread
3631      as solved, and the Monte Carlo noise removed from it.
3632
3633      With transport replica weights, each replica's tallied shift from the
3634      nominal carries Monte Carlo noise, which inflates the replicas' spread
3635      at low history counts. Each replica is then solved a second time from
3636      the even half of the transport's histories, and the spread of the
3637      difference between the two solves measures that noise, independent of
3638      every data draw. ``get_nuclide_uncertainty`` is
3639      ``sqrt(replica_spread**2 - noise**2)``, zero where the noise is the
3640      larger.
3641
3642      Args:
3643          material_id: Material ID number.
3644          nuclide: Nuclide name.
3645          step: As in ``get_nuclide_uncertainty``.
3646
3647      Returns:
3648          ``(replica_spread, noise)`` [atoms/barn-cm], noise 0.0 where it was
3649          not measured, or None if the transmutation was run without
              ``data_uncertainty``. Step 0 reports zeros.
          """
```

meth get_nuclide_uncertainty_evolution

```
get_nuclide_uncertainty_evolution(material_id: int, nuclide: str) ->
Optional[list[float]]
```

Get the standard deviation of a nuclide's density at every timestep.

Parallel to `get_nuclide_evolution`, leading zero included, so the two can be zipped without an index correction.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	Nuclide name.	*required*

Returns:

Type	Description
`Optional[list[float]]`	List of standard deviations [atoms/barn-cm], or None if the
`Optional[list[float]]`	transmutation was run without `data_uncertainty`.

Source code in `yani/_core/__init__.pyi`

```
3650  def get_nuclide_uncertainty_evolution(self, material_id: builtins.int, nuclide:
3651  builtins.str) -> typing.Optional[builtins.list[builtins.float]]:
3652      r"""
3653      Get the standard deviation of a nuclide's density at every timestep.
3654
3655      Parallel to ``get_nuclide_evolution``, leading zero included, so the two
3656      can be zipped without an index correction.
3657
3658      Args:
3659          material_id: Material ID number.
3660          nuclide: Nuclide name.
3661
3662      Returns:
3663          List of standard deviations [atoms/barn-cm], or None if the
3664          transmutation was run without ``data_uncertainty``.
          """
```

`meth` get_uncertainty_inventories

```
get_uncertainty_inventories(material_id: int, step: int) ->
Optional[list[dict[str, float]]]
```

Every replica's inventory at one timestep.

For a quantity that has to be evaluated per sample rather than from the mean. Activity, decay heat and the decay-photon spectrum are all functions of a whole inventory, so evaluating one of them once per entry here and taking the spread keeps the correlations between nuclides. Taking the mean inventory and evaluating once throws them away, and summing per-nuclide sigmas in quadrature assumes an independence that the resampling exists precisely to avoid assuming.

These are the replicas as solved. With transport replica weights their spread still holds the Monte Carlo noise that the sigmas this class reports have removed (see `get_nuclide_uncertainty_noise`).

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial, which has no ensemble).	*required*

Returns:

Type	Description
`Optional[list[dict[str, float]]]`	List of `{nuclide: density}` dicts, one per replica, or None if the
`Optional[list[dict[str, float]]]`	transmutation was run without `data_uncertainty`.

Source code in `yani/_core/__init__.pyi`

```
3665  def get_uncertainty_inventories(self, material_id: builtins.int, step:
3666  builtins.int) -> typing.Optional[builtins.list[builtins.dict[builtins.str,
3667  builtins.float]]]:
3668      r"""
3669      Every replica's inventory at one timestep.
3670
3671      For a quantity that has to be evaluated per sample rather than from the
3672      mean. Activity, decay heat and the decay-photon spectrum are all
3673      functions of a whole inventory, so evaluating one of them once per entry
3674      here and taking the spread keeps the correlations between nuclides.
3675      Taking the mean inventory and evaluating once throws them away, and
3676      summing per-nuclide sigmas in quadrature assumes an independence that the
3677      resampling exists precisely to avoid assuming.
3678
3679      These are the replicas as solved. With transport replica weights their
3680      spread still holds the Monte Carlo noise that the sigmas this class
3681      reports have removed (see ``get_nuclide_uncertainty_noise``).
3682
3683      Args:
3684          material_id: Material ID number.
3685          step: Timestep index (0 = initial, which has no ensemble).
3686
3687      Returns:
3688          List of ``{nuclide: density}`` dicts, one per replica, or None if the
          transmutation was run without ``data_uncertainty``.
      """
```

`meth` get_activity_uncertainty

```
get_activity_uncertainty(material_id: int, step: int, *, by_nuclide: bool = False)
-> Any
```

Activity at one timestep [Bq], with the nuclear-data spread on it.

Evaluated once per replica and summed within each, so the correlations between nuclides survive. Doing it any other way gives a plausible number that is wrong in a specific direction: evaluating from the mean inventory gives no spread at all, and adding the per-nuclide sigmas in quadrature double-counts a variance that partly cancels, since every Mn56 atom in an irradiated iron foil came out of an Fe56 atom.

The volume is taken from the stored step material, so the nominal value and every replica are scaled by the same one.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition). The composition at step 0 is an input, but each replica evaluates it with its own sampled half-lives, so the activity at step 0 has a spread whenever the `"half_life"` source is sampled.	*required*
`by_nuclide`	`bool`	Return a `dict[str, Estimate]` instead of one `Estimate` for the total. These do not add up to the total in quadrature, and are not meant to.	`False`

Returns:

Type	Description
`Any`	Estimate | dict[str, Estimate] | None: None if the transmutation
`Any`	was run without `data_uncertainty`.

Raises:

Type	Description
`ValueError`	if the material has no `volume` in cm^3, which a quantity in becquerel needs.

Source code in `yani/_core/__init__.pyi`

```
def get_activity_uncertainty(self, material_id: builtins.int, step: builtins.int,
*, by_nuclide: builtins.bool = False) -> typing.Any:
    r"""
    Activity at one timestep [Bq], with the nuclear-data spread on it.

    Evaluated once per replica and summed within each, so the correlations
    between nuclides survive. Doing it any other way gives a plausible
    number that is wrong in a specific direction: evaluating from the mean
    inventory gives no spread at all, and adding the per-nuclide sigmas in
    quadrature double-counts a variance that partly cancels, since every
    Mn56 atom in an irradiated iron foil came out of an Fe56 atom.

    The volume is taken from the stored step material, so the nominal value
    and every replica are scaled by the same one.

    Args:
        material_id: Material ID number.
        step: Timestep index (0 = initial composition). The composition at
            step 0 is an input, but each replica evaluates it with its own
            sampled half-lives, so the activity at step 0 has a spread
            whenever the ``"half_life"`` source is sampled.
        by_nuclide (bool): Return a ``dict[str, Estimate]`` instead of one
            ``Estimate`` for the total. These do not add up to the total in
            quadrature, and are not meant to.

    Returns:
        Estimate | dict[str, Estimate] | None: None if the transmutation
        was run without ``data_uncertainty``.

    Raises:
        ValueError: if the material has no ``volume`` in cm^3, which a
            quantity in becquerel needs.
    """
```

meth get_decay_heat_uncertainty

```
get_decay_heat_uncertainty(material_id: int, step: int, *, by_nuclide: bool =
False) -> Any
```

Decay heat at one timestep [W], with the nuclear-data spread on it.

See `get_activity_uncertainty`: same ensemble, same rule, and the quantity the FNS decay-heat benchmarks are written against. A calculated band beside the measured one is what turns a C/E into a statement about whether the disagreement is larger than the data uncertainty allows.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

With the `"decay_photon_lines"` source on, each replica's gamma decay energy E_EM follows its drawn photon lines and continua rather than an independent `"decay_energy"` draw, so its gamma heat and its contact dose come from the same draw of one evaluation. E_EM moves by the drawn change in the photon energy per decay (each line's energy times its intensity, plus each continuum's energy integral), and the part of E_EM the tabulated spectra do not carry is held at nominal. The beta and alpha parts keep their `"decay_energy"` draws. The heat then has a range, `Estimate.std_dev` to `Estimate.std_dev_correlated`, from the photon intensities' unstated correlations.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition). As in `get_activity_uncertainty`, step 0 has a spread whenever the `"half_life"`, `"decay_energy"` or `"decay_photon_lines"` source is sampled.	*required*
`by_nuclide`	`bool`	Return a `dict[str, Estimate]` of W by nuclide instead of one `Estimate` for the total.	`False`

Returns:

Type	Description
`Any`	Estimate | dict[str, Estimate] | None: None if the transmutation
`Any`	was run without `data_uncertainty`.

Raises:

Type	Description
`ValueError`	if the material has no `volume` in cm^3.

Source code in `yani/_core/__init__.pyi`

```
def get_decay_heat_uncertainty(self, material_id: builtins.int, step: builtins.int,
*, by_nuclide: builtins.bool = False) -> typing.Any:
    r"""
    Decay heat at one timestep [W], with the nuclear-data spread on it.

    See ``get_activity_uncertainty``: same ensemble, same rule, and the
    quantity the FNS decay-heat benchmarks are written against. A calculated
    band beside the measured one is what turns a C/E into a statement about
    whether the disagreement is larger than the data uncertainty allows.

    Args:
        material_id: Material ID number.
    With the ``"decay_photon_lines"`` source on, each replica's gamma decay
    energy E_EM follows its drawn photon lines and continua rather than an
    independent ``"decay_energy"`` draw, so its gamma heat and its contact
    dose come from the same draw of one evaluation. E_EM moves by the drawn
    change in the photon energy per decay (each line's energy times its
    intensity, plus each continuum's energy integral), and the part of E_EM
    the tabulated spectra do not carry is held at nominal. The beta and
    alpha parts keep their ``"decay_energy"`` draws. The heat then has a
    range, ``Estimate.std_dev`` to ``Estimate.std_dev_correlated``, from the
    photon intensities' unstated correlations.

    Args:
        material_id: Material ID number.
        step: Timestep index (0 = initial composition). As in
            ``get_activity_uncertainty``, step 0 has a spread whenever the
            ``"half_life"``, ``"decay_energy"`` or ``"decay_photon_lines"``
            source is sampled.
        by_nuclide (bool): Return a ``dict[str, Estimate]`` of W by nuclide
            instead of one ``Estimate`` for the total.

    Returns:
        Estimate | dict[str, Estimate] | None: None if the transmutation
        was run without ``data_uncertainty``.

    Raises:
        ValueError: if the material has no ``volume`` in cm^3.
    """
```

meth get_contact_dose_uncertainty

```
get_contact_dose_uncertainty(material_id: int, step: int, *, dose_quantity: str =
'absorbed-air', build_up: float = 2.0, by_nuclide: bool = False) -> Any
```

Contact dose rate at one timestep, with the nuclear-data spread on it.

The one derived 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: 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 gets the cancellation for free.

Needs no `volume`, unlike the other three, because the estimate takes the material for a half-space.

The band is the spread of the replicas' inventories (each with its own half-lives when the `"half_life"` source is on), and of their decay photon line intensities and energies and continuum normalisations when the `"decay_photon_lines"` source is on. The photon attenuation (XCOM), air energy absorption (NIST SRD 126), ICRP-116 dose coefficients and the build-up factor are held at their nominal values and contribute nothing to it.

With the `"decay_photon_lines"` source on, the band is a range: `Estimate.std_dev` takes each nuclide's photon intensities as independent where the decay data state no correlation, and `Estimate.std_dev_correlated` as fully correlated (the lines of a spectrum, its normalisation and lines, and its gamma and x-ray spectra). ENDF/B-VIII.1 folds each spectrum's normalisation sigma into every line's, so a multi-line emitter's range there is wide; `get_data_uncertainty_info` names those spectra under `decay_photon_spectra_folded`.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition). As in `get_activity_uncertainty`, step 0 has a spread whenever the `"half_life"` or `"decay_photon_lines"` source is sampled.	*required*
`dose_quantity`	`str`	`'absorbed-air'` (Gy/h, the default) or `'effective'` (Sv/h), as `Material.contact_dose` takes them.	`'absorbed-air'`
`build_up`	`float`	Build-up factor, a plain multiplier on the answer.	`2.0`
`by_nuclide`	`bool`	Return a `dict[str, Estimate]` instead of one `Estimate` for the total.	`False`

Returns:

Type	Description
`Any`	Estimate | dict[str, Estimate] | None: None if the transmutation
`Any`	was run without `data_uncertainty`.

Source code in `yani/_core/__init__.pyi`

```
3759  def get_contact_dose_uncertainty(self, material_id: builtins.int, step:
3760  builtins.int, *, dose_quantity: builtins.str = 'absorbed-air', build_up:
3761  builtins.float = 2.0, by_nuclide: builtins.bool = False) -> typing.Any:
3762      r"""
3763      Contact dose rate at one timestep, with the nuclear-data spread on it.
3764
3765      The one derived quantity here that is **not** linear in the atom
3766      densities. The material attenuates its own decay photons, so the
3767      emitters sit in the numerator and the whole inventory in the
3768      denominator: a replica that makes more of an emitter also absorbs more
3769      of it. Scaling the nominal dose by the density spread would report the
3770      activity's uncertainty, which is wrong by the whole value; evaluating
3771      per replica gets the cancellation for free.
3772
3773      Needs no ``volume``, unlike the other three, because the estimate takes
3774      the material for a half-space.
3775
3776      The band is the spread of the replicas' inventories (each with its own
3777      half-lives when the ``"half_life"`` source is on), and of their decay
3778      photon line intensities and energies and continuum normalisations when
3779      the ``"decay_photon_lines"`` source is on. The photon attenuation
3780      (XCOM), air energy absorption (NIST SRD 126), ICRP-116 dose
3781      coefficients and the build-up factor are held at their nominal values
3782      and contribute nothing to it.
3783
3784      With the ``"decay_photon_lines"`` source on, the band is a range:
3785      ``Estimate.std_dev`` takes each nuclide's photon intensities as
3786      independent where the decay data state no correlation, and
3787      ``Estimate.std_dev_correlated`` as fully correlated (the lines of a
3788      spectrum, its normalisation and lines, and its gamma and x-ray
3789      spectra). ENDF/B-VIII.1 folds each spectrum's normalisation sigma into
3790      every line's, so a multi-line emitter's range there is wide;
3791      ``get_data_uncertainty_info`` names those spectra under
3792      ``decay_photon_spectra_folded``.
3793
3794      Args:
3795          material_id: Material ID number.
3796          step: Timestep index (0 = initial composition). As in
3797              ``get_activity_uncertainty``, step 0 has a spread whenever the
3798              ``"half_life"`` or ``"decay_photon_lines"`` source is sampled.
3799          dose_quantity (str): ```'absorbed-air'``` (Gy/h, the default) or
3800              ``'effective'`` (Sv/h), as ``Material.contact_dose`` takes them.
3801          build_up (float): Build-up factor, a plain multiplier on the answer.
3802          by_nuclide (bool): Return a ``dict[str, Estimate]`` instead of one
3803              ``Estimate`` for the total.
3804
3805      Returns:
3806          Estimate | dict[str, Estimate] | None: None if the transmutation
          was run without ``data_uncertainty``.
      """
```

`meth` get_decay_photon_spectrum_uncertainty

```
get_decay_photon_spectrum_uncertainty(material_id: int, step: int) ->
Optional[list[LineEstimate]]
```

The decay-photon spectrum at one timestep, line by line, with the nuclear-data spread on each emission rate.

Ascending in energy and coincident lines summed, exactly as `Material.decay_photon_spectrum` returns them, and over the union of the lines the nominal run and every replica emit. A line a replica does not emit counts as a zero in it -- the same rule the densities follow, and the only one under which two lines' spreads are taken over the same sample -- and `LineEstimate.emitting` reports how many replicas emitted it, which is what the zero-fill would otherwise hide. Lines only, as there: a photon continuum is not a line and is not reported here.

The band is the spread of the replicas' inventories (each with its own half-lives when the `"half_life"` source is on), and of each line's intensity per decay when the `"decay_photon_lines"` source is on. That source draws each line's energy too, so lines are matched across replicas on their nominal energy, and `LineEstimate.energy_std_dev` gives the spread of the energy drawn. A line's rate spread is a range, `LineEstimate.std_dev` to `LineEstimate.std_dev_correlated`, for the reason `get_contact_dose_uncertainty` gives.

```
>>> lines = results.get_decay_photon_spectrum_uncertainty(mid, step)
>>> [(l.energy, l.nominal, l.std_dev) for l in lines[:2]]
```

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition). As in `get_activity_uncertainty`, step 0 has a spread whenever the `"half_life"` or `"decay_photon_lines"` source is sampled.	*required*

Returns:

Type	Description
`Optional[list[LineEstimate]]`	list[LineEstimate] | None: None if the transmutation was run without
`Optional[list[LineEstimate]]`	`data_uncertainty`.

Raises:

Type	Description
`ValueError`	if the material has no `volume` in cm^3.

Source code in `yani/_core/__init__.pyi`

```
def get_decay_photon_spectrum_uncertainty(self, material_id: builtins.int, step:
builtins.int) -> typing.Optional[builtins.list[LineEstimate]]:
    r"""
    The decay-photon spectrum at one timestep, line by line, with the
    nuclear-data spread on each emission rate.

    Ascending in energy and coincident lines summed, exactly as
    ``Material.decay_photon_spectrum`` returns them, and over the union of
    the lines the nominal run and every replica emit. A line a replica does
    not emit counts as a zero in it -- the same rule the densities follow,
    and the only one under which two lines' spreads are taken over the same
    sample -- and ``LineEstimate.emitting`` reports how many replicas
    emitted it, which is what the zero-fill would otherwise hide. Lines
    only, as there: a photon continuum is not a line and is not reported
    here.

    The band is the spread of the replicas' inventories (each with its own
    half-lives when the ``"half_life"`` source is on), and of each line's
    intensity per decay when the ``"decay_photon_lines"`` source is on.
    That source draws each line's energy too, so lines are matched across
    replicas on their nominal energy, and ``LineEstimate.energy_std_dev``
    gives the spread of the energy drawn. A line's rate spread is a range,
    ``LineEstimate.std_dev`` to ``LineEstimate.std_dev_correlated``, for the
    reason ``get_contact_dose_uncertainty`` gives.

        >>> lines = results.get_decay_photon_spectrum_uncertainty(mid, step)
        >>> [(l.energy, l.nominal, l.std_dev) for l in lines[:2]]

    Args:
        material_id: Material ID number.
        step: Timestep index (0 = initial composition). As in
            ``get_activity_uncertainty``, step 0 has a spread whenever the
            ``"half_life"`` or ``"decay_photon_lines"`` source is sampled.

    Returns:
        list[LineEstimate] | None: None if the transmutation was run without
        ``data_uncertainty``.

    Raises:
        ValueError: if the material has no ``volume`` in cm^3.
    """
```

`meth` get_gas_production

```
get_gas_production(material_id: int, *, produced: bool = True) ->
Optional[dict[str, list[float]]]
```

Hydrogen and helium gas production in appm, at every time point.

appm is gas atoms per million **initial** atoms of the material, so the denominator stays fixed as the material transmutes. The gas is what the inventory already holds: H1, H2, H3, He3 and He4 emitted by reactions and by decays, so tritium decaying to He3 during a cooldown shows up as He3 there.

```
>>> gas = results.get_gas_production(material_id=mid)
>>> gas["He4"][-1], gas["H"][-1]
```

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`produced`	`bool`	Subtract the gas the material started with (water, polymers, lithium compounds), the default, so index 0 is zero and each value is what the schedule made by then. A nuclide consumed faster than it is made reads negative, as H1 in water can through H1(n,gamma)H2. `False` gives the gas present, starting inventory included.	`True`

Returns:

Type	Description
`Optional[dict[str, list[float]]]`	dict[str, list[float]] | None: appm keyed `"H1"`, `"H2"`,
`Optional[dict[str, list[float]]]`	`"H3"`, `"He3"`, `"He4"` and the totals `"H"` (H1 + H2 + H3)
`Optional[dict[str, list[float]]]`	and `"He"` (He3 + He4). Each list is parallel to `times`, as
`Optional[dict[str, list[float]]]`	`get_nuclide_evolution` is: index 0 is the initial composition,
`Optional[dict[str, list[float]]]`	index i is after step i. None if the material is not in the results.

Raises:

Type	Description
`ValueError`	if the chain the solve used has no entry for one of the five gas nuclides. The solve follows an emitted particle only when the chain has it, so that gas was dropped and a zero would be wrong rather than measured. The message names the missing nuclides.

Source code in `yani/_core/__init__.pyi`

```
3847  def get_gas_production(self, material_id: builtins.int, *, produced: builtins.bool
3848  = True) -> typing.Optional[builtins.dict[builtins.str,
3849  builtins.list[builtins.float]]]:
3850      r"""
3851      Hydrogen and helium gas production in appm, at every time point.
3852
3853      appm is gas atoms per million **initial** atoms of the material, so the
3854      denominator stays fixed as the material transmutes. The gas is what the
3855      inventory already holds: H1, H2, H3, He3 and He4 emitted by reactions
3856      and by decays, so tritium decaying to He3 during a cooldown shows up as
3857      He3 there.
3858
3859          >>> gas = results.get_gas_production(material_id=mid)
3860          >>> gas["He4"][-1], gas["H"][-1]
3861
3862      Args:
3863          material_id: Material ID number.
3864          produced (bool): Subtract the gas the material started with (water,
3865              polymers, lithium compounds), the default, so index 0 is zero
3866              and each value is what the schedule made by then. A nuclide
3867              consumed faster than it is made reads negative, as H1 in water
3868              can through H1(n,gamma)H2. ``False`` gives the gas present,
3869              starting inventory included.
3870
3871      Returns:
3872          dict[str, list[float]] | None: appm keyed ``"H1"``, ``"H2"``,
3873          ``"H3"``, ``"He3"``, ``"He4"`` and the totals ``"H"`` (H1 + H2 + H3)
3874          and ``"He"`` (He3 + He4). Each list is parallel to ``times``, as
3875          ``get_nuclide_evolution`` is: index 0 is the initial composition,
3876          index i is after step i. None if the material is not in the results.
3877
3878      Raises:
3879          ValueError: if the chain the solve used has no entry for one of the
3880              five gas nuclides. The solve follows an emitted particle only
3881              when the chain has it, so that gas was dropped and a zero would
3882              be wrong rather than measured. The message names the missing
                nuclides.
      """
```

`meth` get_gas_production_uncertainty

```
get_gas_production_uncertainty(material_id: int, step: int, *, produced: bool =
True) -> Optional[dict[str, Estimate]]
```

Gas production in appm at one timestep, with the nuclear-data spread on it.

See `get_gas_production` for the quantity. Evaluated on every replica's inventory against the one initial inventory, which is an input and the same in each, and the totals `"H"` and `"He"` are summed within a replica before the spread is taken, as `get_activity_uncertainty` does.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition, which has no spread).	*required*
`produced`	`bool`	As in `get_gas_production`.	`True`

Returns:

Type	Description
`Optional[dict[str, Estimate]]`	dict[str, Estimate] | None: keyed as `get_gas_production`; None if
`Optional[dict[str, Estimate]]`	the transmutation was run without `data_uncertainty`.

Raises:

Type	Description
`ValueError`	as `get_gas_production`, or if there is no such step.

Source code in `yani/_core/__init__.pyi`

```
3883  def get_gas_production_uncertainty(self, material_id: builtins.int, step:
3884  builtins.int, *, produced: builtins.bool = True) ->
3885  typing.Optional[builtins.dict[builtins.str, Estimate]]:
3886      r"""
3887      Gas production in appm at one timestep, with the nuclear-data spread
3888      on it.
3889
3890      See ``get_gas_production`` for the quantity. Evaluated on every
3891      replica's inventory against the one initial inventory, which is an
3892      input and the same in each, and the totals ``"H"`` and ``"He"`` are
3893      summed within a replica before the spread is taken, as
3894      ``get_activity_uncertainty`` does.
3895
3896      Args:
3897          material_id: Material ID number.
3898          step: Timestep index (0 = initial composition, which has no spread).
3899          produced (bool): As in ``get_gas_production``.
3900
3901      Returns:
3902          dict[str, Estimate] | None: keyed as ``get_gas_production``; None if
3903          the transmutation was run without ``data_uncertainty``.
3904
3905      Raises:
          ValueError: as ``get_gas_production``, or if there is no such step.
      """
```

`meth` get_data_uncertainty_info

```
get_data_uncertainty_info(material_id: int) -> Optional[dict]
```

What the nuclear-data uncertainty covered for one material, and what it did not.

`None` when the transmutation was run without `data_uncertainty`, or the material is not in the results. Otherwise a dict whose job is to make gaps visible rather than let them read as confidence:

- `perturbed` / `no_covariance_data`: which nuclides had usable MF=33 covariance and which had none.
- `rate_fraction_covered_total`: the per-channel shares below, averaged with each channel weighted by the production it drove (the rate this run used times parent density): the share of the production driven from energies where a covariance states a nonzero variance. `None` on a decay-only schedule; on a transport run, where the shares are of the dilute rate over the tally spectrum and the covered share of the tallied production is not computed; and under the `1/E` within-group weight (Rust API only), whose shares are not exact where a covariance edge cuts a group. `None` too when a channel is listed in `partials_above_rate` or `partials_below_rate`, whose rate is not the one its share is of: the `(n,n')` of a nuclide with a metastable, whose rate is the MF=10 production of the metastables while its covariance is MT 4's. Read this before any sigma here. It is a different and much sharper question than how many nuclides carry MF=33: an evaluation can state covariance for every isotope in the

material and none for the channel making the product of interest, and the count then reads as full coverage while the ensemble perturbs almost nothing.

- `rate_fraction_covered`: per nuclide and channel, the share of the reaction rate from energies where the evaluation states a nonzero variance for it, the rate being the dilute one on a dilute run and the shielded one on a self-shielded run. Below one means part of the rate carries no stated uncertainty and dilutes the sigma; on a transport run it is a share of the dilute rate over the tally spectrum, and the dilution applied differs from it. An interval the covariance grid spans with a variance of zero counts as uncovered: ENDF/B-VIII.1 W186 `(n,gamma)` states zero from 1e-5 eV to 10 keV, where nearly all of its capture rate is. Every consumed self-covariance block counts where it states a nonzero variance, relative (LB=1 to 6), absolute (LB=0) and short-range (LB=8) alike. Exact under the default flat within-group weight; under the `1/E` weight (Rust API only) the rate of a group a covariance edge cuts is split by energy width, not lethargy, so the share is off there.
- `partials_above_rate`: per nuclide and channel, where the partial rates the covariance was weighted with, zero variance intervals included, add up to more than the rate it was divided by, their ratio to it. Each entry is a channel whose sigma is overstated. Three known causes: a tallied rate on a transport run, which the transport can self-shield within its bins, against partials weighted flat within each bin; a grafted `(n,n')`, whose rate is the metastables' MF=10 production while the partials are MT 4's; and the `1/E` within-group weight (Rust API only) with a covariance edge inside a group. The share in `rate_fraction_covered` is of the fold's own rate, not of the listed rate, and under the `1/E` weight it is off as well wherever an edge cuts a group. On a channel derived through an NC block, the partials of the reactions the block names are checked the same way, and the check is also that they add up to the one it derives over the block's range; a sum above it lands here.
- `partials_below_rate`: keyed the same way, where a covariance grid spans the whole flux range and its partial rates add up to less than the rate, their ratio to it: a channel whose sigma is understated. The `1/E` weight gives one for a reaction falling with energy when a covariance edge cuts a group. A grid that stops short of the flux range cannot be checked from below, since rate from outside it rightly leaves its partials short. A derived channel whose NC block names reactions adding up to less than the one it derives lands here too: ENDF/B-VIII.1 O16 `(n,d)` above 20 MeV, whose cross section holds MT 660 to 669 while the block names 650 to 659.
- `derived_opposing_uncorrelated`: keyed the same way, for a channel derived through an NC block whose terms name two reactions with opposite signs, each with a variance of its own, and no covariance between them: the `[a, b]` pairs. The absent block is read as zero, since ENDF-102 33.3.2 a.1 lets a tape leave a zero covariance unstated, and with opposing signs that reading sets the sigma. FENDL-3.2d and TENDL-2017 H2 `(n,2n)` is `σ_1 - σ_2 - σ_102` and folds to about 22% at 14 MeV and thousands of percent near threshold, the tape's literal statement. Counted in `has_gaps`.
- `lumped_covariance_not_assignable`: keyed `"Nuclide MT852"`, a lumped reaction (MT 851-870) with several components, to their kinds (`"MT91"` for one that is not a channel). ENDF-102 33.2.3 states its covariance for the sum of the components and for none of them. A lump with

one component is that component, and its covariance is folded as the component's. A lump an LTY=0 block names is folded through that derivation, with its cross section the sum of its components': ENDF/B-VIII.1 and TENDL-2017 U235 and U238 MT 4 is MT 51 plus MT 851. A component with no covariance of its own takes the lump's relative perturbation, so the components of one lump move together and their sum moves as the lump's covariance states, as NJOY's ERRORR leaves a lump and SANDY applies it: ENDF/B-VIII.1, FENDL-3.2d and JEFF-4.0 W180 to W186 give `(n,2n)` only as MT 852, the sum of MT 16 and 41. 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 649. Listed here is only a lump with a block that is not relative, absolute or short-range, which states the sum's spread in barns and nothing of how it splits, where the fold reaches a component; no library yani reads has one. Counted in `has_gaps`.

- `unsupported_layouts`: covariance blocks that were present but not consumed, counted once per spectrum, so a run over several spectra counts the same block once for each.
- `skipped_nc`: per nuclide, NC blocks (a covariance derived from other reactions) that could not be derived. An LTY=0 block is derived from the NI covariances of the reactions it names, cross blocks included, over its own energy range: ENDF/B-VIII.1 O16 `(n,p)` is stated only that way. Left here are LTY 1 to 4 (a covariance from a measurement's ratio to another evaluation's standard, which no library yani reads states), a block in a cross-reaction subsection, one whose list of reactions is empty or does not match its coefficients, one whose own energy range is empty, one naming a reaction with no cross section, and one met only circularly.
- `skipped_cross_material`: per nuclide, blocks on a reaction the fold reaches (a channel, or one a channel is derived from) that correlate it with another evaluation, not applied: the evaluation they name is not in the run (or carries no covariance, so its MAT is not known), or one of the two reactions has no relative covariance to correlate. A block naming the nuclide's own MAT is its own evaluation and is folded. `applied_cross_material`: the same count for the blocks applied, whose two nuclides are drawn jointly, one draw per replica shared by every material of the run. `skipped_other_file`: blocks whose partner is not a cross section, which no library yani reads states.
- `mirrored_disagree`: keyed `"Nuclide (n,a) (n,b)"`, where a pair stored in both orientations has copies that are not each other's transpose, the largest difference relative to the largest entry. The copy in the lower MT's section is the one folded.
- `malformed_blocks`: covariance blocks not consumed because they break ENDF-102's rules for their layout: arrays that disagree with their declared sizes, an LB=0 to 2 block carrying a second energy table, an LB=3 or 4 block without one or whose tables share no energy range, or an LB=8 variance stated between two reactions.
- `covariance_source`: where each perturbed nuclide's covariance came from, `"<library>, MAT <n>"`, the library being the one its data folder records (`"unknown"` when none is recorded).
- `covariance_warnings`: perturbed nuclides whose library documents a problem with this covariance, each with what it says and where: every FENDL-3.2 covariance, which its own paper says should not be used, and the ENDF/B-VIII.1 evaluations its release paper names (Fe, Cr

covariances reused from VIII.0; Cu with no fast-range covariance; Ta, W, Pb resolved-resonance covariance flagged as too low; the Ta181 unresolved covariance overwritten). The evaluation is still sampled as it is; any warning makes `has_gaps` true.

- `covariance_repaired`: nuclides the material can populate (bounded at or above the solver's density floor over the schedule at nominal rates; a replica's rates can sit above them) whose evaluated cell covariance was not positive semi-definite past round-off, with a channel a draw can move (a positive rate on a spectrum the schedule irradiates with). Past round-off means the correlation matrix of the cells has an eigenvalue below `-m * 1e-12` (`m` the number of cells with a positive stated variance), or a cell is stated with a negative variance, or a zero one and a covariance to another cell. The correlation matrix is replaced by the nearest correlation matrix and rescaled by the evaluated sigmas, so every cell keeps its evaluated sigma and only correlations move (a cell stated at zero or negative variance is held at nominal); a channel folding several cells can still be sampled at a sigma other than its evaluation's, either way, and any makes `has_gaps` true. `covariance_repairs` gives one dict 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` (the largest and the Frobenius change of that correlation matrix), `cells` (in the coupled blocks repaired), `held_cells`, `converged` and, per channel keyed by kind, `evaluated_variance` (the folded diagonal as stated, which can be negative), `evaluated_sigma` (`None` when that variance is negative) and `sampled_sigma`. A repair of a nuclide outside the populated bound has no dict; `covariance_repaired_outside_bound` names those with a channel a draw can move. The bound holds at nominal rates only and a replica's rates can populate them, so any also makes `has_gaps` true.
- `worst_sigma_change`: the largest `|sampled / evaluated sigma - 1|` over the repaired channels of populated nuclides with a positive rate on a spectrum the schedule irradiates with, `float('inf')` when a repair gave a spread to a channel whose stated variance is zero or negative. `rate_weighted_sigma_change` is the weighted mean of `|sampled / evaluated sigma - 1|` over every sampled channel of a populated nuclide, each weighted by its unit-flux rate times its spectrum's fluence in the schedule times its parent's initial density, so it covers first-generation reactions only (a produced nuclide carries no weight), shows the decomposition's round-off on matrices that needed no repair, and is `None` when no weighted channel has an evaluated sigma. Both can be `float('inf')`, which strict JSON does not accept.
- `sigma_at_least_one` / `sigma_at_least_ten`: sampled channels of populated nuclides with a positive rate on a spectrum the schedule irradiates with, keyed `"Nuclide (n,x)"`, whose folded relative sigma as evaluated, before any repair, is at least one or ten. At that width the answer depends on the lognormal chosen to carry the evaluation's two moments, not on the evaluation alone. `sigma_at_least_one_outside_bound` is the same for nuclides outside the populated bound, whose wide channels a replica's draw can take past it; the ten-or-more subset reads off its values.
- `lognormal_not_carried`: nuclides whose evaluated relative covariance is not a lognormal's, keyed by nuclide, each with `cells` (cells whose sampled sigma or correlation differs from the

evaluated one), `largest_sigma_change` (the largest `|sampled / evaluated sigma - 1|`), `largest_correlation_change` and `log_space_repair` (`None`, or a dict with the keys of a repair above, of the log-space correlation matrix). Two fully correlated cells with different sigmas, or an anticorrelation with `1 + C <= 0`, are not, and the nearest lognormal is sampled: where the log-space covariance is not PSD its correlation matrix is replaced by the nearest correlation matrix, which keeps every sigma. A property of the distribution rather than a defect of the data, so not a gap. `flux_lognormal_not_carried` is the same for a stated flux covariance, keyed by spectrum index, whose `log_space_repair` is always `None`: a flux covariance's log-space negative eigenvalues are clipped.

- `cross_material_adjusted`: keyed by a group of nuclides drawn jointly through cross-material covariance, their names joined by `+` (`"U235+U238"`), where the covariance stated between their interval averages is not admissible beside each one's own and the closest admissible one was drawn instead, each own covariance kept exactly and the correlations between the averages fitted to the stated ones (each clipped to one first) in the Frobenius norm: `relative_change` (`||X' - X|| / ||X||` of the drawn against the stated log-space covariance between the averages), `least_relative_change` (the same with each stated entry only clipped to a correlation of one: no admissible draw goes below it), `largest_stated_correlation` (past one where an entry alone is inadmissible), `largest_correlation_change` (the largest move of a correlation from its stated value clipped to one), `iterations` and `converged` (the fit's; its result is admissible either way). ENDF/B-VIII.1 U235 and U238 fission read about 0.31. Counted in `has_gaps`.
- `rates_sampled`: cross-section rate draws made, each read off one draw of the nuclide's cross sections. `rates_floored` counts those that came out negative and were floored at zero, which only a channel subtracting reactions or reading an additive (absolute or short-range) covariance can.
- `cross_section_points_floored`: with `method="coupled"`, the cross-section grid values a replica's draw took below zero and that were floored at zero, over every replica.
- `half_lives_perturbed` / `no_half_life_uncertainty`: with the `"half_life"` source, which reachable unstable nuclides had their half-life sampled and which state no sigma to sample from. `half_life_uncertainty_not_carried` names those whose stated sigma no draw can carry (not finite, or not finite relative to the half-life), held at nominal and counted as a gap. `half_lives_sampled` counts the draws made. Each is a lognormal matched to the evaluation's mean and sigma, so none can go non-positive and none is floored.
- `decay_energies_perturbed` / `no_decay_energy_uncertainty`: the same for the `"decay_energy"` source, drawn per nuclide as a lognormal with the stated mean and sigma, per component where the data splits it. `decay_energy_uncertainty_not_carried` names those with a sigma stated on a zero energy, or not finite, which no draw can carry; that energy is held at nominal and counted as a gap.
- `decay_photon_lines_perturbed` / `no_decay_photon_line_uncertainty`: the same for the `"decay_photon_lines"` source, over the reachable unstable nuclides with decay photon data. `decay_photon_line_uncertainty_not_carried` names those with a sigma stated on a zero value, or not finite, which no draw can carry; that value is held at nominal and counted as a gap.

`decay_photon_spectra_folded` maps each perturbed nuclide with a spectrum written the ENDF/B way (a normalisation of 1 with no sigma, its sigma folded into every line's dRI) to the radiation of each such spectrum (`"gamma"`, `"xray"`). How much of those dRI the lines share is not stated, so they are where most of the range between a photon output's `std_dev` and `std_dev_correlated` comes from.

- `fission_yields_perturbed` / `no_fission_yield_uncertainty`: the same for the `"fission_yield"` source, over the reachable parents the run gives a fission rate (one that never fissions is not drawn). `fission_yield_uncertainty_not_carried` names those with a DY on a zero yield, or not finite, which is held while the parent's other yields are drawn. `fission_yields_mapping_mismatch` names those whose tape yields, named and summed by the converter's rule, do not give back the yields the solver reads; they are held rather than drawn through a mapping their yields were not built with. Each is counted as a gap.
- `decay_branchings_perturbed`: with the `"decay_branching"` source, the reachable multi-mode parents whose split was sampled. Held at their evaluated ratios, each a gap: `no_decay_branching_uncertainty` (no mode states a sigma, or one does beside two or more that do not) and `decay_branchings_too_wide` (every mode that could move states a sigma no ratio with its mean can have). Sampled but short of the stated sigmas, each a gap too: `decay_branchings_partly_held` (one or more modes held at their ratios, for such a sigma or for stating none) and `decay_branchings_sigma_adjusted` (a mode drawn more than 0.1% from its stated sigma: the largest exceeds the sum of the others', or the complement's needs a correlation non-negative ratios cannot carry). `decay_branchings_floored` / `decay_branchings_sampled` count draws whose complement was pulled back above zero and draws made.
- `isomer_branchings_perturbed`: with the `"isomer_branching"` source, the reachable reactions, as `"Parent reaction"`, whose isomeric split was drawn from the MF=40 covariance of at least one of their final states. `no_isomer_branching_uncertainty` lists those whose split can move but whose evaluation states no MF=40 on any state the run folds, held at their nominal shares, a gap. `isomer_branching_blocks_not_used` gives, per parent, the MF=40 blocks not used, counted by why, a gap. `isomer_branching_covariance_repaired` and `isomer_branching_lognormal_not_carried` are the MF=40 matrices repaired to the nearest correlation matrix (a gap) and those not a lognormal's (not a gap), with the same entries as `covariance_repairs` and `lognormal_not_carried`. `isomer_production_sigma` gives, as `"Parent reaction Product"`, the relative sigma MF=40 states for the product's production, folded against the spectrum (the largest over the run's spectra). `isomer_branchings_sampled` counts the draws, one per drawn parent per replica. `isomer_branchings_clipped` gives, as `"Parent reaction Product"`, how many list draws (one per replica per spectrum) scaled the product's drawn production back to the drawn reaction total, the stated productions of its list having come out above it; a product never clipped is absent.
- `damage_energy`: with `displacement_damage=True` and the `"cross_sections"` source, per nuclide of the starting composition `{"explained": ..., "sigma": ...}`: the share of its stored MT=444 rate its reactions' deposits reproduce, the part a draw moves (the rest, fission's

share above all, is held, a gap), and the relative sigma of its drawn damage-energy rate, each weighted over the spectra by fluence.

- `statistical_rates`: with the `"statistical"` source on a transport run, how many tallied rates were sampled from their covariance; `statistical_floored` / `statistical_sampled` count draws that came out negative and were floored.
- `resonance_parameters`: per nuclide with resonance parameters (MF=2 and MF=32) and a rate in this run, how its resonance-range uncertainty was sampled. `method` is `"parameters sampled"` (drawn per replica and the cross sections rebuilt from them, on a transport run against the flux it tallied through the resonances) or `"first-order rows"` (the MF=32 rows of `covariance.arrow`), `reason` why the parameters were not sampled (`None` where they were), and `ranges` one sampler report per sampled range: `isotope` and `range` indices, the `gaussian`, `lognormal` and `held` parameter counts, `zero_mean_widths` (widths stated with a zero mean and a nonzero sigma, held at zero) and `negative_mean_widths` (drawn as signed), each with `index`, `location`, `quantity`, `value` and `sigma`, `zero_variance_with_covariance`, `unattainable_pairs`, and `stated_repair` / `transformed_repair` (the nearest-correlation repair of the stated and of the log-space matrix, each with `lambda_min`, `frobenius_change`, `max_change`, `parameters`, `iterations` and `converged`, `None` where none was needed). How far the draws' parameter correlations are from the evaluated ones, in the parameters themselves after both repairs and the lognormal transform, reads off `largest_correlation_change` (the largest change of one correlation) and `correlation_frobenius_change`, per range and per nuclide over its ranges; every drawn parameter keeps its evaluated mean and sigma, so that is the whole of the difference in the first two moments. Widths stay lognormal, so a pair no lognormal carries (`unattainable_pairs`) is where it is largest. Empty on a transport run, which keeps the rows.
- `not_perturbed`: every input this run held at its nominal value, such as any MF=32 resonance-parameter covariance neither sampled nor in the library's `covariance.arrow`, the photon and dose data, the material composition, any source switched off, and, where they applied, the self-shielding correction, the flux's response to a perturbed cross section on a transport run, and the per-branch decay emission of a parent whose branching was drawn.
- `samples` / `converged` / `convergence` / `hit_cap` / `unconverged`: how many replicas ran; whether every tracked sigma reached the target, `convergence`, for the standard error of a sigma relative to it; whether the 1024-replica cap stopped the run first; and, worst first, every sigma that missed, as dicts of `output` (`"density"`, `"activity"`, `"decay_heat"` or `"decay_photon_rate"`), `nuclide` (`None` for a total), `step` and the `relative_standard_error` it reached. Listed with a fixed `samples` too, where `hit_cap` is never set.
- `noise_removed` / `noisy`: whether the Monte Carlo noise in the replicas' spread was measured and removed from every sigma (with transport replica weights, each replica is solved a second time from the even half of the transport's histories, see `get_nuclide_uncertainty_noise`), and, the largest share first, every tracked sigma whose removed noise variance exceeded a quarter of the variance left, as dicts of `output`, `nuclide`, `step`, `replica_spread`, `noise` and `std_dev` (the sigma reported). Those sigmas would sharpen most with more transport histories.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Source code in `yani/_core/__init__.pyi`

```
def get_data_uncertainty_info(self, material_id: builtins.int) ->
typing.Optional[dict]:
    r"""
    What the nuclear-data uncertainty covered for one material, and what it
    did not.

    ``None`` when the transmutation was run without ``data_uncertainty``, or
    the material is not in the results.
    Otherwise a dict whose job is to make gaps visible rather than let them
    read as confidence:

    - ``perturbed`` / ``no_covariance_data``: which nuclides had usable
      MF=33 covariance and which had none.
    - ``rate_fraction_covered_total``: the per-channel shares below,
      averaged with each channel weighted by the production it drove (the
      rate this run used times parent density): the share of the
      production driven from energies where a covariance states a nonzero
      variance. ``None`` on a decay-only schedule; on a transport run,
      where the shares are of the dilute rate over the tally spectrum and
      the covered share of the tallied production is not computed; and
      under the ``1/E`` within-group weight (Rust API only), whose shares
      are not exact where a covariance edge cuts a group. ``None`` too when
      a channel is listed in ``partials_above_rate`` or
      ``partials_below_rate``, whose rate is not the one its share is of:
      the ``(n,n')`` of a nuclide with a metastable, whose rate is the
      MF=10 production of the metastables while its covariance is MT 4's.
      Read this before any sigma here. It is a different and much sharper
      question than how many nuclides carry MF=33: an evaluation can state
      covariance for every isotope in the material and none for the
      channel making the product of interest, and the count then reads as
      full coverage while the ensemble perturbs almost nothing.
    - ``rate_fraction_covered``: per nuclide and channel, the share of the
      reaction rate from energies where the evaluation states a nonzero
      variance for it, the rate being the dilute one on a dilute run and
      the shielded one on a self-shielded run. Below one means part of the
      rate carries no stated uncertainty and dilutes the sigma; on a
      transport run it is a share of the dilute rate over the tally
      spectrum, and the dilution applied differs from it. An
      interval the covariance grid spans with a variance of zero counts as
      uncovered: ENDF/B-VIII.1 W186 ``(n,gamma)`` states zero from 1e-5 eV
      to 10 keV, where nearly all of its capture rate is. Every consumed
      self-covariance block counts where it states a nonzero variance,
      relative (LB=1 to 6), absolute (LB=0) and short-range (LB=8) alike.
      Exact under the default flat within-group weight; under the ``1/E``
      weight (Rust API only) the rate of a group a covariance edge cuts is
      split by energy width, not lethargy, so the share is off there.
    - ``partials_above_rate``: per nuclide and channel, where the partial
      rates the covariance was weighted with, zero variance intervals
      included, add up to more than the rate it was divided by, their
      ratio to it. Each entry is a channel whose sigma is overstated. Three
      known causes: a tallied rate on a transport run, which the transport
      can self-shield within its bins, against partials weighted flat
      within each bin; a grafted ``(n,n')``, whose rate is the metastables'
      MF=10 production while the partials are MT 4's; and the ``1/E``
      within-group weight (Rust API only) with a covariance edge inside a
      group. The share in ``rate_fraction_covered`` is of the fold's own
      rate, not of the listed rate, and under the ``1/E`` weight it is off
      as well wherever an edge cuts a group. On a channel derived through
      an NC block, the partials of the reactions the block names are
      checked the same way, and the check is also that they add up to the
      one it derives over the block's range; a sum above it lands here.
    - ``partials_below_rate``: keyed the same way, where a covariance grid
      spans the whole flux range and its partial rates add up to less than
      the rate, their ratio to it: a channel whose sigma is understated.
```

```
      The ``1/E`` weight gives one for a reaction falling with energy when
      a covariance edge cuts a group. A grid that stops short of the flux
      range cannot be checked from below, since rate from outside it
      rightly leaves its partials short. A derived channel whose NC block
      names reactions adding up to less than the one it derives lands here
      too: ENDF/B-VIII.1 O16 ``(n,d)`` above 20 MeV, whose cross section
      holds MT 660 to 669 while the block names 650 to 659.
    - ``derived_opposing_uncorrelated``: keyed the same way, for a channel
      derived through an NC block whose terms name two reactions with
      opposite signs, each with a variance of its own, and no covariance
      between them: the ``[a, b]`` pairs. The absent block is read as zero,
      since ENDF-102 33.3.2 a.1 lets a tape leave a zero covariance
      unstated, and with opposing signs that reading sets the sigma.
      FENDL-3.2d and TENDL-2017 H2 ``(n,2n)`` is ``σ_1 - σ_2 - σ_102`` and
      folds to about 22% at 14 MeV and thousands of percent near
      threshold, the tape's literal statement. Counted in ``has_gaps``.
    - ``lumped_covariance_not_assignable``: keyed ``"Nuclide MT852"``, a
      lumped reaction (MT 851-870) with several components, to their kinds
      (``"MT91"`` for one that is not a channel). ENDF-102 33.2.3 states
      its covariance for the sum of the components and for none of them. A
      lump with one component is that component, and its covariance is
      folded as the component's. A lump an LTY=0 block names is folded
      through that derivation, with its cross section the sum of its
      components': ENDF/B-VIII.1 and TENDL-2017 U235 and U238 MT 4 is MT 51
      plus MT 851. A component with no covariance of its own takes the
      lump's relative perturbation, so the components of one lump move
      together and their sum moves as the lump's covariance states, as
      NJOY's ERRORR leaves a lump and SANDY applies it: ENDF/B-VIII.1,
      FENDL-3.2d and JEFF-4.0 W180 to W186 give ``(n,2n)`` only as MT 852,
      the sum of MT 16 and 41. 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 649. Listed
      here is only a lump with a block that is not relative, absolute or
      short-range, which states the sum's spread in barns and nothing of how
      it splits, where the fold reaches a component; no library yani reads
      has one. Counted in ``has_gaps``.
    - ``unsupported_layouts``: covariance blocks that were present but not
      consumed, counted once per spectrum, so a run over several spectra
      counts the same block once for each.
    - ``skipped_nc``: per nuclide, NC blocks (a covariance derived from
      other reactions) that could not be derived. An LTY=0 block is
      derived from the NI covariances of the reactions it names, cross
      blocks included, over its own energy range: ENDF/B-VIII.1 O16
      ``(n,p)`` is stated only that way. Left here are LTY 1 to 4 (a
      covariance from a measurement's ratio to another evaluation's
      standard, which no library yani reads states), a block in a
      cross-reaction subsection, one whose list of reactions is empty or
      does not match its coefficients, one whose own energy range is empty,
      one naming a reaction with no cross section, and one met only
      circularly.
    - ``skipped_cross_material``: per nuclide, blocks on a reaction the
      fold reaches (a channel, or one a channel is derived from) that
      correlate it with another evaluation, not applied: the evaluation
      they name is not in the run (or carries no covariance, so its MAT is
      not known), or one of the two reactions has no relative covariance to
      correlate. A block naming the nuclide's own MAT is its own evaluation
      and is folded. ``applied_cross_material``: the same count for the
      blocks applied, whose two nuclides are drawn jointly, one draw per
      replica shared by every material of the run. ``skipped_other_file``:
      blocks whose partner is not a cross section, which no library yani
      reads states.
    - ``mirrored_disagree``: keyed ``"Nuclide (n,a) (n,b)"``, where a pair
      stored in both orientations has copies that are not each other's
      transpose, the largest difference relative to the largest entry.
```

```
        The copy in the lower MT's section is the one folded.
    - ``malformed_blocks``: covariance blocks not consumed because they
      break ENDF-102's rules for their layout: arrays that disagree with
      their declared sizes, an LB=0 to 2 block carrying a second energy
      table, an LB=3 or 4 block without one or whose tables share no
      energy range, or an LB=8 variance stated between two reactions.
    - ``covariance_source``: where each perturbed nuclide's covariance came
      from, ``"<library>, MAT <n>"``, the library being the one its data
      folder records (``"unknown"`` when none is recorded).
    - ``covariance_warnings``: perturbed nuclides whose library documents
      a problem with this covariance, each with what it says and where:
      every FENDL-3.2 covariance, which its own paper says should not be
      used, and the ENDF/B-VIII.1 evaluations its release paper names (Fe,
      Cr covariances reused from VIII.0; Cu with no fast-range covariance;
      Ta, W, Pb resolved-resonance covariance flagged as too low; the Ta181
      unresolved covariance overwritten). The evaluation is still sampled as
      it is; any warning makes ``has_gaps`` true.
    - ``covariance_repaired``: nuclides the material can populate (bounded
      at or above the solver's density floor over the schedule at nominal
      rates; a replica's rates can sit above them) whose evaluated cell
      covariance was not positive semi-definite past round-off, with a
      channel a draw can move (a positive rate on a spectrum the schedule
      irradiates with). Past round-off means the correlation matrix of the
      cells has an eigenvalue below ``-m * 1e-12`` (``m`` the number of
      cells with a positive stated variance), or a cell is stated with a
      negative variance, or a zero one and a covariance to another cell.
      The correlation matrix is replaced by the nearest correlation
      matrix and rescaled by the evaluated sigmas, so every cell keeps its
      evaluated sigma and only correlations move (a cell stated at zero or
      negative variance is held at nominal); a channel folding several
      cells can still be sampled at a sigma other than its evaluation's,
      either way, and any makes ``has_gaps`` true.
      ``covariance_repairs`` gives one dict 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`` (the largest and the Frobenius
      change of that correlation matrix), ``cells`` (in the coupled blocks
      repaired), ``held_cells``, ``converged`` and, per channel keyed by
      kind,
      ``evaluated_variance`` (the folded diagonal as stated, which can be
      negative), ``evaluated_sigma`` (``None`` when that variance is
      negative) and ``sampled_sigma``. A repair of a nuclide outside the
      populated bound has no dict; ``covariance_repaired_outside_bound``
      names those with a channel a draw can move. The bound holds at
      nominal rates only and a replica's rates can populate them, so any
      also makes ``has_gaps`` true.
    - ``worst_sigma_change``: the largest ``|sampled / evaluated sigma -
      1|`` over the repaired channels of populated nuclides with a
      positive rate on a spectrum the schedule irradiates with,
      ``float('inf')`` when a repair gave a spread to a channel whose stated
      variance is zero or negative. ``rate_weighted_sigma_change`` is the
      weighted mean of ``|sampled / evaluated sigma - 1|`` over every
      sampled channel of a populated nuclide, each weighted by its unit-flux
      rate times its spectrum's fluence in the schedule times its parent's
      initial density, so it covers first-generation reactions only (a
      produced nuclide carries no weight), shows the decomposition's
      round-off on matrices that needed no repair, and is ``None`` when no
      weighted channel has an evaluated sigma. Both can be
      ``float('inf')``, which strict JSON does not accept.
    - ``sigma_at_least_one`` / ``sigma_at_least_ten``: sampled channels of
      populated nuclides with a positive rate on a spectrum the schedule
      irradiates with, keyed ``"Nuclide (n,x)"``, whose folded relative
      sigma as evaluated, before any repair, is at least one or ten. At that
```

```
        width the answer depends on the lognormal chosen to carry the
        evaluation's two moments, not on the evaluation alone.
        ``sigma_at_least_one_outside_bound`` is the same for nuclides outside
        the populated bound, whose wide channels a replica's draw can take
        past it; the ten-or-more subset reads off its values.
    - ``lognormal_not_carried``: nuclides whose evaluated relative
      covariance is not a lognormal's, keyed by nuclide, each with
      ``cells`` (cells whose sampled sigma or correlation differs from the
      evaluated one), ``largest_sigma_change`` (the largest
      ``|sampled / evaluated sigma - 1|``), ``largest_correlation_change``
      and ``log_space_repair`` (``None``, or a dict with the keys of a
      repair above, of the log-space correlation matrix).
      Two fully correlated cells with different sigmas, or an
      anticorrelation with ``1 + C <= 0``, are not, and the nearest
      lognormal is sampled: where the log-space covariance is not PSD its
      correlation matrix is replaced by the nearest correlation matrix,
      which keeps every sigma. A property of the distribution rather than a
      defect of the data, so not a gap. ``flux_lognormal_not_carried`` is
      the same for a stated flux covariance, keyed by spectrum index,
      whose ``log_space_repair`` is always ``None``: a flux covariance's
      log-space negative eigenvalues are clipped.
    - ``cross_material_adjusted``: keyed by a group of nuclides drawn jointly
      through cross-material covariance, their names joined by ``+``
      (``"U235+U238"``), where the covariance stated between their interval
      averages is not admissible beside each one's own and the closest
      admissible one was drawn instead, each own covariance kept exactly
      and the correlations between the averages fitted to the stated ones
      (each clipped to one first) in the Frobenius norm:
      ``relative_change`` (``||X' - X|| / ||X||`` of the drawn against the
      stated log-space covariance between the averages),
      ``least_relative_change`` (the same with each stated entry only
      clipped to a correlation of one: no admissible draw goes below it),
      ``largest_stated_correlation`` (past one where an entry alone is
      inadmissible), ``largest_correlation_change`` (the largest move of a
      correlation from its stated value clipped to one), ``iterations``
      and ``converged`` (the fit's; its result is admissible either way).
      ENDF/B-VIII.1 U235 and U238 fission read about 0.31.
      Counted in ``has_gaps``.
    - ``rates_sampled``: cross-section rate draws made, each read off one
      draw of the nuclide's cross sections. ``rates_floored`` counts those
      that came out negative and were floored at zero, which only a channel
      subtracting reactions or reading an additive (absolute or
      short-range) covariance can.
    - ``cross_section_points_floored``: with ``method="coupled"``, the
      cross-section grid values a replica's draw took below zero and that
      were floored at zero, over every replica.
    - ``half_lives_perturbed`` / ``no_half_life_uncertainty``: with the
      ``"half_life"`` source, which reachable unstable nuclides had their
      half-life sampled and which state no sigma to sample from.
      ``half_life_uncertainty_not_carried`` names those whose stated sigma
      no draw can carry (not finite, or not finite relative to the
      half-life), held at nominal and counted as a gap.
      ``half_lives_sampled`` counts the draws made. Each is a lognormal
      matched to the evaluation's mean and sigma, so none can go
      non-positive and none is floored.
    - ``decay_energies_perturbed`` / ``no_decay_energy_uncertainty``: the
      same for the ``"decay_energy"`` source, drawn per nuclide as a
      lognormal with the stated mean and sigma, per component where the
      data splits it. ``decay_energy_uncertainty_not_carried`` names those
      with a sigma stated on a zero energy, or not finite, which no draw
      can carry; that energy is held at nominal and counted as a gap.
    - ``decay_photon_lines_perturbed`` /
      ``no_decay_photon_line_uncertainty``: the same for the
      ``"decay_photon_lines"`` source, over the reachable unstable
```

```
        nuclides with decay photon data.
        ``decay_photon_line_uncertainty_not_carried`` names those with a
        sigma stated on a zero value, or not finite, which no draw can carry;
        that value is held at nominal and counted as a gap.
        ``decay_photon_spectra_folded`` maps each perturbed nuclide with a
        spectrum written the ENDF/B way (a normalisation of 1 with no sigma,
        its sigma folded into every line's dRI) to the radiation of each such
        spectrum (``"gamma"``, ``"xray"``). How much of those dRI the lines
        share is not stated, so they are where most of the range between a
        photon output's ``std_dev`` and ``std_dev_correlated`` comes from.
      - ``fission_yields_perturbed`` / ``no_fission_yield_uncertainty``: the
        same for the ``"fission_yield"`` source, over the reachable
        parents the run gives a fission rate (one that never fissions is
        not drawn). ``fission_yield_uncertainty_not_carried`` names
        those with a DY on a zero yield, or not finite, which is held while
        the parent's other yields are drawn.
        ``fission_yields_mapping_mismatch`` names those whose tape yields,
        named and summed by the converter's rule, do not give back the yields
        the solver reads; they are held rather than drawn through a mapping
        their yields were not built with. Each is counted as a gap.
      - ``decay_branchings_perturbed``: with the ``"decay_branching"``
        source, the reachable multi-mode parents whose split was sampled. Held
        at their evaluated ratios, each a gap: ``no_decay_branching_uncertainty``
        (no mode states a sigma, or one does beside two or more that do not)
        and ``decay_branchings_too_wide`` (every mode that could move states
        a sigma no ratio with its mean can have). Sampled but short of the
        stated sigmas, each a gap too: ``decay_branchings_partly_held`` (one
        or more modes held at their ratios, for such a sigma or for stating
        none) and ``decay_branchings_sigma_adjusted`` (a mode drawn more than
        0.1% from its stated sigma: the largest exceeds the sum of the
        others', or the complement's needs a correlation non-negative ratios
        cannot carry). ``decay_branchings_floored`` /
        ``decay_branchings_sampled`` count draws whose complement was pulled
        back above zero and draws made.
      - ``isomer_branchings_perturbed``: with the ``"isomer_branching"``
        source, the reachable reactions, as ``"Parent reaction"``, whose
        isomeric split was drawn from the MF=40 covariance of at least one of
        their final states. ``no_isomer_branching_uncertainty`` lists those
        whose split can move but whose evaluation states no MF=40 on any
        state the run folds, held at their nominal shares, a gap.
        ``isomer_branching_blocks_not_used`` gives, per parent, the MF=40
        blocks not used, counted by why, a gap.
        ``isomer_branching_covariance_repaired`` and
        ``isomer_branching_lognormal_not_carried`` are the MF=40 matrices
        repaired to the nearest correlation matrix (a gap) and those not a
        lognormal's (not a gap), with the same entries as
        ``covariance_repairs`` and ``lognormal_not_carried``.
        ``isomer_production_sigma`` gives, as ``"Parent reaction Product"``,
        the relative sigma MF=40 states for the product's production, folded
        against the spectrum (the largest over the run's spectra).
        ``isomer_branchings_sampled`` counts the draws, one per drawn parent
        per replica. ``isomer_branchings_clipped`` gives, as
        ``"Parent reaction Product"``, how many list draws (one per replica
        per spectrum) scaled the product's drawn production back to the
        drawn reaction total, the stated productions of its list having
        come out above it; a product never clipped is absent.
      - ``damage_energy``: with ``displacement_damage=True`` and the
        ``"cross_sections"`` source, per nuclide of the starting composition
        ``{"explained": ..., "sigma": ...}``: the share of its stored MT=444
        rate its reactions' deposits reproduce, the part a draw moves (the
        rest, fission's share above all, is held, a gap), and the relative
        sigma of its drawn damage-energy rate, each weighted over the spectra
        by fluence.
      - ``statistical_rates``: with the ``"statistical"`` source on a
```

```
        transport run, how many tallied rates were sampled from their
        covariance; ``statistical_floored`` / ``statistical_sampled`` count
        draws that came out negative and were floored.
    - ``resonance_parameters``: per nuclide with resonance parameters
        (MF=2 and MF=32) and a rate in this run, how its resonance-range
        uncertainty was sampled. ``method`` is ``"parameters sampled"``
        (drawn per replica and the cross sections rebuilt from them, on a
        transport run against the flux it tallied through the resonances) or
        ``"first-order rows"`` (the MF=32 rows of ``covariance.arrow``),
        ``reason`` why the parameters were not sampled (``None`` where they
        were), and ``ranges`` one sampler report per sampled range:
        ``isotope`` and ``range`` indices, the ``gaussian``, ``lognormal``
        and ``held`` parameter counts, ``zero_mean_widths`` (widths stated
        with a zero mean and a nonzero sigma, held at zero) and
        ``negative_mean_widths`` (drawn as signed), each with ``index``,
        ``location``, ``quantity``, ``value`` and ``sigma``,
        ``zero_variance_with_covariance``, ``unattainable_pairs``, and
        ``stated_repair`` / ``transformed_repair`` (the nearest-correlation
        repair of the stated and of the log-space matrix, each with
        ``lambda_min``, ``frobenius_change``, ``max_change``, ``parameters``,
        ``iterations`` and ``converged``, ``None`` where none was needed).
        How far the draws' parameter correlations are from the evaluated
        ones, in the parameters themselves after both repairs and the
        lognormal transform, reads off ``largest_correlation_change`` (the
        largest change of one correlation) and
        ``correlation_frobenius_change``, per range and per nuclide over its
        ranges; every drawn parameter keeps its evaluated mean and sigma, so
        that is the whole of the difference in the first two moments. Widths
        stay lognormal, so a pair no lognormal carries
        (``unattainable_pairs``) is where it is largest. Empty on a transport
        run, which keeps the rows.
    - ``not_perturbed``: every input this run held at its nominal value,
        such as any MF=32 resonance-parameter covariance neither sampled nor
        in the library's ``covariance.arrow``, the photon and dose data, the
        material composition, any source switched off, and, where they
        applied, the self-shielding correction, the flux's response to a
        perturbed cross section on a transport run, and the per-branch decay
        emission of a parent whose branching was drawn.
    - ``samples`` / ``converged`` / ``convergence`` / ``hit_cap`` /
        ``unconverged``: how many replicas ran; whether every tracked sigma
        reached the target, ``convergence``, for the standard error of a
        sigma relative to it; whether the 1024-replica cap stopped the run
        first; and, worst first, every sigma that missed, as dicts of
        ``output`` (``"density"``, ``"activity"``, ``"decay_heat"`` or
        ``"decay_photon_rate"``), ``nuclide`` (``None`` for a total),
        ``step`` and the ``relative_standard_error`` it reached. Listed with
        a fixed ``samples`` too, where ``hit_cap`` is never set.
    - ``noise_removed`` / ``noisy``: whether the Monte Carlo noise in the
        replicas' spread was measured and removed from every sigma (with
        transport replica weights, each replica is solved a second time from
        the even half of the transport's histories, see
        ``get_nuclide_uncertainty_noise``), and, the largest share first,
        every tracked sigma whose removed noise variance exceeded a quarter
        of the variance left, as dicts of ``output``, ``nuclide``, ``step``,
        ``replica_spread``, ``noise`` and ``std_dev`` (the sigma reported).
        Those sigmas would sharpen most with more transport histories.

    Args:
        material_id: Material ID number.
    """
```

`meth` get_self_shielding_info

```
get_self_shielding_info(material_id: int) -> Optional[dict]
```

What the self-shielding did for one material.

Present for every material of a spectrum solve, shielded or not, and `None` for a transport-coupled solve or a material not in the results. `chord_cm` of `None` means the run was dilute and nothing was corrected; otherwise it says how: the `method` and `chord_cm` used, which nuclides were `shielded`, which were `not_shielded` and why, and `strongest_factor`, the smallest factor any group average was multiplied by. A run reporting `1.0` there shielded nothing in practice, which is a different statement from not having tried.

`would_shield` is the other direction, and is filled only on a dilute run: 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. There is no geometry in it: the weight is `1 / (1 + N * sigma_x)` on that one reaction and that nuclide's own density, which fixes the background at 1/cm, while the correction proper uses `1 / chord_cm` against the material's total. So the number scales with how strongly a nuclide's own resonances could bite without predicting what a given lump would see, and 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 than this one reaction can express.

On the FNS tungsten foil, `would_shield` reads 0.634 for W186 while the slowing-down correction on the same foil and spectrum saturates at 0.730 from a millimetre of chord upward. Read it as "this answer may be high, and this is a resonance absorber", which is the warning a dilute run should carry rather than silence. Read the size of the effect off a shielded run, by asking for one.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Source code in `yani/_core/__init__.pyi`

```
4285  def get_self_shielding_info(self, material_id: builtins.int) ->
4286  typing.Optional[dict]:
4287      r"""
4288      What the self-shielding did for one material.
4289
4290      Present for every material of a spectrum solve, shielded or not, and
4291      ``None`` for a transport-coupled solve or a material not in the
4292      results. ``chord_cm`` of ``None`` means the run was dilute and
4293      nothing was corrected; otherwise it says how: the ``method`` and
4294      ``chord_cm`` used, which nuclides were ``shielded``, which were
4295      ``not_shielded`` and why, and ``strongest_factor``, the smallest factor
4296      any group average was multiplied by. A run reporting ``1.0`` there
4297      shielded nothing in practice, which is a different statement from not
4298      having tried.
4299
4300      ``would_shield`` is the other direction, and is filled only on a dilute
4301      run: nuclides whose own resonances are structured enough to have
4302      suppressed a reaction, each mapped to how strongly.
4303
4304      It is an indicator, not a correction and not a bound. There is no
4305      geometry in it: the weight is ``1 / (1 + N * sigma_x)`` on that one
4306      reaction and that nuclide's own density, which fixes the background at
4307      1/cm, while the correction proper uses ``1 / chord_cm`` against the
4308      material's total. So the number scales with how strongly a nuclide's own
4309      resonances could bite without predicting what a given lump would see,
4310      and it can sit either side of the real factor: a lump thinner than a
4311      centimetre of chord shields less, and a material whose other nuclides
4312      dominate the total at the resonance dips the flux further than this one
4313      reaction can express.
4314
4315      On the FNS tungsten foil, ``would_shield`` reads 0.634 for W186 while
4316      the slowing-down correction on the same foil and spectrum saturates at
4317      0.730 from a millimetre of chord upward. Read it as "this answer may be
4318      high, and this is a resonance absorber", which is the warning a dilute
4319      run should carry rather than silence. Read the size of the effect off a
4320      shielded run, by asking for one.
4321
4322      Args:
4323          material_id: Material ID number.
      """
```

meth get_product_screening_info

```
get_product_screening_info(material_id: int) -> Optional[dict]
```

Which transmutation products had their cross sections loaded, and which were left out below `product_threshold`.

A product is loaded once its atom fraction, or its share of the activity, decay heat or decay photon power, reaches the threshold at any state of the schedule; one left out is still produced, decays and is reported, and only its own neutron reactions are dropped.

Keys: `threshold`; `rounds`, the screening solves run; `closure`, how many nuclides with neutron reactions the chain reaches from the composition, which is what loading without screening fetched; `loaded`, the sorted nuclides whose cross sections drive the solve; `unavailable`, products that crossed the threshold but the library has no data for; `below_threshold`, each product left out mapped to its largest atom fraction over the schedule; and `dropped_fraction`, the sum of those, which bounds the share of the material's atoms the left-out reactions could have moved before multiplying by the share of them that react.

`None` for a material not in the results, and for one a transport-coupled solve did not irradiate.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Source code in `yani/_core/__init__.pyi`

```
4324  def get_product_screening_info(self, material_id: builtins.int) ->
4325  typing.Optional[dict]:
4326      r"""
4327      Which transmutation products had their cross sections loaded, and
4328      which were left out below ``product_threshold``.
4329
4330      A product is loaded once its atom fraction, or its share of the
4331      activity, decay heat or decay photon power, reaches the threshold at
4332      any state of the schedule; one left out is still produced, decays and
4333      is reported, and only its own neutron reactions are dropped.
4334
4335      Keys: ``threshold``; ``rounds``, the screening solves run;
4336      ``closure``, how many nuclides with neutron reactions the chain
4337      reaches from the composition, which is what loading without screening
4338      fetched; ``loaded``, the sorted nuclides whose cross sections drive
4339      the solve; ``unavailable``, products that crossed the threshold but
4340      the library has no data for; ``below_threshold``, each product left
4341      out mapped to its largest atom fraction over the schedule; and
4342      ``dropped_fraction``, the sum of those, which bounds the share of the
4343      material's atoms the left-out reactions could have moved before
4344      multiplying by the share of them that react.
4345
4346      ``None`` for a material not in the results, and for one a
4347      transport-coupled solve did not irradiate.
4348
4349      Args:
4350          material_id: Material ID number.
      """
```

`meth` get_material_nuclides

```
get_material_nuclides(material_id: int, step: int) -> Optional[dict[str, float]]
```

Get material composition at a specific timestep as a dict.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial).	*required*

Returns:

Type	Description
`Optional[dict[str, float]]`	Dict of nuclide -> density [atoms/barn-cm], or None if not found.

Source code in `yani/_core/__init__.pyi`

```
4351  def get_material_nuclides(self, material_id: builtins.int, step: builtins.int) ->
4352  typing.Optional[builtins.dict[builtins.str, builtins.float]]:
4353      r"""
4354      Get material composition at a specific timestep as a dict.
4355
4356      Args:
4357          material_id: Material ID number.
4358          step: Timestep index (0 = initial).
4359
4360      Returns:
4361          Dict of nuclide -> density [atoms/barn-cm], or None if not found.
          """
```

`meth` get_material

```
get_material(material_id: int, step: int) -> Optional[Material]
```

Get material composition at a specific timestep.

The material keeps `material_id`, so it can stand in for the original in a model. Placed beside the original it needs an id of its own, since a geometry refuses two different materials with one id.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial).	*required*

Returns:

Type	Description
`Optional[Material]`	Material object with composition at that timestep, or None if not found.

Source code in `yani/_core/__init__.pyi`

```
4362  def get_material(self, material_id: builtins.int, step: builtins.int) ->
4363  typing.Optional[Material]:
4364      r"""
4365      Get material composition at a specific timestep.
4366
4367      The material keeps ``material_id``, so it can stand in for the
4368      original in a model. Placed beside the original it needs an id of its
4369      own, since a geometry refuses two different materials with one id.
4370
4371      Args:
4372          material_id: Material ID number.
4373          step: Timestep index (0 = initial).
4374
4375      Returns:
4376          Material object with composition at that timestep, or None if not found.
      """
```

`meth` step_materials

```
step_materials(material_id: int) -> list[Material]
```

Composition at the end of each schedule step, without the initial one.

Indexed as `timesteps`, so entry `i` is the state at the end of step `i` and pairs with `get_reaction_rates(material_id, i)`. This is the list `Material.transmute` used to return on its own. Each keeps `material_id`, as `get_material` says.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Returns:

Type	Description
`list[Material]`	List[Material]: One material per step; empty if the material is
`list[Material]`	unknown or was never stepped.

Source code in `yani/_core/__init__.pyi`

```
4377  def step_materials(self, material_id: builtins.int) -> builtins.list[Material]:
4378      r"""
4379      Composition at the end of each schedule step, without the initial one.
4380
4381      Indexed as ``timesteps``, so entry ``i`` is the state at the end of step
4382      ``i`` and pairs with ``get_reaction_rates(material_id, i)``. This is the
4383      list ``Material.transmute`` used to return on its own. Each keeps
4384      ``material_id``, as ``get_material`` says.
4385
4386      Args:
4387          material_id: Material ID number.
4388
4389      Returns:
4390          List[Material]: One material per step; empty if the material is
4391          unknown or was never stepped.
4392      """
```

`meth` get_final_material

```
get_final_material(material_id: int) -> Optional[Material]
```

Final composition for a material, after the last step. It keeps `material_id`, as `get_material` says.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Returns:

Type	Description
`Optional[Material]`	Material at the end of the schedule, or None if the material is
`Optional[Material]`	unknown.

Source code in `yani/_core/__init__.pyi`

```
4393  def get_final_material(self, material_id: builtins.int) ->
4394  typing.Optional[Material]:
4395      r"""
4396      Final composition for a material, after the last step. It keeps
4397      ``material_id``, as ``get_material`` says.
4398
4399      Args:
4400          material_id: Material ID number.
4401
4402      Returns:
4403          Material at the end of the schedule, or None if the material is
4404          unknown.
          """
```

`meth` get_dpa

```
get_dpa(material_id: int, element: Optional[str] = None) -> Optional[list[float]]
```

Cumulative NRT displacements per atom (dpa) over the schedule.

Present when the transmute call was given `displacement_damage=True`, and `None` otherwise. One value per state, aligned with `times`: entry 0 is the initial composition and is zero, and entry `i` is the total after schedule step `i - 1`. Cooldowns add nothing.

For an element `X` it is the damage energy deposited per atom of `X`, from `X`'s own nuclides, converted with `X`'s displacement threshold energy as `0.8 * E_damage / (2 * E_d)` (the NRT model, ASTM E521). The material total, with `element` omitted, is the atom-fraction-weighted sum over elements: each element's recoils are treated as slowing down among atoms of their own kind, which reduces to the elemental value for a pure element and leaves out energy transfer between elements in a cascade. MT=444 is already integrated over the recoil spectrum, so the per-recoil threshold steps of the NRT model (no displacement below `E_d`, one up to `2 * E_d / 0.8`) are not applied, which is the standard practice for a damage-energy cross section. The `E_d` used and its source are in `get_displacement_damage_info`, and the nuclear-data spread, when `data_uncertainty` was given too, in `get_dpa_uncertainty`.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`element`	`Optional[str]`	Element symbol, e.g. `"W"`, for that element's dpa; omit it for the material total.	`None`

Returns:

Type	Description
`Optional[list[float]]`	List of cumulative dpa, one per state, or None if damage was not
`Optional[list[float]]`	asked for or the material is not in the results.

Raises:

Type	Description
`ValueError`	If `element` has no dpa: it is not in the material on an irradiated step, or it is a transmutation product with no displacement threshold energy.

Source code in `yani/_core/__init__.pyi`

```
4405  def get_dpa(self, material_id: builtins.int, element: typing.Optional[builtins.str]
4406  = None) -> typing.Optional[builtins.list[builtins.float]]:
4407      r"""
4408      Cumulative NRT displacements per atom (dpa) over the schedule.
4409
4410      Present when the transmute call was given
4411      ``displacement_damage=True``, and ``None`` otherwise. One value per
4412      state, aligned with ``times``: entry 0 is the initial composition and
4413      is zero, and entry ``i`` is the total after schedule step ``i - 1``.
4414      Cooldowns add nothing.
4415
4416      For an element ``X`` it is the damage energy deposited per atom of
4417      ``X``, from ``X``'s own nuclides, converted with ``X``'s displacement
4418      threshold energy as ``0.8 * E_damage / (2 * E_d)`` (the NRT model,
4419      ASTM E521). The material total, with ``element`` omitted, is the
4420      atom-fraction-weighted sum over elements: each element's recoils are
4421      treated as slowing down among atoms of their own kind, which reduces to
4422      the elemental value for a pure element and leaves out energy transfer
4423      between elements in a cascade. MT=444 is already integrated over the
4424      recoil spectrum, so the per-recoil threshold steps of the NRT model
4425      (no displacement below ``E_d``, one up to ``2 * E_d / 0.8``) are not
4426      applied, which is the standard practice for a damage-energy cross
4427      section. The ``E_d`` used and its source are in
4428      ``get_displacement_damage_info``, and the nuclear-data spread, when
4429      ``data_uncertainty`` was given too, in ``get_dpa_uncertainty``.
4430
4431      Args:
4432          material_id: Material ID number.
4433          element: Element symbol, e.g. ``"W"``, for that element's dpa;
4434              omit it for the material total.
4435
4436      Returns:
4437          List of cumulative dpa, one per state, or None if damage was not
4438          asked for or the material is not in the results.
4439
4440      Raises:
4441          ValueError: If ``element`` has no dpa: it is not in the material
4442              on an irradiated step, or it is a transmutation product with no
4443              displacement threshold energy.
      """
```

`meth` get_damage_energy

```
get_damage_energy(material_id: int, element: Optional[str] = None) ->
Optional[list[float]]
```

Cumulative damage energy deposited per atom [eV] over the schedule.

The quantity dpa is computed from, kept separate so the displacement model can be changed without the data: each nuclide's MT=444 damage-energy cross section [eV barn] folded against the pulse spectrum by the same collapse as the reaction rates, times the flux magnitude and the step duration, at each step's composition. Indexed as `get_dpa`. For an element it is per atom of that element; the material total weights the elements by atom fraction, so it is per atom of the material.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`element`	`Optional[str]`	Element symbol for that element's damage energy; omit it for the material total.	`None`

Returns:

Type	Description
`Optional[list[float]]`	List of cumulative damage energy [eV per atom], one per state, or
`Optional[list[float]]`	None if damage was not asked for or the material is not in the
`Optional[list[float]]`	results.

Raises:

Type	Description
`ValueError`	If `element` is not in the material on an irradiated step.

Source code in `yani/_core/__init__.pyi`

```
4444  def get_damage_energy(self, material_id: builtins.int, element:
4445  typing.Optional[builtins.str] = None) ->
4446  typing.Optional[builtins.list[builtins.float]]:
4447      r"""
4448      Cumulative damage energy deposited per atom [eV] over the schedule.
4449
4450      The quantity dpa is computed from, kept separate so the displacement
4451      model can be changed without the data: each nuclide's MT=444
4452      damage-energy cross section [eV barn] folded against the pulse
4453      spectrum by the same collapse as the reaction rates, times the flux
4454      magnitude and the step duration, at each step's composition. Indexed
4455      as ``get_dpa``. For an element it is per atom of that element; the
4456      material total weights the elements by atom fraction, so it is per atom
4457      of the material.
4458
4459      Args:
4460          material_id: Material ID number.
4461          element: Element symbol for that element's damage energy; omit it
4462              for the material total.
4463
4464      Returns:
4465          List of cumulative damage energy [eV per atom], one per state, or
4466          None if damage was not asked for or the material is not in the
4467          results.
4468
4469      Raises:
4470          ValueError: If ``element`` is not in the material on an irradiated
                step.
          """
```

meth get_dpa_uncertainty

```
get_dpa_uncertainty(material_id: int, step: int, element: Optional[str] = None) ->
Optional[Estimate]
```

NRT dpa at one timestep, with the nuclear-data spread on it.

Present when the transmute call was given both `displacement_damage=True` and `data_uncertainty`. Each replica's dpa is accumulated over its own inventory, with each starting nuclide's damage-energy rate moved by the replica's 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. The elastic and inelastic reactions, which carry most of it, are drawn with the activation channels as one evaluation. What no reaction's deposit reproduces (fission's share) is held; `get_data_uncertainty_info(...)["damage_energy"]` says how much that is per nuclide.
- `"flux_spectrum"`: the damage energy's own per-group terms under the replica's flux draw, where the spectrum has a sigma.

Transmutation products' damage energy, short-range (LB=8) covariance and the displacement threshold energies are held at nominal.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition, which has no dpa).	*required*
`element`	`Optional[str]`	Element symbol for that element's dpa; omit it for the material total.	`None`

Returns:

Type	Description
`Optional[Estimate]`	Estimate | None: None if damage or uncertainty was not asked for,
`Optional[Estimate]`	or the material is not in the results.

Raises:

Type	Description
`ValueError`	If `element` has no dpa, or there is no such step.

Source code in `yani/_core/__init__.pyi`

```
4471  def get_dpa_uncertainty(self, material_id: builtins.int, step: builtins.int,
4472  element: typing.Optional[builtins.str] = None) -> typing.Optional[Estimate]:
4473      r"""
4474      NRT dpa at one timestep, with the nuclear-data spread on it.
4475
4476      Present when the transmute call was given both
4477      ``displacement_damage=True`` and ``data_uncertainty``. Each replica's
4478      dpa is accumulated over its own inventory, with each starting nuclide's
4479      damage-energy rate moved by the replica's draw:
4480
4481      - ``"cross_sections"``: MT=444 has no covariance of its own, but it is
4482        the sum over reactions of each cross section times the damage energy
4483        that reaction deposits, so a draw moves it by each reaction's deposit
4484        times its drawn change. The elastic and inelastic reactions, which
4485        carry most of it, are drawn with the activation channels as one
4486        evaluation. What no reaction's deposit reproduces (fission's share)
4487        is held; ``get_data_uncertainty_info(...)["damage_energy"]`` says how much
4488        that is per nuclide.
4489      - ``"flux_spectrum"``: the damage energy's own per-group terms under
4490        the replica's flux draw, where the spectrum has a sigma.
4491
4492      Transmutation products' damage energy, short-range (LB=8) covariance
4493      and the displacement threshold energies are held at nominal.
4494
4495      Args:
4496          material_id: Material ID number.
4497          step: Timestep index (0 = initial composition, which has no dpa).
4498          element: Element symbol for that element's dpa; omit it for the
4499              material total.
4500
4501      Returns:
4502          Estimate | None: None if damage or uncertainty was not asked for,
4503          or the material is not in the results.
4504
4505      Raises:
4506          ValueError: If ``element`` has no dpa, or there is no such step.
      """
```

`meth` get_damage_energy_uncertainty

```
get_damage_energy_uncertainty(material_id: int, step: int, element: Optional[str]
= None) -> Optional[Estimate]
```

Damage energy per atom [eV] at one timestep, with the nuclear-data spread on it.

The same replicas as `get_dpa_uncertainty`, before the NRT conversion, so its relative spread is the dpa's for a pure element.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Timestep index (0 = initial composition).	*required*
`element`	`Optional[str]`	Element symbol for that element's damage energy; omit it for the material total.	`None`

Returns:

Type	Description
`Optional[Estimate]`	Estimate | None: None if damage or uncertainty was not asked for,
`Optional[Estimate]`	or the material is not in the results.

Raises:

Type	Description
`ValueError`	If `element` is not in the material on an irradiated step, or there is no such step.

Source code in `yani/_core/__init__.pyi`

```
def get_damage_energy_uncertainty(self, material_id: builtins.int, step:
builtins.int, element: typing.Optional[builtins.str] = None) ->
typing.Optional[Estimate]:
    r"""
    Damage energy per atom [eV] at one timestep, with the nuclear-data
    spread on it.

    The same replicas as ``get_dpa_uncertainty``, before the NRT
    conversion, so its relative spread is the dpa's for a pure element.

    Args:
        material_id: Material ID number.
        step: Timestep index (0 = initial composition).
        element: Element symbol for that element's damage energy; omit it
            for the material total.

    Returns:
        Estimate | None: None if damage or uncertainty was not asked for,
        or the material is not in the results.

    Raises:
        ValueError: If ``element`` is not in the material on an irradiated
            step, or there is no such step.
    """
```

`meth` get_displacement_damage_info

```
get_displacement_damage_info(material_id: int) -> Optional[dict]
```

What the displacement damage was computed with, and what it could not count.

`None` unless the transmute call was given `displacement_damage=True`. A dict:

- `model`: `"NRT"`, and `efficiency`: `0.8`.
- `displacement_energies`: per element with dpa, `{"energy": E_d in eV, "source": ...}`, where `source` is `"ASTM E521"`, `"OECD-NEA 2015"` (Table 2.4 of NEA/NSC/DOC(2015)9, for an element ASTM E521 does not cover) or `"user"`.
- `without_damage_energy`: nuclides present on an irradiated step whose data has no MT=444, each with the largest atom fraction it reached there. Their damage energy is not counted, so a large entry here means the totals are low by about that share. Only transmutation products can appear: a nuclide of the starting composition without MT=444 is refused.
- `without_displacement_energy`: transmutation-product elements with no displacement threshold energy (hydrogen and helium, typically), each with the largest atom fraction reached. Their damage energy is counted in `get_damage_energy`; they add nothing to `get_dpa`. Pass `displacement_energies` to include them.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*

Source code in `yani/_core/__init__.pyi`

```
4529  def get_displacement_damage_info(self, material_id: builtins.int) ->
4530  typing.Optional[dict]:
4531      r"""
4532      What the displacement damage was computed with, and what it could not
4533      count.
4534
4535      ``None`` unless the transmute call was given
4536      ``displacement_damage=True``. A dict:
4537
4538      - ``model``: ``"NRT"``, and ``efficiency``: ``0.8``.
4539      - ``displacement_energies``: per element with dpa,
4540        ``{"energy": E_d in eV, "source": ...}``, where ``source`` is
4541        ``"ASTM E521"``, ``"OECD-NEA 2015"`` (Table 2.4 of NEA/NSC/DOC(2015)9,
4542        for an element ASTM E521 does not cover) or ``"user"``.
4543      - ``without_damage_energy``: nuclides present on an irradiated step
4544        whose data has no MT=444, each with the largest atom fraction it
4545        reached there. Their damage energy is not counted, so a large entry
4546        here means the totals are low by about that share. Only
4547        transmutation products can appear: a nuclide of the starting
4548        composition without MT=444 is refused.
4549      - ``without_displacement_energy``: transmutation-product elements with
4550        no displacement threshold energy (hydrogen and helium, typically),
4551        each with the largest atom fraction reached. Their damage energy is
4552        counted in ``get_damage_energy``; they add nothing to ``get_dpa``.
4553        Pass ``displacement_energies`` to include them.
4554
4555      Args:
4556          material_id: Material ID number.
      """
```

`meth` get_reaction_rates

```
get_reaction_rates(material_id: int, step: int) -> Optional[Any]
```

Per-edge reaction rates the solve drove one step with.

The rate of each individual production edge, which the solve computes to build the burnup matrix and used to throw away. Without it a consumer can enumerate the routes into a product but not weight them, so a route carrying 99.9% and one carrying 0.1% look alike.

Both methods report the same quantity in the same shape. `method="coupled"` takes the rates from that step's transport tallies and `method="independent"` from the single transport's multigroup fold scaled by the step's source rate; either way the edge rate is that rate times the branching of the chain the step was solved with, isomeric overlay included.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Schedule step index, the same index as `timesteps` and `get_source_rates`. This is one less than the `step` the composition getters take, where 0 is the initial composition.	*required*

Returns:

Type	Description
`Optional[Any]`	dict[str, dict[str, list[tuple[str | None, float]]]] | None: parent
`Optional[Any]`	nuclide -> reaction kind -> [(target, rate [1/s])]. A `target` of
`Optional[Any]`	`None` is a channel naming no single product, which in practice
`Optional[Any]`	means fission, whose products come from the yields rather than from
`Optional[Any]`	an edge. Empty for a decay-only step, and `None` if the material
`Optional[Any]`	or the step is unknown.

Examples:

```
>>> rates = results.get_reaction_rates(material_id=1, step=0)
>>> rates["Fe56"]["(n,gamma)"]
[('Fe57', 1.7e-09)]
>>> # share of Mn56 production arriving down each route
>>> into_mn56 = [
...     (parent, kind, rate)
...     for parent, kinds in rates.items()
...     for kind, edges in kinds.items()
...     for target, rate in edges
...     if target == "Mn56"
... ]
```

Source code in `yani/_core/__init__.pyi`

```
def get_reaction_rates(self, material_id: builtins.int, step: builtins.int) ->
typing.Optional[typing.Any]:
    r"""
    Per-edge reaction rates the solve drove one step with.

    The rate of each individual production edge, which the solve computes to
    build the burnup matrix and used to throw away. Without it a consumer
    can enumerate the routes into a product but not weight them, so a route
    carrying 99.9% and one carrying 0.1% look alike.

    Both methods report the same quantity in the same shape.
    ``method="coupled"`` takes the rates from that step's transport tallies
    and ``method="independent"`` from the single transport's multigroup fold
    scaled by the step's source rate; either way the edge rate is that rate
    times the branching of the chain the step was solved with, isomeric
    overlay included.

    Args:
        material_id: Material ID number.
        step: Schedule step index, the same index as ``timesteps`` and
            ``get_source_rates``. This is one less than the ``step`` the
            composition getters take, where 0 is the initial composition.

    Returns:
        dict[str, dict[str, list[tuple[str | None, float]]]] | None: parent
        nuclide -> 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. Empty for a decay-only step, and ``None`` if the material
        or the step is unknown.

    Examples:
        >>> rates = results.get_reaction_rates(material_id=1, step=0)
        >>> rates["Fe56"]["(n,gamma)"]
        [('Fe57', 1.7e-09)]
        >>> # share of Mn56 production arriving down each route
        >>> into_mn56 = [
        ...     (parent, kind, rate)
        ...     for parent, kinds in rates.items()
        ...     for kind, edges in kinds.items()
        ...     for target, rate in edges
        ...     if target == "Mn56"
        ... ]
    """
```

meth get_reaction_rate_spectrum

```
get_reaction_rate_spectrum(material_id: int, nuclide: str, kind: str, step: int)
> Optional[Any]
```

One channel's reaction rate over one step, resolved onto the groups of the spectrum that drove it.

`get_reaction_rates` answers with one number per edge, already collapsed against the whole spectrum. That number cannot say which part of the spectrum made it, and where a cross section

spans decades the two readings are different physics: an effective `W186(n,gamma)` of 57 mb against a spectrum 89% of which sits in 12-16 MeV and 0.7% of which is below 100 keV is either a fast-capture rate or a resonance-region rate, and only the breakdown says which. A disagreement can then be pinned on resonance processing rather than guessed at, and a covariance grid that stops short of the spectrum can be checked against where the rate actually is.

The per-group entries sum to the collapsed rate for the same channel, to floating-point rounding, because both come from the same walk of the same cross sections with the same self-shielding weighting.

Nothing is stored for this. The breakdown is re-derived from the spectrum when asked for, which is one reaction over the group structure; keeping it for every channel would be tens of megabytes on a 709-group structure, and a run that never asks should not pay that.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`nuclide`	`str`	The parent the reaction happens on, e.g. `"W186"`.	*required*
`kind`	`str`	The reaction, spelled as the chain spells it, e.g. `"(n,gamma)"`.	*required*
`step`	`int`	Schedule step index, as `get_reaction_rates` takes it.	*required*

Returns:

Type	Description
`Optional[Any]`	dict | None: `boundaries`, the group boundaries [eV] ascending and
`Optional[Any]`	one longer than the rates, and `rates`, each group's contribution
`Optional[Any]`	to the rate [1/s]. None when the material, the step, the nuclide or
`Optional[Any]`	the channel is unknown; for a step that drove no flux, whether by
`Optional[Any]`	being a cooldown or by carrying a zero rate; for `(n,n')`, whose
`Optional[Any]`	rate comes from the branching overlay's partials rather than from a
`Optional[Any]`	group average; and for a transport-coupled solve, which scores its
`Optional[Any]`	rates at the collision energy and keeps no group structure to
`Optional[Any]`	resolve them onto.

Examples:

```
>>> spectrum = results.get_reaction_rate_spectrum(
...     material_id=1, nuclide="W186", kind="(n,gamma)", step=0
... )
>>> sum(spectrum["rates"])  # the collapsed rate
1.9e-09
>>> # where in energy that rate came from
>>> below_100_keV = sum(
...     r
...     for lo, r in zip(spectrum["boundaries"], spectrum["rates"])
...     if lo < 1.0e5
... )
```

Source code in `yani/_core/__init__.pyi`

```
def get_reaction_rate_spectrum(self, material_id: builtins.int, nuclide:
builtins.str, kind: builtins.str, step: builtins.int) ->
typing.Optional[typing.Any]:
    r"""
    One channel's reaction rate over one step, resolved onto the groups of
    the spectrum that drove it.

    ``get_reaction_rates`` answers with one number per edge, already
    collapsed against the whole spectrum. That number cannot say which part
    of the spectrum made it, and where a cross section spans decades the two
    readings are different physics: an effective ``W186(n,gamma)`` of 57 mb
    against a spectrum 89% of which sits in 12-16 MeV and 0.7% of which is
    below 100 keV is either a fast-capture rate or a resonance-region rate,
    and only the breakdown says which. A disagreement can then be pinned on
    resonance processing rather than guessed at, and a covariance grid that
    stops short of the spectrum can be checked against where the rate
    actually is.

    The per-group entries sum to the collapsed rate for the same channel, to
    floating-point rounding, because both come from the same walk of the
    same cross sections with the same self-shielding weighting.

    Nothing is stored for this. The breakdown is re-derived from the
    spectrum when asked for, which is one reaction over the group structure;
    keeping it for every channel would be tens of megabytes on a 709-group
    structure, and a run that never asks should not pay that.

    Args:
        material_id: Material ID number.
        nuclide: The parent the reaction happens on, e.g. ``"W186"``.
        kind: The reaction, spelled as the chain spells it, e.g.
            ``"(n,gamma)"``.
        step: Schedule step index, as ``get_reaction_rates`` takes it.

    Returns:
        dict | None: ``boundaries``, the group boundaries [eV] ascending and
        one longer than the rates, and ``rates``, each group's contribution
        to the rate [1/s]. None when the material, the step, the nuclide or
        the channel is unknown; for a step that drove no flux, whether by
        being a cooldown or by carrying a zero rate; for ``(n,n')``, whose
        rate comes from the branching overlay's partials rather than from a
        group average; and for a transport-coupled solve, which scores its
        rates at the collision energy and keeps no group structure to
        resolve them onto.

    Examples:
        >>> spectrum = results.get_reaction_rate_spectrum(
        ...     material_id=1, nuclide="W186", kind="(n,gamma)", step=0
        ... )
        >>> sum(spectrum["rates"])  # the collapsed rate
        1.9e-09
        >>> # where in energy that rate came from
        >>> below_100_keV = sum(
        ...     r
        ...     for lo, r in zip(spectrum["boundaries"], spectrum["rates"])
        ...     if lo < 1.0e5
        ... )
    """
```

meth get_isomeric_branching

```
get_isomeric_branching(material_id: int, step: int) -> Optional[Any]
```

Flux-weighted isomeric branching at one step.

Which state a reaction leaves its product in is energy dependent, so the single number describing a spectrum is the branching collapsed against it, and that number exists only inside a solve. The chain file carries the unweighted ratios, and where the branching overlay supplies the split it carries a placeholder instead: on TENDL-2025 the dominant tungsten channel reads `W186 (n,2n) -> W185 1.000000` and `W186 (n,2n) -> W185_m1 0.000000` in the file, and the overlay replaces both at solve time with roughly the 54/46 that spectrum gives. On a foil whose decay heat comes from an isomer, that difference is the whole answer.

Only channels landing in more than one final state appear. A channel with one product has no branching to report, and listing it at 1.0 buries the ones that do; `get_reaction_rates` has the unnormalised edges if the rest is wanted.

This is the number that says whether a disagreement belongs to a cross section or to a branching ratio, which are different data and different fixes.

Channels come back ordered by production, the channel's rate times its parent's atom density at the start of the step. A rate on its own is per atom of the parent, so ordering on it promotes whatever sits on a trace isotope: on an FNS tungsten foil `W180(n,2n)` has the highest per-atom rate of any channel in the foil, and W180 is 0.12% of it, so by what it made the channel falls to fifth, two orders of magnitude below the `W186(n,2n)` carrying most of that foil's decay heat.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Schedule step index, the same index `get_reaction_rates` takes, which is one less than the composition getters' step.	*required*

Returns:

Type	Description
`Optional[Any]`	list[dict] | None: one entry per splitting channel, most produced
`Optional[Any]`	first, each with `parent`, `reaction`, `production` and
`Optional[Any]`	`split`. `split` is [(target, fraction)] summing to one and
`Optional[Any]`	ordered with the largest share first. Empty for a decay-only step,
`Optional[Any]`	and None if the material or the step is unknown. A parent absent
`Optional[Any]`	from the step's starting composition has production 0.0 and sorts
`Optional[Any]`	last rather than being dropped.

Examples:

```
>>> results.get_isomeric_branching(material_id=1, step=0)[0]
{'parent': 'W186', 'reaction': '(n,2n)', 'production': 9.35e-14,
 'split': [('W185_m1', 0.535), ('W185', 0.465)]}
```

Source code in `yani/_core/__init__.pyi`

```
4656  def get_isomeric_branching(self, material_id: builtins.int, step: builtins.int) ->
4657  typing.Optional[typing.Any]:
4658      r"""
4659      Flux-weighted isomeric branching at one step.
4660
4661      Which state a reaction leaves its product in is energy dependent, so the
4662      single number describing a spectrum is the branching collapsed against
4663      it, and that number exists only inside a solve. The chain file carries
4664      the unweighted ratios, and where the branching overlay supplies the
4665      split it carries a placeholder instead: on TENDL-2025 the dominant
4666      tungsten channel reads ``W186 (n,2n) -> W185 1.000000`` and
4667      ``W186 (n,2n) -> W185_m1 0.000000`` in the file, and the overlay
4668      replaces both at solve time with roughly the 54/46 that spectrum gives.
4669      On a foil whose decay heat comes from an isomer, that difference is the
4670      whole answer.
4671
4672      Only channels landing in more than one final state appear. A channel
4673      with one product has no branching to report, and listing it at 1.0
4674      buries the ones that do; ``get_reaction_rates`` has the unnormalised
4675      edges if the rest is wanted.
4676
4677      This is the number that says whether a disagreement belongs to a cross
4678      section or to a branching ratio, which are different data and different
4679      fixes.
4680
4681      Channels come back ordered by production, the channel's rate times its
4682      parent's atom density at the start of the step. A rate on its own is per
4683      atom of the parent, so ordering on it promotes whatever sits on a trace
4684      isotope: on an FNS tungsten foil ``W180(n,2n)`` has the highest per-atom
4685      rate of any channel in the foil, and W180 is 0.12% of it, so by what it
4686      made the channel falls to fifth, two orders of magnitude below the
4687      ``W186(n,2n)`` carrying most of that foil's decay heat.
4688
4689      Args:
4690          material_id: Material ID number.
4691          step: Schedule step index, the same index ``get_reaction_rates``
4692              takes, which is one less than the composition getters' step.
4693
4694      Returns:
4695          list[dict] | None: one entry per splitting channel, most produced
4696          first, each with ``parent``, ``reaction``, ``production`` and
4697          ``split``. ``split`` is [(target, fraction)] summing to one and
4698          ordered with the largest share first. Empty for a decay-only step,
4699          and None if the material or the step is unknown. A parent absent
4700          from the step's starting composition has production 0.0 and sorts
4701          last rather than being dropped.
4702
4703      Examples:
4704          >>> results.get_isomeric_branching(material_id=1, step=0)[0]
4705          {'parent': 'W186', 'reaction': '(n,2n)', 'production': 9.35e-14,
4706           'split': [('W185_m1', 0.535), ('W185', 0.465)]}
      """
```

meth get_branching_report

```
get_branching_report(material_id: int, step: int) -> Optional[dict]
```

What the isomeric-branching rule did over one step's spectrum.

The branching evaluation gives the split and the cross-section library the total. How a list's values are read is decided by how the evaluation gives them, which the converter records: a complete MF=10 list (its ground state listed) and every MF=9 list are shares of the transport total, applied at each energy; an MF=10 list of isomers only, and every `(n,n')` list, are absolute productions, the ground state taking the rest. This says, per channel, which of those applied and how much of the parent's removal rate rests on anything the evaluation does not give.

It also says where each split came from. With several branching sources (`transmutation_branch_ratios = ["irdff-ii", "endf-b8.1"]`) each channel takes its split from the first source that evaluates it, and its `source` names that one. `mixed` is True when the splits come from more than one library, or a split from a library other than its reaction's total, so a mixed run reads as one at the top of the report.

MT=5, `(n,anything)`, is the `(n,X)` reaction: its residuals, read from the reaction library's MF=6 MT=5, are shares of the MT=5 total at each energy (`file` 6, `representation` `"share"`), and its light particles H1 to He4 are their multiplicities times that total (`representation` `"multiplicity"`, each state's `share` the multiplicity folded over the spectrum, which can exceed one).

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; each channel's `clipped_share` and `extrapolated_share` are of its own parent's removal. A multiplicity above what the target's nucleons allow is clipped like any other impossible value. `unmodelled_mt5` lists the parents whose MT=5 residuals the chain does not model, with the reason; on a reactions subsection that carries MT=5, a run warns when those of the material's own nuclides carry more than 0.1% of the material's removal rate (one written before MT=5 was carried is reported only). The solve continues with an incomplete inventory: absent residuals are not invented, and the missing removal fraction does not bound activity or decay-heat error. The clipped/held production refusals are unchanged.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`step`	`int`	Schedule step index, as `get_reaction_rates` takes it.	*required*

Returns:

Type	Description
`Optional[dict]`	dict | None: `channels`, `dropped`, `unmodelled_mt5`,
`Optional[dict]`	`sources` and `mixed`, or
`Optional[dict]`	None if the material or the step is unknown. Each channel has
`Optional[dict]`	`parent`, `reaction`, `mt`, `file` (6, 9 or 10),
`Optional[dict]`	`representation` (`"share"`, `"absolute"` or
`Optional[dict]`	`"multiplicity"`), `complete`,
`Optional[dict]`	`completeness_source`, `denominator`, `states` (each with
`Optional[dict]`	`target`, `lfs`, `level_route`, `level_energy_difference`
`Optional[dict]`	and `share`, its share of the reaction), `removal_share` (the
`Optional[dict]`	reaction's share of the parent's removal rate), `clipped_share`
`Optional[dict]`	and `extrapolated_share` (of the same removal rate),
`Optional[dict]`	`own_total_excess` (`(energy_ev, ratio)` where the listed values
`Optional[dict]`	most exceed the evaluation's own total, or None),
`Optional[dict]`	`normalisation`, `source` (the branching source the split came
`Optional[dict]`	from, as named in `transmutation_branch_ratios`, or None for an
`Optional[dict]`	`(n,X)` channel, whose products come from the reactions
`Optional[dict]`	subsection) and `cross_section_library` (where the reaction's
`Optional[dict]`	total came from, or None where that is not known). `sources` lists
`Optional[dict]`	the branching sources the channels used, in the configured order,
`Optional[dict]`	and `mixed` says whether the run mixes libraries as above. Each

Type	Description
`Optional[dict]`	dropped channel has `parent`,
`Optional[dict]`	`reaction`, `target`, `reason` and `removal_share` (None
`Optional[dict]`	where it cannot be folded). `unmodelled_mt5` is
`Optional[dict]`	`[(nuclide, share, reason)]`, MT=5's share of the removal rate
`Optional[dict]`	of each parent whose MT=5 residuals are not modelled, largest
`Optional[dict]`	first. Empty for a decay-only step.

Examples:

```
>>> report = results.get_branching_report(material_id=1, step=0)
>>> report["channels"][0]["representation"]
'absolute'
>>> # A Nb93 foil, transmutation_branch_ratios = ["irdff-ii", "endf-b8.1"]
>>> [(c["reaction"], c["source"]) for c in report["channels"] if c["parent"] ==
"Nb93"]
[('(n,2n)', 'irdff-ii'), ('(n,gamma)', 'irdff-ii'), ("(n,n')", 'irdff-ii')]
>>> report["mixed"]
True
```

Source code in `yani/_core/__init__.pyi`

```
def get_branching_report(self, material_id: builtins.int, step: builtins.int) ->
typing.Optional[dict]:
    r"""
    What the isomeric-branching rule did over one step's spectrum.

    The branching evaluation gives the split and the cross-section library
    the total. How a list's values are read is decided by how the
    evaluation gives them, which the converter records: a complete MF=10
    list (its ground state listed) and every MF=9 list are shares of the
    transport total, applied at each energy; an MF=10 list of isomers only,
    and every ``(n,n')`` list, are absolute productions, the ground state
    taking the rest. This says, per channel, which of those applied and how
    much of the parent's removal rate rests on anything the evaluation
    does not give.

    It also says where each split came from. With several branching
    sources (``transmutation_branch_ratios = ["irdff-ii", "endf-b8.1"]``)
    each channel takes its split from the first source that evaluates it,
    and its ``source`` names that one. ``mixed`` is True when the splits
    come from more than one library, or a split from a library other than
    its reaction's total, so a mixed run reads as one at the top of the
    report.

    MT=5, ``(n,anything)``, is the ``(n,X)`` reaction: its residuals, read
    from the reaction library's MF=6 MT=5, are shares of the MT=5 total
    at each energy (``file`` 6, ``representation`` ``"share"``), and its
    light particles H1 to He4 are their multiplicities times that total
    (``representation`` ``"multiplicity"``, each state's ``share`` the
    multiplicity folded over the spectrum, which can exceed one).

    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; each channel's ``clipped_share`` and
    ``extrapolated_share`` are of its own parent's removal. A multiplicity above
what the target's nucleons allow is
    clipped like any other impossible value. ``unmodelled_mt5`` lists the
    parents whose MT=5 residuals the chain does not model, with the reason;
    on a reactions subsection that carries MT=5, a run warns when those
    of the material's own nuclides carry more than 0.1% of the material's
    removal rate (one written before MT=5 was carried is reported only).
    The solve continues with an incomplete inventory: absent residuals are
    not invented, and the missing removal fraction does not bound activity
    or decay-heat error. The clipped/held production refusals are unchanged.

    Args:
        material_id: Material ID number.
        step: Schedule step index, as ``get_reaction_rates`` takes it.

    Returns:
        dict | None: ``channels``, ``dropped``, ``unmodelled_mt5``,
        ``sources`` and ``mixed``, or
        None if the material or the step is unknown. Each channel has
        ``parent``, ``reaction``, ``mt``, ``file`` (6, 9 or 10),
        ``representation`` (``"share"``, ``"absolute"`` or
        ``"multiplicity"``), ``complete``,
        ``completeness_source``, ``denominator``, ``states`` (each with
        ``target``, ``lfs``, ``level_route``, ``level_energy_difference``
        and ``share``, its share of the reaction), ``removal_share`` (the
        reaction's share of the parent's removal rate), ``clipped_share``
        and ``extrapolated_share`` (of the same removal rate),
        ``own_total_excess`` (``(energy_ev, ratio)`` where the listed values
        most exceed the evaluation's own total, or None),
        ``normalisation``, ``source`` (the branching source the split came
        from, as named in ``transmutation_branch_ratios``, or None for an
```

```
4771          ``(n,X)`` channel, whose products come from the reactions
4772          subsection) and ``cross_section_library`` (where the reaction's
4773          total came from, or None where that is not known). ``sources`` lists
4774          the branching sources the channels used, in the configured order,
4775          and ``mixed`` says whether the run mixes libraries as above. Each
4776          dropped channel has ``parent``,
4777          ``reaction``, ``target``, ``reason`` and ``removal_share`` (None
4778          where it cannot be folded). ``unmodelled_mt5`` is
4779          ``[(nuclide, share, reason)]``, MT=5's share of the removal rate
4780          of each parent whose MT=5 residuals are not modelled, largest
4781          first. Empty for a decay-only step.
4782
4783      Examples:
4784          >>> report = results.get_branching_report(material_id=1, step=0)
4785          >>> report["channels"][0]["representation"]
4786          'absolute'
4787          >>> # A Nb93 foil, transmutation_branch_ratios = ["irdff-ii", "endf-b8.1"]
4788          >>> [(c["reaction"], c["source"]) for c in report["channels"] if
4789  c["parent"] == "Nb93"]
4790          [('(n,2n)', 'irdff-ii'), ('(n,gamma)', 'irdff-ii'), ("(n,n')", 'irdff-ii')]
              >>> report["mixed"]
              True
          """
```

meth get_production_routes

```
get_production_routes(material_id: int, product: str, step: int, reaction_depth:
int = 1, decay_depth: int = 3) -> Optional[Any]
```

Every way a product was made over one step, weighted by how much of it arrived down each.

Enumerating routes is easy and weighting them is not. Asked what makes W187, a chain answers `Os190(n,a)` and `Ir192(n,npa)` as readily as `W186(n,gamma)`, and nothing in a tungsten foil is osmium. So the walk starts from the nuclides the material actually began with, and each route is weighted by what its own reactions drove rather than by anything read off the chain.

A route is `reaction_depth` neutron reactions followed by up to `decay_depth` decays. Its weight is the atoms it starts from, times each reaction step's per-atom production over the step, times the branching of every decay it passes through: a route through a 1% branch delivers 1% of what the reaction made. Reaction steps carry the step duration, so a two-reaction route is in the same units as a one-reaction route and comes out smaller by roughly a factor of the fluence, which is the honest answer for an irradiation short enough that products barely burn.

Parameters:

Name	Type	Description	Default
`material_id`	`int`	Material ID number.	*required*
`product`	`str`	The nuclide whose production is being explained.	*required*
`step`	`int`	Schedule step index, as `get_reaction_rates` takes it.	*required*
`reaction_depth`	`int`	Neutron reactions a route may use. 1 is what the published pathway tables carry.	`1`
`decay_depth`	`int`	Decays a route may follow after them.	`3`

Returns:

Type	Description
`Optional[Any]`	list[dict] | None: one entry per route, largest share first, each
`Optional[Any]`	with `route` (the string the published tables print, e.g.
`Optional[Any]`	`"W186(n,2n)W185_m1(IT)W185"`), `steps` (the same thing as
`Optional[Any]`	`(parent, kind, target)` triples), `share` (of this product's
`Optional[Any]`	production, summing to one) and `production` (atoms per barn-cm,
`Optional[Any]`	before normalising, so that 100% of almost nothing is
`Optional[Any]`	distinguishable from 100% of the inventory). Empty when nothing in
`Optional[Any]`	this material makes the product, which is an answer. None if the
`Optional[Any]`	material, the step, or the chain is unknown.

Examples:

```
>>> for r in results.get_production_routes(1, "W185", 0):
...     print(f"{r['route']:<32} {r['share']:.1%}")
W186(n,2n)W185_m1(IT)W185        53.0%
W186(n,2n)W185                   46.8%
```

Source code in `yani/_core/__init__.pyi`

```
4791    def get_production_routes(self, material_id: builtins.int, product: builtins.str,
4792    step: builtins.int, reaction_depth: builtins.int = 1, decay_depth: builtins.int =
4793    3) -> typing.Optional[typing.Any]:
4794        r"""
4795        Every way a product was made over one step, weighted by how much of it
4796        arrived down each.
4797
4798        Enumerating routes is easy and weighting them is not. Asked what makes
4799        W187, a chain answers ``Os190(n,a)`` and ``Ir192(n,npa)`` as readily as
4800        ``W186(n,gamma)``, and nothing in a tungsten foil is osmium. So the walk
4801        starts from the nuclides the material actually began with, and each
4802        route is weighted by what its own reactions drove rather than by
4803        anything read off the chain.
4804
4805        A route is ``reaction_depth`` neutron reactions followed by up to
4806        ``decay_depth`` decays. Its weight is the atoms it starts from, times
4807        each reaction step's per-atom production over the step, times the
4808        branching of every decay it passes through: a route through a 1% branch
4809        delivers 1% of what the reaction made. Reaction steps carry the step
4810        duration, so a two-reaction route is in the same units as a one-reaction
4811        route and comes out smaller by roughly a factor of the fluence, which is
4812        the honest answer for an irradiation short enough that products barely
4813        burn.
4814
4815        Args:
4816            material_id: Material ID number.
4817            product: The nuclide whose production is being explained.
4818            step: Schedule step index, as ``get_reaction_rates`` takes it.
4819            reaction_depth (int): Neutron reactions a route may use. 1 is what
4820                the published pathway tables carry.
4821            decay_depth (int): Decays a route may follow after them.
4822
4823        Returns:
4824            list[dict] | None: one entry per route, largest share first, each
4825            with ``route`` (the string the published tables print, e.g.
4826            ``"W186(n,2n)W185_m1(IT)W185"``), ``steps`` (the same thing as
4827            ``(parent, kind, target)`` triples), ``share`` (of this product's
4828            production, summing to one) and ``production`` (atoms per barn-cm,
4829            before normalising, so that 100% of almost nothing is
4830            distinguishable from 100% of the inventory). Empty when nothing in
4831            this material makes the product, which is an answer. None if the
4832            material, the step, or the chain is unknown.
4833
4834        Examples:
4835            >>> for r in results.get_production_routes(1, "W185", 0):
4836            ...     print(f"{r['route']:<32} {r['share']:.1%}")
4837            W186(n,2n)W185_m1(IT)W185        53.0%
                W186(n,2n)W185                   46.8%
        """
```

`meth` to_arrow

```
to_arrow(path: Optional[str | PathLike | Path] = None) -> Optional[bytes]
```

Save the result as an Arrow IPC file.

One file holds one result. Every file has the same schema, so the files of a run split into parts read back as one table with `polars.scan_ipc("parts/*.arrow")` or `pyarrow.dataset`: one row per stored number, with columns `material_id`, `material_name`, `step`, `quantity` (`"density"`, `"reaction_rate"` or `"source_rate"`), `nuclide`, `reaction`, `product` and `value`. Everything else the result holds is in the schema metadata, so `TransmutationResults.from_arrow` gives back the same result.

The file pins, rather than stores, the transmutation chain and the nuclear data the run read; see `from_arrow`.

Parameters:

Name	Type	Description	Default
`path`	`str | PathLike | None`	Where to write. None returns the file's bytes instead, for a browser or Pyodide session with no file system to write to.	`None`

Returns:

Type	Description
`Optional[bytes]`	bytes | None: The file's bytes when `path` is None.

Examples:

```
>>> results.to_arrow("part_1.arrow")
>>> data = results.to_arrow()
```

Source code in `yani/_core/__init__.pyi`

```
4838  def to_arrow(self, path: typing.Optional[builtins.str | os.PathLike | pathlib.Path]
4839  = None) -> typing.Optional[bytes]:
4840      r"""
4841      Save the result as an Arrow IPC file.
4842
4843      One file holds one result. Every file has the same schema, so the
4844      files of a run split into parts read back as one table with
4845      ``polars.scan_ipc("parts/*.arrow")`` or ``pyarrow.dataset``: one row
4846      per stored number, with columns ``material_id``, ``material_name``,
4847      ``step``, ``quantity`` (``"density"``, ``"reaction_rate"`` or
4848      ``"source_rate"``), ``nuclide``, ``reaction``, ``product`` and
4849      ``value``. Everything else the result holds is in the schema metadata,
4850      so ``TransmutationResults.from_arrow`` gives back the same result.
4851
4852      The file pins, rather than stores, the transmutation chain and the
4853      nuclear data the run read; see ``from_arrow``.
4854
4855      Args:
4856          path (str | os.PathLike | None): Where to write. None returns the
4857              file's bytes instead, for a browser or Pyodide session with no
4858              file system to write to.
4859
4860      Returns:
4861          bytes | None: The file's bytes when ``path`` is None.
4862
4863      Examples:
4864          >>> results.to_arrow("part_1.arrow")
4865          >>> data = results.to_arrow()
      """
```

meth from_arrow staticmethod

```
from_arrow(source: Any) -> TransmutationResults
```

Load a result that `to_arrow` saved.

Every getter answers as it did on the saved result, but three of them re-derive from data the run read rather than from what it computed: `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 sha256 of its contents, and each nuclide by the file it was read from (for a library keyword, a folder of the release the run used). On load the chain is assembled from the configured `yamc.transmutation_*` sources and kept only if it is the same data, and each nuclide is read again from its file. Whatever cannot be reattached is reported in one `UserWarning`, and the getters that need it raise `RuntimeError` rather than answer from other data.

Parameters:

Name	Type	Description	Default
`source`	`str | PathLike | bytes`	A file path, or the bytes `to_arrow()` returned.	*required*

Returns:

Name	Type	Description
`TransmutationResults`	`TransmutationResults`	The saved result.

Raises:

Type	Description
`ValueError`	If `source` is not a transmutation results file, or is from another format version.

Examples:

```
>>> results = yani.TransmutationResults.from_arrow("part_1.arrow")
```

Source code in `yani/_core/__init__.pyi`

```
4866  @staticmethod
4867  def from_arrow(source: typing.Any) -> TransmutationResults:
4868      r"""
4869      Load a result that ``to_arrow`` saved.
4870
4871      Every getter answers as it did on the saved result, but three of them
4872      re-derive from data the run read rather than from what it computed:
4873      ``get_production_routes`` and ``get_isomeric_branching`` walk the
4874      transmutation chain, and ``get_reaction_rate_spectrum`` reads the
4875      initial composition's cross sections. The file pins both: the chain by
4876      a sha256 of its contents, and each nuclide by the file it was read
4877      from (for a library keyword, a folder of the release the run used).
4878      On load the chain is assembled from the configured
4879      ``yamc.transmutation_*`` sources and kept only if it is the same data,
4880      and each nuclide is read again from its file. Whatever cannot be
4881      reattached is reported in one ``UserWarning``, and the getters that
4882      need it raise ``RuntimeError`` rather than answer from other data.
4883
4884      Args:
4885          source (str | os.PathLike | bytes): A file path, or the bytes
4886              ``to_arrow()`` returned.
4887
4888      Returns:
4889          TransmutationResults: The saved result.
4890
4891      Raises:
4892          ValueError: If ``source`` is not a transmutation results file, or
4893              is from another format version.
4894
4895      Examples:
4896          >>> results = yani.TransmutationResults.from_arrow("part_1.arrow")
4897      """
```

---

class yani.DoseCoefficients

Fluence-to-dose conversion coefficients (effective dose or ambient dose equivalent H*(10)), returned by :func:`yani.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 `yani/_core/__init__.pyi`

```
689  def as_energy_function(self) -> tuple[builtins.list[builtins.float],
690  builtins.list[builtins.float], builtins.str, builtins.str]:
691      r"""
692      Repackage as the ``(energy, coefficients, units, interpolation)`` tuple
693      accepted by ``Tally(energy_function=...)``. The interpolation is
694      ``"log-log"``, how the tabulation is read everywhere else.
         """
```

---

class yani.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 `yani/_core/__init__.pyi`

```
202  def assessed_activity(self, nuclide: builtins.str) -> builtins.float:
203      r"""
204      The activity actually charged against the limit for one nuclide: its
205      total activity, less any part a parent accounted for.
206
207      Args:
208          nuclide (str): Nuclide name, such as ``"Co60"``.
209      """
```

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 `yani/_core/__init__.pyi`

```
210  def dominant(self, count: builtins.int = 10) -> builtins.list[tuple[builtins.str,
211  builtins.float]]:
212      r"""
213      The nuclides contributing most to the index, largest first.
214
215      Args:
216          count (int): How many to return.
217      """
```

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 `yani/_core/__init__.pyi`

```
217  def to_json(self, path: typing.Optional[builtins.str | os.PathLike | pathlib.Path] =
218  None) -> typing.Optional[bytes]:
219      r"""
220      Save the result as JSON: the limit set, material, index, threshold
221      and units, whether it is clearable or out of scope, and per nuclide
222      the index contributions, activities and limits used, with the
223      defaulted, excluded, credited, uncovered and unlimited nuclides.
224
225      Args:
226          path (str | os.PathLike | None): Where to write. None returns the
227              JSON as bytes instead, for a browser or Pyodide session.
228
229      Returns:
230          bytes | None: The JSON when ``path`` is None.
231
232      Examples:
233          >>> result.to_json("clearance.json")
234          >>> json.loads(result.to_json())["clearable"]
235          True
      """
```

class yani.PhotonCoefficients

A photon coefficient tabulated against energy, returned by
:func:`yani.data.mass_attenuation_coefficient` and
:func:`yani.data.mass_energy_absorption_coefficient`.

Carries the energy grid and the coefficients, and evaluates itself between the tabulated points with the log-log interpolation NIST publishes these for.

attr energy property

```
energy: list[float]
```

Tabulated energies in eV. An absorption edge appears as two energies one ulp apart, carrying the value below and above the edge.

attr coefficients property

```
coefficients: list[float]
```

The coefficient at each tabulated energy, in cm²/g.

attr units property

```
units: str
```

Units of the coefficients (`"cm2/g"`).

meth interpolate

```
interpolate(energy: float) -> float
```

The coefficient at `energy` (eV), log-log interpolated.

Energies outside the tabulated range return the nearest end value rather than extrapolating: the tabulation stops where the data does.

Source code in `yani/_core/__init__.pyi`

```
2444  def interpolate(self, energy: builtins.float) -> builtins.float:
2445      r"""
2446      The coefficient at ``energy`` (eV), log-log interpolated.
2447
2448      Energies outside the tabulated range return the nearest end value rather
2449      than extrapolating: the tabulation stops where the data does.
2450      """
```

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"`, matching :meth:`interpolate`, so a tally weights by the same values; its absorption edges are steps rather than the ringing a cubic spline gives across them.

Source code in `yani/_core/__init__.pyi`

```
2451  def as_energy_function(self) -> tuple[builtins.list[builtins.float],
2452  builtins.list[builtins.float], builtins.str, builtins.str]:
2453      r"""
2454      Repackage as the ``(energy, coefficients, units, interpolation)`` tuple
2455      accepted by ``Tally(energy_function=...)``. The interpolation is
2456      ``"log-log"``, matching :meth:`interpolate`, so a tally weights by the
2457      same values; its absorption edges are steps rather than the ringing a
2458      cubic spline gives across them.
      """
```

---

class yani.PhotonContinuum

One nuclide's decay photon continuum within a material's inventory.

ENDF gives part of some decay spectra as a density over energy rather than as lines: the spontaneous-fission photons of an actinide, or the whole photon emission of a nuclide far from stability, whose lines were never measured. The values are a rate per eV, so they are not line rates and cannot be added to `decay_photon_spectrum()`'s. Their rate is the integral, which `emission_rate` gives. Both follow the `per` argument of `decay_photon_continua()`: for the whole material, or per cm³ or per g.

attr nuclide property

```
nuclide: str
```

The emitting nuclide.

attr energies property

```
energies: list[float]
```

Tabulated energies [eV], ascending.

attr rates property

```
rates: list[float]
```

The emission-rate density at each energy: photons/s/eV for the whole material, or per cm³ or per g following the `per` argument the continuum was requested with.

attr interpolation property

```
interpolation: Optional[str]
```

How `rates` is read between the energies: the ENDF law by name (`"histogram"`, `"linear-linear"`, ...), or None where the data states no law, which data written before the law was stored does.

attr emission_rate property

```
emission_rate: float
```

The emission rate over the whole continuum, its integral read under its law: photons/s for the whole material, or per cm³ or per g following the `per` argument the continuum was requested with.

Raises:

Type	Description
`ValueError`	If the law is not stated, or is one this build does not integrate, or if the energy and rate lists are unpaired, the energies are not finite or descend, or the rates are negative or not finite. The integral is then unknown, and no number is returned in its place.

Nuclear-data uncertainty

`DataUncertainty` is passed to `Material.transmute()`, not read from it; the sigmas and the coverage report it produces come back on `TransmutationResults` above. `Estimate` and `LineEstimate` are what the derived quantities come back as, a nominal value with the ensemble's spread beside it, from `get_activity_uncertainty` and the three getters like it. What the uncertainty covers, and what a sigma of zero does and does not mean, is in Nuclear-data uncertainty.

`class` yani.DataUncertainty

```
DataUncertainty(seed: int = 1, samples: Optional[int] = None, sources:
Optional[Sequence[str]] = None, attribution: bool = False, convergence:
Optional[float] = None)
```

Request nuclear-data uncertainty on a transmutation.

Pass one to :meth:`Material.transmute` and the result carries a standard deviation on every nuclide density alongside the mean.

What it can cover, by source (`DataUncertainty.available_sources()`):

- `"cross_sections"`: the activation cross sections, sampled from the ENDF MF=33 covariance folded against this material's own spectrum, and where the library folder carries the evaluation's resonance parameters (MF=2 and MF=32, `resonance_parameters.arrow`), the parameters themselves: each replica draws them from their covariance, rebuilds the resonance range's cross sections from the draw and Doppler broadens the change to the temperature in use, so the resonance range's uncertainty is exact in the parameters rather than first order. On a transport-coupled run the change is folded against the flux the transport tallied on the nuclide's own energy grid through its resonances, so a self-shielded resonance is weighted with its own flux dip. A nuclide whose parameters cannot be sampled keeps the first-order MF=32 rows of `covariance.arrow`, and the report says which and why;
- `"flux_spectrum"`: the spectrum itself, from the per-bin `flux_std_dev` given on a `Pulse`;
- `"half_life"`: every reachable nuclide's half-life, from the decay data's own standard deviation. A replica's half-lives are used in its solve AND in the activity, decay heat and dose evaluated from it, so a saturated activity (`lambda N = R`) is correctly insensitive to its own half-life rather than inheriting the density's spread;
- `"decay_branching"`: the decay branching ratios of every reachable parent with several modes and a stated sigma, drawn so the parent's total is kept and every ratio stays between zero and it: each mode but one from a logit-normal with the evaluated ratio as its mean and the stated sigma as its standard deviation, the last their complement, coupled so the modes carry the zero-sum covariance of largest entropy that realises every stated sigma (for two or three modes the only one). A small

branch with a large relative sigma is drawn skewed, never below zero. A largest sigma above the sum of the others' is drawn as that sum; a single mode stating no sigma is the complement of the rest when it is the largest or the only way to trade; a mode whose sigma no ratio with its mean can have (`sigma**2 >= b * (T - b)`), and a mode stating none otherwise, are held at their ratios and the rest share what is left. The report names every parent short of its stated sigmas by why;

- `"statistical"` : the Monte Carlo uncertainty of transport-tallied reaction rates, from their per-history covariance, drawn independently for each material since their tallies are separate estimates. It applies to `Model.simulate_transmutation` , as `"flux_spectrum"` applies only to `Material.transmute` ; each call ignores the other's, and the report's `sources` lists what actually applied;
- `"decay_energy"` : each nuclide's mean decay energy, from the sigma the decay data gives each recoverable-heat component (beta, gamma, alpha), or the total's where it gives no split. It moves decay heat only: a decay energy never enters the solve, so the inventory and activity are untouched;
- `"decay_photon_lines"` : each decay photon spectrum's normalisation (FD for lines, FC for a continuum), one draw per spectrum common to 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 E_EM, which follows each replica's drawn lines in place of a `"decay_energy"` draw, the decay heat; no photon enters the solve. The decay data do not state how a nuclide's photon intensities are correlated beyond the normalisation (between the lines' dRI, between a normalisation and its lines, between gamma and x-ray spectra), so each output is evaluated at both ends: `Estimate.std_dev` takes them independent and `Estimate.std_dev_correlated` fully correlated. A D1S dose (`PulseSchedule.time_correct_tally`) draws the normalisations alone;
- `"fission_yield"` : each fissioning parent's independent yields (MT=454), from the DY 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 the yields of a draw are not renormalised. An actinide that borrows another's yields shares its draw. Needs a chain that carries the evaluated yields (`fission_yields/evaluated_yields.arrow`); without one 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 cross sections (the MF=10 partials of the branching subsection, from its `branching_covariance.arrow`). Each stated state's production is drawn on the covariance's own energy grid, folded against the same spectrum and weight as the nominal split, and the split re-made from the draw by the nominal branching rule. Where the list names its ground state only the shares move, and they still sum to one, while the reaction's total stays the cross sections' (MF=33), so the uncertainty of the total is not counted twice. An isomer listed alone, and every `(n,n')` isomer, is an absolute production: it is made at its drawn value, the ground state taking the rest of the total, so its own MF=40 sigma replaces the MF=33 sigma of its reaction that it carries with `"cross_sections"` alone, and can be the smaller. Applies to `Material.transmute` ; a transport run's split comes from tallied partials and is held.

Each cross-section draw is a lognormal multiplier with the covariance's own mean and variance, so a sampled rate is never negative and nothing is floored. The correlated Gaussian deviates are kept (a Gaussian copula), so the ordering between channels is preserved, but the Pearson correlations come out weaker than evaluated as the sigmas grow. Half-lives and decay energies are drawn the same way, one nuclide at a time: the decay data states a mean and a sigma for each and no correlation, so the draws carry exactly what the evaluation states and are never negative. So are the decay photon normalisations, intensities and energies, with the one correlation the data does state, a spectrum's normalisation common to its lines, and the ones it does not state bounded rather than assumed.

Held at their nominal values, with uncertainties of their own that this does not propagate:

- the decay branching ratios `"decay_branching"` does not sample (a sigma wider than the ratio allows, or no sigma to draw), and the per-decay photon lines and decay energy of a drawn parent, which follow its nominal branching;
- the isomeric split of a reaction whose evaluation states no MF=40 covariance on any of its states, listed in `no_isomer_branching_uncertainty`, and MF=40 blocks `"isomer_branching"` cannot use (another evaluation's partner, a partner that is not an MF=10 production, NC, absolute and short-range layouts, a state the branching rule folds no curve for), counted in `isomer_branching_blocks_not_used`;
- covariance correlating an evaluation with one the run does not hold (MAT1 naming a material not in the run, counted in `skipped_cross_material`), covariance with a quantity that is not a cross section (XMF1 not 0 or 3), covariance derived from other sections by an NC block that cannot be derived (LTY 1-4, or an LTY=0 block counted in `skipped_nc`), the covariance of a lumped reaction (MT=851-870) stated in barns, listed in `lumped_covariance_not_assignable`, and the resonance-parameter covariance (MF=32) of a range neither sampled nor in `covariance.arrow`: a library converted before the converter wrote either, or a resonance range whose formalism `endf` does not reconstruct. No library yani reads states an XMF1 other than 0 or 3, an NC block with LTY 1-4, or a lump in barns. What is sampled is each reaction's explicit MF=33 blocks, the resonance parameters where the folder carries them and otherwise the resonance-range blocks the converter derives from MF=32 and writes beside them, the blocks of a lumped reaction for each of its components (one component is the lump; several each take the lump's relative perturbation, moving together), for a level sum with no covariance of its own its levels' (W186 `(n,p)` as the lumped MT 855 of MT 600 and 649), and for a reaction an LTY=0 NC block states as a sum of others (ENDF/B-VIII.1 O16 (n,p) as MT 600 to 603, U235 MT 4 as MT 51 plus the lumped MT 851), the covariance derived from the named reactions' own blocks and the cross blocks between them. A block correlating two evaluations the run both holds (the neutron standards: ENDF/B-VIII.1 U235 fission with U238 fission and capture, Pu239 fission, Au197 capture and Li6 (n,t)) is applied, counted in `applied_cross_material`: the two nuclides are drawn jointly, one draw per replica shared by every material;
- the self-shielding correction, when `self_shielding_chord` or `self_shielding_shape` is given: the shielded flux is built once from the nominal cross sections and reused by every replica, sampled resonance parameters included;

- on a transport run, the flux's response to a perturbed cross section: there is one transport, not one per replica. The tallied values themselves are still drawn by the `"statistical"` source;
- the decay photon spectrum covariance and continuum shape (MF=8 MT=457 LCOV, which no library states), photon attenuation (XCOM), air energy absorption (NIST SRD 126), the ICRP-116 fluence-to-dose coefficients and the contact-dose build-up factor;
- the material's composition, density, natural isotopic abundances and the AME2020 atomic masses used to convert mass fractions;
- any source switched off with `sources`, or with nothing to act on (a spectrum given without `flux_std_dev`). When only some of a material's spectra have one, the entry is `"flux spectrum (spectra without a sigma only)"` and `spectra_without_flux_sigma` gives the count. A rate under a spectrum with a sigma that the flux draw cannot move is named in `flux_rates_without_terms`.

`TransmutationResults.get_data_uncertainty_info` lists every one of these that applied to a material under `not_perturbed`, along with any nuclide whose evaluation carries no covariance, any unstable nuclide whose half-life or decay energy has no stated sigma, and any whose stated half-life or decay-energy sigma no draw can carry.

Parameters:

Name	Type	Description	Default
`seed`	`int`	Base seed. A given nuclide's perturbation in a given replica is a pure function of `(seed, replica, nuclide)`, so the same seed reproduces the same answer regardless of replica count, iteration order, or what else is in the material.	`1`
`samples`	`int`	Fixed replica count. Leave as `None` (the default) to let the solver add replicas, 64 at a time, until every tracked standard deviation is known to `convergence`. A number here bounds cost or reproduces a specific run: exactly that many replicas run, and the info still lists any standard deviation that did not reach `convergence`.	`None`
`sources`	`list[str]`	Which inputs to perturb. `None` (the default) means every source this build implements. Restricting it is how a run isolates one contribution, so that adding a source and watching the inventory sigma grow is a measurement rather than a guess. Naming a source this build cannot perturb **raises**, rather than being ignored. A source that has not landed yet must not look like one that contributed nothing. `DataUncertainty.available_sources()` lists what there is.	`None`
`attribution`	`bool`	Also say where the uncertainty comes from, read with `TransmutationResults.get_uncertainty_breakdown`. Off by default because it costs further solves: one ensemble per source, each source alone, and one deterministic solve per contributor. It changes none of the numbers the run otherwise reports.	`False`
`convergence`	`float`	How well each standard deviation must be known before the run stops: the standard error of the sample standard deviation, relative to it, `SE(sigma) / sigma`. `None` (the default) means 0.05. The standard error allows for a heavy tail (it carries the sample kurtosis), so a skewed output needs more replicas than a Gaussian one, for which a target `t` needs about `1 / (2 t^2) + 1`: 201 at 0.05. It is judged at every step, not only the last, on every nuclide density within 1e-6 of the largest at that step and on the total activity, decay heat and decay photon line rate, leaving out any	`None`

Name	Type	Description	Default
		whose standard deviation is under 0.01% of its value. The photon spectrum line by line and the contact dose are not judged, as their dose quantity and build-up are chosen when they are read. At least 128 replicas run whatever the target, since the kurtosis is too noisy to trust on fewer, and at most 1024: a run that stops there sets `hit_cap` in the info and lists every output, nuclide and step that missed under `unconverged`. Must be between 0 and 1.	

Examples:

```
>>> results = iron.transmute(
...     schedule=sched,
...     data_uncertainty=yani.DataUncertainty(seed=42),
... )
>>> results.get_nuclide_density(iron.id, "Mn56", 1)
1.234e-08
>>> results.get_nuclide_uncertainty(iron.id, "Mn56", 1)
8.7e-10
```

Source code in `yani/_core/__init__.pyi`

```
597  def __new__(cls, seed: builtins.int = 1, samples: typing.Optional[builtins.int] = None,
     sources: typing.Optional[typing.Sequence[builtins.str]] = None, attribution:
     builtins.bool = False, convergence: typing.Optional[builtins.float] = None) ->
     DataUncertainty: ...
```

`attr` attribution `property`

```
attribution: bool
```

Whether the run also says where the uncertainty comes from.

`attr` convergence `property`

```
convergence: float
```

The target for each standard deviation's standard error, relative to it, that an adaptive run stops on.

meth available_sources staticmethod

```
available_sources() -> list[str]
```

The uncertainty sources this build can perturb.

Returns:

Type	Description
`list[str]`	list[str]: Names accepted by the `sources` argument.

Source code in `yani/_core/__init__.pyi`

```
598  @staticmethod
599  def available_sources() -> builtins.list[builtins.str]:
600      r"""
601      The uncertainty sources this build can perturb.
602
603      Returns:
604          list[str]: Names accepted by the ``sources`` argument.
605      """
```

---

class yani.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 yani.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` yani.Nuclide

```
Nuclide(name: Optional[str] = None)
```

Nuclide data container exposed to Python.

Create a new (optionally named) nuclide instance.

Parameters:

Name	Type	Description	Default
`name`	`Optional[str]`	Optional nuclide identifier (e.g. "Li6", "Fe56"). If not supplied you must pass `path` to `load` later.	`None`

Notes

Individual fields (e.g. `name`, `atomic_number`, `available_temperatures`, `loaded_temperatures`) are exposed as read-only attributes via PyO3 getters. Detailed descriptions appear once each in the generated documentation--this summary omits a full per-attribute list to avoid duplication.

Create a new (optionally named) nuclide.

Parameters:

Name	Type	Description	Default
`name`	`Optional[str]`	Optional nuclide identifier (e.g. "Li6", "Fe56"). If not supplied you must pass `path` to `load` later.	`None`

Returns:

Name	Type	Description
`Nuclide`	`Nuclide`	A nuclide object with no data loaded yet.

Source code in `yani/_core/__init__.pyi`

```
2257  def __new__(cls, name: typing.Optional[builtins.str] = None) -> Nuclide:
2258      r"""
2259      Create a new (optionally named) nuclide.
2260
2261      Args:
2262          name (Optional[str]): Optional nuclide identifier (e.g. "Li6", "Fe56"). If not
2263              supplied you must pass `path` to `load` later.
2264
2265      Returns:
2266          Nuclide: A nuclide object with no data loaded yet.
2267      """
```

attr name property

```
name: Optional[str]
```

Name / identifier for the nuclide (e.g. "Li6", "Fe56").

Returns:

Type	Description
`Optional[str]`	Optional[str]: Nuclide name or None if not yet set.

attr element property

```
element: Optional[str]
```

Chemical element symbol (e.g. "Fe").

Returns:

Type	Description
`Optional[str]`	Optional[str]: Element symbol or None if data not loaded.

attr atomic_symbol property

```
atomic_symbol: Optional[str]
```

Atomic symbol (currently same as element symbol).

Returns:

Type	Description
`Optional[str]`	Optional[str]: Atomic symbol string.

attr atomic_number property

```
atomic_number: Optional[int]
```

Proton number Z.

Returns:

Type	Description
`Optional[int]`	Optional[int]: Atomic number.

attr neutron_number property

```
neutron_number: Optional[int]
```

Neutron number N.

Returns:

Type	Description
`Optional[int]`	Optional[int]: Neutron count.

attr mass_number property

```
mass_number: Optional[int]
```

Mass number A = Z + N.

Returns:

Type	Description
`Optional[int]`	Optional[int]: Mass number.

attr atomic_weight_ratio property

```
atomic_weight_ratio: Optional[float]
```

Atomic weight ratio (target mass / neutron mass) from nuclear data file.

Returns:

Type	Description
`Optional[float]`	Optional[float]: Atomic weight ratio.

attr library property

```
library: Optional[str]
```

Originating nuclear data library identifier.

Returns:

Type	Description
`Optional[str]`	Optional[str]: Library name/code.

attr fissionable property

```
fissionable: bool
```

Whether the nuclide is fissionable.

Returns:

Name	Type	Description
`bool`	`bool`	True if fissionable.

attr available_temperatures property

```
available_temperatures: list[str]
```

All temperatures present in the source data file.

Returns:

Type	Description
`list[str]`	List[str]: Temperature labels in Kelvin (e.g. ["293"]).

attr loaded_temperatures property

```
loaded_temperatures: list[str]
```

Temperatures actually loaded into memory (subset of available_temperatures).

Returns:

Type	Description
`list[str]`	List[str]: Loaded temperatures.

attr data_path property

```
data_path: Optional[str]
```

Path to the data file used to populate this nuclide (if known).

Returns:

Type	Description
`Optional[str]`	Optional[str]: Filesystem path or None.

attr reactions property

```
reactions: Any
```

Mapping of temperature -> MT number -> reaction data.

Returns:

Type	Description
`Any`	Dict[str, Dict[int, Dict[str, Any]]]: Nested dictionary. The innermost
`Any`	dictionary has these keys: • cross_section (List[float]) • threshold_idx (int) • interpolation (List[int]) • energy (Optional[List[float]]): Present when reaction has its own grid

attr reaction_mts property

```
reaction_mts: Optional[list[int]]
```

List of MT numbers available for the (first) loaded temperature.

Returns:

Type	Description
`Optional[list[int]]`	Optional[List[int]]: List of MT identifiers or None if no data.

attr energy property

```
energy: Optional[Any]
```

Energy grids by temperature.

Returns:

Type	Description
`Optional[Any]`	Optional[Dict[str, List[float]]]: Map of temperature key to energy grid
`Optional[Any]`	or None if no energy data loaded.

`meth` read_nuclear_data

```
read_nuclear_data(path: Optional[str] = None, temperatures:
Optional[Sequence[str]] = None) -> None
```

Read nuclear data from a file, keyword, or directory.

You can either provide a source explicitly via `path` or rely on the `name` given at construction and the global configuration to resolve it. Providing an explicit `path` will override any global configuration.

The `path` argument accepts: - A file path (e.g. "tests/Li6.arrow") - A keyword (e.g. "tendl-2025", "fendl-3.2d", "endf-b8.1") - A directory path (e.g. "/path/to/neutron/") -- resolves to `<dir>/<name>.arrow` - None -- looks up the path from yani.neutron_cross_section_data

When `temperatures` is provided only those temperatures are loaded while `available_temperatures` always lists every temperature present in the file. The subset actually loaded is stored in `loaded_temperatures`.

Parameters:

Name	Type	Description	Default
`path`	`Optional[str]`	Optional path to the nuclide data file, keyword, or directory. If omitted, the constructor `name` is used to look up the source from yani.neutron_cross_section_data.	`None`
`temperatures`	`Optional[List[str]]`	Temperature strings in Kelvin (e.g. ["293"]). If given only these temperatures are loaded.	`None`

Returns:

Type	Description
`None`	None

Raises:

Type	Description
`ValueError`	If neither `path` nor `name` is available, if the nuclide name is not found in global configuration (when path not provided), or if the data cannot be read / parsed.

Examples:

Compare the same nuclide from different data sources:

```
>>> # Set global default
>>> yani.neutron_cross_section_data = "tendl-2025"
>>>
>>> # Load from global config (will use TENDL)
>>> li6_tendl = yani.Nuclide("Li6")
>>> li6_tendl.read_nuclear_data()
>>>
>>> # Override to use FENDL for comparison
>>> li6_fendl = yani.Nuclide("Li6")
>>> li6_fendl.read_nuclear_data("fendl-3.2d")
>>>
>>> # Use a directory
>>> li6_dir = yani.Nuclide("Li6")
>>> li6_dir.read_nuclear_data("/path/to/neutron/")
>>>
>>> # Use custom local file
>>> li6_custom = yani.Nuclide("Li6")
>>> li6_custom.read_nuclear_data("path/to/custom_Li6.arrow")
```

Source code in `yani/_core/__init__.pyi`

```
2268  def read_nuclear_data(self, path: typing.Optional[builtins.str] = None,
2269  temperatures: typing.Optional[typing.Sequence[builtins.str]] = None) -> None:
2270      r"""
2271      Read nuclear data from a file, keyword, or directory.
2272
2273      You can either provide a source explicitly via `path` or rely on
2274      the `name` given at construction and the global configuration to resolve it.
2275      Providing an explicit `path` will override any global configuration.
2276
2277      The `path` argument accepts:
2278        - A file path (e.g. "tests/Li6.arrow")
2279        - A keyword (e.g. "tendl-2025", "fendl-3.2d", "endf-b8.1")
2280        - A directory path (e.g. "/path/to/neutron/") -- resolves to
2281  `<dir>/<name>.arrow`
2282        - None -- looks up the path from yani.neutron_cross_section_data
2283
2284      When `temperatures` is provided only those temperatures are loaded while
2285      `available_temperatures` always lists every temperature present in the
2286      file. The subset actually loaded is stored in `loaded_temperatures`.
2287
2288      Args:
2289          path (Optional[str]): Optional path to the nuclide data file, keyword,
2290              or directory. If omitted, the constructor `name` is used to look up
2291              the source from yani.neutron_cross_section_data.
2292          temperatures (Optional[List[str]]): Temperature strings in Kelvin (e.g.
2293  ["293"]).
2294              If given only these temperatures are loaded.
2295
2296      Returns:
2297          None
2298
2299      Raises:
2300          ValueError: If neither `path` nor `name` is available, if the nuclide
2301              name is not found in global configuration (when path not provided),
2302              or if the data cannot be read / parsed.
2303
2304      Examples:
2305          Compare the same nuclide from different data sources:
2306
2307          >>> # Set global default
2308          >>> yani.neutron_cross_section_data = "tendl-2025"
2309          >>>
2310          >>> # Load from global config (will use TENDL)
2311          >>> li6_tendl = yani.Nuclide("Li6")
2312          >>> li6_tendl.read_nuclear_data()
2313          >>>
2314          >>> # Override to use FENDL for comparison
2315          >>> li6_fendl = yani.Nuclide("Li6")
2316          >>> li6_fendl.read_nuclear_data("fendl-3.2d")
2317          >>>
2318          >>> # Use a directory
2319          >>> li6_dir = yani.Nuclide("Li6")
2320          >>> li6_dir.read_nuclear_data("/path/to/neutron/")
2321          >>>
2322          >>> # Use custom local file
              >>> li6_custom = yani.Nuclide("Li6")
              >>> li6_custom.read_nuclear_data("path/to/custom_Li6.arrow")
      """
```

meth energy_grid

```
energy_grid(temperature: str) -> Optional[list[float]]
```

Get the energy grid for a specific temperature.

Parameters:

Name	Type	Description	Default
`temperature`	`str`	Temperature key in Kelvin (e.g. "293").	*required*

Returns:

Type	Description
`Optional[list[float]]`	Optional[List[float]]: The energy grid or None if not present.

Source code in `yani/_core/__init__.pyi`

```
2323  def energy_grid(self, temperature: builtins.str) ->
2324  typing.Optional[builtins.list[builtins.float]]:
2325      r"""
2326      Get the energy grid for a specific temperature.
2327
2328      Args:
2329          temperature (str): Temperature key in Kelvin (e.g. "293").
2330
2331      Returns:
2332          Optional[List[float]]: The energy grid or None if not present.
          """
```

meth get_reaction_energy_grid

```
get_reaction_energy_grid(temperature: str, mt: int) -> Optional[list[float]]
```

Get energy grid for a specific temperature and MT number.

Parameters:

Name	Type	Description	Default
`temperature`	`str`	Temperature to use for reaction data.	*required*
`mt`	`int`	ENDF/MT number for the reaction channel.	*required*

Returns:

Type	Description
`Optional[list[float]]`	Optional[List[float]]: Reaction energy grid if present.

Source code in `yani/_core/__init__.pyi`

```
2333  def get_reaction_energy_grid(self, temperature: builtins.str, mt: builtins.int) ->
2334  typing.Optional[builtins.list[builtins.float]]:
2335      r"""
2336      Get energy grid for a specific temperature and MT number.
2337
2338      Args:
2339          temperature (str): Temperature to use for reaction data.
2340          mt (int): ENDF/MT number for the reaction channel.
2341
2342      Returns:
2343          Optional[List[float]]: Reaction energy grid if present.
          """
```

`meth` microscopic_cross_section

```
microscopic_cross_section(reaction: int | str, temperature: Optional[str] = None)
-> tuple[list[float], list[float]]
```

Get microscopic cross section data for a specific reaction and temperature.

Parameters:

Name	Type	Description	Default
`reaction`	`Union[int, str]`	Either an ENDF/MT number (int) or reaction name (str) like "(n,gamma)", "(n,elastic)", "fission", etc.	*required*
`temperature`	`Optional[str]`	Temperature to use. If None, uses the single loaded temperature if only one is available.	`None`

Returns:

Type	Description
`tuple[list[float], list[float]]`	Tuple[List[float], List[float]]: A tuple of (cross_section_values, energy_grid).

Raises:

Type	Description
`Exception`	If temperature not found, reaction not found, multiple temperatures loaded without specifying one, or no data available.

Source code in `yani/_core/__init__.pyi`

```
def microscopic_cross_section(self, reaction: builtins.int | builtins.str,
temperature: typing.Optional[builtins.str] = None) ->
tuple[builtins.list[builtins.float], builtins.list[builtins.float]]:
    r"""
    Get microscopic cross section data for a specific reaction and temperature.

    Args:
        reaction (Union[int, str]): Either an ENDF/MT number (int) or reaction name
(str)
            like "(n,gamma)", "(n,elastic)", "fission", etc.
        temperature (Optional[str]): Temperature to use. If None, uses the single
            loaded temperature if only one is available.

    Returns:
        Tuple[List[float], List[float]]: A tuple of (cross_section_values,
energy_grid).

    Raises:
        Exception: If temperature not found, reaction not found, multiple
temperatures loaded
            without specifying one, or no data available.
    """
```

---

class yani.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 `yani/_core/__init__.pyi`

```
798  def __new__(cls, name: builtins.str) -> Element:
799      r"""
800      Create a new Element.
801
802      Args:
803          name (str): Element symbol (e.g. "Fe", "U", "H").
804      """
```

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 `yani/_core/__init__.pyi`

```
806  def get_nuclides(self) -> builtins.list[builtins.str]:
807      r"""
808      Return list of isotope (nuclide) identifiers for this element.
809
810      Returns:
811          List[str]: Isotope names sorted by mass number (e.g. ["Fe54", "Fe56"]).
812      """
```

class yani.Reaction

```
Reaction(products: Sequence[ReactionProduct], cross_section: Sequence[float],
threshold_idx: int, mt_number: int, q_value: float, energy_grid: Sequence[float],
scatter_in_cm: bool)
```

Lightweight reaction summary exposed to Python.

Represents a single nuclear reaction channel with its products and cross section data.

Source code in `yani/_core/__init__.pyi`

```
2856  def __new__(cls, products: typing.Sequence[ReactionProduct], cross_section:
      typing.Sequence[builtins.float], threshold_idx: builtins.int, mt_number: builtins.int,
      q_value: builtins.float, energy_grid: typing.Sequence[builtins.float], scatter_in_cm:
      builtins.bool) -> Reaction: ...
```

attr products property

```
products: list[ReactionProduct]
```

List of outgoing particles with their yield and secondary distributions.

attr cross_section property

```
cross_section: list[float]
```

Cross section values in barns at each energy point.

attr threshold_idx property

```
threshold_idx: int
```

Index into parent energy grid where this reaction becomes active (threshold).

attr mt_number property

```
mt_number: int
```

ENDF MT number identifying the reaction type (e.g., 2 for elastic, 16 for (n,2n)).

attr q_value property

```
q_value: float
```

Q-value of the reaction in eV (energy released if positive, absorbed if negative).

attr energy_grid property

```
energy_grid: list[float]
```

Reaction-specific energy grid in eV (subset of parent nuclide energy grid).

attr scatter_in_cm property

```
scatter_in_cm: bool
```

Whether scattering is in center-of-mass frame (requires CM to LAB conversion).

---

class yani.TransmutationChain

A parsed transmutation chain.

Wraps the parsed chain data so it can be reused without re-parsing it.

Examples:

```
>>> chain = yani.TransmutationChain.from_arrow("transmutation-endf-b8.1-sfr.arrow")
>>> reduced = chain.reduce(["Fe56"])
>>> reduced.nuclide_names
```

attr nuclide_names property

```
nuclide_names: list[str]
```

List of nuclide names in the chain.

attr half_lives property

```
half_lives: Any
```

Half-lives of the unstable nuclides in the chain.

Returns:

Type	Description
`Any`	dict[str, float]: nuclide name -> half-life in seconds. Stable
`Any`	nuclides (no half-life) are omitted.

`attr` decay_energies `property`

```
decay_energies: Any
```

Mean decay energies of nuclides in the chain.

Returns:

Type	Description
`Any`	dict[str, float]: nuclide name -> mean decay energy in eV. Nuclides
`Any`	with zero decay energy are omitted.

`attr` reactions `property`

```
reactions: Any
```

Neutron-induced reactions of each nuclide in the chain.

Every nuclide in the chain is a key, including those with no reactions. For the decay half of a production route see :attr: `decays` .

Returns:

Type	Description
`Any`	dict[str, list[tuple[str, str | None, float]]]: nuclide name ->
`Any`	list of (reaction_type, target, branching_ratio). `target` is
`Any`	`None` for a channel that names no single product, which in
`Any`	practice means fission. `target` can also be the parent itself,
`Any`	where the evaluation's product has no decay data and the chain
`Any`	builder stepped from it to a nuclide that has: ENDF/B-VIII.1's
`Any`	Pu245 `(n,p)` makes Np245, which has none, and the beta- step
`Any`	from it lands back on Pu245. Such a channel leaves the parent's
`Any`	count unchanged and makes no product, so a production-route walk
`Any`	should skip it, as :meth:`TransmutationResults.get_production_routes`
`Any`	does.

`attr` decays `property`

```
decays: Any
```

Decay modes of each unstable nuclide in the chain.

The other half of a production route. :attr:`reactions` gives the neutron-induced edge, and a walk with only that stops at the first product: it reaches the Hf183 in `W186(n,a)Hf183` and not the `beta-` step that carries it on to Ta183, or the `IT` step that turns an isomer into its own ground state. Multi-step routes are what a decay heat pathway analysis is made of.

Same tuple shape as :attr:`reactions`, and the same underlying `ChainReaction`, so a walk can concatenate the two into one edge list.

Returns:

Name	Type	Description
	`Any`	dict[str, list[tuple[str, str | None, float]]]: nuclide name ->
	`Any`	list of (decay_mode, daughter, branching_ratio), the branchings for
	`Any`	one parent summing to 1. Modes carry the evaluation's own
`spellings`	`Any`	`"beta-"`, `"ec/beta+"`, `"alpha"`, `"IT"`,
	`Any`	`"sf"`, `"p"`, `"n"`, and multi-particle emissions written as
	`Any`	`"beta-,n"`. `daughter` is `None` where the chain models no
`product`	`Any`	on any mode involving spontaneous fission (`"sf"`,
	`Any`	`"ec/beta+,sf"`), whose fission products the chain does not
	`Any`	carry, on a mode whose stored daughter is the parent itself, and where
	`Any`	the mode's product is outside the chain. The branching is kept, so a
	`Any`	parent still decays at its full half-life and that share of it
	`Any`	leaves the chain. Stable nuclides have no decay modes and are
	`Any`	omitted, as they are from :attr:`half_lives`.

`attr` photon_sources `property`

```
photon_sources: Any
```

Decay photon sources of each nuclide that has them (D1S data).

A source is lines or a continuum, and the two are in different units, so each one says which it is. Each ENDF spectrum is its own source, so a nuclide emitting gammas and x-rays has one line source for each. Each has its own normalisation in the data file, which this tuple does not expose, and the tuple does not say which radiation a source is, so the gamma and x-ray sources cannot be

told apart from Python. In data that keeps the spectra apart (`decay/sources.arrow` with a `radiation` column), lines are listed as the evaluation writes them: an energy can appear in both sources, and can repeat within one. Older data holds one merged line source per nuclide, sorted by energy with coincident energies summed. `Material.decay_photon_spectrum()` sums the lines by energy.

Returns:

Type	Description
`Any`	dict[str, list[tuple[str, list[float], list[float], str | None]]]:
`Any`	nuclide name -> one `(type, energies, intensities, interpolation)`
`Any`	per photon source. A `"discrete"` source lists lines, each
`Any`	intensity its emission rate per atom [1/s], and its interpolation is
`Any`	None. A `"tabular"` one is a continuum: each intensity is the
`Any`	emission-rate density per atom [1/s/eV] at that energy, read between
`Any`	energies by `interpolation`, the ENDF law by name (e.g.
`Any`	`"histogram"` or `"linear-linear"`), which is None where the
`Any`	data states no law.

`meth` from_arrow `staticmethod`

```
from_arrow(path: str) -> TransmutationChain
```

Parse a transmutation chain from a split (v2) chain directory that holds a `decay/` subsection subdir, and the `reactions/` and `fission_yields/` ones where the chain has them.

A chain is a directory of files, so it is read from a path only.

Parameters:

Name	Type	Description	Default
`path`	`str`	Path to the chain root directory.	*required*

Examples:

```
>>> chain = yani.TransmutationChain.from_arrow("chain_dir")
```

Source code in `yani/_core/__init__.pyi`

```
3325  @staticmethod
3326  def from_arrow(path: builtins.str) -> TransmutationChain:
3327      r"""
3328      Parse a transmutation chain from a split (v2) chain directory that
3329      holds a `decay/` subsection subdir, and the `reactions/` and
3330      `fission_yields/` ones where the chain has them.
3331
3332      A chain is a directory of files, so it is read from a path only.
3333
3334      Args:
3335          path (str): Path to the chain root directory.
3336
3337      Examples:
3338          >>> chain = yani.TransmutationChain.from_arrow("chain_dir")
3339      """
```

`meth` reduce

```
reduce(nuclides: Sequence[str], level: Optional[int] = None) -> TransmutationChain
```

Reduce the chain to nuclides reachable from seed nuclides.

By default this walks until the reachable set stops growing, so the result holds every nuclide the seeds can ever produce, no matter how long they are irradiated. Pass `level` only to build a deliberately truncated chain (a small test fixture, say); a truncated chain will miss inventory at high fluence.

Parameters:

Name	Type	Description	Default
`nuclides`	`list[str]`	Seed nuclide names.	*required*
`level`	`int | None`	Cap on the number of reaction/decay steps to walk. Default None, meaning walk to saturation.	`None`

Returns:

Name	Type	Description
`TransmutationChain`	`TransmutationChain`	Reduced chain containing only reachable nuclides.

Source code in `yani/_core/__init__.pyi`

```
3340  def reduce(self, nuclides: typing.Sequence[builtins.str], level:
3341  typing.Optional[builtins.int] = None) -> TransmutationChain:
3342      r"""
3343      Reduce the chain to nuclides reachable from seed nuclides.
3344
3345      By default this walks until the reachable set stops growing, so the
3346      result holds every nuclide the seeds can ever produce, no matter how
3347      long they are irradiated. Pass ``level`` only to build a deliberately
3348      truncated chain (a small test fixture, say); a truncated chain will
3349      miss inventory at high fluence.
3350
3351      Args:
3352          nuclides (list[str]): Seed nuclide names.
3353          level (int | None): Cap on the number of reaction/decay steps to
3354              walk. Default None, meaning walk to saturation.
3355
3356      Returns:
3357          TransmutationChain: Reduced chain containing only reachable nuclides.
          """
```

`meth` to_arrow

```
to_arrow(path: str) -> None
```

Write the chain to a split (v2) chain directory with `decay/`, `reactions/` and `fission_yields/` subsection subdirs, which :meth: `from_arrow` reads back. A directory, so a path is required.

Decay modes the chain models no product for (any mode involving spontaneous fission, and a mode whose stored daughter is the parent itself) are written with no target, as :attr: `decays` reports them. Loading the export gives the same chain, but for those rows the file does not repeat the target the source file stored.

Parameters:

Name	Type	Description	Default
`path`	`str`	Output directory path.	*required*

Source code in `yani/_core/__init__.pyi`

```
3363  def to_arrow(self, path: builtins.str) -> None:
3364      r"""
3365      Write the chain to a split (v2) chain directory with `decay/`,
3366      `reactions/` and `fission_yields/` subsection subdirs, which
3367      :meth:`from_arrow` reads back. A directory, so a path is required.
3368
3369      Decay modes the chain models no product for (any mode involving
3370      spontaneous fission, and a mode whose stored daughter is the parent
3371      itself) are written with no target, as :attr:`decays` reports them.
3372      Loading the export gives the same chain, but for those rows the file
3373      does not repeat the target the source file stored.
3374
3375      Args:
3376          path (str): Output directory path.
3377      """
```

func yani.cache_dir

```
cache_dir() -> str
```

The directory downloaded nuclear data is cached in.

A folder in the platform's per-user cache directory (under `~/.cache` on Linux, or `$XDG_CACHE_HOME` when that is set; `~/Library/Caches` on macOS; `%LOCALAPPDATA%` on Windows), shared by both wheels. Inside it each library keeps one folder per published release, `<keyword>/<release>/`, holding that release's `manifest.json` and the files downloaded from it, each verified against the manifest. Read only: to use data from somewhere else, point the data source at a local directory instead.

Returns:

Type	Description
`str`	The cache directory as a string, whether or not it exists yet.

Raises:

Type	Description
`RuntimeError`	If the platform resolves no home directory.

Source code in `yani/_core/__init__.pyi`

```
5001  def cache_dir() -> builtins.str:
5002      r"""
5003      The directory downloaded nuclear data is cached in.
5004
5005      A folder in the platform's per-user cache directory (under ``~/.cache`` on
5006      Linux, or ``$XDG_CACHE_HOME`` when that is set; ``~/Library/Caches`` on
5007      macOS; ``%LOCALAPPDATA%`` on Windows), shared by both wheels. Inside it
5008      each library keeps one folder per published release,
5009      ``<keyword>/<release>/``, holding that release's ``manifest.json`` and the
5010      files downloaded from it, each verified against the manifest. Read only:
5011      to use data from somewhere else, point the data source at a local
5012      directory instead.
5013
5014      Returns:
5015          The cache directory as a string, whether or not it exists yet.
5016
5017      Raises:
5018          RuntimeError: If the platform resolves no home directory.
5019      """
```

func yani.available_nuclear_data_libraries

```
available_nuclear_data_libraries() -> dict
```

What each hosted nuclear data library provides, keyed by library keyword.

Read from each library's release manifest, which lists every file the release publishes, so this downloads no data: only the small release pointer and the manifest, which are cached after the first call. The release is the one a calculation would read now, or, with the data origin unreachable, the newest cached one.

Every list is complete, so an empty list means the library has none of that: `"photon": []` says it publishes no photo-atomic data, not that it was not looked at. A library that could not be resolved at all has `None` for every entry and its reason under `"error"`.

Each library's entry is a dict:

- `release`: the release identifier, and `offline`: whether it is the newest cached release because the origin could not be reached;
- `neutron`: nuclides with neutron cross sections;
- `neutron_covariances`: per kind, the nuclides that carry it: `cross_section` (MF=33), `resonance_parameters` (MF=32), `angular_distribution` (MF=34), `nubar` (MF=31) and `fission_spectrum` (MF=35);
- `thermal_scattering`: thermal scattering tables;
- `photon`: elements with photo-atomic data, what photon transport reads;

- `photonuclear`: targets with photonuclear data, a different thing from `photon`;
- `decay`, `reactions`, `fission_yields`, `isomeric_branching`: whether the library publishes that transmutation subsection;
- `sources`: the upstream files the release was built from, each a dict with `name`, `bytes` and `sha256`.

Returns:

Type	Description
`dict`	dict[str, dict]: Per library keyword, what it provides.

Examples:

```
>>> libraries = yani.available_nuclear_data_libraries()
>>> list(libraries)
['tendl-2025', 'tendl-2017', 'fendl-3.2d', 'endf-b8.1', 'jeff-4.0', 'jendl-5.0',
'irdff-ii']
>>> libraries["jeff-4.0"]["photon"]
[]
>>> "Fe56" in libraries["endf-b8.1"]["neutron_covariances"]["cross_section"]
True
```

Source code in `yani/_core/__init__.pyi`

```
def available_nuclear_data_libraries() -> dict:
    r"""
    What each hosted nuclear data library provides, keyed by library keyword.

    Read from each library's release manifest, which lists every file the
    release publishes, so this downloads no data: only the small release
    pointer and the manifest, which are cached after the first call. The
    release is the one a calculation would read now, or, with the data
    origin unreachable, the newest cached one.

    Every list is complete, so an empty list means the library has none of
    that: ``"photon": []`` says it publishes no photo-atomic data, not that it
    was not looked at. A library that could not be resolved at all has
    ``None`` for every entry and its reason under ``"error"``.

    Each library's entry is a dict:

    - ``release``: the release identifier, and ``offline``: whether it is the
      newest cached release because the origin could not be reached;
    - ``neutron``: nuclides with neutron cross sections;
    - ``neutron_covariances``: per kind, the nuclides that carry it:
      ``cross_section`` (MF=33), ``resonance_parameters`` (MF=32),
      ``angular_distribution`` (MF=34), ``nubar`` (MF=31) and
      ``fission_spectrum`` (MF=35);
    - ``thermal_scattering``: thermal scattering tables;
    - ``photon``: elements with photo-atomic data, what photon transport
      reads;
    - ``photonuclear``: targets with photonuclear data, a different thing
      from ``photon``;
    - ``decay``, ``reactions``, ``fission_yields``, ``isomeric_branching``:
      whether the library publishes that transmutation subsection;
    - ``sources``: the upstream files the release was built from, each a dict
      with ``name``, ``bytes`` and ``sha256``.

    Returns:
        dict[str, dict]: Per library keyword, what it provides.

    Examples:
        >>> libraries = yani.available_nuclear_data_libraries()
        >>> list(libraries)
        ['tendl-2025', 'tendl-2017', 'fendl-3.2d', 'endf-b8.1', 'jeff-4.0', 'jendl-
5.0', 'irdff-ii']
        >>> libraries["jeff-4.0"]["photon"]
        []
        >>> "Fe56" in libraries["endf-b8.1"]["neutron_covariances"]["cross_section"]
        True
    """
```

func yani.verify_library

```
verify_library(path: str) -> dict
```

Check a nuclear-data library folder against its `manifest.json`.

Re-hashes every file the manifest lists and compares its size and sha256 with the manifest, then looks for files the manifest does not list. Loading data from a library folder only checks the sizes of the files it reads; this is the full check, for a library kept for offline use or built locally.

Parameters

path : str A library folder: `manifest.json` beside the `neutron/`, `photon/`, `photonuclear/` and `transmutation/` folders it describes, laid out as each release in :func:`cache_dir` is. A release folder in the cache holds only the files runs have downloaded, so the rest are reported missing.

Returns

dict `keyword`, `release`, `format_version` and `manifest_sha256` identify the release the manifest describes, and `file_count` and `total_bytes` count what it lists. `missing` (listed but absent), `changed` (listed, with another size or sha256) and `unlisted` (in the release's `<Name>.arrow/` folders, or a particle's `index.json`, but not listed) are sorted paths relative to the folder. `ok` is `True` when all three are empty.

```
An unlisted file fails the check: results record the manifest's sha256
as the identity of the library a run read, and a file the manifest
does not list is data that identity does not cover. Files outside the
release's folders, such as a local build's ``.build/`` folder, are not
looked at.
```

Raises

RuntimeError If `path` is not a folder or holds no `manifest.json`, if the manifest is not a version 1 release manifest or is in a `format_version` this build does not read, or if a listed file cannot be read.

Source code in `yani/_core/__init__.pyi`

```
def verify_library(path: builtins.str) -> dict:
    r"""
    Check a nuclear-data library folder against its ``manifest.json``.

    Re-hashes every file the manifest lists and compares its size and sha256
    with the manifest, then looks for files the manifest does not list.
    Loading data from a library folder only checks the sizes of the files it
    reads; this is the full check, for a library kept for offline use or built
    locally.

    Parameters
    ----------
    path : str
        A library folder: ``manifest.json`` beside the ``neutron/``,
        ``photon/``, ``photonuclear/`` and ``transmutation/`` folders it
        describes, laid out as each release in :func:`cache_dir` is. A release
        folder in the cache holds only the files runs have downloaded, so the
        rest are reported missing.

    Returns
    -------
    dict
        ``keyword``, ``release``, ``format_version`` and ``manifest_sha256``
        identify the release the manifest describes, and ``file_count`` and
        ``total_bytes`` count what it lists. ``missing`` (listed but absent),
        ``changed`` (listed, with another size or sha256) and ``unlisted``
        (in the release's ``<Name>.arrow/`` folders, or a particle's
        ``index.json``, but not listed) are sorted paths relative to the
        folder. ``ok`` is ``True`` when all three are empty.

        An unlisted file fails the check: results record the manifest's sha256
        as the identity of the library a run read, and a file the manifest
        does not list is data that identity does not cover. Files outside the
        release's folders, such as a local build's ``.build/`` folder, are not
        looked at.

    Raises
    ------
    RuntimeError
        If ``path`` is not a folder or holds no ``manifest.json``, if the
        manifest is not a version 1 release manifest or is in a
        ``format_version`` this build does not read, or if a listed file
        cannot be read.
    """
```

func yani.lookup_neutron_cross_section_data

```
lookup_neutron_cross_section_data(nuclide: str) -> Optional[str]
```

Look up the cross-section data path for a single nuclide, applying the global default if no per-nuclide entry is set.

Source code in `yani/_core/__init__.pyi`

```
5570  def lookup_neutron_cross_section_data(nuclide: builtins.str) ->
5571  typing.Optional[builtins.str]:
5572      r"""
5573      Look up the cross-section data path for a single nuclide, applying the global
5574      default if no per-nuclide entry is set.
      """
```

func yani.set_neutron_cross_section_data_entry

```
set_neutron_cross_section_data_entry(nuclide_or_keyword: str, path: Optional[str] =
None) -> None
```

Set the cross-section data path for a single nuclide, or set the global default if `path` is omitted and `nuclide_or_keyword` is a library keyword / directory.

Source code in `yani/_core/__init__.pyi`

```
5770  def set_neutron_cross_section_data_entry(nuclide_or_keyword: builtins.str, path:
5771  typing.Optional[builtins.str] = None) -> None:
5772      r"""
5773      Set the cross-section data path for a single nuclide, or set the global default
5774      if ``path`` is omitted and ``nuclide_or_keyword`` is a library keyword /
5775      directory.
      """
```

func yani.lookup_photon_cross_section_data

```
lookup_photon_cross_section_data(element: str) -> Optional[str]
```

Resolve one photon element from the explicit photon selection only.

Source code in `yani/_core/__init__.pyi`

```
5576  def lookup_photon_cross_section_data(element: builtins.str) ->
5577  typing.Optional[builtins.str]:
5578      r"""
5579      Resolve one photon element from the explicit photon selection only.
      """
```

func yani.group_structure

```
group_structure(name: str) -> list[float]
```

Energy boundaries of a built-in group structure, in eV.

Parameters:

Name	Type	Description	Default
`name`	`str`	Name of the group structure, case-sensitive, one of the names :func: `group_structure_names` returns (e.g. `"UKAEA-1102"`).	*required*

Returns:

Type	Description
`list[float]`	list[float]: The bin edges in eV, ascending. One more than the number
`list[float]`	of groups, so `len(edges) - 1` groups.

Raises:

Type	Description
`ValueError`	If the name is not a built-in group structure. The message lists the ones that are.

Examples:

```
>>> import yani
>>> edges = yani.group_structure("UKAEA-1102")
>>> len(edges), edges[0], edges[-1]
(1103, 1e-05, 1000000000.0)
>>> # the energies a 1102-group spectrum belongs to
>>> midpoints = [(lo + hi) / 2 for lo, hi in zip(edges, edges[1:])]
```

Source code in `yani/_core/__init__.pyi`

```
5524  def group_structure(name: builtins.str) -> builtins.list[builtins.float]:
5525      r"""
5526      Energy boundaries of a built-in group structure, in eV.
5527
5528      Args:
5529          name: Name of the group structure, case-sensitive, one of the names
5530              :func:`group_structure_names` returns (e.g. ``"UKAEA-1102"``).
5531
5532      Returns:
5533          list[float]: The bin edges in eV, ascending. One more than the number
5534          of groups, so ``len(edges) - 1`` groups.
5535
5536      Raises:
5537          ValueError: If the name is not a built-in group structure. The message
5538                      lists the ones that are.
5539
5540      Examples:
5541          >>> import yani
5542          >>> edges = yani.group_structure("UKAEA-1102")
5543          >>> len(edges), edges[0], edges[-1]
5544          (1103, 1e-05, 1000000000.0)
5545          >>> # the energies a 1102-group spectrum belongs to
5546          >>> midpoints = [(lo + hi) / 2 for lo, hi in zip(edges, edges[1:])]
5547      """
```

func yani.group_structure_names

```
group_structure_names() -> list[str]
```

Names of the built-in group structures.

Neutron structures first by increasing group count, then photon. Every name except `"CCFE-24-PHOTON"` carries the same boundaries as OpenMC's `openmc.mgxs.GROUP_STRUCTURES` entry of that name.

Returns:

Type	Description
`list[str]`	list[str]: The names accepted by :func:`group_structure`, and anywhere
`list[str]`	else a group structure can be named.

Examples:

```
>>> import yani
>>> names = yani.group_structure_names()
>>> "CCFE-709" in names
True
>>> {n: len(yani.group_structure(n)) - 1 for n in names[:2]}
{'XMAS-172': 172, 'VITAMIN-J-175': 175}
```

Source code in `yani/_core/__init__.pyi`

```
5549  def group_structure_names() -> builtins.list[builtins.str]:
5550      r"""
5551      Names of the built-in group structures.
5552
5553      Neutron structures first by increasing group count, then photon. Every name
5554      except ``"CCFE-24-PHOTON"`` carries the same boundaries as OpenMC's
5555      ``openmc.mgxs.GROUP_STRUCTURES`` entry of that name.
5556
5557      Returns:
5558          list[str]: The names accepted by :func:`group_structure`, and anywhere
5559          else a group structure can be named.
5560
5561      Examples:
5562          >>> import yani
5563          >>> names = yani.group_structure_names()
5564          >>> "CCFE-709" in names
5565          True
5566          >>> {n: len(yani.group_structure(n)) - 1 for n in names[:2]}
5567          {'XMAS-172': 172, 'VITAMIN-J-175': 175}
5568      """
```

Making data

func yani.generate_library

```
generate_library(keyword: str, path: str, *, parts: Optional[Sequence[str]] = None,
nuclides: Optional[Sequence[str]] = None, temperatures: Optional[Sequence[float]] =
None, njoy_exec: Optional[str] = None, downloads: Optional[str] = None) -> dict
```

Build a whole nuclear-data library from its evaluated sources into a folder.

Downloads the library's source archives (each checked against the size and sha256 it is pinned to), unpacks them, converts every evaluation, builds the transmutation chain, checks the result and writes `manifest.json` last. The folder is laid out as a published release, so it serves every setting:

.. code-block:: python

```
generate_library("endf-b8.1", "/data/endf-b8.1")
neutron_cross_section_data = "/data/endf-b8.1"
transmutation_decay_data = "/data/endf-b8.1"
```

Running it again resumes: finished folders are kept and only the rest are converted. A folder holds one library built by one build of this package: the same version compiled from the same core commit.

The build is recorded in the release. `manifest.json` carries `provenance.builder` (`"yani-python <version>"`), `provenance.builder_commit` (the core commit this extension was compiled from, :data: `__commit__`) and `provenance.builder_dirty` (:data: `__commit_dirty__`), both `null` when it was built outside a git checkout, beside `provenance.njoy`. Every `version.json` and `provenance.json` carries the same as one `converter` stamp, such as `"yani-python 0.22.1 (core 250c439e...)"`, and the manifest lists each distinct stamp as `converters`.

An extension compiled from a tree with uncommitted changes (`__commit_dirty__` is `True`) writes data no commit describes, so it builds only a partial library (`parts`, `nuclides` or `temperatures` given), which is never published. Commit the changes and rebuild to build a whole library.

Some hosts serve their archives only to a web browser (the IAEA's FENDL and IRDFF-II files, JAEA's JENDL-5 files). The error then lists each file with its link; download them with a browser and run again. They are found in the Downloads folder, in `downloads`, or in `<path>/.build/downloads/`.

Parameters

keyword : str The library: `"endf-b8.1"`, `"jeff-4.0"`, `"jendl-5.0"`, `"tendl-2025"`, `"tendl-2017"`, `"fendl-3.2d"` or `"irdff-ii"`. path : str The folder to build in, created if absent. It must be new, empty or an earlier build of the same library, and not inside :func:`cache_dir`. parts : list of str, optional What to build, of `"neutron"`, `"photon"`, `"photonuclear"`, `"thermal"`, `"decay"`, `"reactions"`, `"fission_yields"` and `"branching"`. The default is every part the library's published release carries. Only `"neutron"` and `"thermal"` run NJOY, so a chain alone builds in minutes. `"thermal"` is the thermal scattering tables (`thermal/c_H_in_H2O.arrow/` and so on), which only `"endf-b8.1"`, `"jeff-4.0"` and `"jendl-5.0"` publish. nuclides : list of str, optional Only these nuclides (`"Fe56"`, `"Am242_m1"`) and elements (`"Fe"`, meaning every Fe evaluation and the Fe photon data), and thermal tables by name (`"c_H_in_H2O"`) or by what they bind to (`"H"` is every table binding hydrogen). The transmutation chain is always built whole. A name the library does not have is refused. temperatures : list of float, optional Neutron temperatures in Kelvin. The default is 250, 293.6, 600, 900, 1200 and 2500 K. The 0 K grid is always kept. njoy_exec : str, optional The NJOY executable. The default is the `njoy` installed beside this Python (`pip install njoy2016`), then the one on PATH. FENDL needs the IAEA-NDS fork of NJOY, and a refusal says how to build it. downloads : str, optional A folder of source archives fetched by hand, searched first.

Returns

dict `keyword`, `release` (the date the build started, the folder's release stamp), `path`, `parts`, `converted` and `reused` (sorted `"<particle>/<Name>"` keys), `not_converted` (the evaluations the library's recipe declares NJOY cannot process, with the reason each failed), `stale_declarations` (declared ones that converted after all), `njoy` (the NJOY version used, or `None`), `chain_nuclides`, `manifest_sha256`, `file_count` and `total_bytes`.

Raises

ValueError An unknown keyword or part, or bad temperatures. RuntimeError The build cannot start (an extension compiled from uncommitted changes asked for a whole library, the folder or another build in it, the NJOY, the disk space, or source files that need a browser), or did not finish: evaluations that failed and are not declared, a gate that failed, or Ctrl-C. Nothing finished is lost, and running again retries only what is left. Each failed evaluation's reason is in `<path>/.build/failed/`.

Source code in `yani/_core/__init__.pyi`

```
def generate_library(keyword: builtins.str, path: builtins.str, *, parts:
typing.Optional[typing.Sequence[builtins.str]] = None, nuclides:
typing.Optional[typing.Sequence[builtins.str]] = None, temperatures:
typing.Optional[typing.Sequence[builtins.float]] = None, njoy_exec:
typing.Optional[builtins.str] = None, downloads: typing.Optional[builtins.str] = None)
-> dict:
    r"""
    Build a whole nuclear-data library from its evaluated sources into a folder.

    Downloads the library's source archives (each checked against the size and
    sha256 it is pinned to), unpacks them, converts every evaluation, builds the
    transmutation chain, checks the result and writes ``manifest.json`` last. The
    folder is laid out as a published release, so it serves every setting:

    .. code-block:: python

        generate_library("endf-b8.1", "/data/endf-b8.1")
        neutron_cross_section_data = "/data/endf-b8.1"
        transmutation_decay_data = "/data/endf-b8.1"

    Running it again resumes: finished folders are kept and only the rest are
    converted. A folder holds one library built by one build of this package:
    the same version compiled from the same core commit.

    The build is recorded in the release. ``manifest.json`` carries
    ``provenance.builder`` (``"yani-python <version>"``),
    ``provenance.builder_commit`` (the core commit this extension was compiled
    from, :data:`__commit__`) and ``provenance.builder_dirty``
    (:data:`__commit_dirty__`), both ``null`` when it was built outside a git
    checkout, beside ``provenance.njoy``. Every ``version.json`` and
    ``provenance.json`` carries the same as one ``converter`` stamp, such as
    ``"yani-python 0.22.1 (core 250c439e...)"``, and the manifest lists each
    distinct stamp as ``converters``.

    An extension compiled from a tree with uncommitted changes
    (``__commit_dirty__`` is ``True``) writes data no commit describes, so it
    builds only a partial library (``parts``, ``nuclides`` or ``temperatures``
    given), which is never published. Commit the changes and rebuild to build
    a whole library.

    Some hosts serve their archives only to a web browser (the IAEA's FENDL and
    IRDFF-II files, JAEA's JENDL-5 files). The error then lists each file with
    its link; download them with a browser and run again. They are found in the
    Downloads folder, in ``downloads``, or in ``<path>/.build/downloads/``.

    Parameters
    ----------
    keyword : str
        The library: ``"endf-b8.1"``, ``"jeff-4.0"``, ``"jendl-5.0"``,
        ``"tendl-2025"``, ``"tendl-2017"``, ``"fendl-3.2d"`` or ``"irdff-ii"``.
    path : str
        The folder to build in, created if absent. It must be new, empty or an
        earlier build of the same library, and not inside :func:`cache_dir`.
    parts : list of str, optional
        What to build, of ``"neutron"``, ``"photon"``, ``"photonuclear"``,
        ``"thermal"``, ``"decay"``, ``"reactions"``, ``"fission_yields"`` and
        ``"branching"``. The default is every part the library's published
        release carries. Only ``"neutron"`` and ``"thermal"`` run NJOY, so a
        chain alone builds in minutes. ``"thermal"`` is the thermal scattering
        tables (``thermal/c_H_in_H2O.arrow/`` and so on), which only
        ``"endf-b8.1"``, ``"jeff-4.0"`` and ``"jendl-5.0"`` publish.
    nuclides : list of str, optional
        Only these nuclides (``"Fe56"``, ``"Am242_m1"``) and elements (``"Fe"``,
        meaning every Fe evaluation and the Fe photon data), and thermal tables
```

```
5447          by name (``"c_H_in_H2O"``) or by what they bind to (``"H"`` is every
5448          table binding hydrogen). The transmutation chain is always built whole.
5449          A name the library does not have is refused.
5450      temperatures : list of float, optional
5451          Neutron temperatures in Kelvin. The default is 250, 293.6, 600, 900, 1200
5452          and 2500 K. The 0 K grid is always kept.
5453      njoy_exec : str, optional
5454          The NJOY executable. The default is the ``njoy`` installed beside this
5455          Python (``pip install njoy2016``), then the one on PATH. FENDL needs the
5456          IAEA-NDS fork of NJOY, and a refusal says how to build it.
5457      downloads : str, optional
5458          A folder of source archives fetched by hand, searched first.
5459
5460      Returns
5461      -------
5462      dict
5463          ``keyword``, ``release`` (the date the build started, the folder's
5464          release stamp), ``path``, ``parts``, ``converted`` and ``reused`` (sorted
5465          ``"<particle>/<Name>"`` keys), ``not_converted`` (the evaluations the
5466          library's recipe declares NJOY cannot process, with the reason each
5467          failed), ``stale_declarations`` (declared ones that converted after all),
5468          ``njoy`` (the NJOY version used, or ``None``), ``chain_nuclides``,
5469          ``manifest_sha256``, ``file_count`` and ``total_bytes``.
5470
5471      Raises
5472      ------
5473      ValueError
5474          An unknown keyword or part, or bad temperatures.
5475      RuntimeError
5476          The build cannot start (an extension compiled from uncommitted changes
5477          asked for a whole library, the folder or another build in it, the NJOY,
5478          the disk space, or source files that need a browser), or did not
              finish: evaluations that failed and are not declared, a gate that
              failed, or Ctrl-C. Nothing finished is lost, and running again retries
              only what is left. Each failed evaluation's reason is in
              ``<path>/.build/failed/``.
      """
```

func yani.convert_neutron_xs

```
convert_neutron_xs(input_path: str, output_dir: str, source_format: str = 'endf',
njoy_exec: str = 'njoy', temperatures: Optional[Sequence[float]] = None, library: str
= '', data_version: str = '', created_utc: Optional[str] = None, covariance: bool =
False) -> str
```

Convert an evaluation into the cross-section data an activation calculation reads.

Writes a `{nuclide}.arrow/` directory holding `nuclide.arrow`, `energy.arrow`, `reactions.arrow` and `version.json`, which is exactly what a transport-free reaction-rate collapse reads. The secondary distributions and lookup tables a transport run needs are deliberately not written: an activation calculation never touches them, and building them from an ACE table would cost heating data it does not carry.

Parameters

input_path : str An ENDF evaluation, or an ACE table when `source_format="ace"`. output_dir : str Directory to write `{nuclide}.arrow/` into. With `source_format="endf"`, NJOY also works in a hidden directory here, so this disk needs room for its tapes; the directory is removed when the conversion ends. source_format : {"endf", "ace"} `"endf"` runs NJOY to reconstruct resonances and Doppler broaden, which is the documented route and the only one that can reach an arbitrary temperature. `"ace"` reads a table you already have and needs no NJOY, at the cost of being stuck with the temperature it was processed at. njoy_exec : str, optional The NJOY executable, for `source_format="endf"`. Note that FENDL is processed with the IAEA-NDS fork and upstream NJOY does not reproduce the official release; both builds succeed, so the wrong choice is quiet. temperatures : list[float], optional Temperatures in Kelvin. Defaults to NJOY's own default. Ignored for `source_format="ace"`. library, data_version, created_utc Recorded in `version.json`. `data_version` identifies the published release and is what a consumer compares a cached copy against. covariance : bool Also write `covariance.arrow`, the MF=33 cross-section covariance, and the evaluation's other covariance sections where it has them: `angular_covariance.arrow` (MF=34), `nubar_covariance.arrow` (MF=31), `spectrum_covariance.arrow` (MF=35) and `resonance_parameters.arrow` (the MF=2 resonance parameters with their MF=32 covariance). Off by default: the matrices are large and only an uncertainty calculation reads them. Requires `source_format="endf"` -- MF=33 is not carried through ACER, so asking for it from an ACE table raises.

Returns

str Path to the directory written.

Source code in `yani/_core/__init__.pyi`

```
def convert_neutron_xs(input_path: builtins.str, output_dir: builtins.str,
source_format: builtins.str = 'endf', njoy_exec: builtins.str = 'njoy', temperatures:
typing.Optional[typing.Sequence[builtins.float]] = None, library: builtins.str = '',
data_version: builtins.str = '', created_utc: typing.Optional[builtins.str] = None,
covariance: builtins.bool = False) -> builtins.str:
    r"""
    Convert an evaluation into the cross-section data an activation
    calculation reads.

    Writes a ``{nuclide}.arrow/`` directory holding ``nuclide.arrow``,
    ``energy.arrow``, ``reactions.arrow`` and ``version.json``, which is exactly
    what a transport-free reaction-rate collapse reads. The secondary distributions and
    lookup tables a transport run needs are deliberately not written: an
    activation calculation never touches them, and building them from an ACE
    table would cost heating data it does not carry.

    Parameters
    ----------
    input_path : str
        An ENDF evaluation, or an ACE table when ``source_format="ace"``.
    output_dir : str
        Directory to write ``{nuclide}.arrow/`` into. With
        ``source_format="endf"``, NJOY also works in a hidden directory here,
        so this disk needs room for its tapes; the directory is removed when
        the conversion ends.
    source_format : {"endf", "ace"}
        ``"endf"`` runs NJOY to reconstruct resonances and Doppler broaden,
        which is the documented route and the only one that can reach an
        arbitrary temperature. ``"ace"`` reads a table you already have and
        needs no NJOY, at the cost of being stuck with the temperature it was
        processed at.
    njoy_exec : str, optional
        The NJOY executable, for ``source_format="endf"``. Note that FENDL is
        processed with the IAEA-NDS fork and upstream NJOY does not reproduce
        the official release; both builds succeed, so the wrong choice is quiet.
    temperatures : list[float], optional
        Temperatures in Kelvin. Defaults to NJOY's own default. Ignored for
        ``source_format="ace"``.
    library, data_version, created_utc
        Recorded in ``version.json``. ``data_version`` identifies the published
        release and is what a consumer compares a cached copy against.
    covariance : bool
        Also write ``covariance.arrow``, the MF=33 cross-section covariance,
        and the evaluation's other covariance sections where it has them:
        ``angular_covariance.arrow`` (MF=34), ``nubar_covariance.arrow``
        (MF=31), ``spectrum_covariance.arrow`` (MF=35) and
        ``resonance_parameters.arrow`` (the MF=2 resonance parameters with
        their MF=32 covariance). Off by default: the matrices are large and
        only an uncertainty
        calculation reads them. Requires ``source_format="endf"`` -- MF=33 is
        not carried through ACER, so asking for it from an ACE table raises.

    Returns
    -------
    str
        Path to the directory written.
    """
```

func yani.convert_transmutation

```
convert_transmutation(decay_files: Sequence[str], fpy_files: Sequence[str],
neutron_files: Sequence[str], output_path: str, library: str = '', decay_library: str
= '', data_version: str = '', created_utc: Optional[str] = None, reactions:
Optional[Sequence[str]] = None, branch_ratios: Optional[str] = None, subsections:
Optional[Sequence[str]] = None, decay_fill_files: Sequence[str] = [],
decay_fill_library: str = '') -> int
```

Convert ENDF decay, fission product yield and neutron evaluations into a transmutation Arrow directory.

Parameters

decay_files, fpy_files, neutron_files : list[str] ENDF inputs. All three are needed to build a network: the decay data says what decays into what, the neutron data what transmutes into what, and the fission yields what fission produces. A file may hold one evaluation or a whole tape of them, such as JEFF-4.0's decay and fission yield tapes; every material in it is read. output_path : str Directory to write. Created if absent, merged into if it already holds other subsections of the same library. library : str Library name recorded in the provenance, e.g. `"endf-b8.1"`. data_version : str Identifier of the published release. This is what a consumer compares a cached copy against, so a rebuild that changes values must change this. It is not the converter version: two rebuilds from one converter are different data. created_utc : str, optional Timestamp recorded in the provenance. Supply one for a reproducible directory; left unset, the current time is used. branch_ratios : str, optional Path to a JSON branching table in `openmc_data` shape, `{reaction: {parent: {target: fraction}}}`. Splits a reaction between a ground state and its metastable partners. Without it every `(n,gamma)` gives its whole rate to the ground state and the metastable product is absent from the network entirely. subsections : list[str], optional Which of `decay`, `reactions` and `fission_yields` to write. Defaults to all three. A caller assembling a chain from more than one library writes each subsection from the library that has it, and the manifest merges rather than being overwritten. reactions : list[str], optional Reaction names to follow. Defaults to every reaction the chain builder knows, not the six-name short set, so nothing is silently left out. decay_fill_files : list[str], optional Decay evaluations from a second library, read only to replace the placeholder average decay energies in `decay_files`. Some libraries write a stand-in for nuclides nobody has evaluated: a third of each beta or electron-capture branch's Q to the light particles and a third to the photons, which for an electron-capture emitter can be several times the recoverable energy (Sn111 in ENDF/B-VIII.1 is 1.63 MeV per decay against 0.69 MeV from its decay scheme). A placeholder is replaced only where the second library has an evaluated decay scheme for the same nuclide with a half-life within 25% of the first's. Half-lives and decay modes are never touched. The decay subsection's `provenance.json` lists every placeholder and every replacement, with or without a fill. A tape is read whole, as for `decay_files`. decay_fill_library : str, optional The library `decay_fill_files` came from, e.g. `"jendl-5.0"`. Required with `decay_fill_files`.

Returns

int The number of nuclides in the network written.

Source code in `yani/_core/__init__.pyi`

```
def convert_transmutation(decay_files: typing.Sequence[builtins.str], fpy_files:
typing.Sequence[builtins.str], neutron_files: typing.Sequence[builtins.str],
output_path: builtins.str, library: builtins.str = '', decay_library: builtins.str =
'', data_version: builtins.str = '', created_utc: typing.Optional[builtins.str] = None,
reactions: typing.Optional[typing.Sequence[builtins.str]] = None, branch_ratios:
typing.Optional[builtins.str] = None, subsections:
typing.Optional[typing.Sequence[builtins.str]] = None, decay_fill_files:
typing.Sequence[builtins.str] = [], decay_fill_library: builtins.str = '') ->
builtins.int:
    r"""
    Convert ENDF decay, fission product yield and neutron evaluations into a
    transmutation Arrow directory.

    Parameters
    ----------
    decay_files, fpy_files, neutron_files : list[str]
        ENDF inputs. All three are needed to build a network: the decay data
        says what decays into what, the neutron data what transmutes into what,
        and the fission yields what fission produces. A file may hold one
        evaluation or a whole tape of them, such as JEFF-4.0's decay and
        fission yield tapes; every material in it is read.
    output_path : str
        Directory to write. Created if absent, merged into if it already holds
        other subsections of the same library.
    library : str
        Library name recorded in the provenance, e.g. ``"endf-b8.1"``.
    data_version : str
        Identifier of the published release. This is what a consumer compares a
        cached copy against, so a rebuild that changes values must change this.
        It is not the converter version: two rebuilds from one converter are
        different data.
    created_utc : str, optional
        Timestamp recorded in the provenance. Supply one for a reproducible
        directory; left unset, the current time is used.
    branch_ratios : str, optional
        Path to a JSON branching table in ``openmc_data`` shape,
        ``{reaction: {parent: {target: fraction}}}``. Splits a reaction between
        a ground state and its metastable partners. Without it every
        ``(n,gamma)`` gives its whole rate to the ground state and the
        metastable product is absent from the network entirely.
    subsections : list[str], optional
        Which of ``decay``, ``reactions`` and ``fission_yields`` to write.
        Defaults to all three. A caller assembling a chain from more than one
        library writes each subsection from the library that has it, and the
        manifest merges rather than being overwritten.
    reactions : list[str], optional
        Reaction names to follow. Defaults to every reaction the chain builder
        knows, not the six-name short set, so nothing is silently left out.
    decay_fill_files : list[str], optional
        Decay evaluations from a second library, read only to replace the
        placeholder average decay energies in ``decay_files``. Some libraries
        write a stand-in for nuclides nobody has evaluated: a third of each
        beta or electron-capture branch's Q to the light particles and a third
        to the photons, which for an electron-capture emitter can be several
        times the recoverable energy (Sn111 in ENDF/B-VIII.1 is 1.63 MeV per
        decay against 0.69 MeV from its decay scheme). A placeholder is
        replaced only where the second library has an evaluated decay scheme
        for the same nuclide with a half-life within 25% of the first's.
        Half-lives and decay modes are never touched. The decay subsection's
        ``provenance.json`` lists every placeholder and every replacement,
        with or without a fill. A tape is read whole, as for ``decay_files``.
    decay_fill_library : str, optional
        The library ``decay_fill_files`` came from, e.g. ``"jendl-5.0"``.
        Required with ``decay_fill_files``.
```

```
        Returns
        -------
        int
            The number of nuclides in the network written.
        """
```

func yani.convert_branching

```
convert_branching(neutron_files: Sequence[str], decay_files: Sequence[str],
output_path: str, library: str = '', decay_library: str = '', data_version: str = '',
created_utc: Optional[str] = None, tol_ev: float = 3000.0, linearize_tol: float =
0.001) -> dict
```

Extract isomeric branching from neutron evaluations and write the `branching/` subsection.

Separate from :func:`convert_transmutation` because the branching curves and the decay data routinely come from different libraries: TENDL branching beside ENDF/B decay is the usual combination. Both calls merge into the same `manifest.json`.

Parameters

neutron_files : list[str] Neutron evaluations of the parent nuclides. MF=8/9/10 radionuclide production is what carries the branching. A file may hold many materials, such as IRDFF-II's tape of 70, and every one is read. decay_files : list[str] Decay evaluations, read only to build the isomer table, so the metastable ones suffice. A tape is read whole, as for the neutron files. output_path : str The transmutation directory to write `branching/` into. library, data_version, created_utc As for :func:`convert_transmutation`. The library here is the branching source, which need not match the chain's. tol_ev : float, optional Energy tolerance when matching a nuclear level to an isomeric state. linearize_tol : float, optional Relative tolerance for resampling a non-lin-lin region onto the lin-lin pairs the format stores.

Returns

dict Coverage: parents read, parents with data, curves linearized, duplicate groups merged, the metastable targets found, `level_routes` (how many excited production levels were matched to an isomer by energy, by energy within a tenth, by level index, as the only isomer, or not at all) and `flagged_levels` (one line per level that was unresolved, taken as ground because the decay data has no isomer for its product, matched only by the looser energy pass, matched by energy while its level index pointed at another isomer, or taken as its product's only isomer while further from it than either energy pass accepts; every excited level that ends up at ground is listed), `partial_sum_mismatches` (one line per reaction whose MF=10 partial cross sections do not sum to its MF=3 total, or whose MF=9 yields do not sum to one, within two percent below 20 MeV), and `skipped_states` (one line per production state that names no single product nuclide, and so gives no row: fission, an IZAP of zero that no single MF=8 subsection resolves, or any other ZAP whose Z or A is not positive), `list_facts` (one line per production list, a parent's MT in MF=9 or MF=10: whether the ground state is listed, whether the file has an MF=3 section for the MT, each state's LFS, LMF, target, route and level energy difference, and

the MF=1 normalisation lines naming the MT) and `list_counts` (how many lists are complete or isomers only in each file, have no MF=3 section for their MT, or are normalised). The same facts are stored per row in `branching.arrow`; they change no rate. The MF=40 production covariance, written as the tape gives it to `branching/branching_covariance.arrow`, is counted by `mf40_sections` (sections read, whatever the MT), `mf40_blocks` (blocks written), `mf40_blocks_by_lb` (the NI blocks by layout), `mf40_nc_blocks`, `mf40_unmatched_states` (one line per product state that matches no MF=9 or MF=10 state, matches several within the tolerance, or is excited with no stated excitation to confirm a level by, written with no target), `mf40_states_without_chain_kind` (states of an MT with no chain reaction, such as MT 18, written with no reaction), `mf40_on_yield_channels` (states matched to a level whose production MF=9 gives as a yield rather than an MF=10 cross section; where that yield is merged with another state's, the row carries the state's own yield with quantity `"yield"`), `mf40_mat1_naming_itself` (sub-subsections whose MAT1 is the evaluation's own MAT, written as given), `mf40_cross_state_blocks` (blocks other than a state's covariance with itself; key them on MT, LFS, MT1 and XLFS1, since both states can resolve to one target), `mf40_blocks_outside_mf10` (blocks whose partner is in another material or has XMF1 other than 10), `mf40_without_blocks` (one line per section, state or sub-subsection holding no block, with its tape values, the only part of MF=40 the file cannot show), `mf40_partner_unresolved` (one line per sub-subsection written with no partner target because the tape does not pin the partner to one state, among them a partner whose level XLFS1 MF=10 numbers differently from MF=40), and `mf40_states_placed_by_excitation` (one line per product state whose target was matched by excitation because MF=9 and MF=10 give its IZAP and LFS no state or one at another excitation; its self blocks name it by MF=40's own LFS). The four line lists are also written to `branching/provenance.json`. An evaluation set without MF=40 writes no covariance file, and removes one an earlier conversion left there.

Source code in `yani/_core/__init__.pyi`

```
def convert_branching(neutron_files: typing.Sequence[builtins.str], decay_files:
typing.Sequence[builtins.str], output_path: builtins.str, library: builtins.str = '',
decay_library: builtins.str = '', data_version: builtins.str = '', created_utc:
typing.Optional[builtins.str] = None, tol_ev: builtins.float = 3000.0, linearize_tol:
builtins.float = 0.001) -> dict:
    r"""
    Extract isomeric branching from neutron evaluations and write the
    ``branching/`` subsection.

    Separate from :func:`convert_transmutation` because the branching curves and
    the decay data routinely come from different libraries: TENDL branching
    beside ENDF/B decay is the usual combination. Both calls merge into the same
    ``manifest.json``.

    Parameters
    ----------
    neutron_files : list[str]
        Neutron evaluations of the parent nuclides. MF=8/9/10 radionuclide
        production is what carries the branching. A file may hold many
        materials, such as IRDFF-II's tape of 70, and every one is read.
    decay_files : list[str]
        Decay evaluations, read only to build the isomer table, so the
        metastable ones suffice. A tape is read whole, as for the neutron files.
    output_path : str
        The transmutation directory to write ``branching/`` into.
    library, data_version, created_utc
        As for :func:`convert_transmutation`. The library here is the branching
        source, which need not match the chain's.
    tol_ev : float, optional
        Energy tolerance when matching a nuclear level to an isomeric state.
    linearize_tol : float, optional
        Relative tolerance for resampling a non-lin-lin region onto the lin-lin
        pairs the format stores.

    Returns
    -------
    dict
        Coverage: parents read, parents with data, curves linearized, duplicate
        groups merged, the metastable targets found, ``level_routes`` (how many
        excited production levels were matched to an isomer by energy, by
        energy within a tenth, by level index, as the only isomer, or not at
        all) and ``flagged_levels`` (one line per level that was unresolved,
        taken as ground because the decay data has no isomer for its product,
        matched only by the looser energy pass, matched by energy while its
        level index pointed at another isomer, or taken as its product's only
        isomer while further from it than either energy pass accepts; every
        excited level that ends up at ground is listed), ``partial_sum_mismatches``
        (one line per reaction whose MF=10 partial cross sections do not sum to
        its MF=3 total, or whose MF=9 yields do not sum to one, within two
        percent below 20 MeV), and ``skipped_states`` (one line per production
        state that names no single product nuclide, and so gives no row:
        fission, an IZAP of zero that no single MF=8 subsection resolves, or
        any other ZAP whose Z or A is not positive), ``list_facts`` (one line
        per production list, a parent's MT in MF=9 or MF=10: whether the ground
        state is listed, whether the file has an MF=3 section for the MT, each
        state's LFS, LMF, target, route and level energy difference, and the
        MF=1 normalisation lines naming the MT) and ``list_counts`` (how many
        lists are complete or isomers only in each file, have no MF=3 section
        for their MT, or are normalised). The same facts are stored per row in
        ``branching.arrow``; they change no rate.
        The MF=40 production
        covariance, written as the tape gives it to
        ``branching/branching_covariance.arrow``, is counted by
        ``mf40_sections`` (sections read, whatever the MT), ``mf40_blocks``
```

```
        (blocks written), ``mf40_blocks_by_lb`` (the NI blocks by layout),
        ``mf40_nc_blocks``, ``mf40_unmatched_states`` (one line per product
        state that matches no MF=9 or MF=10 state, matches several within the
        tolerance, or is excited with no stated excitation to confirm a level
        by, written with no target), ``mf40_states_without_chain_kind``
        (states of an MT with no chain reaction, such as MT 18, written with
        no reaction),
        ``mf40_on_yield_channels`` (states matched to a level whose production
        MF=9 gives as a yield rather than an MF=10 cross section; where that
        yield is merged with another state's, the row carries the state's own
        yield with quantity ``"yield"``),
        ``mf40_mat1_naming_itself`` (sub-subsections whose MAT1 is the
        evaluation's own MAT, written as given), ``mf40_cross_state_blocks``
        (blocks other than a state's covariance with itself; key them on MT,
        LFS, MT1 and XLFS1, since both states can resolve to one target),
        ``mf40_blocks_outside_mf10`` (blocks whose partner is in another
        material or has XMF1 other than 10), ``mf40_without_blocks`` (one line
        per section, state or sub-subsection holding no block, with its tape
        values, the only part of MF=40 the file cannot show),
        ``mf40_partner_unresolved`` (one line per sub-subsection written
        with no partner target because the tape does not pin the partner to
        one state, among them a partner whose level XLFS1 MF=10 numbers
        differently from MF=40), and ``mf40_states_placed_by_excitation`` (one
        line per product state whose target was matched by excitation because
        MF=9 and MF=10 give its IZAP and LFS no state or one at another
        excitation; its self blocks name it by MF=40's own LFS). The four line
        lists are also written to ``branching/provenance.json``. An evaluation
        set without MF=40 writes no covariance file, and removes one an
        earlier conversion left there.
    """
```

func yani.radionuclide_production

```
radionuclide_production(neutron_files: Sequence[str]) -> list
```

List the final states each reaction of an evaluation can leave its product in.

A reaction that can leave its product in a metastable state says so in MF=8, one subsection per final state. An evaluation that lists only the ground state is not merely less accurate: the isomer is absent from any network built from it, so no code can make it, and a measurement that sees its decay heat cannot be reproduced by any means. TENDL-2017 omits the 1706 keV state from `Os190(n,n')`, which is why the FNS osmium foil comes out at a third of the measured heat with that library, in yani and in FISPACT-II alike. Nothing about the reaction looks wrong from outside: it is present, its cross section is reasonable, and only the state list is short. So comparing the state lists of several libraries is how such a gap is found, and this is the read that makes the comparison possible.

A read, not a conversion. Nothing is written, no decay data is involved, and the answer is what the files say rather than what a network built from them would hold.

Parameters

neutron_files : list[str] Neutron evaluations, one or many materials per file. Read one file at a time rather than held, so a whole sublibrary is a valid argument.

Returns

list[dict] One entry per (parent, reaction) carrying MF=9 or MF=10, in file order and then tape order, each with `parent`, `mt`, `reaction` (the transmutation reaction name, or `None` for an MT no chain reaction covers) and `states`. Each state has `excitation_energy_eV`, `level_index`, `product` and `source`. `excitation_energy_eV` is `None` for an excited state whose evaluation gives neither a positive MF=8 ELFS nor a positive `QM - QI`.

```
``product`` is the product's **ground-state** name even for an excited
state, because naming the isomer needs decay data to say which
isomeric ordinal a level is; pair it with ``excitation_energy_eV``.
It is ``None`` for a state naming no single nuclide: fission, a
subsection whose IZAP is zero with no single MF=8 subsection to name
it, or any other ZAP whose Z or A is not positive.
``level_index`` is the evaluation's own LFS and is not comparable
between libraries: Ir190's 377 keV isomer is level 3 in ENDF/B-VIII.1
and level 37 in JEFF-4.0. ``source`` is ``"cross_section"`` for MF=10
or ``"yield"`` for MF=9.
```

Examples

```
>>> [c for c in yani.radionuclide_production(["n-Os190.tendl"])
...  if c["mt"] == 4][0]["states"]
[{'excitation_energy_eV': 0.0, 'level_index': 0, 'product': 'Os190',
  'source': 'cross_section'}]
```

Source code in `yani/_core/__init__.pyi`

```
def radionuclide_production(neutron_files: typing.Sequence[builtins.str]) -> list:
    r"""
    List the final states each reaction of an evaluation can leave its product
    in.

    A reaction that can leave its product in a metastable state says so in
    MF=8, one subsection per final state. An evaluation that lists only the
    ground state is not merely less accurate: the isomer is absent from any
    network built from it, so no code can make it, and a measurement that sees
    its decay heat cannot be reproduced by any means. TENDL-2017 omits the
    1706 keV state from ``Os190(n,n')``, which is why the FNS osmium foil comes
    out at a third of the measured heat with that library, in yani and in
    FISPACT-II alike. Nothing about the reaction looks wrong from outside: it
    is present, its cross section is reasonable, and only the state list is
    short. So comparing the state lists of several libraries is how such a gap
    is found, and this is the read that makes the comparison possible.

    A read, not a conversion. Nothing is written, no decay data is involved,
    and the answer is what the files say rather than what a network built from
    them would hold.

    Parameters
    ----------
    neutron_files : list[str]
        Neutron evaluations, one or many materials per file. Read one file at
        a time rather than held, so a whole sublibrary is a valid argument.

    Returns
    -------
    list[dict]
        One entry per (parent, reaction) carrying MF=9 or MF=10, in file order
        and then tape order, each with ``parent``, ``mt``, ``reaction`` (the
        transmutation reaction name, or ``None`` for an MT no chain reaction
        covers) and ``states``.
        Each state has ``excitation_energy_eV``, ``level_index``, ``product``
        and ``source``. ``excitation_energy_eV`` is ``None`` for an excited
        state whose evaluation gives neither a positive MF=8 ELFS nor a
        positive ``QM - QI``.

        ``product`` is the product's **ground-state** name even for an excited
        state, because naming the isomer needs decay data to say which
        isomeric ordinal a level is; pair it with ``excitation_energy_eV``.
        It is ``None`` for a state naming no single nuclide: fission, a
        subsection whose IZAP is zero with no single MF=8 subsection to name
        it, or any other ZAP whose Z or A is not positive.
        ``level_index`` is the evaluation's own LFS and is not comparable
        between libraries: Ir190's 377 keV isomer is level 3 in ENDF/B-VIII.1
        and level 37 in JEFF-4.0. ``source`` is ``"cross_section"`` for MF=10
        or ``"yield"`` for MF=9.

    Examples
    --------
        >>> [c for c in yani.radionuclide_production(["n-Os190.tendl"])
        ...  if c["mt"] == 4][0]["states"]
        [{'excitation_energy_eV': 0.0, 'level_index': 0, 'product': 'Os190',
          'source': 'cross_section'}]
    """
```

The two below write sections only transport reads, so nothing here consumes their output. They are on this wheel so that a data-generation script works against either wheel, which the bindings repository pins with a parity test. `convert_photon` carries its auxiliary tabulations, the same ones yamc's does.

func yani.convert_neutron_transport

```
convert_neutron_transport(input_path: str, output_dir: str, njoy_exec: str = 'njoy',
temperatures: Optional[Sequence[float]] = None, library: str = '', data_version: str =
'', created_utc: Optional[str] = None, covariance: bool = False) -> str
```

Convert an ENDF evaluation into the full neutron transport data set.

Everything :func: `convert_neutron_xs` writes, plus the reaction products, their secondary angle and energy distributions, the lookup accelerator the transport hot path reads, and the two fissile-only sections.

ENDF only, deliberately. An ACE table holds one temperature, carries no MT 901 heating and no fission energy release, so a transport conversion from one would be missing the local KERMA and the fission photon scaling with nothing on disk to say so.

Parameters

input_path : str Path to the ENDF evaluation. output_dir : str Directory to write `{Nuclide}.arrow/` into. NJOY also works in a hidden directory here, so this disk needs room for its tapes; the directory is removed when the conversion ends. njoy_exec : str The NJOY executable. FENDL needs the IAEA-NDS fork; upstream NJOY does not reproduce the official release, and both builds succeed, so the wrong choice is quiet. temperatures : list[float], optional Temperatures in Kelvin. Defaults to NJOY's own default. library, data_version, created_utc Recorded in `version.json`. covariance : bool Also write `covariance.arrow`, the MF=33 cross-section covariance, and the evaluation's other covariance sections where it has them: `angular_covariance.arrow` (MF=34), `nubar_covariance.arrow` (MF=31), `spectrum_covariance.arrow` (MF=35) and `resonance_parameters.arrow` (the MF=2 resonance parameters with their MF=32 covariance). Off by default: the matrices are large and only an uncertainty calculation reads them.

Returns

str Path to the directory written.

Source code in `yani/_core/__init__.pyi`

```
def convert_neutron_transport(input_path: builtins.str, output_dir: builtins.str,
njoy_exec: builtins.str = 'njoy', temperatures:
typing.Optional[typing.Sequence[builtins.float]] = None, library: builtins.str = '',
data_version: builtins.str = '', created_utc: typing.Optional[builtins.str] = None,
covariance: builtins.bool = False) -> builtins.str:
    r"""
    Convert an ENDF evaluation into the full neutron transport data set.

    Everything :func:`convert_neutron_xs` writes, plus the reaction products,
    their secondary angle and energy distributions, the lookup accelerator the
    transport hot path reads, and the two fissile-only sections.

    ENDF only, deliberately. An ACE table holds one temperature, carries no
    MT 901 heating and no fission energy release, so a transport conversion
    from one would be missing the local KERMA and the fission photon scaling
    with nothing on disk to say so.

    Parameters
    ----------
    input_path : str
        Path to the ENDF evaluation.
    output_dir : str
        Directory to write ``{Nuclide}.arrow/`` into. NJOY also works in a
        hidden directory here, so this disk needs room for its tapes; the
        directory is removed when the conversion ends.
    njoy_exec : str
        The NJOY executable. FENDL needs the IAEA-NDS fork; upstream NJOY does
        not reproduce the official release, and both builds succeed, so the
        wrong choice is quiet.
    temperatures : list[float], optional
        Temperatures in Kelvin. Defaults to NJOY's own default.
    library, data_version, created_utc
        Recorded in ``version.json``.
    covariance : bool
        Also write ``covariance.arrow``, the MF=33 cross-section covariance,
        and the evaluation's other covariance sections where it has them:
        ``angular_covariance.arrow`` (MF=34), ``nubar_covariance.arrow``
        (MF=31), ``spectrum_covariance.arrow`` (MF=35) and
        ``resonance_parameters.arrow`` (the MF=2 resonance parameters with
        their MF=32 covariance). Off by default: the matrices are large and
        only an uncertainty
        calculation reads them.

    Returns
    -------
    str
        Path to the directory written.
    """
```

func yani.convert_photon

```
convert_photon(photoatomic_path: str | PathLike | Path, output_dir: str | PathLike | Path, relaxation_path: Optional[str | PathLike | Path] = None, compton_profiles: Optional[str | PathLike | Path] = None, density_effect: Optional[str | PathLike | Path] = None, bremsstrahlung: Optional[str | PathLike | Path] = None, library: str = '', data_version: str = '', created_utc: Optional[str] = None) -> list[str]
```

Convert a photoatomic evaluation into the per-element photon sections.

Writes `element.arrow` always, and `subshells.arrow`, `compton.arrow` and `bremsstrahlung.arrow` where there is data for them. No NJOY: a photoatomic evaluation is already pointwise.

Parameters

photoatomic_path : str The photoatomic evaluation. One file may hold several elements, and each is written to its own directory. output_dir : str Directory to write `{Element}.arrow/` into. relaxation_path : str, optional The atomic relaxation evaluation for the same element, carrying binding energies, occupancies and the transition cascade. Without it the subshells are written with zero binding energy and no transitions, which is a fluorescence-free atom rather than an error. compton_profiles, density_effect, bremsstrahlung : str or path, optional Other copies of the auxiliary tabulations, which no evaluation carries (Compton profiles, the density effect and bremsstrahlung). The default is the copies compiled into this package, the same in yani and yani. Give all three or none. library, data_version, created_utc Recorded in `version.json`.

Returns

list[str] One path per element written.

Source code in `yani/_core/__init__.pyi`

```
def convert_photon(photoatomic_path: builtins.str | os.PathLike | pathlib.Path,
output_dir: builtins.str | os.PathLike | pathlib.Path, relaxation_path:
typing.Optional[builtins.str | os.PathLike | pathlib.Path] = None, compton_profiles:
typing.Optional[builtins.str | os.PathLike | pathlib.Path] = None, density_effect:
typing.Optional[builtins.str | os.PathLike | pathlib.Path] = None, bremsstrahlung:
typing.Optional[builtins.str | os.PathLike | pathlib.Path] = None, library:
builtins.str = '', data_version: builtins.str = '', created_utc:
typing.Optional[builtins.str] = None) -> builtins.list[builtins.str]:
    r"""
    Convert a photoatomic evaluation into the per-element photon sections.

    Writes ``element.arrow`` always, and ``subshells.arrow``, ``compton.arrow``
    and ``bremsstrahlung.arrow`` where there is data for them. No NJOY: a
    photoatomic evaluation is already pointwise.

    Parameters
    ----------
    photoatomic_path : str
        The photoatomic evaluation. One file may hold several elements, and
        each is written to its own directory.
    output_dir : str
        Directory to write ``{Element}.arrow/`` into.
    relaxation_path : str, optional
        The atomic relaxation evaluation for the same element, carrying binding
        energies, occupancies and the transition cascade. Without it the
        subshells are written with zero binding energy and no transitions,
        which is a fluorescence-free atom rather than an error.
    compton_profiles, density_effect, bremsstrahlung : str or path, optional
        Other copies of the auxiliary tabulations, which no evaluation carries
        (Compton profiles, the density effect and bremsstrahlung). The default
        is the copies compiled into this package, the same in yani and yani.
        Give all three or none.
    library, data_version, created_utc
        Recorded in ``version.json``.

    Returns
    -------
    list[str]
        One path per element written.
    """
```

The photonuclear converter preserves evaluated MF=3 cross sections and the original ENDF sections. It does not enable photon-induced transmutation or photon transport; see TENDL photonuclear data.

func yani.convert_photonuclear

```
convert_photonuclear(input_path: str, output_dir: str, library: str = '',
data_version: str = '', created_utc: Optional[str] = None) -> str
```

Preserve a photonuclear ENDF evaluation as lossless Arrow tables. Writes native-grid MF=3 cross sections with their ENDF interpolation and every original section, including outgoing-particle data. Requires NSUB=0. No neutron Doppler broadening or photoatomic relaxation is applied. This processes

evaluated data; it does not enable photonuclear transport. `library`, `data_version` and `created_utc` are recorded in version.json.

Source code in `yani/_core/__init__.pyi`

```
5245  def convert_photonuclear(input_path: builtins.str, output_dir: builtins.str, library:
5246  builtins.str = '', data_version: builtins.str = '', created_utc:
5247  typing.Optional[builtins.str] = None) -> builtins.str:
5248      r"""
5249      Preserve a photonuclear ENDF evaluation as lossless Arrow tables.
5250      Writes native-grid MF=3 cross sections with their ENDF interpolation and
5251      every original section, including outgoing-particle data. Requires NSUB=0.
5252      No neutron Doppler broadening or photoatomic relaxation is applied. This
5253      processes evaluated data; it does not enable photonuclear transport.
          ``library``, ``data_version`` and ``created_utc`` are recorded in version.json.
          """
```

Reading data back

Both of these read a converted Arrow directory through the same loader transport uses and return a description of what is in it, which is how a conversion is checked without standing up a transport run. Neither goes through the process-wide stores, and the photon one publishes none of the process-wide photon grids, so inspecting two builds of the same nuclide or element in one interpreter reports on each of them rather than handing back the first.

func yani.read_nuclide_from_arrow

```
read_nuclide_from_arrow(path: str, scope: str = 'full') -> Any
```

Read a nuclide Arrow directory through yani's own loader and describe it.

Parameters

path : str A `{Nuclide}.arrow/` section directory. Taken verbatim: no keyword expansion, no directory search and no download, so a validation run cannot silently check a different file from the one it named. scope : str, optional `"full"` (default) reads every section, which is what transport needs. `"xs"` reads `nuclide.arrow`, `energy.arrow` and `reactions.arrow` only, which is what a transport-free reaction-rate collapse touches.

Returns

dict `name`, `atomic_number`, `mass_number`, `atomic_weight_ratio`, `fissionable`, `urr_present`, `available_temperatures`, `loaded_temperatures`, `mts`, `energy_points`, `angular_covariance`, `nubar_covariance`, `spectrum_covariance`, `resonance_parameters` and `scope_loaded`.

```
``energy_points`` is a dict of temperature to grid length, over the
loaded temperatures. Not one number: the reader also keeps the 0 K union
grid, which is longer than any of them and belongs to no temperature the
other two lists mention, so a single maximum would report a grid nothing
reads and hide a per-temperature grid that had been truncated.

There is no ``library`` key. The Arrow loader does not populate that
field, so it would report ``None`` for every directory, correct or not.
Read ``version.json`` for it.

``angular_covariance`` is the folder's MF=34 (``angular_covariance.arrow``),
summarised as ``{mt: [(l, l1), ...]}``: per reaction, the sorted (L, L1)
pairs of Legendre orders a covariance block correlates. ``None`` when
the folder has none. It is read whatever ``scope`` says, since it is a
separate optional section.

``nubar_covariance`` is the folder's MF=31 (``nubar_covariance.arrow``),
summarised as the sorted fission multiplicity MTs (of 452, 455 and 456)
it carries covariance for, and ``spectrum_covariance`` its MF=35
(``spectrum_covariance.arrow``), as ``{mt: [(e1, e2), ...]}``: per
reaction, the incident energy range of each covariance block, in tape
order. Each is ``None`` when the folder has none, and both are read
whatever ``scope`` says, as ``angular_covariance`` is.

``resonance_parameters`` is the folder's MF=2 and MF=32
(``resonance_parameters.arrow``), both parsed, summarised as the
``(lru, lrf)`` of each MF=2 energy range, isotope by isotope in tape
order: LRU 1 resolved or 2 unresolved, LRF the formalism. ``None`` when
the folder has none, and read whatever ``scope`` says.

``scope_loaded`` is the one to assert on, and it is not always the
``scope`` asked for: a directory holding no transport sections narrows a
```"full"``` request to ```"xs"``` rather than failing it, on the theory
that it is a cross-sections-only conversion. A gate that asks for
```"full"``` and does not check what it got will pass a directory with no
distributions and no products.
```

Raises

ValueError `scope` is neither `"full"` nor `"xs"`. RuntimeError The directory is absent, is not the Arrow format version this build reads, or any section fails to parse, the stored MF=2 and MF=32 text included.

Source code in `yani/_core/__init__.pyi`

```
def read_nuclide_from_arrow(path: builtins.str, scope: builtins.str = 'full') ->
typing.Any:
    r"""
    Read a nuclide Arrow directory through yani's own loader and describe it.

    Parameters
    ----------
    path : str
        A ``{Nuclide}.arrow/`` section directory. Taken verbatim: no keyword
        expansion, no directory search and no download, so a validation run
        cannot silently check a different file from the one it named.
    scope : str, optional
        ``"full"`` (default) reads every section, which is what transport
        needs. ``"xs"`` reads ``nuclide.arrow``, ``energy.arrow`` and
        ``reactions.arrow`` only, which is what a transport-free reaction-rate
        collapse touches.

    Returns
    -------
    dict
        ``name``, ``atomic_number``, ``mass_number``, ``atomic_weight_ratio``,
        ``fissionable``, ``urr_present``, ``available_temperatures``,
        ``loaded_temperatures``, ``mts``, ``energy_points``,
        ``angular_covariance``, ``nubar_covariance``, ``spectrum_covariance``,
        ``resonance_parameters`` and ``scope_loaded``.

        ``energy_points`` is a dict of temperature to grid length, over the
        loaded temperatures. Not one number: the reader also keeps the 0 K union
        grid, which is longer than any of them and belongs to no temperature the
        other two lists mention, so a single maximum would report a grid nothing
        reads and hide a per-temperature grid that had been truncated.

        There is no ``library`` key. The Arrow loader does not populate that
        field, so it would report ``None`` for every directory, correct or not.
        Read ``version.json`` for it.

        ``angular_covariance`` is the folder's MF=34 (``angular_covariance.arrow``),
        summarised as ``{mt: [(l, l1), ...]}``: per reaction, the sorted (L, L1)
        pairs of Legendre orders a covariance block correlates. ``None`` when
        the folder has none. It is read whatever ``scope`` says, since it is a
        separate optional section.

        ``nubar_covariance`` is the folder's MF=31 (``nubar_covariance.arrow``),
        summarised as the sorted fission multiplicity MTs (of 452, 455 and 456)
        it carries covariance for, and ``spectrum_covariance`` its MF=35
        (``spectrum_covariance.arrow``), as ``{mt: [(e1, e2), ...]}``: per
        reaction, the incident energy range of each covariance block, in tape
        order. Each is ``None`` when the folder has none, and both are read
        whatever ``scope`` says, as ``angular_covariance`` is.

        ``resonance_parameters`` is the folder's MF=2 and MF=32
        (``resonance_parameters.arrow``), both parsed, summarised as the
        ``(lru, lrf)`` of each MF=2 energy range, isotope by isotope in tape
        order: LRU 1 resolved or 2 unresolved, LRF the formalism. ``None`` when
        the folder has none, and read whatever ``scope`` says.

        ``scope_loaded`` is the one to assert on, and it is not always the
        ``scope`` asked for: a directory holding no transport sections narrows a
        ``"full"`` request to ``"xs"`` rather than failing it, on the theory
        that it is a cross-sections-only conversion. A gate that asks for
        ``"full"`` and does not check what it got will pass a directory with no
        distributions and no products.

    Raises
```

```
    ------
    ValueError
        ``scope`` is neither ``"full"`` nor ``"xs"``.
    RuntimeError
        The directory is absent, is not the Arrow format version this build
        reads, or any section fails to parse, the stored MF=2 and MF=32 text
        included.
    """
```

func yani.read_element_from_arrow

```
read_element_from_arrow(path: str) -> Any
```

Read a photoatomic Arrow directory through yani's own loader and describe it.

Parameters

path : str An `{Element}.arrow/` directory. Taken verbatim, and read straight rather than through the process-global element store: that store is keyed by element name, so validating two builds of `Fe.arrow` in one process would get the first one back for the second call and report a false pass.

```
Publishes none of the process-wide photon grids either. An ordinary
read installs the Compton momentum grid and the two TTB grids on first
use, so inspecting a candidate directory in an interpreter that also
runs transport would hand that candidate's grids to every later element
load and every photon collision. The sampler reads the profile tables
against whichever grid is installed, so a different grid changes every
Doppler sample and says nothing about it.
```

Returns

dict `name`, `atomic_number`, `n_energy_points`, `n_subshells`, `has_atomic_relaxation`, `has_compton_profiles` and `has_bremsstrahlung`.

```
The last three are the auxiliary tabulations that are not in any
evaluation, so a directory converted without them reads back fine and
is still not what transport wants. There is no ``scope`` here: the
photon reader has one mode.
```

Raises

RuntimeError The directory is absent, or any section fails to parse.

Source code in `yani/_core/__init__.pyi`

```
5639  def read_element_from_arrow(path: builtins.str) -> typing.Any:
5640      r"""
5641      Read a photoatomic Arrow directory through yani's own loader and describe it.
5642
5643      Parameters
5644      ----------
5645      path : str
5646          An ``{Element}.arrow/`` directory. Taken verbatim, and read straight
5647          rather than through the process-global element store: that store is
5648          keyed by element name, so validating two builds of ``Fe.arrow`` in one
5649          process would get the first one back for the second call and report a
5650          false pass.
5651
5652          Publishes none of the process-wide photon grids either. An ordinary
5653          read installs the Compton momentum grid and the two TTB grids on first
5654          use, so inspecting a candidate directory in an interpreter that also
5655          runs transport would hand that candidate's grids to every later element
5656          load and every photon collision. The sampler reads the profile tables
5657          against whichever grid is installed, so a different grid changes every
5658          Doppler sample and says nothing about it.
5659
5660      Returns
5661      -------
5662      dict
5663          ``name``, ``atomic_number``, ``n_energy_points``, ``n_subshells``,
5664          ``has_atomic_relaxation``, ``has_compton_profiles`` and
5665          ``has_bremsstrahlung``.
5666
5667          The last three are the auxiliary tabulations that are not in any
5668          evaluation, so a directory converted without them reads back fine and
5669          is still not what transport wants. There is no ``scope`` here: the
5670          photon reader has one mode.
5671
5672      Raises
5673      ------
5674      RuntimeError
5675          The directory is absent, or any section fails to parse.
5676      """
```

The photonuclear readback validates its cross-section tables against the preserved MF=3 source rather than invoking either transport loader.

func yani.read_photonuclear_from_arrow

```
read_photonuclear_from_arrow(path: str) -> Any
```

Validate evaluated photonuclear Arrow tables against their preserved ENDF. Returns the nuclide identity, MTs, per-MT energy-point counts and preserved (MF, MT) section keys. Does not invoke neutron or photoatomic transport.

Source code in `yani/_core/__init__.pyi`

```
5750  def read_photonuclear_from_arrow(path: builtins.str) -> typing.Any:
5751      r"""
5752      Validate evaluated photonuclear Arrow tables against their preserved ENDF.
5753      Returns the nuclide identity, MTs, per-MT energy-point counts and preserved
5754      (MF, MT) section keys. Does not invoke neutron or photoatomic transport.
5755      """
```

Data helpers

Cross-section inspection, dose and attenuation coefficients, available element and reaction names, and the nuclide cache are accessed through `yani.data`.

mod yani.data

func dose_coefficients

```
dose_coefficients(particle: str, geometry: Optional[str] = None, data_source:
Optional[str] = None, dose_quantity: str = 'effective') -> DoseCoefficients
```

Return fluence-to-dose conversion coefficients.

`dose_quantity='effective'` (the default) returns fluence-to-effective-dose coefficients from ICRP Publication 74 or 116 for an irradiation geometry.

`dose_quantity='ambient'` returns fluence-to-ambient-dose-equivalent H*(10) coefficients from ICRP Publication 74 (Table A.21 for photons, Table A.42 for neutrons), the ICRU-57 definition most regulations still reference.* H(10) is defined at 10 mm depth in the ICRU sphere in an aligned and expanded field, so it takes no `geometry`, and ICRP-116 does not tabulate it. It is designed to over-estimate effective dose for area monitoring, but for high-energy neutrons it under-reads it. The tables cover 1 meV to 20 MeV for neutrons and 10 keV to 10 MeV for photons, and are not extrapolated: a tally folded with them scores nothing for particles outside that range.

Parameters:

Name	Type	Description	Default
`particle`	`str`	'neutron' or 'photon'.	*required*
`geometry`	`str`	Irradiation geometry for effective dose. One of: 'AP' (Anterior-Posterior), 'PA' (Posterior-Anterior), 'LLAT' (Left Lateral), 'RLAT' (Right Lateral), 'ROT' (Rotational), 'ISO' (Isotropic). Defaults to 'AP'. Must not be given for `dose_quantity='ambient'`.	`None`
`data_source`	`str`	'icrp74' or 'icrp116'. Defaults to 'icrp116' for effective dose and 'icrp74' for ambient dose equivalent, which only ICRP-74 tabulates.	`None`
`dose_quantity`	`str`	'effective' (default) for effective dose or 'ambient' for ambient dose equivalent H*(10).	`'effective'`

Returns:

Name	Type	Description
`DoseCoefficients`	`DoseCoefficients`	an object with `.energy` (eV), `.coefficients`
	`DoseCoefficients`	(pSv cm2), `.units`, and `.as_energy_function()`.

Examples:

import yani

Get ICRP-116 AP geometry dose coefficients

dc = yani.data.dose_coefficients('neutron', 'AP') dc.energy # incident energies in eV dc.coefficients # fluence-to-dose coefficients dc.units # 'pSv cm2'

ICRP-74 ambient dose equivalent H*(10)

h10 = yani.data.dose_coefficients('neutron', dose_quantity='ambient')

Fold them into a flux tally via the energy_function= argument

dose_tally = yamc.Tally(scores=['flux'], energy_function=dc.as_energy_function())

Or let the tally fetch the same coefficients itself:

dose_tally = yamc.Tally(scores=['flux'], dose_coefficients=('neutron', 'AP'))

References

- ICRP Publication 74: https://doi.org/10.1016/S0146-6453(96)90010-X
- ICRP Publication 116: https://doi.org/10.1016/j.icrp.2011.10.001

Source code in `yani/_core/data.pyi`

```
def dose_coefficients(particle: builtins.str, geometry: typing.Optional[builtins.str]
= None, data_source: typing.Optional[builtins.str] = None, dose_quantity:
builtins.str = 'effective') -> DoseCoefficients:
    r"""
    Return fluence-to-dose conversion coefficients.

    ``dose_quantity='effective'`` (the default) returns fluence-to-effective-dose
    coefficients from ICRP Publication 74 or 116 for an irradiation geometry.

    ``dose_quantity='ambient'`` returns fluence-to-ambient-dose-equivalent
    H*(10) coefficients from ICRP Publication 74 (Table A.21 for photons, Table
    A.42 for neutrons), the ICRU-57 definition most regulations still reference.
    H*(10) is defined at 10 mm depth in the ICRU sphere in an aligned and
    expanded field, so it takes no ``geometry``, and ICRP-116 does not tabulate
    it. It is designed to over-estimate effective dose for area monitoring, but
    for high-energy neutrons it under-reads it. The tables cover 1 meV to 20 MeV
    for neutrons and 10 keV to 10 MeV for photons, and are not extrapolated: a
    tally folded with them scores nothing for particles outside that range.

    Args:
        particle (str): 'neutron' or 'photon'.
        geometry (str, optional): Irradiation geometry for effective dose. One of:
            'AP' (Anterior-Posterior), 'PA' (Posterior-Anterior),
            'LLAT' (Left Lateral), 'RLAT' (Right Lateral),
            'ROT' (Rotational), 'ISO' (Isotropic). Defaults to 'AP'. Must not be
            given for ``dose_quantity='ambient'``.
        data_source (str, optional): 'icrp74' or 'icrp116'. Defaults to
            'icrp116' for effective dose and 'icrp74' for ambient dose
            equivalent, which only ICRP-74 tabulates.
        dose_quantity (str): 'effective' (default) for effective dose or
            'ambient' for ambient dose equivalent H*(10).

    Returns:
        DoseCoefficients: an object with ``.energy`` (eV), ``.coefficients``
        (pSv cm2), ``.units``, and ``.as_energy_function()``.

    Examples:
        import yani

        # Get ICRP-116 AP geometry dose coefficients
        dc = yani.data.dose_coefficients('neutron', 'AP')
        dc.energy          # incident energies in eV
        dc.coefficients    # fluence-to-dose coefficients
        dc.units           # 'pSv cm2'

        # ICRP-74 ambient dose equivalent H*(10)
        h10 = yani.data.dose_coefficients('neutron', dose_quantity='ambient')

        # Fold them into a flux tally via the energy_function= argument
        dose_tally = yamc.Tally(scores=['flux'],
energy_function=dc.as_energy_function())

        # Or let the tally fetch the same coefficients itself:
        dose_tally = yamc.Tally(scores=['flux'], dose_coefficients=('neutron', 'AP'))

    References:
        - ICRP Publication 74: https://doi.org/10.1016/S0146-6453(96)90010-X
        - ICRP Publication 116: https://doi.org/10.1016/j.icrp.2011.10.001
    """
```

`func` mass_attenuation_coefficient

```
mass_attenuation_coefficient(element: Any) -> PhotonCoefficients
```

Return the photon mass attenuation coefficient mu/rho of an element.

Total attenuation with coherent scattering, from the NIST XCOM database, tabulated from 1 keV to 20 MeV for Z = 1 to 100. Dividing a material's dose or fluence by `sum(density_i * mu_over_rho_i(E))` is what a self-shielding estimate like :meth:`Material.contact_dose` does with it.

Parameters:

Name	Type	Description	Default
`element`	`str | int`	Element symbol (`'Fe'`) or atomic number (`26`).	*required*

Returns:

Name	Type	Description
`PhotonCoefficients`	`PhotonCoefficients`	mu/rho in cm²/g against energy in eV.

Examples:

import yani

iron = yani.data.mass_attenuation_coefficient('Fe') iron.interpolate(1.0e6) # 0.05995 cm2/g at 1 MeV iron.energy, iron.coefficients

References

- NIST Standard Reference Database 8 (XCOM): https://doi.org/10.18434/T48G6X

Source code in `yani/_core/data.pyi`

```
def mass_attenuation_coefficient(element: typing.Any) -> PhotonCoefficients:
    r"""
    Return the photon mass attenuation coefficient mu/rho of an element.

    Total attenuation with coherent scattering, from the NIST XCOM database,
    tabulated from 1 keV to 20 MeV for Z = 1 to 100. Dividing a material's dose
    or fluence by ``sum(density_i * mu_over_rho_i(E))`` is what a self-shielding
    estimate like :meth:`Material.contact_dose` does with it.

    Args:
        element (str | int): Element symbol (``'Fe'``) or atomic number (``26``).

    Returns:
        PhotonCoefficients: mu/rho in cm²/g against energy in eV.

    Examples:
        import yani

        iron = yani.data.mass_attenuation_coefficient('Fe')
        iron.interpolate(1.0e6)     # 0.05995 cm2/g at 1 MeV
        iron.energy, iron.coefficients

    References:
        - NIST Standard Reference Database 8 (XCOM):
<https://doi.org/10.18434/T48G6X>
    """
```

func mass_energy_absorption_coefficient

```
mass_energy_absorption_coefficient(material: str = 'air') -> PhotonCoefficients
```

Return the mass energy-absorption coefficient mu_en/rho of a material.

The fraction of incident photon energy actually absorbed per unit mass, less what scattered photons carry away, from NIST SRD 126. This is the response an absorbed dose in air folds against, and it is a different quantity from the attenuation coefficient: attenuation counts photons removed from the beam, this counts energy deposited.

Parameters:

Name	Type	Description	Default
`material`	`str`	Currently only `'air'` (dry, near sea level), tabulated from 1 keV to 20 MeV.	`'air'`

Returns:

Name	Type	Description
`PhotonCoefficients`	`PhotonCoefficients`	mu_en/rho in cm²/g against energy in eV.

Examples:

import yani

air = yani.data.mass_energy_absorption_coefficient('air') air.interpolate(1.0e6) # 0.02789 cm2/g at 1 MeV

References

- NIST Standard Reference Database 126: https://doi.org/10.18434/T4D01F

Source code in `yani/_core/data.pyi`

```
111 def mass_energy_absorption_coefficient(material: builtins.str = 'air') ->
112 PhotonCoefficients:
113     r"""
114     Return the mass energy-absorption coefficient mu_en/rho of a material.
115 
116     The fraction of incident photon energy actually absorbed per unit mass, less
117     what scattered photons carry away, from NIST SRD 126. This is the response an
118     absorbed dose in air folds against, and it is a different quantity from the
119     attenuation coefficient: attenuation counts photons removed from the beam,
120     this counts energy deposited.
121 
122     Args:
123         material (str): Currently only ``'air'`` (dry, near sea level), tabulated
124             from 1 keV to 20 MeV.
125 
126     Returns:
127         PhotonCoefficients: mu_en/rho in cm²/g against energy in eV.
128 
129     Examples:
130         import yani
131 
132         air = yani.data.mass_energy_absorption_coefficient('air')
133         air.interpolate(1.0e6)      # 0.02789 cm2/g at 1 MeV
134 
135     References:
136         - NIST Standard Reference Database 126: <https://doi.org/10.18434/T4D01F>
    """
```

`func` natural_abundance

```
natural_abundance() -> Any
```

Return a dict of natural isotopic abundances by nuclide name.

Returns:

Type	Description
`Any`	dict[str, float]: e.g. `{"Li6": 0.07589, "Li7": 0.92411, ...}`

Source code in `yani/_core/data.pyi`

```
138  def natural_abundance() -> typing.Any:
139      r"""
140      Return a dict of natural isotopic abundances by nuclide name.
141
142      Returns:
143          dict[str, float]: e.g. ``{"Li6": 0.07589, "Li7": 0.92411, ...}``
144      """
```

`func` natural_abundance_records

```
natural_abundance_records() -> Any
```

Return the TICE 2013 record behind each natural abundance, by nuclide name.

Table 1 of Meija et al., "Isotopic compositions of the elements 2013 (IUPAC Technical Report)", Pure Appl. Chem. 88(3), 293-306 (2016), doi:10.1515/pac-2015-0503 (© IUPAC, De Gruyter 2016), columns 4, 5, 6 and 9. :func: `natural_abundance` gives column 9 where it is a value and the column 6 best measurement where column 9 is an interval.

Each value is a dict with keys:

- `representative_value`, `representative_uncertainty`: column 9. TICE gives no coverage factor for this uncertainty.
- `representative_interval`: column 9 as `(low, high)` for the 12 elements given an interval instead of a value (H, Li, B, C, N, O, Mg, Si, S, Cl, Br, Tl).
- `observed_interval`: column 4, `(low, high)` of natural variation.
- `best_measurement`, `best_measurement_uncertainty`: column 6.
- `best_measurement_coverage`: column 6 coverage, e.g. `"2s"`.
- `best_measurement_calibration`: `"C"`, `"F"` or `"N"`. TICE prints this and the coverage once per element; they apply to every isotope of it.
- `annotations`: column 5, e.g. `"g,r"`.

`None` means the table leaves that field empty ("not stated", never zero). These uncertainties are reference data only: no calculation samples or propagates them.

Returns:

Type	Description
`Any`	dict[str, dict[str, Any]]: e.g. `{"Fe58": {"representative_value": 0.00282, "representative_uncertainty": 0.00012, ...}, ...}`

Source code in `yani/_core/data.pyi`

```
146  def natural_abundance_records() -> typing.Any:
147      r"""
148      Return the TICE 2013 record behind each natural abundance, by nuclide name.
149
150      Table 1 of Meija et al., "Isotopic compositions of the elements 2013
151      (IUPAC Technical Report)", Pure Appl. Chem. 88(3), 293-306 (2016),
152      doi:10.1515/pac-2015-0503 (© IUPAC, De Gruyter 2016), columns 4, 5, 6
153      and 9.
154      :func:`natural_abundance` gives column 9 where it is a value and the
155      column 6 best measurement where column 9 is an interval.
156
157      Each value is a dict with keys:
158
159      - ``representative_value``, ``representative_uncertainty``: column 9. TICE
160        gives no coverage factor for this uncertainty.
161      - ``representative_interval``: column 9 as ``(low, high)`` for the 12
162        elements given an interval instead of a value (H, Li, B, C, N, O, Mg,
163        Si, S, Cl, Br, Tl).
164      - ``observed_interval``: column 4, ``(low, high)`` of natural variation.
165      - ``best_measurement``, ``best_measurement_uncertainty``: column 6.
166      - ``best_measurement_coverage``: column 6 coverage, e.g. ``"2s"``.
167      - ``best_measurement_calibration``: ``"C"``, ``"F"`` or ``"N"``. TICE
168        prints this and the coverage once per element; they apply to every
169        isotope of it.
170      - ``annotations``: column 5, e.g. ``"g,r"``.
171
172      ``None`` means the table leaves that field empty ("not stated", never
173      zero). These uncertainties are reference data only: no calculation
174      samples or propagates them.
175
176      Returns:
177          dict[str, dict[str, Any]]: e.g. ``{"Fe58": {"representative_value": 0.00282,
178  "representative_uncertainty": 0.00012, ...}, ...}``
      """
```

`func` element_nuclides

```
element_nuclides() -> Any
```

Return a dict mapping element symbol to its naturally occurring nuclides.

Each value is a sorted list of nuclide names for that element.

Returns:

Type	Description
`Any`	dict[str, list[str]]: e.g. `{"Fe": ["Fe54", "Fe56", "Fe57", "Fe58"], ...}`

Source code in `yani/_core/data.pyi`

```
180 def element_nuclides() -> typing.Any:
181     r"""
182     Return a dict mapping element symbol to its naturally occurring nuclides.
183 
184     Each value is a sorted list of nuclide names for that element.
185 
186     Returns:
187         dict[str, list[str]]: e.g. ``{"Fe": ["Fe54", "Fe56", "Fe57", "Fe58"], ...}``
188     """
```

func element_names

```
element_names() -> Any
```

Return a dict mapping element symbol to full element name.

Returns:

Type	Description
`Any`	dict[str, str]: e.g. `{"Fe": "iron", "Li": "lithium", ...}`

Source code in `yani/_core/data.pyi`

```
190 def element_names() -> typing.Any:
191     r"""
192     Return a dict mapping element symbol to full element name.
193 
194     Returns:
195         dict[str, str]: e.g. ``{"Fe": "iron", "Li": "lithium", ...}``
196     """
```

func reaction_names

```
reaction_names() -> Any
```

Return a dict mapping MT numbers (int) to ENDF reaction notation strings. e.g. {2: "(n,elastic)", 18: "(n,fission)", 102: "(n,gamma)", ...}

Each MT's own name, not an inversion of `REACTION_MT`. Several names map to one MT (`"fission"` and `"(n,fission)"` are both MT 18), so inverting a HashMap picked whichever the iteration order reached last: the same build could return `{18: "fission"}` on one run and `{18: "(n,fission)"}` on the next.

Source code in `yani/_core/data.pyi`

```
198  def reaction_names() -> typing.Any:
199      r"""
200      Return a dict mapping MT numbers (int) to ENDF reaction notation strings.
201      e.g. {2: "(n,elastic)", 18: "(n,fission)", 102: "(n,gamma)", ...}
202
203      Each MT's own name, not an inversion of `REACTION_MT`. Several names map to
204      one MT (`"fission"` and `"(n,fission)"` are both MT 18), so inverting a
205      HashMap picked whichever the iteration order reached last: the same build
206      could return `{18: "fission"}` on one run and `{18: "(n,fission)"}` on the
207      next.
208      """
```

func clear_nuclide_cache

```
clear_nuclide_cache() -> None
```

Clear any internally cached nuclide data.

This forces subsequent reads to re-parse JSON files.

Returns:

Type	Description
`None`	None

Source code in `yani/_core/data.pyi`

```
210  def clear_nuclide_cache() -> None:
211      r"""
212      Clear any internally cached nuclide data.
213
214      This forces subsequent reads to re-parse JSON files.
215
216      Returns:
217          None
218      """
```

func split_nuclide

```
split_nuclide(name: str) -> tuple[str, int, int]
```

Split a nuclide name into its element symbol, mass number and isomeric state.

The one place downstream code should ask, because the obvious way of doing it by hand is wrong for exactly the entry that is easiest to forget: stripping the trailing digits off `"Ta180_m1"` leaves `"Ta180_m"` and a mass number of 1, and natural tantalum really does contain Ta180m, so a composition built from :func:`natural_abundance` will hand you that name.

Parameters:

Name	Type	Description	Default
`name`	`str`	A nuclide name, e.g. `"Fe56"` or `"Ta180_m1"`.	*required*

Returns:

Type	Description
`tuple[str, int, int]`	tuple[str, int, int]: Element symbol, mass number, and isomeric state (0 for a ground state, 1 for the first metastable, and so on).

Raises:

Type	Description
`ValueError`	If the name is not a nuclide name.

Examples:

```
>>> yani.data.split_nuclide("Fe56")
('Fe', 56, 0)
>>> yani.data.split_nuclide("Ta180_m1")
('Ta', 180, 1)
```

Source code in `yani/_core/data.pyi`

```
def split_nuclide(name: builtins.str) -> tuple[builtins.str, builtins.int,
builtins.int]:
    r"""
    Split a nuclide name into its element symbol, mass number and isomeric state.

    The one place downstream code should ask, because the obvious way of doing
    it by hand is wrong for exactly the entry that is easiest to forget:
    stripping the trailing digits off ``"Ta180_m1"`` leaves ``"Ta180_m"`` and a
    mass number of 1, and natural tantalum really does contain Ta180m, so a
    composition built from :func:`natural_abundance` will hand you that name.

    Args:
        name (str): A nuclide name, e.g. ``"Fe56"`` or ``"Ta180_m1"``.

    Returns:
        tuple[str, int, int]: Element symbol, mass number, and isomeric state
            (0 for a ground state, 1 for the first metastable, and so on).

    Raises:
        ValueError: If the name is not a nuclide name.

    Examples:
        >>> yani.data.split_nuclide("Fe56")
        ('Fe', 56, 0)
        >>> yani.data.split_nuclide("Ta180_m1")
        ('Ta', 180, 1)
    """
```

func atomic_symbol

```
atomic_symbol(z: int) -> str
```

Return the chemical symbol of an atomic number.

Bound so that a caller naming files or directories from ENDF data does not have to carry its own copy of the periodic table, or install a second nuclear-data package to borrow one.

Parameters:

Name	Type	Description	Default
`z`	`int`	Atomic number, 0 to 118.	*required*

Returns:

Name	Type	Description
`str`	`str`	The chemical symbol, e.g. `"Fe"`.

Raises:

Type	Description
`ValueError`	If there is no element with that atomic number.

Examples:

```
>>> yani.data.atomic_symbol(26)
'Fe'
>>> yani.data.atomic_symbol(0)
'n'
```

Note

Z = 0 is the neutron, `"n"`, and is not an error. ENDF libraries really do ship a neutron material (JENDL is one), so a caller walking a sublibrary will meet it.

Source code in `yani/_core/data.pyi`

```
247  def atomic_symbol(z: builtins.int) -> builtins.str:
248      r"""
249      Return the chemical symbol of an atomic number.
250
251      Bound so that a caller naming files or directories from ENDF data does not
252      have to carry its own copy of the periodic table, or install a second
253      nuclear-data package to borrow one.
254
255      Args:
256          z (int): Atomic number, 0 to 118.
257
258      Returns:
259          str: The chemical symbol, e.g. ``"Fe"``.
260
261      Raises:
262          ValueError: If there is no element with that atomic number.
263
264      Examples:
265          >>> yani.data.atomic_symbol(26)
266          'Fe'
267          >>> yani.data.atomic_symbol(0)
268          'n'
269
270      Note:
271          Z = 0 is the neutron, ``"n"``, and is not an error. ENDF libraries
272          really do ship a neutron material (JENDL is one), so a caller walking a
273          sublibrary will meet it.
274      """
```

`func` atomic_number

```
atomic_number(symbol: str) -> int
```

Return the atomic number of a chemical symbol.

The inverse of :func:`atomic_symbol`.

Parameters:

Name	Type	Description	Default
`symbol`	`str`	A chemical symbol, e.g. `"Fe"`. Case sensitive, and the symbol only: `"fe"` and `"iron"` are both refused.	*required*

Returns:

Name	Type	Description
`int`	`int`	The atomic number.

Raises:

Type	Description
`ValueError`	If the symbol names no element.

Examples:

```
>>> yani.data.atomic_number("Fe")
26
```

Source code in `yani/_core/data.pyi`

```
276  def atomic_number(symbol: builtins.str) -> builtins.int:
277      r"""
278      Return the atomic number of a chemical symbol.
279
280      The inverse of :func:`atomic_symbol`.
281
282      Args:
283          symbol (str): A chemical symbol, e.g. ``"Fe"``. Case sensitive, and the
284              symbol only: ``"fe"`` and ``"iron"`` are both refused.
285
286      Returns:
287          int: The atomic number.
288
289      Raises:
290          ValueError: If the symbol names no element.
291
292      Examples:
293          >>> yani.data.atomic_number("Fe")
294          26
295      """
```

Material collections

The bundled PNNL collection is available as `yani.materials.pnnl`; see Materials for selecting a composition.

mod yani.materials

attr pnnl module-attribute

```
pnnl: MaterialCollection
```

The PNNL Compendium of Material Composition Data for Radiation Transport Modeling (PNNL-15870, Rev. 2): 410 materials keyed by the report's own names.

---

class MaterialCollection

A read-only, name-keyed table of published material compositions.

Indexing returns an ordinary :class:`Material`, freshly built each time, so it can be renamed and mutated without affecting the collection or any other lookup of the same name.

Examples:

```
>>> import yani
>>> steel = yani.materials.pnnl["Steel, Stainless 304"]
>>> steel.name = "firstwall_material"
>>> yani.materials.pnnl.search("concrete")[:2]
['Concrete, Barite (Type BA)', 'Concrete, Barytes-Limonite']
```

attr citation property

```
citation: str
```

Short citation for the source document, e.g. `"PNNL-15870, Rev. 2"`.

attr url property

```
url: str
```

URL of the source document.

meth names

```
names() -> list[str]
```

Every material name, in source-document order.

Returns:

Type	Description
`list[str]`	list[str]: All names in the collection.

Source code in `yani/_core/materials.pyi`

```
74  def names(self) -> builtins.list[builtins.str]:
75      r"""
76      Every material name, in source-document order.
77
78      Returns:
79          list[str]: All names in the collection.
80      """
```

meth search

```
search(query: str) -> list[str]
```

Names containing `query`, compared case-insensitively.

Parameters:

Name	Type	Description	Default
`query`	`str`	Substring to look for, e.g. `"concrete"`.	*required*

Returns:

Type	Description
`list[str]`	list[str]: Matching names, ready to pass straight back to `[]`.

Source code in `yani/_core/materials.pyi`

```
81  def search(self, query: builtins.str) -> builtins.list[builtins.str]:
82      r"""
83      Names containing ``query``, compared case-insensitively.
84
85      Args:
86          query (str): Substring to look for, e.g. ``"concrete"``.
87
88      Returns:
89          list[str]: Matching names, ready to pass straight back to ``[]``.
90      """
```

`meth` material

```
material(key: str, *, name: Optional[str] = None, density: Optional[float] =
None, density_units: str = 'g/cm3', temperature: Optional[float] = None,
volume: Optional[float] = None, transmutable: bool = False, id: Optional[int]
= None) -> Material
```

Build the named material, overriding fields the collection does not set.

Equivalent to `collection[key]` followed by attribute assignment, except for `density`, which :class:`Material` does not allow to be reassigned after construction. Use it for a material used at other than its reference density, e.g. a concrete at a different porosity or a steel diluted by a coolant void fraction.

Examples:

```
>>> steel = yani.materials.pnnl.material("Steel, Stainless 316",
temperature=600)
>>> porous = yani.materials.pnnl.material(
...     "Steel, Stainless 316", density=7000, density_units="kg/m3"
... )
```

Parameters:

Name	Type	Description	Default
`key`	`str`	The source document's name, verbatim.	*required*
`name`	`Optional[str]`	Rename the material, e.g. to match a mesh geometry's material name.	`None`
`density`	`Optional[float]`	Density in `density_units`, replacing the collection's reference density. Warns when it is higher than the reference density, which no void fraction or porosity can give.	`None`
`density_units`	`str`	Units of `density`: `"g/cm3"` (default), `"g/cc"`, `"kg/m3"` or `"atom/barn-cm"`.	`'g/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.	`None`
`volume`	`Optional[float]`	Volume in cm³.	`None`
`transmutable`	`bool`	Mark for depletion by `Model.simulate_transmutation`.	`False`
`id`	`Optional[int]`	Material ID. Defaults to a fresh one, which no earlier material was given.	`None`

Returns:

Name	Type	Description
`Material`	`Material`	The material with the overrides applied.

Source code in `yani/_core/materials.pyi`

```
def material(self, key: builtins.str, *, name: typing.Optional[builtins.str] =
None, density: typing.Optional[builtins.float] = None, density_units:
builtins.str = 'g/cm3', temperature: typing.Optional[builtins.float] = None,
volume: typing.Optional[builtins.float] = None, transmutable: builtins.bool =
False, id: typing.Optional[builtins.int] = None) -> Material:
    r"""
    Build the named material, overriding fields the collection does not set.

    Equivalent to ``collection[key]`` followed by attribute assignment,
    except for ``density``, which :class:`Material` does not allow to be
    reassigned after construction. Use it for a material used at other
    than its reference density, e.g. a concrete at a different porosity
    or a steel diluted by a coolant void fraction.

    Examples:
        >>> steel = yani.materials.pnnl.material("Steel, Stainless 316",
temperature=600)
        >>> porous = yani.materials.pnnl.material(
        ...     "Steel, Stainless 316", density=7000, density_units="kg/m3"
        ... )

    Args:
        key (str): The source document's name, verbatim.
        name (Optional[str]): Rename the material, e.g. to match a mesh
            geometry's material name.
        density (Optional[float]): Density in ``density_units``, replacing
            the collection's reference density. Warns when it is higher
            than the reference density, which no void fraction or porosity
            can give.
        density_units (str): Units of ``density``: ``"g/cm3"`` (default),
            ``"g/cc"``, ``"kg/m3"`` or ``"atom/barn-cm"``.
        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.
        volume (Optional[float]): Volume in cm³.
        transmutable (bool): Mark for depletion by
            ``Model.simulate_transmutation``.
        id (Optional[int]): Material ID. Defaults to a fresh one, which no
            earlier material was given.

    Returns:
        Material: The material with the overrides applied.
    """
```

`meth` entry

```
entry(name: str) -> dict
```

The source document's own record for a material, for citation.

Parameters:

Name	Type	Description	Default
`name`	`str`	The source document's name, verbatim.	*required*

Returns:

Name	Type	Description
`dict`	`dict`	`name`, `number` (the entry number in the source
	`dict`	document), `density` (g/cm3), `formula` (or None),
	`dict`	`composition` (element symbol or nuclide name -> atom
	`dict`	fraction), `citation` and `url`.

Source code in `yani/_core/materials.pyi`

```
129  def entry(self, name: builtins.str) -> dict:
130      r"""
131      The source document's own record for a material, for citation.
132
133      Args:
134          name (str): The source document's name, verbatim.
135
136      Returns:
137          dict: ``name``, ``number`` (the entry number in the source
138          document), ``density`` (g/cm3), ``formula`` (or None),
139          ``composition`` (element symbol or nuclide name -> atom
140          fraction), ``citation`` and ``url``.
141      """
```

`class` NameIter

Iterator over a collection's material names.

`func` collections

```
collections() -> list[str]
```

Names of the available material collections.

Returns:

Type	Description
`list[str]`	list[str]: Collection names, each an attribute of `yani.materials`.

Source code in `yani/_core/materials.pyi`

```
151  def collections() -> builtins.list[builtins.str]:
152      r"""
153      Names of the available material collections.
154
155      Returns:
156          list[str]: Collection names, each an attribute of ``yani.materials``.
157      """
```

Source distributions

The energy distributions used by irradiation spectra live in `yani.sources`. See Spectra for setting a pulse's shape and flux magnitude.

mod yani.sources

---

class Histogram

```
Histogram(boundaries: Any, probabilities: Sequence[float])
```

Histogram (piecewise-constant) energy distribution over energy bins.

Create a histogram (piecewise-constant) energy distribution.

Each bin `i` spans `boundaries[i]` to `boundaries[i + 1]` and holds probability mass `probabilities[i]`; an energy is sampled by choosing a bin in proportion to its mass and then drawing uniformly within it.

Parameters:

Name	Type	Description	Default
`boundaries`	`Any`	Either explicit energy bin edges in eV (length `n + 1`, strictly ascending) or the name of a built-in group structure (e.g. `"CCFE-709"`, `"VITAMIN-J-175"`), whose edges are used.	*required*
`probabilities`	`Sequence[float]`	Relative probability mass for each of the `n` bins. Normalized internally, so only the relative values matter (a raw multigroup flux per group can be passed directly).	*required*

Returns:

Name	Type	Description
`Histogram`	`Histogram`	A histogram energy distribution.

Raises:

Type	Description
`ValueError`	If a named group structure is unknown, if `boundaries` is not exactly one longer than `probabilities`, if the boundaries are not strictly ascending, if any probability is negative, or if all are zero.

Examples:

```
>>> import yani
>>> # 30% of source neutrons in [0, 1] MeV, 70% in [1, 20] MeV
>>> energy_dist = yani.sources.Histogram([0.0, 1e6, 20e6], [0.3, 0.7])
>>> # or from a named group structure (709 per-group weights)
>>> energy_dist = yani.sources.Histogram("CCFE-709", flux_709)
```

Source code in `yani/_core/sources.pyi`

```
def __new__(cls, boundaries: typing.Any, probabilities:
typing.Sequence[builtins.float]) -> Histogram:
    r"""
    Create a histogram (piecewise-constant) energy distribution.

    Each bin ``i`` spans ``boundaries[i]`` to ``boundaries[i + 1]`` and holds
    probability mass ``probabilities[i]``; an energy is sampled by choosing a
    bin in proportion to its mass and then drawing uniformly within it.

    Args:
        boundaries: Either explicit energy bin edges in eV (length ``n + 1``,
            strictly ascending) or the name of a built-in group structure
            (e.g. ``"CCFE-709"``, ``"VITAMIN-J-175"``), whose edges are used.
        probabilities: Relative probability mass for each of the ``n`` bins.
            Normalized internally, so only the relative values matter (a raw
            multigroup flux per group can be passed directly).

    Returns:
        Histogram: A histogram energy distribution.

    Raises:
        ValueError: If a named group structure is unknown, if ``boundaries``
            is not exactly one longer than ``probabilities``, if the
            boundaries are not strictly ascending, if any probability is
            negative, or if all are zero.

    Examples:
        >>> import yani
        >>> # 30% of source neutrons in [0, 1] MeV, 70% in [1, 20] MeV
        >>> energy_dist = yani.sources.Histogram([0.0, 1e6, 20e6], [0.3, 0.7])
        >>> # or from a named group structure (709 per-group weights)
        >>> energy_dist = yani.sources.Histogram("CCFE-709", flux_709)
    """
```

attr boundaries property

```
boundaries: list[float]
```

Get the bin boundaries.

Returns:

Type	Description
`list[float]`	List[float]: Energy bin edges in eV, length `n + 1`.

attr probabilities property

```
probabilities: list[float]
```

Get the per-bin probabilities, as supplied.

Returns:

Type	Description
`list[float]`	List[float]: Relative probability mass for each bin.

`meth` sample

```
sample() -> float
```

Sample an energy value from the distribution.

Returns:

Name	Type	Description
`float`	`float`	Sampled energy in eV.

Source code in `yani/_core/sources.pyi`

```
70  def sample(self) -> builtins.float:
71      r"""
72      Sample an energy value from the distribution.
73
74      Returns:
75          float: Sampled energy in eV.
76      """
```

Self-shielding shapes

The lump shapes passed to `Material.transmute()` control its resonance self-shielding correction; see Self-shielding.

mod yani.shapes

class SphereLump

```
SphereLump()
```

A sphere, sized by the material's volume.

The extreme case: a sphere has the least surface for its volume, so it has the longest mean chord and shields more than any other shape of the same volume. Convenient, but an upper bound rather than a neutral choice.

A sphere. Its radius follows from `Material(volume=...)`.

Source code in `yani/_core/shapes.pyi`

```
30  def __new__(cls) -> SphereLump:
31      r"""
32      A sphere. Its radius follows from ``Material(volume=...)``.
33      """
```

class CubeLump

```
CubeLump()
```

A cube, sized by the material's volume.

A cube. Its side follows from `Material(volume=...)`.

Source code in `yani/_core/shapes.pyi`

```
41  def __new__(cls) -> CubeLump:
42      r"""
43      A cube. Its side follows from ``Material(volume=...)``.
44      """
```

class FoilLump

```
FoilLump(thickness: float)
```

A flat slab of a stated thickness.

The chord is twice the thickness whatever the area, so a foil's shielding does not depend on the volume at all -- the volume is still what makes `activity` and `decay_heat` available, but it plays no part here.

Args: thickness (float): Foil thickness in cm.

Source code in `yani/_core/shapes.pyi`

```
58  def __new__(cls, thickness: builtins.float) -> FoilLump:
59      r"""
60      Args:
61      thickness (float): Foil thickness in cm.
62      """
```

class CylinderLump

```
CylinderLump(radius: float)
```

A cylinder of a stated radius; its height follows from the volume.

Args: radius (float): Cylinder radius in cm.

Source code in `yani/_core/shapes.pyi`

```
72  def __new__(cls, radius: builtins.float) -> CylinderLump:
73      r"""
74      Args:
75      radius (float): Cylinder radius in cm.
76      """
```

`class` WireLump

```
WireLump(radius: float)
```

A cylinder long enough that its ends do not matter. Chord `2r`.

Args: radius (float): Wire radius in cm.

Source code in `yani/_core/shapes.pyi`

```
86  def __new__(cls, radius: builtins.float) -> WireLump:
87      r"""
88      Args:
89      radius (float): Wire radius in cm.
90      """
```