Curie Documentation

Release 0.3.0

Jonathan Morrell

Jul 18, 2026

CONTENTS

Curie is a Python toolkit for the analysis of experimental nuclear data, with primary applications in (gamma-ray) activation analysis and the charged-particle stacked-target activation technique. Its name is inspired by Marie Curie, who pioneered the study of radioactivity. Alongside the analysis tools, it provides access to nuclear structure, decay, and reaction databases, and to atomic properties such as photon attenuation coefficients and charged-particle stopping powers.

```
>>> import curie as ci
>>> ci.Isotope('225RA').half_life('d')
14.9
>>> ci.Reaction('115IN(n,g)').plot(scale='loglog')   # a cross section, straight from␣
↪the libraries
```

Curie's functionality is provided by a small set of classes. The table below maps common tasks to the classes that carry them out (linked to the API reference) and the guide section that walks through each one.

Task	Classes	Guide
Fit full-energy peaks in HPGe (high-purity germanium) spectra; energy, resolution and efficiency calibration; extract activities	*Spectrum*, *Calibration*	*Spectroscopy*
Solve the Bateman equations; fit production rates and end-of-bombardment activities to counting data	*DecayChain*	*Isotopes & Decay Chains*
Retrieve, interpolate, and flux-average evaluated cross sections (ENDF/B-VIII.1, TENDL-2025, IRDFF-II, IAEA)	*Reaction*, *Library*	*Reactions*
Compute charged-particle energy loss and flux distributions through a target stack	*Stack*	*Stopping Power Calculations*
Look up decay data: half-lives, gamma intensities, branching ratios, atomic masses	*Isotope*	*Isotopes & Decay Chains*
Stopping powers, ranges, and photon attenuation coefficients for elements and compounds	*Element*, *Compound*	*Stopping Power Calculations*

New to activation analysis itself? The *A Beginner's Guide to Activation Analysis* introduces the field from the ground up. Otherwise, install Curie (*Installation*) and take the *Quickstart* tour. Throughout the documentation (and in all of the docstring examples), Curie is imported as `import curie as ci`.

Citing Curie. If Curie contributes to published work, please cite it, for example: J. T. Morrell, *Curie: A Python toolkit to aid in the analysis of experimental nuclear data* (2019–), https://jtmorrell.github.io/curie/.

CHAPTER

ONE

INSTALLATION

Curie is available through the Python Package Index, which allows installation using Python's standard command line utility pip. Assuming Python and pip are installed already, you can install Curie with the command:

```
pip install --user curie
```

or:

```
python -m pip install --user curie
```

If you'd like to install the most recent (unreleased) version of Curie, you can clone the Curie GitHub repository and install it from the source tree:

```
git clone https://github.com/jtmorrell/curie.git
python -m pip install --user ./curie
```

1.1 Nuclear data

Curie downloads the nuclear data files it needs the first time they are used: each database is fetched from the curie data release on first access, verified against its published SHA256 checksum, and stored in a per-user data directory. The large cross-section libraries (ENDF and the TENDL variants) are fetched in small per-target pieces, so looking up one reaction downloads well under a MB rather than a 37-748 MB library. No action is needed to make this happen — the first `ci.Reaction(...)` or `ci.Isotope(...)` just works.

To prepare a machine for offline use (or to pre-populate the data directory in one step), download everything explicitly:

```
import curie as ci
ci.download()
```

If your data files are stale or corrupted, re-download them with:

```
ci.download(overwrite=True)
```

The data directory follows your platform's convention: `~/.local/share/curie` on Linux, `~/Library/Application Support/curie` on macOS, and `%LOCALAPPDATA%\curie` on Windows. To place the data somewhere else — a shared network drive, an air-gapped machine's data directory — set the `CURIE_DATA_DIR` environment variable; Curie will use that directory for all of its data. On an air-gapped machine, copy the data directory from a connected machine (or the individual `.db` files from the curie data release) into `CURIE_DATA_DIR`.

On a shared (multi-user) machine, an administrator can instead populate the site-wide data directory once — `/usr/local/share/curie` on Linux, `/Library/Application Support/curie` on macOS, `C:\ProgramData\curie` on Windows — and Curie will use those files read-only in place from every account, with nothing downloaded or

duplicated per user. Files in the site-wide directory (like those in `CURIE_DATA_DIR`) are trusted as provided: keep them in sync with the data release matching the installed Curie version when upgrading.

Data installed by Curie versions before 0.1.0 (inside the package's own directory) is found and adopted into the cache automatically — nothing is re-downloaded after an upgrade.

CHAPTER

TWO

QUICKSTART

Once Curie is installed (see *Installation*), everything is reached through a single import:

```
import curie as ci
```

This page maps out the toolkit — which class does what, and where to read more — and gives a few short examples to run.

2.1 The Curie toolbox

Curie's functionality lives in a handful of classes, grouped here by what you would use them for. The classes are designed to work together: peaks fit from a spectrum feed a decay-chain fit, a foil's particle flux feeds a cross-section average, and so on.

Fitting gamma-ray spectra. `Spectrum` fits the peaks in HPGe (high-purity germanium) detector data, using a `Calibration` (energy, efficiency and resolution) and `Isotope` decay data to turn peak areas into activities. See *Spectroscopy*.

Production and decay calculations. `Isotope` provides decay data — half-lives, decay radiations, dose rates — for a single nuclide, while `DecayChain` solves the Bateman equations for a whole chain: forward (predicting activities from a production rate) or inverse (fitting a production rate or initial activity to measured decays, including peaks read straight from a `Spectrum`). See *Isotopes & Decay Chains*.

Nuclear reaction data. `Reaction` gives a cross section as a function of energy, with interpolation, flux-averaging and plotting; `Library` searches the evaluated libraries (ENDF, TENDL, IRDFF, IAEA) for what is available. See *Reactions*.

Stopping powers and stacked targets. `Element` and `Compound` compute charged-particle stopping powers, ranges and photon attenuation coefficients; `Stack` transports a beam through a stack of foils to find the particle energy in each — a flux that can be handed straight to `Reaction.average`. See *Stopping Power Calculations*.

For the models and formulas behind these methods, see the *Theory & Methodology* chapter; for every method and attribute, the *API*.

2.2 A few things to try

The first two examples need no local files — Curie fetches the nuclear data they use on first access.

Look up the decay data for a radionuclide:

```
>>> ip = ci.Isotope('225RA')
>>> print(ip.half_life('d'))
14.9
```

(continues on next page)

(continued from previous page)

```
>>> print(ip.gammas())          # energies (keV) and intensities (%)
   energy  intensity  unc_intensity
0    40.0       30.0            1.5
```

Plot an evaluated reaction cross section (this one resolves to IRDFF-II):

```
rx = ci.Reaction('115IN(n,g)')  # neutron capture on 115In
rx.plot(scale='loglog')
```

The third fits peaks in a real gamma-ray spectrum, so it needs a data file. Download `eu_calib_7cm.Spe` from the examples directory of the repository and run from the folder you saved it in (or point the path at your own `.Spe`, `.Chn`, `.CNF` or `.IEC` spectrum):

```
sp = ci.Spectrum('eu_calib_7cm.Spe')
sp.isotopes = ['152EU']
sp.plot()                       # the spectrum, with its fitted peaks
```

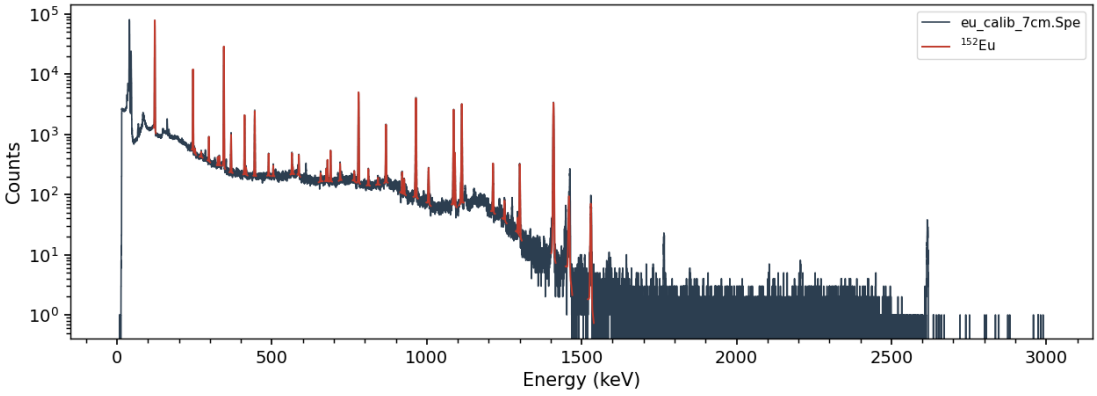


Fig. 1: The fitted ^{152}Eu spectrum the code above produces.

Each links to a fuller, worked walk-through: *Spectroscopy*, *Isotopes & Decay Chains*, *Reactions*.

2.3 Example scripts

Complete, runnable scripts covering each of these areas ship in the examples directory of the Curie repository, alongside the example data files they use.

CHAPTER

THREE

A BEGINNER'S GUIDE TO ACTIVATION ANALYSIS

This chapter is a conceptual introduction to activation analysis — the field Curie was built to serve — for readers who are new to it. Where the *User's Guide* shows *how* to drive Curie and the *Theory & Methodology* chapter gives the equations, this guide explains *why* the analysis looks the way it does: what is being measured, and how each number leads to the next. Each section ends with a short pointer to the Curie classes that carry out that step.

3.1 What is activation analysis?

The idea behind activation analysis is to learn about matter by making it briefly radioactive and watching it decay. A sample is exposed to a beam or flux of particles — neutrons from a reactor, or charged particles (protons, deuterons, alphas) from an accelerator. A fraction of the nuclei in the sample undergo nuclear reactions and are transformed into new, often radioactive, nuclei. Those product nuclei then decay, emitting radiation — most usefully gamma rays, whose energies are sharp and characteristic of the emitting isotope. By detecting that radiation, you can identify which isotopes were produced and, from how much radiation you see, *how much* of each.

That single chain of events — **produce, decay, detect** — underlies two kinds of measurement Curie supports:

- **Quantifying activity.** Given a sample that has been irradiated, how active is each isotope in it? This is the everyday task of gamma-ray spectroscopy: turn a measured spectrum into a table of activities.
- **Measuring cross sections.** How readily does a particular reaction occur, as a function of the beam energy? This is the goal of the charged-particle stacked-target technique, where the measured activities are worked backward into the underlying reaction probabilities — the data that let you predict, before an irradiation, how much of a desired radioisotope (and how much co-produced impurity) a given beam and target will make.

The rest of this guide follows the produce–decay–detect chain forward and backward: the physics of production and decay, how activity is measured, how a measured activity becomes a production rate, how charged particles behave in a target, and finally how all of this is combined to predict yields and design an experiment.

In Curie. Every class in the toolkit sits somewhere on this chain; the *Quickstart* maps them out.

3.2 Production, decay, and counting

Three intervals of time structure every activation measurement: the sample is **irradiated**, then it **cools** (decays) for a while, and then it is **counted** in a detector. Following one product isotope through these intervals gives the whole quantitative framework.

Production. During irradiation the product is created at some rate R (atoms per second) and, being radioactive, decays at the same time with decay constant $\lambda = \ln 2 / t_{1/2}$. The number of product atoms climbs until creation and decay balance. For a constant production rate over an irradiation of duration t_{irr}, the activity (the number of decays per second) at the end of the irradiation — the *end of bombardment*, or EoB — is

$$A_{\mathrm{EoB}} = R\left(1 - e^{-\lambda t_{\mathrm{irr}}}\right).$$

The term in parentheses is the **saturation factor**. Irradiate for one half-life and you reach half the maximum possible activity; irradiate for several half-lives and the activity *saturates* at $A = R$ — no matter how long you keep going, the isotope decays as fast as it is made. This is the single most important consequence of the half-life for planning: short-lived products saturate quickly (little is gained by a long irradiation) while long-lived products build up slowly.

Decay. After the beam is off, the activity simply decays:

$$A(t) = A_{\mathrm{EoB}}\, e^{-\lambda t},$$

with t measured from EoB. Time zero for the whole analysis is conventionally this end-of-bombardment moment.

Counting. A detector does not measure activity directly; over a counting interval it records the *number of decays* that occur in that window (times the fraction it manages to detect). Because the activity is itself falling during a long count, the number of decays is the integral of $A(t)$ across the counting interval, not simply activity times time. Keeping these three intervals — and their clocks — straight is the bookkeeping that the whole analysis depends on.

In Curie. `DecayChain` implements exactly this, and generalizes it from one isotope to a full decay chain (parents feeding daughters) via the Bateman equations. See *Isotopes & Decay Chains* and, for the equations, *Radioactive Decay Chains*.

3.3 Measuring activity: gamma-ray spectroscopy

The most common way to observe the decays is gamma-ray spectroscopy with a high-purity germanium (HPGe) detector. Each decaying isotope emits gamma rays at a set of characteristic energies, each with a known *intensity* — its emission probability, the fraction of decays that produce that gamma. The detector sorts detected gammas by energy into a **spectrum** — a histogram of counts versus energy — in which each isotope appears as peaks at its own line energies.

To turn the area of a peak into an activity, three calibrations are needed, and it is worth understanding why each is required:

- **Energy calibration** maps detector channel to gamma energy, so that a peak can be matched to the isotope and line that produced it.
- **Resolution calibration** describes how wide the peaks are, which the fitting routine needs to separate nearby lines.
- **Efficiency calibration** gives the fraction of gammas emitted at a given energy that are actually recorded as full-energy counts — counts from gammas that deposited their entire energy in the detector. Without it a peak area is only a relative number; with it, the count rate becomes an absolute emission rate. Efficiency falls with distance and varies strongly with energy, so it must be measured for the specific detector and geometry, using a source of known activity.

Given these, the number of counts in a peak relates to the activity through the gamma intensity, the efficiency, and the counting-time factors of the previous section. Inverting that relation for every clean peak yields the isotope's activity — the measured quantity that the rest of the analysis builds on.

In Curie. `Spectrum` fits the peaks and `Calibration` produces and stores the three calibrations. See *Spectroscopy*, and *Gamma-ray Peak Fitting* and *Detector Calibration* for the models.

3.4 Turning activities into production rates

Measuring an activity is a means to an end. What you usually want is the quantity that was under experimental control: the **production rate** during irradiation (or, for a decay-only sample, the activity at some reference time). This is the produce–decay–detect chain run *backward*.

For a single isotope it is a matter of algebra: rearranging the saturation relation from *Production, decay, and counting*,

$$R = \frac{A_{\mathrm{EoB}}}{1 - e^{-\lambda t_{\mathrm{irr}}}},$$

and A_{EoB} itself is obtained by decay-correcting the activity measured at count time back to EoB. In practice several complications enter at once: the isotope of interest may be fed by the decay of a parent, several spectra taken at different cooling times each constrain the same production rate, and the beam may have varied during the irradiation. The robust way to handle all of this is to *fit*: adjust the production rate until the decays it predicts, across the whole chain and all counting intervals, best match the measured ones.

A useful distinction: fitting a **production rate** applies when the sample was being made during an irradiation, whereas fitting an **initial activity** applies to a sample that was simply decaying from some reference time (a calibration source, say). The two answer different questions and take different starting information.

In Curie. `DecayChain.get_counts` reads measured decays (including straight from fitted `Spectrum` peaks), and `fit_R` and `fit_A0` fit a production rate or an initial activity to them. See the *Decay Chain Worked Examples* for a full inverse example.

3.5 Charged-particle interactions in a target

So far the beam could have been anything. Charged-particle beams behave in a way that shapes the entire stacked-target technique, so they deserve their own discussion.

Unlike neutrons, which travel until they happen to react, a charged particle interacts continuously with the electrons of the material it traverses, losing energy little by little. The rate of energy loss per unit path length is the **stopping power**. Two consequences follow, and both are central to activation work:

- **The beam energy decreases with depth.** A particle that enters a foil at one energy leaves it at a lower one, and eventually — after a distance called its **range** — stops entirely. Because a reaction's cross section depends on energy, the production rate is not uniform through a thick target: different depths are effectively irradiated at different energies.
- **The beam spreads in energy.** A foil sees a *range* of energies, not a single value: partly because the beam loses energy continuously across the foil's own thickness, and partly because any initial energy spread grows as the beam slows (slower particles lose energy faster). Random fluctuations in the energy loss (*straggling*) broaden it further still.

These facts are usually a nuisance — but they can be turned into an advantage. If a thin foil barely changes the beam energy, it measures a reaction essentially at a single energy. Stack many thin foils, with degraders between them to step the energy down, and one irradiation samples the reaction at a whole ladder of energies at once. This is the **stacked-target** technique, and reading the energy in each foil correctly is the crux of it.

In Curie. `Element` and `Compound` compute stopping powers and ranges; `Stack` transports a beam through a stack of foils and reports the energy distribution in each. See *Stopping Power Calculations* and *Stopping Power and Particle Transport*.

3.6 Predicting production rates and isotope yields

With the pieces in hand, the forward calculation — how much of a product will an irradiation make? — comes together. In a single foil, the production rate is the reaction cross section combined with the beam and the target:

$$R = I_p \, n_t \, \langle \sigma \rangle,$$

where I_p is the beam current (particles per second), n_t the number of target atoms per unit area, and $\langle \sigma \rangle$ the cross section *averaged over the energy distribution of the beam in that foil* — the foil's energy spectrum, which `Stack.get_flux` supplies. (Curie also folds in the unit factor converting mb to cm2; the *Reactions* page gives the fully dimensioned

form.) That flux-averaging is where the physics of the previous section enters: for a thin foil the average is essentially the cross section at the beam energy, but for a thick foil, where the beam spans a wide energy range, using the cross section at the mean energy alone can be badly wrong — the average must be taken over the real distribution.

Once R is known, the activation equation of *Production, decay, and counting* turns it into an activity at end of bombardment, and the decay and counting relations turn *that* into the number of decays a detector would record at any later time. The chain is now closed: a cross section, a beam, and a target predict the very counts that a measurement would produce — and, run the other way, measured counts yield the cross section.

In Curie. `Reaction.average` (and `Reaction.integrate`) combine a cross section with a beam spectrum — for a foil, `Stack.get_flux` supplies that spectrum — and the result feeds `DecayChain` as a production rate. See the "Averages vs. integrals" discussion on the *Reactions* page, and the *Stopping Power Worked Examples* for the thin-versus-thick comparison.

3.7 Designing a stacked-target experiment

A stacked-target cross-section measurement is planned backward from what it needs to produce: measurable activities of the product isotopes, at a set of well-known beam energies, with the beam current under control. The main design choices are:

- **Energy coverage.** The incident beam energy and the stack's degraders set the range of energies sampled. Enough foils are included, with degraders sized so that consecutive target foils sit at usefully spaced energies across the region of interest.
- **Target foils.** Each target foil must be thin enough that the beam energy is well defined across it (so the measured cross section belongs to a narrow energy), yet thick enough to produce enough activity to measure. These pull in opposite directions and are traded off against the expected cross section and beam current.
- **Monitor foils.** Interleaved foils of well-characterized materials carry *monitor reactions* — reactions whose cross sections are known to high accuracy. Measuring their activities reconstructs the actual beam current and energy at each position in the stack, which is how the experiment calibrates itself rather than trusting the nominal beam parameters.
- **Cooling and counting schedule.** The half-lives of the products dictate the timing: short-lived isotopes must be counted soon after irradiation, long-lived ones may need to cool so that short-lived activities decay away first.
- **Range check.** The beam must actually reach the last foil of interest: the material in front of it has to stay within the particle's range at the incident energy (degraders or catchers — passive foils that stop the beam or recoiling reaction products — further downstream may lie beyond the range without harm).

The Curie workflow mirrors the experiment. *Before* the beam time, define the foils and use `Stack` to predict the energy in each, then combine monitor and product cross sections with those fluxes to estimate the activities you will produce — confirming the design will yield measurable, well-placed data. *After* the irradiation, fit the counted spectra with `Spectrum` and `Calibration`, decay-correct and fit production rates with `DecayChain`, normalize to the monitor foils, and divide out the beam and target to recover the cross sections.

In Curie. This ties together every part of the toolkit; the four topic groups of the *User's Guide* cover the individual steps, and their troubleshooting pages collect the pitfalls (unit mix-ups, a beam that stops in the stack, thick-foil energy spread, monitor-reaction choice) that most often trip up a first experiment.

For a single script that runs this whole chain on a real dataset, see examples/stacked_target_analysis.py: it builds the foil stack of a published ^{nat}La(p,x) measurement (Morrell et al., arXiv:1907.04431), transports the beam, flux-averages an evaluated excitation function (a cross section as a function of energy, taken from a nuclear-data library) over each foil, and compares the result against the published cross sections.

CHAPTER

FOUR

USER'S GUIDE

Each topic opens with a short introduction, then three pages: a **How-to Guide** (recipes for specific jobs), **Worked Examples** (complete analyses run start to finish), and **Troubleshooting** (common pitfalls). For a quick tour of the classes first, see the *Quickstart*; for the full details of every method and attribute, see the *API*.

4.1 Spectroscopy

Curie provides two classes for analyzing gamma-ray spectra from high-purity germanium (HPGe) detectors: the `Spectrum` class, which reads spectra and performs peak fitting, and the `Calibration` class, which generates and stores the energy, resolution and efficiency calibrations needed to convert fitted peaks into activities.

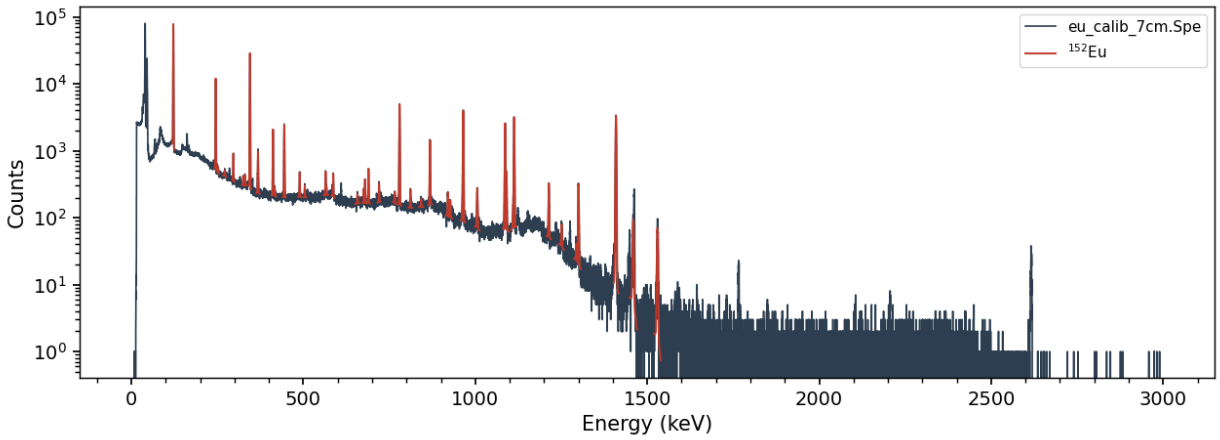


Fig. 1: A ^{152}Eu calibration spectrum with fitted peaks.

Workflow. A typical spectroscopy analysis proceeds in five steps:

1. **Load** a spectrum from disk: `sp = ci.Spectrum('eu_calib_7cm.Spe')`. Ortec .Spe and .Chn and Canberra .CNF and .IEC formats are supported.
2. **Identify** the gamma-decaying isotopes present: `sp.isotopes = ['152EU', '40K']`. Curie retrieves their gamma lines from its decay data.
3. **Calibrate**: apply a saved calibration for this detector and counting geometry: `sp.cb = 'eu_calib.json'`.
4. **Fit** the peaks: `sp.fit_peaks()`, tuned by the `fit_config` options. The result is the `sp.peaks` table of counts, decays and activities per gamma line.
5. **Inspect and export**: `sp.summarize()`, `sp.plot()`, and `sp.saveas()` to .csv, .json, .db or .Chn.

These steps describe routine analysis with an existing calibration. *Creating* that calibration is a separate, usually one-time task: `Calibration.calibrate()` fits the peaks of reference-source spectra itself, so it runs before any manual peak fitting (see the *Spectroscopy Worked Examples*).

See the *Spectroscopy How-to Guide* for each step in detail, the *Spectroscopy Worked Examples* for a complete efficiency calibration with a ^{152}Eu source, and *Spectroscopy Troubleshooting* for the most common pitfalls — most of them involving the energy calibration.

Uses and limitations. These classes are designed for *activation analysis*: quantifying the activities of known gamma-emitting isotopes in a counted sample. The peak-fit and efficiency models (described in *Gamma-ray Peak Fitting* and *Detector Calibration*) are tuned for HPGe data, and are not intended for low-resolution detectors such as NaI. Peak identification is driven by the assigned isotope list — Curie fits the gamma lines it expects from those isotopes, rather than hunting for unidentified peaks. True-coincidence summing — two gammas from the same decay cascade arriving together and counted as one event — is not corrected; it matters mainly for samples counted very close to the detector, so calibrate at a moderate source-to-detector distance or with single-line sources.

4.1.1 Spectroscopy How-to Guide

This page is a task-by-task reference for the `Spectrum` and `Calibration` classes. All examples assume `import curie as ci`. For the underlying models and formulas, see *Gamma-ray Peak Fitting* and *Detector Calibration*; for a complete worked example, see the *Spectroscopy Worked Examples*.

Loading spectra

`Spectrum` reads Ortec .Spe (ASCII) and .Chn (binary), and Canberra .CNF and .IEC files:

```
sp = ci.Spectrum('eu_calib_7cm.Spe')
```

The parsed data are available as attributes: `sp.hist` (the channel histogram), `sp.start_time`, `sp.live_time` and `sp.real_time`. Spectra of the same length can be summed with +, and rebinned with `sp.rebin(N_bins)`.

Identifying peaks

Assign the isotopes present in the sample, using Curie's isotope naming convention (e.g. `'152EU'`, `'Eu-152'`, `'115INm'`):

```
sp.isotopes = ['152EU']
```

Curie generates the list of gamma lines to fit from the decay data of these isotopes. Lines can also be given directly to `fit_peaks` — for example a line from an isotope you have not assigned, or one missing from the decay data — with energies in keV and intensities in percent:

```
sp.fit_peaks(gammas=[{'energy':1460.82, 'intensity':10.66,
                      'unc_intensity':0.17, 'isotope':'40K'}])
```

Fitting peaks

`sp.fit_peaks()` fits all selected lines — grouping overlapping peaks into *multiplets* that are fit together — and returns the peak table; it is also called automatically the first time `sp.peaks`, `sp.plot()`, `sp.summarize()` or `sp.saveas()` is used. The fits are cached: if you change the calibration, isotope list or fit options afterwards, call `sp.fit_peaks()` again to re-fit.

`sp.peaks` is a pandas DataFrame with one row per fitted gamma line. Its central columns are:

Column	Meaning
`isotope`	The emitting isotope
`energy`	Gamma-line energy, in keV
`counts`	Net counts in the peak (background subtracted)
`intensity`	Gamma intensity I_γ — the fraction of decays that emit this line (its branching ratio)
`efficiency`	Peak efficiency at the line energy, from the calibration
`decays`	Decays of the isotope during the count
`decay_rate`	Average decay rate during the count — i.e. the activity, in Bq (`summarize()` prints this value as "activity")
`chi2`	Reduced chi-square (chi-square per degree of freedom) of the multiplet fit

Each quantity is paired with an `unc_` column giving its absolute uncertainty. `decays` and `decay_rate` are corrected for dead time (the fraction of the count during which the detector was unable to record events), efficiency, intensity, and any attenuation/geometry corrections that have been applied.

Configuring the fit

Fit options are set through the `fit_config` dictionary — as an attribute (`sp.fit_config = {...}`, persistent) or as keyword arguments to `sp.fit_peaks(...)`. The complete reference for every option is the `Spectrum.fit_peaks()` API entry; they fall into three groups:

Peak-shape and background options — `bg` selects the background model (`'snip'` default, or `'constant'`/`'linear'`/`'quadratic'`) and `snip_adj` scales the SNIP background parameters; `R`, `alpha` and `step` set the skew and step components of the peak shape, and `skew_fit`/`step_fit` control whether they are fit per peak or held fixed (see *Gamma-ray Peak Fitting* for the functional forms).

Peak-selection options — `xrays`, `E_min`, `I_min` and `dE_511` filter the candidate gamma lines; `SNR_min` drops lines whose predicted signal-to-noise ratio is too low to fit reliably; `ident_idx` controls how overlapping (identical-energy) lines are merged or flagged.

Fit-window and bound options — `pk_width` sets the fitted window around each peak; `multi_max` limits the number of peaks fit together as a multiplet; `A_bound`, `mu_bound` and `sig_bound` scale the bounds on the amplitude, centroid and width parameters.

For example, to include x-ray lines down to 20 keV on a quadratic background:

```
sp.fit_config = {'xrays':True, 'E_min':20.0, 'bg':'quadratic'}
```

Tuning the fit

The defaults are chosen for typical activation spectra; these are the adjustments that real analyses most often need. In every case, start from the summary line printed by the fit (next section), which reports how many lines each filter dropped — it identifies which parameter to change before you change it.

Short or weak counts — a low-activity sample or a short count leaves marginal peaks below the `SNR_min` threshold (default 4). Lower it to admit them (`SNR_min=2.0`): the reported uncertainty grows with the weakness of the peak, so admitting a marginal line does not overstate its precision — it simply carries a proportionally large error bar.

Busy, many-isotope spectra — a foil with dozens of activation products produces hundreds of candidate lines and crowded multiplets. Raise `I_min` (e.g. `0.5` for only the strongest branches) and `SNR_min` (e.g. `6`) to fit only the quantifiable lines, and raise `multi_max` if wide multiplets are being truncated. When two isotopes share a line energy, the intensity split between them is genuinely ambiguous — the fit warns that the intensities may be misattributed, and those intensities should be checked individually. `ident_idx` controls how close (in channels) two lines must be to be treated as one.

Low-energy work — x-ray and low-gamma lines need `xrays=True` and a lower `E_min`; the efficiency calibration should then use the 7-parameter model, which the calibration selects automatically when its spectra include x-rays (see *Detector Calibration*).

Distorted peak shapes — strong low-energy tailing or stepped backgrounds are shape problems, not selection problems; see the *Spectroscopy Troubleshooting* entries on bad fits (`skew_fit`/`step_fit`, `bg`, bounds).

Reading the fit log and diagnostics

Every fit prints a summary of its results and selections to the console, in messages of the form `[LEVEL] Class.method: message`. After any `fit_peaks()` call, the first thing to check is its `INFO` summary:

```
[INFO] Spectrum(eu_calib_7cm.Spe).fit_peaks: fit 43 peaks in 33 multiplets
from 1 isotopes; dropped 127 candidates (5 SNR<4.0, 122 intensity<0.05%);
2 multiplets with chi2/dof>10; 3 peaks with parameters at fit bounds
(see sp.diagnostics)
```

Nothing is silently discarded: every candidate line that was not fit is in one of those counts, with its filter named. `WARNING` messages mark things that can affect results — a failed multiplet, identical gammas fit with shared bounds, a calibration evaluated beyond its fitted range. The per-item detail behind each summary count is logged at `DEBUG`:

```
ci.set_log_level('DEBUG')   # show every dropped line with its reason
ci.quiet_warnings()         # errors only
ci.log_to('curie.log')      # write messages to a file (overwrites it;
                            # mode='append' accumulates runs instead)
```

The same accounting is available as a table. `sp.diagnostics` has one row per *attempted* multiplet — including failed ones — with the reduced chi-square, the model, the uncertainty scale factor, and a `flags` column drawn from a fixed vocabulary (`at_bound:<param>`, `chi2_high`, `fit_failed`); the `message` column holds the full text of everything reported about that fit. To select the fits with a nonempty flag:

```
d = sp.diagnostics
print(d[d['flags'] != ''])
```

The fitted parameter sets themselves are in `sp.fits` (one entry per multiplet), and `sp.plot()` marks any failed multiplet by crosshatching the unfitted counts in red.

`Calibration` and `DecayChain` follow the same pattern: `cb.diagnostics` and `dc.diagnostics` summarize their fits, and the calibration point tables `cb.engcal_data`, `cb.rescal_data` and `cb.effcal_data` show every measured point with a `used` flag and the `reason` any point was rejected — rejected points also stay visible as grey open markers on the calibration plots. The conventions — message levels, the flags vocabulary, and how uncertainty scale factors are applied — are defined in *Fit Reporting Conventions*.

Calibrating

A `Calibration` holds three calibrations (see *Detector Calibration* for the functional forms and fitting procedure):

- **energy** — `cb.engcal`, channel to keV, used by `cb.eng()`;
- **resolution** — `cb.rescal`, peak width vs. channel, used by `cb.res()`;
- **efficiency** — `cb.effcal`, with the fitted-parameter covariance matrix (uncertainties and their correlations) in `cb.unc_effcal`, used by `cb.eff()` and `cb.unc_eff()`.

To fit all three from spectra of reference sources, give the source activities at a reference date:

```
sp = ci.Spectrum('eu_calib_7cm.Spe')
sp.isotopes = ['152EU']

cb = ci.Calibration()
cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.7E4,
                             'ref_date':'01/01/2009 12:00:00'}])
cb.plot()
```

`sources` can also be a .csv, .json or .db file with columns 'isotope', 'A0' and 'ref_date' (an 'unc_A0' column is used if present). The calibration is applied to the input spectra, and can be saved and re-used across spectra counted on the same detector and geometry:

```
cb.saveas('eu_calib.json')

sp2 = ci.Spectrum('sample_7cm.Spe')
sp2.cb = 'eu_calib.json'
```

The energy calibration can also be set directly (`sp.cb.engcal = [0.3, 0.184]`) — after changing it, re-fit with `sp.fit_peaks()`.

Choosing the calibration models

Like the peak fit, `calibrate()` is configured through a `fit_config` dictionary (`cb.fit_config = {...}` or keyword arguments, which merge and persist). Each calibration has a selectable model — by default the form of the current calibration is kept, so nothing changes unless you ask:

Key	Models
`engcal_mod`	`'linear'`, `'quadratic'`, `'cubic'`
`rescal_mod`	`'sqrt'`, `'linear'`, `'sqrt_quad'`
`effcal_mod`	`'vidmar'` (5/7-parameter automatic, the default), `'vidmar-5'`, `'vidmar-7'`, `'loglog'` (order 4) or `'loglog-2'` through `'loglog-8'`

For the functional forms and when each is appropriate, see *Detector Calibration*. The practical guidance for the efficiency model: **stay with Vidmar unless it visibly misfits your detector.** The semi-empirical form extrapolates smoothly beyond the calibration points; the log-log polynomial often fits tighter *within* them but diverges rapidly outside — Curie stores the fitted energy range and warns the first time an efficiency is evaluated beyond it:

```
cb.calibrate([sp], sources=srcs, effcal_model='loglog-5')
```

The point-selection thresholds are also configurable: `engcal_max_error`/`rescal_max_error`/`effcal_max_error` set the pre-fit uncertainty cuts (defaults 25%/33%/33%), and `outlier_sigma` (default 3.16) sets the post-fit residual clip. The fitted models are saved in the calibration .json and restored on load; files from older versions of Curie load unchanged.

Extending the efficiency calibration with known points

`calibrate(..., eff_points=...)` appends user-supplied efficiency points to the measured ones before the efficiency fit — a DataFrame (or list of dicts) with columns `energy` (keV), `efficiency`, `unc_efficiency` and optionally `isotope`. Two situations where this is the right tool:

Covering an energy range your sources don't reach — e.g. your analysis needs efficiencies at 2-3 MeV but the calibration sources stop at 1.4 MeV. Add points from a source measured on another occasion, or from a detector simulation, and the fit (and its stored energy range) extends to cover them:

```
ep = [{'energy':2000.0, 'efficiency':3.1E-3, 'unc_efficiency':3E-4,
       'isotope':'56CO'}]
cb.calibrate([sp], sources=srcs, eff_points=ep)
```

Merging counting geometries — point-source efficiencies measured at another distance can anchor the curve shape: scale their efficiencies by the far-field solid-angle ratio $(R_1/R_2)^2$ to carry them from distance R_1 to the calibration distance R_2 (the first parameter of the Vidmar model is exactly this solid-angle scale). The user-supplied points are treated as independent measurements (no shared intensity or source-activity uncertainty), reported in the console log, and carried in `cb.effcal_data` with the rest.

Adjusting a drifted energy calibration

If the stored energy calibration is slightly off (peaks visibly displaced from their lines), `sp.auto_calibrate()` re-fits it using a forward-fit of the assigned isotopes' lines to the spectrum. It only converges from a starting point within about half a percent; for larger drifts, give it a guess or a list of (channel, energy) anchor points:

```
sp.auto_calibrate(guess=[0.3, 0.1835])
sp.auto_calibrate(peaks=[[664, 121.8]])
```

Correcting for attenuation and geometry

Two multiplicative corrections can be applied to a spectrum before fitting; both modify the `decays/decay_rate` columns of the peak table.

`sp.attenuation_correction(compounds, x=...)` computes the energy-dependent self-attenuation of the sample: the first entry is the sample itself (its correction accounts for emission throughout the thickness), and subsequent entries are absorbing layers between sample and detector. Thicknesses `x` are in cm (or give areal densities `ad` in g/cm2 — note this is g/cm2 here, not the mg/cm2 that `Stack` uses):

```
sp.attenuation_correction(['Fe', ci.Compound('H2O', density=1.0)],
                          x=[0.1, 0.5])
```

`sp.geometry_correction(...)` computes the solid-angle correction for a sample counted close to the detector, relative to the point-source geometry of the efficiency calibration, by Monte Carlo integration. The dimensions are unitless but must all be in the *same* unit (e.g. all cm), and `sample_size` is the radius for `shape='circle'` (the default), the side length for `'square'`, or an (x, y) pair for `'rectangle'`:

```
sp.geometry_correction(distance=4, r_det=5, thickness=0.1,
                       sample_size=2, shape='square')
```

Plotting, summarizing and saving

`sp.plot()` draws the spectrum with its fits (`fit=False` for the raw histogram, `xcalib=False` for raw analog-to-digital converter (ADC) channels on the x-axis). `sp.summarize()` prints the counts, decays and activity of each fitted line. `sp.saveas()` writes the peak table to .csv, .json or .db, the spectrum itself to .Spe or .Chn (format conversion), or the plot to an image format:

```
sp.saveas('peaks.csv')          # peak table
sp.saveas('eu_calib_7cm.Chn')   # converted spectrum
sp.saveas('spectrum.png')       # plot
```

Calibrations plot with `cb.plot()` (all three curves) or individually (`cb.plot_engcal()`, `cb.plot_rescal()`, `cb.plot_effcal()`).

4.1.2 Spectroscopy Worked Examples

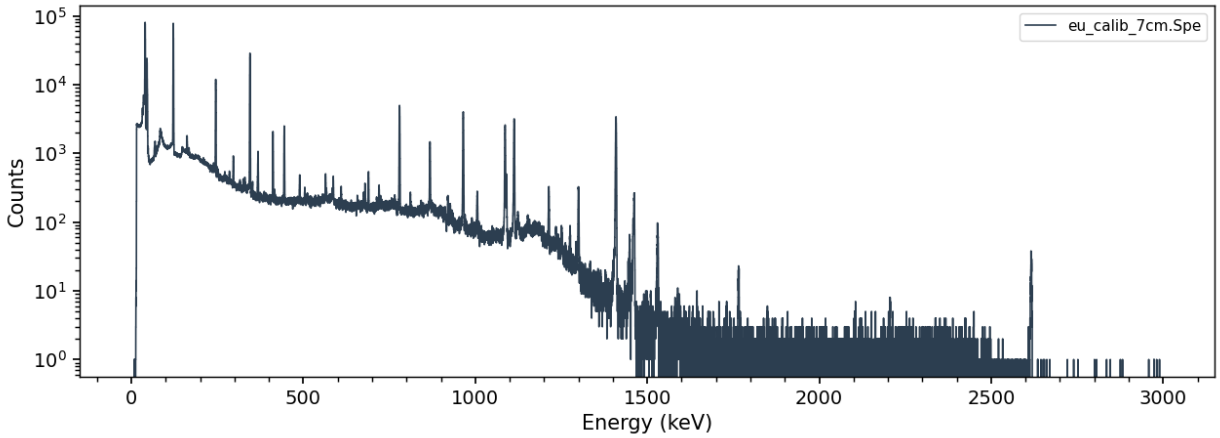
This page walks through a complete efficiency calibration using `eu_calib_7cm.Spe` — a spectrum of a ^{152}Eu reference source counted 7 cm from an HPGe detector, available in the examples folder of the Curie repository. The source had a certified activity of 37.0 kBq (1.0 uCi) on 01/01/2009.

Loading and inspecting the spectrum

Load the spectrum and plot the raw histogram:

```
import curie as ci

sp = ci.Spectrum('eu_calib_7cm.Spe')
sp.plot(fit=False)
```



The energy calibration stored in the .Spe file header was read automatically:

```
>>> print(sp.cb.engcal)
[3.3973e-01 1.8297e-01 5.5683e-09]
```

Fitting the peaks

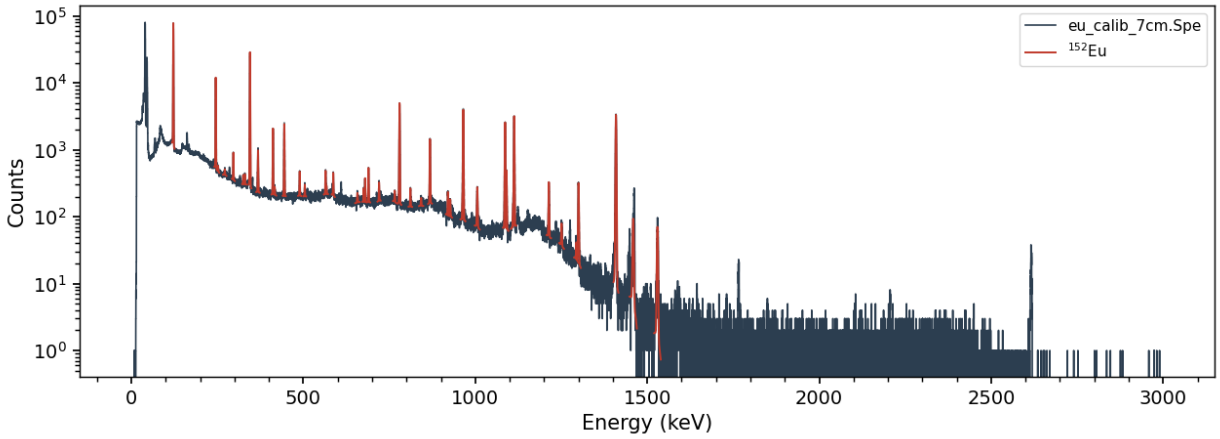
Tell Curie which isotopes are present, and it will fit every ^{152}Eu line that passes the selection criteria (see *Spectroscopy How-to Guide*):

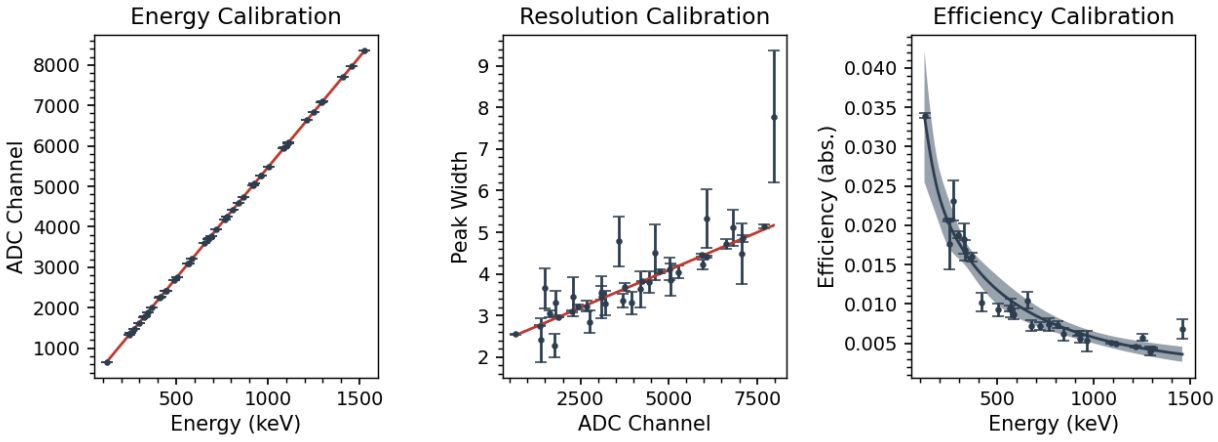
```
sp.isotopes = ['152EU']
sp.plot()
```

Fitting the efficiency calibration

The peak counts are correct at this point, but the decay rates are not: they are computed with the built-in default efficiency curve, not one measured for this detector. Calibrating requires the reference activity and date of the source:

```
cb = ci.Calibration()
cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.7E4,
                             'ref_date':'01/01/2009 12:00:00'}])
cb.plot()
```


eu_calib_7cm.Spe
^{152}Eu
Counts
Energy (keV)


Energy Calibration
ADC Channel
Energy (keV)
Resolution Calibration
Peak Width
ADC Channel
Efficiency Calibration
Efficiency (abs.)
Energy (keV)

`calibrate()` refit the energy, resolution and efficiency calibrations from the fitted peaks (the procedure and functional forms are described in *Detector Calibration*), and applied them to the spectrum. The fitted efficiency parameters (and their uncertainties) are stored on the calibration:

```
>>> print(cb.effcal)
[5.02300989e-02 1.00000000e+02 2.82394962e+00 2.45721823e+00
 2.91455256e-01]
```

Checking the results

With the calibration applied, the peak table now reports consistent decay rates across lines spanning 122 to 1408 keV:

```
>>> print(sp.peaks[['isotope','energy','counts','unc_counts','intensity',
...                 'efficiency','decay_rate','unc_decay_rate','chi2']].head(8))
  isotope    energy    counts  unc_counts  intensity  efficiency  decay_rate  unc_decay_
↪rate   chi2
0   152EU  121.7817  498786.0      3118.0   0.285300     0.03385     22564.0         ␣
↪5570.0  20.18
1   152EU  244.6974   80193.0       403.0   0.075500     0.02042     22725.0         ␣
↪3445.0   1.96
2   152EU  251.6330     610.0       112.0   0.000670     0.02002     19864.0         ␣
↪4734.0   1.96
3   152EU  271.0800     853.0        93.0   0.000715     0.01903     27381.0         ␣
↪4856.0   0.93
4   152EU  295.9387    4258.0        98.0   0.004400     0.01792     23594.0         ␣
↪2803.0   0.90
5   152EU  324.8300     642.0        64.0   0.000681     0.01676     24556.0         ␣
↪3486.0   0.92
6   152EU  329.4100    1055.0        78.0   0.001213     0.01659     22900.0         ␣
↪2792.0   0.92
7   152EU  344.2785  213990.0       669.0   0.265900     0.01605     21911.0         ␣
↪2030.0   2.19
```

The `decay_rate` column — the activity of the source during the count — clusters around 22.6 kBq for every line: the 37.0 kBq source decayed for 9.7 years — a bit under one 13.5 y half-life — between the reference date and the count. (The large `chi2` on the very intense 122 keV peak is expected for a peak with half a million counts — see the note on high-statistics peaks in *Spectroscopy Troubleshooting*.) `sp.summarize()` prints the same information line by line:

```
>>> sp.summarize()
...
152EU - 244.6974 keV (I = 7.55%)
--------------------------------
counts: 80193 +/- 403
decays: 5.577e+07 +/- 8.454e+06
activity (Bq): 2.273e+04 +/- 3.445e+03
activity (uCi): 6.142e-01 +/- 9.311e-02
chi2/dof: 1.958
...
```

Saving and re-using the calibration

Save the calibration, and apply it to any other spectrum counted in the same geometry — here `your_sample.Spe` and ^{64}Cu stand in for a later measurement of your own (substitute your own spectrum):

```
cb.saveas('eu_calib.json')

sp2 = ci.Spectrum('your_sample.Spe')
sp2.cb = 'eu_calib.json'
sp2.isotopes = ['64CU']
sp2.summarize()
```

Note that the saved file replaces the new spectrum's energy calibration too — if peaks shift or vanish after this step, see *Spectroscopy Troubleshooting*.

The peak data can be exported for further analysis — for example to fit a decay curve with `DecayChain.get_counts()` (see *Isotopes & Decay Chains*):

```
sp.saveas('eu_peaks.csv')
```

4.1.3 Spectroscopy Troubleshooting

The most common spectroscopy problems trace back to one thing: Curie fits the peaks it *expects* from the assigned isotopes and the current calibration, rather than searching the spectrum for unknown peaks. When the expectation is wrong — usually the energy calibration — peaks are missed or misfit. This page collects the failure modes we see most often, with their symptoms and fixes.

Whatever the symptom, check the fit's output first: the `fit_peaks` summary line accounts for every candidate line (fit, or dropped with the responsible filter named), and `sp.diagnostics` records each attempted fit with flags for the problematic ones (see *Spectroscopy How-to Guide*). Most of the entries below appear as a line item in one or the other.

No peaks are fit, and the spectrum looks uncalibrated

Symptom: `sp.peaks` comes back empty (or nearly so) even though the spectrum plainly shows peaks, and known lines are nowhere near their energies in `sp.plot()`.

Cause: the energy calibration stored in the spectrum file is wrong or missing. Curie predicts the channel of every gamma line from the energy calibration and evaluates its expected signal-to-noise ratio *at that channel*; if the calibration points at featureless background, every line fails the `SNR_min` test and is dropped without an error.

Fix: plot the raw histogram against channel number and identify one or two known lines by eye:

```
sp.plot(fit=False, xcalib=False)
```

then anchor the calibration with (channel, energy) pairs — here using the 121.8 keV line of ^{152}Eu seen at channel 664:

```
sp.isotopes = ['152EU']
sp.auto_calibrate(peaks=[[664, 121.8]])
```

or set it directly if the coefficients are known:

```
sp.cb.engcal = [0.3, 0.184]
sp.fit_peaks()
```

Peaks fit correctly — until a saved calibration is applied

Symptom: the spectrum fits well on its own, but after assigning a saved calibration (`sp.cb = 'eu_calib.json'`) the peaks shift or disappear. This one is trickier, because applying the calibration is exactly what you are supposed to do.

Cause: assigning a calibration replaces *all three* calibrations — energy, resolution and efficiency. The efficiency calibration is usually what you want from the file (it is a property of the detector and counting geometry), but the energy calibration it carries describes the electronics *on the day the calibration was measured*. Drift in the amplifier gain since then (or an intervening, forgotten recalibration of the ADC) means the file's energy calibration no longer matches this spectrum, while the spectrum's own header calibration was correct all along.

Fix: apply the saved calibration for its efficiency, but keep the spectrum's own energy calibration:

```
sp = ci.Spectrum('sample_7cm.Spe')
engcal = sp.cb.engcal          # calibration from the file header
sp.cb = 'eu_calib.json'        # detector efficiency (overwrites engcal)
sp.cb.engcal = engcal          # restore this spectrum's energy calibration
sp.fit_peaks()
```

If the header calibration is also imperfect, follow with `sp.auto_calibrate()`, which makes small corrections using the assigned isotopes' lines.

An expected peak is missing from the results

Symptom: a line you can see in the spectrum — and that you know the isotope emits — does not appear in `sp.peaks`. Everything else fits fine.

Cause: one of the peak-selection filters excluded it before fitting. The exclusion is reported: the summary line counts the dropped lines per filter, and `ci.set_log_level('DEBUG')` names each one individually:

```
[INFO] Spectrum(sample.Spe).fit_peaks: fit 12 peaks in 9 multiplets from
1 isotopes; dropped 31 candidates (2 SNR<4.0, 29 intensity<0.05%)
```

The defaults are chosen for typical activation spectra and will drop some legitimate lines:

- `E_min` (default 75 keV) excludes all lower-energy lines — the 59.5 keV line of ^{241}Am, for example, never fits at the default.
- `I_min` (default 0.05%) excludes weak branches.
- `dE_511` excludes lines within a few keV of the 511 keV annihilation peak.
- `SNR_min` excludes lines *predicted* to be too weak to fit — and the prediction uses the current efficiency calibration, so a badly wrong efficiency curve can veto a clearly visible peak.
- X-rays are excluded entirely unless `xrays=True`.

Fix: relax the relevant filter:

```
sp.fit_peaks(E_min=40.0, I_min=0.01)   # admit lower-energy / weaker gammas
sp.fit_peaks(xrays=True, E_min=20.0)   # or: also include x-ray lines
sp.fit_peaks(SNR_min=2.0)              # or: admit marginal peaks
```

Each line is an independent alternative, not a sequence. If a *clearly visible* peak is being vetoed by `SNR_min`, the real problem is usually an efficiency calibration that badly underpredicts the peak — recalibrate (see the *Spectroscopy Worked Examples*) rather than lowering the threshold. If a line is missing because the isotope's decay data doesn't include it, add it manually with the `gammas` argument (see *Spectroscopy How-to Guide*).

Fits succeed, but they are bad fits

Symptom: fitted curves that visibly miss the data, large `chi2` values in the peak table, or `peak fit failed` warnings for particular multiplets (groups of overlapping peaks that are fit together). Failed multiplets are crosshatched in red on `sp.plot()`, and every high-chi2 or at-bound fit is flagged in `sp.diagnostics`.

Cause and fix, by situation:

- **The peak sits on structure** — the peak sits on a broad spectral feature rather than flat background: a backscatter peak or Compton edge (both produced by scattered gamma rays), or the shoulder of a much larger neighboring peak. The default SNIP background assumes a smoothly varying continuum and will not follow sharp features. Refit with a polynomial background, which is fit jointly with the peaks: `sp.fit_peaks(bg='quadratic')`.
- **The fit window is too wide or too crowded.** Wide windows (`pk_width`, default 7.5 peak widths) can pull neighboring structure into the fit, and crowded regions can exceed the multiplet limit. Reduce `pk_width`, or raise `multi_max` so overlapping peaks are fit together rather than truncated.
- **The peak shape doesn't match** — strong low-energy tailing on intense peaks (degraded detectors, high count rates) that the default fixed skew parameters underestimate. Let the skew be fit per peak with `skew_fit=True` (and `step_fit=True` for a per-peak step), at the cost of more free parameters per peak.
- **The starting point is too constrained.** Amplitudes, centroids and widths are bounded around their predicted values; if the prediction is poor (e.g. a marginal energy calibration), loosen the bounds with the multipliers `A_bound`, `mu_bound` and `sig_bound`.

A `chi2` well above one on a very high-statistics peak — values of ten or more are common on peaks with hundreds of thousands of counts, like the 122 keV peak in the *Spectroscopy Worked Examples* — is not by itself a bad fit: with that many counts, even percent-level imperfections of the peak model are statistically resolvable. Curie accounts for this by inflating the fit uncertainties by $\sqrt{\chi^2_\nu}$ (see *Gamma-ray Peak Fitting*).

4.2 Isotopes & Decay Chains

Curie provides two classes for working with radioactive decay. The `Isotope` class looks up nuclear data for a single isotope: masses, abundances, half-lives, the energies and intensities of its emissions (gammas, betas, alphas, conversion electrons), and dose rates. The `DecayChain` class does the bookkeeping of decay itself: starting from a parent isotope it builds the full chain of decay products, computes every member's activity as a function of time — during production as well as decay — and can fit production rates or initial activities to measured data.

Workflow. `Isotope` needs no workflow — construct it and read off the data (see *Isotope & Decay Chain How-to Guide*). Decay-chain problems come in two directions:

- **Forward** — you know (or assume) how much of an isotope was made, and want activities at later times: build the chain with an initial activity `A0` or a production-rate history `R`, then call `dc.activity()`, `dc.decays()` or `dc.plot()`.
- **Inverse** — you measured decays (typically peaks in gamma-ray spectra) and want the production rate or activity that explains them: load the measurements with `dc.get_counts()` and fit with `dc.fit_R()` (for production) or `dc.fit_A0()` (for decay-only).

Both directions share one time convention: **t = 0 is the end of production** (the end of bombardment in an activation experiment), and all times are in the chain's `units`.

See the *Isotope & Decay Chain How-to Guide* for each task in detail, the *Decay Chain Worked Examples* for both directions worked on real examples, and *Isotope & Decay Chain Troubleshooting* for the common pitfalls.

Uses and limitations. `Isotope` serves quick lookups (a half-life in sensible units, a table of gamma lines above some intensity) and provides the decay data that `Spectrum`, `Calibration` and `DecayChain` use internally. The decay data are compiled from ENSDF (via the IAEA LiveChart interface), with NUBASE2020/AME2020 closures and ENDF/B-VIII.1 fission yields.

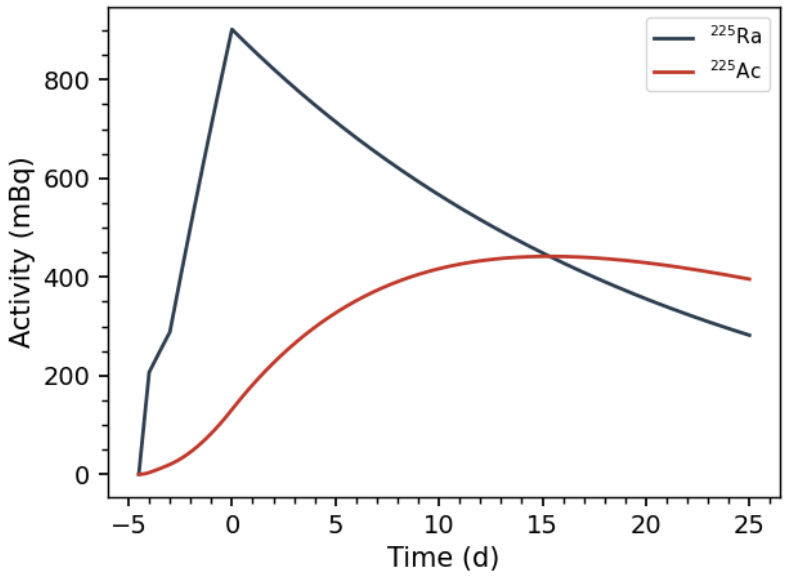


Fig. 2: ^{225}Ra during and after production: the daughter ^{225}Ac grows in as the parent decays.

`DecayChain` solves the Bateman equations exactly (see *Radioactive Decay Chains*), including chains with branching and same-half-life members, for one parent isotope at a time with a piecewise-constant production history. Production physics is *not* computed — the production rate is an input, which you can fit from measured counts, or estimate yourself from cross sections and particle fluxes (the *Reactions* pages cover the cross-section side).

4.2.1 Isotope & Decay Chain How-to Guide

This page is a task-by-task reference for the `Isotope` and `DecayChain` classes. All examples assume `import curie as ci`. The underlying equations are in *Radioactive Decay Chains*.

Looking up an isotope

Isotope names have three parts: a mass number, an element symbol, and an optional state suffix. Two written forms are accepted:

```
ip = ci.Isotope('60CO')     # compact:    mass + SYMBOL (all caps)
ip = ci.Isotope('Co-60')    # hyphenated: Symbol-mass (any case)
```

In the **compact** form the element symbol must be entirely upper-case — the parser reads the capital letters as the symbol, so `'60Co'` or `'60co'` raise errors rather than finding cobalt. The **hyphenated** form is case-insensitive (`'co-60'` works). When in doubt, hyphenate.

The state suffix, always lower-case, selects the nuclear state: `g` for the ground state (assumed when no suffix is given), `m` or `m1` for the first isomer (metastable excited state), `m2` for the second, and so on:

```
ip = ci.Isotope('115INm')   # 115In first isomer
ip = ci.Isotope('Hf-178m2') # 178Hf second isomer (the 31 y state)
```

Whatever form you write, Curie normalizes it to one canonical name — `ip.name`, in compact form with an explicit state — and that canonical form is what appears everywhere else (`DecayChain.isotopes`, peak tables, count data):

You write	ip.name	Meaning
'60CO'	60COg	^{60}Co ground state
'co-60'	60COg	the same
'115INm'	115INm1	^{115}In, first isomer
'Eu-152m2'	152EUm2	^{152}Eu, second isomer
'99TCg'	99TCg	^{99}Tc ground state, explicit
'n'	1ng	a free neutron

Structure properties are attributes: `ip.mass` (amu), `ip.abundance` (percent), `ip.E_level` (MeV), `ip.J_pi` (spin-parity), `ip.Delta` (mass excess, MeV), `ip.TeX` (a LaTeX-formatted name for plot labels).

Half-lives and decay constants

`half_life()` and `decay_const()` take a units argument and can return uncertainties:

```
ip = ci.Isotope('60CO')
print(ip.half_life('y'))              # 5.271...
print(ip.half_life('y', unc=True))    # (value, uncertainty)
```

`ip.optimum_units()` picks the most readable unit for the half-life — convenient when looping over isotopes spanning seconds to gigayears:

```
print(ip.half_life(ip.optimum_units()), ip.optimum_units())
```

Emission tables

The decay radiations are returned as pandas DataFrames, with energies in keV and intensities in percent (per decay). Each accepts an intensity filter (`I_lim`) and an energy filter (`E_lim` — or `Endpoint_lim` for the beta spectra):

```
ip = ci.Isotope('Co-60')
print(ip.gammas(I_lim=1.0))           # gamma lines above 1%
print(ip.electrons(CE_only=True))     # conversion electrons
print(ip.beta_minus())                # beta- endpoint & mean energies
print(ip.beta_plus())                 # beta+ (positron) emissions
print(ip.alphas(I_lim=1.0))
```

`gammas()` also takes `xrays=True` to include fluorescence x-rays, and `dE_511` (a width, in keV) to drop lines within that many keV of the 511 keV annihilation peak — the same filters `Spectrum` applies when fitting peaks.

Fission yields

For fissioning isotopes, independent fission-product yields (the yield of each product as directly formed in fission, before any subsequent decay) are available: `get_SFY()` for spontaneous fission, and `get_NFY(E)` for neutron-induced fission. Unlike everywhere else in Curie, the incident energy `E` here is in *eV*, on the ENDF grid (0.0253 for thermal, 5E5, 2E6 or 14E6):

```
ip = ci.Isotope('235U')
print(ip.get_NFY(E=0.0253))    # thermal fission yields
```

Dose rate

`dose_rate()` estimates the dose rate from a point source of the isotope, with no attenuation between source and receptor, for a given activity (Bq) and distance (cm):

```
ip = ci.Isotope('Co-60')
print(ip.dose_rate(activity=3.7E10, units='R/hr'))   # 1 Ci at 30 cm
print(ip.dose_rate(activity=3.7E10, distance=100.0, units='uSv/hr'))
```

The result is a dictionary with the contribution of each particle type. This model applies no attenuation, not even from air, so the charged-particle entries are near-contact estimates; at any realistic distance, alphas and betas are stopped by the air in between. Use the `gammas` entry for distance-dose estimates, and treat `total` accordingly.

Building a decay chain

A `DecayChain` needs two things at minimum: the parent isotope, and the time units that every number in the chain — production histories, count intervals, the times passed to `activity()` — will be interpreted in:

```
dc = ci.DecayChain('225RA', units='d')
print(dc.isotopes)
```

Valid units are `'ns'`, `'us'`, `'ms'`, `'s'`, `'m'`, `'h'`, `'d'`, `'y'`, `'ky'`, `'My'`, `'Gy'` (with `'sec'`, `'min'`, `'hr'`, `'yr'` accepted as synonyms).

At construction, Curie follows every decay mode in the decay data — alpha, beta, electron capture, isomeric transition, even spontaneous fission — from the parent down to stable nuclei, and `dc.isotopes` lists what it found, in canonical names with explicit `g`/`m` suffixes. Always worth printing when a chain misbehaves. The parent can itself be an isomer, in which case the chain proceeds through its decay modes:

```
>>> dc = ci.DecayChain('178HFm2', units='y')
>>> print(dc.isotopes)
['178HFm2', '178HFm1', '178HFg']
```

The chain does *no* production physics: what gets made, and how fast, is an input — the subject of the next section.

Production and initial activity

Beyond the parent and units, the chain needs a starting condition — where the atoms come from. There are two, which correspond to the two workflow directions on the *Isotopes & Decay Chains* page:

Initial activity `A0`, for a sample that already exists and simply decays. A float is the parent's activity in Bq at t = 0:

```
dc = ci.DecayChain('152EU', A0=3.7E4, units='y')
```

A dict sets several members at once — for example a sample that already contains some of the daughter:

```
dc = ci.DecayChain('99MO', A0={'99MO': 3.7E4, '99TCm': 1.0E3}, units='h')
```

Production rate `R`, for a sample being made (an irradiation). `R` is the number of atoms produced per second, as a piecewise-constant history: each row is `[rate, time]`, where `time` is the *end* of that rate's interval:

```
# 9/s until t=0.5 d, then 2/s until 1.5 d, then 5/s until 4.5 d
dc = ci.DecayChain('225RA', R=[[9, 0.5], [2, 1.5], [5, 4.5]], units='d')
```

Because the rate is atoms per second, activities come out in Bq — a chain member at saturation has an activity (decays/s) equal to its production rate (atoms/s). With `timestamp=False` the times are read as interval *durations* instead (`[[9, 0.5], [2, 1.0], [5, 3.0]]` is the same history).

Note the two clocks: the times inside `R` run over the irradiation itself, from 0 (beam on) to the last entry (beam off). Once the chain is built, its clock re-zeros — **t = 0 is the end of production**, and production history appears at negative times (as in the figure on the *Isotopes & Decay Chains* page).

Like `A0`, `R` accepts a dict keyed by isotope, for daughters that are also produced directly by the reaction (a common situation — e.g. ^{99m}Tc made directly alongside its ^{99}Mo parent):

```
dc = ci.DecayChain('99MO', R={'99MO': [[10, 24]], '99TCm': [[2, 24]]}, units='h')
```

All isotopes in the dict must share the same time grid. `R` can also be a DataFrame or a .csv/.json/.db file with columns 'isotope', 'R' and 'time'. `dc.R_avg` gives the time-averaged rate for each produced isotope — the quantity usually quoted from an irradiation.

Activities and decays

With the chain defined, any member's activity (Bq) or number of decays follows, at times measured from the end of production:

```
print(dc.activity('225AC', time=10))            # chain units
print(dc.decays('225AC', t_start=5, t_stop=5.1))
dc.plot()                                        # all members vs time
dc.plot(max_plot=5)                              # only the first 5 chain members
```

Feeding in measured decays

For the inverse problem, the chain needs measured decays. A note on words: in a decay chain, "counts" means the number of *nuclear decays* in a counting interval — what the spectrum's peak table calls `decays`, not its `counts` (net peak areas). The most direct route is from fitted spectra: `get_counts()` reads the `decays` column of each spectrum's peak table, keeps the isotopes that are in this chain, and converts the count timestamps to decay times using the end-of-bombardment time you supply:

```
dc.get_counts([sp1, sp2], EoB='06/12/2026 13:45:00')
```

`EoB` (a datetime or a string in `'%m/%d/%Y %H:%M:%S'` format) defines t = 0; each spectrum's start time and duration are taken from its file header. Peak data saved earlier with `sp.saveas()` works too (`peak_data='peaks.csv'` with filenames in `spectra`). Counts can also be entered by hand as `[t_start, t_stop, decays, unc_decays]` in chain units:

```
dc.counts = {'225AC': [[5.0, 5.1, 6E5, 2E4],
                       [6.0, 6.1, 7E5, 3E4]]}
```

Fitting production rates or activities

`fit_R()` scales the production-rate history to best match the measured decays; `fit_A0()` does the same for the initial activity. In both cases the *shape* of what you provided is preserved — the fit adjusts one overall multiplier per produced isotope, and returns the isotopes, the fitted quantities (the time-averaged rate for `fit_R`, the initial activity for `fit_A0`) and their covariance matrix, whose diagonal holds each fitted quantity's variance — take the square root of a diagonal entry for that quantity's one-sigma uncertainty (as in the worked example):

```
dc = ci.DecayChain('225RA', R=[[9, 0.5], [2, 1.5], [5, 4.5]], units='d')
dc.counts = {'225AC': [[5.0, 5.1, 6E5, 2E4]]}
isotopes, R_fit, cov = dc.fit_R()
```

After the fit, `dc.R`, `dc.R_avg` and all activities reflect the fitted scale, and `dc.plot()` shows the measurements against the fitted curves — with a shaded band giving the one-sigma uncertainty of the fit, and any measurement the fit did not

use drawn as a grey open marker rather than hidden. When the counts came from `get_counts()`, the fit accounts for shared (correlated) uncertainties between measurements — the gamma intensities and the efficiency calibration — as described in *Radioactive Decay Chains*.

Filtering the count data

Fit options are set through `dc.fit_config` — as an attribute or as keyword arguments to `fit_R()/fit_A0()`, which merge and persist for later fits. Two filters are on by default: `max_error` (default 0.4) excludes counts with relative errors above 40%, and `min_counts` (default 1) excludes near-empty measurements. Four more are off until you set them:

```
dc.fit_R(max_chi2=25,                              # drop counts whose PEAK fit was poor
         exclude_lines=[('196TL', 425.7)],         # drop a specific gamma line
         time_range={'196AUm2': (None, 60.0)}, # use an isotope's counts only in a window
         unc_R_floor=0.05)                         # returned unc_R at least 5% of R
```

- `max_chi2` acts on the reduced chi-square of the *peak fit* each count came from (carried into `dc.counts` by `get_counts()`) — it excludes measurements whose underlying peak fits were unreliable.
- `exclude_lines` takes isotope names or (isotope, energy) pairs; the energy matches the isotope's nearest line within 0.5 keV, and an entry that matches nothing excludes nothing and warns (typo safety).
- `time_range` limits an isotope's counts to a decay-time window in chain units — e.g. only early counts, before a longer-lived contaminant's line grows in.
- `unc_R_floor` is not a filter but a floor: the returned production-rate uncertainty is raised to at least this fraction of the rate.

Every filter reports its drops in the fit's summary line, each excluded count is named at `DEBUG`, and the accounting lands in `dc.diagnostics` (see the reporting conventions in *Fit Reporting Conventions*). A starting estimate can also be supplied in production-rate units with `fit_R(p0={'134CE': 2.3E8})` when the automatic starting estimate is poor.

4.2.2 Decay Chain Worked Examples

This page works a decay-chain problem in each direction: first *forward* — modeling the production and decay of ^{225}Ra, a generator of the medical isotope ^{225}Ac — and then *inverse* — fitting the activity of the ^{152}Eu source measured in the *Spectroscopy Worked Examples*, and checking it against the source's certificate.

Forward: producing 225Ra

Suppose a target is irradiated for 4.5 days, producing ^{225}Ra (half-life 14.9 d), but the beam current varied: the production rate was 9 atoms/s for the first half day, dropped to 2/s for the next day, and recovered to 5/s for the remaining three days. This piecewise-constant history is exactly what the `R` argument describes — each row is `[rate, time]` with `time` the end of that interval:

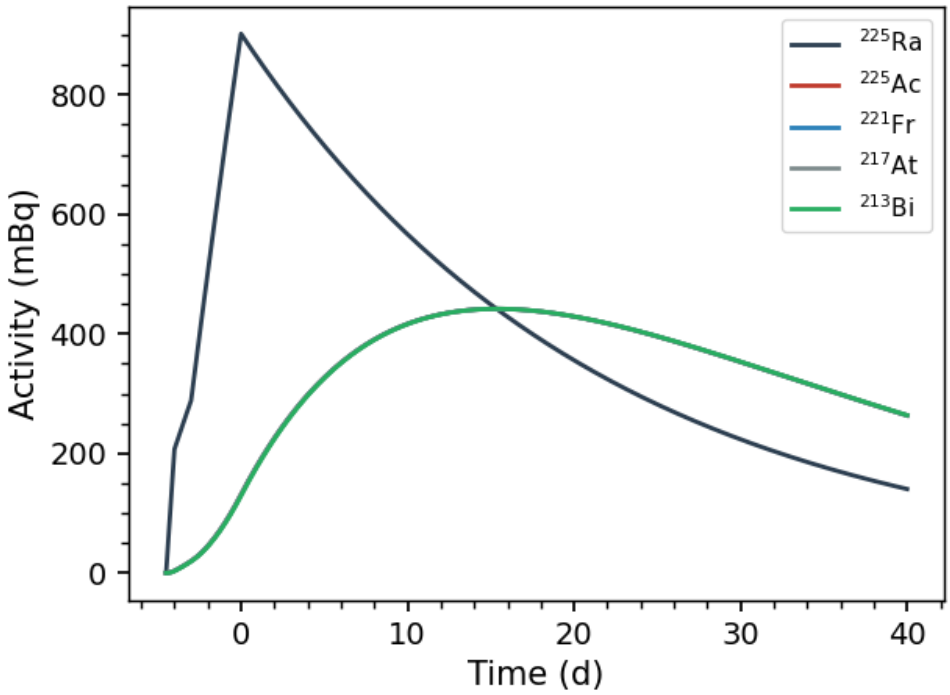
```
import curie as ci
import numpy as np

dc = ci.DecayChain('225RA', R=[[9, 0.5], [2, 1.5], [5, 4.5]], units='d')
print(dc.isotopes)
```

```
['225RAg', '225ACg', '221FRg', '217ATg', '213BIg', '217RNg',
 '209TLg', '213POg', '209PBg', '209BIg', '205TLg']
```

Curie followed the decay data all the way to stability: the chain includes ^{225}Ac and its short-lived descendants down to stable thallium and bismuth. Plotting shows the whole history — production before t = 0, decay after:

```
dc.plot(time=np.linspace(0, 40, 500), max_plot=5)
```



The parent's activity peaks at the end of bombardment (t = 0), while ^{225}Ac grows in for about two weeks afterwards — the daughters sit on a common curve because the sub-^{225}Ac chain reaches equilibrium within hours. Any single value is available directly, in the chain's units (days here):

```
>>> print(dc.activity('225AC', time=10))
0.4168212066113324
>>> print(dc.R_avg)
  isotope     R_avg
0  225RAg  4.777778
```

If measured counts of any chain member are available, `dc.fit_R()` rescales this production history to match them (the shape — the beam's ups and downs — is preserved; only the overall scale is fit).

Inverse: how active was the 152Eu source?

The *Spectroscopy Worked Examples* calibrated a detector with a ^{152}Eu source certified at 37.0 kBq on 01/01/2009, using a spectrum counted in 2018. Here we ask the reverse question: given only that spectrum's fitted peaks, what was the source activity on the certificate date?

Starting from the calibrated spectrum `sp` of the *Spectroscopy Worked Examples*, build a decay-only chain (an `A0` guess rather than a production history) and load the measured decays from the spectrum. The certificate date is our t = 0, passed as the `EoB` (end-of-bombardment) argument — despite the name, it is simply the t = 0 reference time, even when, as here, nothing was bombarded:

```
dc = ci.DecayChain('152EU', A0=3.7E4, units='y')
dc.get_counts([sp], EoB='01/01/2009 12:00:00')
```

`get_counts()` took each fitted gamma line's decays from `sp.peaks` and placed the count interval on the chain's clock — 9.7 years after t = 0:

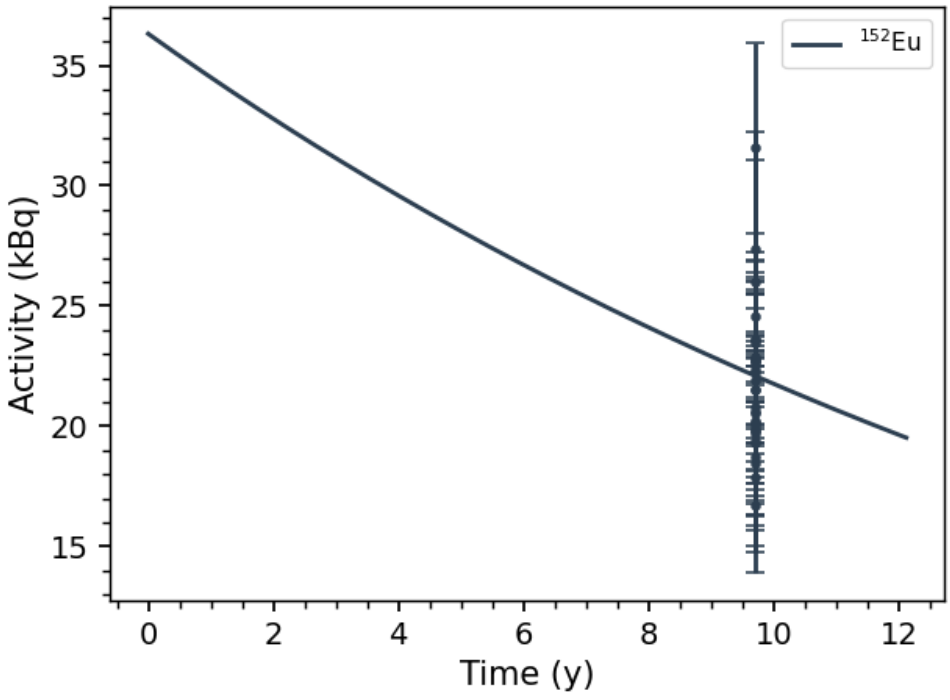
```
>>> print(dc.counts[['isotope', 'start', 'stop', 'counts', 'unc_counts']].head(3))
  isotope     start      stop        counts    unc_counts
0  152EUg  9.697086  9.697163  5.537235e+07  1.366835e+07
1  152EUg  9.697086  9.697163  5.576780e+07  8.454064e+06
2  152EUg  9.697086  9.697163  4.874544e+07  1.161682e+07
```

(The same works without the live spectrum: `get_counts` also accepts the peak file saved in the *Spectroscopy Worked Examples*, as `dc.get_counts(['eu_calib_7cm.Spe'], EoB=..., peak_data='eu_peaks.csv')`.) Now fit the initial activity:

```
>>> isotopes, A0_fit, cov = dc.fit_A0()
>>> print('{:.3g} +/- {:.2g} Bq'.format(A0_fit[0], np.sqrt(cov[0][0])))
3.63e+04 +/- 2.5e+03 Bq
```

and plot the fitted decay curve against the measurements:

```
dc.plot(max_plot=1, max_plot_error=0.2)
```



The fitted activity at the certificate date is 36.3 +/- 2.5 kBq — in agreement with the certified 37.0 kBq. Each point in the plot is one gamma line's estimate of the activity; the fit combines them while accounting for their shared uncertainties, the efficiency calibration and the line intensities. Shared errors cannot be averaged away by adding more lines, so the combined uncertainty is larger — and more honest — than a naive average of the points would suggest.

Here the source decayed freely, so `fit_A0()` was the right tool. In an activation experiment you would instead give the chain your estimated production history `R` and call `fit_R()` — everything else, `get_counts()` included, works the same way.

4.2.3 Isotope & Decay Chain Troubleshooting

The failure modes on this page share a theme: `DecayChain` trusts the names, times and units you give it, and small mismatches — an isomer suffix, an hour of daylight-saving time — produce quietly wrong answers rather than errors.

Isotope not found, or the wrong state of the right isotope

Symptom: `ValueError: Isotope 99TCm2 not found in the decay database` — or, more insidiously, everything runs but half-lives and gamma lines belong to a different state than the one in your sample.

Cause: the isomer suffix. Curie's naming convention is `AAAEL` + state, where the state is `g` (ground), `m` or `m1` (first isomer), `m2` (second isomer). If no state is given, **the ground state is assumed**: `ci.Isotope('99TC')` is the 211,000-year ground state, not the 6-hour ^{99m}Tc used in nuclear medicine. Nothing warns you, because the ground state is a perfectly valid isotope.

Fix: be explicit about isomers (`'99TCm'`), and when a chain misbehaves, print what it actually contains — the names carry explicit suffixes:

```
>>> dc = ci.DecayChain('99MO', units='h')
>>> print(dc.isotopes)
['99MOg', '99TCg', '99TCm1', '99RUg']
```

This matters doubly for measured counts, which are matched by these exact names. `get_counts()` keeps only isotopes that are in the chain, so a peak table whose isotope column says `99TC` feeds the ground state `99TCg` and contributes nothing to a fit of `99TCm1` — silently. (Hand-entered `dc.counts` fails louder: naming an isotope that is not a radioactive chain member raises `Cannot assign counts to ...`.)

Everything is shifted: time zero and daylight-saving time

Symptom: fitted activities or production rates that are consistently a few percent off (or, for short-lived isotopes, wildly off), with a good-looking fit.

Cause: the decay clock. All of `DecayChain`'s times are measured from t = 0, which is the **end of production** — for counts loaded with `get_counts()`, the `EoB` (end-of-bombardment) argument you supply *is* that zero point, and every spectrum's decay time is computed as (count start - EoB). Two things commonly corrupt it:

- **A wrong or mistyped EoB** (or one in `'%d/%m/%Y'` order — the format is `'%m/%d/%Y %H:%M:%S'`) shifts every decay time by the same amount, which biases every fitted activity by the decay factor of that shift.
- **Daylight-saving time.** Curie's datetimes are timezone-naive; it subtracts wall-clock times. If the irradiation happened in March and the count in July, and your EoB is written in winter (standard) time while the acquisition computer stamped summer time, every decay time is off by one hour. A one-hour shift is about a sixth of a ^{99m}Tc half-life — roughly an 11% error in activity — or negligible for ^{152}Eu. Whether it matters depends entirely on your shortest-lived isotope.

Fix: record EoB and verify the spectrum's `start_time` (printed from the file header: `print(sp.start_time)`) on the *same* clock, ideally one that does not observe daylight-saving shifts (UTC, or your local standard time year-round). A quick sanity check is to print the decay times Curie computed:

```
dc.get_counts([sp], EoB='03/12/2026 06:30:00')
print(dc.counts[['isotope', 'start', 'stop']])
```

and confirm `start` (in chain units) is the decay interval you expect.

"Cannot fit R" — or, which fit do I use?

Symptom: `ValueError: Cannot fit R: R=0.` from `fit_R()`, or a `fit_A0()` result that makes no sense for an irradiated sample.

Cause: the two fits answer different questions, and each needs the matching starting condition:

- `fit_R()` scales a **production-rate history** — it requires the chain to have been built with `R`, and answers "what production rate explains my measured decays?" (The error above means the chain has no `R` to scale.)
- `fit_A0()` scales an **initial activity** — for a sample that was simply decaying from t = 0, with no production being modeled.

Fix: for activation experiments, build the chain with your best estimate of the production history (even a single interval, `R=[[1, t_irr]]`, works — the absolute scale is what's being fit) and use `fit_R()`. For a decaying source with no production, give `A0` (any nonzero guess) and use `fit_A0()`. In both cases the fitted quantity is a single scale factor per isotope: the *shape* of what you provided — the beam's time structure, or the relative activities — is preserved, so a wrong shape (e.g. ignoring a beam interruption) is not corrected by the fit.

4.3 Reactions

Curie provides evaluated nuclear reaction cross sections through two classes: the `Reaction` class holds the cross section for one specific reaction — its energy grid, values, uncertainties (where available), and methods to interpolate, flux-average and plot — and the `Library` class searches the underlying data libraries for what reactions exist.

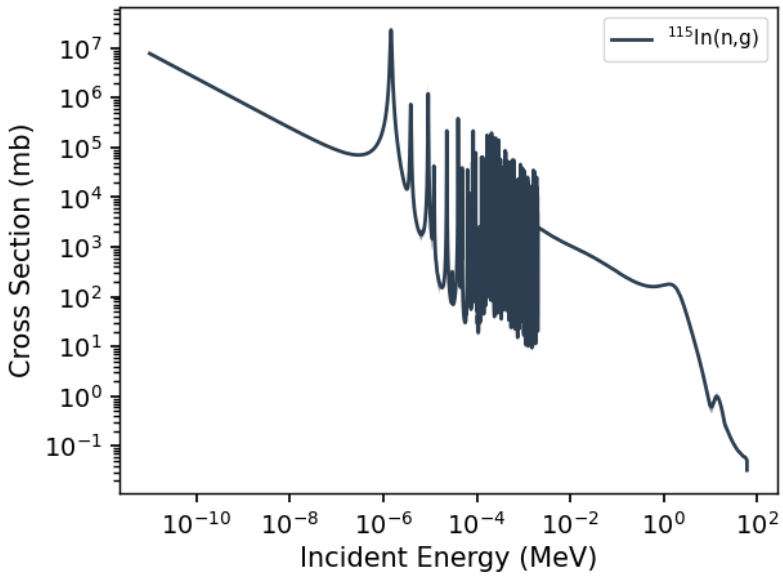


Fig. 3: The ^{115}In(n,g) capture cross section (IRDFF-II), across ten orders of magnitude in incident energy.

Reactions are named in standard nuclear reaction notation — the pattern is `TARGET(incident,outgoing)PRODUCT`, with isotopes written in either of Curie's name forms:

```
rx = ci.Reaction('115IN(n,g)')              # neutron capture on 115In
rx = ci.Reaction('Ra-226(n,2n)Ra-225')      # (n,2n), naming the product
rx = ci.Reaction('natTI(p,x)48V')           # any route from natural Ti to 48V
```

All cross sections are in mb, and all energies in MeV.

Workflow. Using reaction data usually takes three steps:

1. **Find** the reaction. If you know its name, construct it directly; if not, `Library.search()` lists what a library contains (`ci.Library('iaea').search(target='natTI', incident='p')`).
2. **Choose** the source. By default (`library='best'`) Curie picks the highest-priority library carrying the reaction; pass a library name to choose yourself. `rx.library.name` always reports which library was selected.
3. **Use** the data: `rx.interpolate(energy)` for values on your own energy grid, `rx.average(energy, flux)` for the flux-averaged cross section of a measurement, `rx.plot()` to look at it.

See the *Reaction How-to Guide* for each step in detail, the *Reactions Worked Examples* for a complete monitor-reaction example with library comparisons, and *Reactions Troubleshooting* for the common failure modes.

Uses and limitations. These are *evaluated* libraries — smooth, recommended curves produced by evaluators from experimental data and nuclear-model calculations — not raw experimental data points (EXFOR, the international compilation of such measurements, is not included). Each library has a scope and a character: the IAEA library is a set of precisely evaluated monitor (beam-normalization) reactions with uncertainties; IRDFF-II is a neutron-dosimetry standard, also with uncertainties; ENDF/B-VIII.1 is the general-purpose neutron evaluation; and the TENDL-2025 libraries are theory-driven (TALYS-based) evaluations whose strength is complete coverage — nearly every target and product, including reactions no one has measured. Where they overlap, they will not agree perfectly; which to prefer, and what to watch out for (energy-grid limits, natural vs isotopic targets, cumulative vs direct/independent production), is covered in the *Reactions Worked Examples*.

Averages vs. integrals. `Reaction` offers two ways to combine a cross section with a particle spectrum, and which one you want depends on what your flux array *means*:

- `rx.average(eng, phi)` returns the **flux-weighted average cross section** $\langle\sigma\rangle$, in mb. Only the *shape* of `phi` matters — its normalization cancels — so this is the right tool when the spectrum is relative: the energy distribution of beam particles in a foil from `Stack`, for example, or any spectrum you know only up to a constant. The absolute physics then enters through numbers you know separately. For a thin target in a charged-particle beam, the production rate that `DecayChain` takes as its `R` input is

  $$R\ [\text{atoms/s}] = I_p\, n_t\, \langle\sigma\rangle \times 10^{-27}$$

  with I_p the beam current (particles/s), n_t the areal number density of target atoms (atoms/cm2), and 10^{-27} converting mb to cm2. (Note n_t is *atoms* per area — from a foil's areal density in mg/cm2, as `Stack` reports it, multiply by $10^{-3}N_A/M$ with M the molar mass in g/mol.)

- `rx.integrate(eng, phi)` returns the **flux integral** $\int \sigma(E)\,\phi(E)\,dE$. Here the normalization of `phi` is the point: give an absolute differential flux (particles per cm2 per second per MeV) and the integral is proportional to the reaction rate per target atom (raw units mb/cm2/s) — multiply by 10^{-27} to convert mb to cm2, giving the per-atom rate in 1/s, and by the number of target atoms for the production rate. This is the natural form for yield calculations in a fully specified field, such as a reactor neutron spectrum quoted as a differential flux.

The two are related by the total flux — `integrate(eng, phi) = average(eng, phi) * sum(phi*dE)` — so this is one calculation viewed two ways: use `average` when the shape and the magnitude of your flux come from different places, and `integrate` when one spectrum carries both.

4.3.1 Reaction How-to Guide

This page is a task-by-task reference for the `Reaction` and `Library` classes. All examples assume `import curie as ci`. The library descriptions and integration conventions are detailed in *Nuclear Reaction Data*.

Getting a reaction

Reactions are written `TARGET(incident,outgoing)PRODUCT`. The target and product take the same isotope names as everywhere in Curie (see *Isotope & Decay Chain How-to Guide*), the incident particle is `n`, `p` or `d` (plus `a` alpha, `h` helion — a ^{3}He nucleus — and `g` photon in the IAEA library), and the outgoing particle is a shorthand like `g`, `2n`, `p`, `a`, `inl` (inelastic), `f` (fission) — or `x`, meaning "anything": only the product is specified:

```
rx = ci.Reaction('115IN(n,g)')            # radiative capture
rx = ci.Reaction('Ra-226(n,2n)Ra-225')
rx = ci.Reaction('natTI(p,x)48V')          # any route from natTi to 48V
```

A natural-abundance elemental target is written with the `nat` prefix (`natTI`, `natCU`); this notation is specific to reaction targets — it is not a valid `Isotope` name.

The product can be omitted when the outgoing particle determines it (`'115IN(n,g)'` is `'115IN(n,g)116IN'`), but is needed to select an isomer — a longer-lived excited state of the product nucleus, written with an `m` suffix (`'115IN(n,inl)115INm1'`). The data are attributes: `rx.eng` (MeV), `rx.xs` (mb), `rx.unc_xs` (mb; zeros if the library provides no uncertainties), and `rx.TeX` for plot labels.

How Curie picks the library

With the default `library='best'`, Curie tries the libraries in a fixed priority order and keeps the first one that carries the reaction:

Incident particle	Priority order
neutron	IRDFF-II -> ENDF/B-VIII.1 -> IAEA -> TENDL-2025 -> TENDL-2025 (residual product)
proton, deuteron, alpha	IAEA -> TENDL-2025 (residual product)
helion, photon	IAEA

The order reflects evaluation care: dosimetry and monitor standards first, general-purpose evaluations next, all-encompassing theoretical libraries last. Check which library was selected, and pin it explicitly when reproducibility matters (in a publication, name the library — 'best' can resolve differently as libraries are added or updated):

```
rx = ci.Reaction('90ZR(n,2n)')
print(rx.library.name)           # IRDFF-II
rx = ci.Reaction('90ZR(n,2n)', 'endf')
```

Exclusive vs. residual-product libraries

The neutron libraries (ENDF, TENDL, IRDFF) are organized by *exclusive reaction channel* — the outgoing particles are specified, as in `(n,2n)`. The residual-product libraries (the IAEA library, and the TENDL `tendl_n`/`tendl_p`/`tendl_d` variants) are instead organized by what nucleus is produced: the reaction is written `(p,x)` and only the product matters, summing over every route to it (the full library taxonomy is in *Nuclear Reaction Data*). This is why proton reactions are written `'natTI(p,x)48V'` rather than `'48TI(p,n)48V'`.

In the TENDL residual-product libraries every product state is a separate entry: `86SR(p,x)86Yg` and `86SR(p,x)86Ym1` are different reactions. If you give no isomer suffix, **the ground state is assumed** (a warning prints, once per library instance).

Searching a library

When you don't know the exact reaction name, search with any combination of target, incident/outgoing particle, and product:

```
>>> lb = ci.Library('tendl_p')
>>> print(lb.search(target='Sr-86', product='Y-86g'))
['86SR(p,x)86Yg']
>>> lb = ci.Library('endf')
>>> print(lb.search(target='226RA', outgoing='2n'))
['226RA(n,2n)225RA']
```

Searching by target alone lists everything the library evaluates for that nucleus — for ENDF that includes totals, elastic scattering and level-by-level inelastic channels, not just activation products.

Library names are `'endf'`, `'tendl'`, `'irdff'`, `'iaea'`, and `'tendl_n'`/`'tendl_p'`/`'tendl_d'`/`'tendl_a'` for the residual-product libraries. `search` returns reaction names ready to pass to `Reaction`.

Values on your energy grid

`rx.interpolate(energy)` evaluates the cross section wherever you need it, and `rx.interpolate_unc(energy)` its uncertainty:

```
rx = ci.Reaction('115IN(n,g)', 'irdff')
print(rx.interpolate([0.5, 1.0, 5.0]))       # mb, at 0.5, 1 and 5 MeV
```

Interpolation is linear for the pointwise-linearized libraries and monotone PCHIP in sqrt(E)-sqrt(sigma) space for the smooth TENDL curves (exact through the evaluated points, no overshoot at thresholds); either scheme can be selected per reaction:

```
rx.interpolate(energy, interpolation='linear')   # kept for later calls
rx.interp_config = {'interpolation': 'pchip-sqrt'}
```

One convention to know: **outside the library's evaluated energy range the interpolated cross section is zero** — Curie never extrapolates. Check `rx.eng.min()` and `rx.eng.max()` when working near the edges of a library's grid; the consequences for flux averages are shown in the *Reactions Worked Examples*.

Flux averages and integrals

For an activation measurement, the effective cross section is the flux-weighted average over the particle spectrum. Give `average()` your energy grid and the flux on that grid:

```
import numpy as np

rx = ci.Reaction('90ZR(n,2n)', 'irdff')      # a threshold reaction
eng = np.linspace(10, 30, 50)
phi = np.exp(-0.5*((eng - 20)/3.5)**2)       # any relative shape
print(rx.average(eng, phi))
print(rx.average(eng, phi, unc=True))        # (value, uncertainty)
```

The flux needs no normalization — only its shape matters for the average. `rx.integrate(eng, phi)` returns the flux integral $\int \sigma(E)\,\phi(E)\,dE$ instead, for which the flux's absolute normalization *does* matter (which to use when, and how each becomes a production rate, is laid out on the *Reactions* overview page). Cross-section uncertainties are treated as fully correlated between energies — see *Nuclear Reaction Data* for the exact conventions.

Plotting

`rx.plot()` shows the cross section over its full grid, with an uncertainty band when the library provides one. Overlay reactions or libraries by passing figure and axes through:

```
rx = ci.Reaction('90ZR(n,2n)', 'irdff')
f, ax = rx.plot(return_plot=True, label='library')
rx = ci.Reaction('90ZR(n,2n)', 'endf')
rx.plot(f=f, ax=ax, label='library')
```

`label` can be `'reaction'`, `'library'` or `'both'`; pass `energy=` to plot on your own grid; `scale='loglog'` sets the axes (the full list of scale options is on the *Plotting* page).

4.3.2 Reactions Worked Examples

This page works through the reaction data behind a typical charged-particle activation measurement — a *monitor reaction* — and uses it to show how the libraries differ, and why those differences matter: natural versus isotopic targets, cumulative versus direct production, and the limits of each library's energy grid.

A monitor reaction

In a stacked-target irradiation, thin foils of well-studied materials are interleaved with the samples: the activities induced in these *monitor foils*, together with precisely evaluated cross sections, measure the beam current and energy in each position. The IAEA maintains the reference library for exactly this purpose, and it is where Curie's default library search sends charged-particle reactions:

```
>>> import curie as ci
>>> import numpy as np

>>> rx = ci.Reaction('natTI(p,x)48V')
>>> print(rx.library.name)
IAEA CP-Reference (2017)
```

Note the notation: a *natural* titanium target (`natTI` — monitor foils are natural metal), and `(p,x)` — any reaction route that ends at ^{48}V. The IAEA library provides uncertainties, so the flux-averaged cross section of a measurement comes with an error bar. For a proton spectrum centered at 20 MeV:

```
>>> eng = np.linspace(10, 30, 50)
>>> phi = np.exp(-0.5*((eng - 20)/3.5)**2)
>>> print(rx.average(eng, phi, unc=True))
(115.2..., 5.2...)
```

Natural vs. isotopic targets

The TENDL-2025 residual-product libraries carry this reaction for the natural target directly — `natTI(p,x)48V` works there just as it does in the IAEA library. The natural-element entries are abundance-weighted sums of the isotopic tables, computed when the databases are built:

```
r_nat = ci.Reaction('natTi(p,x)48Vg', 'tendl_p')
```

(The explicit `g` on the product picks the ground state — leaving it off works, but prints the ground-state-assumed warning.)

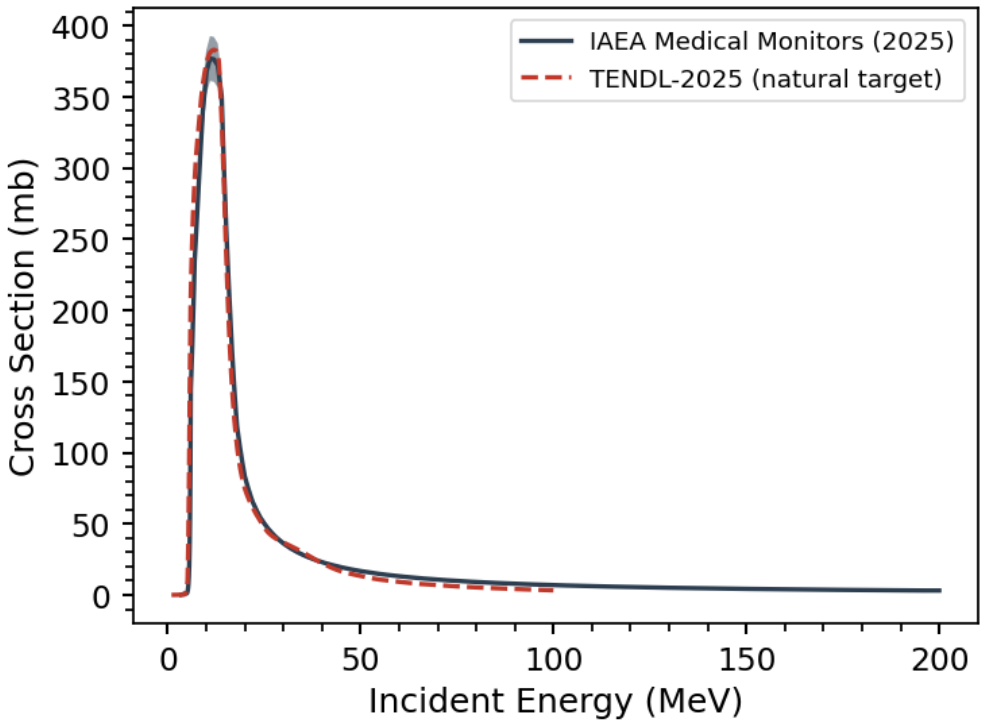
The weighted sum is also easy to carry out by hand, which shows exactly what the natural entry *is*. `ci.Element('Ti').abundances` provides the table the sum needs — the stable isotopes with their percent abundances (columns `isotope` and `abundance`):

```
el = ci.Element('Ti')
eng = np.linspace(1, 100, 500)
xs_nat = np.zeros_like(eng)

for n, row in el.abundances.iterrows():
        try:
                r = ci.Reaction('{0}(p,x)48Vg'.format(row['isotope']), 'tendl_p')
        except ValueError:
                continue      # this isotope has no route to 48V
        xs_nat += 1E-2*row['abundance']*r.interpolate(eng)
```

The `try/except` is doing real physics: ^{46}Ti + p has only 47 nucleons, so it cannot make ^{48}V at any energy, and TENDL has no such entry. Overlaying the natural-target entry on the IAEA evaluation:

```
f, ax = rx.plot(return_plot=True, label='library')
ax.plot(eng, r_nat.interpolate(eng), ls='--', label='TENDL-2025 (natural target)')
ax.legend()
```



The two agree in shape, and for this reaction within ~1% at the peak — closer than is typical. The distinction still matters, and it reflects what each library *is*: the IAEA curve is an evaluation of decades of monitor-foil measurements; the TENDL curve comes from nuclear-model (TALYS) calculations, whose strength is covering every reaction — including ones never measured — rather than pinpoint accuracy on any one. For beam monitoring, use the IAEA values.

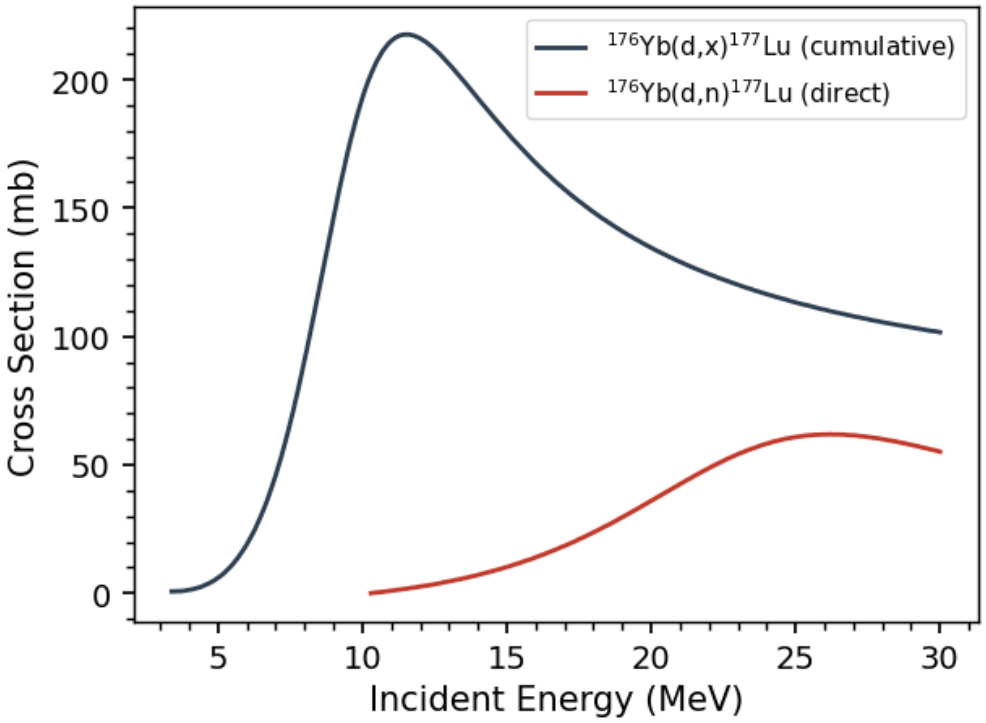
Cumulative vs. direct production

A subtlety of residual-product data: does "production of X" include the decay of short-lived parents into X, or only X made directly? The IAEA library distinguishes the two where it matters. For ^{177}Lu — the therapy isotope — produced by deuterons on ^{176}Yb, both are evaluated:

```
ra = ci.Reaction('176YB(d,x)177LUg', 'iaea')   # cumulative
rb = ci.Reaction('176YB(d,n)177LUg', 'iaea')   # direct only

f, ax = ra.plot(return_plot=True, label='reaction')
rb.plot(f=f, ax=ax, label='reaction')
```

The cumulative curve — here several times the direct one — includes ^{177}Yb, which is also made by the beam and decays entirely to ^{177}Lu with a 1.9 h half-life. After a few hours of cooling, the cumulative cross section is what an activity measurement of ^{177}Lu actually sees. TENDL's residual-product entries are, by contrast, *independent*: each state's direct production only, with decay feeding left to you (`DecayChain` does exactly that bookkeeping — see *Isotopes & Decay Chains*). When comparing libraries or using a cross section to predict an activity, always ask which convention you are



holding.

Neutron libraries and the 20 MeV cutoff

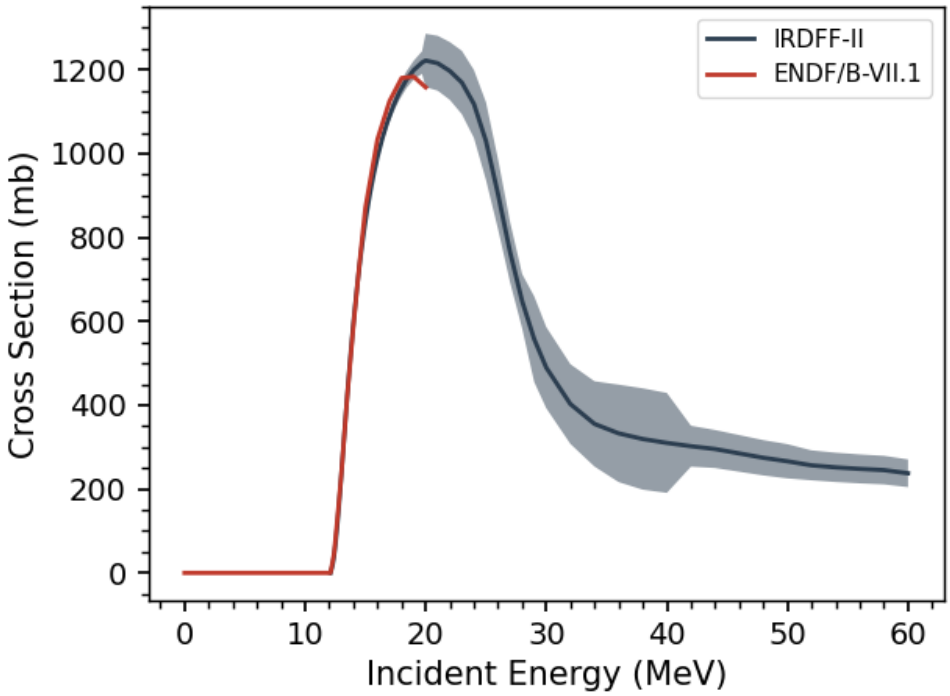
The same look-before-you-use advice applies to energy grids. Most ENDF/B-VIII.1 neutron evaluations stop at 20 MeV; IRDFF-II extends its dosimetry reactions to 60 MeV:

```
r1 = ci.Reaction('90ZR(n,2n)', 'irdff')
f, ax = r1.plot(return_plot=True, label='library')
r2 = ci.Reaction('90ZR(n,2n)', 'endf')
r2.plot(f=f, ax=ax, label='library')
```

The two agree where they overlap — but the ENDF curve simply ends at 20 MeV, in the middle of the peak. Beyond a library's grid Curie returns *zero* rather than extrapolating, so a flux average silently loses whatever part of the spectrum the library doesn't cover. For a flat 12–30 MeV flux:

```
>>> eng = np.linspace(12, 30, 50)
>>> print(r1.average(eng, np.ones(50)))   # IRDFF-II
885.1...
>>> print(r2.average(eng, np.ones(50)))   # ENDF: zero above 20 MeV
362.6...
```

Same reaction, same flux — and a factor of 2.4 between the answers, entirely from the grid. Whenever your spectrum extends above ~20 MeV, check the reaction's `.eng.max()` against it, and prefer IRDFF-II (or TENDL, which extends to 200 MeV) for the high-energy part.



4.3.3 Reactions Troubleshooting

Most reaction-data problems are one of three things: the reaction was written in a form the library doesn't use, the automatic library choice wasn't what you assumed, or the target was natural where the library is isotopic (or vice versa).

"Reaction ... not found or not unique"

Symptom: `ValueError: Reaction 48TI(p,x) not found or not unique.` for a target the library certainly evaluates.

Cause: the error covers two opposite situations — *nothing* matched (not found), or *several* things matched (not unique) — and each library organizes its reactions in one particular form. The common cases:

- **No product given to a residual-product library** ("not unique"): `'48TI(p,x)'` alone matches every one of the ~200 products TENDL evaluates for that target. Specify the product.
- **A channel form the library doesn't use** ("not found"): `'natTI(p,n)48V'` fails in the IAEA library — its entries are indexed by product, as `(p,x)` — while `'90ZR(n,x)89ZR'` fails in ENDF, which evaluates exclusive channels: `'90ZR(n,2n)'`.
- **The reaction genuinely isn't in that library** — the target isn't evaluated, or the product is out of reach.

(Related, but not an error: a TENDL residual-product search without an isomer suffix quietly assumes the ground state — `'86SR(p,x)86Y'` means `86Yg`, with a warning that prints once per library instance.)

Fix: ask the library what it has, with the loosest search that brackets your case:

```
>>> lb = ci.Library('tendl_p')
>>> print(lb.search(target='48TI'))
['48TI(p,x)44SCg', '48TI(p,x)44SCm1', ..., '48TI(p,x)48Vg', ...]
```

and pass one of the returned names verbatim to `Reaction`.

Which library am I actually using?

Symptom: results change when a colleague runs the "same" analysis; plots have a different energy range than expected; a cross section has no uncertainties even though you thought it should.

Cause: `library='best'` is a *search order*, not a merge: Curie takes the first library in the priority list (see *Reaction How-to Guide*) that contains a unique match. Two consequences are easy to miss. First, similar-looking reactions can come from different libraries — `'226RA(n,2n)'` resolves to ENDF/B-VIII.1, but the residual-product form `'226RA(n,x)225RA'` resolves to TENDL-2025 — with different grids, values and uncertainty availability. Second, the resolution can change over time, as data libraries are updated or the priority evolves between Curie versions.

Fix: `print(rx.library.name)` — always, before using a number — and pass the library explicitly (`ci.Reaction('115IN(n,g)', 'irdff')`) in any analysis you intend to reproduce or publish.

Natural target or isotopic target?

Symptom: `'natTI(p,x)48V'` works by default (it resolves to the IAEA library) — but the same reaction with `library='tendl_p'` raises "not found". Or: an isotopic cross section is a factor of several above the measured production on a natural foil.

Cause: the libraries differ. The IAEA monitor library evaluates *natural* targets (monitor foils are natural metal); TENDL's residual-product libraries are *isotopic only*. And the two aren't interchangeable: an isotopic cross section overstates production on a natural target by roughly the inverse of the isotope's abundance, and a natural cross section understates the yield from an enriched target.

Fix: match the target to your actual material. For a natural target with only isotopic data available, build the abundance-weighted sum (the *Reactions Worked Examples* shows the ~6-line loop, including why some isotopes contribute nothing). For an enriched target with only natural data, there is no clean inversion — use the isotopic library.

4.4 Stopping Power Calculations

Three classes handle the interaction of particles with bulk matter. `Element` and `Compound` describe the materials: their charged-particle stopping powers and ranges (Andersen–Ziegler formulation, any element up to uranium), and their photon attenuation coefficients. `Stack` puts materials in a beam: it transports charged particles through a stack of foils and computes the energy distribution of the beam in every layer — the quantity that connects an irradiation to its cross sections.

Workflow. `Element` and `Compound` answer design questions directly — how much energy does a proton lose in this foil (`S`), how thick a degrader do I need (`range`), how strongly does this sample absorb gamma rays (`attenuation`). A `Stack` calculation follows three steps:

1. **Define the materials**: natural elements by symbol, compounds by chemical formula, preset name (`ci.COMPOUND_LIST`), or explicit elemental weights.
2. **Define the stack**: an ordered list of foils (first foil hit first), each with a compound and enough information to fix its areal density (mass per unit beam area, in mg/cm2), and a `name` for each foil you want tallied.
3. **Transport and use**: `ci.Stack(stack, E0=..., particle='p')` computes every foil's mean energy, energy width and full flux distribution (the "flux" here is the beam's energy spectrum within the foil, not a particles-per-area rate) — and `rx.average(*st.get_flux('name'))` turns the latter into the effective cross section of that foil.

See the *Stopping Power How-to Guide* for each step in detail, the *Stopping Power Worked Examples* for the build-up from single stopping powers to the thin-vs-thick foil comparison, and *Stopping Power Troubleshooting* for the common pitfalls.

Uses and limitations. The stopping powers are the Andersen–Ziegler semi-empirical parameterization (see *Stopping Power and Particle Transport*): protons, deuterons, tritons and alphas are supported directly, and heavier ions through

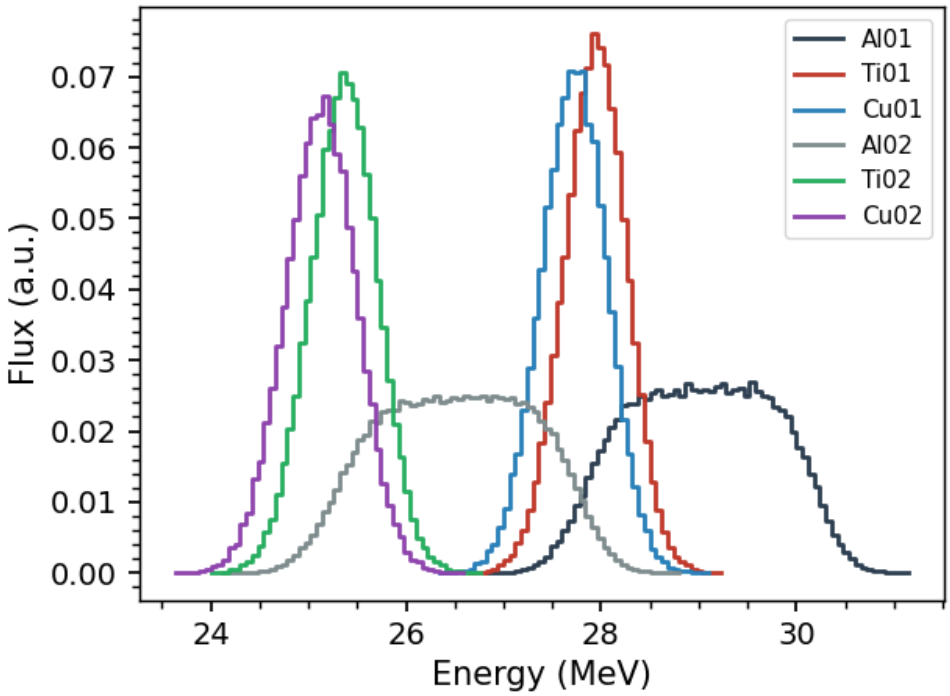


Fig. 4: A 30 MeV proton beam degrading through a six-foil stack: each foil sees a lower, wider energy distribution.

an effective-charge scaling. Compound stopping powers use Bragg additivity (weighted sums of the elemental values), which neglects chemical-bonding effects — a percent-level approximation, worst for light compounds at low energies.

`Stack` models energy loss only: particles slow down but are never absorbed or deflected, so the computed flux distributions describe the beam's *energy*, not its attenuated intensity, and lateral spread is not modeled. The width of each foil's energy distribution has two parts: the energy the beam loses between the foil's front and back faces (which alone makes a thick degrader's distribution wide, even for a monoenergetic incident beam), plus the incident beam spread (`dE0`) and the extra spread the beam picks up as it degrades — slower particles lose energy faster, so an initially narrow beam broadens as it slows. The width is thus the range of energies the foil samples, not an uncertainty on its mean energy. Collisional straggling — the statistical spread in energy loss from the randomness of individual collisions — is not added on top, so very thick degraders will in reality produce a somewhat wider distribution than computed (see *Stopping Power and Particle Transport*).

4.4.1 Stopping Power How-to Guide

This page is a task-by-task reference for the `Element`, `Compound` and `Stack` classes. All examples assume `import curie as ci`. The stopping-power formulation and the transport algorithm are described in *Stopping Power and Particle Transport*.

Elements

An `Element` is a natural-abundance element, constructed from its symbol:

```
el = ci.Element('Fe')
print(el.mass)        # amu, abundance-weighted
print(el.density)     # g/cm3, at standard conditions
print(el.abundances)  # table of stable isotopes and abundances
```

The `abundances` table (columns `isotope`, `abundance` in percent) is the same one used to abundance-weight isotopic cross sections in the *Reactions Worked Examples*. The preset `density` is used as the default by the methods below; pass `density=` to override it.

Photon attenuation

`mu()` and `mu_en()` return the mass-attenuation and mass-energy-absorption coefficients (cm2/g) at a given photon energy (in keV, as everywhere photons appear), and `attenuation()` the fraction of photons transmitted through a thickness `x` (cm):

```
el = ci.Element('Pb')
print(el.mu(661.7))                  # cm2/g at the 137Cs line
print(el.attenuation(661.7, x=1.0))  # transmission through 1 cm
```

These are the same coefficients `Spectrum.attenuation_correction()` uses for sample self-absorption (see *Spectroscopy How-to Guide*).

Stopping powers and ranges

`S()` returns the stopping power $-dE/dx$ for a charged particle, at an energy in MeV:

```
el = ci.Element('Ti')
print(el.S(15.0))                 # MeV/cm, at the preset density
print(el.S(15.0, density=1E-3))   # MeV/(mg/cm2) - mass stopping power
```

The `density=1E-3` idiom returns the *mass* stopping power, which is what stacked-target work uses (areal densities in mg/cm2). The particle is `'p'` (default), `'d'`, `'t'` or `'a'` — or any element or isotope name (`'Fe'`, `'40CA'`) for heavy ions.

`range()` integrates the stopping power to give the distance a particle travels before stopping, in cm:

```
print(el.range(15.0))             # ~0.087 cm for 15 MeV protons in Ti
```

Both have plotting companions, `plot_S()` and `plot_range()`, which accept the shared plotting keywords (overlay with `f`/`ax`).

Compounds

A `Compound` is defined by elemental weights and a density. Three ways to make one:

```
cm = ci.Compound('H2O', density=1.0)     # chemical formula
cm = ci.Compound('Kapton')               # preset (see ci.COMPOUND_LIST)
cm = ci.Compound('Brass', weights={'Cu':-66, 'Zn':-33}, density=8.5)
```

Formulas support decimal subscripts (`'C0.5O'`) but not parentheses — write Ca3(PO4)2 as `'Ca3P2O8'`. In a `weights` dict, **positive numbers are atom fractions, negative numbers are mass fractions** (the brass above is 66% copper *by weight*); either kind is normalized for you. Weights can also come from a .csv/.json/.db file describing many compounds at once. A `Compound` has the same `mu`, `mu_en`, `attenuation`, `S`, `range` and plotting methods as an `Element`, with elemental values combined as mass-fraction-weighted sums of the elemental ones (Bragg additivity — see *Stopping Power and Particle Transport*).

Building a stack

A stack is an ordered list of foils — first foil hit first — passed to `Stack` with the beam parameters:

```
stack = [{'cm':'Al', 't':0.5,   'name':'Al01'},
         {'cm':'Ti', 't':0.025, 'name':'Ti01'},
         {'cm':'Kapton', 't':0.05},
         {'cm':'Cu', 't':0.025, 'name':'Cu01'}]

st = ci.Stack(stack, E0=30.0, particle='p')
```

Each foil needs a compound (`'compound'`, shorthand `'cm'`) and enough information to fix its **areal density** in mg/cm2, given directly (`'areal_density'`/`'ad'`) or computed from: `'mass'` (g) with `'area'` (cm2), or `'thickness'` (mm) with `'density'` (g/cm3) — the density coming from the compound definition if not given per-foil. Every key has a shorthand, used in the example above: `cm` = compound, `t` = thickness, `d` = density, `m` = mass, `a` = area, `ad` = areal_density, `nm` = name.

Foils with a `'name'` are tallied; unnamed foils (the Kapton above) still degrade the beam and still appear in `st.stack`, but are excluded from the tallied results (`st.fluxes`, `get_flux`, `plot`, `summarize`, `saveas`) — the convention for passive foils such as degraders and beam catchers (foils placed to slow or stop the beam rather than to be analyzed). The stack can equally be a DataFrame or a .csv/.json/.db file with the same columns, and custom compounds can be supplied with the `compounds=` argument.

Reading the results

`st.stack` summarizes every foil: its areal density, the mean beam energy `mu_E` and the 1-sigma energy width `sig_E` (MeV):

```
>>> print(st.stack)
   name compound  areal_density       mu_E     sig_E
0  Al01       Al       134.9000  29.015085  0.730443
1  Ti01       Ti        11.2975  27.935964  0.321319
...
```

The full energy distributions are in `st.fluxes`, plotted with `st.plot()`, and retrieved per foil with `get_flux()` — whose output feeds directly into a cross-section average:

```
rx = ci.Reaction('natTI(p,x)48V')     # a reaction on the foil material
eng, phi = st.get_flux('Ti01')
print(rx.average(eng, phi))
```

`st.summarize()` prints the energy assignments, and `st.saveas()` writes the stack table (and optionally the fluxes) to .csv/.json/.db.

Solver options

The defaults suit most problems; the parameters, all keyword arguments to `Stack`: `dE0` (1-sigma width of the incident beam energy, default 1% of `E0`), `N` (number of Monte Carlo particles, default 10000), `accuracy`/`min_steps`/`max_steps` (energy-loss stepping control per foil), and `dp` (a density multiplier applied to the whole stack — useful as a fit parameter when tuning a stack model to measured monitor data).

4.4.2 Stopping Power Worked Examples

This page builds from single stopping powers up to a foil stack, and ends with the comparison that motivates the whole machinery: how the beam's energy *distribution* in a foil — not just its mean energy — determines the effective cross section.

Stopping powers

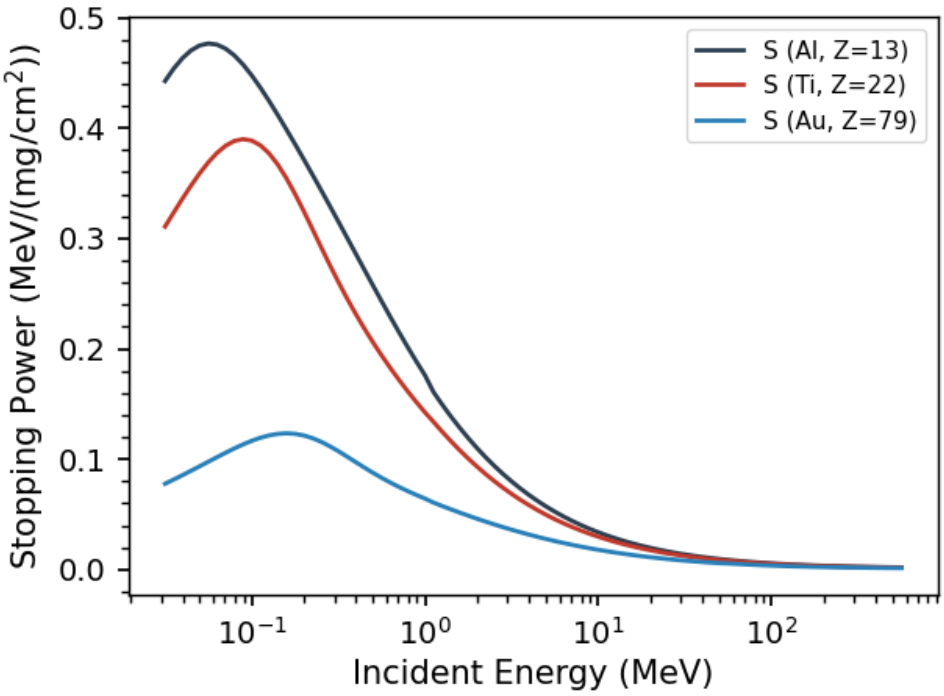
A charged particle slows continuously in matter, losing energy at the rate $S = -dE/dx$ — the *stopping power*. Curie computes it from the Andersen–Ziegler parameterization (see *Stopping Power and Particle Transport*) for any element or compound:

```
>>> import curie as ci
>>> import numpy as np

>>> el = ci.Element('Ti')
>>> print(el.S(15.0))              # MeV/cm for 15 MeV protons
99.179...
>>> print(el.S(15.0, density=1E-3))  # MeV/(mg/cm2), the mass stopping power
0.02194...
```

Overlaying a few elements shows the systematics — stopping falls with energy (faster particles interact less) and, per unit mass, with the atomic number of the material:

```
f, ax = ci.Element('Al').plot_S(return_plot=True)
ci.Element('Ti').plot_S(f=f, ax=ax)
ci.Element('Au').plot_S(f=f, ax=ax)
```



Ranges

Integrating the stopping power gives the *range* — how far the particle travels before stopping:

```
>>> print(el.range(15.0))       # cm
0.0871...
>>> print(10*el.range(15.0))    # mm
0.871...
```

A 15 MeV proton stops in under a millimeter of titanium. Numbers like these are the first sanity check of any stack design: a foil much thinner than the range perturbs the beam gently; one comparable to the range nearly stops it. Compounds work identically, with presets for common materials:

```
>>> cm = ci.Compound('Kapton')      # in ci.COMPOUND_LIST
>>> print(cm.density)
1.42
>>> print(cm.range(15.0))           # cm, as for elements
0.2036...
>>> print(cm.range(15.0, density=1E-3))
289.24...
```

The second form (the `density=1E-3` idiom again) gives the range as a *areal density* (mass thickness) in mg/cm2 — the unit stacked-target work is done in, since foils are weighed rather than measured.

A first stack

Now put foils in a beam. A common pattern is repeating groups of monitor and target foils separated by degraders (a monitor foil carries a reaction whose cross section is already well known, so its measured activity determines the beam current at that position; a target foil carries the reaction under study) — here two Al–Ti–Cu groups in a 30 MeV proton beam:

```
stack = [{'cm':'Al', 't':0.5,   'name':'Al01'},
         {'cm':'Ti', 't':0.025, 'name':'Ti01'},
         {'cm':'Cu', 't':0.025, 'name':'Cu01'},
         {'cm':'Al', 't':0.5,   'name':'Al02'},
         {'cm':'Ti', 't':0.025, 'name':'Ti02'},
         {'cm':'Cu', 't':0.025, 'name':'Cu02'}]

st = ci.Stack(stack, E0=30.0, particle='p')
```

Thicknesses are in mm; Curie converts them to areal densities using each material's preset density. The result is an energy assignment for every foil:

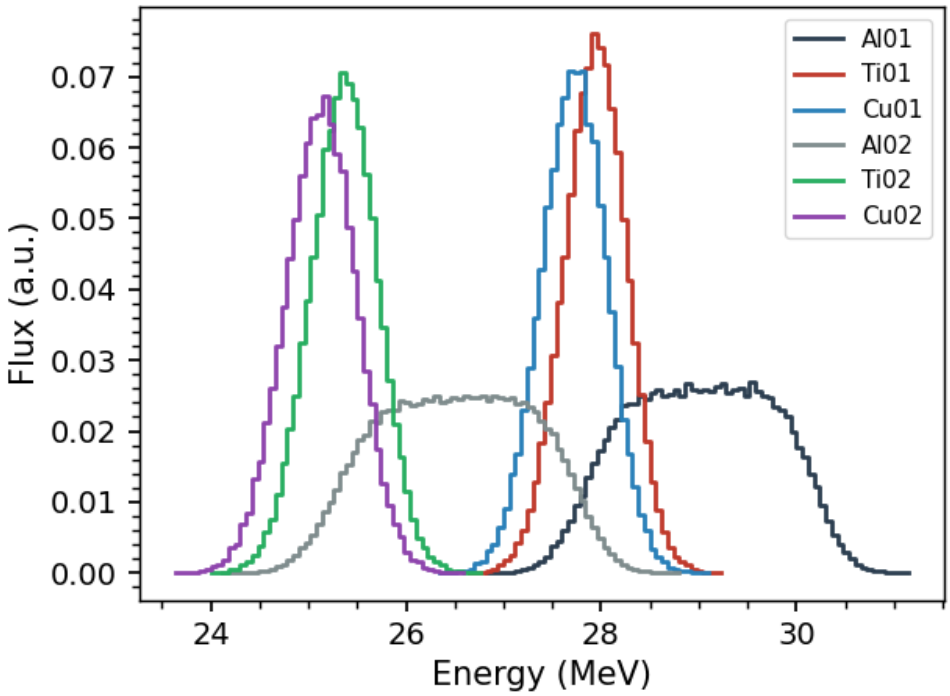
```
>>> print(st.stack)
   name compound  areal_density       mu_E     sig_E
0  Al01       Al       134.9000  29.015085  0.730443
1  Ti01       Ti        11.2975  27.935964  0.321319
2  Cu01       Cu        22.3725  27.717674  0.337406
3  Al02       Al       134.9000  26.517634  0.772389
4  Ti02       Ti        11.2975  25.357176  0.346504
5  Cu02       Cu        22.3725  25.121974  0.364116

st.plot()
```

Each successive foil sees a lower mean energy. The widths follow the foils: `sig_E` is dominated by how much energy the beam loses *within* each foil, so the thick Al degraders show wide, path-averaged distributions (0.73, 0.77 MeV) while the thin Ti and Cu foils record the beam's instantaneous spread (~0.33 MeV) at their depth.

Thin and thick foils: when the mean energy isn't enough

It is tempting to characterize each foil by `mu_E` alone and read the cross section off at that energy. For thin foils that works. For thick foils it can fail badly, because the beam spans a wide slice of the excitation function (the cross section as a function of energy) inside the foil. Compare a 25 um and a 0.75 mm titanium foil, both hit by 15 MeV protons (the range is 0.87 mm — the thick foil absorbs most of the beam's energy):




```
rx = ci.Reaction('natTI(p,x)48V')

st_thin  = ci.Stack([{'cm':'Ti', 't':0.025, 'name':'thin'}],  E0=15.0)
st_thick = ci.Stack([{'cm':'Ti', 't':0.75,  'name':'thick'}], E0=15.0)
```

The thin foil's flux is a narrow spike near 15 MeV; the thick foil's stretches from 15 MeV all the way down to ~4 MeV, across the entire peak of the excitation function (dotted). Comparing the flux-averaged cross section against the shortcut of evaluating at the mean energy:

```
thin:  mu_E = 14.88 MeV, sig_E = 0.18 MeV
       <sigma> = 287.4 mb    sigma(mu_E) = 287.5 mb    (<0.1%)
thick: mu_E = 10.51 MeV, sig_E = 2.96 MeV
       <sigma> = 309.4 mb    sigma(mu_E) = 384.7 mb    (+24%)
```

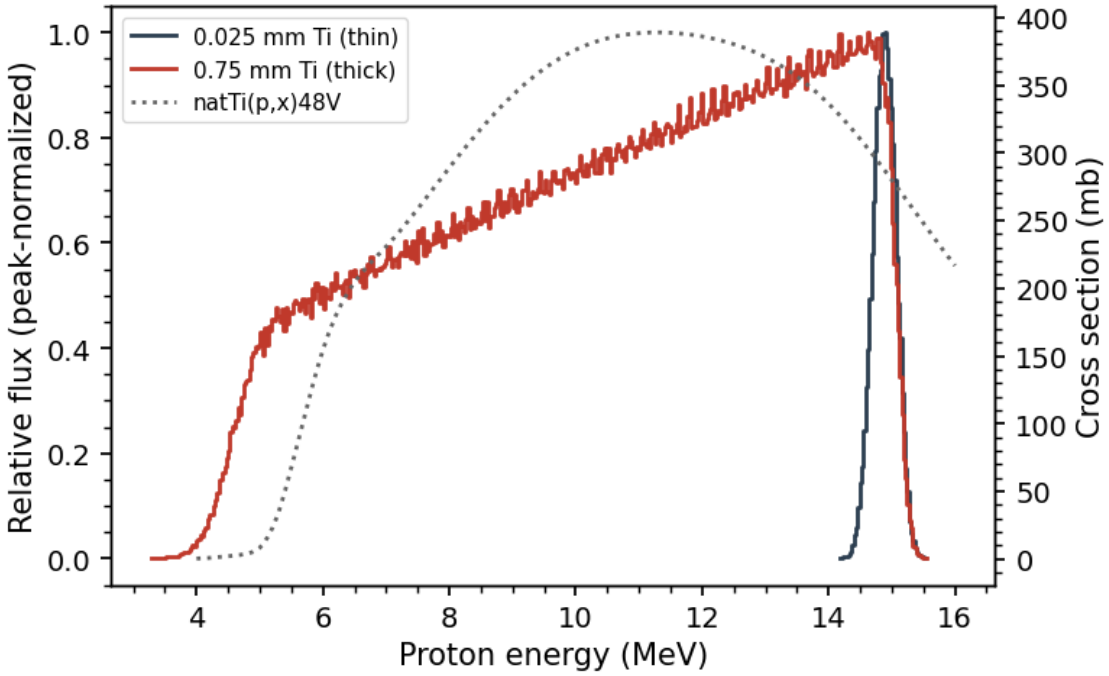
computed, for each stack, as:

```
eng, phi = st_thin.get_flux('thin')
print(rx.average(eng, phi))                          # <sigma>
print(rx.interpolate(st_thin.stack['mu_E'][0]))      # sigma(mu_E)

eng, phi = st_thick.get_flux('thick')
print(rx.average(eng, phi))
print(rx.interpolate(st_thick.stack['mu_E'][0]))
```

(The figure overlays the two `get_flux` distributions with the excitation function `rx.interpolate` on a second axis.)

For the thin foil the two agree to well under a percent; for the thick foil the mean-energy shortcut overestimates the effective cross section by about 24%, because the mean energy happens to sit near the peak while much of the flux does not. (The exact `Stack` numbers vary slightly run to run, since the beam energy is sampled stochastically, but this



thin-versus-thick contrast is robust.) The rule of thumb: whenever `sig_E` is not small compared to the features of the excitation function, use `rx.average(*st.get_flux(...))` — that is precisely what the flux distributions are for.

4.4.3 Stopping Power Troubleshooting

Stack problems usually appear as energies that don't match expectations. The three failure modes below cover most cases; for all of them, the first diagnostic is the same — `print(st.stack)` and check the `areal_density` column against what you think is in the beam.

Foil energies are far from expectation

Symptom: `mu_E` values much too low (or the beam exhausts halfway down the stack) even though the foils are thin; or energies that barely change through what should be a substantial degrader.

Cause: almost always units. The stack input mixes several, and each must be right:

- `thickness` (`'t'`) is in **mm** — a 25 um foil is `0.025`, and typing `25` makes it a 2.5 cm slab.
- `areal_density` (`'ad'`) is in **mg/cm2**.
- `mass` is in **g** and `area` in **cm2**.
- `density` is in **g/cm3**.

Fix: print `st.stack` and inspect the computed `areal_density` of each foil — a 25 um Ti foil should be ~11 mg/cm2, a 0.5 mm Al foil ~135 mg/cm2. If a foil is off by a factor of 10–1000, one of its input units is wrong.

"WARNING: Beam stopped in foil N"

Symptom: the warning above, foils downstream of foil N with `mu_E` near zero, and nonsense fluxes from `get_flux` for those foils.

Cause: the beam physically ranges out — the summed thickness of the stack exceeds the particle range at `E0`. (`N` is the row index shown by `print(st.stack)`, counting from 0, so the offending foil is `st.stack.iloc[N]`.) Every downstream energy and flux is meaningless (Curie transports energy loss only; a stopped beam has no energy left to

assign). This is sometimes intentional — a beam dump or catcher at the end — and then harmless, as long as the foils you *analyze* sit upstream of the stopping point.

Fix: if the stack *should* pass the beam easily, suspect a thickness-unit error first (previous section) before concluding it truly ranged out. Then compare the range against the summed foil thicknesses — in matching units:

```
>>> print(10*ci.Element('Al').range(30.0))   # x10: range in mm, like 't'
4.359...
```

and either raise `E0`, thin the stack, or — if the stop is intentional — simply ignore the foils at and beyond the stopping depth.

A foil is missing from the results

Symptom: a foil you defined does not appear in `st.summarize()`, `st.plot()` or `st.fluxes`; or `get_flux('Ti-01')` returns empty arrays.

Cause: foils without a `'name'` are deliberately untallied — they degrade the beam but are treated as passive (degraders, spacers, catchers). Only named foils are stored in the results. And `get_flux` matches the name **exactly**: `'Ti01'` and `'Ti-01'` are different strings.

Fix: give every foil you care about a `'name'` in the stack definition, and read the exact spellings back from `print(st.stack)` (the `name` column) before calling `get_flux`. Unnamed foils still affect the transport — their absence from the results does not mean they were skipped.

CHAPTER

FIVE

THEORY & METHODOLOGY

This chapter documents the models and numerical methods that Curie uses, in enough detail to reproduce its results or to describe them in a publication. The user's guide shows *how* to invoke these methods; this chapter defines *what* is computed.

5.1 Gamma-ray Peak Fitting

5.1.1 Peak shape

Curie models a gamma-ray peak in a spectrum from a high-purity germanium (HPGe) detector as a Gaussian with an optional low-energy skew component and an optional step in the background, as a function of the analog-to-digital converter (ADC) channel number x (a bin index that increases with detected energy, which the energy calibration below maps to keV):

$$F(x) = A\,e^{-\frac{(x-\mu)^2}{2\sigma^2}} + R\,A\,e^{\frac{x-\mu}{\alpha\sigma}}\,\mathrm{erfc}\left(\frac{x-\mu}{\sqrt{2}\sigma} + \frac{1}{\sqrt{2}\alpha}\right) + \mathrm{step}\cdot A\,\mathrm{erfc}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right)$$

where A, μ and σ are the amplitude, centroid and width of the Gaussian, and erfc is the complementary error function. The second term is a skewed-Gaussian tail on the low-energy side of the peak, characteristic of incomplete charge collection; its relative amplitude R and decay constant α are shared by all peaks in the spectrum. The third term is a step function (amplitude `step` relative to the peak) that accounts for the difference in the Compton continuum — the broad background left by gamma rays that deposited only part of their energy in the detector — on either side of the peak. By default $R = 0.1$, $\alpha = 0.9$ and $\mathrm{step} = 0$, and these three parameters are held fixed; the `skew_fit` and `step_fit` options of `Spectrum.fit_peaks()` add them to the fitted parameters instead.

5.1.2 Background

The continuum under the peaks is modeled in one of two ways, selected by the `bg` option:

- A polynomial — constant, linear or quadratic in channel number — fit jointly with the peaks in each multiplet (a group of peaks close enough on the spectrum that their fit windows overlap and they must be fit together).
- The default, a variant of the SNIP algorithm of Ryan *et al.* [Ryan1988], which estimates a smooth background non-parametrically and removes it from the fit entirely. The histogram is first compressed with the double-log operator $v = \ln(\ln(\sqrt{y+1}+1)+1)$; each point is then iteratively replaced by the minimum of itself and the mean of its neighbors at $\pm M w(x)$, where the half-width $M w(x)$ grows in ten steps to 7.5 times the local peak width $w(x)$ from the resolution calibration. Because the operator only ever lowers a point toward the average of its neighbors, structures narrower than the window (peaks) are clipped away while the smooth continuum is preserved. The result is transformed back, given a small positive safety margin (about $1.5\sqrt{B}$), and smoothed with a resolution-matched exponential filter. The `snip_adj` option scales the window width and margin.

The SNIP background is appropriate when the continuum varies smoothly under the peak; for peaks sitting on rapidly-varying features (e.g. the electron backscatter edge), a polynomial background will perform better.

5.1.3 Peak selection

The list of candidate peaks is generated from the gamma-ray lines of the isotopes assigned to the spectrum (plus any user-supplied lines), filtered by the criteria in `fit_config`: minimum energy (`E_min`), minimum intensity (`I_min`), proximity to the 511 keV annihilation line (`dE_511`), and predicted signal-to-noise ratio. The predicted SNR of a line is

$$\mathrm{SNR} = \frac{A_{\mathrm{pred}}}{\sqrt{B_{\mathrm{SNIP}}(\mu)}}$$

where the predicted amplitude A_{pred} is computed from the line intensity, the current efficiency calibration and the expected peak width, and $B_{\mathrm{SNIP}}(\mu)$ is the SNIP background at the expected centroid. Lines with $\mathrm{SNR} < \mathrm{SNR}_{\mathrm{min}}$ (default 4.0) are excluded. Lines whose fit windows overlap are fit together as a multiplet, up to `multi_max` peaks per fit; each fit spans `pk_width` (default 7.5) expected peak widths around the peaks.

5.1.4 Fitting and uncertainties

Each multiplet is fit by weighted least squares, minimizing

$$\chi^2 = \sum_i \frac{(y_i - F(x_i))^2}{y_i + 0.1}$$

i.e. with Poisson standard deviations as weights (the small constant keeps empty channels finite). Because these are true counting uncertainties, the parameter covariance — the matrix of the fitted parameters' variances and their correlations — is taken directly from the fit, without the customary rescaling by the reduced chi-square χ^2_ν (the χ^2 per degree of freedom, near one for a well-modeled fit) that would shrink the uncertainties of clean peaks below the floor set by counting statistics. When the reduced chi-square (evaluated over the non-empty channels) exceeds one, the covariance is inflated by χ^2_ν: an imperfect peak model can increase the uncertainties, but never reduce them.

The net counts in a peak are the analytic integrals of the Gaussian and skew terms of the fitted shape,

$$N_c = \sqrt{2\pi}\, A\sigma + 2\, R\, A\, \alpha\, \sigma\, e^{-\frac{1}{2\alpha^2}}$$

with uncertainty propagated from the full parameter covariance. The step term is part of the background and does not contribute counts.

The number of decays and the average decay rate during the count are

$$D = \frac{N_c}{I_\gamma\, \varepsilon(E)\, f_{\mathrm{corr}}\, (t_{\mathrm{live}}/t_{\mathrm{real}})} \qquad \bar{A} = \frac{D}{t_{\mathrm{real}}}$$

where I_γ is the gamma intensity (branching ratio), $\varepsilon(E)$ the peak efficiency, f_{corr} the product of any geometry and attenuation corrections, and $t_{\mathrm{live}}/t_{\mathrm{real}}$ the dead-time correction. The relative uncertainties of the counts, efficiency and intensity are combined in quadrature; the counting statistics of the peak area are already contained in the fitted covariance.

5.2 Detector Calibration

`Calibration.calibrate()` performs three sequential fits to the peaks of one or more spectra of reference sources with known activities: an energy calibration, a resolution calibration, and an efficiency calibration.

5.2.1 Energy calibration

The energy calibration maps ADC channel number to gamma-ray energy with a linear, quadratic or cubic polynomial,

$$E(x) = a_0 + a_1 x\ (+ a_2 x^2 + a_3 x^3)$$

fit by weighted least squares to the fitted peak centroids versus the known line energies, weighted by the centroid uncertainties. Points with energy uncertainties larger than a set fraction of the energy (`engcal_max_error`, default 25%) are excluded. The model is selected with `fit_config['engcal_model']`; by default the form of the current calibration is kept. The linear and quadratic forms invert analytically for the energy-to-channel map; the cubic is inverted numerically (the real root nearest the linear estimate), and a fitted cubic whose derivative changes sign inside the calibrated channel range is flagged as non-monotonic with a warning.

5.2.2 Resolution calibration

The peak width (the Gaussian σ, in channels) is modeled as one of three functions of channel number, selected with `fit_config['rescal_model']`:

$$\sigma(x) = b_0 + b_1 x \qquad \text{or} \qquad \sigma(x) = b_0\sqrt{x} \qquad \text{or} \qquad \sigma(x) = \sqrt{b_0 + b_1 x + b_2 x^2}$$

fit to the fitted peak widths. The square-root form (`'sqrt'`) is the expectation from pure counting statistics of charge-carrier generation; the linear form (`'linear'`, the default) accounts for the additional electronic-noise contribution typical of real HPGe systems. The square-root-of-quadratic form (`'sqrt_quad'`) is the modern Genie/InterSpec-family model: its three terms under the root map onto the physical width decomposition — constant electronic noise, charge-carrier (Fano) statistics growing linearly, and charge-collection variations growing quadratically — and it contains the `'sqrt'` form as the special case $b_0 = b_2 = 0$. Strongly discrepant points (residuals beyond `outlier_sigma`, default $\sqrt{10} \approx 3.16$ standard deviations) are clipped from the retained calibration data; they remain part of the fit itself and stay visible, with their reasons, in `cb.rescal_data` and the calibration plots.

5.2.3 Efficiency measurement

Each fitted peak of a reference source provides one measurement of the absolute peak efficiency at its energy. Accounting for the decay of the source during the count, the efficiency is

$$\varepsilon(E) = \frac{N_c\,\lambda}{f_{\mathrm{corr}}\,(1 - e^{-\lambda t_{\mathrm{real}}})\,e^{-\lambda t_d}\;I_\gamma\,A_0\,(t_{\mathrm{live}}/t_{\mathrm{real}})}$$

where A_0 is the reference activity, t_d the elapsed time from the reference date to the start of the count, λ the decay constant, and the remaining symbols are as defined above.

5.2.4 Efficiency model

The efficiency curve is a modified form of the physically founded efficiency model of Vidmar *et al.* [Vidmar2001], built from the tabulated photon interaction coefficients of germanium — total attenuation $\mu(E)$, photoelectric $\tau(E)$ and Compton $\sigma_C(E)$ — rather than from an arbitrary fitting function:

$$\varepsilon(E) = S\,\left(1 - e^{-\mu(E)L}\right)\,\frac{\tau(E) + \sigma_C(E)\,\left(1 - e^{-(\mu(E)L_0)^{\alpha}}\right)\kappa}{\mu(E)}$$

with five fitted parameters: a solid-angle scale S, an effective crystal length L, and three parameters (L_0, α, κ) describing the probability that a Compton-scattered photon is subsequently absorbed. The photoelectric term contributes full-energy events directly; the Compton term contributes only when the scattered photon is reabsorbed. When x-rays are included in the peak fits (`fit_config['xrays'] = True`), two low-energy attenuation factors extend the model to seven parameters,

$$\varepsilon_7(E) = e^{-\mu_{\mathrm{w}}(E)\,w}\;e^{-\mu(E)\,d}\;\varepsilon(E)$$

representing the detector entrance window (beryllium attenuation coefficient μ_{w}, thickness w) and the germanium dead layer (thickness d). Both the 5- and 7-parameter forms are fit, and the model whose goodness-of-fit is closer to one is kept (the choice is reported; `effcal_model='vidmar-5'` or `'vidmar-7'` forces one form). The interaction coefficients are log-log interpolations of the NIST XCOM photon cross-section tabulations.

With `fit_config['effcal_model'] = 'loglog'` the efficiency is instead the standard empirical log-log polynomial,

$$\ln \varepsilon(E) = \sum_{i=0}^{n} a_i \, (\ln E)^i$$

with the order selected by the model name (`'loglog'` is order 4; `'loglog-2'` through `'loglog-8'` choose it explicitly). High-order log-log polynomials reproduce germanium efficiency curves very well *within* the fitted energy range [Kis1998], but being polynomials they diverge rapidly outside it — unlike the semi-empirical model, whose physical form extrapolates smoothly. Curie stores the fitted energy range with the calibration and warns the first time the efficiency is evaluated beyond it. The Vidmar form remains the default and the recommended choice; use the log-log form when the semi-empirical model visibly misfits a particular detector. The fitted model is saved with the calibration .json as an explicit tag — necessary because a log-log parameter array can have the same length as a Vidmar one.

5.2.5 Efficiency uncertainties are correlated

The efficiency measurements from a calibration source are not independent: all points share the reference activity A_0 and decay constant λ of their source, and repeated measurements of the same gamma line (from multiple spectra) share that line's intensity uncertainty. The efficiency fit therefore uses the full covariance

$$V = V_{\mathrm{stat}} + V_{\mathrm{line}} + V_{\mathrm{src}}$$

where V_{stat} is diagonal (counting statistics), V_{line} couples measurements of the same line (intensity uncertainty), and V_{src} couples all measurements of the same source (activity and decay-constant uncertainty). The correlated terms are computed from the fitted curve rather than from the individual measurements, which keeps noisy points from dragging the shared uncertainties low.

If the reduced chi-square of the fit (computed with the full covariance V) exceeds one, the counting-statistics component V_{stat} is inflated by an iterated scale factor until $\chi^2_\nu \approx 1$. Scatter between the points is evidence about the point-to-point uncertainties, not about the shared ones, so only the statistical component is scaled — and, as in the peak fits, the scaling only ever inflates.

The parameter covariance of the efficiency fit is stored with the calibration, and `Calibration.unc_eff()` propagates it to any energy, so efficiencies interpolated from the calibration carry correlated, energy-dependent uncertainties.

5.3 Fit Reporting Conventions

Every fit in Curie — the peak fits, the three calibrations, and the decay-chain fits — reports its results, selections and problems through two channels that share one vocabulary: console messages and per-object diagnostics tables.

Console messages have the form `[LEVEL] Class.method: message`, with the instance identified where several may run in a loop (e.g. `Spectrum(<filename>).fit_peaks`). In order of escalation:

- `ERROR` — paired with a raised exception.
- `WARNING` — something that may affect results: failed fits, identical gammas fit with shared bounds, filters that matched nothing, extrapolation beyond a fitted range.
- `INFO` — routine accounting: fit summaries with their drop counts, model selections. Nothing at INFO changes a result.
- `DEBUG` — the per-item detail behind every summary count (each dropped candidate with its reason, each parameter at a fit bound).

The configured level is a threshold: everything at that level *and above* is shown, so the default `'INFO'` prints INFO, WARNING and ERROR messages, and `ci.set_log_level('DEBUG')` shows everything including the per-item detail. `ci.quiet_warnings()` restricts output to errors, and `ci.log_to('curie.log')` also writes the session's messages to a file — overwriting an existing file at that path (analysis scripts are typically re-run many times) unless `mode='append'` (or `'a'`) is given.

Diagnostics tables (`sp.diagnostics`, `cb.diagnostics`, `dc.diagnostics`) record one row per fit with a shared schema: the reduced chi-square and degrees of freedom, the points used and dropped, the model tag, the uncertainty scale factor applied (1.0 = none), a `flags` column from a fixed vocabulary — `at_bound:<param>` (a parameter ended on a fit bound), `chi2_high` (reduced chi-square above 10), `fit_failed`, `unmoved` (fit returned its starting estimate), `singular_cov` — and the full message text. The tables are rebuilt on each fit and reading them never triggers one.

Calibration point tables (`cb.engcal_data`, `cb.rescal_data`, `cb.effcal_data`) present every measured calibration point with a `used` column and the `reason` a point was rejected. Pre-fit cuts (`unc>25%`/`unc>33%`) exclude points from the fit; post-fit outlier clips (`outlier >3.16 sigma`) remove points from the *stored* calibration data but not from the fit that produced it. Nothing is silently discarded: rejected points stay in these tables and appear as grey open markers on the calibration plots.

Uncertainty scale factors follow one rule everywhere: when data are mutually inconsistent (reduced chi-square above one), only the independent (statistical) uncertainty component is inflated, iteratively, until the reduced chi-square (computed with the full covariance) is consistent with one — shared systematic components are never scaled, and no uncertainty is ever deflated. The applied factor is reported in the fit's summary line and diagnostics row.

5.4 Radioactive Decay Chains

5.4.1 The Bateman equations

A radioactive decay chain is governed by a system of coupled first-order differential equations: the number of atoms N_m of each member changes through its own decay, through the decay of its parents, and through any external production,

$$\frac{dN_m}{dt} = R_m(t) - \lambda_m N_m + \sum_j \mathrm{BR}_{j\to m}\,\lambda_j N_j$$

where λ_m is the decay constant, $R_m(t)$ the production rate (e.g. from a nuclear reaction during irradiation), and $\mathrm{BR}_{j\to m}$ the branching ratio for parent j decaying to m. `DecayChain` builds the system automatically: it starts from the parent isotope and follows every decay branch in the decay database — alpha, beta, electron capture, isomeric transition, spontaneous fission — until it reaches stable nuclei.

For a linear chain $1 \to 2 \to \cdots \to m$, the classic solution of Bateman [Bateman1910] expresses the activity $A_m = \lambda_m N_m$ as a sum of exponentials. For an initial activity $A_i(0)$ of member i,

$$A_m(t) = \sum_{i=1}^{m} A_i(0)\,\frac{\lambda_m}{\lambda_i}\left(\prod_{k=i}^{m-1}\mathrm{BR}_k\lambda_k\right)\sum_{j=i}^{m}\frac{e^{-\lambda_j t}}{\prod_{k\neq j}(\lambda_k-\lambda_j)}$$

and a constant production rate R_i into member i contributes

$$A_m(t) = R_i\,\lambda_m\left(\prod_{k=i}^{m-1}\mathrm{BR}_k\lambda_k\right)\sum_{j=i}^{m}\frac{1-e^{-\lambda_j t}}{\lambda_j\prod_{k\neq j}(\lambda_k-\lambda_j)}$$

which for a single isotope reduces to the saturation curve $A(t) = R\,(1 - e^{-\lambda t})$: during a constant irradiation the activity builds up from zero toward the saturation value R, where production and decay balance. When the decay graph branches, Curie enumerates every distinct path from the parent to the isotope of interest and sums the contributions, each weighted by its product of branching ratios.

Production schedules are piecewise constant: within each time interval the production rates are held fixed and the solution above is applied, with the activities at the end of one interval becoming the initial activities of the next. When a production schedule is given, time zero is the *end* of production (the end of bombardment, for activation experiments); for a chain specified only by initial activities, time zero is the moment those activities held. All subsequent times — counting intervals, activities — are measured from that point.

5.4.2 Numerical treatment

The Bateman denominators $\prod_{k \neq j}(\lambda_k - \lambda_j)$ diverge when two members of a chain have (nearly) equal decay constants — which genuinely happens in the decay database, where two half-lives can be identical as rounded. Rather than perturbing the values, Curie groups decay constants that agree to within one part in 10^9 and evaluates the exact *confluent* form of the solution for each group (the mathematical limit of the Bateman coefficients as the constants coincide), computed with a recursion that avoids the catastrophic cancellation of the raw sum. Each branch is additionally rescaled by the geometric mean of its decay constants — the solution is invariant under this rescaling — so that long chains with widely separated half-lives remain within floating-point range in any choice of time units.

5.4.3 Fitting to measured decays

`fit_R()` and `fit_A0()` adjust one scalar multiplier per produced isotope — scaling the production-rate history $R(t)$ (its shape is preserved) or the initial activity — to match the observed number of decays in each counting interval. The fit compares *decays per interval*, not instantaneous activities, so counts integrated over long measurements are treated exactly. Where the count data come from fitted spectra (`get_counts()`), each measurement's uncertainty is decomposed into an independent counting part, a part shared by all measurements of the same gamma line (its intensity), and a part shared by all measurements using the same efficiency calibration; the fit is a generalized least squares with the resulting covariance, using the same one-sided scale-factor convention as the calibration fits (see *Detector Calibration*).

5.5 Nuclear Reaction Data

5.5.1 The libraries

Curie distributes cross sections from four evaluated libraries:

Name	Evaluation	Particles	Organization	Uncertainties
`endf`	ENDF/B-VIII.1 [ENDF]	n	exclusive	no
`tendl`	TENDL-2025 [TENDL]	n	exclusive	no
`tendl_n/p/d/a`	TENDL-2025 [TENDL]	n, p, d, a	residual	no
`irdff`	IRDFF-II [IRDFF]	n	exclusive	yes
`iaea`	IAEA Medical Monitors (2025) [IAEA]	n, p, d, a, h, g	residual	yes

An *exclusive* library indexes reactions by their outgoing channel — (n,2n), (n,p), (n,inl) (inelastic scattering) — while a *residual-product* library indexes by what nucleus is produced, written (p,x), summing all routes. Independent of that split, a residual-product cross section may be *independent* (direct production of the state only — TENDL's convention, each isomer a separate entry) or *cumulative* (including decay feeding from short-lived co-produced parents — common in the IAEA monitor evaluations, which distinguish the two where both are useful). IRDFF-II is a dosimetry standard: most of its entries are exclusive channels, with a few indexed by product.

The TENDL residual-product libraries also carry *natural-element* targets (`natFe`, `natTi`, ...): abundance-weighted sums of the isotopic tables, computed when the databases are built from the same decay-data generation's abundances. Elements whose lightest isotopes TENDL does not evaluate (H, He, Li, Be) have no natural entry.

Energies are in MeV and cross sections in mb throughout (converted on retrieval from any source library that uses eV and barns). On retrieval, energy grids are sorted and exact duplicate points dropped; duplicated energy values that encode step discontinuities (e.g. threshold steps) are preserved for all libraries except TENDL, whose point-wise smooth model output makes a repeated energy value a build artifact, so there they are removed.

5.5.2 Interpolation and flux averages

`Reaction.interpolate()` selects its scheme per library, through the `Reaction.interp_config` parameter `'interpolation'`. The pointwise-linearized libraries (ENDF — reconstructed to a 0.5 % linearization tolerance — IRDFF and IAEA) default to `'linear'`, the convention their tabulated grids are generated for. The TENDL libraries, whose smooth model calculations sit on a coarse energy grid, default to `'pchip-sqrt'`: monotone piecewise-cubic (PCHIP) interpolation carried out in $\sqrt{E}$–$\sqrt{\sigma}$ space. The monotone construction passes exactly through the evaluated points and cannot overshoot or ring between them (a conventional quadratic or cubic spline can, badly, across a sharp threshold rise), and the square-root energy axis linearizes the $\sigma \propto \sqrt{E - E_{thr}}$ turn-on just above threshold. Interpolants are clamped non-negative, and below the reaction threshold the cross section is zero. **Outside the evaluated energy range the interpolated cross section is zero** — Curie never extrapolates evaluated data. Either scheme can be selected per reaction: `rx.interpolate(E, interpolation='linear')`.

Flux integrals treat the energy points as bin centers, with bin edges at the midpoints between them; the first and last points take the full spacing to their single neighbor:

$$\langle\sigma\rangle_\phi = \frac{\sum_i \sigma(E_i)\,\phi_i\,\Delta E_i}{\sum_i \phi_i\,\Delta E_i} \qquad \Delta E_i = \tfrac{1}{2}(E_{i+1} - E_{i-1}), \quad \Delta E_0 = E_1 - E_0, \quad \Delta E_{n-1} = E_{n-1} - E_{n-2}$$

For a uniform grid the widths cancel from the average, and histogram-like fluxes (such as those from `Stack`) are integrated without the endpoint under-weighting a trapezoidal rule would introduce. The cross-section uncertainty of an average is propagated as fully correlated between energies — uncertainties are summed linearly, not in quadrature — which is the conservative choice for evaluated curves, whose errors are dominated by common normalization rather than point-to-point scatter.

5.6 Stopping Power and Particle Transport

5.6.1 Stopping powers

A charged particle moving through matter loses energy continuously, at a rate characterized by the stopping power $S = -dE/dx$. Curie uses the semi-empirical parameterization of Andersen and Ziegler [AZ1977] [Z1977]: the electronic stopping power for protons in each element is a fitted piecewise form — a velocity-proportional regime at the lowest energies, an intermediate form joining smoothly to it, and a Bethe-like form with fitted corrections at high energy — with tabulated coefficients for every element up to uranium. Alpha particles have their own coefficient set; deuterons and tritons use the proton stopping evaluated at the same velocity (i.e. scaled energy E/M). For heavier ions at low-to-moderate velocity, the proton stopping is scaled by the square of an effective charge ratio (a Bohr/Northcliffe-type parameterization) that accounts for partial neutralization of the ion; above roughly 1 MeV per nucleon a relativistic Bethe-like form is used directly. A nuclear-stopping term, significant only at the lowest energies, is added in all cases.

The stopping power of a compound is the mass-weighted sum of its elemental stopping powers (Bragg additivity), which neglects the influence of chemical bonds on the electronic structure — typically a percent-level approximation, largest for light compounds at low energies. The range of a particle is the continuous-slowing-down integral

$$R(E_0) = \int_0^{E_0} \frac{dE}{S(E)}$$

5.6.2 Photon attenuation coefficients

`Element` and `Compound` also carry photon interaction data: the mass-attenuation coefficient μ/ρ and the mass energy-absorption coefficient μ_{en}/ρ are log-log interpolations of the NIST XCOM tabulations, with compound values likewise combined by mass-weighted additivity.

5.6.3 Transport through a stack

`Stack` propagates an ensemble of particles whose initial energies are Monte Carlo-sampled from a Gaussian of mean `E0` and width `dE0` (default 1% of `E0`); the transport itself is then deterministic, foil by foil in beam order. Within each foil the energy loss is integrated by a predictor–corrector (Heun) scheme: a trial step with the stopping power at the current energy, corrected by the average of the stopping powers at the start and trial energies. The number of steps is set so that the expected fractional energy loss per step is approximately the `accuracy` argument (default 0.01, i.e. about 1% of the current energy lost per step), clipped to the range `min_steps` to `max_steps` (default 2 to 50): a foil that barely degrades the beam is integrated in the minimum number of steps, a thick degrader in more. The energy distribution — the "flux" — assigned to a foil is the histogram of the ensemble's energies over all integration steps inside that foil, i.e. a path-averaged distribution through the foil's thickness, from which the reported mean energy `mu_E` and width `sig_E` are the first two moments. Because each integration step advances the beam by an equal increment of areal density, tallying the ensemble's energy at every step weights each energy by the path length the beam spends there. That path-length weighting is, up to normalization, the particle fluence — track length per unit volume — as a function of energy, which is precisely the spectral weight a reaction rate integrates over: convolving this distribution with an excitation function (the flux average of *Nuclear Reaction Data*) yields the effective cross section the foil actually experiences. `mu_E` is thus the fluence-weighted mean energy of the beam within the foil.

The width `sig_E` of a foil's distribution has two distinct physical origins. The first is the energy the beam loses between the front and back faces of the foil: because the histogram is accumulated over the entire path through the foil, a thick degrader that lowers the beam by several MeV produces a broad distribution however narrow the incident beam was, and for such a foil this within-foil loss — not the beam spread — dominates `sig_E`. The second is the incident beam spread `dE0` together with its growth as the ensemble slows: slower particles lose energy faster, so an initially narrow beam diverges in energy as it degrades. `sig_E` is therefore the physical range of energies the foil samples, and it is the energy uncertainty to attach to a cross section measured in that foil — not the statistical uncertainty of the mean `mu_E`. Collisional energy-loss straggling — the stochastic variance growth described by Bohr and Tschalar — is *not* added, so for very thick degraders the true energy spread is somewhat larger than computed. Particles are neither absorbed nor deflected: the distributions describe the beam's energy, not its intensity, and lateral spread is not modeled.

Because the beam is degraded foil by foil, uncertainties in `E0`, the areal densities, and the stopping powers accumulate down the stack, so the energy assigned to a deep foil can be off by more than its computed width. In stacked-target work this is corrected empirically by including monitor foils with independently evaluated cross sections and adjusting `E0` and the overall density multiplier `dp` until the beam current inferred from each monitor reaction varies smoothly along the stack. `Stack` does not perform this fit itself; it exposes `E0` and `dp` so the transport can be repeated inside a user-driven minimization.

5.7 Data Provenance and Integrity

Curie's nuclear data ship as pre-built databases, fetched on first use and verified against published SHA256 checksums (see *Installation*); the large cross-section libraries are fetched in per-target pieces. The reaction libraries are described in *Nuclear Reaction Data*. The decay data (half-lives, branching ratios, emission energies and intensities) are compiled from ENSDF via the IAEA LiveChart interface, with NUBASE2020/AME2020 masses and half-life closures and ENDF/B-VIII.1 fission yields (see *Data Sources* for full provenance); photon attenuation coefficients derive from the NIST XCOM tabulations; charged-particle stopping powers use the Andersen–Ziegler formulation. Gamma-ray energies are in keV in the spectroscopy classes, and every half-life is in seconds unless a unit argument says otherwise.

CHAPTER

SIX

DATA SOURCES

Curie's nuclear data ship as pre-built databases, fetched on first use from versioned data releases and verified against published SHA256 checksums. Each database carries a generation stamp; this page records what the current generation (v2, built July 2026) contains and where every number comes from. Connecting to data from an earlier generation logs a warning naming the `ci.download(...)` update command.

Database	Contents and source	Version
`decay`	Nuclear structure and decay data: level energies, half-lives, branching ratios, and radiation (gamma, X-ray, electron, alpha, beta) energies and intensities from ENSDF, retrieved through the IAEA LiveChart of Nuclides interface; atomic masses and half-life closures from AME2020/NUBASE2020; fission yields (independent and cumulative) from the ENDF/B-VIII.1 MF8 sublibraries; gamma–gamma coincidence probabilities from paceENSDF.	ENSDF via LiveChart (mirrored 2026-07); NUBASE2020/AME2020; ENDF/B-VIII.1
`endf`	Exclusive-channel neutron cross sections, reconstructed from the ENDF/B-VIII.1 neutron sublibrary (resonance reconstruction with NJOY2016 RECONR at a 0.5 % linearization tolerance).	ENDF/B-VIII.1 (2024)
`tendl`, `tendl_n/p/d/a`	TENDL exclusive-channel (neutron) and residual-product (n, p, d, alpha) cross sections, from the TALYS-computed TENDL-2025 tables. Targets are curated to the naturally occurring nuclides and states with half-lives of at least one year; natural-element targets are abundance-weighted sums of the isotopic tables. The proton library's Nb targets are carried from TENDL-2023 ENDF-6 files (no public host serves TENDL-2025 proton Nb).	TENDL-2025 (TALYS-2.1), CC BY 4.0, from https://tendl.imperial.ac.uk
`irdff`	The IRDFF-II neutron metrology (dosimetry) standard, with uncertainties, from the IAEA's tabulated distribution.	IRDFF-II (2020)
`iaea`	IAEA recommended charged-particle monitor reactions and medical isotope production cross sections, with uncertainties.	IAEA re-evaluation, site as of 2026-07
`ziegler`	Charged-particle stopping-power parameters (Andersen–Ziegler formulation), element densities, and NIST XCOM photon mass-attenuation coefficients.	carried over unchanged between data generations

6.1 Acknowledgements and citations

Curie's decay data are extracted from ENSDF through the IAEA Nuclear Data Section's *LiveChart of Nuclides* API; the fission-yield and neutron cross-section data derive from ENDF/B-VIII.1 (National Nuclear Data Center, Brookhaven National Laboratory); TENDL-2025 is distributed under CC BY 4.0 by the TALYS team; IRDFF-II and the medical monitor evaluations are published by the IAEA Nuclear Data Section; coincidence data derive from paceENSDF. Users publishing results that rest on a particular evaluation should cite that evaluation directly — the references collected in *Theory & Methodology* — in addition to Curie itself.

CHAPTER

SEVEN

API

The API reference documents Curie's functions, classes and class methods; follow the links below.

7.1 Isotope

class curie.**Isotope**(*isotope*)

Retrieve isotopic structure and decay data

The Isotope class provides isotope specific nuclear data, such as masses, abundances, half-lives, decay data and fission yields. The main data sources are ENSDF (via the IAEA LiveChart interface), NUBASE2020/AME2020, and the ENDF/B-VIII.1 fission-yield sublibraries.

Parameters

isotope
[str] Name of the isotope/isomer. For element El of mass AAA, and (optionally) isomeric state m# (where # is an integer starting with 1 for the first isomer) or g for ground state, the name must be formatted as either AAAELm# or El-AAAm# or EL-AAAm#. If the isomeric state isn't specified, the ground state is assumed, and if just m is given, the first isomer is assumed.

Attributes

element
[str] Elemental symbol, e.g. Xe, Ar, Kr

A
[int] Mass number A of isotope

isomer
[str] Isomeric state, e.g. g, m1, m2

name
[str] Formatted isotope name

E_level
[float] Energy level of the state, in MeV

J_pi
[str] Spin and partiy of the state

Z
[int] Atomic number Z of the isotope

N
[int] Neutron number N of the isotope

dc
[float] Isotope decay constant in units of 1/seconds

stable
[bool] `True` if isotope is stable

mass
[float] Atomic mass of the isotope in amu

abundance
[float] Natural abundance in percent.

unc_abundance
[float] Uncertainty in natural abundance in percent.

Delta
[float] Mass excess of the isotope in MeV

decay_products
[dict] Dictionary of decay products and their absolute branching ratios as `{'istp':BR}`

TeX
[str] LaTeX formatted isotope name

Methods

alphas([I_lim, E_lim])	Alpha-particles emitted by the decay of the isotope
beta_minus([I_lim, Endpoint_lim])	Electrons from beta-minus decay
beta_plus([I_lim, Endpoint_lim])	Positrons from beta-plus decay
decay_const([units, unc])	Decay constant of isotope
dose_rate([activity, distance, units])	Dose-rate from a point source of the isotope
electrons([I_lim, E_lim, CE_only, Auger_only])	Electrons (conversion and Auger) emitted by the decay of the isotope
gammas([I_lim, E_lim, xrays, dE_511, istp_col])	Gamma-rays emitted by the decay of the isotope
get_NFY(E)	Neutron induced fission yields
get_SFY()	Spontaneous fission yields
half_life([units, unc])	Half-life of isotope
optimum_units()	Appropriate units for half-life

Examples

```
>>> ip = ci.Isotope('115INm')
>>> ip = ci.Isotope('Co-60')
>>> ip = ci.Isotope('58NI')
>>> print(ip.abundance)
68.0769
```

```
>>> ip = ci.Isotope('135CEm')
>>> print(ip.dc)
0.03465735902799726
```

```
>>> ip = ci.Isotope('235U')
>>> print(ip.half_life('My'))
703.9849635622725
```

alphas(*I_lim=None*, *E_lim=None*)

Alpha-particles emitted by the decay of the isotope

Returns a DataFrame of alpha-particle energies, intensities and intensity-uncertainties based on an optional set of selection critieria.

Parameters

I_lim
[float or 2-tuple of floats, optional] Limits on the intensity of the alphas, in percent. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

E_lim
[float or 2-tuple of floats, optional] Limits on the energy of the alphas, in keV. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

Returns

alphas
[pd.DataFrame] Tabular alpha emission data, with keys 'energy', 'intensity' and 'unc_intensity'. Units of energy are in keV and units of intensity are in percent.

Examples

```
>>> ip = ci.Isotope('210PO')
>>> print(ip.alphas(I_lim=1.0))
    energy  intensity  unc_intensity
0  5304.33      100.0            NaN
```

beta_minus(*I_lim=None*, *Endpoint_lim=None*)

Electrons from beta-minus decay

Returns a DataFrame of beta-minus mean and enpoint energies, intensities and intensity-uncertainties based on an optional set of selection critieria.

Parameters

I_lim
[float or 2-tuple of floats, optional] Limits on the intensity of the beta-minus decay, in percent. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

Endpoint_lim
[float or 2-tuple of floats, optional] Limits on the endpoint energy of the beta-minus decay, in keV. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

Returns

beta_minus
[pd.DataFrame] Tabular beta-minus data, with keywords 'mean_energy', 'endpoint_energy', 'intensity', 'unc_intensity'. Units of energy are in keV and units of intensity are in percent.

Examples

```
>>> ip = ci.Isotope('35S')
>>> print(ip.beta_minus())
```

(continues on next page)

(continued from previous page)

```
   mean_energy  endpoint_energy  intensity  unc_intensity
0       48.758           167.33      100.0            0.0
```

beta_plus(*I_lim=None*, *Endpoint_lim=None*)

Positrons from beta-plus decay

Returns a DataFrame of beta-plus mean and enpoint energies, intensities and intensity-uncertainties based on an optional set of selection critieria.

Parameters

I_lim
[float or 2-tuple of floats, optional] Limits on the intensity of the beta-plus decay, in percent. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

Endpoint_lim
[float or 2-tuple of floats, optional] Limits on the endpoint energy of the beta-plus decay, in keV. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

Returns

beta_plus
[pd.DataFrame] Tabular beta-plus data, with keywords 'mean_energy', 'endpoint_energy', 'intensity', 'unc_intensity'. Units of energy are in keV and units of intensity are in percent.

Examples

```
>>> ip = ci.Isotope('18F')
>>> print(ip.beta_plus())
   mean_energy  endpoint_energy  intensity  unc_intensity
0        249.8            633.4      96.73           0.04
```

decay_const(*units='s'*, *unc=False*)

Decay constant of isotope

Returns isotope decay constant, in specified units, with or without uncertainty

Parameters

units
[str, optional] Units for which to return decay constant. Options are ns, us, ms, s, m, h, d, y, ky, My, Gy Actual value will be given in units of inverse time

unc
[str, optional] If `True`, uncertainty in decay constant (in specified units) will also be returned

Returns

decay_const
[float or 2-tuple of floats] Decay constant in specified units, with uncertainty only if `unc=True`

Examples

```
>>> ip = ci.Isotope('221AT')
>>> print(ip.decay_const())
```

(continues on next page)

(continued from previous page)

```
0.0050228056562314875
>>> print(ip.decay_const('m', True))
(0.3013683393738893, 0.026205942554251245)
```

dose_rate(*activity=1.0*, *distance=30.0*, *units='R/hr'*)

Dose-rate from a point source of the isotope

Assuming a point source with no attenuation, this function calculates the dose-rate from each emitted particle type using a specified source activity and distance.

Parameters

activity
[float, optional] Source activity in Bq. Default is 1.0

distance
[float, optional] Source distance in cm. Default is 30.0

units
[str, optional] Desired units of dose rate. Format: `dose/time` where dose is one of R (Roentgen), Sv, Gy, with the Si prefixes u, m, k, and M supported, and time is in the same format described under the `Isotope.half_life()` function. E.g. mR/hr, uSv/m, kGy/y. Default is 'R/hr'

Returns

dose_rate
[dict] Dictionary of dose from each of the following particle type: 'gamma', 'electron', 'beta_minus', 'beta_plus', 'alpha', and 'total'. Units are specified as an input.

Examples

```
>>> ip = ci.Isotope('Co-60')
>>> print(ip.dose_rate(activity=3.7E10, units='R/hr')) # 1 Ci of Co-60 at 30 cm
{'gammas': 14.454, 'alphas': 0.0, 'beta_minus': 1545.1658,
'beta_plus': 0.0, 'electrons': 17.032, 'total': 1576.6518}
>>> print(ip.dose_rate(activity=3.7E10, units='uSv/hr')) # the same, in uSv/hr
{'gammas': 126761.9304, 'alphas': 0.0, 'beta_minus': 13551103.67,
'beta_plus': 0.0, 'electrons': 149370.8364, 'total': 13827236.4368}
```

electrons(*I_lim=None*, *E_lim=None*, *CE_only=False*, *Auger_only=False*)

Electrons (conversion and Auger) emitted by the decay of the isotope

Returns a DataFrame of electron energies, intensities and intensity-uncertainties based on an optional set of selection critieria.

Parameters

I_lim
[float or 2-tuple of floats, optional] Limits on the intensity of the electrons, in percent. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

E_lim
[float or 2-tuple of floats, optional] Limits on the energies of the electrons, in keV. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

CE_only
[bool] If `True`, only conversion electrons will be returned. Default `False`

Auger_only
[bool] If `True`, only Auger electrons will be returned. Default `False`.

Returns

electrons
[pd.DataFrame] Tabular electron data, with keywords 'energy', 'intensity', 'unc_intensity'. Units of energy are in keV and units of intensity are in percent.

Examples

```
>>> ip = ci.Isotope('Pt-193m')
>>> print(ip.electrons(I_lim=5.0, E_lim=(10.0, 130.0)))
    energy  intensity  unc_intensity
0   11.907       17.1            1.0
1   57.101       15.5            0.3
2  121.617       60.0            0.8
```

gammas(*I_lim=None*, *E_lim=None*, *xrays=False*, *dE_511=0.0*, *istp_col=False*)

Gamma-rays emitted by the decay of the isotope

Returns a DataFrame of gamma-ray energies, intensities and intensity-uncertainties based on an optional set of selection critieria.

Parameters

I_lim
[float or 2-tuple of floats, optional] Limits on the intensity of the gamma-rays, in percent. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

E_lim
[float or 2-tuple of floats, optional] Limits on the energy of the gamma-rays, in keV. If a single float is given, this is a lower bound, if a 2-tuple then (upper, lower) bounds.

xrays
[bool, optional] If `True`, retrieved data will include x-rays. Default `False`.

dE_511
[float, optional] Filter out gamma-rays that are within `dE_511` keV of the annihilation peak. Default 0.0.

Returns

gammas
[pd.DataFrame] Tabular gamma-ray data, with keys 'energy', 'intensity' and 'unc_intensity'. Units of energy are in keV and units of intensity are in percent.

Examples

```
>>> ip = ci.Isotope('Co-60')
>>> print(ip.gammas(I_lim=1.0))
     energy  intensity  unc_intensity
0  1173.228    99.8500         0.0300
1  1332.492    99.9826         0.0006
```

```
>>> ip = ci.Isotope('64CU')
>>> print(ip.gammas())
    energy  intensity  unc_intensity
```

(continues on next page)

(continued from previous page)

```
0   511.00     34.980           NaN
1  1345.77      0.472         0.004
>>> print(ip.gammas(xrays=True, dE_511=1.0))
     energy  intensity  unc_intensity
0     0.874   0.491121       0.023528
1     7.461   4.896772       0.058365
2     7.478   9.556543       0.103925
3     8.296   1.991667       0.029457
4  1345.770   0.472000       0.004000
```

get_NFY(*E*)

Neutron induced fission yields

Returns the absolute neutron induced fission yields (independent) for a given nuclide, where available from ENDF.

Parameters

E
[float] Incident neutron energy, in eV. Available energies are 0.0253, 5E5, 2E6 and 14E6.

Returns

nfy
[pd.DataFrame] Tabular neutron induced fission yields, with keys 'daughter', 'yield' and 'unc_yield'

Examples

```
>>> ip = ci.Isotope('235U')
>>> print(ip.get_NFY(E=0.0253))
```

get_SFY()

Spontaneous fission yields

Returns the absolute spontaneous fission yields (independent) for a given nuclide, where available from ENDF.

Returns

sfy
[pd.DataFrame] Tabular spontaneous fission yields, with keys 'daughter', 'yield' and 'unc_yield'

Examples

```
>>> ip = ci.Isotope('238U')
>>> print(ip.get_SFY())
```

half_life(*units='s'*, *unc=False*)

Half-life of isotope

Returns isotope half-life, in specified units, with or without uncertainty.

Parameters

units
[str, optional] Units for which to return half-life. Options are ns, us, ms, s, m, h, d, y, ky, My, Gy

unc
[bool, optional] If `True`, uncertainty in half life (in specified units) will also be returned

Returns

half_life
[float or 2-tuple of floats] Half-life in specified units, with uncertainty only if `unc=True`

Examples

```
>>> ip = ci.Isotope('67CU')
>>> print(ip.half_life('d'))
2.57625
>>> print(ip.half_life('h', True))
(61.83, 0.12)
```

`optimum_units()`

Appropriate units for half-life

Returns

units
[str] Units that will represent half-life as a number greater than 1, but with as few digits as possible.

Examples

```
>>> ip = ci.Isotope('226RA')
>>> print(ip.half_life())
50491081559.3472
>>> print(ip.optimum_units())
y
>>> print(ip.half_life(ip.optimum_units()))
1599.965826277892
```

7.2 Element

`class curie.Element(element)`

Elemental data and properties

The Element class provides useful data about the natural elements, such as mass, density, and isotopic composition. Additionally, it contains functions for determining the interaction of radiation with the natural elements. Principally it provides the mass-attenuation of photons, and the stopping power/ranges of charged particles.

Parameters

element
[str] Symbol for the element, e.g. 'H', 'In', 'Zn', 'Fe'. Case insensitive. Note that 'n' ("neutron") is not considered a valid element in this context, and will be interpreted as 'N' ("nitrogen").

Attributes

name
[str] Symbol for the element, in title-case. E.g. if input was 'fe', name will be 'Fe'.

Z
[int] Atomic number of the element.

mass
[float] Molar mass of the natural element in atomic mass units (amu).

isotopes
[list] Isotopes with non-zero natural abundance, e.g.

abundances
[pd.DataFrame] Natural abundances of the element's isotopes

density
[float] Density of the natural element in g/cm^3. The density is used in calculations of charged particle dEdx and photon attenuation, so you can assign a new density using `el.density = new_density` if needed, or using the `density` keyword in either of those functions.

mass_coeff
[pd.DataFrame] Table of mass-attenuation coefficients as a function of photon energy, from the NIST XCOM database. Energies are in keV, and mass-attenuation coefficients, or mu/rho, are given in cm^2/g. DataFrame columns are 'energy', 'mu' and 'mu_en' for the mass-energy absorption coefficient.

Methods

S(energy[, particle, density])	Charged particle stopping power in matter
attenuation(energy, x[, density])	Photon attenuation in matter
mu(energy)	Mass-attenuation coefficient
mu_en(energy)	Mass energy-absorption coefficient
plot_S([particle, energy])	Plot the stopping power in the element
plot_mass_coeff([energy])	Plot the mass-attenuation coefficient in the element
plot_mass_coeff_en([energy])	Plot the mass energy-absorption coefficient in the element
plot_range([particle, energy, density])	Plot the charged particle range in the element
range(energy[, particle, density])	Charged particle range in matter

Examples

```
>>> el = ci.Element('Fe')
>>> print(el.mass)
55.847
>>> print(el.density)
7.866
```

S(*energy*, *particle='p'*, *density=None*)

Charged particle stopping power in matter

Calculate the stopping power, S=-dE/dx, for a given ion as a function of the ion energy in MeV. Units of S are MeV/cm. To return stopping power in units of MeV/(mg/cm^2), use option `density=1E-3`.

Parameters

energy
[array_like] Incident ion energy in MeV.

particle
[str, optional] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'. For light ions, the charge state is assumed to be fully stripped. For heavy ions the charge state is handled by a Bohr/Northcliffe parameterization consistent with the Andersen-Ziegler formalism.

density
[float, optional] Density of the element in g/cm^3. Default behavior is to use `Element.density`. To return stopping power in units of MeV/(mg/cm^2), i.e. the mass-stopping power, use `density=1E-3`.

Returns

stopping_power
[numpy.ndarray] Stopping power, S=-dE/dx, for a given ion as a function of the ion energy in MeV. Units of S are MeV/cm.

Examples

```
>>> el = ci.Element('La')
>>> print(round(el.S(60.0), 6))
36.868775
>>> print(round(el.S(55.0, density=1E-3), 6)) ### S in MeV/(mg/cm^2)
0.006372
```

`property abundances`

Natural abundances of the element's isotopes

pd.DataFrame with columns 'isotope', 'abundance' (%) and 'unc_abundance'. Loaded from the decay library on first access.

`attenuation`(*energy*, *x*, *density=None*)

Photon attenuation in matter

Calculate the attenuation factor I(x)/I_0 = e^(-mu*x) for a given photon energy (in keV) and slab thickness (in cm).

Parameters

energy
[array_like] Incident photon energy in keV.

x
[float] Thickness of slab of given element, in cm.

density
[float, optional] Density of the element in g/cm^3. Default behavior is to use `Element.density`.

Returns

attenuation
[numpy.ndarray] The slab attenuation factor as an absolute number (i.e. from 0 to 1). E.g. if the incident intensity is I_0, the transmitted intensity I(x) is I_0 times the attenuation factor.

Examples

```
>>> el = ci.Element('Fe')
>>> print(el.attenuation(511, x=0.3))
0.821621630674751
>>> print(el.attenuation(300, x=0.5, density=8))
0.6442940871813587
```

property isotopes

Isotopes with non-zero natural abundance, e.g. [‘54FE’, ‘56FE’, ...]

mu(*energy*)

Mass-attenuation coefficient

Interpolates the mass-attenuation coefficient, mu/rho, for the element along the input energy grid.

Parameters

energy
[array_like] The incident photon energy, in keV.

Returns

mu
[np.ndarray] Mass attenuation coefficient, mu/rho, in cm^2/g.

Examples

```
>>> el = ci.Element('Hg')
>>> print(round(el.mu(200), 6))
0.9456
```

mu_en(*energy*)

Mass energy-absorption coefficient

Interpolates the mass-energy absorption coefficient, mu_en/rho, for the element along the input energy grid.

Parameters

energy
[array_like] The incident photon energy, in keV.

Returns

mu_en
[np.ndarray] Mass energy absorption coefficient, mu_en/rho, in cm^2/g.

Examples

```
>>> el = ci.Element('Hg')
>>> print(round(el.mu_en(200), 6))
0.5661
```

plot_S(*particle='p'*, *energy=None*, ***kwargs*)

Plot the stopping power in the element

Creates a plot of the charged particle stopping power (in MeV/(mg/cm^2)) in the element as a function of the incident ion energy (in MeV).

Parameters

particle
[str] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'.

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in MeV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> el = ci.Element('He')
>>> el.plot_S(particle='a')
>>> el = ci.Element('Fe')
>>> el.plot_S(particle='d')
```

plot_mass_coeff(*energy=None*, ***kwargs*)

Plot the mass-attenuation coefficient in the element

Creates a plot of the mass-attenuation coefficient (in cm^2/g) as a function of photon energy in keV.

Parameters

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in keV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> el = ci.Element('Fe')
>>> el.plot_mass_coeff()
>>> el.plot_mass_coeff(style='poster')
```

plot_mass_coeff_en(*energy=None*, ***kwargs*)

Plot the mass energy-absorption coefficient in the element

Creates a plot of the mass energy-absorption coefficient (in cm^2/g) as a function of photon energy in keV.

Parameters

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in keV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> el = ci.Element('Hf')
```

Example plotting the mass-attenuation coefficient together with the mass energy-absorption coefficient, on the same axes.

```
>>> f,ax = el.plot_mass_coeff(return_plot=True)
>>> el.plot_mass_coeff_en(f=f, ax=ax)
```

plot_range(*particle='p'*, *energy=None*, *density=None*, ***kwargs*)

Plot the charged particle range in the element

Creates a plot of the charged particle range (in cm) in the element as a function of the incident ion energy (in MeV).

Parameters

particle
[str] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'.

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in MeV.

density
[float, optional] Density of the element in g/cm^3. Default behavior is to use `Element.density`.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> el = ci.Element('Ar')
>>> el.plot_range()
>>> el.plot_range(density=0.5)
```

range(*energy*, *particle='p'*, *density=None*)

Charged particle range in matter

Calculates the charged particle range in the element, in cm. Incident energy should be in MeV, and the particle type definition is identical to `Element.S()`.

Parameters

energy
[array_like] Incident ion energy in MeV.

particle
[str, optional] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'. For light ions, the charge state is assumed to be fully stripped. For heavy ions the charge state is handled by a Bohr/Northcliffe parameterization consistent with the Andersen-Ziegler formalism.

density
[float, optional] Density of the element in g/cm^3. Default behavior is to use `Element.density`.

Returns

range
[np.ndarray] Charged particle range in the element, in cm.

Examples

```
>>> el = ci.Element('Fe')
>>> print(round(float(el.range(60.0)), 6))
0.586631
>>> el = ci.Element('U')
>>> print(round(float(el.range(60.0)), 6))
0.377021
```

7.3 Compound

class curie.**Compound**(*compound*, *weights=None*, *density=None*)

Data and properties of atomic compounds

The compound class provides the same functionality as the element class, but for compounds of atomic elements rather than the individual atomic elements. The compound is described by a set of elements, a set of weights for each element, and a density.

The weights can be either given as atom-weights, e.g. in H2O the atom weights are 0.67 for H and 0.33 for O, or as mass-weights, e.g. brass is 0.33 Zn by weight and 0.67 Cu by weight. Some preset compounds are available; their names can be found by printing `ci.COMPOUND_LIST`.

Parameters

compound
[str] The name of the compound. If the compound is in `ci.COMPOUND_LIST`, the weights and density will take preset values, if not given. If the name of the compound is given in chemical notation, e.g. NaCl or H2O2, the atom-weights can be inferred if not given explicitly. Note that full chemical notation is not supported, so Ca3(PO4)2 must be written as Ca3P2O8. Decimal weights are supported, e.g. C0.5O is equivalent to CO2.

weights
[dict, str or pd.DataFrame, optional] The weights of each element in the compound. Multiple formats are supported. If weights is a dict, it must be formatted as {'el1':wt1, 'el2':wt2}, where atom-weights are positive, and mass-weights are negative.

If weights is a pandas DataFrame, it must contain an 'element' column, and one of 'weight', 'atom_weight', or 'mass_weight'. If 'weight' is the column given, the convention of positive atom-weights and negative mass-weights is followed.

If weights is a str, it can either be formatted as 'el1:wt1, el2:wt2', or it can be a path to a .csv, .json or .db file. These files must contain the same information as the DataFrame option, and can contain weights for multiple compounds, if 'compound' is one of the columns/keys.

If a .json file, it must follow the 'records' formatting convention (see pandas docs).

density
[float, optional] Density of the compound in g/cm^3. The density is required for the cm.attenuation(), cm.S(), cm.range() and cm.plot_range() functions, but is an optional argument in each of those functions if not provided at construction. Can also be specified by using a 'density' column/key in the file/DataFrame for weights.

Attributes

name
[str] The name of the compound.

weights
[pd.DataFrame] The weights for each element in the compound. DataFrame columns are 'element', 'Z', 'mass_weight', 'atom_weight'.

density
[float] Density of the compound in g/cm^3. The density is used in calculations of charged particle dEdx and photon attenuation, so if the density was not explicitly given at construction, you can assign a new density using `cm.density = new_density` if needed, or using the `density` keyword in either of those functions.

elements
[list of ci.Element] Elements in the compound.

mass_coeff
[pd.DataFrame] Table of mass-attenuation coefficients as a function of photon energy, from the NIST XCOM database. Energies are in keV, and mass-attenuation coefficients, or mu/rho, are given in cm^2/g. DataFrame columns are 'energy', 'mu' and 'mu_en' for the mass-energy absorption coefficient.

Methods

S(energy[, particle, density])	Charged particle stopping power in matter
attenuation(energy, x[, density])	Photon attenuation in matter
mu(energy)	Mass-attenuation coefficient
mu_en(energy)	Mass energy-absorption coefficient
plot_S([particle, energy])	Plot the stopping power in the compound
plot_mass_coeff([energy])	Plot the mass-attenuation coefficient in the compound
plot_mass_coeff_en([energy])	Plot the mass energy-absorption coefficient in the compound
plot_range([particle, energy, density])	Plot the charged particle range in the compound
range(energy[, particle, density])	Charged particle range in matter
saveas(filename[, replace])	Save the compound definition to a file

Examples

```
>>> print('Silicone' in ci.COMPOUND_LIST)
True
>>> cm = ci.Compound('Silicone') # preset compound
>>> print(list(map(str, cm.elements)))
```

(continues on next page)

(continued from previous page)

```
['H', 'C', 'O', 'Si']
>>> cm = ci.Compound('H2O', density=1.0)
>>> print(cm.weights)
  element  Z  atom_weight  mass_weight
0       H  1     0.666667     0.111907
1       O  8     0.333333     0.888093
>>> cm = ci.Compound('Brass', weights={'Zn':-33,'Cu':-66})
>>> print(cm.weights)
  element   Z  atom_weight  mass_weight
0      Zn  30     0.327041     0.333333
1      Cu  29     0.672959     0.666667
>>> cm.saveas('brass.csv')
```

S(*energy*, *particle='p'*, *density=None*)

Charged particle stopping power in matter

Calculate the stopping power, S=-dE/dx, for a given ion as a function of the ion energy in MeV. Units of S are MeV/cm. To return stopping power in units of MeV/(mg/cm^2), use option `density=1E-3`. The stopping power is calculated using the Element.S() methods for each element in cm.elements, added using Bragg's rule.

Parameters

energy
[array_like] Incident ion energy in MeV.

particle
[str, optional] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'. For light ions, the charge state is assumed to be fully stripped. For heavy ions the charge state is handled by a Bohr/Northcliffe parameterization consistent with the Andersen-Ziegler formalism.

density
[float, optional] Density of the compound in g/cm^3. Default behavior is to use `Compound.density`. To return stopping power in units of MeV/(mg/cm^2), i.e. the mass-stopping power, use `density=1E-3`.

Returns

stopping_power
[numpy.ndarray] Stopping power, S=-dE/dx, for a given ion as a function of the ion energy in MeV. Units of S are MeV/cm.

Examples

```
>>> cm = ci.Compound('SrCO3', density=3.5)
>>> print(round(cm.S(60.0), 6))
27.196387
>>> print(round(cm.S(55.0, density=1E-3), 6)) ### S in MeV/(mg/cm^2)
0.008308
```

attenuation(*energy*, *x*, *density=None*)

Photon attenuation in matter

Calculate the attenuation factor I(x)/I_0 = e^(-mu*x) for a given photon energy (in keV) and slab thickness (in cm).

Parameters

energy
[array_like] Incident photon energy in keV.

x
[float] Thickness of slab of given compound, in cm.

density
[float, optional] Density of the compound in g/cm^3. Default behavior is to use `Compound.density`, which must be supplied at construction.

Returns

attenuation
[numpy.ndarray] The slab attenuation factor as an absolute number (i.e. from 0 to 1). E.g. if the incident intensity is I_0, the transmitted intensity I(x) is I_0 times the attenuation factor.

Examples

```
>>> cm = ci.Compound('SS_316') # preset compound for 316 Stainless
>>> print(cm.attenuation(511, x=0.3))
0.8199829388434694
>>> print(cm.attenuation(300, x=1.0, density=5.0))
0.5752140388004373
```

mu(*energy*)

Mass-attenuation coefficient

Interpolates the mass-attenuation coefficient, mu/rho, for the compound along the input energy grid.

Parameters

energy
[array_like] The incident photon energy, in keV.

Returns

mu
[np.ndarray] Mass attenuation coefficient, mu/rho, in cm^2/g.

Examples

```
>>> cm = ci.Compound('H2O')
>>> print(round(cm.mu(200), 6))
0.137039
```

mu_en(*energy*)

Mass energy-absorption coefficient

Interpolates the mass-energy absorption coefficient, mu_en/rho, for the compound along the input energy grid.

Parameters

energy
[array_like] The incident photon energy, in keV.

Returns

mu_en
[np.ndarray] Mass energy absorption coefficient, mu_en/rho, in cm^2/g.

Examples

```
>>> cm = ci.Compound('H2O')
>>> print(round(cm.mu_en(200), 6))
0.029672
```

plot_S(*particle='p'*, *energy=None*, ***kwargs*)

Plot the stopping power in the compound

Creates a plot of the charged particle stopping power (in MeV/(mg/cm^2)) in the compound as a function of the incident ion energy (in MeV).

Parameters

particle
[str] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'.

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in MeV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> cm = ci.Compound('He') # same as element
>>> cm.plot_S(particle='a')
>>> cm = ci.Compound('Kapton')
>>> cm.plot_S(particle='d')
```

plot_mass_coeff(*energy=None*, ***kwargs*)

Plot the mass-attenuation coefficient in the compound

Creates a plot of the mass-attenuation coefficient (in cm^2/g) as a function of photon energy in keV.

Parameters

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in keV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> cm = ci.Compound('Fe')
>>> cm.plot_mass_coeff()
>>> cm = ci.Compound('H2O')
>>> cm.plot_mass_coeff(style='poster')
```

plot_mass_coeff_en(*energy=None*, ***kwargs*)

Plot the mass energy-absorption coefficient in the compound

Creates a plot of the mass energy-absorption coefficient (in cm^2/g) as a function of photon energy in keV.

Parameters

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in keV.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> cm = ci.Compound('Silicone') # preset compound
```

Example plotting the mass-attenuation coefficient together with the mass energy-absorption coefficient, on the same axes.

```
>>> f,ax = cm.plot_mass_coeff(return_plot=True)
>>> cm.plot_mass_coeff_en(f=f, ax=ax)
```

plot_range(*particle='p'*, *energy=None*, *density=None*, ***kwargs*)

Plot the charged particle range in the compound

Creates a plot of the charged particle range (in cm) in the compound as a function of the incident ion energy (in MeV).

Parameters

particle
[str] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'.

energy
[array_like, optional] Energy grid on which to plot, replacing the default energy grid. Units are in MeV.

density
[float, optional] Density of the compound in g/cm^3. Default behavior is to use `Compound.density`.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> cm = ci.Compound('Bronze', weights={'Cu':-80, 'Sn':-20}, density=8.9)
>>> f,ax = cm.plot_range(return_plot=True)
>>> cm.plot_range(particle='d', f=f, ax=ax)
```

range(*energy*, *particle='p'*, *density=None*)

Charged particle range in matter

Calculates the charged particle range in the compound, in cm. Incident energy should be in MeV, and the particle type definition is identical to `Compound.S()`.

Parameters

energy
[array_like] Incident ion energy in MeV.

particle
[str, optional] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'. For light ions, the charge state is assumed to be fully stripped. For heavy ions the charge state is handled by a Bohr/Northcliffe parameterization consistent with the Andersen-Ziegler formalism.

density
[float, optional] Density of the compound in g/cm^3. Default behavior is to use `Compound.density`, which must be supplied at construction.

Returns

range
[np.ndarray] Charged particle range in the compound, in cm.

Examples

```
>>> cm = ci.Compound('Fe') # same behavior as element
>>> print(round(float(cm.range(60.0)), 6))
0.586631
>>> cm = ci.Compound('SS_316') # preset compound
>>> print(round(float(cm.range(60.0)), 6))
0.580735
```

saveas(*filename*, *replace=False*)

Save the compound definition to a file

The weights and density of the compound can be saved to one of the following file formats: .csv, .json, .db. If the file exists, the data will be appended, unless `replace=True`, in which case the file will be replaced. If a definition for the compound exists in the file already, it will be replaced.

Parameters

filename
[str] Filename where the compound will be saved. Available formats are .csv, .json and .db.

replace
[bool, optional] If `True`, replace the file if it exists. Default `False`, which appends the data to the file.

Examples

```
>>> cm = ci.Compound('Brass', weights={'Zn':-33,'Cu':-66})
>>> cm.saveas('brass.csv')
>>> cm = ci.Compound('Water', weights={'H':2, 'O':1}, density=1.0)
>>> cm.saveas('water.json')
```

7.4 Stack

class curie.**Stack**(*stack*, *particle='p'*, *E0=60.0*, ***kwargs*)

Foil pack for stacked target calculations

Computes the energy loss and (relative) charged particle flux through a stack of foils using the Andersen-Ziegler formulation for stopping powers.

Parameters

stack
[list of dicts, pd.DataFrame or str] Definition of the foils in the stack. The 'compound' for each foil in the stack must be given, and the 'areal_density' or some combination of parameters that allow the areal density to be calculated must also be given. Foils must also be given a 'name' if they are to be filtered by the .saveas(), .summarize(), and .plot() methods. By default, foils without 'name' are not included by these methods.

There are three acceptable formats for `stack`. The first is a pd.DataFrame with the columns described. The second is a list of dicts, where each dict contains the appropriate keys. The last is a str, which is a path to a file in either .csv, .json or .db format, where the headers of the file contain the correct information. Note that the .json file must follow the 'records' format (see pandas docs). If a .db file, it must have a table named 'stack'.

The 'areal_density' can be given directly, in units of mg/cm^2, or will be calculated from the following: 'mass' (in g) and 'area' (in cm^2), 'thickness' (in mm) and 'density' (in g/cm^3), or just 'thickness' if the compound is a natural element, or is in `ci.COMPOUND_LIST` or the 'compounds' argument.

Also, the following shorthand indices are supported: 'cm' for 'compound', 'd' for 'density', 't' for 'thickness', 'm' for 'mass', 'a' for 'area', 'ad' for 'areal_density', and 'nm' for 'name'.

particle
[str] Incident ion. For light ions, options are 'p' (default), 'd', 't', 'a' for proton, deuteron, triton and alpha, respectively. Additionally, heavy ions can be specified either by element or isotope, e.g. 'Fe', '40CA', 'U', 'Bi-209'.For light ions, the charge state is assumed to be fully stripped. For heavy ions the charge state is handled by a Bohr/Northcliffe parameterization consistent with the Andersen-Ziegler formalism.

E0
[float] Incident particle energy, in MeV. If dE0 is not provided, it will default to 1 percent of E0.

Attributes

stack
[pandas.DataFrame] 'name', 'compound', 'areal_density', mean energy 'mu_E', and 1-sigma energy width 'sig_E' for each foil in the stack (energies in MeV).

fluxes
[pandas.DataFrame] 'flux' as a function of 'energy' for each foil in the stack where 'name' is not None.

compounds
[dict] Dictionary with compound names as keys, and ci.Compound classes as values.

Methods

get_flux(sample_name)	Returns the computed energy grid and flux for a sample
plot([filter_name])	Plots the fluxes for each foil in the stack calculation
saveas(filename[, save_fluxes, filter_name])	Saves the results of the stack calculation
summarize([filter_name])	Summarize the stack calculation

Other Parameters

compounds
[str, pandas.DataFrame, list or dict] Compound definitions for the compounds included in the foil stack. If the compounds are not natural elements, or `ci.COMPOUND_LIST`, or if different weights or densities are required, they can be specified here. (Note specifying specific densities in the 'stack' argument is probably more appropriate.) Also, if the 'compound' name in the stack is a chemical formula, e.g. 'H2O', 'SrCO3', the weights can be inferred and 'compounds' doesn't need to be given.

If compounds is a pandas DataFrame, it must have the columns 'compound', 'element', one of 'weight', 'atom_weight', or 'mass_weight', and optionally 'density'. If a str, it must be a path to a .csv, .json or .db file, where .json files must be in the 'records' format and .db files must have a 'compounds' table. All must have the above information. For .csv files, the compound only needs to be given for the first line of that compound definition.

If compounds is a list, it must be a list of ci.Element or ci.Compound classes. If it is a dict, it must have the compound names as keys, and weights as values, e.g. {'Water':{'H':2, 'O':1}, 'Brass':{'Cu':-66,'Zn':-33}}

dE0
[float] 1-sigma width of the energy distribution from which the initial particle energies are sampled, in MeV. Default is to 1 percent of E0.

N
[int] Number of particles to simulate. Default is 10000.

dp
[float] Density multiplier. dp is uniformly multiplied to all areal densities in the stack. Default 1.0.

chunk_size
[int] If N is large, split the stack calculation in to multiple "chunks" of size `chunk_size`. Default 1E7.

accuracy
[float] Target fractional energy loss per step in the predictor-corrector method. Default 0.01. Each foil is solved in a number of steps equal to its expected fractional energy loss divided by `accuracy`, bounded between `min_steps` and `max_steps`.

min_steps
[int] The minimum number of steps per foil, in the predictor-corrector solver. Default 2.

max_steps
[int] The maximum number of steps per foil, in the predictor-corrector solver. Default 50.

Examples

```
>>> stack = [{'cm':'H2O', 'ad':800.0, 'name':'water'},
                        {'cm':'RbCl', 'density':3.0, 't':0.03, 'name':'salt'},
                        {'cm':'Kapton', 't':0.025},
                        {'cm':'Brass', 'm':3.5, 'a':1.0, 'name':'metal'}]
>>> st = ci.Stack(stack, compounds='example_compounds.json')
>>> st = ci.Stack(stack, compounds={'Brass':{'Cu':-66, 'Zn':-33}}, E0=60.0)
>>> print(st.stack)
    name compound  areal_density       mu_E     sig_E
0  water      H2O         800.00  55.444815  2.935233
1   salt     RbCl           9.00  50.668313  0.683532
2    NaN   Kapton           3.55  50.612543  0.683325
3  metal    Brass         350.00  49.159245  1.205481
>>> st.saveas('stack_calc.csv')
```

get_flux(*sample_name*)

Returns the computed energy grid and flux for a sample

Units of energy are in MeV, and the flux is a relative flux (normalized to 1). Note that sample_name must be an exact match with the 'name' property in the stack specification.

Parameters

sample_name
[str] Name of the sample for which to return the flux. Must be an exact match for an element in the stack.

Returns

(energy, flux)
[tuple(np.ndarray, np.ndarray)] Energy grid in MeV, and relative flux in the sample, along the energy grid.

Examples

```
>>> stack = [{'cm':'H2O', 'ad':800.0, 'name':'water'},
                {'cm':'RbCl', 'density':3.0, 't':0.03, 'name':'salt'},
                {'cm':'Kapton', 't':0.025},
                {'cm':'Brass','ad':350, 'name':'metal'}]
>>> st = ci.Stack(stack, compounds={'Brass':{'Cu':-66, 'Zn':-33}}, E0=60.0)
>>> print(st.get_flux('water'))
(array([47.95, 48.05, 48.15, 48.25, 48.35, 48.45, 48.55, 48.65, 48.75,
48.85, 48.95, 49.05, 49.15...
>>> rx = ci.Reaction('16O(p,x)16F')
>>> print(rx.average(*st.get_flux('water')))
1.495163043176288
```

plot(*filter_name=None*, ***kwargs*)

Plots the fluxes for each foil in the stack calculation

Plots the flux distribution for each foil in the stack, or the filtered stack depending on the behaviour of `filter_name`.

Parameters

filter_name
[str, optional] Applies a filter to the fluxes before plotting. If a str, foils with a 'name' matching a regex search with filter_name are plotted. Default, None.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> stack = [{'cm':'H2O', 'ad':800.0, 'name':'water'},
                {'cm':'RbCl', 'density':3.0, 't':0.03, 'name':'salt'},
                {'cm':'Kapton', 't':0.025},
                {'cm':'Brass','ad':350, 'name':'metal'}]
```

```
>>> st = ci.Stack(stack, compounds={'Brass':{'Cu':-66, 'Zn':-33}}, E0=60.0)
>>> st.plot()
>>> st.plot(filter_name='salt')
```

saveas(*filename*, *save_fluxes=True*, *filter_name=True*)

Saves the results of the stack calculation

Saves the stack design, mean energies, and (optionally) the flux profile for each foil in the stack. Supported file types are '.csv', '.db' and '.json'.

Parameters

filename
[str] Name of file to save to. Supported file types are '.csv', '.db' and '.json'. If `save_fluxes=True`, an additional file will be saved to 'fname_fluxes.ftype'.

save_fluxes
[bool, optional] If True, an additional file will be saved with the flux profile for each foil in the stack. The foil must have a 'name' keyword, and can be further filtered with the `filter_name` argument. If false, only the stack design and mean energies are saved. Defaut, True.

filter_name
[bool or str, optional] Applies a filter to the stack and fluxes before saving. If True, only foils with a 'name' keyword will be saved. If 'False', foils without a 'name' will be saved in the stack design file, but not the fluxes file. If a str, foils with a 'name' matching a regex search with filter_name are saved. This applies to both files. Default, True.

Examples

```
>>> stack = [{'cm':'H2O', 'ad':800.0, 'name':'water'},
                {'cm':'RbCl', 'density':3.0, 't':0.03, 'name':'salt'},
                {'cm':'Kapton', 't':0.025},
                {'cm':'Brass','ad':350, 'name':'metal'}]
```

```
>>> st = ci.Stack(stack, compounds={'Brass':{'Cu':-66, 'Zn':-33}}, E0=60.0)
>>> st.saveas('example_stack.csv')
>>> st.saveas('example_stack.json', filter_name=False)
>>> st.saveas('example_stack.db', save_fluxes=False)
```

summarize(*filter_name=True*)

Summarize the stack calculation

Prints out the mean energies and 1-sigma energy widths of each foil in the stack, or the filtered stack depending on the behavior of `filter_name`.

Parameters

filter_name
[bool or str, optional] Applies a filter to the stack. If True, only foils with a 'name' keyword will be included. If 'False', a summary of all foils will be printed. If a str, foils with a 'name' matching a regex search with filter_name are included. Default, True.

Examples

```
>>> stack = [{'cm':'H2O', 'ad':800.0, 'name':'water'},
                {'cm':'RbCl', 'density':3.0, 't':0.03, 'name':'salt'},
                {'cm':'Kapton', 't':0.025},
                {'cm':'Brass','ad':350, 'name':'metal'}]
```

```
>>> st = ci.Stack(stack, compounds={'Brass':{'Cu':-66, 'Zn':-33}}, E0=60.0)
>>> st.summarize()
water: 55.45 +/- 2.94 (MeV)
salt: 50.68 +/- 0.69 (MeV)
metal: 49.17 +/- 1.21 (MeV)
>>> st.summarize(filter_name=False)
water: 55.45 +/- 2.94 (MeV)
salt: 50.68 +/- 0.69 (MeV)
Kapton-1: 50.62 +/- 0.69 (MeV)
metal: 49.17 +/- 1.21 (MeV)
```

7.5 Library

class curie.**Library**(*name*)

Library of nuclear reaction data

Provides a means of searching and retrieving data from various nuclear reaction libraries. The currently available libraries are ENDF/B-VIII.1, TENDL-2025, IRDFF-II, and the IAEA Medical Monitor reaction library. For neutrons, the libraries are categorized by either the exlusive reaction, as in ENDF, TENDL, and partially IRDFF, or by the residual product (rp), in TENDL and partially IRDFF. For charged particles, all libraries (TENDL, IAEA) are categorized by the residual product. As such, TENDL is split into separate libraries based on incident particle, and by exclusive reaction type or residual product.

Parameters

name
[str] The name of the library. For exclusive neutron reactions, use 'endf', 'tendl', or 'irdff'. For residual product neutron reactions, use 'tendl_n' or 'irdff'. Use 'tendl_p' for proton reactions, 'tendl_d' for deuteron reactions, 'tendl_a' for alpha reactions, or 'iaea' for any of them.

Methods

retrieve([target, incident, outgoing, product])	Pull reaction data from the library
search([target, incident, outgoing, ...])	Searches the library for reactions

Examples

```
>>> lb = ci.Library('tendl_n')
>>> print(lb.name)
TENDL-2025
>>> lb = ci.Library('endf')
>>> print(lb.name)
ENDF/B-VIII.1
```

retrieve(*target=None*, *incident=None*, *outgoing=None*, *product=None*)

Pull reaction data from the library

Returns a numpy ndarray containing the reaction energy grid (in MeV), the cross section (in mb), and for IRDFF and IAEA, the uncertainty in the cross section (mb). The arguments are the same as for `Library.search()`.

Parameters

target
[str, optional] The target nucleus. The IAEA monitor, IRDFF and TENDL residual-product libraries also carry natural elements, e.g. 'natEl'.

incident
[str, optional] Incident particle. Must be one of 'n', 'p', 'd' or 'a'. Only needed for IAEA library with multiple incident projectiles optional.

outgoing
[str, optional] Outgoing particle, or reaction shorthand. E.g. '2n', 'd', 'f', 'inl', 'x'. Does not need to be specified for TENDL residual product libraries.

product
[str, optional] The product isotope. Not required for some exclusive reactions, e.g. '115IN(n,g)116IN' is the same as '115IN(n,g)', but not the same as '115IN(n,g)116INm1'.

Returns

reaction
[np.ndarray] Numpy ndarray containing the reaction data. Energy is in MeV, cross section is in mb. The first column is energy, the second column is cross section, and if the library provides uncertainties in cross section this will be in the third column.

Examples

```
>>> lb = ci.Library('endf')
>>> print(lb.retrieve(target='226RA', product='225RA')[-8:])
[[18.5      11.97908 ]
 [18.75     10.30011 ]
 [19.        8.62115 ]
 [19.25      6.942188]
 [19.5       5.263225]
 [19.75      3.584263]
```

(continues on next page)

(continued from previous page)

```
[19.875     2.744781]
[20.        1.9053  ]]
```

search(*target=None*, *incident=None*, *outgoing=None*, *product=None*, *_label=False*)

Searches the library for reactions

Returns a list of reactions that match the specified target, incident or outgoing projectile, or product. Any combination of arguments can be specified, but at least one must be specified. Note that if the library is a TENDL residual-product library, outgoing does not need to be specified.

Parameters

target
[str, optional] The target nucleus. The IAEA monitor, IRDFF and TENDL residual-product libraries also carry natural elements, e.g. 'natEl'.

incident
[str, optional] Incident particle. Must be one of 'n', 'p', 'd' or 'a'. Only needed for IAEA library with multiple incident projectiles optional.

outgoing
[str, optional] Outgoing particle, or reaction shorthand. E.g. '2n', 'd', 'f', 'inl', 'x'. Does not need to be specified for TENDL residual product libraries.

product
[str, optional] The product isotope. Not required for some exclusive reactions, e.g. '115IN(n,g)116IN' is the same as '115IN(n,g)', but not the same as '115IN(n,g)116INm1'.

Returns

available_reactions
[list of str (reaction names)] Reactions that were found in the libary matching the search criteria.

Examples

```
>>> lb = ci.Library('tendl_p')
>>> print(lb.search(target='Sr-86', product='Y-86g'))
['86SR(p,x)86Yg']
>>> lb = ci.Library('endf')
>>> print(lb.search(target='226RA', product='225RA'))
['226RA(n,2n)225RA']
```

7.6 Reaction

class curie.**Reaction**(*reaction_name*, *library='best'*)

Cross section data for nuclear reactions

Contains reaction cross sections as a function of incident energy, and some useful methods for manipulating cross section data, such as flux-averages, integrated cross-sections, and interpolation. All cross sections (and uncertainties) are in mb, and all energies are in MeV.

Parameters

reaction_name
[str] Name of the reaction, in nuclear reaction notation. E.g. '115IN(n,g)', '235U(n,f)', '139LA(p,x)134CE', 'Ra-226(n,2n)Ra-225', 'Al-27(n,a)', etc.

library
[str, optional] Name of the library to use, or 'best' (default).

Attributes

target
[str] The target nucleus. The IAEA monitor, IRDFF and TENDL residual-product libraries also carry natural elements, e.g. 'natEl'.

incident
[str] Incident particle. E.g. 'n', 'p', 'd'.

outgoing
[str] Outgoing particle, or reaction shorthand. E.g. '2n', 'd', 'f', 'inl', 'x'. Will always be 'x' for (TENDL) residual product libraries.

product
[str] The product isotope.

interp_config
[dict] Interpolation configuration, set as a property or through keyword arguments to `interpolate`/`interpolate_unc`. The one key is 'interpolation': 'pchip-sqrt' (default for the TENDL libraries) or 'linear' (default for ENDF, IRDFF and IAEA).

eng
[np.ndarray] Incident particle energy, in MeV.

xs
[np.ndarray] Reaction cross section, in mb.

unc_xs
[np.ndarray] Uncertainty in the cross section, in mb. If not provided by the library, default is zeros of same shape as xs.

name
[str] Name of the reaction in nuclear reaction notation.

library
[ci.Library] Nuclear reaction library. printing `rx.library.name` will give the name of the library.

TeX
[str] LaTeX formatted reaction name.

Methods

average(energy, flux[, unc])	Flux averaged reaction cross section
integrate(energy, flux[, unc])	Reaction flux integral
interpolate(energy, **interp_config)	Interpolated cross section
interpolate_unc(energy, **interp_config)	Uncertainty in interpolated cross section
plot([energy, label, title])	Plot the cross section

Examples

```
>>> rx = ci.Reaction('226RA(n,2n)')
>>> print(rx.library.name)
ENDF/B-VIII.1
>>> rx = ci.Reaction('226RA(n,x)225RA')
```

(continues on next page)

(continued from previous page)

```
>>> print(rx.library.name)
TENDL-2025
>>> rx = ci.Reaction('115IN(n,inl)')
>>> print(rx.library.name)
IRDFF-II
```

average(*energy*, *flux*, *unc=False*)

Flux averaged reaction cross section

Calculates the flux-weighted average reaction cross section, using the input flux and energy grid.

Parameters

energy
[array_like] Incident particle energy, in MeV.

flux
[array_like] Incident particle flux as a function of the input energy grid.

unc
[bool, optional] If `True`, returns the both the flux average cross section and the uncertainty. If `False`, just the average cross section is returned. Default `False`.

Returns

average_xs
[float or tuple] Flux-averaged reaction cross section if `unc=False` (default), or average and uncertainty, if `unc=True`.

Examples

```
>>> rx = ci.Reaction('Ni-58(n,p)')
>>> eng = np.linspace(1, 5, 20)
>>> phi = np.ones(20)
>>> print(rx.average(eng, phi))
210.3226525877...
>>> print(rx.average(eng, phi, unc=True))
(210.3226525877..., ...)
```

integrate(*energy*, *flux*, *unc=False*)

Reaction flux integral

Integrate the product of the cross section and flux along the input energy grid.

Parameters

energy
[array_like] Incident particle energy, in MeV.

flux
[array_like] Incident particle flux as a function of the input energy grid.

unc
[bool, optional] If `True`, returns the both the flux integral and the uncertainty. If `False`, just the flux integral is returned. Default `False`.

Returns

xs_integral
[float or tuple] Reaction flux integral if `unc=False` (default), or reaction flux integral and uncertainty, if `unc=True`.

Examples

```
>>> rx = ci.Reaction('Ni-58(n,p)')
>>> eng = np.linspace(1, 5, 20)
>>> phi = np.ones(20)
>>> print(rx.integrate(eng, phi))
885.5690635272...
>>> print(rx.integrate(eng, phi, unc=True))
(885.5690635272..., ...)
```

`interpolate`(*energy*, ***interp_config*)

Interpolated cross section

Interpolation of the reaction cross section along the input energy grid. The scheme comes from `interp_config` (per-library defaults: 'pchip-sqrt' for the TENDL libraries — monotone PCHIP interpolation in sqrt(E)-sqrt(sigma) space, exact through the evaluated points with no overshoot at thresholds — and 'linear' for the pointwise-linearized ENDF, IRDFF and IAEA libraries). Energies outside the evaluated grid return 0 rather than an extrapolation.

Parameters

energy
[array_like] Incident particle energy, in MeV.

interp_config
[optional keyword arguments] Updates to `interp_config`, applied before interpolating and kept for later calls, e.g. `rx.interpolate(E, interpolation='linear')`.

Returns

cross_section
[np.ndarray] Interpolated cross section, in mb.

Examples

```
>>> rx = ci.Reaction('115IN(n,g)', 'IRDFF')
>>> print(rx.interpolate(0.5))
161.41646650941306
>>> print(rx.interpolate([0.5, 1.0, 5.0]))
[161.41646651 171.81486757 8.8822]
```

`interpolate_unc`(*energy*, ***interp_config*)

Uncertainty in interpolated cross section

Interpolation of the uncertainty in the reaction cross section along the input energy grid, for libraries where uncertainties are provided, using the same `interp_config` scheme as `interpolate`.

Parameters

energy
[array_like] Incident particle energy, in MeV.

interp_config
[optional keyword arguments] Updates to `interp_config`, applied before interpolating and kept for later calls.

Returns

unc_cross_section
[np.ndarray] Uncertainty in the interpolated cross section, in mb.

Examples

```
>>> rx = ci.Reaction('115IN(n,g)', 'IRDFF')
>>> print(rx.interpolate_unc(0.5))
3.9542683715745546
>>> print(rx.interpolate_unc([0.5, 1.0, 5.0]))
[3.95426837 5.88023936 0.4654]
```

plot(*energy=None*, *label='reaction'*, *title=False*, ***kwargs*)

Plot the cross section

Plots the energy differential cross section.

Parameters

energy
[array_like, optional] Energy grid along which to plot the cross section. If None, the energy grid provided by the library will be used.

label
[str, optional] Axes label. If label='reaction', the label will be the reaction name. If 'library', it will be the name of the cross section library. If 'both', then the reaction name and library will be given. If none of these options, pyplot will be called with `ax.plot(..., label=label)`.

title
[bool, optional] Display the reaction name as the plot title. Default, False.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> rx = ci.Reaction('115IN(n,g)')
>>> rx.plot(scale='loglog')
>>> rx = ci.Reaction('35CL(n,p)')
>>> f, ax = rx.plot(return_plot=True)
>>> rx = ci.Reaction('35CL(n,el)')
>>> rx.plot(f=f, ax=ax, scale='loglog')
```

7.7 Calibration

class curie.**Calibration**(*filename=None*)

Calibration for HPGe spectra

Provides methods for calculating and fitting energy, efficiency and resolution calibrations for HPGe spectra. Each Spectrum class contains a Calibration, which can be loaded from a file, or fit to calibration spectra from known sources.

Parameters

filename
[str, optional] Path to a .json file storing calibration data. The .json file can be produced by calling `cb.saveas('example_calib.json')` after performing a calibration fit with the `cb.calibrate()` function. This allows past calibrations to be recalled without needing to re-perform the calibration fits.

Attributes

engcal
[np.ndarray] Energy calibration parameters. length 2, 3 or 4 array, depending on whether the calibration is linear, quadratic or cubic.

effcal
[np.ndarray] Efficiency calibration parameters, for the semi-empirical efficiency model (see the `eff` method). Length 5 array, or length 7 if the two low-energy attenuation terms (detector window and dead layer) are included; for the 'loglog' model, the polynomial coefficients of ln(eff) in powers of ln(E).

unc_effcal
[np.ndarray] Efficiency calibration covariance matrix, of the same dimension as effcal (5x5 or 7x7 for the Vidmar models).

rescal
[np.ndarray] Resolution calibration parameters. length 2 array if resolution calibration is of the form R = a + b*chan (default), length 1 if R = a*sqrt(chan), or length 3 if R = sqrt(a + b*chan + c*chan^2).

fit_config
[dict] Calibration-fit configuration (see the class Attributes and `calibrate`)

Methods

calibrate(spectra, sources[, eff_points])	Generate calibration parameters from spectra
eff(energy[, effcal, model])	Efficiency calibration function
eng(channel[, engcal, model])	Energy calibration function
map_channel(energy[, engcal])	Energy to channel calibration
plot(**kwargs)	Plots energy, resolution and efficiency calibrations
plot_effcal(**kwargs)	Plot the efficiency calibration
plot_engcal(**kwargs)	Plot the energy calibration
plot_rescal(**kwargs)	Plot the resolution calibration
res(channel[, rescal, model])	Resolution calibration
saveas(filename)	Save the calibration as a .json file
unc_eff(energy[, effcal, unc_effcal, model])	Uncertainty in the efficiency

Examples

```
>>> cb = Calibration()
>>> print(cb.engcal)
[0.  0.3]
>>> cb.engcal = [0.1, 0.2, 0.003]
>>> print(cb.engcal)
[0.1   0.2   0.003]
>>> cb.saveas('test_calib.json')
```

(continues on next page)

(continued from previous page)

```
>>> cb = Calibration('test_calib.json')
>>> print(cb.engcal)
[0.1   0.2   0.003]
```

calibrate(*spectra*, *sources*, *eff_points=None*, ***kwargs*)

Generate calibration parameters from spectra

Performs an energy, resolution and efficiency calibration on peak fits to a given list of spectra. Reference activities must be given for the efficiency calibration. Spectra are allowed to have isotopes that are not in `sources`, but these will not be included in the efficiency calibration.

Keyword arguments other than `eff_points` are `fit_config` keys: they merge into `cb.fit_config` and persist for subsequent calibrations.

Parameters

spectra
[list of sp.Spectrum] List of calibration spectra. Must have sp.isotopes defined, and matching the isotopes given in `sources`.

sources
[str, list, dict or pd.DataFrame] Datatype or (if str) file that can be converted into a pandas DataFrame. Required keys are 'isotope', 'A0' (reference activity), and 'ref_date' (reference date).

eff_points
[pd.DataFrame, optional] User-supplied efficiency points appended to the measured points before the efficiency fit: columns energy (keV), efficiency, unc_efficiency, and optionally isotope. The public form of efficiency-extrapolation and multi-geometry merges. Their uncertainties are treated as independent.

engcal_model
[str, optional] 'linear', 'quadratic' or 'cubic'. Default `None`: the model is inferred from the current calibration's parameter length.

rescal_model
[str, optional] 'sqrt' (a*sqrt(ch)), 'linear' (a + b*ch) or 'sqrt_quad' (sqrt(a + b*ch + c*ch^2)). Default `None`: inferred by length.

effcal_model
[str, optional] 'vidmar' (the 5/7-parameter automatic choice, the default behavior), 'vidmar-5', 'vidmar-7', or a polynomial of ln(eff) in ln(E): 'loglog' (order 4) or 'loglog-2' through 'loglog-8' for an explicit order. Default `None` = 'vidmar'.

engcal_max_error
[float, optional] Pre-fit cut: drop energy-calibration points whose relative uncertainty exceeds this. Default 0.25.

rescal_max_error
[float, optional] As above, for resolution points. Default 0.33.

effcal_max_error
[float, optional] As above, for efficiency points. Default 0.33.

outlier_sigma
[float, optional] Post-fit residual threshold, in sigma, above which a point is clipped from the stored calibration points (it stays visible in the `*_data` tables and plots). Default sqrt(10), about 3.16.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.7E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> print(cb.effcal)
[4.78771190e-02 9.99910335e+01 2.10108097e+00 1.90374157e+00
 3.95525964e-01]
>>> cb.plot()
```

property diagnostics

Fit diagnostics from the most recent calibration

Read-only pd.DataFrame with one row per calibration sub-fit (`fit` is 'engcal', 'rescal' or 'effcal') and columns chi2 (reduced, over all n_points that entered the fit - flagged outliers included), dof, n_points (points the fit used), n_dropped (pre-fit drops plus post-fit outlier clips; clipping affects the stored points, not the fit), converged, model, scale_factor (uncertainty inflation applied; 1.0 = none), flags (comma-joined, e.g. 'chi2_high', 'at_bound:l') and message (the associated summary text). Empty (with the full schema) before any calibration has been performed; rebuilt on each `calibrate()` call. Accessing it never triggers a fit.

eff(*energy*, *effcal=None*, *model=None*)

Efficiency calibration function

Returns the calculated (absolute) efficiency given an input array of energies. The `effcal` can be supplied, or if `effcal=None`, the calibration object's internal efficiency calibration (cb.effcal) is used.

The functional form of the efficiency used is a modified version of the semi-empirical formula proposed by Vidmar (2001): eff(E) = c[0]*(1.0-exp(-mu(eng)*c[1]))*(tau(eng)+sigma(eng)*(1.0-exp(-(mu(eng)*c[3])**c[2]))*c[4])/mu(eng) if the effcal is length 5 or eff(E) = c[0]*exp(-mu_w(eng)*c[5])*exp(-mu(eng)*c[6])*(1.0-exp(-mu(eng)*c[1]))*(tau(eng)+sigma(eng)*(1.0-exp(-(mu(eng)*c[3])**c[2]))*c[4])/mu(eng) if the effcal is length 7.

Parameters

energy
[array_like] Peak energy in keV.

effcal
[array_like, optional] Efficiency calibration parameters. length 5 or 7 array, depending on whether the efficiency fit includes the low-energy components; for the 'loglog' model, the polynomial coefficients of ln(eff) in powers of ln(E).

model
[str, optional] Efficiency model: 'vidmar-5', 'vidmar-7' or 'loglog'. Default `None`: the calibration's own model tag if `effcal` is not given, else inferred from the parameter length (5 or 7 = Vidmar). A loglog parameter array can be the same length as a Vidmar one, so explicitly supplied loglog parameters require `model='loglog'`.

Returns

efficiency
[np.ndarray] Absolute efficiency at the given energies.

Examples

```
>>> cb = ci.Calibration()
>>> print(cb.effcal)
[1.87867638e-02 8.82671055e+01 2.09673703e+00 1.66193806e+00
 3.90089610e-01 3.79361265e+00 1.24058591e-02]
>>> print(cb.eff(50*np.arange(1,10)))
[0.00469671 0.00553412 0.00502548 0.00436836 0.00369016 0.00315241
 0.002754   0.00245546 0.00222482]
```

property effcal_data

Efficiency-calibration points from the most recent calibration

Read-only pd.DataFrame with one row per measured point, including the points the fit rejected: columns energy, efficiency, unc_efficiency, isotope (source provenance - which isotope the point came from, derived from the stored line label; '' for calibrations saved before it was recorded), used (False for rejected points - outlier rows entered the fit but are clipped from the stored calibration points), reason ('' when used, else 'unc>33%' or 'outlier >3.16 sigma') and residual (measured minus fitted efficiency). Empty (with the full schema) if no calibration data is present.

eng(*channel*, *engcal=None*, *model=None*)

Energy calibration function

Returns the calculated energy given an input array of channel numbers. The `engcal` can be supplied, or if `engcal=None` the Calibration object's energy calibration is used (cb.engcal).

Parameters

channel
[array_like] Spectrum channel number. The maximum channel number should be the length of the spectrum.

engcal
[array_like, optional] Optional energy calibration. If a length 2 array, calibration will be engcal[0] + engcal[1]*channel. If length 3, then engcal[0] + engcal[1]*channel + engcal[2]*channel**2, and if length 4 the cubic polynomial continues the same pattern.

model
[str, optional] Energy calibration model: 'linear', 'quadratic' or 'cubic'. Default `None`: the calibration's own model tag if `engcal` is not given, else inferred from the parameter length.

Returns

energy: np.ndarray
Calibrated energy corresponding to the given channels.

Examples

```
>>> cb = ci.Calibration()
>>> print(cb.engcal)
[0.  0.3]
>>> print(cb.eng(np.arange(10)))
[0.  0.3 0.6 0.9 1.2 1.5 1.8 2.1 2.4 2.7]
>>> cb.engcal = [0.1, 0.2, 0.003]
>>> print(cb.eng(np.arange(10)))
[0.1   0.303 0.512 0.727 0.948 1.175 1.408 1.647 1.892 2.143]
```

property engcal_data

Energy-calibration points from the most recent calibration

Read-only pd.DataFrame with one row per measured point, including the points the fit rejected: columns channel, energy, unc_channel, used (False for rejected points), reason (" when used, else e.g. 'unc>25%') and residual (measured minus fitted energy, keV). Empty (with the full schema) if no calibration data is present.

property fit_config

Calibration-fit configuration (see the class Attributes and `calibrate`)

map_channel(*energy*, *engcal=None*)

Energy to channel calibration

Calculates the spectrum channel number corresponding to a given energy array. This should return the inverse of the energy calibration, but as an integer-type channel number.

Parameters

energy
[array_like] Peak energy in keV

engcal
[array_like, optional] Energy calibration parameters. length 2, 3 or 4 array, depending on whether the calibration is linear, quadratic or cubic (the cubic is inverted numerically).

Returns

channel
[np.ndarray] Calculated channel number given the input energy array

Examples

```
>>> cb = ci.Calibration()
>>> print(cb.engcal)
[0.  0.3]
>>> print(cb.map_channel(300))
1000
>>> print(cb.eng(cb.map_channel(300)))
300.0
```

plot(***kwargs*)

Plots energy, resolution and efficiency calibrations

Draws all three of energy, resolution and efficiency calibrations on a single figure, and shows measured values from peak data, if available.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.5E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> cb.plot()
```

plot_effcal(***kwargs*)

Plot the efficiency calibration

Draws the efficiency calibration, with measurements from peak fit data if available.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.5E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> cb.plot_effcal()
```

plot_engcal(***kwargs*)

Plot the energy calibration

Draws the energy calibration, with measurements from peak fit data if available.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.5E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> cb.plot_engcal()
```

plot_rescal(***kwargs*)

Plot the resolution calibration

Draws the resolution calibration, with measurements from peak fit data if available.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.5E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> cb.plot_rescal()
```

res(*channel*, *rescal=None*, *model=None*)

Resolution calibration

Calculates the expected 1-sigma peak widths for a given input array of channel numbers. If `rescal` is given, it is used instead of the calibration object's internal value (cb.rescal).

Parameters

channel
[array_like] Spectrum channel number. The maximum channel number should be the length of the spectrum.

rescal
[array_like, optional] Resolution calibration parameters. length 2 array if resolution calibration is of the form R = a + b*chan (default), length 1 if R = a*sqrt(chan), or length 3 if R = sqrt(a + b*chan + c*chan^2) (the 'sqrt_quad' model).

model
[str, optional] Resolution model: 'sqrt', 'linear' or 'sqrt_quad'. Default `None`: the calibration's own model tag if `rescal` is not given, else inferred from the parameter length.

Returns

resolution
[np.ndarray] Calculated 1-sigma width of the peaks given the input channel numbers.

Examples

```
>>> cb = ci.Calibration()
>>> print(cb.rescal)
[2.e+00 4.e-04]
>>> print(cb.res(100*np.arange(1,10)))
[2.04 2.08 2.12 2.16 2.2  2.24 2.28 2.32 2.36]
```

property rescal_data

Resolution-calibration points from the most recent calibration

Read-only pd.DataFrame with one row per measured point, including the points the fit rejected: columns channel, width, unc_width, used (False for rejected points - outlier rows entered the fit but are clipped from the stored calibration points), reason (" when used, else 'unc>33%' or 'outlier >3.16 sigma') and residual (measured minus fitted width, channels). Empty (with the full schema) if no calibration data is present.

saveas(*filename*)

Save the calibration as a .json file

Saves the energy, resolution and efficiency calibration to a .json file, which can be recalled by a new calibration object by passing the 'filename' keyword to Calibration() upon construction.

Parameters

filename
[str] Complete filename to save the calibration. Must end in '.json'.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> cb = ci.Calibration()
>>> cb.calibrate([sp], sources=[{'isotope':'152EU', 'A0':3.7E4, 'ref_date':'01/
↪01/2009 12:00:00'}])
>>> print(cb.effcal)
[4.78771190e-02 9.99910335e+01 2.10108097e+00 1.90374157e+00
 3.95525964e-01]
>>> cb.saveas('example_calib.json')
```

```
>>> cb = ci.Calibration('example_calib.json')
>>> print(cb.effcal)
[4.78771190e-02 9.99910335e+01 2.10108097e+00 1.90374157e+00
 3.95525964e-01]
```

unc_eff(*energy*, *effcal=None*, *unc_effcal=None*, *model=None*)

Uncertainty in the efficiency

Returns the calculated uncertainty in efficiency for an input array of energies. If `effcal` or `unc_effcal` are not none, they are used instead of the calibration object's internal values. (cb.unc_effcal) unc_effcal must be a covariance matrix of the same dimension as effcal.

Parameters

energy
[array_like] Peak energy in keV.

effcal
[array_like, optional] Efficiency calibration parameters. length 5 or 7 array, depending on whether the efficiency fit includes the low-energy components; for the 'loglog' model, the polynomial coefficients of ln(eff) in powers of ln(E).

unc_effcal
[array_like, optional] Efficiency calibration covariance matrix, of the same dimension as effcal (5x5 or 7x7 for the Vidmar models).

model
[str, optional] Efficiency model, as in `eff` (needed when explicitly supplied parameters are for the 'loglog' model).

Returns

unc_efficiency
[np.ndarray] Absolute uncertainty in efficiency for the given energies.

Examples

```
>>> cb = ci.Calibration()
>>> print(cb.eff(50*np.arange(1,10)))
[0.00469671 0.00553412 0.00502548 0.00436836 0.00369016 0.00315241
```

(continues on next page)

(continued from previous page)

```
 0.002754   0.00245546 0.00222482]
>>> print(cb.unc_eff(50*np.arange(1,10)))
[1.84927072e-05 2.06720560e-05 4.00073148e-05 3.34061820e-05
 1.49979136e-05 1.05818602e-05 9.72833934e-06 8.95311822e-06
 8.10774373e-06]
```

7.8 Spectrum

class curie.**Spectrum**(*filename=None*, ***kwargs*)

Gamma-ray spectrum from High-Purity Germanium (HPGe) detectors

Provides methods for reading, converting, and fitting gamma-ray spectra from HPGe data.

Parameters

filename
[str] File path to the gamma-ray spectrum. Supported file types are Ortec .Spe and .Chn files, as well as Canberra .CNF and .IEC files.

Attributes

cb
[ci.Calibration] Gamma-ray energy, efficiency and resolution calibration.

isotopes
[list] List of gamma-decaying isotopes observed in the spectrum.

fit_config
[dict] Dictionary of fit configuration parameters passed as keyword arguments to `self.fit_peaks()`. See that function for more details on these arguments.

filename
[str] File path to the gamma-ray spectrum.

hist
[np.ndarray] Histogram of counts observed in the spectrum.

start_time
[datetime.datetime] Date-time marking the start of the count.

live_time
[float] Total time the spectrum was counting, in seconds, minus the dead time.

real_time
[float] Total time the spectrum was counting, in seconds.

peaks
[pd.DataFrame] Table of fits to the gamma-ray data. Includes counts, isotopes, energies, intensities, efficiencies, calculated decays and decay-rates, chi^2, and other information. The complete list of columns is 'filename', 'isotope', 'energy', 'counts', 'unc_counts', 'intensity', 'unc_intensity', 'efficiency', 'unc_efficiency', 'decays', 'unc_decays', 'decay_rate', 'unc_decay_rate', 'chi2', 'start_time', 'live_time', and 'real_time'.

Methods

attenuation_correction(compounds[, x, ad])	Efficiency correction for sample attenuation
auto_calibrate([guess, peaks, adjust])	Attempt to automatically adjust the energy calibration
fit_peaks([gammas])	Fit the peaks in the spectrum
geometry_correction(distance, r_det, ...[, ...])	Efficiency correction for sample geometry
plot([fit, xcalib])	Plot the spectrum
rebin(N_bins)	Rebin the histogram
saveas(filename[, replace])	Save the spectrum or peak information to a file
summarize()	Summarize the fitted peaks in the spectrum

Other Parameters

cb
[str or ci.Calibration] Calibration to use, or path to a calibration .json file.

isotopes
[list] List of gamma-decaying isotopes observed in the spectrum.

fit_config
[dict] Dictionary of fit configuration parameters passed as keyword arguments to `self.fit_peaks()`. See that function for more details on these arguments.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['Eu-152', '40K']
>>> sp.cb = 'example_calib.json'
>>> sp.fit_config = {'bg':'quadratic', 'xrays':False}
>>> sp.saveas('eu_calib_7cm.Chn')
>>> sp = ci.Spectrum('eu_calib_7cm.Chn')
```

attenuation_correction(*compounds*, *x=None*, *ad=None*)

Efficiency correction for sample attenuation

Corrects the efficiency used in calculating observed decays from each peak for sample attenuation. A list of compounds (and their areal densities) is given, and a correction factor as a function of energy is determined using the photon attenuation data from XCOM. The first compound in the list is assumed to be the radiogenic sample (the "self"), and the other compounds are presumed between the detector and sample. If it is desired to neglect self-attenuation, specify the thickness of the first compound to be zero. Either thickness in units of centimeters must be given for each compound, or the areal density in units of g/cm^2. Note that calling `sp.attenuation_correction()` will modify the efficiencies/decays in the peak fits, but it also returns the correction factor function used to do so.

Parameters

compounds
[list of str or ci.Compound] List of compounds to correct for. The first compound is assumed to be the radiogenic sample ("self"), the following compounds are assumed to be between the source and detector. If str, the compounds must be either a natural element, or be included in `ci.COMPOUND_LIST`. They can also be a ci.Compound object.

x
[list of float] List of the thicknesses (in cm) of each compound to be corrected for. len(x) must be the same as len(compounds). If a density other than the density of the correspond-

ing compound is desired, use the `ad` keyword. Only one of `x` or `ad` should be given, but at least one is required.

ad
[list of float] List of the areal densities (in g/cm^2) of each compound to be corrected for. len(ad) must be the same as len(compounds). Only one of `x` or `ad` should be given, but at least one is required.

Returns

correction_factor
[scipy.interpolate.interp1d] Interpolation function that gives the correction factor as a function of energy, in keV. Correction factor should be multiplied with the efficiency.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> print(sp.attenuation_correction(['Fe', ci.Compound('H2O', density=1.0)],␣
↪x=[0.1, 0.5])(100*np.arange(1, 10)))
[0.79614452 0.88215107 0.90281046 0.91410313 0.92187566 0.92780998
 0.93217524 0.9365405  0.93969564]
>>> print(sp.attenuation_correction(['La', ci.Compound('Kapton', density=12.0)],
↪ ad=[0.1, 0.5])(100*np.arange(1, 10)))
[0.82705455 0.92060463 0.93878407 0.94717435 0.95248909 0.95638036
 0.95914978 0.9619192  0.96387558]
```

auto_calibrate(*guess=None*, *peaks=None*, *adjust=1.0*)

Attempt to automatically adjust the energy calibration

Uses a genetic forward fitting algorithm to attempt to automatically adjust the energy calibration in cases where the peak centroids are significantly mis-calibrated. The algorithm requires a list of isotopes in the spectrum, so self.isotopes cannot be None. Also, the feature will only make small adjustments to the calibration, so if the current calibration is greater than 0.5 percent off, either a guess or a list of peak locations must be given

Parameters

guess
[array_like, optional] Guess parameters for the energy calibration. Must be a length 2 or 3 array. If None, the energy calibration from self.cb.engcal will be used.

peaks
[array_like, optional] List of peak locations. Must be a 2-d array of format [[channel, energy], . . .]

adjust
[float, optional] Parameter to proportionally adjust the range of calibration parameters explored by `auto_calibrate()`. Default, 1.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> print(sp.cb.engcal)
[3.3973e-01 1.8297e-01 5.5683e-09]
>>> sp.cb.engcal = [0.3, 0.184]
>>> sp.isotopes = ['152EU']
>>> sp.plot()
```

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.cb.engcal = [0.3, 0.1835]
>>> sp.isotopes = ['152EU']
>>> sp.auto_calibrate()
>>> print(sp.cb.engcal)
[0.3        0.18281362 0.        ]
>>> sp.plot()
```

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.cb.engcal = [0.3, 0.1]
>>> sp.isotopes = ['152EU']
>>> sp.auto_calibrate(peaks=[[664, 121.8]])
>>> print(sp.cb.engcal)
[0.         0.18289791 0.        ]
>>> sp.plot()
```

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.cb.engcal = [0.3, 0.1]
>>> sp.isotopes = ['152EU']
>>> sp.auto_calibrate(guess=[0.3, 0.1835])
>>> print(sp.cb.engcal)
[0.3        0.18281398 0.        ]
>>> sp.plot()
```

property diagnostics

Fit diagnostics from the most recent fit_peaks call

Read-only pd.DataFrame with one row per *attempted* multiplet, failed fits included: columns chi2 (reduced), dof, n_points (channels in the fit window), n_dropped (candidates removed while assembling this multiplet), converged, model (background model), scale_factor (uncertainty inflation applied; 1.0 = none), flags (comma-joined, e.g. 'chi2_high', 'at_bound:A', 'fit_failed'), message (associated summary, warning and detail text - per-peak at-bound and per-multiplet high-chi2 detail lands here and at DEBUG, not the default console), energy_min/energy_max (fit window in keV), isotopes and n_peaks. Empty (with the full schema) before any fit; rebuilt on each `fit_peaks()` call. Accessing it never triggers a fit.

fit_peaks(*gammas=None*, ***kwargs*)

Fit the peaks in the spectrum

Fits a multiplet of peaks, with configurable parameters, to the spectrum. The list of gammas can either be generated automatically by setting the `Spectrum.isotopes` attribute, or by using the 'gammas' keyword, which adds to any gammas generated from the isotopes attribute. This function is called once when any of the following happen: the `Spectrum.peaks` property is accessed, the `Spectrum.saveas()` method is called, or the `Spectrum.summarize()` or `Spectrum.plot()` methods are called. The peaks are then saved, and won't be re-fit unless `Spectrum.fit_peaks()` (this method) is called explicitly. This is important if parameters such as the calibration, isotopes, or fit_config are changed after the peaks are generated the first time.

Parameters

gammas

[list, dict or pd.DataFrame, optional] Manual entry for specifying peaks in the spectrum. 'gammas' must be an object that can be converted into a pandas DataFrame, with the keywords 'energy', 'intensity', 'unc_intensity' and optionally 'isotope'. Units of intensity should be percent, units of energy should be keV.

Returns

peaks
[pd.DataFrame] Table of peaks that were successfully fit.

Other Parameters

xrays
[bool] Whether peak fits should include x-rays, as given by Nudat2. Default False.

E_min
[float] Minimum peak energy to fit, in keV. Default 75.0.

I_min
[float or 2-tuple] Minimum peak intensity to fit, in percent. A (lower, upper) 2-tuple restricts fitting to intensities within that closed range. Default 0.05

dE_511
[float] Gammas that are fewer than dE_511 keV from the 511 keV annihilation peak are excluded from the fit. Default 3.5

bg
[str] Type of background fit to use. Options are 'constant', 'linear', 'quadratic' or 'snip'. Default 'snip'. The 'snip' algorithm interpolates the background under the peaks using the detector resolution, and the assumption that the background is smoothly varying. Will fail for peaks that are on top of quickly varying features such as the electron backscatter peak, but has the benefit of removing a free parameter from the fit.

skew_fit
[bool] If skew_fit is True, the fit parameters include the skewed gaussian component, otherwise only the gaussian parameters are fit. Default False. The skewed gaussian is characterized by the parameters R and alpha, and has the functional form R*A*exp((x-mu)/(alpha*sig))*erfc((x-mu)/(sqrt(2)*sig)+1.0/(sqrt(2)*alpha)) where A, mu and sigma are the amplitude, centroid and width of the gaussian peak function. Note that unless R and alpha are explicitly set to zero, they will be included in the peak function. skew_fit only specifies whether or not their values are to be fit for each peak.

step_fit
[bool] If step_fit is True, a step function is added to the background fit, which arises from compton scattering. Default False. The functional form of the step function is step*A*erfc((x-mu)/(sqrt(2)*sig)), where A, mu and sigma are the amplitude, centroid and width of the gaussian peak. Similar to skew_fit, unless step is explicitly set to zero (it is by default), a step component is included in every peak. step_fit only specifies whether or not the value of this parameter is fit for every peak.

R
[float] Expected value of R (amplitude of skewed component of gaussian). Default 0.1.

alpha
[float] Expected value of alpha (width of skewed component of gaussian). Default 0.9.

step
[float] Expected value of step (amplitude of compton background). Default 0.0.

pk_width
[float] Number of peak standard deviations (width) to include in the spectrum data passed to the curve fit function. Default 7.5. Should be wide enough such that several background channels not in the peak can be included in the fit. If too wide, multiplets may begin to overlap.

snip_adj
[float] Multiplier for parameters in the snip background calculation. Default 1.0.

SNR_min
[float] Minimum acceptable peak amplitude to noise (sqrt(background)) ratio. Default 4.0.

A_bound
[float] Multiplier for the bounds on the fit parameter A (peak amplitude). Default 1.0.

mu_bound
[float] Multiplier for the bounds on the fit parameter mu (peak centroid). Default 1.0.

sig_bound
[float] Multiplier for the bounds on the fit parameter sig (peak width). Default 1.0.

multi_max
[int] Maximum number of peaks allowed in a multiplet. Default 8.

ident_idx
[int] Channel separation below which two gammas are identical/overlapping. Default 0. The comparison uses the unrounded channel distance: two gammas within ident_idx + 0.5 channels of each other will be combined (with summed intensity) if from the same isotope, or flagged if from two different isotopes; if one of the identical peaks is estimated to be <1% of the peak height of the other, the smaller peak is removed. Setting ident_idx=-1 turns this feature off.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
>>> print(sp.fit_peaks(SNR_min=5, dE_511=12))
>>> print(sp.fit_peaks(bg='quadratic'))
```

property fits

Detailed fit results for each successfully fitted multiplet

Read-only list with one dict per fitted multiplet, in energy order: `l/h` (channel range of the fit window), `p0` (starting parameter estimates), `bounds` (the (lower, upper) parameter bounds), `istp` (isotope of each peak), `df` (the candidate gamma table for the multiplet), `fit` (fitted parameters) and `unc` (their covariance matrix). Parameters are the background terms (none for the 'snip' model) followed by A, mu, sig (and R, alpha with skew_fit, step with step_fit) for each peak. `None` before any fit; failed multiplets do not appear (see `Spectrum.diagnostics`). Accessing this never triggers a fit - call `fit_peaks()` or access `peaks` first.

geometry_correction(*distance*, *r_det*, *thickness*, *sample_size*, *shape='circle'*, *N=1000000*)

Efficiency correction for sample geometry

Correction to the efficiency for non-point source geometries. The correction factor is a multiplier to the efficiency, that corrects for a sample that was efficiency calibrated using a point source, but that is not a point source itself. Note that the input sizes do not need units, but they are assumed to all be in the same units system (e.g. they are all in cm, or inches). Also, like the attenuation_correction, the correction factor is automatically applied to the peak data, but it is returned by the function as well.

Parameters

distance
[float] Distance from the front of the sample to the detector. Units must be consistent with other inputs.

r_det
[float] Radius of the detector. Units must be consistent with other inputs.

thickness
[float] Thickness of the sample, in the same units as the other inputs.

sample_size
[float] Characteristic sample dimensions (in same units). If shape is circle, this is sample radius. If square, it is side length. If rectangle, sample_size must be a length 2 tuple/array/list, corresponding to the x and y dimensions of the rectangle.

shape
[str, optional] Shape of the sample. Options are 'circle' (default), 'square' and 'rectangle'.

N
[int, optional] Number of particles to use in Monte-Carlo solid angle calculation. Default is 1E6. If the sampling error in the correction factor is greater than 1 percent, a warning will be given to increase N.

Returns

correction_factor
[float] Geometry correction factor, to be multiplied with the efficiency.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> print(sp.geometry_correction(distance=4, r_det=5, thickness=0.1, sample_
↪size=2, shape='square'))
0.9744182633801829
>>> print(sp.geometry_correction(distance=30, r_det=5, thickness=10, sample_
↪size=1))
0.7586316490264302
>>> print(sp.geometry_correction(distance=4, r_det=5, thickness=0.1, sample_
↪size=(2,1.5), shape='rectangle'))
0.9784496516955806
```

`plot`(*fit=True*, *xcalib=True*, ***kwargs*)

Plot the spectrum

Draws a plot of the spectrum, and any successful peak fits. Peaks are colored by isotope, and a dashed grey line is drawn over multiplets to help evaluate goodness of fit. Multiplets whose fit failed are marked by red hatching of the counts above the background (the peaks no fit describes); a warning reports how many.

Parameters

fit
[bool, optional] If True, include peak fits. Else, only the spectrum is drawn. Default, True.

xcalib
[bool, optional] If True, the x-axis is the calibrated energy in keV. If False, it is the ADC channel number. This may help locate peaks to give the `Spectrum.auto_calibrate()` function if the energy calibration is poor. Default, True.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
>>> sp.plot()
>>> sp.plot(xcalib=False)
>>> sp.plot(style='poster')
```

rebin(*N_bins*)

Rebin the histogram

Rebins the histogram to the closest power of 2 to the given `N_bins`. Energy and resolution calibrations will be adjusted to match the new bin length. Note N_bins must be less than the current histogram length.

Parameters

N_bins
[int] Number of desired bins. Rounds to closest power of 2, e.g. 1000 rounds to 1024.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> print(len(sp.hist))
16384
>>> sp.rebin(1000)
>>> print(len(sp.hist))
1024
```

saveas(*filename*, *replace=False*)

Save the spectrum or peak information to a file

If the file type is one of '.png', '.pdf', '.svg' or another graphical file type, a plot of the spectrum will be saved. If it is one of the supported spectra formats, '.Spe', '.Chn' or '.spe', the spectrum will be converted and saved. Note that only '.Spe' and '.Chn' can be read by ci.Spectrum. If the file type is one of '.csv', '.db' or '.json', the peak data will be saved. All three can be read by the ci.DecayChain.get_counts method.

Parameters

filename
[str] Name of the file to save to. Supported file types are '.png', '.pdf', '.pickle', '.eps', '.pgf', '.ps', '.raw', '.rgba', '.svg', '.svgz', '.Spe', '.Chn', '.spe', '.csv', '.json', and '.db'.

replace
[bool, optional] If True, when saving to one of the three supported peak data file types, any existing file will be completely overwritten. If False, then only peak data that matches the spectrum name will be overwritten. Default, False.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
>>> sp.saveas('test_plot.png')
>>> sp.saveas('eu_calib.Chn')
>>> sp.saveas('peak_data.csv')
```

summarize()

Summarize the fitted peaks in the spectrum

Prints a summary of the observed counts in each peak, the number of decays of the given isotope it corresponds to, and an estimate of the activity of the isotope.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
>>> sp.summarize()
152EU - 121.7817 keV (I = 28.53%)
----------------------------------
counts: 498785 +/- 3195
decays: 2.515e+07 +/- 2.702e+06
activity (Bq): 1.025e+04 +/- 1.101e+03
activity (uCi): 2.770e-01 +/- 2.976e-02
chi2/dof: 21.923
...
```

7.9 Decay Chain

class curie.**DecayChain**(*parent_isotope*, *R=None*, *A0=None*, *units='s'*, *timestamp=True*)

Radioactive Decay Chain

Uses the Bateman equations to calculate the activities and number of decays from a radioactive decay chain as a function of time, both in production and decay. Also, initial isotope activities and production rates can be fit to observed count data, or directly fit to HPGe spectra using the `get_counts()` function.

Parameters

parent_isotope
[str] Parent isotope in the chain.

R
[array_like, dict, str or pd.DataFrame] Production rate for each isotope in the decay chain as a function of time. If a Nx2 np.ndarray, element n gives the production rate R_n up until time t_n for the parent isotope. E.g. If the production rate of the parent is 5 for 1 hour, and 8 for 3 hours, the array will be [[5, 1], [8, 4]]. If instead time intervals are preferred to a monotonically increasing grid of timestamps, set 'timestamp=False'. In this case the production rate array will be [[5, 1], [8, 3]]. (R=5 for 1 hour, R=8 for 3 hours).

If R is a dict, it specifies the production rate for multiple isotopes, where the keys are the isotopes and the values are type np.ndarray.

If R is a pd.DataFrame, it must have columns 'R' and 'time', and optionally 'isotope' if R>0 for any isotopes other than the parent. If R is a str, it must be a path to a file where the same data is provided. Supported file types are .csv, .json and .db files, where .json files must be in the 'records' format, and .db files must have a table named 'R'. Also, each isotope must have the same time grid, for both timestamp=True and timestamp=False.

A0
[float or dict] Initial activity. If a float, the initial activity of the parent isotope. If a dict, the keys are the isotopes for which the values represent the initial activity.

units
[str, optional] Units of time for the chain. Options are 'ns', 'us', 'ms', 's', 'm', 'h', 'd', 'y', 'ky', 'My', 'Gy'. Default is 's'.

timestamp
[bool, optional] Determines if the 'time' variable in R is to be read as a timestamp-like grid, i.e. in monotonically increasing order, or as a series of time intervals. Default is `True`.

Attributes

R
[pd.DataFrame] Production rate as a function of time, for each isotope in the chain. This will be modified if `fit_R()` is called.

A0
[dict] Initial activity of each isotope in the chain.

isotopes
[list] List of isotopes in the decay chain.

counts
[pd.DataFrame] Observed counts from isotopes in the decay chain, which can be used to determine the initial activities or average production rates using the `fit_R()` or `fit_A0()` functions.

R_avg
[pd.DataFrame] Time-averaged production rate for each isotope where R>0. This will be modified if `fit_R()` is called.

Methods

activity(isotope, time[, units, _R_dict, ...])	Activity of an isotope in the chain
decays(isotope, t_start, t_stop[, units, ...])	Number of decays in a given time interval
fit_A0(**kwargs)	Fit the initial activity to count data
fit_R(**kwargs)	Fit the production rate to count data
get_counts(spectra, EoB[, peak_data])	Retrieves the number of measured decays
plot([time, max_plot, max_label, ...])	Plot the activities in the decay chain

Examples

```
>>> dc = ci.DecayChain('Ra-225', R=[[1.0, 1.0], [0.5, 1.5], [2.0, 6]], units='d')
>>> print(dc.isotopes)
['225RAg', '225ACg', '221FRg', '217ATg', '213BIg', '217RNg', '209TLg', '213POg',
↪'209PBg', '209BIg', '205TLg']
>>> print(dc.R_avg)
  R_avg isotope
0  1.708333  225RAg
```

```
>>> dc = ci.DecayChain('152EU', A0=3.7E3, units='h')
>>> print(dc.isotopes)
['152EUg', '152GDg', '152SMg', '148SMg', '144NDg', '140CEg']
```

activity(*isotope*, *time*, *units=None*, *_R_dict=None*, *_A_dict=None*)

Activity of an isotope in the chain

Computes the activity of a given isotope in the decay chain at a given time. Units of activity are in Bq. Units of time must be either the units for the DecayChain (default 's'), or specified by the `units` keyword.

Parameters

isotope
[str] Isotope for which the activity is calculated.

time
[array_like] Time to calculate the activity. Units of time must be the same as the decay chain, or be given by `units`. Note that if R!=0, time=0 is defined as the end of production time. Else, if A0!=0, time=0 is defined as the time at which the specified activities equaled A0. t<0 is not allowed.

units
[str, optional] Units of time, if different from the units of the decay chain.

Returns

activity
[np.ndarray] Activity of the given isotope in Bq.

Examples

```
>>> dc = ci.DecayChain('152EU', A0=3.7E3, units='h')
>>> print(dc.activity('152EU', time=0))
3700.0
>>> print(dc.activity('152EU', time=13.537, units='y'))
1849.999906346199
```

decays(*isotope*, *t_start*, *t_stop*, *units=None*, *_A_dict=None*)

Number of decays in a given time interval

Computes the number of decays from a given isotope in the decay chain in the time interal t_start to t_stop. The units of t_start and t_stop must be either the same units as the decay chain, or be specified by the `units` keyword.

Parameters

isotope
[str] Isotope for which the number of decays is calculated.

t_start
[array_like] Time of the start of the interval.

t_stop
[array_like] Time of the end of the interval.

units
[str, optional] Units of time, if different from the units of the decay chain.

Returns

decays
[np.ndarray] Number of decays

Examples

```
>>> dc = ci.DecayChain('152EU', A0=3.7E3, units='h')
>>> print(dc.decays('152EU', t_start=1, t_stop=2))
13319883.293399204
>>> print(dc.decays('152EU', t_start=50, t_stop=50.1, units='y'))
900151618.5228329
```

property diagnostics

Fit diagnostics from the most recent fit_R or fit_A0 call

Read-only pd.DataFrame with one row per fitted isotope (`fit` is 'fit_R' or 'fit_A0'; the fit is joint, so chi2, dof, n_points, n_dropped and scale_factor repeat on every row): columns chi2 (reduced, of the unscaled fit), dof, n_points (counts the fit used), n_dropped (counts removed by the max_error/min_counts filters), converged, model (" - the chain fit has no model selection), scale_factor (uncertainty inflation applied; 1.0 = none), flags (comma-joined, e.g. 'unmoved', 'at_bound:R', 'singular_cov'), message (summary and warning text) and isotope. Empty (with the full schema) before any fit; rebuilt on each `fit_R()`/`fit_A0()` call. Accessing it never triggers a fit.

fit_A0(***kwargs*)

Fit the initial activity to count data

Fits a scalar multiplier to the initial activity for each isotope specified in self.A0. The fit minimizes to the number of measured decays (self.counts) as a function of time, rather than the activity, because the activity at each time point may be sensitive to the shape of the decay curve.

Keyword arguments other than `cov` are `fit_config` keys: they merge into `dc.fit_config` (like `Spectrum.fit_config`) and persist for subsequent fits.

Parameters

max_error
[float, optional] The maximum relative error of a count datum to include in the fit. E.g. 0.25=25% (ony points with less than 25% error will be shown). Default, 0.4 (40%).

min_counts
[float or int, optional] The minimum number of counts (decays) for a datum in self.counts to be included in the fit. Default, 1.

max_chi2
[float, optional] Exclude counts whose originating peak fit had chi2/dof above this bound (needs the `chi2` column `get_counts` carries over from the peak data; counts without one are unaffected). Default `None` (off).

exclude_lines
[list, optional] Named lines to exclude from the fit: entries are an isotope name ('152EU') or an (isotope, energy_keV) tuple. The energy matches the closest of that isotope's line energies in the counts, accepted within +-0.5 keV; an entry matching nothing excludes nothing and warns, naming the nearest candidate. Default `None` (off).

time_range
[dict, optional] Per-isotope count-time window, `{isotope: (t_lo, t_hi)}` in chain units; `None` for an open bound. Counts starting outside the window are excluded. Default `None` (off).

corr
[float, optional] Opt-in uniform-correlation mode for counts that carry only total uncertainties: the fraction (0-1) of each point's variance treated as fully correlated across points (or within `corr_group` groups). Default `None` (off).

corr_group
[str, optional] Name of a counts column defining the correlation groups for `corr`. Default `None` (global).

norm_frac
[float, optional] Fraction of each line's intensity variance treated as common to all lines of the isotope (the decay-scheme normalization). Default 1.0 (fully common - conservative until the intensity data carry the normalization uncertainty separately).

scale_factor
[bool, optional] If `True` (default), the fitted covariance is inflated by chi-square per degree of freedom when that exceeds 1 (mutually inconsistent count data); it is never deflated.

cov
[array_like, optional] Full covariance matrix of the count data, overriding the assembled one. For advanced use (kwarg only - not a fit_config key).

Returns

isotopes
[list] List of isotopes where A0>0. Same indices as fit. (i.e. isotope[0] corresponds to fit[0] and cov[0][0].)

fit
[np.ndarray] The initial activity for each isotope where A0>0.

cov
[np.ndarray] Covariance matrix on the fit.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> dc = ci.DecayChain('152EU', A0=3.7E4, units='d')
>>> dc.get_counts([sp], EoB='01/01/2016 08:39:08')
>>> itp, A0, cov = dc.fit_A0()
>>> print(itp[0], A0[0])
152EUg 138582.75831764835
```

`fit_R`(***kwargs*)

Fit the production rate to count data

Fits a scalar multiplier to the production rate (as a function of time) for each isotope specified in self.R. The fit minimizes to the number of measured decays (self.counts) as a function of time, rather than the activity, because the activity at each time point may be sensitive to the shape of the decay curve.

Keyword arguments other than `p0` and `cov` are `fit_config` keys: they merge into `dc.fit_config` (like `Spectrum.fit_config`) and persist for subsequent fits.

Parameters

max_error
[float, optional] The maximum relative error of a count datum to include in the fit. E.g. 0.25=25% (ony points with less than 25% error will be shown). Default, 0.4 (40%).

min_counts
[float or int, optional] The minimum number of counts (decays) for a datum in self.counts to be included in the fit. Default, 1.

max_chi2
[float, optional] Exclude counts whose originating peak fit had chi2/dof above this bound (needs the `chi2` column `get_counts` carries over from the peak data; counts without one are unaffected). Default `None` (off).

exclude_lines
[list, optional] Named lines to exclude from the fit: entries are an isotope name ('152EU') or an (isotope, energy_keV) tuple. The energy matches the closest of that isotope's line energies in the counts, accepted within +-0.5 keV; an entry matching nothing excludes nothing and warns, naming the nearest candidate. Default `None` (off).

time_range
[dict, optional] Per-isotope count-time window, `{isotope: (t_lo, t_hi)}` in chain units; `None` for an open bound (e.g. `{'196AUm2': (None, 60.0)}`). Counts starting outside the window are excluded. Default `None` (off).

unc_R_floor
[float, optional] Relative floor on the returned production-rate uncertainty: unc_R >= floor x R, per isotope, applied after the fit (announced). E.g. 0.05 keeps every unc_R at or above 5%. Default `None` (off).

corr
[float, optional] Opt-in uniform-correlation mode for counts that carry only total uncertainties: the fraction (0-1) of each point's variance treated as fully correlated across points (or within `corr_group` groups). Default `None` (off).

corr_group
[str, optional] Name of a counts column defining the correlation groups for `corr`. Default `None` (global).

norm_frac
[float, optional] Fraction of each line's intensity variance treated as common to all lines of the isotope (the decay-scheme normalization). Default 1.0 (fully common - conservative until the intensity data carry the normalization uncertainty separately).

scale_factor
[bool, optional] If `True` (default), the fitted covariance is inflated by chi-square per degree of freedom when that exceeds 1 (mutually inconsistent count data); it is never deflated.

p0
[dict, optional] Starting estimate override, `{isotope: R_estimate}` in production-rate units: seeds the fit at the given rate instead of the data-derived estimate (kwarg only - not a fit_config key). Default `None`.

cov
[array_like, optional] Full covariance matrix of the count data, overriding the assembled one. For advanced use (kwarg only - not a fit_config key).

Returns

isotopes
[list] List of isotopes where R>0. Same indices as fit. (i.e. isotope[0] corresponds to fit[0] and cov[0][0].)

fit
[np.ndarray] The fitted time-averaged production rate for each isotope where R>0.

cov
[np.ndarray] Covariance matrix on the fit.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> dc = ci.DecayChain('152EU', R=[[3E5, 36.0]], units='d')
>>> dc.get_counts([sp], EoB='01/01/2016 08:39:08')
>>> print(dc.fit_R())
(['152EUg'], array([27527059.31414273]), array([[2.03699956e+11]]))
```

get_counts(*spectra*, *EoB*, *peak_data=None*)

Retrieves the number of measured decays

Takes the number of measured decays from one of the following: a list of spectra, a file with peak data, or a pandas DataFrame with peak data.

Parameters

spectra
[list or str] List of ci.Spectrum objects, or str of spectra filenames. If list of str, peak_data **must** be specified. In this case the filenames must be an exact match of the filenames in `peak_data`. If spectra is a str, it is assumed to be a regex match for the filenames in `peak_data`.

EoB
[str or datetime.datetime] Date/time of end-of-bombardment (t=0). Must be a datetime object or a string in the format '%m/%d/%Y %H:%M:%S'. This is used to calculate the decay time for the count.

peak_data
[str or pd.DataFrame, optional] Either a file path to a file that was created using `ci.Spectrum.saveas()` or a DataFrame with the same structure as `ci.Spectrum.peaks`.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
>>> sp.saveas('test_spec.json')
```

```
>>> dc = ci.DecayChain('152EU', A0=3.7E3, units='h')
>>> dc.get_counts([sp], EoB='01/01/2016 08:39:08')
>>> dc.get_counts(['eu_calib_7cm.Spe'], EoB='01/01/2016 08:39:08', peak_data=
↪'test_spec.json')
>>> print(dc.counts)
```

plot(*time=None*, *max_plot=10*, *max_label=10*, *max_plot_error=0.4*, *max_plot_chi2=10*, ***kwargs*)

Plot the activities in the decay chain

Plots the activities as a function of time for all radioactive isotopes in the decay chain. Can plot along a specified time grid, else the time will be inferred from the half-life of the parent isotope, or any count information given to self.counts.

Parameters

time
[array_like, optional] Time grid along which to plot. Units must be the same as the decay chain.

max_plot
[int, optional] Maximum number of isotope activities to plot in the decay chain. Default, 10.

max_label
[int, optional] Maximum number of isotope activities to label in the legend. Default, 10.

max_plot_error
[float, optional] The maximum relative error of a count point to draw as a filled marker. E.g. 0.25=25%. Points above the threshold are drawn as open grey markers rather than hidden. Default, 0.4.

max_plot_chi2
[float or int, optional] Maximum chi^2 of a count point to draw as a filled marker. Points above the threshold are drawn as open grey markers rather than hidden. Default, 10.

Other Parameters

****kwargs**
Optional keyword arguments for plotting. See the plotting section of the curie API for a complete list of kwargs.

Examples

```
>>> sp = ci.Spectrum('eu_calib_7cm.Spe')
>>> sp.isotopes = ['152EU']
```

```
>>> dc = ci.DecayChain('152EU', R=[[3E5, 36.0]], units='d')
>>> dc.get_counts([sp], EoB='01/01/2016 08:39:08')
>>> dc.fit_R()
>>> dc.plot()
```

```
>>> dc = ci.DecayChain('99MO', A0=350E6, units='d')
>>> dc.plot()
```

7.10 Data

`curie.download`(*db='all'*, *overwrite=False*)

Download nuclear data files as sqlite .db files.

Data files are fetched automatically the first time they are needed, so calling this function is only necessary to prepare an offline machine, to pre-populate a fresh data directory, or to repair/update existing data. An installed file from an older data generation is replaced by the current release automatically; `overwrite=True` replaces files unconditionally. Files are stored in a per-user data directory (printed by `ci.data._data_path()`), which can be overridden with the `CURIE_DATA_DIR` environment variable. Every download is verified against the SHA256 recorded for the curie data release.

Note that the large cross-section libraries (ENDF, the TENDL variants) are normally assembled on demand from small per-target files; downloading them here fetches each complete library in one file. Files already provided by a site-wide data directory are reported and skipped unless `overwrite=True`, which fetches a fresh copy into the user data directory (shadowing the site copy).

Parameters

db
[str, optional] Name of database to download, default is 'all'. Options include: 'all', 'decay',

'ziegler', 'endf', 'tendl', 'tendl_n_rp', 'tendl_p_rp', 'tendl_d_rp', 'tendl_a_rp', 'IRDFF', 'iaea_monitors'.

overwrite
[bool, optional] If overwrite is `True`, will save write over existing data. Default is `False`.

Examples

The most common use case will be to (re)download all the data files

```
>>> ci.download(overwrite=True)
```

Some other use cases: To update only the 'endf' library

```
>>> ci.download('endf', True)
```

Or to download the 'decay' library for the first time

```
>>> ci.download('decay')
```

7.11 Logging

Curie prints a summary of what each fit computed and which data it kept or dropped through console messages of the form `[LEVEL] Class.method: message` (see *Fit Reporting Conventions* for the conventions). Three functions control the output:

`curie.set_log_level(level)`

Set the console message level

Controls which curie messages are printed to the console. Messages of the given level and above are shown. The default level is 'INFO', which shows per-fit summary lines and warnings; 'DEBUG' additionally shows each dropped peak or count with the threshold that removed it; 'WARNING' shows only likely result-changing messages; 'ERROR' silences everything except messages accompanying raised exceptions.

This affects the console only; log files created with `ci.log_to()` keep their own level.

Parameters

level
[str] One of 'DEBUG', 'INFO', 'WARNING', 'ERROR' (case-insensitive), or a numeric level from the logging module.

Examples

```
>>> ci.set_log_level('DEBUG')
>>> ci.set_log_level('INFO')
```

`curie.quiet_warnings()`

Silence curie's console messages

One-call opt-out of curie's default console output: raises the console level to 'ERROR', silencing both the routine INFO summary lines and WARNING messages. Errors accompanying raised exceptions are still shown. To keep warnings but silence the routine lines, use `ci.set_log_level('WARNING')` instead. Log files created with `ci.log_to()` are unaffected.

Examples

```
>>> ci.quiet_warnings()
>>> ci.set_log_level('INFO')  # restore the default
```

curie.**log_to**(*filename*, *level='INFO'*, *mode='overwrite'*)

Write curie's messages to a log file

Adds a log file receiving curie's messages at the given level, independent of the console level (e.g. a 'DEBUG' file under a quiet console). By default an existing file at the same path is overwritten, since analysis scripts are typically re-run many times; pass `mode='append'` to accumulate runs instead. Calling `log_to` again with the same path replaces the previous file handler rather than duplicating messages.

Parameters

filename
[str] Path of the log file.

level
[str, optional] Minimum message level written to the file: 'DEBUG', 'INFO', 'WARNING' or 'ERROR'. Default 'INFO'.

mode
[str, optional] 'overwrite' (default) or 'append'. The open()-style spellings 'w' and 'a' are also accepted.

Examples

```
>>> ci.log_to('analysis.log')
>>> ci.log_to('debug.log', level='DEBUG', mode='append')
```

7.12 Plotting

curie.**colormap**(*palette=None*, *shade=None*, *aslist=False*)

Broad spectrum of hex-formatted colors for plotting

Function to provide a broad palette of colors for plotting, inspired by the FlatUI color picker at https://flatuicolors.com/

Parameters

palette
[str, optional] Choice of color palette. Options are `default`, `american`, `aussie`, `british`, `canadian`, `chinese`, `french`, `german`, `indian`, `russian`, `spanish`, `swedish` and `turkish`.

shade
[str, optional] `light` or `dark` theme. Default is `dark`.

aslist
[bool, optional] Specifies whether to return the colormap as a dictionary, e.g. {'k':'#000000', 'w':'#ffffff'} or as a list, e.g. ['#000000', '#ffffff']. Default is `False` (returns dictionary)

Returns

colormap
[dict or list] Colormap in list or dictionary format

Examples

```
>>> cm = ci.colormap()
>>> print(cm)
{'b': '#2980b9', 'g': '#27ae60', 'k': '#2c3e50', 'o': '#d35400', 'aq': '#16a085',
'p': '#8e44ad', 'r': '#c0392b', 'gy': '#7f8c8d', 'w': '#bdc3c7', 'y': '#f39c12'}
>>> cm_list = ci.colormap(aslist=True)
>>> cm_british_light = ci.colormap('british', 'light')
```

curie.**set_style**(*style='default'*, *palette='default'*, *shade='dark'*)

Preset styles for generating plots

Adjusts default font sizes, colors and spacing for plots, with presets based on the intended application (i.e. `poster`, `paper`, etc.)

Parameters

style
[str, optional] Preset style based on intended application of the plot. Options are `paper`, `poster`, `presentation`, and the default: `default`.

palette
[str, optional] Sets colormap palette for plots, see `colormap` for detailed options.

shade
[str, optional] Sets colormap shade for plots, see `colormap` for detailed options

Examples

```
>>> ci.set_style('paper')
>>> ci.set_style('poster', shade='dark')
>>> ci.set_style('presentation', palette='aussie')
```

7.12.1 Class Plotting Methods

The following classes have plotting methods callable as `obj.plot()`, where `obj` is an instance of the appropriate class:

- Calibration
- Spectrum
- DecayChain
- Reaction
- Stack

All `obj.plot()` methods have no required arguments, and while some have specific optional arguments that are documented in the respective classes, all plotting methods share the same optional **keyword arguments**, denoted in the code by `**kwargs`.

f
[matplotlib.pyplot figure] The figure to draw on. Supply `f` *and* `ax` together to draw on top of existing axes (e.g. to overlay several plots).

ax
[matplotlib.pyplot axes] The axes to draw on. Supply `f` *and* `ax` together to draw on top of existing axes.

figsize
[tuple] Width and height of the figure in inches, passed to `matplotlib.pyplot.subplots`, e.g. `(8, 5)`.

scale
[str] Can specify the scale for the x and y axes with the following options:

- `log` or `logy` : Set *only* y-scale to log
- `logx` : Set *only* x-scale to log
- `lin`, `liny` or `linear` : Set *only* y-scale to linear
- `linx` : Set *only* x-scale to linear
- `linlin` : x-scale *and* y-scale linear
- `loglog` : x-scale *and* y-scale log
- `linlog` : linear x-scale and log y-scale
- `loglin` : log x-scale and linear y-scale

show
[bool] Whether or not to show the figure using the matplotlib GUI.

saveas
[str] Full filename for saving the figure. Any matplotlib-supported image format is accepted; a `.pickle` filename instead saves the figure object itself to disk, so it can be reloaded and edited later in Python.

return_plot
[bool] If `True`, then a tuple of (fig, axes) will be returned when calling the `obj.plot()` method. This can be used to draw multiple plots over each other, e.g. for plotting multiple cross-sections.

style, **palette** and **shade** are also available as keyword arguments, and have the same behavior as in `curie.set_style()`.

CHAPTER

EIGHT

LICENSE AGREEMENT

BIBLIOGRAPHY

[Ryan1988] C.G. Ryan et al., "SNIP, a statistics-sensitive background treatment for the quantitative analysis of PIXE spectra in geoscience applications", *Nucl. Instrum. Methods Phys. Res. B* **34** (1988) 396.

[Vidmar2001] T. Vidmar, M. Korun, A. Likar and M. Lipoglavsek, "A physically founded model of the efficiency curve in gamma-ray spectrometry", *J. Phys. D: Appl. Phys.* **34** (2001) 2555, doi:10.1088/0022-3727/34/16/323.

[Kis1998] Z. Kis et al., "Comparison of efficiency functions for Ge gamma-ray detectors in a wide energy range", *Nucl. Instrum. Methods A* **418** (1998) 374, doi:10.1016/S0168-9002(98)00778-5.

[Bateman1910] H. Bateman, "The solution of a system of differential equations occurring in the theory of radio-active transformations", *Proc. Cambridge Philos. Soc.* **15** (1910) 423.

[AZ1977] H.H. Andersen and J.F. Ziegler, *Hydrogen: Stopping Powers and Ranges in All Elements* (Pergamon, New York, 1977).

[Z1977] J.F. Ziegler, *Helium: Stopping Powers and Ranges in All Elemental Matter* (Pergamon, New York, 1977).

[ENDF] "ENDF/B-VIII.1", National Nuclear Data Center, Brookhaven National Laboratory (2024), https://www.nndc.bnl.gov/endf-b8.1/.

[TENDL] A.J. Koning and D. Rochman, "Modern Nuclear Data Evaluation with the TALYS Code System", *Nucl. Data Sheets* **113** (2012) 2841; TENDL-2025 files from https://tendl.imperial.ac.uk (CC BY 4.0).

[IRDFF] A. Trkov et al., "IRDFF-II: A New Neutron Metrology Library", *Nucl. Data Sheets* **163** (2020) 1.

[IAEA] A. Hermanne et al., "Reference Cross Sections for Charged-particle Monitor Reactions", *Nucl. Data Sheets* **148** (2018) 338.

INDEX

Q

R

S

U