FOX: Fourier-based Observation of aXions

AKA our analysis chain

Elena Gramellini

Scope of the "Data Analysis"

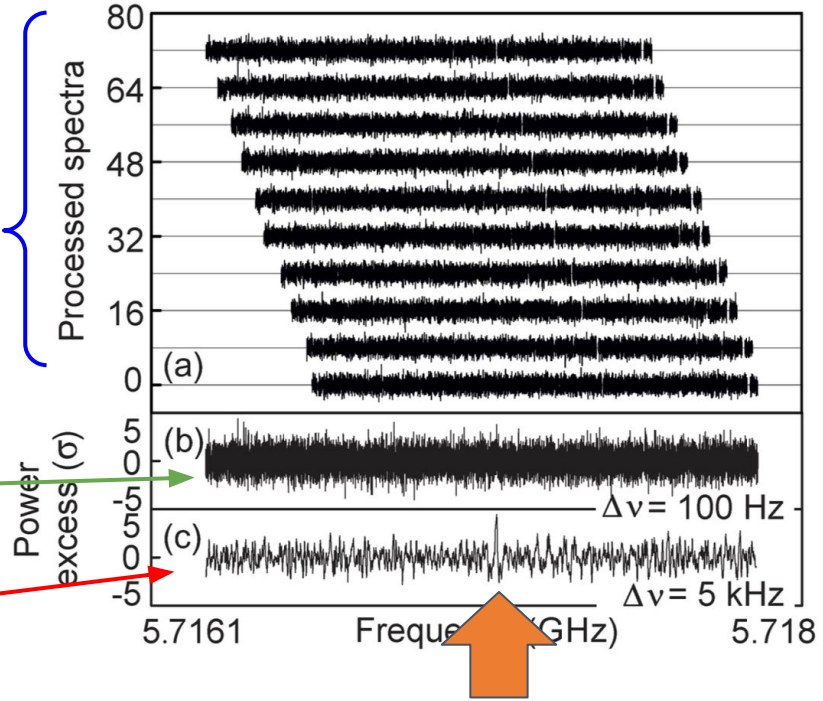
Combine multiple thousands of spectra into one optimized axion-sensitive spectrum.

Must remove baselines, rescale by noise, apply optimal weights.

15-min integrated data @ "fine" resolution (e.g. 100 Hz)

Weight-combined spectrum fine resolution

Final product = "grand spectrum" at "low" resolution (e.g. 5kHz) where axion signal would appear.



Injected synthetic signal

Steps of Analysis

(0) Simulation

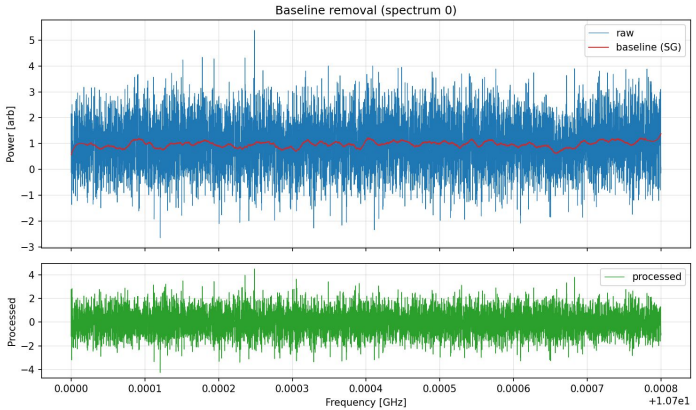
(1) Data quality cuts.

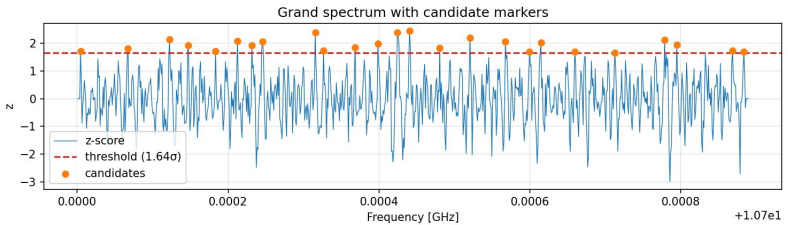
(2) Baseline removal with Savitzky–Golay (SG) filter.

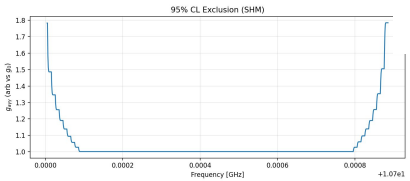
(3) Rescaling & vertical combining.

(4) Horizontal combining → grand spectrum.

(5) Candidate selection & rescans







Steps of Analysis

(0) Simulation → IS WHAT WE ARE SIMULATING REALISTIC?

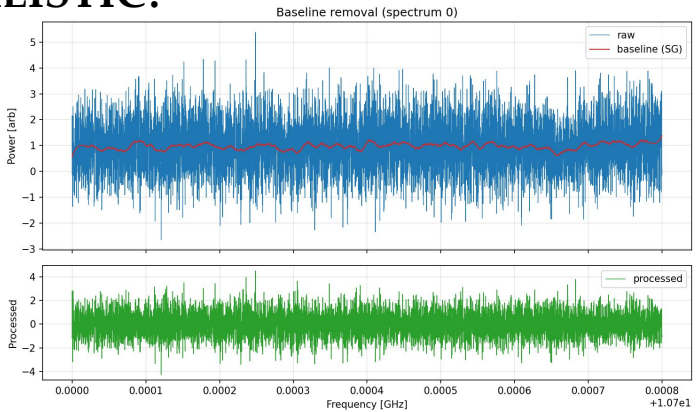
(1) Data quality cuts.

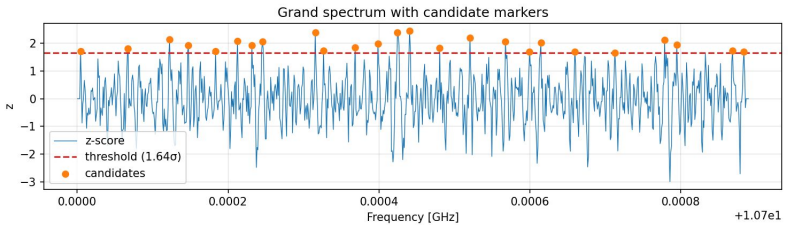
(2) Baseline removal with Savitzky–Golay (SG) filter.

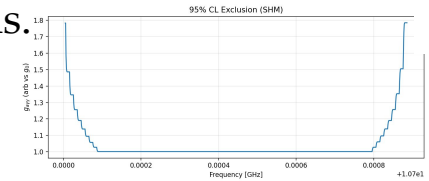
(3) Rescaling & vertical combining.

(4) Horizontal combining → grand spectrum.

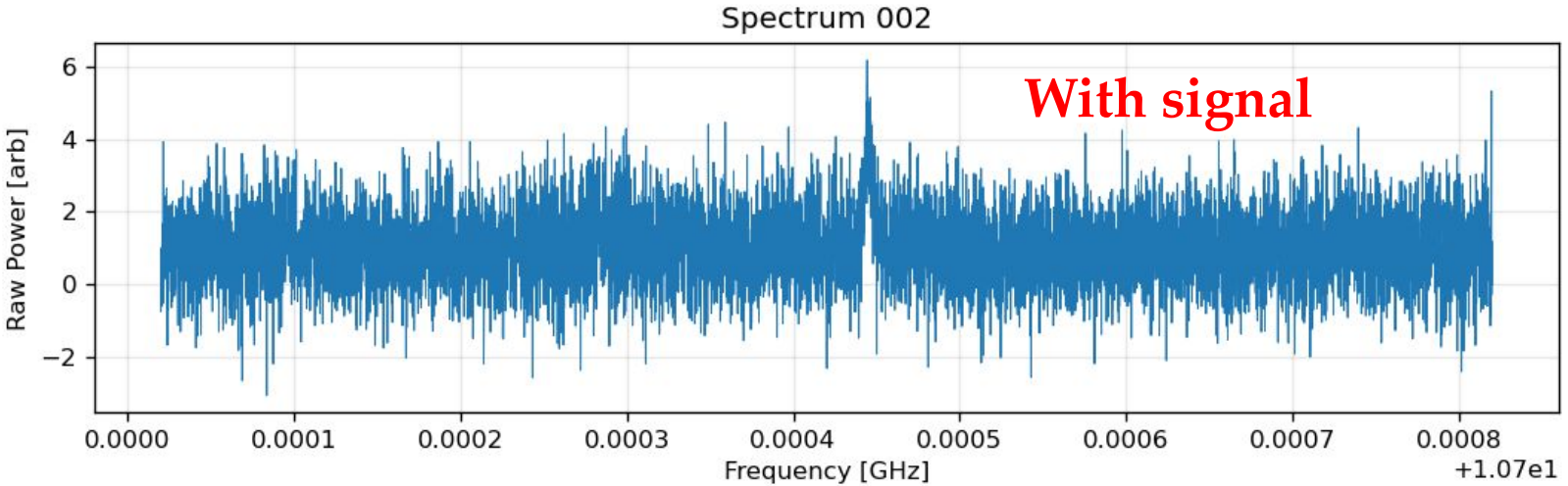
(5) Candidate selection & rescans.







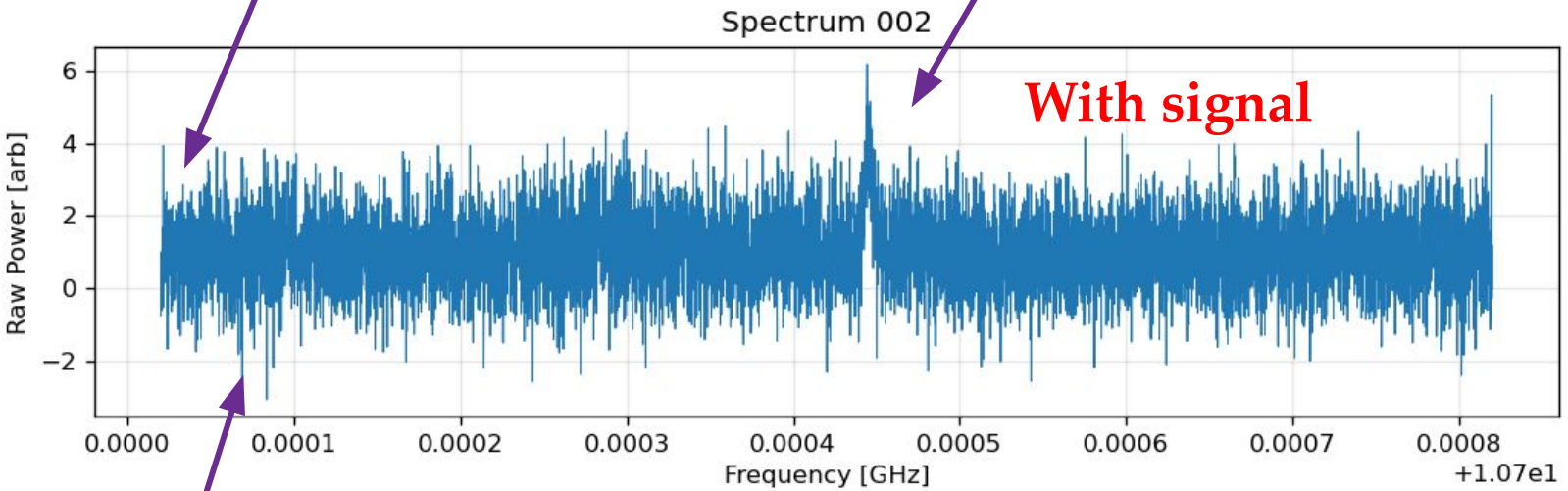
NO.



NO. Issues:

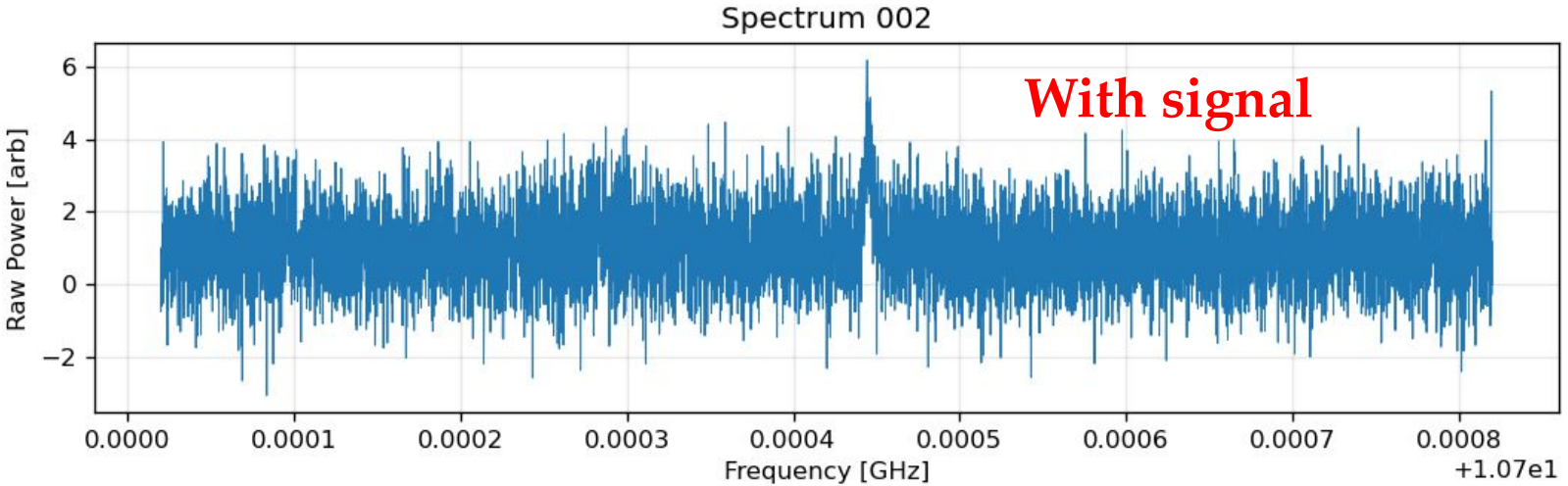
Size of this noise

How much g_a this corresponds to?



Negative power (unphysical)

NO. The real issue.



In order to have a meaningful S/N ration (aka a meaningful data analysis), our signal and background simulations need to be somewhat grounded in reality…

… so, in the words of Eazy-E, we went back back to the * basis. Blue will tell you how we did that.

Steps of Analysis: caveats aside, we do have a framework

(0) Simulation

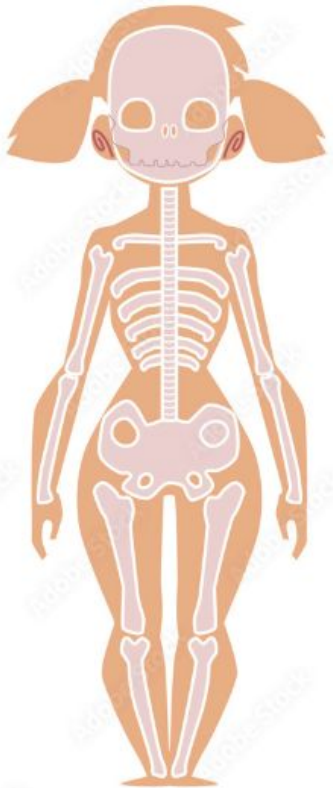
(1) Data quality cuts.

(2) Baseline removal with Savitzky–Golay (SG) filter.

(3) Rescaling & vertical combining.

(4) Horizontal combining → grand spectrum.

(5) Candidate selection & exclusion



Analysis framework: 3 parts

Python package called ***axion_haloscope***

Package functions

```
└── axion_haloscope/
    __init__.py
    baseline.py
    combine.py
    cli.py
    cli_example.sh
    config.py
    detection.py
    how_to_replace_lineshape.py
    io.py
    limit.py
    lineshape.py
    rebin.py
    run_pipeline.py
    simulation.py
```

Script to run them (+ I/O)

```
└── scripts/
    full_chain_template.py
    read_npz_pipeline.py
    read_spectra_npz.py
  └── simulation/
      simulate_run_yaml.py
      simulate_single_spectra_yaml.py
└── configs/
    simulate_run.yaml
    simulate_spectra_only.yaml
└── output/
  └── sim_spectra/
    └── run_20250908_104941/
```

Unit tests to check code issues

```
└── tests/
    test_combine.py
    test_detection.py
    test_data_quality.py
    test_io_hdf5.py
    test_lineshape.py
    test_simulation_plot.py
    test_simulation_smoke.py

└── test_output/
  └── test_hdf5_roundtrip0/
  └── test_plot_first_spectrum0/
```

Stuff you need for package development (not analysis specific)

Python package called ***axion_haloscope***

Link to code: https://github.com/HALO-DM/FOX/

pyproject.toml is the main project configuration file to setup and build Python package.

Here we define:

1. How to build/install the package
2. The package metadata
3. Python and dependency requirements
   When you run the "pip install -e ".[dev]" command,
   pip reads this file and installs those dependencies
4. Optional development tools
5. Command-line tools
   e.g. create the shell command "axion-haloscope" that creates that runs axion_haloscope/cli.py
6. Pytest configuration
   e.g. tells pytest where to look for tests

Unit test

Unit tests to check code issues: protect the analysis pipeline from silent code mistakes. They do not prove the physics result, but they make the code safer and easier to trust.

Examples:

`simulation.py` → correct spectra and frequency grids
`io.py` → saved HDF5/NPZ data reloads unchanged
`baseline.py` → baseline removal returns finite spectra
`combine.py` → overlapping bins combine correctly
`lineshape.py` → SHM template is normalized
`detection.py` → thresholds behave as expected
`data_quality.py` → bad spectra are removed

Unit test pass

Unit tests to check code issues: when you run unit-tests this is what success looks like

```
(MANCX_Env) eduroam-174072445-20:toy_m k00088eg$ pytest --basetemp=./test_output
============================= test session starts ==============================
platform darwin -- Python 3.12.11, pytest-8.4.1, pluggy-1.6.0
rootdir: /Users/user/Desktop/toy/toy/toy_m
configfile: pyproject.toml
testpaths: tests
collected 19 items

tests/test_combine.py .                                                  [  5%]
tests/test_data_quality.py ....                                          [ 26%]
tests/test_detection.py .                                                [ 31%]
tests/test_io_hdf5.py .                                                  [ 36%]
tests/test_lineshape.py ........                                         [ 78%]
tests/test_simulation_plot.py ..                                         [ 89%]
tests/test_simulation_smoke.py ..                                        [100%]

============================== 19 passed in 1.22s ==============================
(MANCX_Env) eduroam-174072445-20:toy_m k00088eg$
```

In current directory:

```
└── tests/
    test_combine.py
    test_detection.py
    test_data_quality.py
    test_io_hdf5.py
    test_lineshape.py
    test_simulation_plot.py
    test_simulation_smoke.py

└── test_output/
    └── test_hdf5_roundtrip0/
    └── test_plot_first_spectrum0/
```

+ a number of test plots in the test_output folder

Analysis framework

Python package called ***axion_haloscope***

Script to run them (+ I/O)

```
└── scripts/
    full_chain_template.py
    read_npz_pipeline.py
    read_spectra_npz.py
  └── simulation/
      simulate_run_yaml.py
      simulate_single_spectra_yaml.py
└── configs/
    simulate_run.yaml
    simulate_spectra_only.yaml
└── output/
  └── sim_spectra/
    └── run_20250908_104941/
```

I/O module

The current **axion_haloscope/io.py** supports a common I/O interface for simulated and real spectra through a **SpectrumSet** container.

```
SpectrumSet(
    spectra=[...],          # list of raw power spectra
    freqs_per_spec=[...],   # frequency axis for each spectrum
    rf_grid=...,            # common global RF grid
    rf_index_map=[...]      # maps each spectrum onto rf_grid
)
```

The I/O module lets the rest of the analysis pipeline ignore where the data came from.
Whether the spectra are simulated, loaded from CSV, loaded from .npz, or loaded from HDF5, they all become the same object: **SpectrumSet**

Steps of Analysis

(0) Simulation

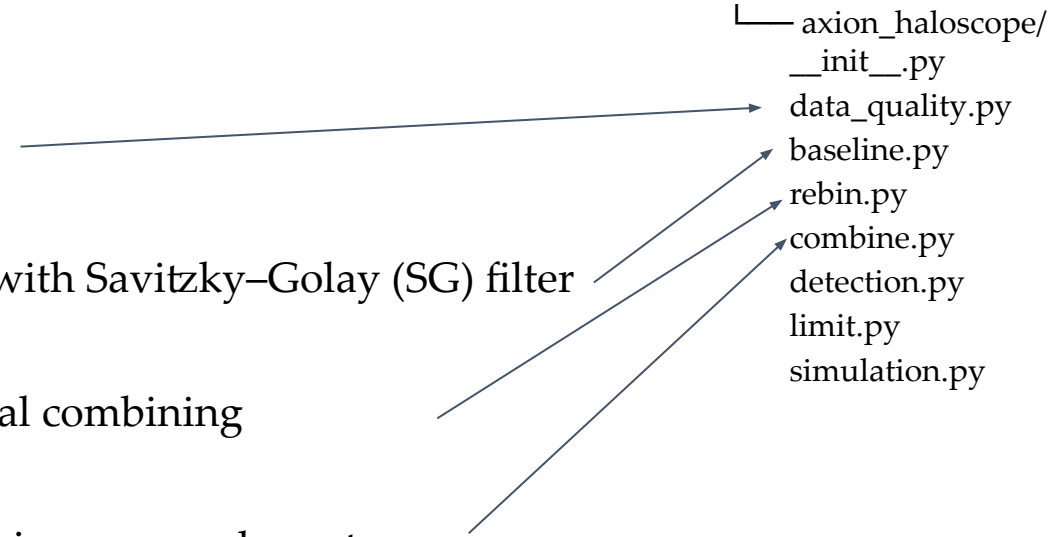
(1) Data quality cuts

(2) Baseline removal with Savitzky–Golay (SG) filter

(3) Rescaling & vertical combining

(4) Horizontal combining → grand spectrum

(5) Candidate selection & rescans



Simulation

Simulate noise + axion shape and strength. Current analysis framework can write and read HDF5 (or npz)

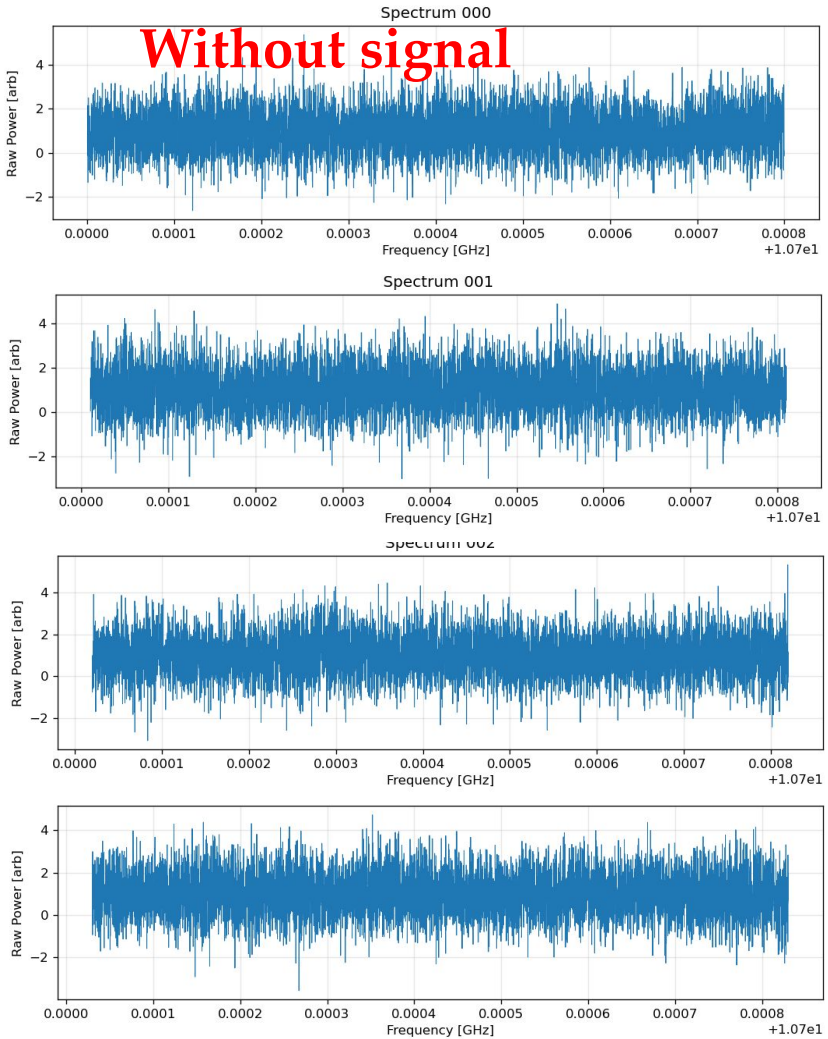
Configurable via yaml file:

```yaml
simulation:
  n_spectra: 10
  n_bins: 8000
  bin_width_hz: 100.0
  f_start_hz: 10.70e9
  tune_step_bins: 100
  rng_seed: 1234
  noise_sigma: 1.0

injection:
  enabled: true
  # f_axion_hz: 5.725e9   # optional; if omitted, placed at mid-span
  sigma_hz: 2500.0
  total_power: 200.0


output:
  save_data: true       # save per-spectrum PNGs + spectra.npz
  plots_step: 1         # plot every Nth spectrum
  # max_plots: 100      # optional cap
  root: "output"        # base output folder
  subdir_prefix: "run"  # subfolder prefix: run_<timestamp>
```

Simulation

> python scripts/simulation/simulate_single_spectra_yaml.py configs/simulate_spectra_only.yaml

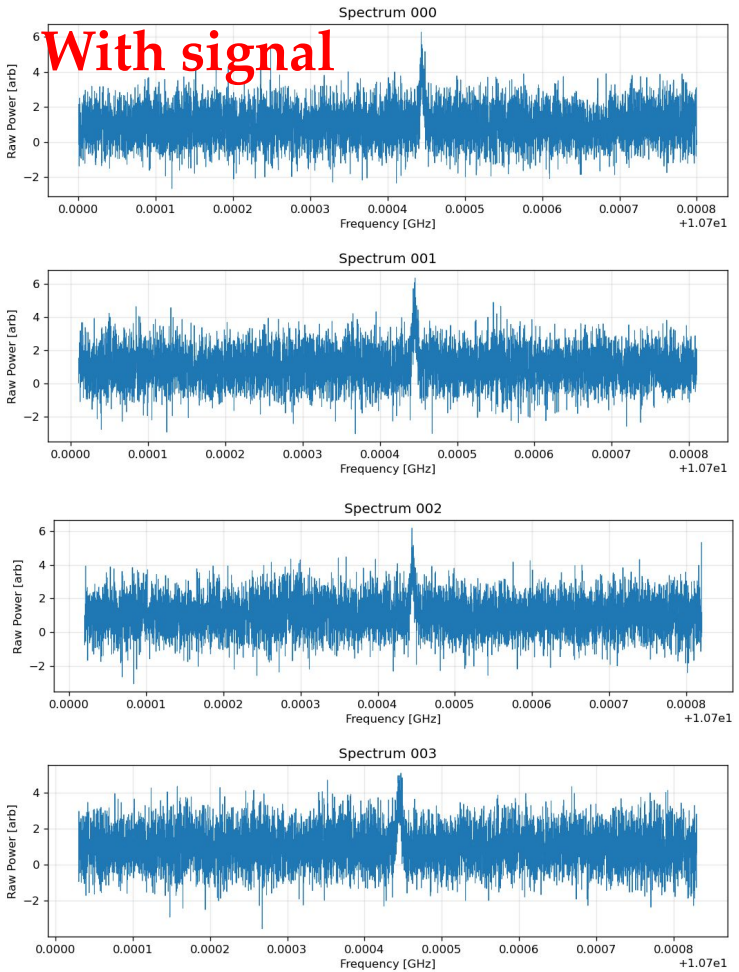
Output: N fine-grained noise spectra
number of spectra, starting frequency, bin width, axion injecti
is configurable



Simulation

> python scripts/simulation/simulate_single_spectra_yaml.py configs/simulate_spectra_only.yaml

Output: N fine-grained noise spectra
number of spectra, starting frequency, bin width, axion injection is configurable



Simulation: signal injection & functional baseline

We begin by generating synthetic power spectra that mimic haloscope data. Each spectrum represents the measured receiver power vs. frequency for a fixed tuning of the cavity

In `simulation.py`, the injected signal is currently a **Gaussian axion-like line added directly in RF frequency space**.

Each simulated raw spectrum is generated as baseline × noise/external background:

```
raw = baseline * (external + noise)
```

The relevant slice of the global axion signal is then added to each spectrum:

```
raw = raw + axion_power_global[idx]
```

where idx = rf_index_map[i], so each tuning only receives the part of the signal that lies inside its frequency coverage.

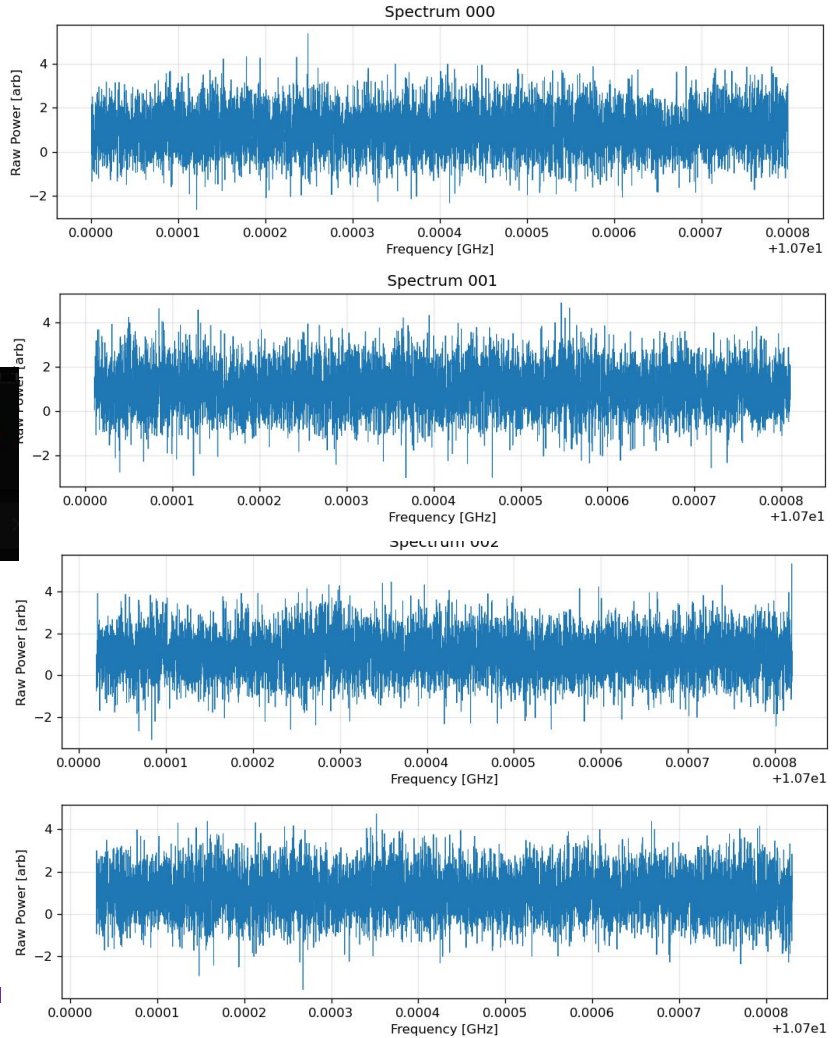
So the injection is additive power injection after the receiver/noise baseline is generated.
→ current implementation is a flat baseline, but Helen (MPhys) did implement different functional shapes

Data quality aka QC [quality control]

This step removes ENTIRE spectra in case they are flagged as BAD (under whatever condition TBD)

Default: keep all spectra

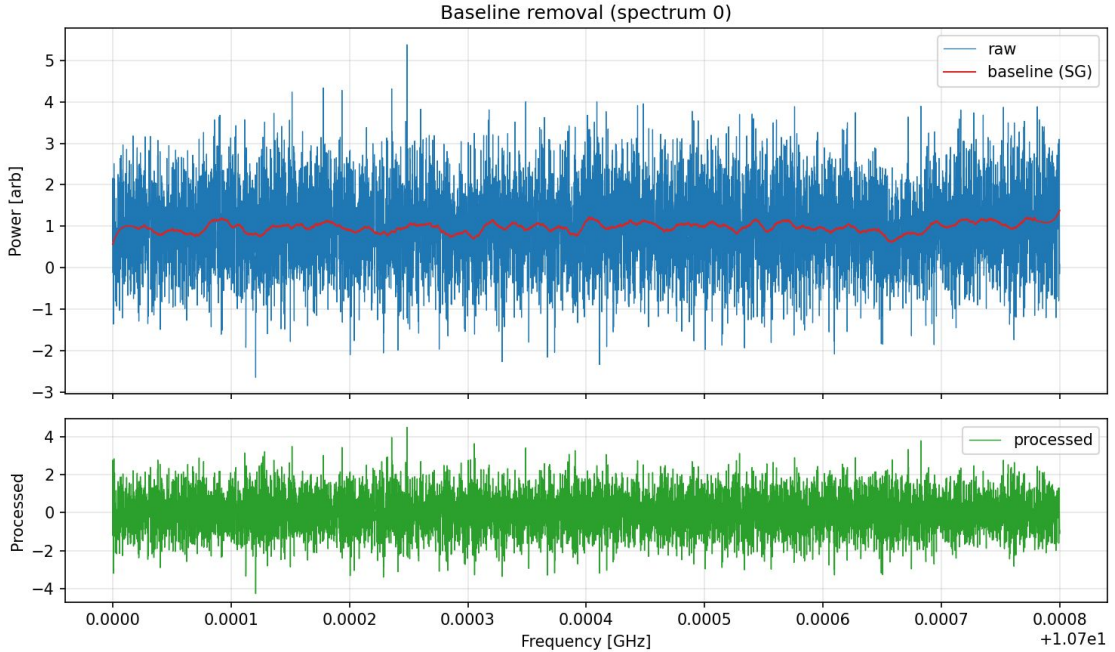
```
[QC] kept 10/10 spectra; dropped: []
[OK] Run dir: output/sim_spectra/run_20250909_085841
Candidates flagged: 25  (threshold = 1.64σ)
(MANCX_Env) eduroam-174072445-20:toy_m k00088eg$
```



Baseline removal filter

Every spectrum will have has two components mixed together:

1. **Broad, smooth baseline structure**
   → from amplifier gain ripple, cavity response, digitizer bandpass, etc.
   → Varies slowly with frequency (kHz–MHz scale).
2. **Fine, narrow features**
   → an axion line (a few kHz wide), or narrow RFI spikes.
   → Buried on top of the smooth background.



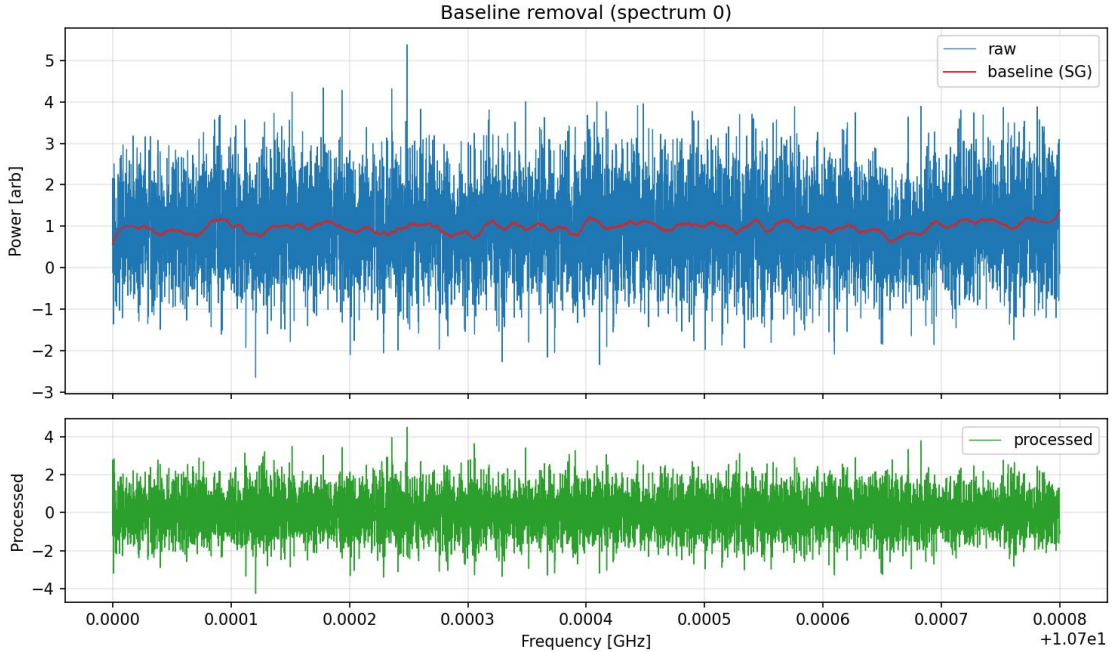
The **baseline filter** tries to separate these two:

→ Estimate the slowly-varying baseline
→ Divide (or subtract depending on mode) the raw spectrum by that baseline
→ Result is a flattened spectrum where noise has ~unit variance and any sharp excess (like an axion).

Baseline removal filter

Every spectrum will have has two components mixed together:

1. **Broad, smooth baseline structure**
   → from amplifier gain ripple, cavity response, digitizer bandpass, etc.
   → Varies slowly with frequency (kHz–MHz scale).
2. **Fine, narrow features**
   → an axion line (a few kHz wide), or narrow RFI spikes.
   → Buried on top of the smooth background.



The **baseline estimation w/ Savitzky–Golay (SG) filter**: a fancy rolling average.
To do: implement number of bins safeguards & quantify filter attenuation

Vertically combined spectrum (pre-rebinning)

combine.py takes many baseline-removed spectra and places them onto one common RF frequency grid.

Where spectra overlap in frequency, it combines them with maximum-likelihood / inverse-variance weighting. The file initializes arrays for the weighted sum, total weights, and number of contributing spectra per RF bin.

It estimates each spectrum's noise with `np.std(s)`.
Then each spectrum is added to the global grid using its `rf_index_map`

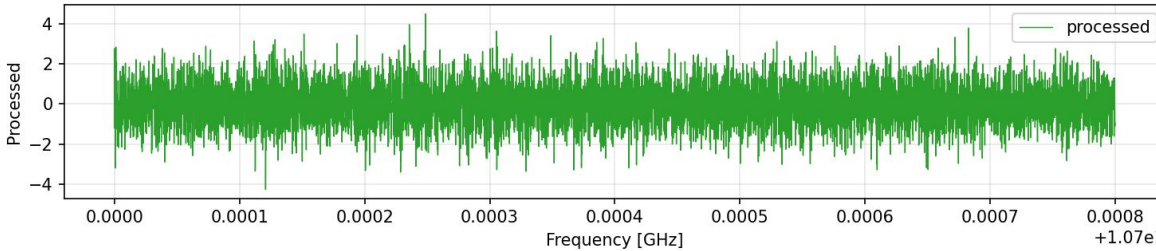
It returns three arrays

- `combined_spectrum` is the weighted average power excess on the global RF grid.
- `combined_sigma` is the estimated uncertainty per RF bin.
- `counts` tells how many spectra contributed to each RF bin.

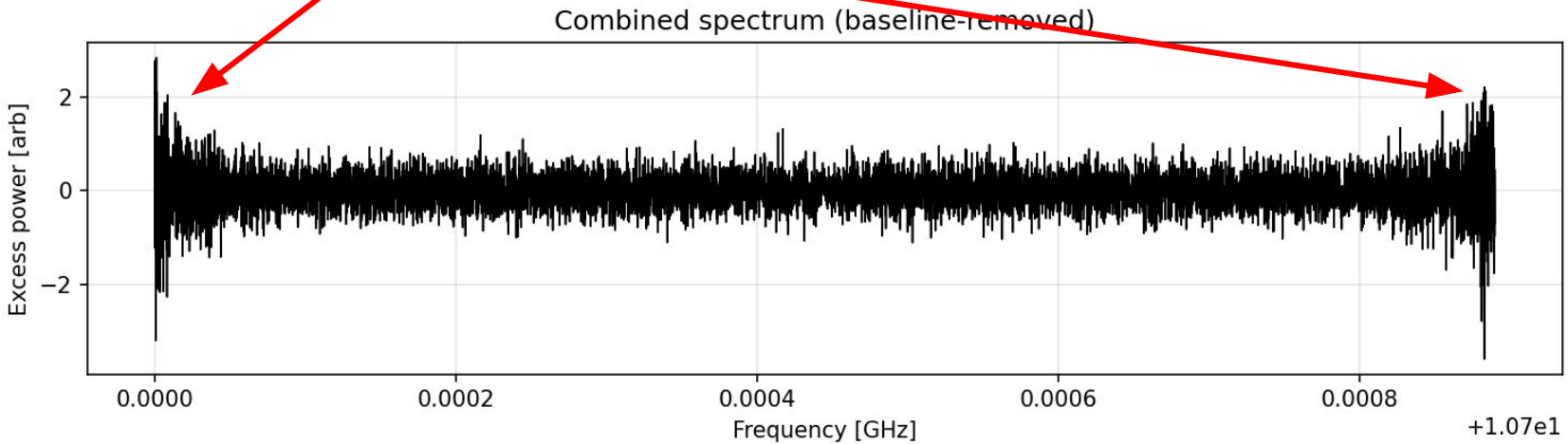
To do: add a signal rescaling functionality

Vertically combined spectrum (pre-rebinning)

Processed x10



shifting f_0 of raw spectra



Grand spectrum (post-rebinning + ML fit to SHM)

After combining all spectra, we rebin by factor C to reduce correlations. This gives:

→ D_r: rebinned power excess in each bin

→ σ_r: its standard deviation (noise estimate) {need to add visualizer}

Fit to Standard Halo Model

We slide a lineshape template *L* of length *K* (both can be changed) across the rebinned data

The lineshape template is where our expectation of the axion parameters comes in.

Assumptions:

→ Dark matter forms an **isothermal sphere** in the Milky Way halo.

→ The velocity distribution is a **Maxwell–Boltzmann** with dispersion $v_0 \approx 220$ km/s

→ There's an upper cutoff at the Galactic escape speed (~544 km/s).

→ On Earth, we see this boosted by our orbital speed through the halo (~232 km/s)

Grand spectrum (post-rebinning + ML fit to SHM)

After combining all spectra, we rebin by factor C to reduce correlations. This gives:

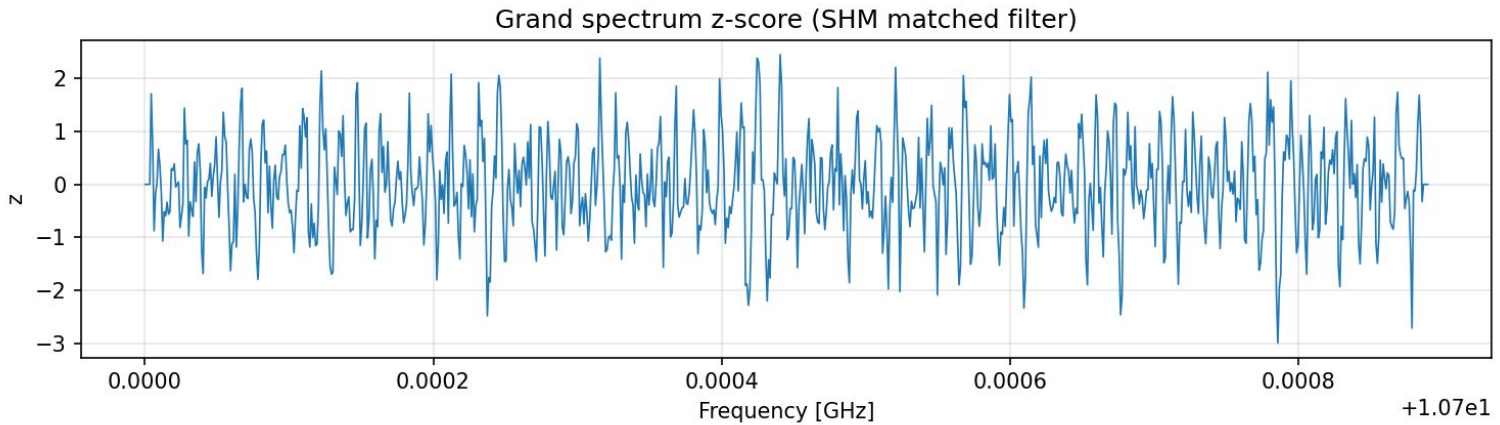
→ D_r: rebinned power excess in each bin

→ σ_r: its standard deviation (noise estimate) {need to add visualizer}

Fit to Standard Halo Model

We slide a lineshape template *L* of length *K* (both can be changed) across the rebinned data

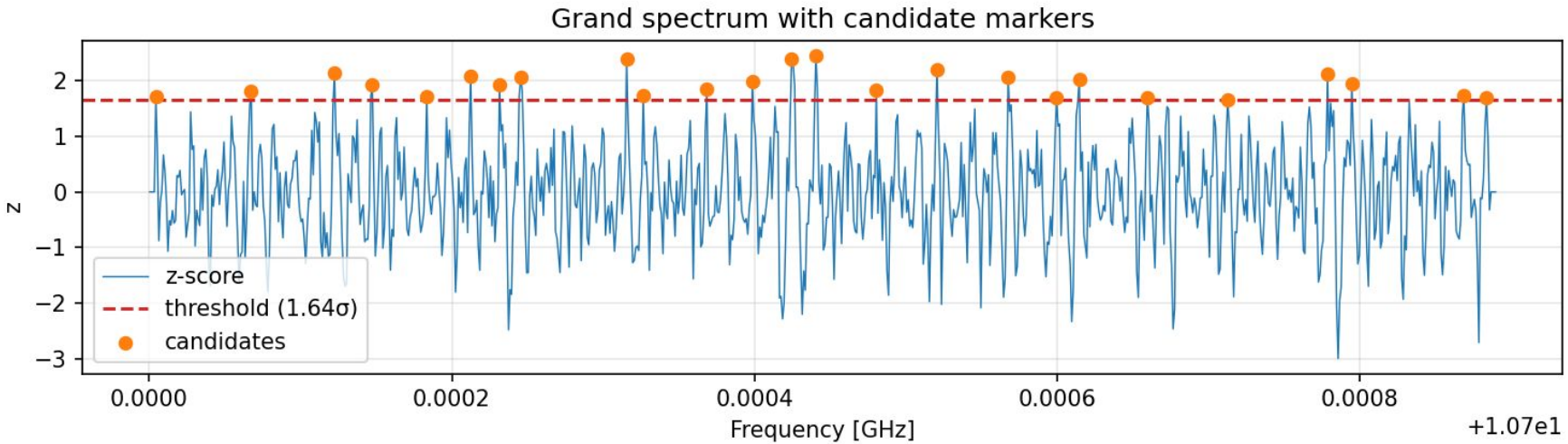
Calculate at each position the maximum-likelihood weighted sum Dg, its expected noise standard deviation σ_g and the test statistic z-score $z = D_g / \sigma_g$. Obtain the Grand Spectrum z-score



Candidates

The Grand spectrum is searched for candidates over a certain std threshold (configurable)

```
Candidates flagged: 25  (threshold = 1.64σ)
```

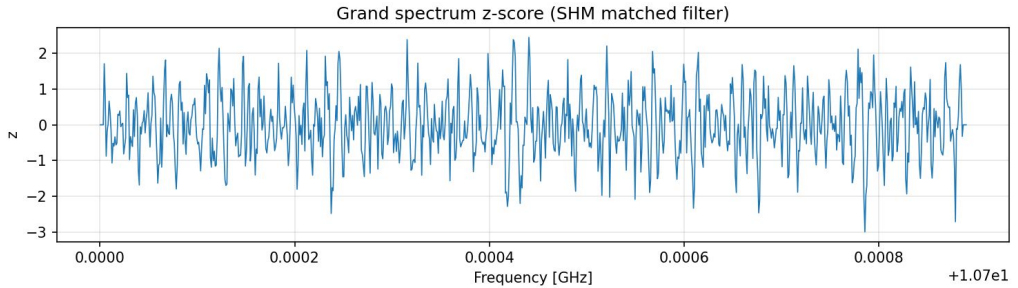


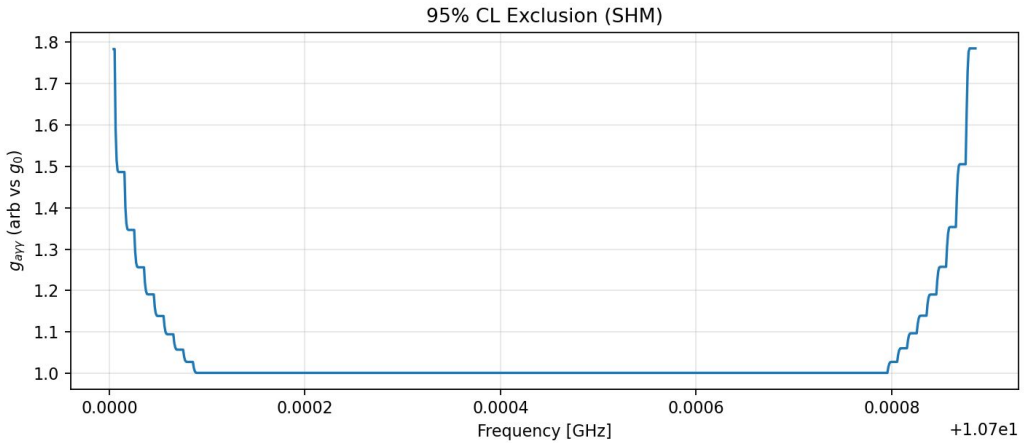
Exclusion

Currently, the code expresses limits relative to a reference coupling g_0
→ physics needed!
→ analysis pipeline is agnostic to the absolute scale (e.g. detector gain)
η = efficiency

$$g_{min}(f) = g_0 \times (SNR_{target} / (Response(f)\ \eta^2))^{0.5}$$

{right now just a translator
from z plot bins consistent w/ 0 and combined spectrum noise to a reference g_0}





One slide summary

simulation.py – Functions to generate synthetic spectra or to ingest real spectral data. Allows configuration of noise level, bin width, number of spectra, injected signals, etc.

baseline.py – Implements baseline removal (Savitzky–Golay filter). e.g. remove_baseline(spec, window, poly) returns a baseline-flattened spectrum. This module also contains utilities for identifying and masking bad bins (like known RFI spikes) before combination

combine.py – Functions for vertical combination of spectra. For example, combine_spectra(spec_list) takes a list of processed spectra and produces the combined spectrum with ML weighting. It handles frequency alignment and weight calculations (assuming each spectrum object carries information about its noise or uses internal estimates).

rebin.py – Tools for horizontal rebinning. It might define axion_lineshape = compute_lineshape(v_dispersion, ...) to get the template shape, and construct_grand_spectrum(combined_spec, lineshape) to perform the matched filtering (ML horizontal sum). This yields the grand spectrum ready for thresholding.

detection.py – Contains the logic for SNR calculation, threshold setting, and candidate identification. find_candidates(grand_spec, threshold) returns candidate indices. If rescan is enabled, rescan_and_confirm(candidates, data) will simulate additional data for each and re-check the grand spectrum (possibly using functions in the above modules).

limit.py – Functions to compute the coupling exclusion limit once the analysis is complete. For example, compute_limit(grand_spec, noise, target_snr) applies the relationship $g_{\min}(\nu) \propto \sqrt{\text{target SNR}}$ (accounting for any frequency-dependent noise or missing data).