qetpy Documentation

Release 1.1.0

Caleb Fink, Sam Watkins

Jul 26, 2022

CONTENTS

QETpy (Quasiparticle-trap-assisted Electrothermal-feedback Transition-edge sensors) provides tools for TES based detector calibration and analysis. It contains submodules for noise modeling, IV analysis, complex impedance fitting, non-linear optimum filter pulse fitting, and many other useful detector R&D analysis tools.

Descriptions of all the functions can be found below, as well as examples of how to use most of the functionality of QETpy.

This code is available on GitHub: https://github.com/spice-herald/QETpy

CHAPTER

ONE

INSTALLATION

For the most recent stable release, at the command line:

```
pip install qetpy
```

Or, if you want to work with the lastest development version. Clone the repository from: https://github.com/spice-herald/qetpy.git. Then within the repository, from the command line:

```
python setup.py clean
python setpy.py install --user
```

Note, you can check what version is available on PyPi by looking at the badge on the README on the GitHub homepage.

CHAPTER

TWO

QETPY.CORE PACKAGE

2.1 qetpy.core.didv module

2.2 qetpy.core._fitting module

2.3 qetpy.core._iv module

2.4 qetpy.core._noise module

CHAPTER THREE

QETPY.CUT PACKAGE

CHAPTER

FOUR

QETPY.PLOTTING PACKAGE

CHAPTER

FIVE

QETPY.UTILS PACKAGE

CHAPTER

SIX

QETPY.SIM PACKAGE

CHAPTER SEVEN

EXAMPLE USAGE

Jupyter notebooks providing detailed instructions and examples of how to use the various classes and functions of QETpy can be found below. The notebooks themselves (and corresponding data) can be downloaded from: https://github.com/spice-herald/QETpy/tree/master/demos

7.1 Example Code for using the $\partial I/\partial V$ Fitting Routines

Import the needed packages to run the test script.

```
import qetpy as qp
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(0)

%matplotlib inline
```

Set all of the necessary parameters for initalizing the class, as well as specify the TES parameters that we will use to simulate a TES square wave response.

```
# Setting various parameters that are specific to the dataset
rsh = 5e-3
rbias_sg = 20000
fs = 625e3
sgfreq = 100
sgamp = 0.009381 / rbias_sg

rfb = 5000
loopgain = 2.4
drivergain = 4
adcpervolt = 65536 / 2
tracegain = rfb * loopgain * drivergain * adcpervolt

true_params = {
    'rsh': rsh,
    'rp': 0.006,
    'r0': 0.0756,
    'beta': 2,
    'l': 10,
    'L': 1e-7,
```

(continues on next page)

(continued from previous page)

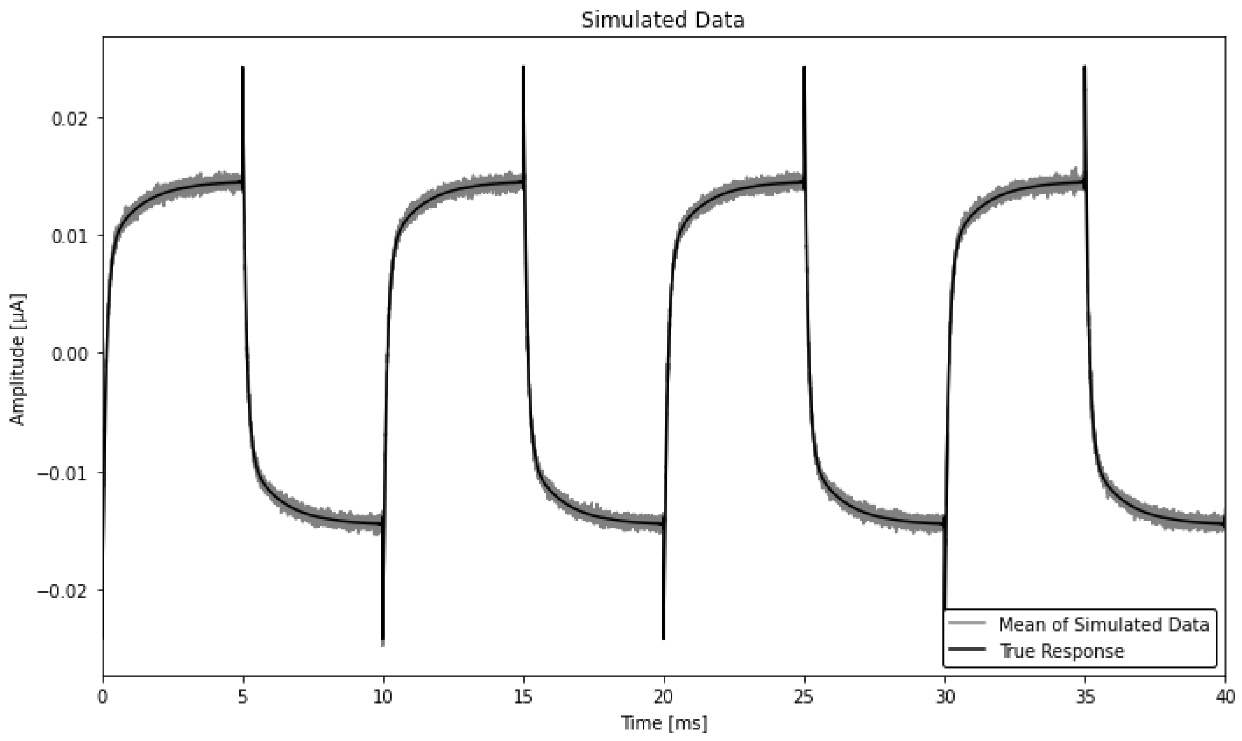
```
    'tau0': 500e-6,
    'gratio': 0.5,
    'tau3': 1e-3,
}
```

Make simulated data, where we simply have a TES square response with added white noise (not exactly physical, but this is just a demo!).

```
psd_test = np.ones(int(4 * fs / sgfreq)) / tracegain**2 / 1e4
rawnoise = qp.gen_noise(psd_test, fs=fs, ntraces=300)
```

```
t = np.arange(rawnoise.shape[-1]) / fs
didv_response = qp.squarewaveresponse(
    t, sgamp, sgfreq, **true_params,
)
rawtraces = didv_response + rawnoise
```

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(
    t * 1e3,
    rawtraces.mean(axis=0) * 1e6,
    color='gray',
    label='Mean of Simulated Data',
)
ax.plot(
    t * 1e3,
    didv_response * 1e6,
    color='k',
    label='True Response',
)
ax.set_ylabel('Amplitude [A]')
ax.set_xlabel('Time [ms]')
ax.set_title('Simulated Data')
ax.set_xlim(0, 40)
ax.legend(loc='lower right', edgecolor='k', framealpha=1)
fig.tight_layout()
```



7.1.1 Using the `DIDV` Class

Run the processing package on the data.

Note that the parameterization used by this class is such that there are no degenerate fitting parameters. Depending on the fit, the model changes.

From the Notes in `DIDV.dofit`:

```
Notes
-----
Depending on the fit, there are three possible models to be
used with different parameterizations:

1-pole model
    - has the form:
        dV/dI = A * (1.0 + 2.0j * pi * freq * tau2)

2-pole model
    - has the form:
        dV/dI = A * (1.0 + 2.0j * pi * freq * tau2)
              + B / (1.0 + 2.0j * pi * freq * tau1)

3-pole model
    - note the placement of the parentheses in the last term of
      this model, such that pole related to `C` is in the
      denominator of the `B` term
    - has the form:
        dV/dI = A * (1.0 + 2.0j * pi * freq * tau2)
```

(continues on next page)

(continued from previous page)

```
              + B / (1.0 + 2.0j * pi * freq * tau1
              - C / (1.0 + 2.0j * pi * freq * tau3))
```

```
didvfit = qp.DIDV(
    rawtraces,
    fs,
    sgfreq,
    sgamp,
    rsh,
    tracegain=1.0,
    r0=true_params['r0'], # the expected r0 should be specified if the estimated small-
↪signal parameters are desired (otherwise they will be nonsensical)
    rp=true_params['rp'], # the expected rp should be specified if the estimated small-
↪signal parameters are desired (otherwise they will be nonsensical)
    dt0=-1e-6, # a good estimate of the time shift value will likely speed up the fit/
↪improve accuracy
    add180phase=False, # if the fits aren't working, set this to True to see if the␣
↪square wave is off by half a period
)

# didvfit.dofit(1) # we skip the 1-pole fit, as it takes a long time to run due to being␣
↪a bad model.
didvfit.dofit(2)
didvfit.dofit(3)
```

Let's look at the fit parameters for the fits.

```
result2 = didvfit.fitresult(2)
result3 = didvfit.fitresult(3)
```

Each of these `result` variables are dictionaries that contain various information that have to do with the fits, as shown by looking at the keys.

```
result3.keys()
```

```
dict_keys(['params', 'cov', 'errors', 'smallsignalparams', 'falltimes', 'cost', 'didv0'])
```

- `'params'` contains the fitted parameters from the minimization method (in the case of `DIDV`, this is in the parameterization used by the fitting algorithm)
- `'cov'` contains the corresponding covariance matrix
- `'errors'` is simply the square root of the diagonal of the covariance matrix
- `'smallsignalparams'` contains the corresponding parameters in the small-signal parameterization of the complex impedance, as shown by Irwin and Hilton for the two-pole model and Maasilta for the three-pole model.
- `'falltimes'` contains the physical fall times of the specified model
- `'cost'` is the value of the chi-square at the fitted values
- `'didv0'` is the zero frequency component of the $\partial I/\partial V$ fitted model.

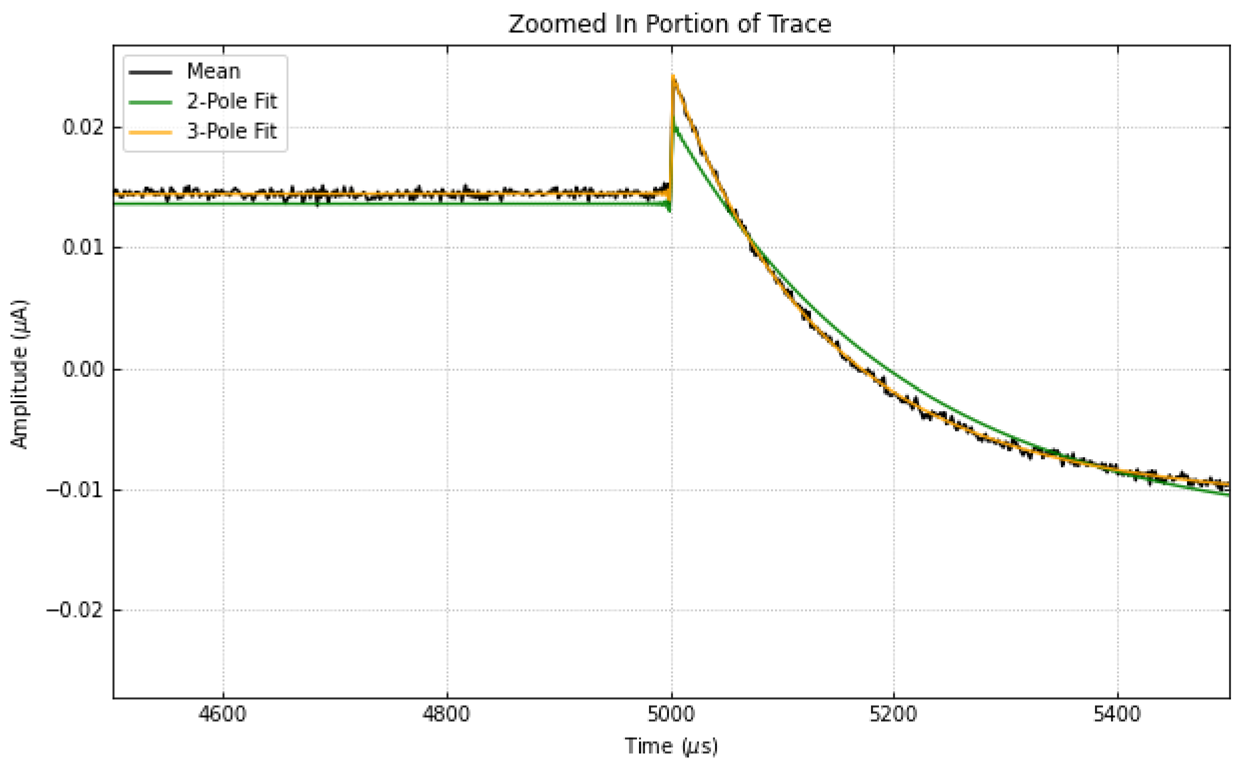
We can also use `qetpy.complexadmittance` along with the `'smallsignalparams'` dictionary to quickly calculate the zero-frequency component of the $\partial I/\partial V$.

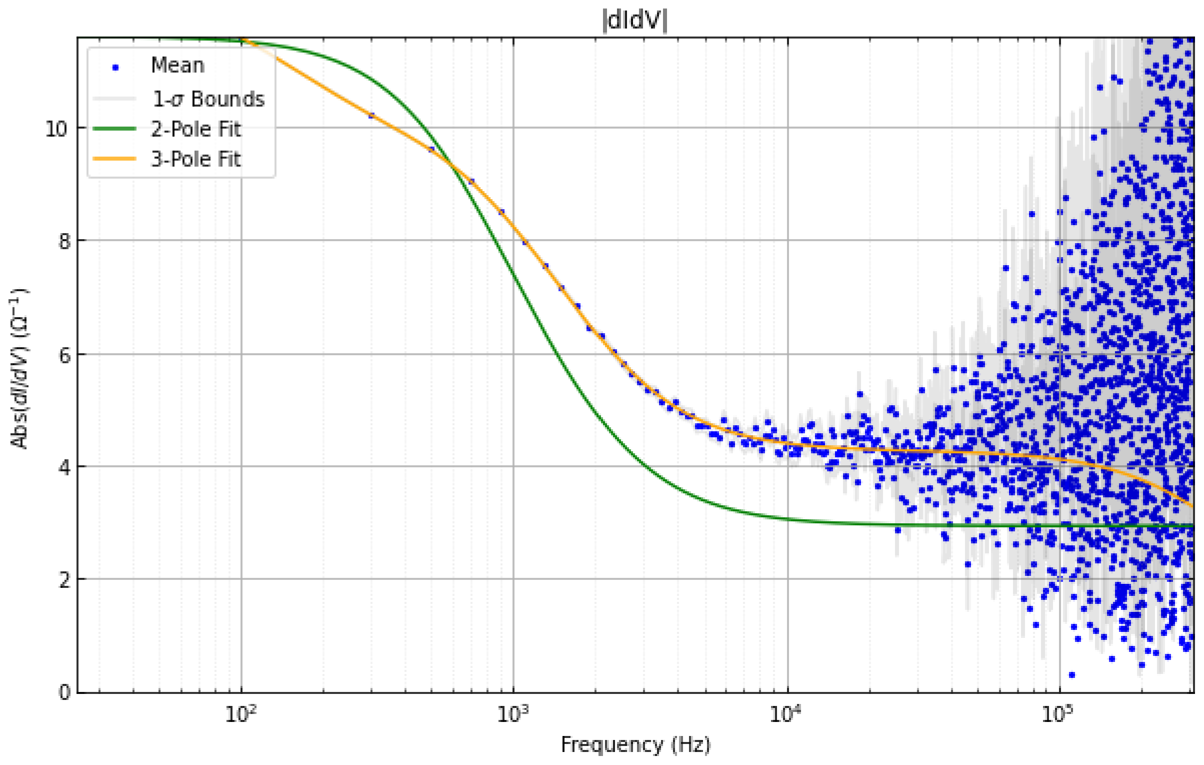
```
print(f"The 2-pole dI/dV(0) is: {result2['didv0']:.2f}")
print(f"The 3-pole dI/dV(0) is: {result3['didv0']:.2f}")
```

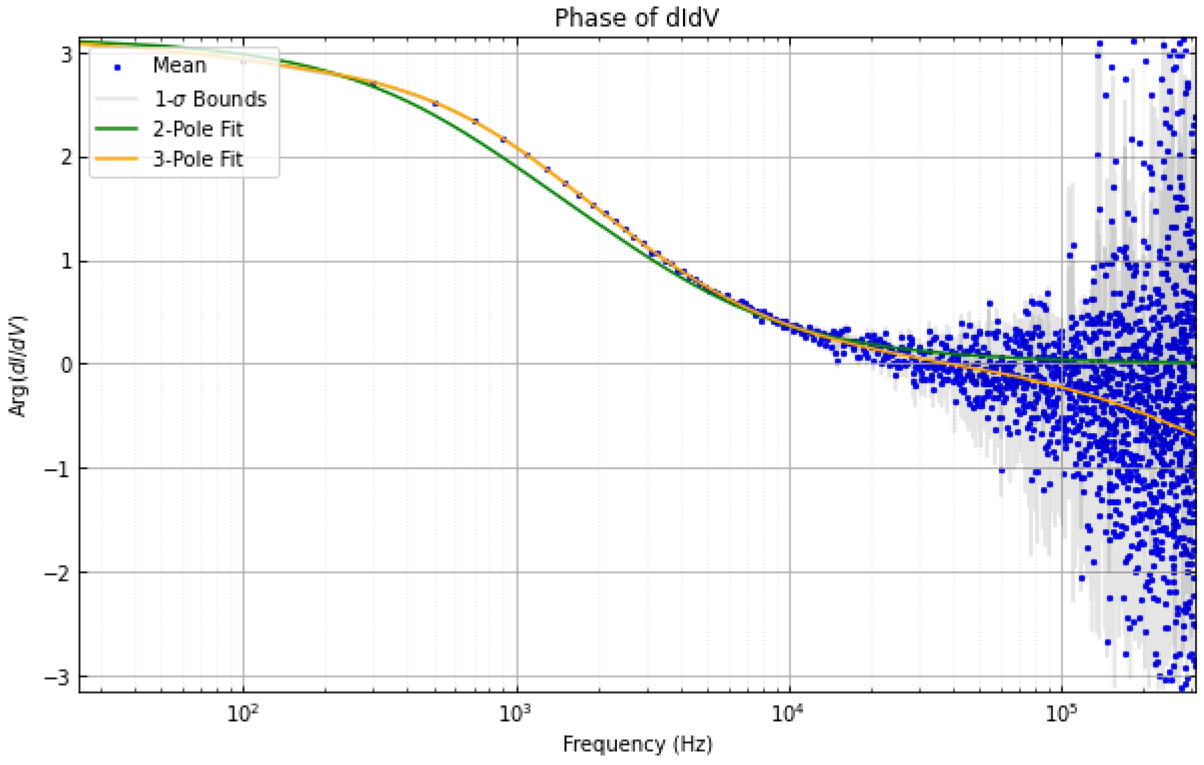
```
The 2-pole dI/dV(0) is: -11.61
The 3-pole dI/dV(0) is: -12.43
```

There are a handful of plotting functions that can be used to plot the results, where we use two of them below.

```
# didvfit.plot_full_trace()
# didvfit.plot_single_period_of_trace()
# didvfit.plot_didv_flipped()
# didvfit.plot_re_vs_im_dvdi()
# didvfit.plot_re_im_didv()

didvfit.plot_zoomed_in_trace(zoomfactor=0.1)
didvfit.plot_abs_phase_didv()
```




|dIdV|
Mean
1-σ Bounds
2-Pole Fit
3-Pole Fit
Abs(dI/dV) (Ω⁻¹)
Frequency (Hz)


Phase of dIdV
Mean
1-σ Bounds
2-Pole Fit
3-Pole Fit
Arg(dI/dV)
Frequency (Hz)

7.1.2 Advanced Usage: Using `DIDVPriors`

There is a separate class, which uses the small-signal parameterization to do the fitting. However, the small-signal parameterization has more parameters than degrees of freedom, resulting in degeneracy in the model (as opposed to the models used by `DIDV`, which are non-degenerate). However, if the uncertainties on the small-signal parameters are desired, then the `DIDVPriors` class is recommended, which uses a prior probability distribution on known values to remove the degeneracies.

The initialization of the class is quite similar.

```
didvfitprior = qp.DIDVPriors(
    rawtraces,
    fs,
    sgfreq,
    sgamp,
    rsh,
    tracegain=1.0,
    dt0=-18.8e-6,
)
```

Running the fit requires more information, as we need to know the prior probability distribution. In this case, `priors` and `priorscov` are additionally required in the `dofit` method.

- `priors` is a 1-d ndarray that contains the prior known values of the specified model.
- `priorscov` is a 2-d ndarray that contains the corresponding covariance matrix of the values in `priors`
  - For any values that are not known, the corresponding elements should be set to zero for both `priors` and `priorscov`.

The values of `priors` and `priorscov` for each model are: - 1-pole - `priors` is a 1-d array of length 4 containing (`rsh`, `rp`, `L`, `dt`) **(in that order!)** - `priorscov` is a 2-d array of shape (4, 4) containing the corresponding covariance matrix - 2-pole - `priors` is a 1-d array of length 8 containing (`rsh`, `rp`, `r0`, `beta`, `l`, `L`, `tau0`, `dt`) **(in that order!)** - `priorscov` is a 2-d array of shape (8, 8) containing the corresponding covariance matrix - 3-pole - `priors` is a 1-d array of length 10 containing (`rsh`, `rp`, `r0`, `beta`, `l`, `L`, `tau0`, `gratio`, `tau3`, `dt`) **(in that order!)** - `priorscov` is a 2-d array of shape (10, 10) containing the corresponding covariance matrix

We note that, the more parameters passed to the fitting algorithm, the ‘better’ the result will be, as degeneracies will be removed.

At minimum, we recommend passing `rsh`, `rp`, and `r0` (the last of which should not be passed to the 1-pole fit).

Below, we show an example of using the fitting algorithm for the 3-pole case, assuming 10% errors on `rsh`, `rp`, and `r0`.

```
priors = np.zeros(10)
priorscov = np.zeros((10, 10))

priors[0] = true_params['rsh']
priorscov[0, 0] = (0.1 * priors[0])**2
priors[1] = true_params['rp']
priorscov[1, 1] = (0.1 * priors[1])**2
priors[2] = true_params['r0']
priorscov[2, 2] = (0.1 * priors[2])**2

didvfitprior.dofit(3, priors, priorscov)
```

Extracting the results of the fit is very similar, where a dictionary is returned, which has slightly different form than `DIDV`.

```
result3_priors = didvfitprior.fitresult(3)
result3_priors.keys()
```

```
dict_keys(['params', 'cov', 'errors', 'falltimes', 'cost', 'didv0', 'priors', 'priorscov
↪'])
```

- `'params'` contains the fitted parameters from the minimization method (which are the small-signal parameters), as shown by Irwin and Hilton for the two-pole model and Maasilta for the three-pole model.
- `'cov'` contains the corresponding covariance matrix
- `'errors'` is simply the square root of the diagonal of the covariance matrix
- `'falltimes'` contains the physical fall times of the specified model
- `'cost'` is the value of the chi-square at the fitted values
- `'didv0'` is the zero frequency component of the $\partial I/\partial V$ fitted model.
- `'priors'` simply contains the inputted priors
- `'priorscov'` simply contains the inputted priors covariance matrix

Note the lack of `smallsignalparams`, as compared to `DIDV`. Since we fit directly using the small-signal parameterization, there is no need to convert parameters.

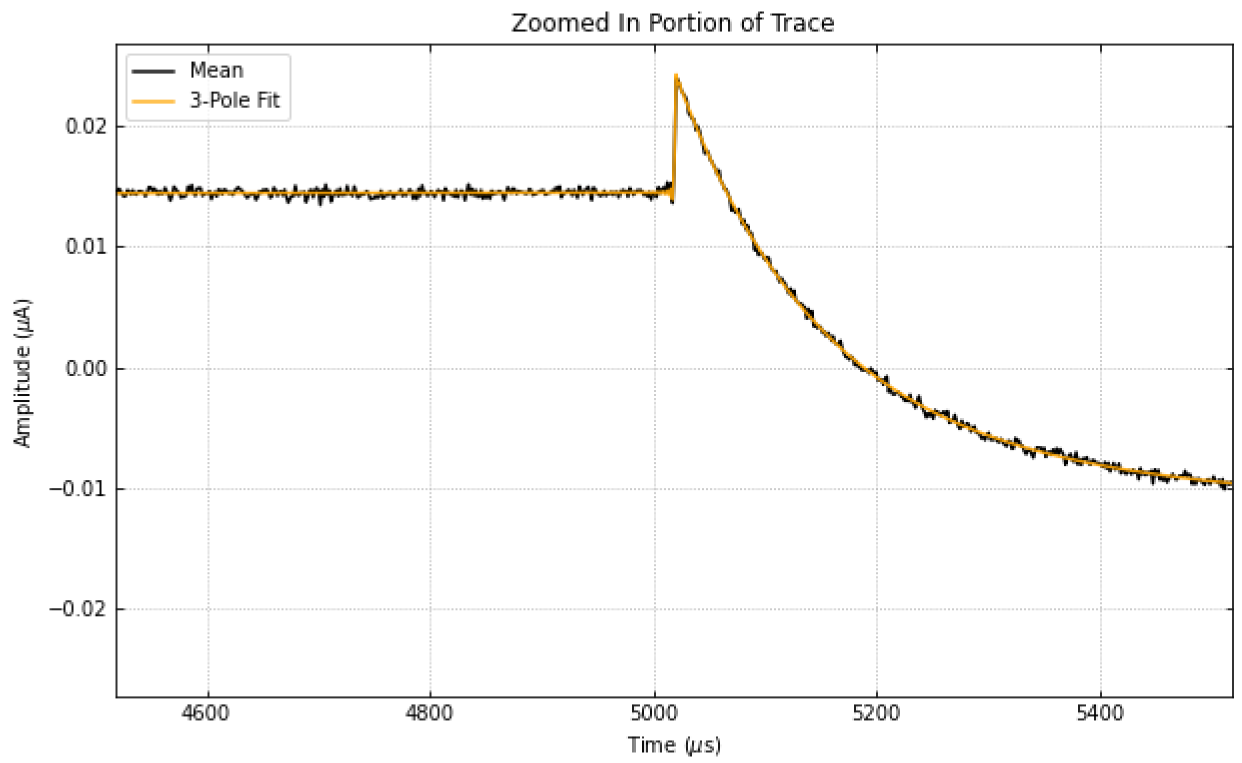
We can again calculate the zero-frequency component of the $\partial I/\partial V$ using `qetpy.complexadmittance`, this time using the `'params'` key in the results dictionary.

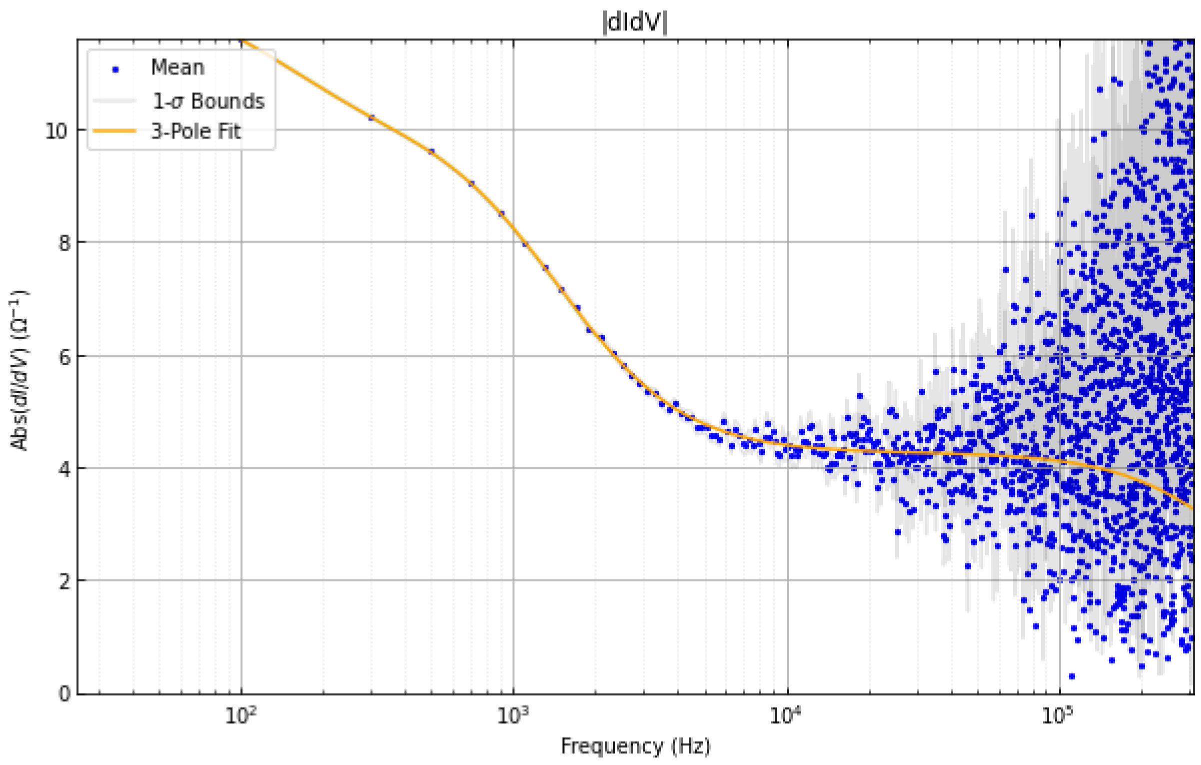
```
print(f"The 3-pole dI/dV(0) is: {result3_priors['didv0']:.2f}")
```

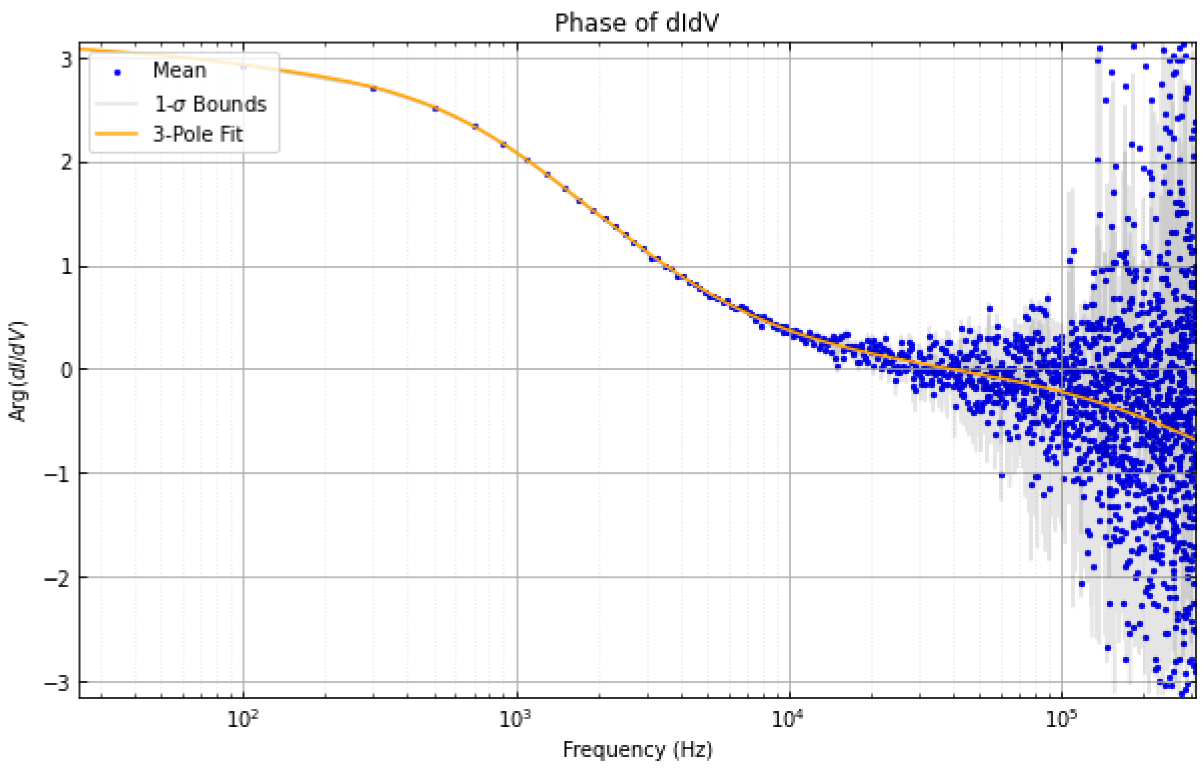
```
The 3-pole dI/dV(0) is: -12.43
```

This class uses the same plotting functions as `DIDV`, such that they can be called in the same way.

```
didvfitprior.plot_zoomed_in_trace(zoomfactor=0.1)
didvfitprior.plot_abs_phase_didv()
```


Zoomed In Portion of Trace
Mean
3-Pole Fit
0.02
0.01
0.00
−0.01
−0.02
Amplitude (μA)
4600
4800
5000
5200
5400
Time (μs)


|dIdV|
Mean
1-σ Bounds
3-Pole Fit
10
8
6
4
2
0
Abs(dI/dV) (Ω⁻¹)
10²
10³
10⁴
10⁵
Frequency (Hz)



7.2 Using the `autocuts` and `IterCut` Algorithms

This is a quick look at how to use the `autocuts` and the `IterCut` algorithms. The `autocuts` function acts as a black box (the user cannot see what is going on under the hood), while `IterCut` allows the user to understand each cut being applied to data. For quick results, `autocuts` is usually sufficient, but `IterCut` is very useful to actually understand what is happening.

Note that there are many more optional arguments than what are shown here in the notebook. As always, we recommend reading the docstrings!

First, let's import our functions.

```
import numpy as np
import matplotlib.pyplot as plt
from qetpy import autocuts, calc_psd, IterCut

# ensure that the notebook is repeatable by using the same random seed
np.random.seed(1)
```

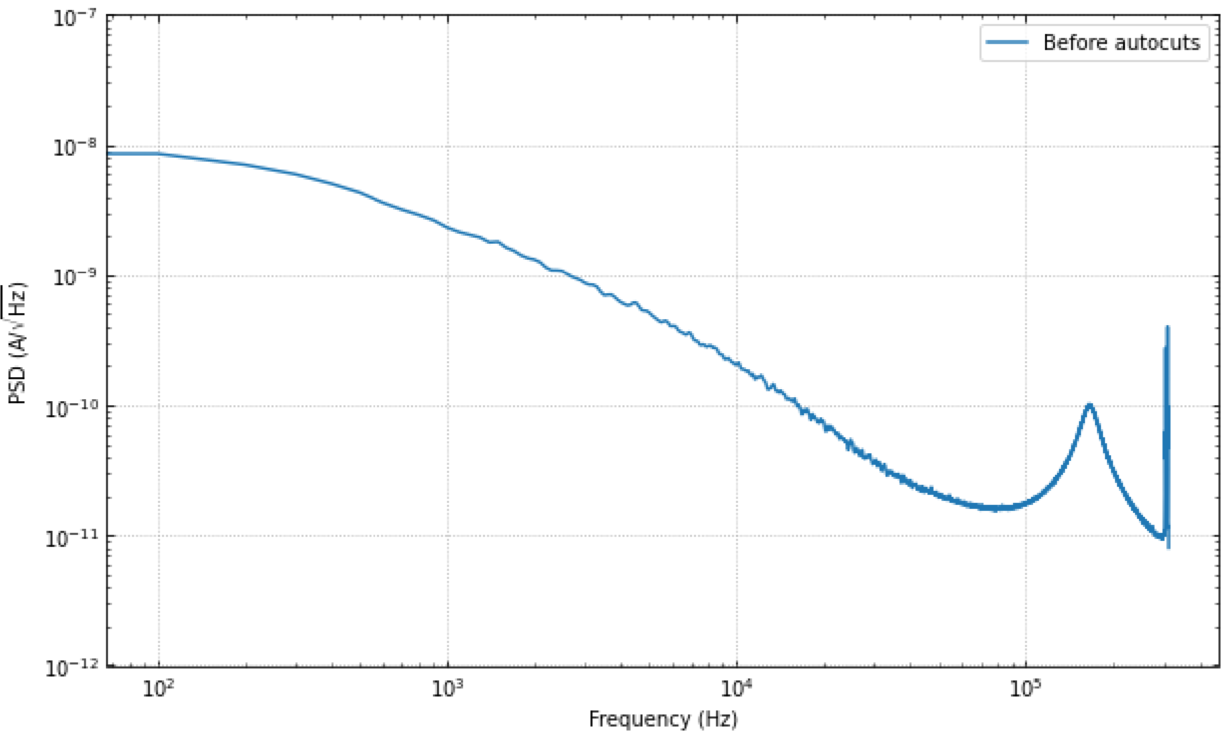
Now, let's load the data.

```
pathtodata = "test_autocuts_data.npy"
traces = np.load(pathtodata)
fs = 625e3
```

Let's look at the PSD before the cuts, to get a sense of the change.

```
f, psd = calc_psd(traces, fs=fs, folded_over=True)
```

```
fig, ax = plt.subplots(figsize=(10,6))
ax.loglog(f, np.sqrt(psd), label="Before autocuts")
ax.set_ylim([1e-12,1e-7])
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r'PSD (A/$\sqrt{\mathrm{Hz}}$)')
ax.legend(loc="upper right")
ax.grid(linestyle='dotted')
ax.tick_params(which='both',direction='in',right=True,top=True)
```



7.2.1 Using `autocuts`

Apply the autocuts function.

```
?autocuts
```

```
Function to automatically cut out bad traces based on the optimum
filter amplitude, slope, baseline, and chi^2 of the traces.

Parameters
----------
traces : ndarray
    2-dimensional array of traces to do cuts on
fs : float, optional
    Sample rate that the data was taken at
is_didv : bool, optional
```

(continues on next page)

(continued from previous page)

```
    Boolean flag on whether or not the trace is a dIdV curve
outlieralgo : string, optional
    Which outlier algorithm to use. If set to "removeoutliers",
    uses the removeoutliers algorithm that removes data based on
    the skewness of the dataset. If set to "iterstat", uses the
    iterstat algorithm to remove data based on being outside a
    certain number of standard deviations from the mean. Can also
    be set to astropy's "sigma_clip".
lgcpileup1 : boolean, optional
    Boolean value on whether or not do the pileup1 cut (this is the
    initial pileup cut that is always done whether or not we have
    dIdV data). Default is True.
lgcslope : boolean, optional
    Boolean value on whether or not do the slope cut. Default is
    True.
lgcbaseline : boolean, optional
    Boolean value on whether or not do the baseline cut. Default is
    True.
lgcpileup2 : boolean, optional
    Boolean value on whether or not do the pileup2 cut (this cut is
    only done when is_didv is also True). Default is True.
lgcchi2 : boolean, optional
    Boolean value on whether or not do the chi2 cut. Default is
    True.
nsigpileup1 : float, optional
    If outlieralgo is "iterstat", this can be used to tune the
    number of standard deviations from the mean to cut outliers
    from the data when using iterstat on the optimum filter
    amplitudes. Default is 2.
nsigslope : float, optional
    If outlieralgo is "iterstat", this can be used to tune the
    number of standard deviations from the mean to cut outliers
    from the data when using iterstat on the slopes. Default is 2.
nsigbaseline : float, optional
    If outlieralgo is "iterstat", this can be used to tune the
    number of standard deviations from the mean to cut outliers
    from the data when using iterstat on the baselines. Default is
    2.
nsigpileup2 : float, optional
    If outlieralgo is "iterstat", this can be used to tune the
    number of standard deviations from the mean to cut outliers
    from the data when using iterstat on the optimum filter
    amplitudes after the mean has been subtracted. (only used if
    is_didv is True). Default is 2.
nsigchi2 : float, optional
    This can be used to tune the number of standard deviations
    from the mean to cut outliers from the data when using iterstat
    on the chi^2 values. Default is 3.
**kwargs
    Placeholder kwargs for backwards compatibility.

Returns
```

(continues on next page)

(continued from previous page)

```
-------
ctot : ndarray
    Boolean array giving which indices to keep or throw out based
    on the autocuts algorithm.
```

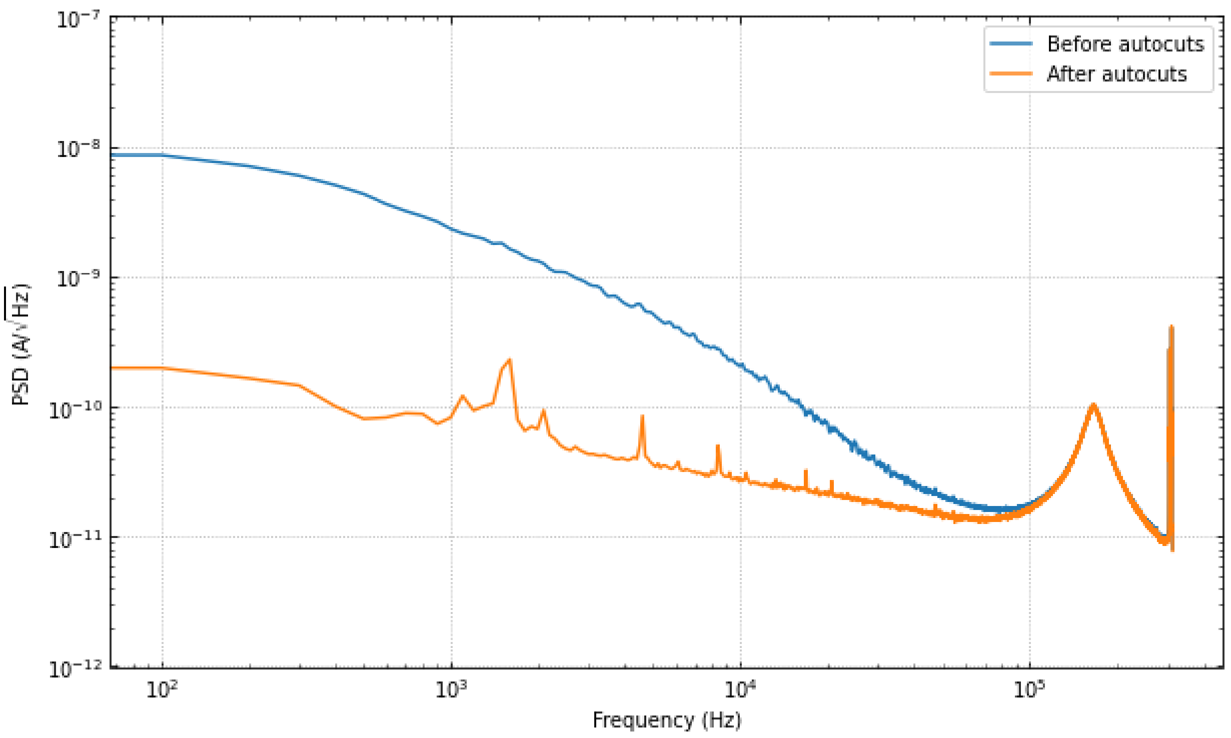
```
cut = autocuts(
    traces,
    fs=fs,
)
print(f"The cut efficiency is {np.sum(cut)/len(traces):.3f}.")
```

```
The cut efficiency is 0.428.
```

Let's compare the PSD after the cuts, we should see the noise go down by a fair amount.

```
psd_cut = calc_psd(traces[cut], fs=fs, folded_over=True)[1]
```

```
fig, ax = plt.subplots(figsize=(10,6))
ax.loglog(f, np.sqrt(psd), label="Before autocuts")
ax.loglog(f, np.sqrt(psd_cut), label="After autocuts")
ax.set_ylim([1e-12,1e-7])
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r'PSD (A/$\sqrt{\mathrm{Hz}}$)')
ax.legend(loc="upper right")
ax.grid(linestyle='dotted')
ax.tick_params(which='both',direction='in',right=True,top=True)
```



The change is huge! Which makes sense, as we have removed many of the pulses, muon tails, etc. Please note that there may still be “bad” traces in the data, as the autocuts function is not perfect. There may be more cuts that one would like to do that are more specific to the dataset.

7.2.2 Using IterCut for better cut control

A good way of understanding the cuts further than using the black box that is `autocuts` is to use the object-oriented version `IterCut`. This class allows the user freedom in cut order, which cuts are used, which algorithms are used for outlier removal, and more.

Below, we match the default parameters and outlier algorithm (`iterstat`) to show that the cut efficiency is the same.

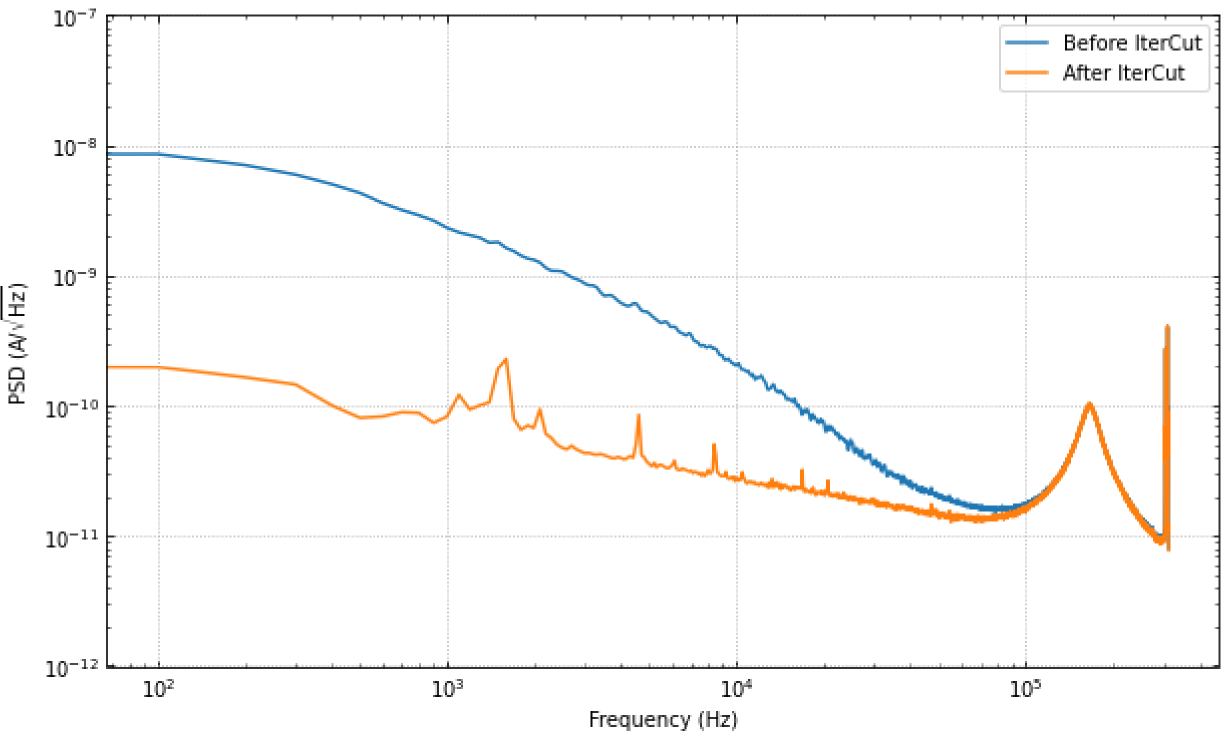
```
IC = IterCut(traces, fs)
IC.pileupcut(cut=2)
IC.slopecut(cut=2)
IC.baselinecut(cut=2)
IC.chi2cut(cut=3)
cut_ic = IC.cmask
```

```
print(f"The cut efficiency is {np.sum(cut_ic)/len(traces):.3f}.")
```

```
The cut efficiency is 0.428.
```

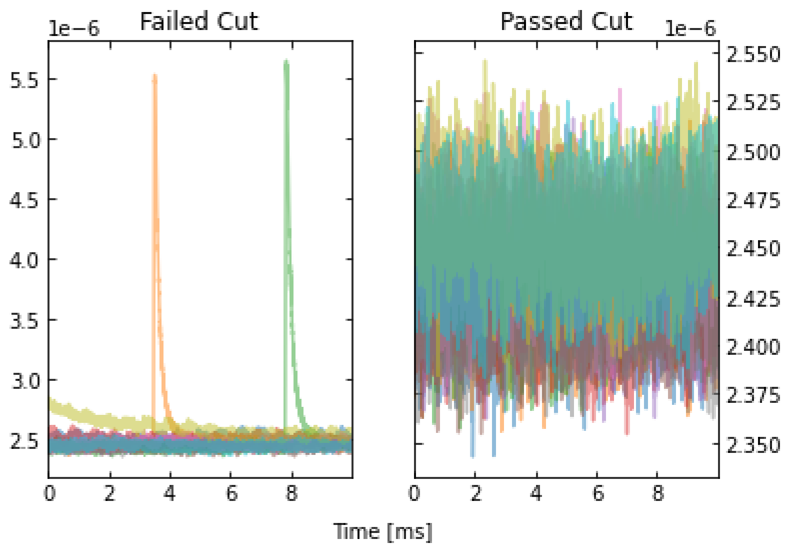
```
psd_cut = calc_psd(traces[cut_ic], fs=fs, folded_over=True)[1]
```

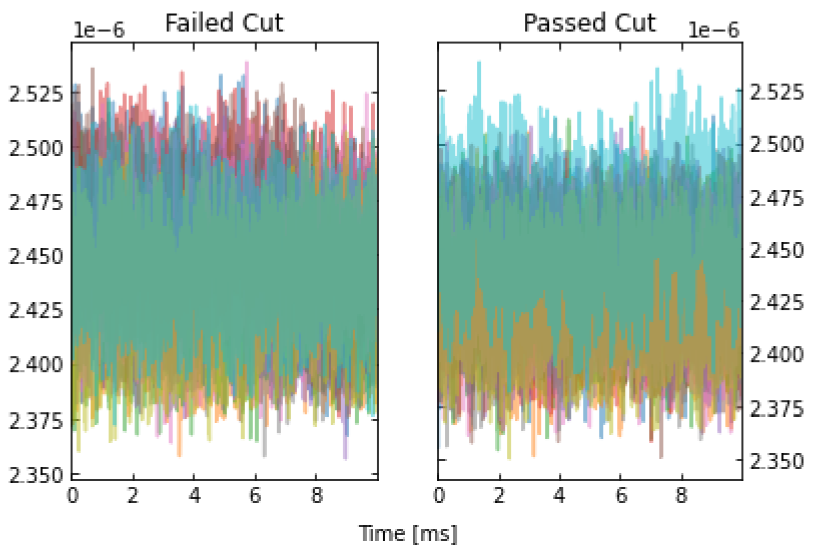
```
fig, ax = plt.subplots(figsize=(10,6))
ax.loglog(f, np.sqrt(psd), label="Before IterCut")
ax.loglog(f, np.sqrt(psd_cut), label="After IterCut")
ax.set_ylim([1e-12,1e-7])
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r'PSD (A/$\sqrt{\mathrm{Hz}}$)')
ax.legend(loc="upper right")
ax.grid(linestyle='dotted')
ax.tick_params(which='both',direction='in',right=True,top=True)
```

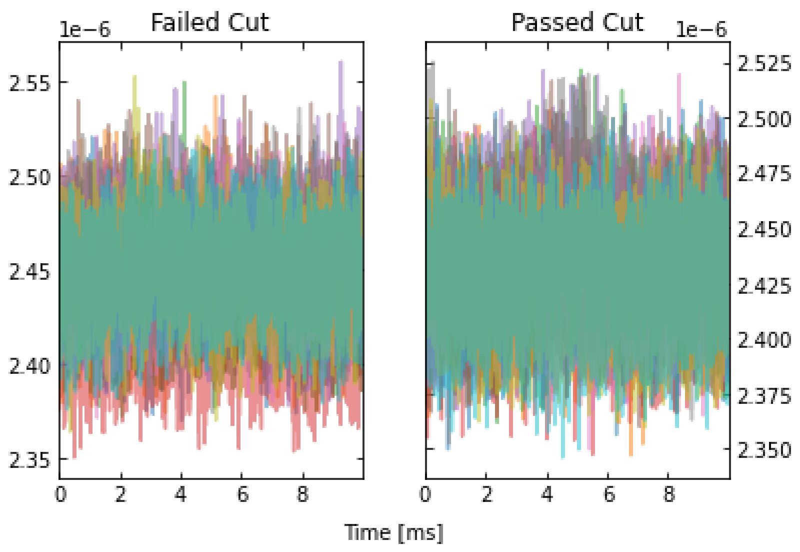


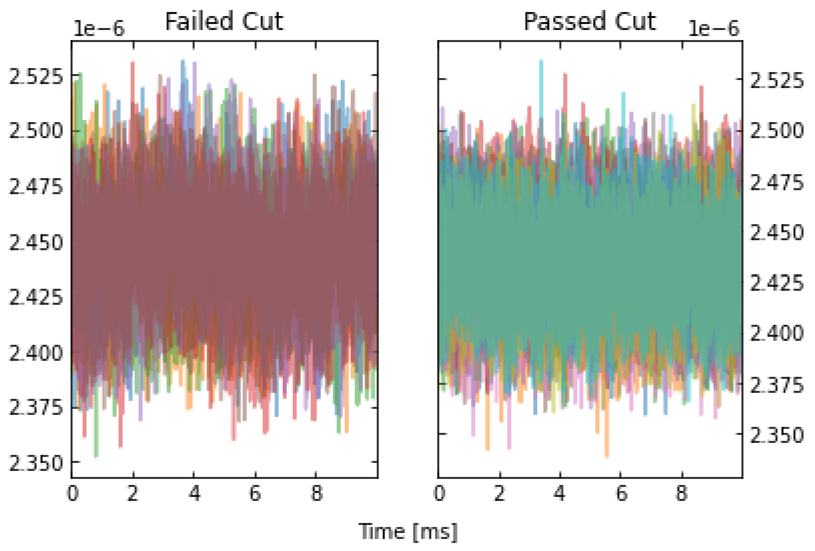
With `IterCut` we can also access the cuts at each step as they have been iteratively applied, and there is a verbose option for plotting the passing event and failing events for each cut.

```
IC = IterCut(traces, fs, plotall=True, nplot=10)
cpileup = IC.pileupcut(cut=2)
cpileup1 = IC.cmask
cslope = IC.slopecut(cut=2)
cbaseline = IC.baselinecut(cut=2)
cchi2 = IC.chi2cut(cut=3)
cut_ic = IC.cmask
```




Failed Cut
1e−6
2.525
2.500
2.475
2.450
2.425
2.400
2.375
2.350
0
2
4
6
8
Passed Cut
1e−6
2.525
2.500
2.475
2.450
2.425
2.400
2.375
2.350
Time [ms]


Failed Cut
1e−6
2.55
2.50
2.45
2.40
2.35
0
2
4
6
8
Passed Cut
1e−6
2.525
2.500
2.475
2.450
2.425
2.400
2.375
2.350
Time [ms]


Failed Cut
1e−6
2.525
2.500
2.475
2.450
2.425
2.400
2.375
2.350
0
2
4
6
8
Passed Cut
1e−6
2.525
2.500
2.475
2.450
2.425
2.400
2.375
2.350
Time [ms]

This allows to calculate the efficiency of each cut, and we can see what cuts are going the heavy lifting. Note the importance of the denominator being the number of events that passed the previous cuts when calculating these efficiencies. If we were to divide by the number of traces each time, then this would be the total cut efficiency up to that cut. Below, we show the individual performance of each cut.

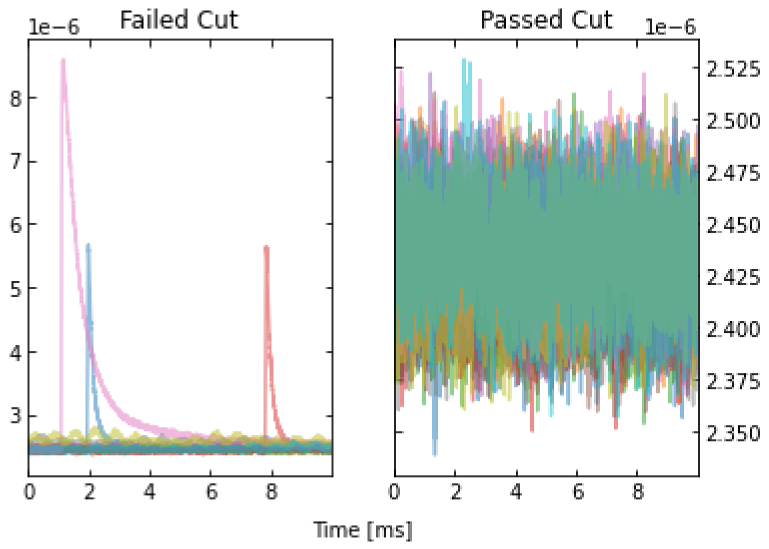
```
print(f"The pileup cut efficiency is {np.sum(cpileup)/len(traces):.3f}.")
print(f"The slope cut efficiency is {np.sum(cslope)/np.sum(cpileup):.3f}.")
print(f"The baseline cut efficiency is {np.sum(cbaseline)/np.sum(cslope):.3f}.")
print(f"The chi2 cut efficiency is {np.sum(cchi2)/np.sum(cbaseline):.3f}.")
print("-------------")
print(f"The total cut efficiency is {np.sum(cut_ic)/len(traces):.3f}.")
```

```
The pileup cut efficiency is 0.679.
The slope cut efficiency is 0.719.
The baseline cut efficiency is 0.889.
The chi2 cut efficiency is 0.986.
-------------
The total cut efficiency is 0.428.
```

Thus, we see that the pileup cut is has the lowest efficiency, with the slope cut as a close second. If we were to remove the baseline and chi-squared cuts, then we would expect the PSD to not change noticeably. Let's test this expectation.

Note that we can also plot the events passing/failing a specific cut by passing `verbose=True`, as shown below.

```
IC = IterCut(traces, fs)
cpileup = IC.pileupcut(cut=2, verbose=True)
cslope = IC.slopecut(cut=2)
cut_ic = IC.cmask
```



```
print(f"The pileup cut efficiency is {np.sum(cpileup)/len(traces):.3f}.")
print(f"The slope cut efficiency is {np.sum(cslope)/np.sum(cpileup):.3f}.")
print("-------------")
print(f"The total cut efficiency is {np.sum(cut_ic)/len(traces):.3f}.")
```

```
The pileup cut efficiency is 0.679.
```

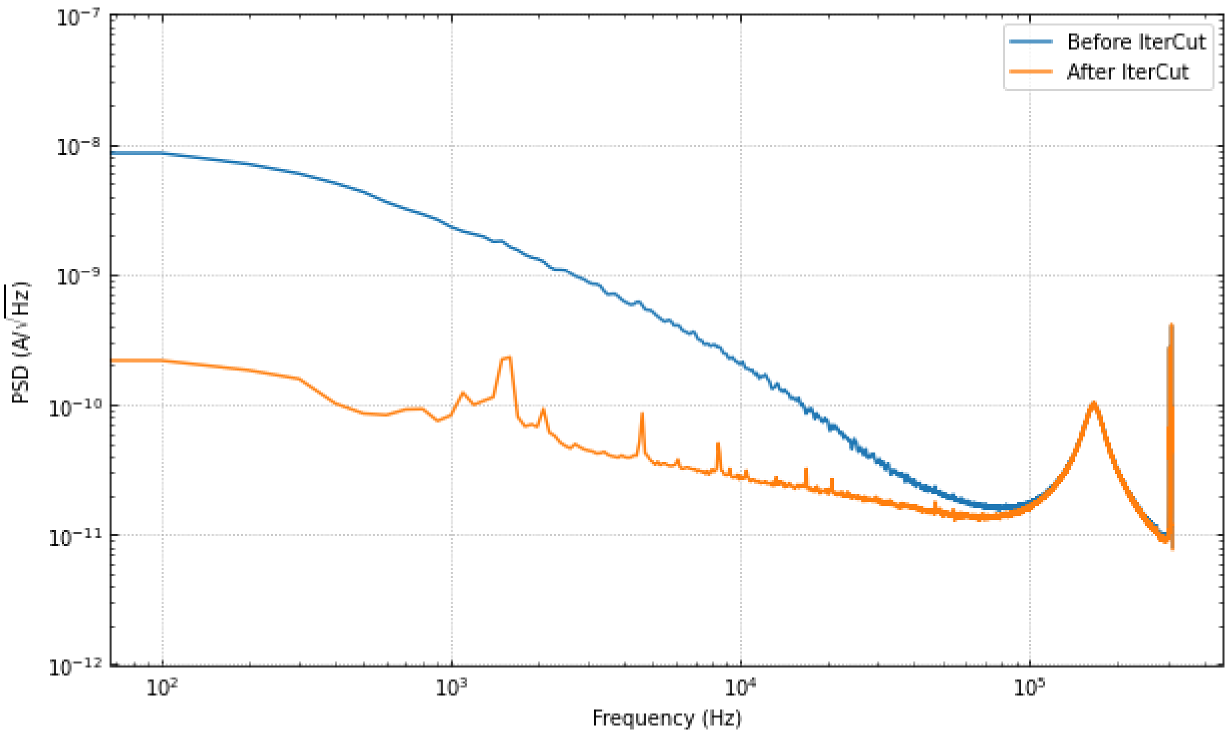
(continues on next page)

(continued from previous page)

```
The slope cut efficiency is 0.719.
-------------
The total cut efficiency is 0.488.
```

```
psd_cut = calc_psd(traces[cut_ic], fs=fs, folded_over=True)[1]
```

```
fig, ax = plt.subplots(figsize=(10,6))
ax.loglog(f, np.sqrt(psd), label="Before IterCut")
ax.loglog(f, np.sqrt(psd_cut), label="After IterCut")
ax.set_ylim([1e-12,1e-7])
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r'PSD (A/$\sqrt{\mathrm{Hz}}$)')
ax.legend(loc="upper right")
ax.grid(linestyle='dotted')
ax.tick_params(which='both',direction='in',right=True,top=True)
```



What if we reversed the cut order? How does this affect the cut efficiencies?

```
IC = IterCut(traces, fs, plotall=False)
cchi2 = IC.chi2cut(cut=3)
cbaseline = IC.baselinecut(cut=2)
cslope = IC.slopecut(cut=2)
cpileup = IC.pileupcut(cut=2)
cut_ic = IC.cmask
```

```
print(f"The chi2 cut efficiency is {np.sum(cchi2)/len(traces):.3f}.")
print(f"The baseline cut efficiency is {np.sum(cbaseline)/np.sum(cchi2):.3f}.")
```

(continues on next page)

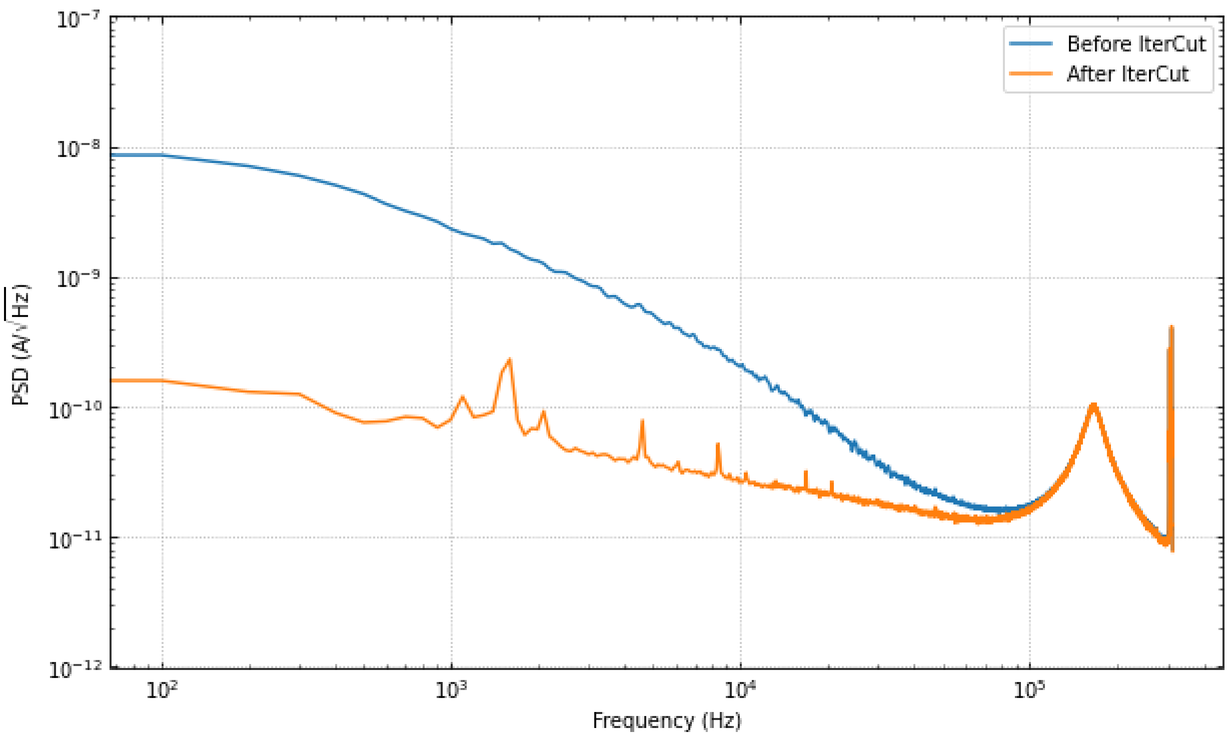
(continued from previous page)

```
print(f"The slope cut efficiency is {np.sum(cslope)/np.sum(cbaseline):.3f}.")
print(f"The pileup cut efficiency is {np.sum(cpileup)/np.sum(cslope):.3f}.")
print("-------------")
print(f"The total cut efficiency is {np.sum(cut_ic)/len(traces):.3f}.")
```

```
The chi2 cut efficiency is 0.840.
The baseline cut efficiency is 0.739.
The slope cut efficiency is 0.718.
The pileup cut efficiency is 0.706.
-------------
The total cut efficiency is 0.315.
```

```
psd_cut = calc_psd(traces[cut_ic], fs=fs, folded_over=True)[1]
```

```
fig, ax = plt.subplots(figsize=(10,6))
ax.loglog(f, np.sqrt(psd), label="Before IterCut")
ax.loglog(f, np.sqrt(psd_cut), label="After IterCut")
ax.set_ylim([1e-12,1e-7])
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel(r'PSD (A/$\sqrt{\mathrm{Hz}}$)')
ax.legend(loc="upper right")
ax.grid(linestyle='dotted')
ax.tick_params(which='both',direction='in',right=True,top=True)
```



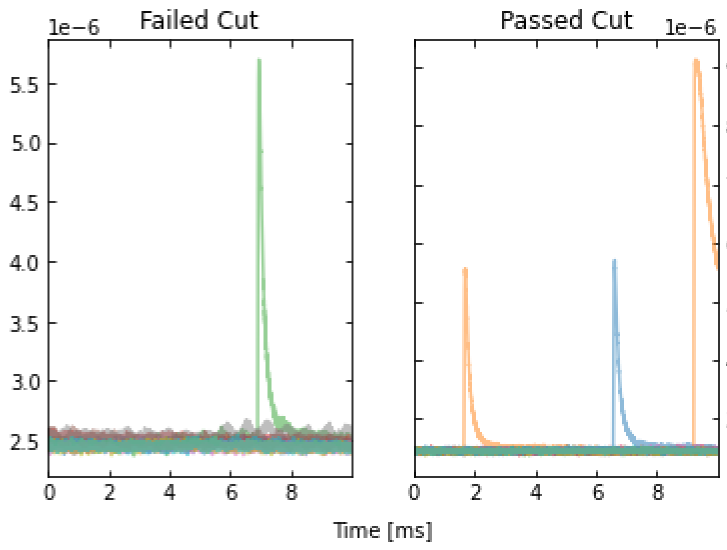
The PSD is essentially the same, but the pileup cut is no longer doing much, as we did it last (the previous three cuts ended cutting out a lot of pileup!). Thus, this shows that order does matter, and its worth thinking about what order makes the most sense in one's application.

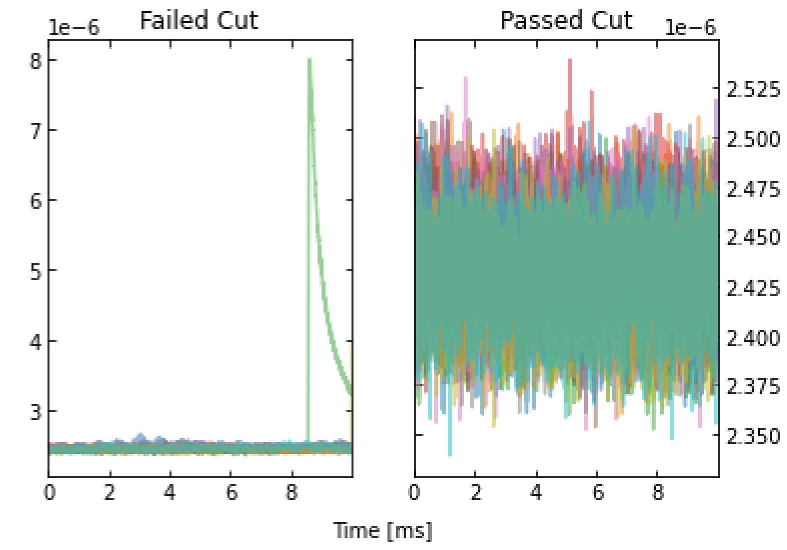
7.2.3 Advanced Usage: Arbitrary Cuts

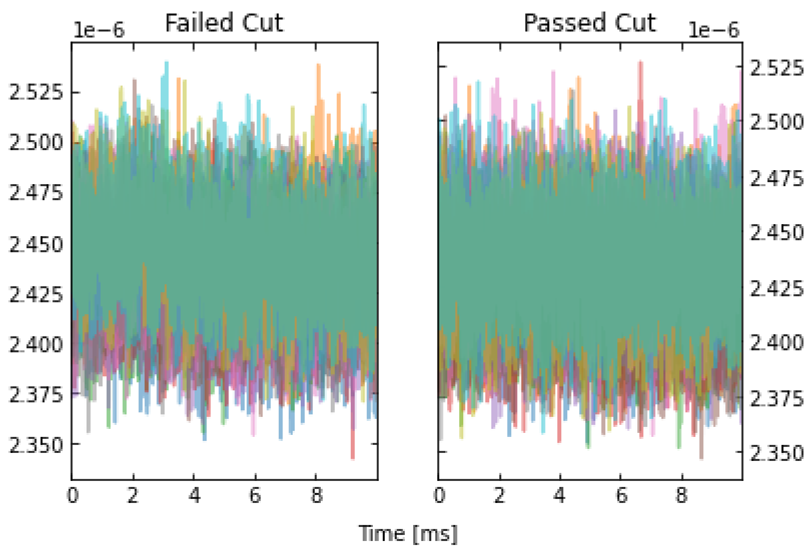
For advanced users, `IterCut` includes an option to apply some arbitrary cut based on some function that isn't included by default (or some one-off user-defined function). As an example, let's add a maximum cut via `numpy.max`, but only finding the maximum up to some specified bin number in the trace.

```
maximum = lambda traces, end_index: np.max(traces[..., :end_index], axis=-1)
```

```
IC = IterCut(traces, fs, plotall=True)
cmaximum = IC.arbitrarycut(maximum, 200, cut=2)
cpileup = IC.pileupcut(cut=2)
cslope = IC.slopecut(cut=2)
cut_ic = IC.cmask
```







```
print(f"The maximum cut efficiency is {np.sum(cmaximum)/len(traces):.3f}.")
print(f"The pileup cut efficiency is {np.sum(cpileup)/np.sum(cmaximum):.3f}.")
print(f"The slope cut efficiency is {np.sum(cslope)/np.sum(cpileup):.3f}.")
print("-------------")
print(f"The total cut efficiency is {np.sum(cut_ic)/len(traces):.3f}.")
```

```
The maximum cut efficiency is 0.721.
The pileup cut efficiency is 0.742.
The slope cut efficiency is 0.763.
-------------
The total cut efficiency is 0.408.
```

Looking at the events that passed, we see that the maximum cut we applied allowed a "bad" trace (a trace with a pulse). This makes sense since we only looked at a portion of the trace for that cut, so it was not a very good cut. Fortunately, our pileup and slope cuts did a good job of removing the bad traces that passed the maximum cut.

Lastly, it's worth simply printing out the docstrings of the three different supported outlier algorithms. The `kwargs` vary considerably between each of them, so to specify them in `IterCut`, one must know which ones to use! For example, `iterstat` has the `cut` `kwarg`, which we were using in the above examples (because `iterstat` is the default outlier algorithm for these automated cut routines).

```
from qetpy.cut import iterstat, removeoutliers
from astropy.stats import sigma_clip
```

```
?iterstat
```

```
Function to iteratively remove outliers based on how many standard
deviations they are from the mean, where the mean and standard
deviation are recalculated after each cut.

Parameters
----------
data : ndarray
    Array of data that we want to remove outliers from.
cut : float, optional
```

(continues on next page)

(continued from previous page)

```
    Number of standard deviations from the mean to be used for
    outlier rejection
precision : float, optional
    Threshold for change in mean or standard deviation such that we
    stop iterating. The threshold is determined by
    np.std(data)/precision. This means that a higher number for
    precision means a lower threshold (i.e. more iterations).
return_unbiased_estimates : bool, optional
    Boolean flag for whether or not to return the biased or
    unbiased estimates of the mean and standard deviation of the
    data. Default is False.

Returns
-------
datamean : float
    Mean of the data after outliers have been removed.
datastd : float
    Standard deviation of the data after outliers have been
    removed.
datamask : ndarray
    Boolean array indicating which values to keep or reject in
    data, same length as data.
```

```
?removeoutliers
```

```
Function to return indices of inlying points, removing points
by minimizing the skewness.

Parameters
----------
x : ndarray
    Array of real-valued variables from which to remove outliers.
maxiter : float, optional
    Maximum number of iterations to continue to minimize skewness.
    Default is 20.
skewtarget : float, optional
    Desired residual skewness of distribution. Default is 0.05.

Returns
-------
inds : ndarray
    Boolean indices indicating which values to select/reject, same
    length as x.
```

```
?sigma_clip
```

```
Perform sigma-clipping on the provided data.

The data will be iterated over, each time rejecting values that are
less or more than a specified number of standard deviations from a
center value.

Clipped (rejected) pixels are those where::
    data < center - (sigma_lower * std)
    data > center + (sigma_upper * std)

where::

    center = cenfunc(data [, axis=])
    std = stdfunc(data [, axis=])

Invalid data values (i.e., NaN or inf) are automatically clipped.

For an object-oriented interface to sigma clipping, see
SigmaClip.

.. note::
    scipy.stats.sigmaclip provides a subset of the functionality
    in this class. Also, its input data cannot be a masked array
    and it does not handle data that contains invalid values (i.e.,
    NaN or inf). Also note that it uses the mean as the centering
    function. The equivalent settings to scipy.stats.sigmaclip
    are::

        sigma_clip(sigma=4., cenfunc='mean', maxiters=None, axis=None,
        ...        masked=False, return_bounds=True)

Parameters
----------
data : array-like or ~numpy.ma.MaskedArray
    The data to be sigma clipped.

sigma : float, optional
    The number of standard deviations to use for both the lower
    and upper clipping limit. These limits are overridden by
    sigma_lower and sigma_upper, if input. The default is 3.

sigma_lower : float or None, optional
    The number of standard deviations to use as the lower bound for
    the clipping limit. If None then the value of sigma is
    used. The default is None.

sigma_upper : float or None, optional
    The number of standard deviations to use as the upper bound for
    the clipping limit. If None then the value of sigma is
    used. The default is None.

maxiters : int or None, optional
    The maximum number of sigma-clipping iterations to perform or
    None to clip until convergence is achieved (i.e., iterate
    until the last iteration clips nothing). If convergence is
    achieved prior to maxiters iterations, the clipping
    iterations will stop. The default is 5.

cenfunc : {'median', 'mean'} or callable, optional
    The statistic or callable function/object used to compute
    the center value for the clipping. If using a callable
    function/object and the axis keyword is used, then it must
    be able to ignore NaNs (e.g., numpy.nanmean) and it must have
    an axis keyword to return an array with axis dimension(s)
    removed. The default is 'median'.

stdfunc : {'std', 'mad_std'} or callable, optional
    The statistic or callable function/object used to compute the
    standard deviation about the center value. If using a callable
    function/object and the axis keyword is used, then it must
    be able to ignore NaNs (e.g., numpy.nanstd) and it must have
    an axis keyword to return an array with axis dimension(s)
    removed. The default is 'std'.

axis : None or int or tuple of int, optional
    The axis or axes along which to sigma clip the data. If None,
    then the flattened data will be used. axis is passed to the
    cenfunc and stdfunc. The default is None.

masked : bool, optional
    If True, then a ~numpy.ma.MaskedArray is returned, where
    the mask is True for clipped values. If False, then a
    ~numpy.ndarray and the minimum and maximum clipping thresholds
    are returned. The default is True.

return_bounds : bool, optional
    If True, then the minimum and maximum clipping bounds are also
    returned.

copy : bool, optional
    If True, then the data array will be copied. If False
    and masked=True, then the returned masked array data will
    contain the same array as the input data (if data is a
    ~numpy.ndarray or ~numpy.ma.MaskedArray). If False and
    masked=False, the input data is modified in-place. The
    default is True.

grow : float or False, optional
    Radius within which to mask the neighbouring pixels of those
    that fall outwith the clipping limits (only applied along
    axis, if specified). As an example, for a 2D image a value
    of 1 will mask the nearest pixels in a cross pattern around each
    deviant pixel, while 1.5 will also reject the nearest diagonal
    neighbours and so on.

Returns
-------
result : array-like
    If masked=True, then a ~numpy.ma.MaskedArray is returned,
    where the mask is True for clipped values and where the input
    mask was True.

    If masked=False, then a ~numpy.ndarray is returned.
    If return_bounds=True, then in addition to the masked array
    or array above, the minimum and maximum clipping bounds are
    returned.

    If masked=False and axis=None, then the output array
    is a flattened 1D ~numpy.ndarray where the clipped values
    have been removed. If return_bounds=True then the returned
    minimum and maximum thresholds are scalars.

    If masked=False and axis is specified, then the output
    ~numpy.ndarray will have the same shape as the input data
    and contain np.nan where values were clipped. If the input
    data was a masked array, then the output ~numpy.ndarray
    will also contain np.nan where the input mask was True.
    If return_bounds=True then the returned minimum and maximum
    clipping thresholds will be be ~numpy.ndarrays.

See Also
--------
SigmaClip, sigma_clipped_stats

Notes
-----
The best performance will typically be obtained by setting
cenfunc and stdfunc to one of the built-in functions
specified as as string. If one of the options is set to a string
while the other has a custom callable, you may in some cases see
better performance if you have the `bottleneck`_ package installed.

.. _bottleneck:  https://github.com/pydata/bottleneck

Examples
--------
This example uses a data array of random variates from a Gaussian
distribution. We clip all points that are more than 2 sample
standard deviations from the median. The result is a masked array,
where the mask is True for clipped data::

    >>> from astropy.stats import sigma_clip
    >>> from numpy.random import randn
    >>> randvar = randn(10000)
    >>> filtered_data = sigma_clip(randvar, sigma=2, maxiters=5)

This example clips all points that are more than 3 sigma relative
to the sample mean, clips until convergence, returns an unmasked
~numpy.ndarray, and does not copy the data::

    >>> from astropy.stats import sigma_clip
    >>> from numpy.random import randn
    >>> from numpy import mean
    >>> randvar = randn(10000)
    >>> filtered_data = sigma_clip(randvar, sigma=3, maxiters=None,
    ...                            cenfunc=mean, masked=False, copy=False)
```

```
This example sigma clips along one axis::

    >>> from astropy.stats import sigma_clip
    >>> from numpy.random import normal
    >>> from numpy import arange, diag, ones
    >>> data = arange(5) + normal(0., 0.05, (5, 5)) + diag(ones(5))
    >>> filtered_data = sigma_clip(data, sigma=2.3, axis=0)

Note that along the other axis, no points would be clipped, as the
standard deviation is higher.
```

Table of Contents

7.3 Example Code for using the IV class

Import the need packages to run the test script

```
import numpy as np
from qetpy import IV
```

Let's load the example data provided, which is from a dataset taken at SLAC for multiple devices and multiple bath temperatures.

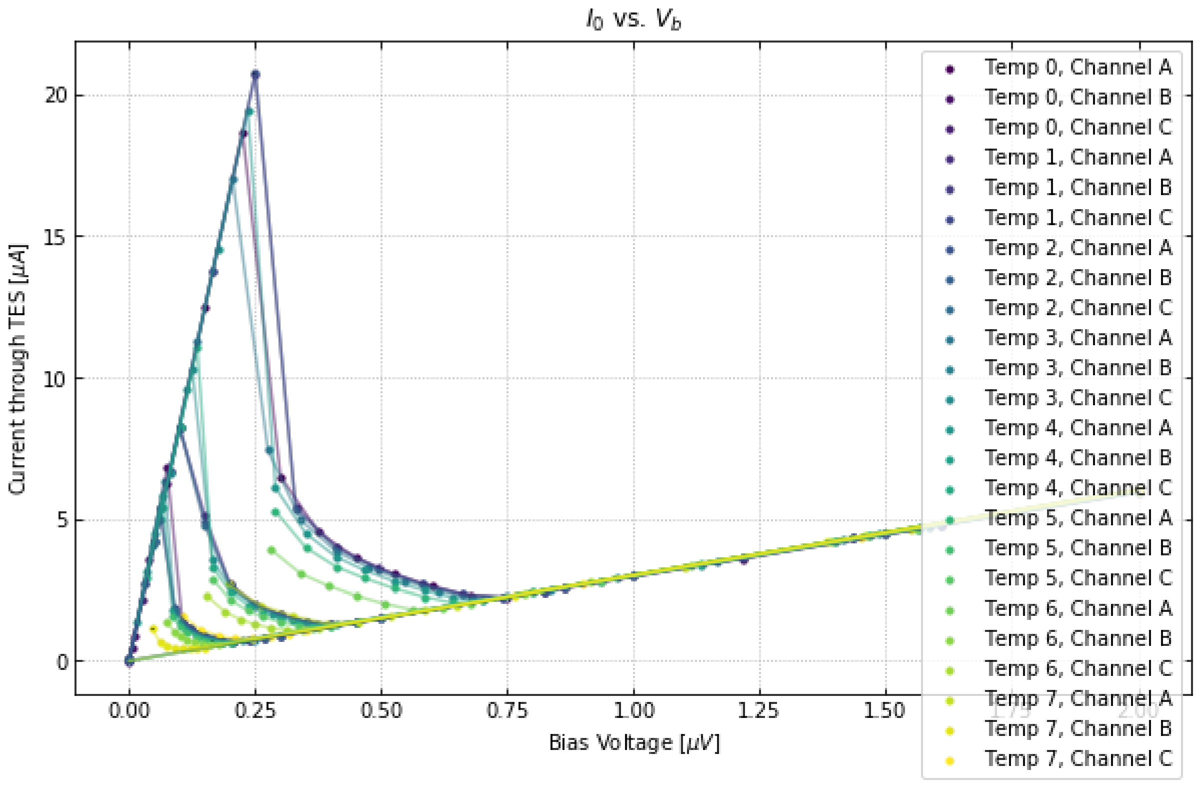
```
testdata = np.load("test_iv_data.npz")
dites = testdata["dites"]
dites_err = testdata["dites_err"]
vb = testdata["vb"]
vb_err = testdata["vb_err"]
rload = testdata["rload"]
rload_err = testdata["rload_err"]
```

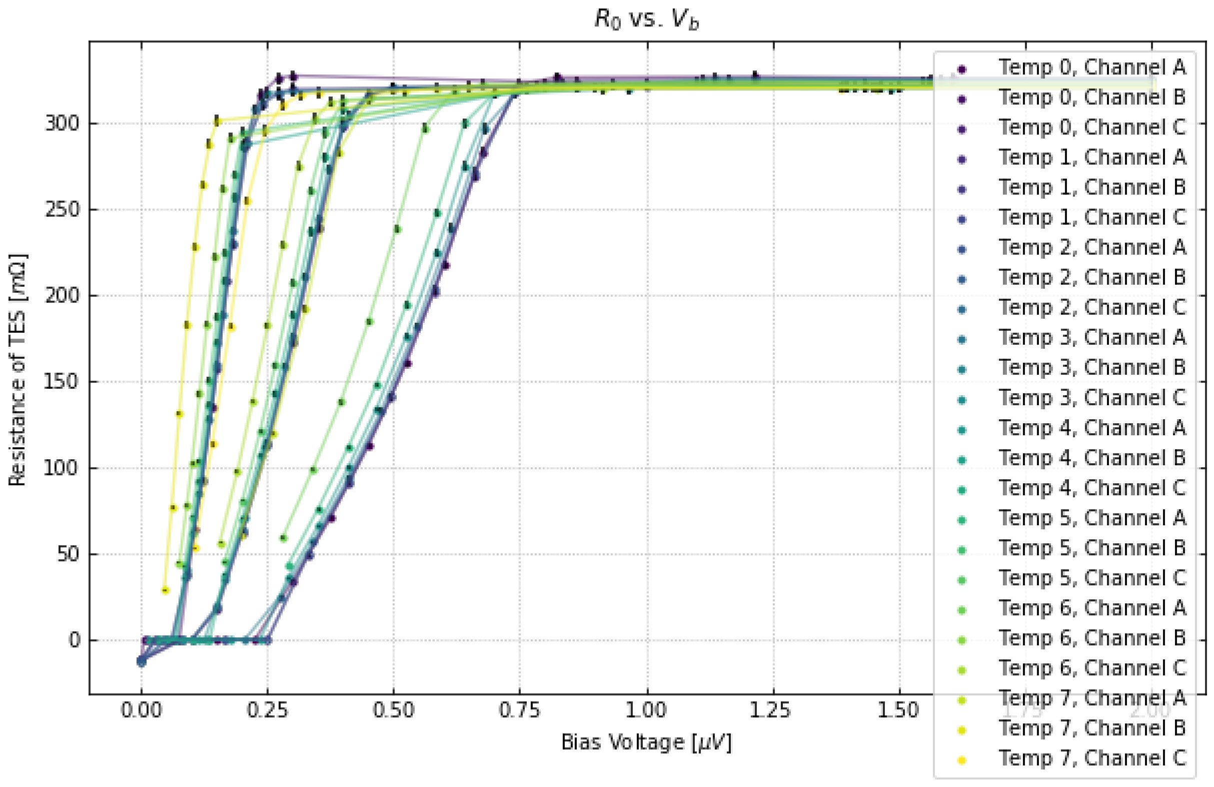
Now let's use the IV class and calculate the IV curves

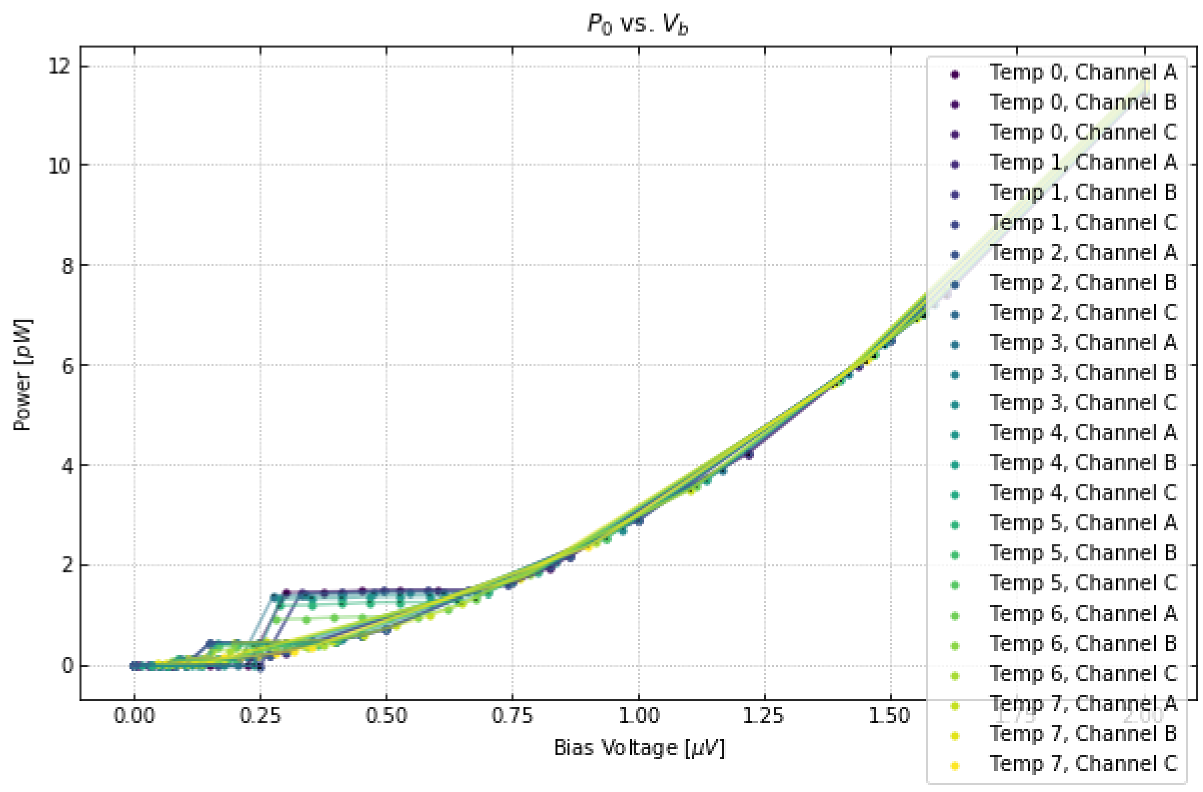
```
ivdata = IV(dites, dites_err, vb, vb_err, rload, rload_err, ["A","B","C"])
ivdata.calc_iv()
```

Let's take a look at the plotting. We can plot all of the curves together.

```
ivdata.plot_all_curves()
```

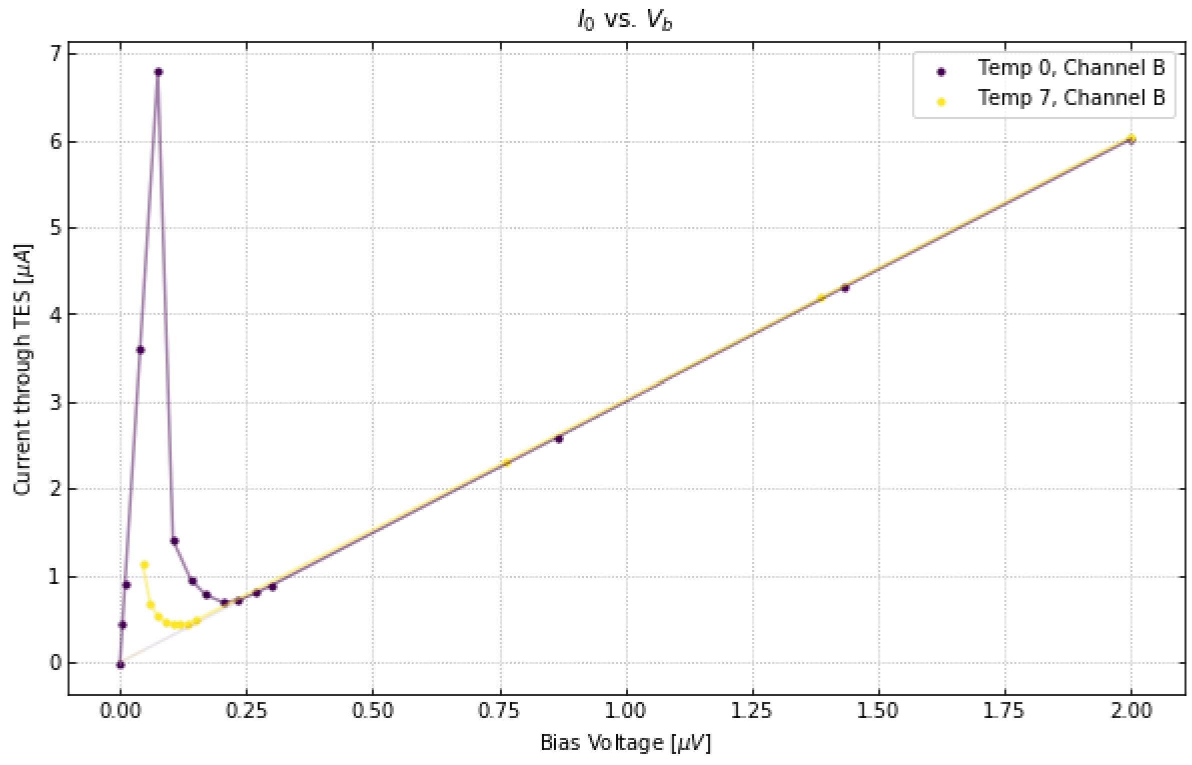

I_0 vs. V_b
Current through TES [μA]
Bias Voltage [μV]
Temp 0, Channel A
Temp 0, Channel B
Temp 0, Channel C
Temp 1, Channel A
Temp 1, Channel B
Temp 1, Channel C
Temp 2, Channel A
Temp 2, Channel B
Temp 2, Channel C
Temp 3, Channel A
Temp 3, Channel B
Temp 3, Channel C
Temp 4, Channel A
Temp 4, Channel B
Temp 4, Channel C
Temp 5, Channel A
Temp 5, Channel B
Temp 5, Channel C
Temp 6, Channel A
Temp 6, Channel B
Temp 6, Channel C
Temp 7, Channel A
Temp 7, Channel B
Temp 7, Channel C

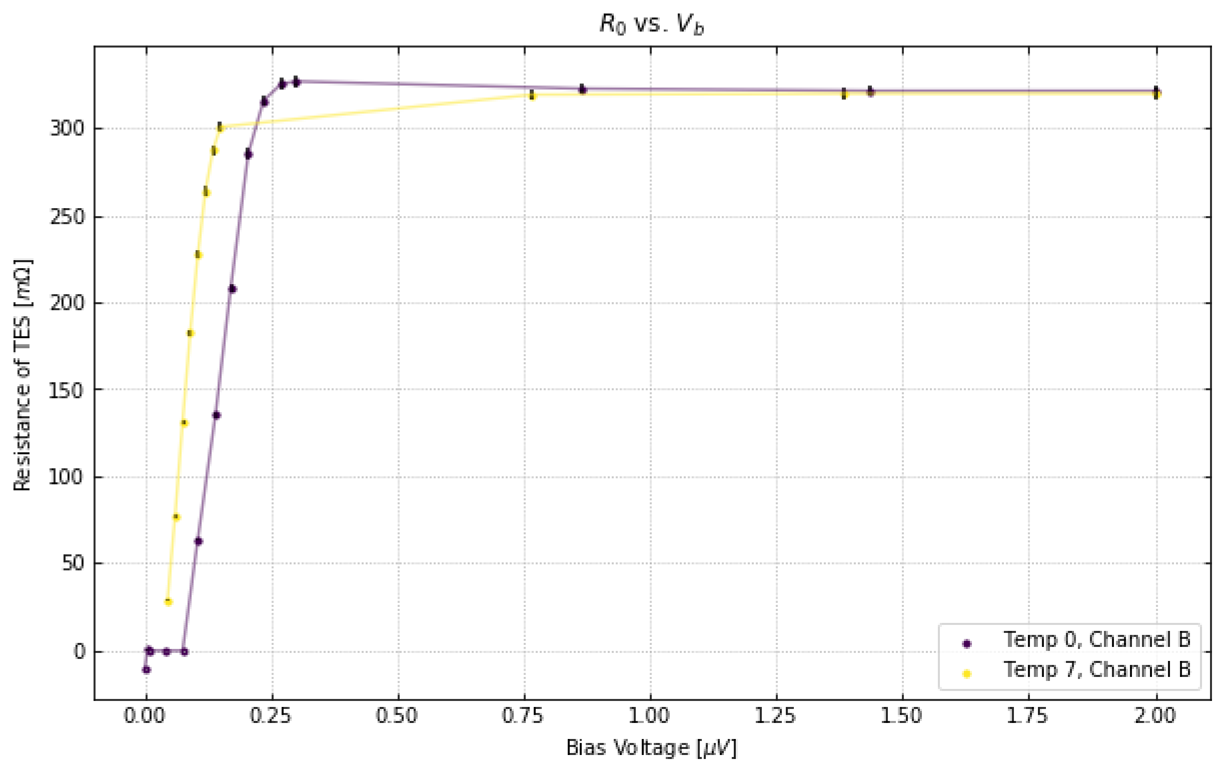

R_0 vs. V_b
Resistance of TES [$m\Omega$]
Bias Voltage [μV]
Temp 0, Channel A
Temp 0, Channel B
Temp 0, Channel C
Temp 1, Channel A
Temp 1, Channel B
Temp 1, Channel C
Temp 2, Channel A
Temp 2, Channel B
Temp 2, Channel C
Temp 3, Channel A
Temp 3, Channel B
Temp 3, Channel C
Temp 4, Channel A
Temp 4, Channel B
Temp 4, Channel C
Temp 5, Channel A
Temp 5, Channel B
Temp 5, Channel C
Temp 6, Channel A
Temp 6, Channel B
Temp 6, Channel C
Temp 7, Channel A
Temp 7, Channel B
Temp 7, Channel C

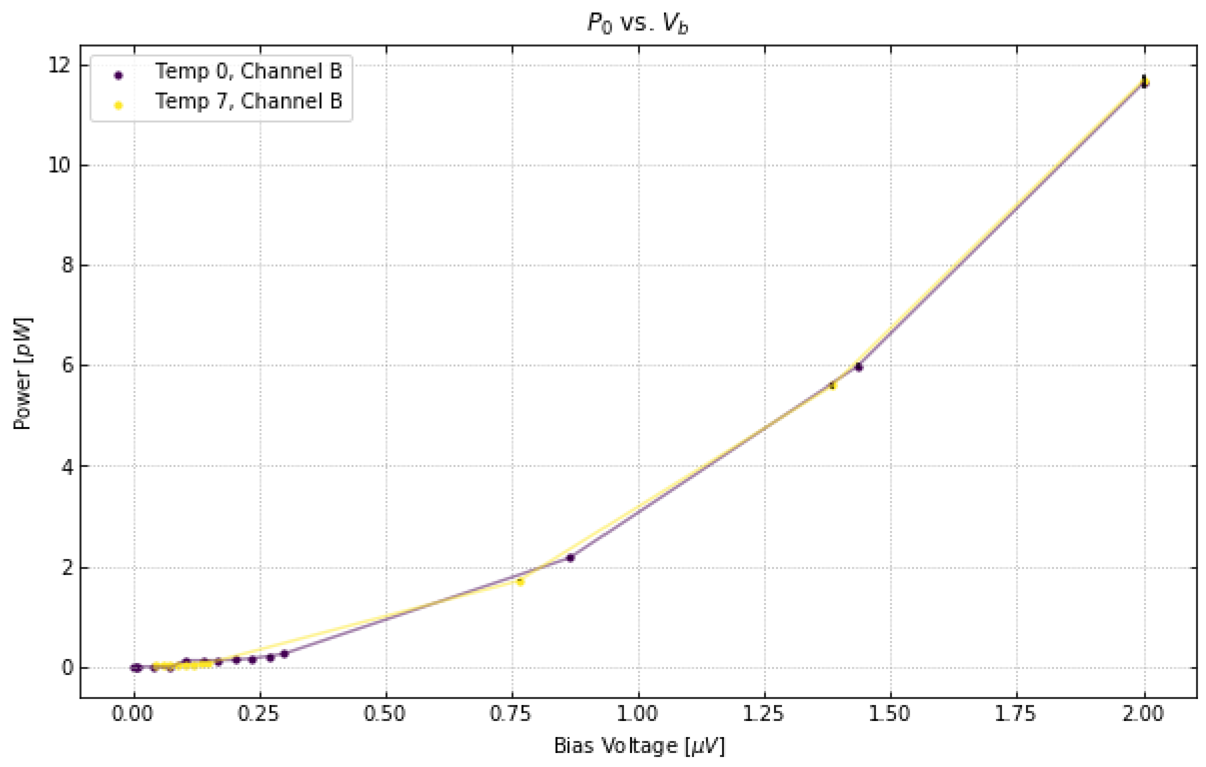


If you want to plot certain channels or certain bath temperatures, use the chans and temps flags.

```
ivdata.plot_all_curves(temps=[0,7], chans=1)
```

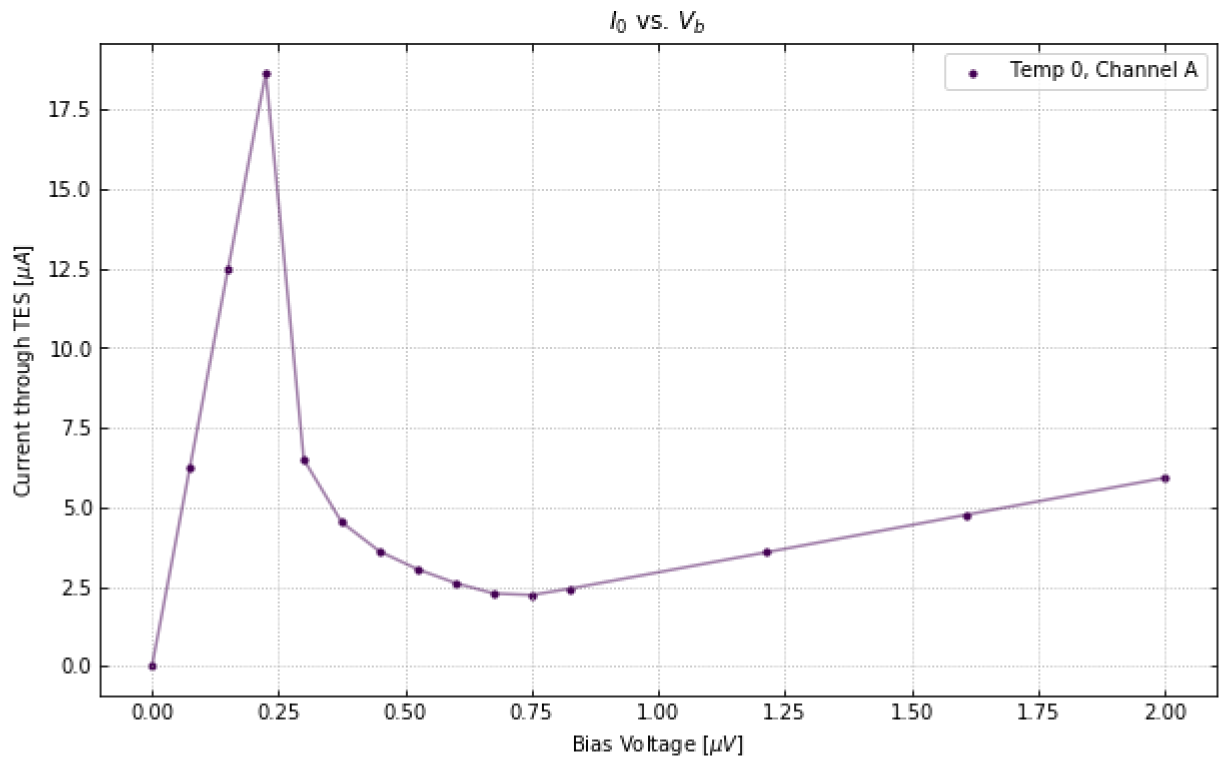


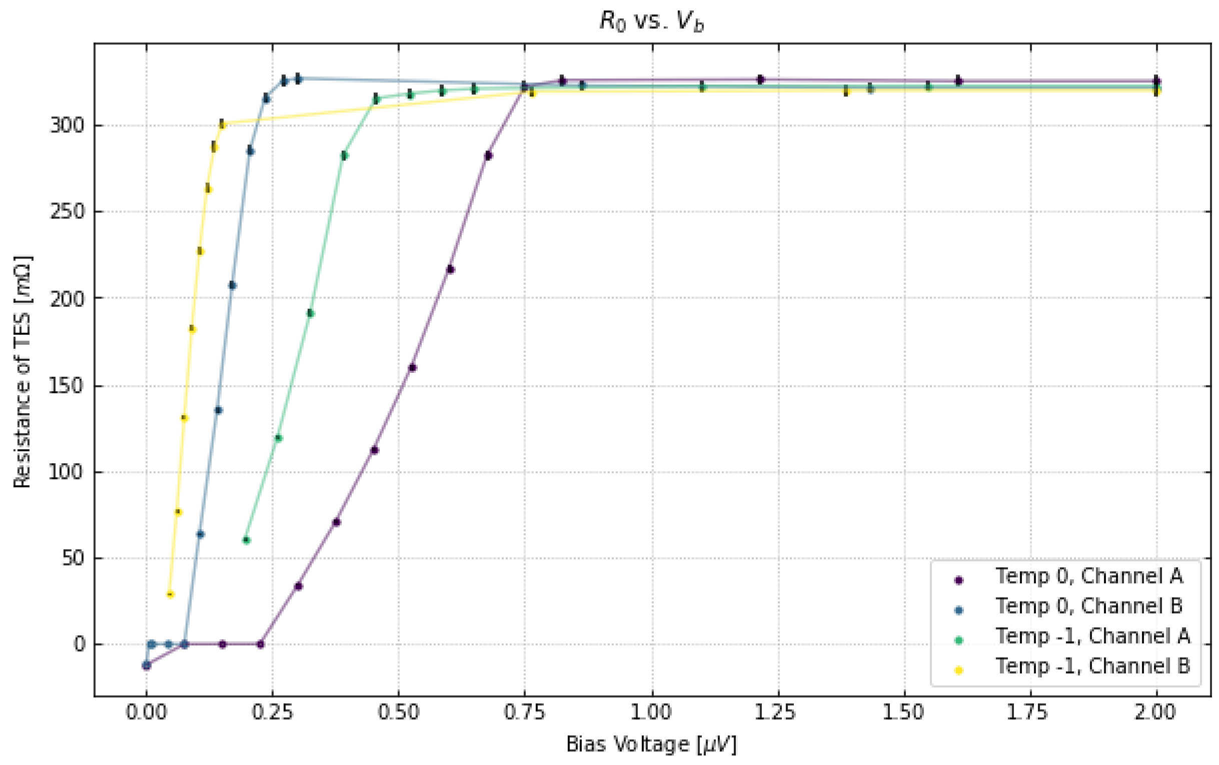




We can also plot the IV, resistance, and power curves separately. See the documentation for more information on the plotting options.

```
ivdata.plot_iv(temps=0, chans=0, showfit=False);
ivdata.plot_rv(temps=[0,-1], chans=[0,1]);
ivdata.plot_pv(temps="all", chans=2);
```





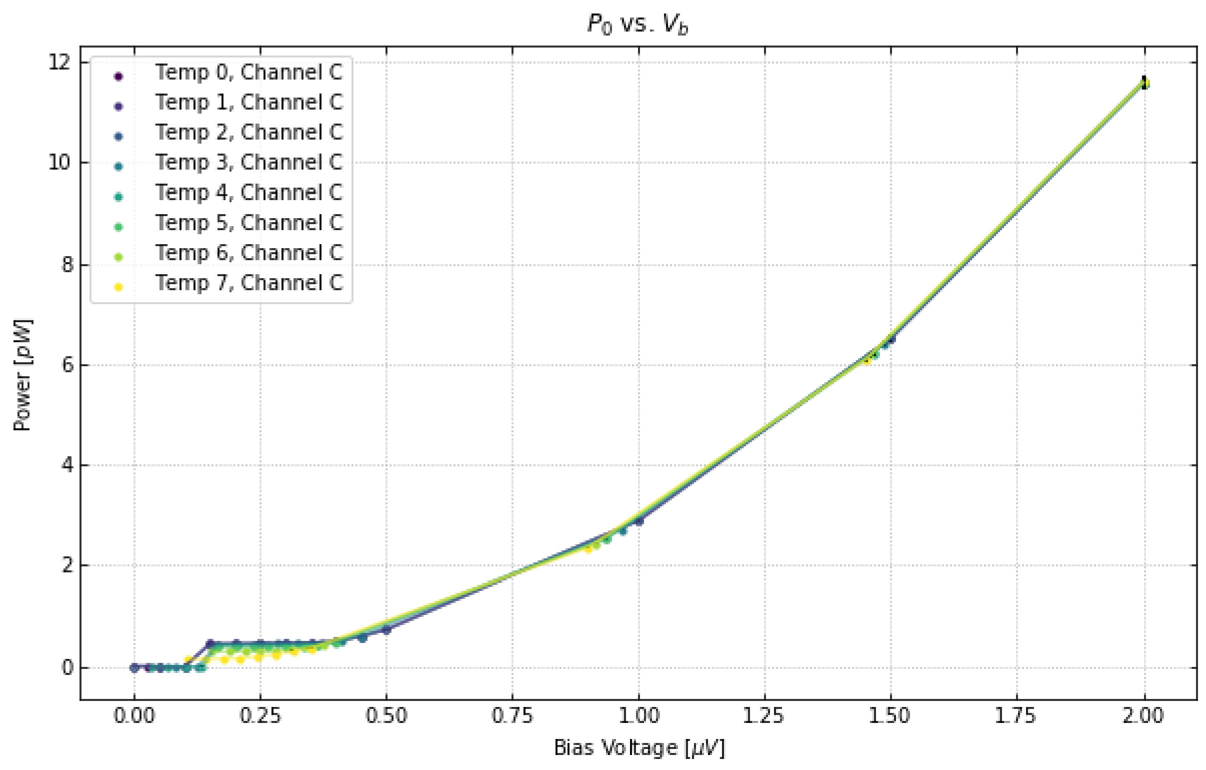


Table of Contents

7.4 Noise Analysis Test

Imports

```
from qetpy import Noise
from qetpy.sim import TESnoise
from qetpy.plotting import compare_noise, plot_noise_sim
import numpy as np
import matplotlib.pyplot as plt
```

Load test data

```
pathToTraces = ''
traces_PT_on = np.load(pathToTraces+'traces.npy')
```

Create noise object

```
Noise?
```

```
#savePath = 'example_Figs/' #used for test, user should define new path for test so they␣
↪don't save over these figs
savePath = '' #user needs to define new path

fs = 625e3 #define sample rate
channels = [ 'PCS1' , 'PES1' , 'PFS1' , 'PAS2' , 'PBS2' , 'PES2' , 'PDS2' ] #define the␣
↪channel names
```

(continues on next page)

(continued from previous page)

```
g124_noise = Noise(traces=traces_PT_on,
                   fs=fs,
                   channames=channels,
                   name= 'G124 SLAC Run 37 Pulse Tube On') #initialize a noise object
```

Calculate the PSD and corrCoeff

```
g124_noise.calculate_psd()
g124_noise.calculate_corrcoeff()
g124_noise.calculate_csd()
```

Calculate unCorrelated noise

```
g124_noise.calculate_uncorr_noise()
```

Test saving.

Uncomment to save and re-load

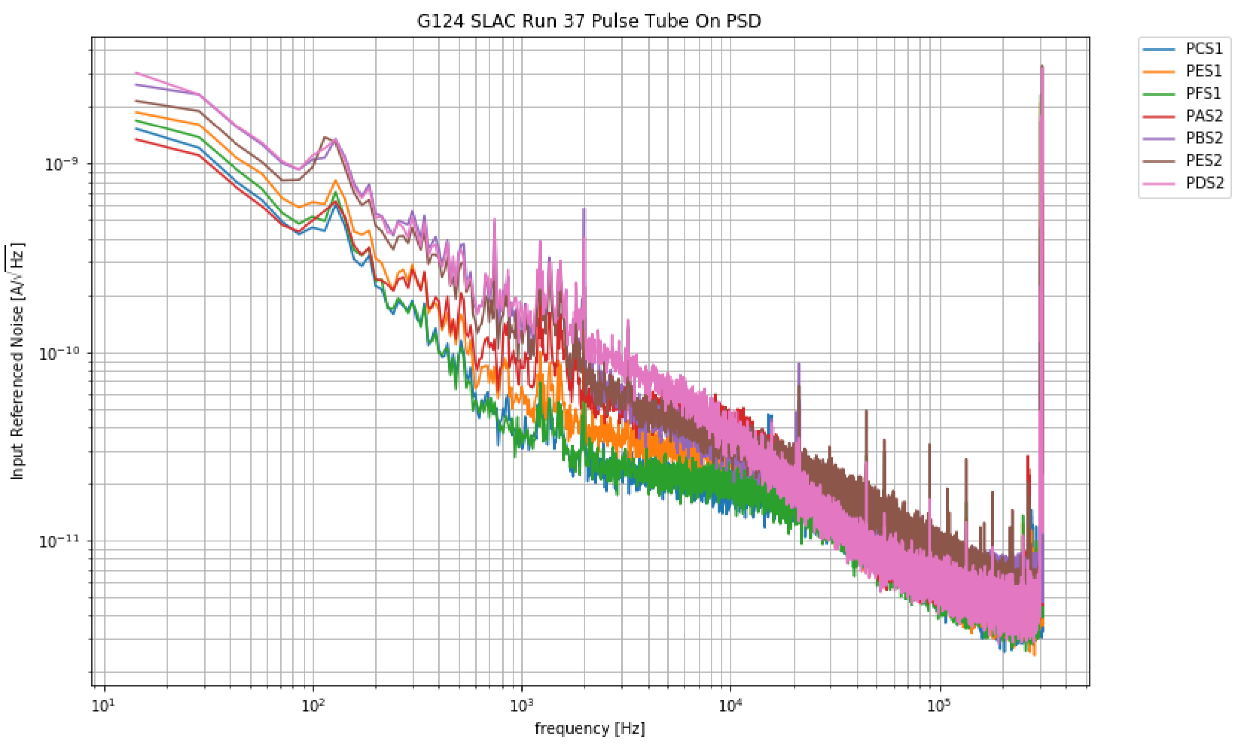
```
#saveName = pathToTraces + g124_noise.name.replace(" ", "_") + '.pkl'
#g124_noise.save(pathToTraces)
```

```
#del g124_noise
```

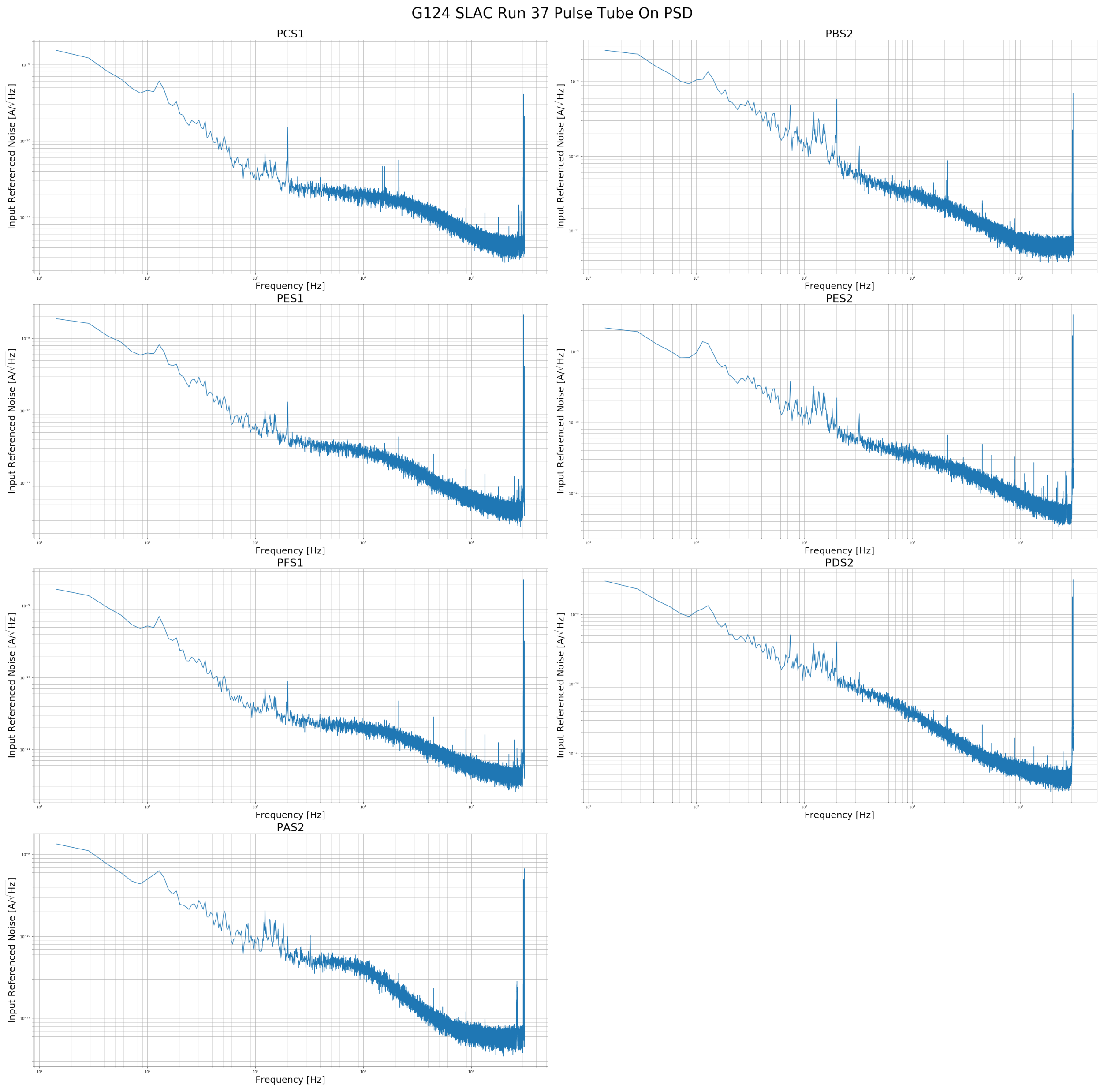
```
# with open(pathToTraces,'rb') as savefile:
#     g124_noise = pickle.load(savefile)
```

Test plotting of PSD and corrCoeff

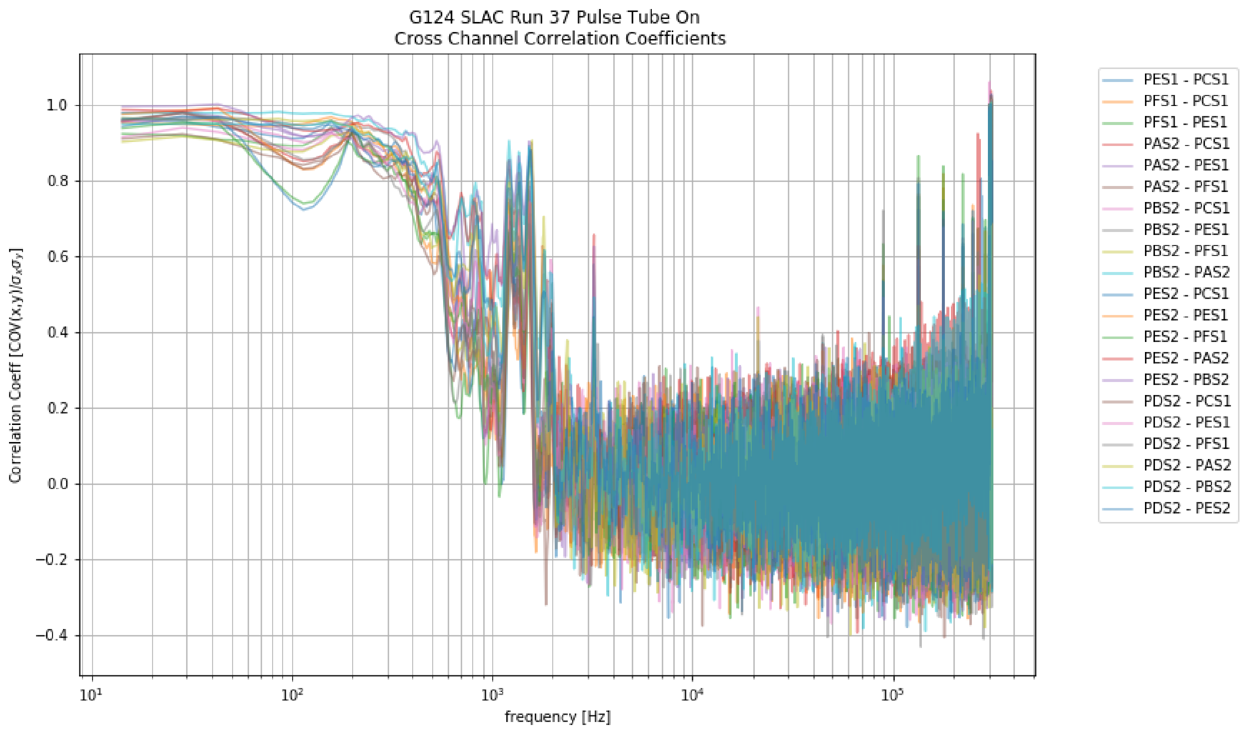
```
g124_noise.plot_psd(lgcoverlay=True)
```



```
g124_noise.plot_psd(lgcoverlay=False)
```

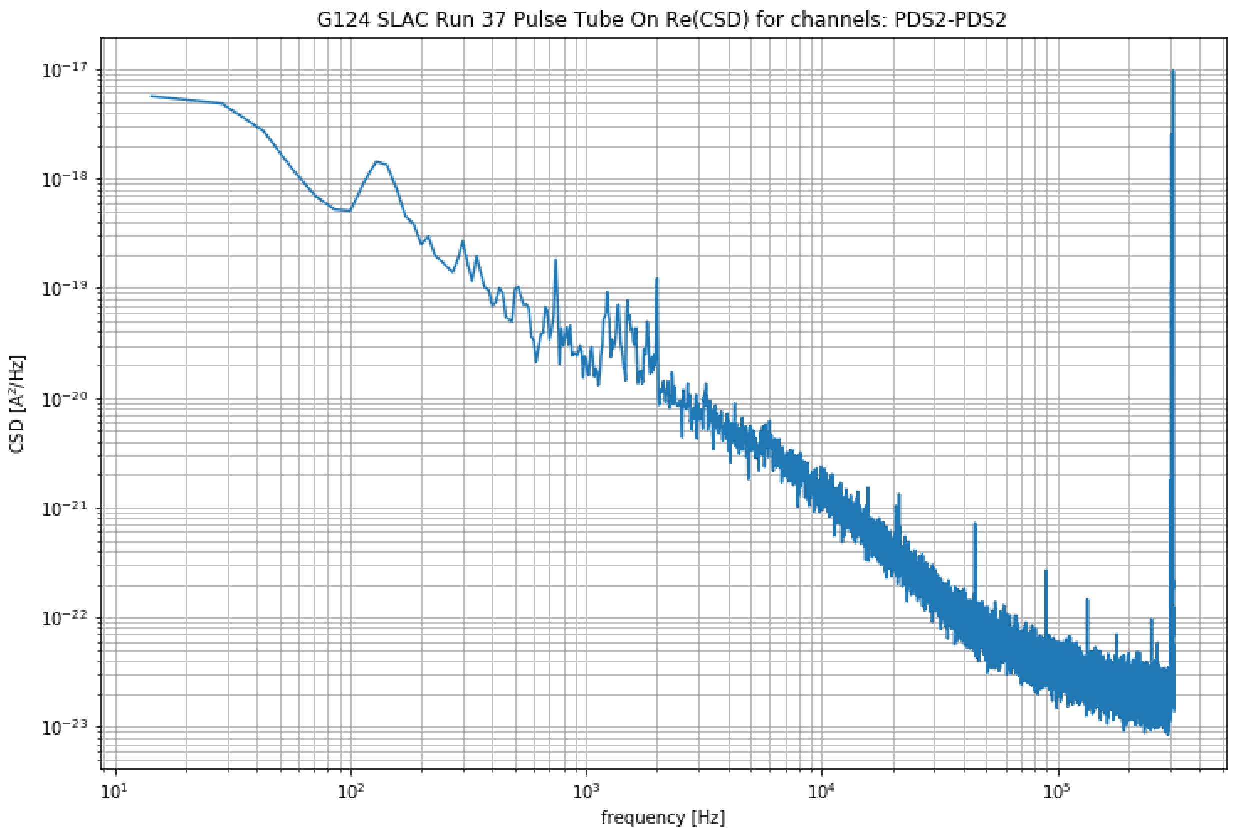


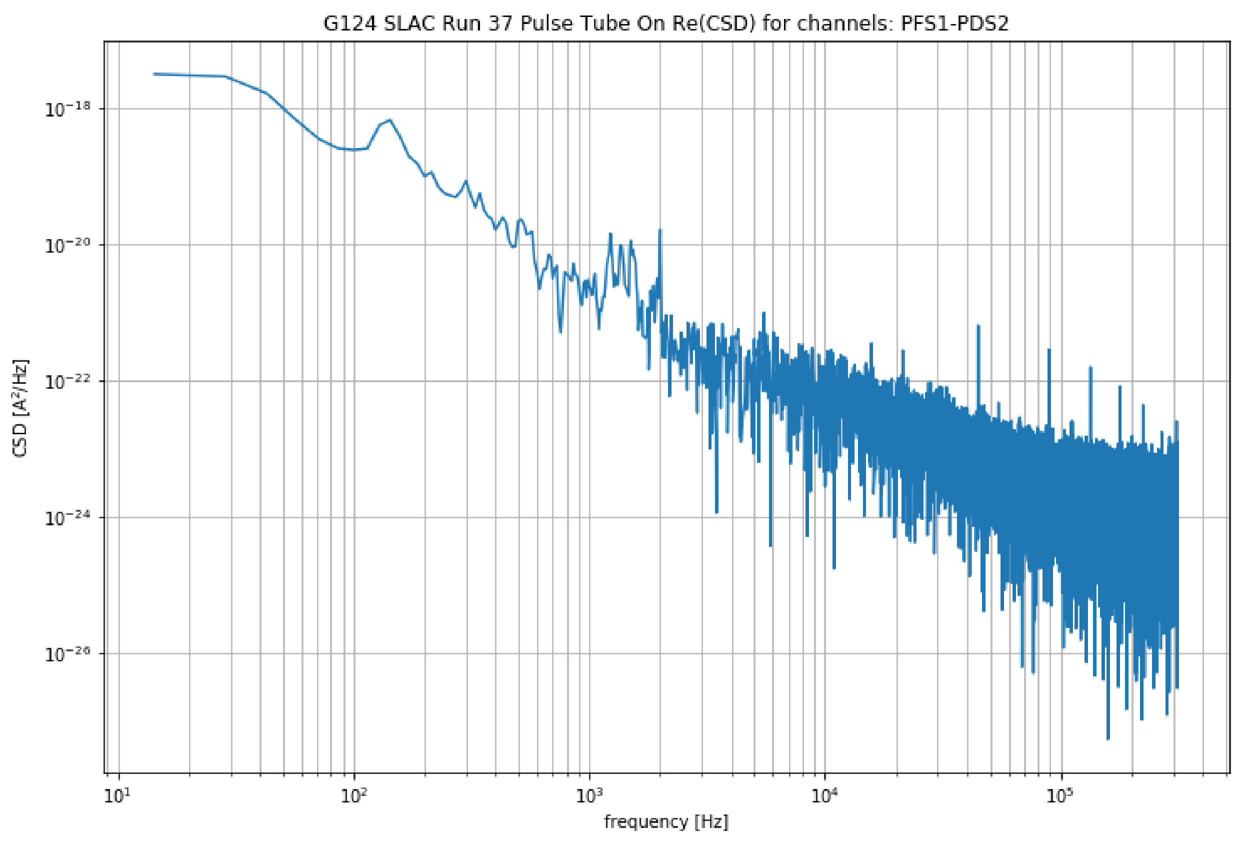
```
g124_noise.plot_corrcoeff(lgcsave=False, lgcsmooth=True, nwindow=13 )
```



Plot a few CSDs

```
g124_noise.plot_csd(whichcsd=['66','26'])
```





Try to plot a CSD for a non existant channel

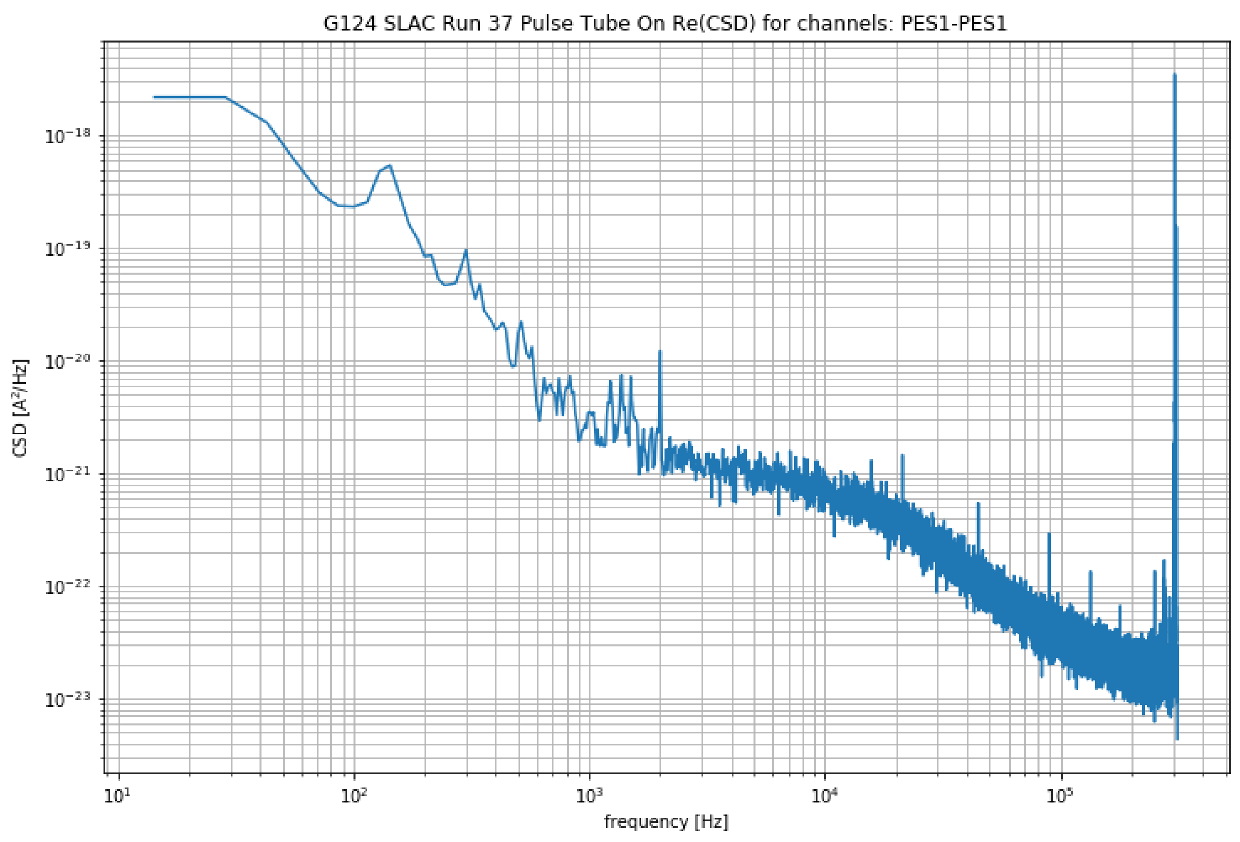
```
g124_noise.plot_csd(whichcsd=['68'])
```

```
index out of range
```

Try to save a figure with a bad path

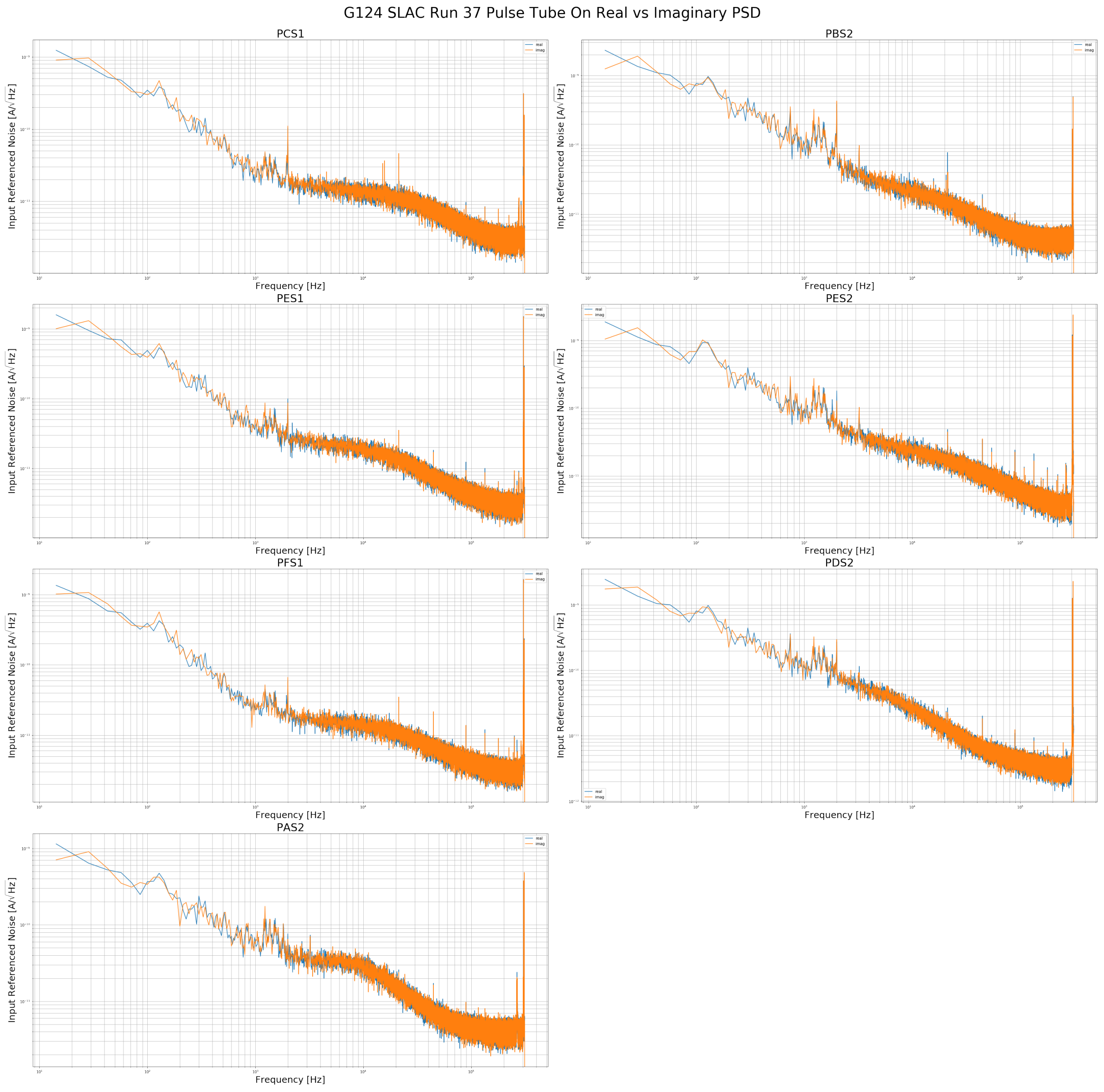
```
g124_noise.plot_csd(whichcsd=['11'], lgcsave=True, savepath = 'yay/python/is/great')
```

```
Invalid save path. Figure not saved
```



Plot Real vs Imaginary PSDs

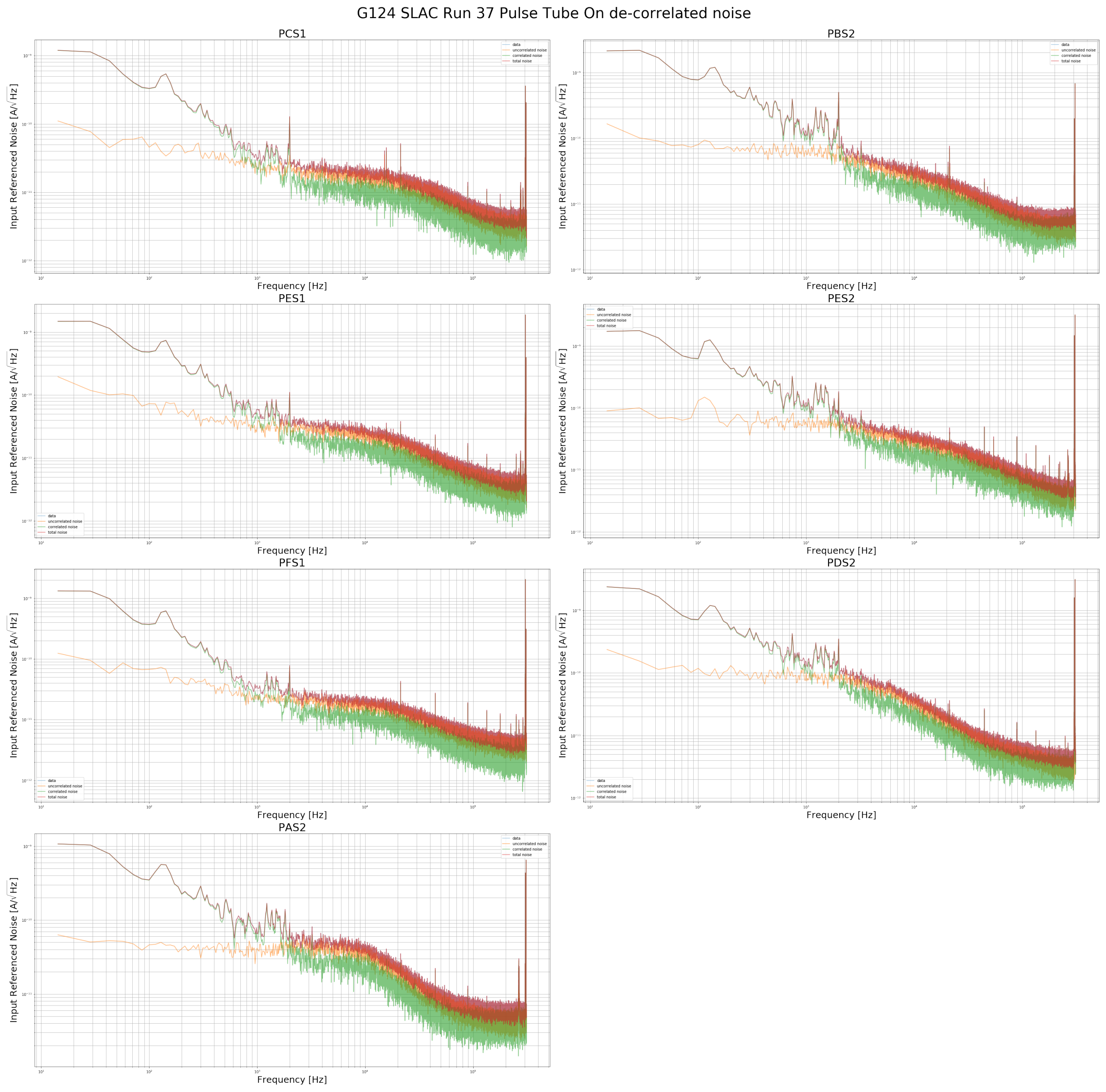
```
g124_noise.plot_reim_psd()
```



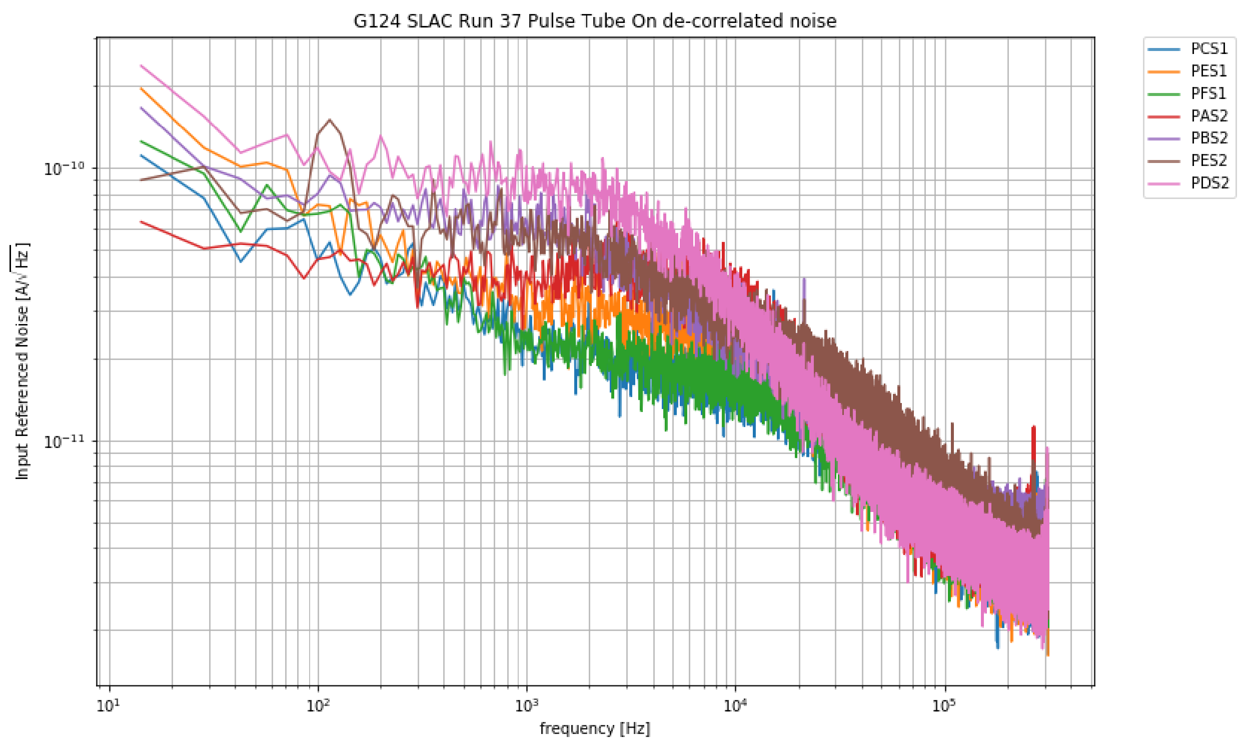
Plot unCorrelated part of the noise PSD

```
g124_noise.calculate_uncorr_noise()
```

```
g124_noise.plot_decorrelatednoise(lgccorrelated=True,lgcsum = True, lgcsave=False)
```



```
g124_noise.plot_decorrelatednoise(lgcoverlay = True)
```

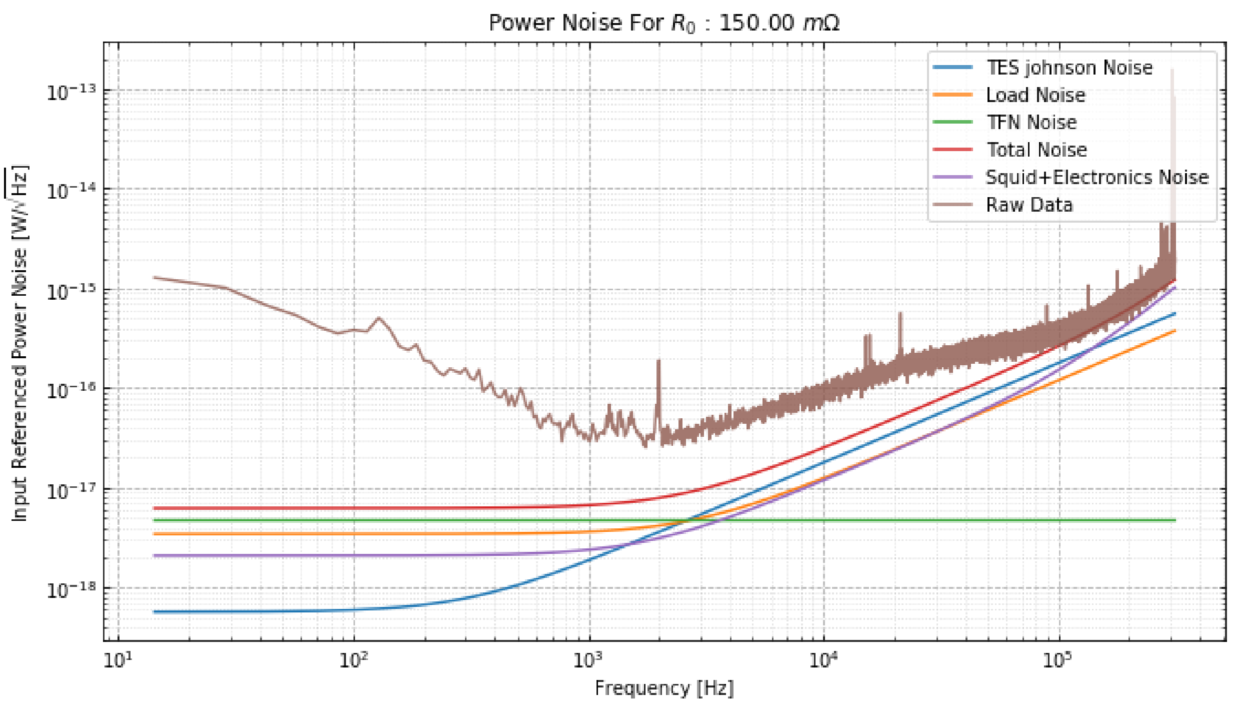


Create noise simulation object

```
noise_sim = TESnoise(freqs = g124_noise.freqs[1:])
```

Note, these default noise parameters are completely made up, just for demostration

```
plot_noise_sim(g124_noise.freqs, g124_noise.psd[0,:], noise_sim, istype='power',␣
↪qetbias=0)
```



7.5 # Example Code for the Optimum Filters

Import `qetpy` and other necessary packages.

```
import numpy as np
import matplotlib.pyplot as plt
import qetpy as qp
from pprint import pprint
```

7.6 Use `QETpy` to generate some simulated TES noise

We can use `qetpy.sim.TESnoise` to help create a simulated PSD with characteristic TES parameters.

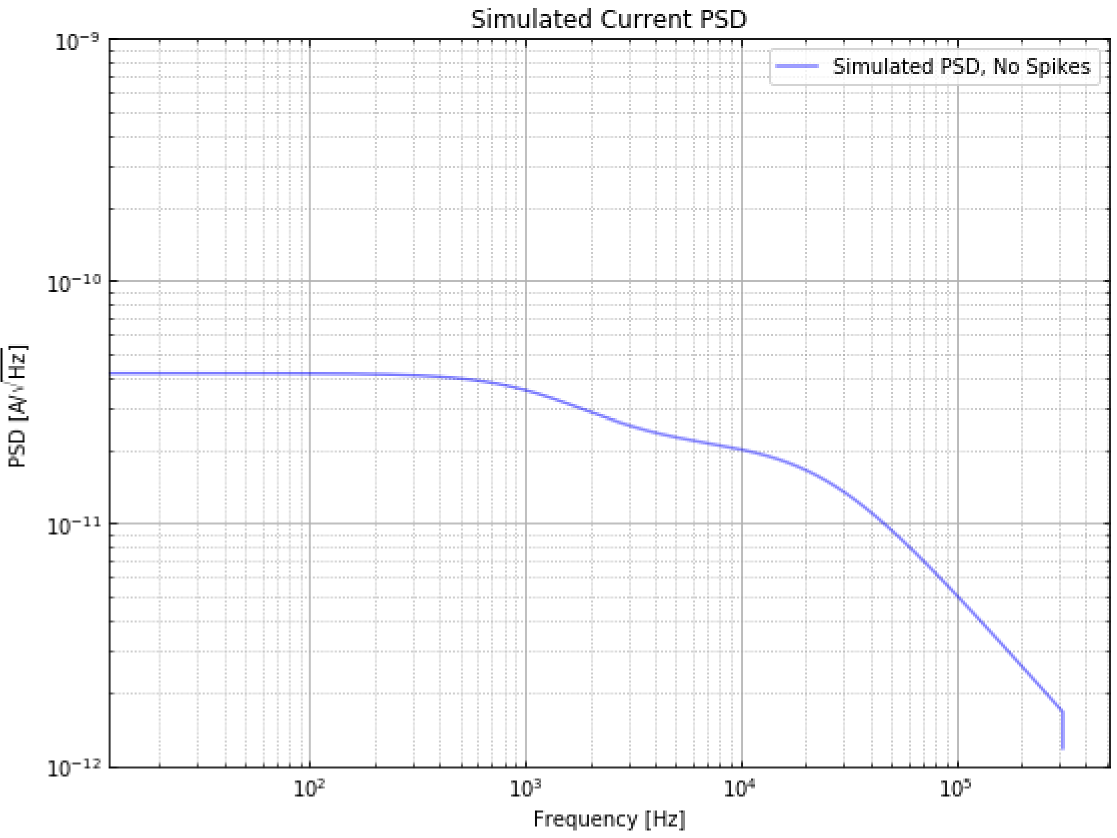
```
fs = 625e3
f = np.fft.fftfreq(32768, d=1/fs)
noisesim = qp.sim.TESnoise(r0=0.03)
psd_sim = noisesim.s_iload(freqs=f) + noisesim.s_ites(freqs=f) + noisesim.s_itfn(freqs=f)

f_fold, psd_sim_fold = qp.foldpsd(psd_sim, fs=fs)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.loglog(f_fold, psd_sim_fold**0.5, color="blue", alpha=0.5, label="Simulated PSD, No␣
↪Spikes")
ax.set_ylim(1e-12,1e-9)
ax.grid()
ax.grid(which="minor", linestyle="dotted")
```

(continues on next page)

(continued from previous page)

```
ax.tick_params(which="both", direction="in", right=True, top=True)
ax.legend(loc="best")
ax.set_title("Simulated Current PSD")
ax.set_ylabel("PSD [A/$\sqrt{\mathrm{Hz}}$]")
ax.set_xlabel("Frequency [Hz]")
fig.tight_layout()
```



With a PSD, we can use `qetpy.gen_noise` to generate random noise from the PSD (assuming the frequencies are uncorrelated). Then, we will create an example pulse.

```
# create a template
pulse_amp = 1e-6 # [A]
tau_f = 66e-6 # [s]
tau_r = 20e-6 # [s]

t = np.arange(len(psd_sim))/fs

pulse = np.exp(-t/tau_f)-np.exp(-t/tau_r)
pulse_shifted = np.roll(pulse, len(t)//2)
template = pulse_shifted/pulse_shifted.max()

# use the PSD to create an example trace to fit
noise = qp.gen_noise(psd_sim, fs=fs, ntraces=1)[0]
signal = noise + np.roll(template, 100)*pulse_amp # note the shift we have added, 160 us
```

7.7 Fit a single pulse with `OptimumFilter`

With a pulse created, let's use the `qetpy.OptimumFilter` class to run different Optimum Filters.

```
qp.OptimumFilter?
```

Below, we print the different methods available in `qetpy.OptimumFilter`. In this notebook, we will demo the `ofamp_nodelay`, `ofamp_withdelay`, `ofamp_pileup_iterative`, and `update_signal` methods. It is highly recommend to read the documentation for the other methods, as there are many useful ones!

```
method_list = sorted([func for func in dir(qp.OptimumFilter) if callable(getattr(qp.
↪OptimumFilter, func)) and not func.startswith("__")])
pprint(method_list)
```

```
['chi2_lowfreq',
 'chi2_nopulse',
 'energy_resolution',
 'ofamp_baseline',
 'ofamp_nodelay',
 'ofamp_pileup_iterative',
 'ofamp_pileup_stationary',
 'ofamp_withdelay',
 'time_resolution',
 'update_signal']
```

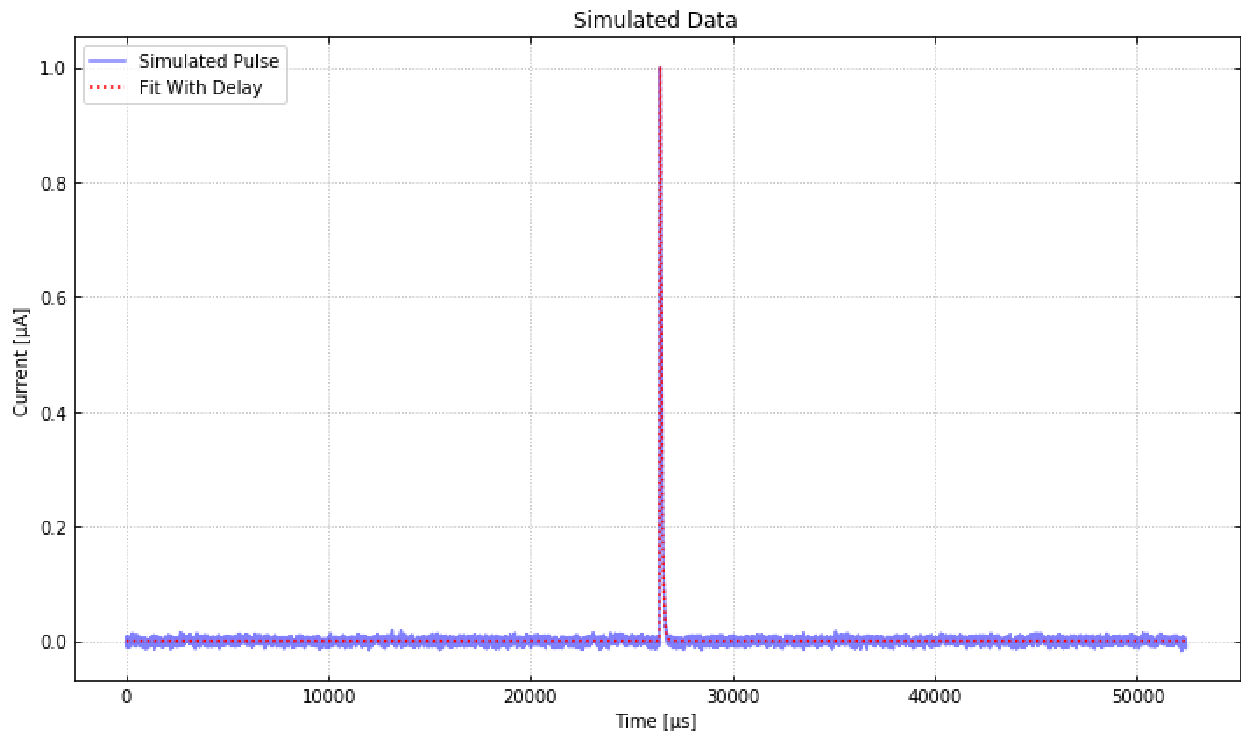
Let's run the Optimum Filter without and with a time-shifting degree of freedom.

```
OF = qp.OptimumFilter(signal, template, psd_sim, fs) # initialize the OptimumFilter class
amp_nodelay, chi2_nodelay = OF.ofamp_nodelay()
amp_withdelay, t0_withdelay, chi2_withdelay = OF.ofamp_withdelay()

print(f"No Delay Fit: amp = {amp_nodelay*1e6:.2f} A, ^2 = {chi2_nodelay:.2f}")
print(f"With Delay Fit: amp = {amp_withdelay*1e6:.2f} A, t_0 = {t0_withdelay*1e6} s, ^2␣
↪= {chi2_withdelay:.2f}")
```

```
No Delay Fit: amp = -0.04 A, ^2 = 210399.75
With Delay Fit: amp = 1.00 A, t_0 = 160.0 s, ^2 = 32407.30
```

Since we have added a 160 us shift, we see that the "with delay" optimum filter fit the time-shift perfectly, and the chi-squared is very close to the number of degrees of freedom (32768), as we would expect for a good fit.

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(t*1e6, signal*1e6, label="Simulated Pulse", color="blue", alpha=0.5)
ax.plot(t*1e6, amp_withdelay*np.roll(template, int(t0_withdelay*fs))*1e6,
        label="Fit With Delay", color="red", linestyle="dotted")
ax.set_ylabel("Current [A]")
ax.set_xlabel("Time [s]")
ax.set_title("Simulated Data")
lgd = ax.legend(loc="upper left")
ax.tick_params(which="both", direction="in", right=True, top=True)
ax.grid(linestyle="dotted")
fig.tight_layout()
```



7.8 Add a pileup pulse and fit with `OptimumFilter`

Let's now add a second (pileup) pulse in order to see how we can use `ofamp_pileup_iterative`.

```
pileup = signal + np.roll(template, 10000)*pulse_amp

OF.update_signal(pileup) # update the signal in order to fit a new trace
amp_withdelay, t0_withdelay, chi2_withdelay = OF.ofamp_withdelay(nconstrain=300)
amp_pileup, t0_pileup, chi2_pileup = OF.ofamp_pileup_iterative(amp_withdelay, t0_
↪withdelay)

print(f"With Delay Fit: amp = {amp_withdelay*1e6:.2f} A, t_0 = {t0_withdelay*1e6} s, ^2␣
↪= {chi2_withdelay:.2f}")
print(f"Pileup Fit: amp = {amp_pileup*1e6:.2f} A, t_0 = {t0_pileup*1e6} s, ^2 = {chi2_
↪pileup:.2f}")
```

```
With Delay Fit: amp = 1.00 A, t_0 = 160.0 s, ^2 = 209503.04
Pileup Fit: amp = 1.00 A, t_0 = 16000.0 s, ^2 = 32407.28
```

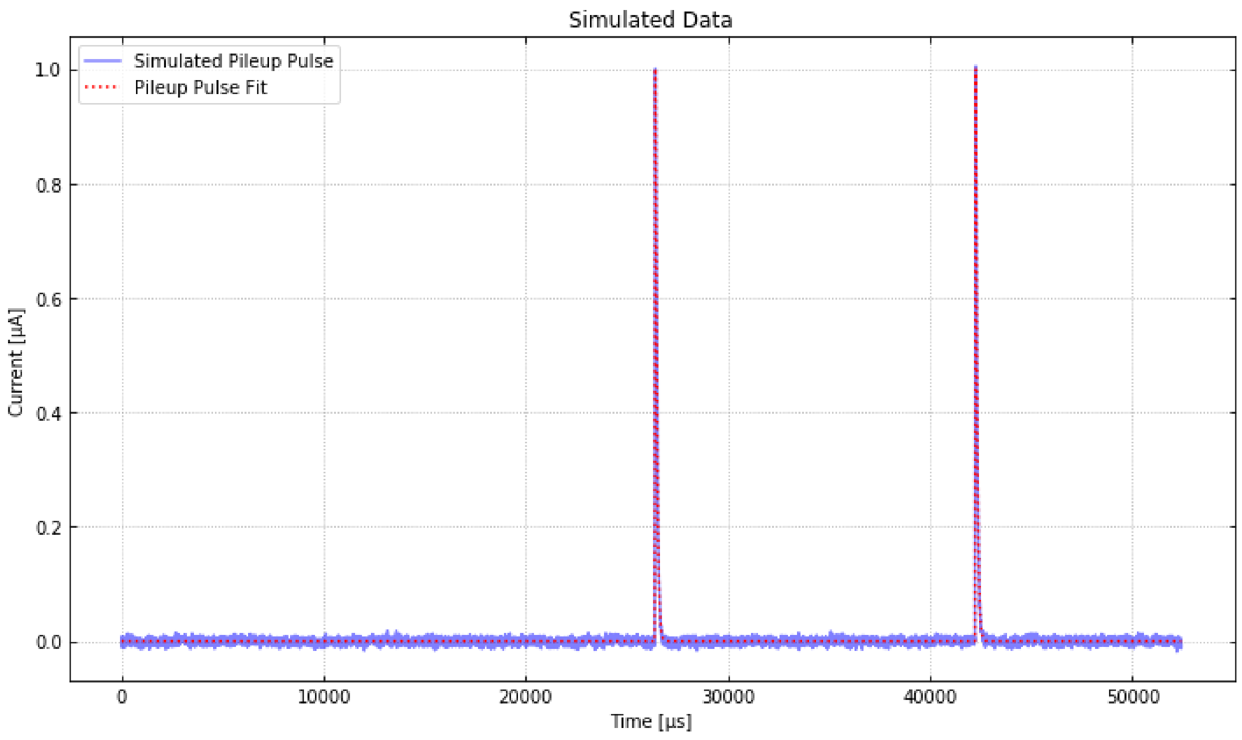
As expected, the pileup optimum filter fit the data very well, as we can see from the chi-squared above.

```
fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(t*1e6, pileup*1e6, label="Simulated Pileup Pulse", color="blue", alpha=0.5)
ax.plot(t*1e6, amp_withdelay*np.roll(template, int(t0_withdelay*fs))*1e6 + \
        amp_pileup*np.roll(template, int(t0_pileup*fs))*1e6,
        label="Pileup Pulse Fit", color="red", linestyle="dotted")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel("Current [A]")
ax.set_xlabel("Time [s]")
ax.set_title("Simulated Data")
lgd = ax.legend(loc="upper left")
ax.tick_params(which="both", direction="in", right=True, top=True)
ax.grid(linestyle="dotted")
fig.tight_layout()
```



7.9 Nonlinear Fitting

What about when we do not have a template? The `qetpy.OFnonlin` class has been written to fit the fall times as well, which is useful for cases where we do not have a template, or we know that the template will not match the data well.

```
pulse_amp = 1e-6
tau_f = 160e-6
tau_r = 20e-6

t = np.arange(len(psd_sim))/fs

pulse = np.exp(-t/tau_f)-np.exp(-t/tau_r)
pulse_shifted = np.roll(pulse, len(t)//2)
template = pulse_shifted/pulse_shifted.max()

noise = qp.gen_noise(psd_sim, fs=fs, ntraces=1)[0]
signal = noise + np.roll(template, 100)*pulse_amp
```

We can try using our "bad" template (with a 66 us fall time), but we will see that the chi-squared indicates a non-ideal fit.

```
OF.update_signal(signal) # update the signal in order to fit a new trace
amp_withdelay, t0_withdelay, chi2_withdelay = OF.ofamp_withdelay(nconstrain=300)

print(f"With Delay Fit: amp = {amp_withdelay*1e6:.2f} A, t_0 = {t0_withdelay*1e6:.2f} s,␣
↪^2 = {chi2_withdelay:.2f}")
```

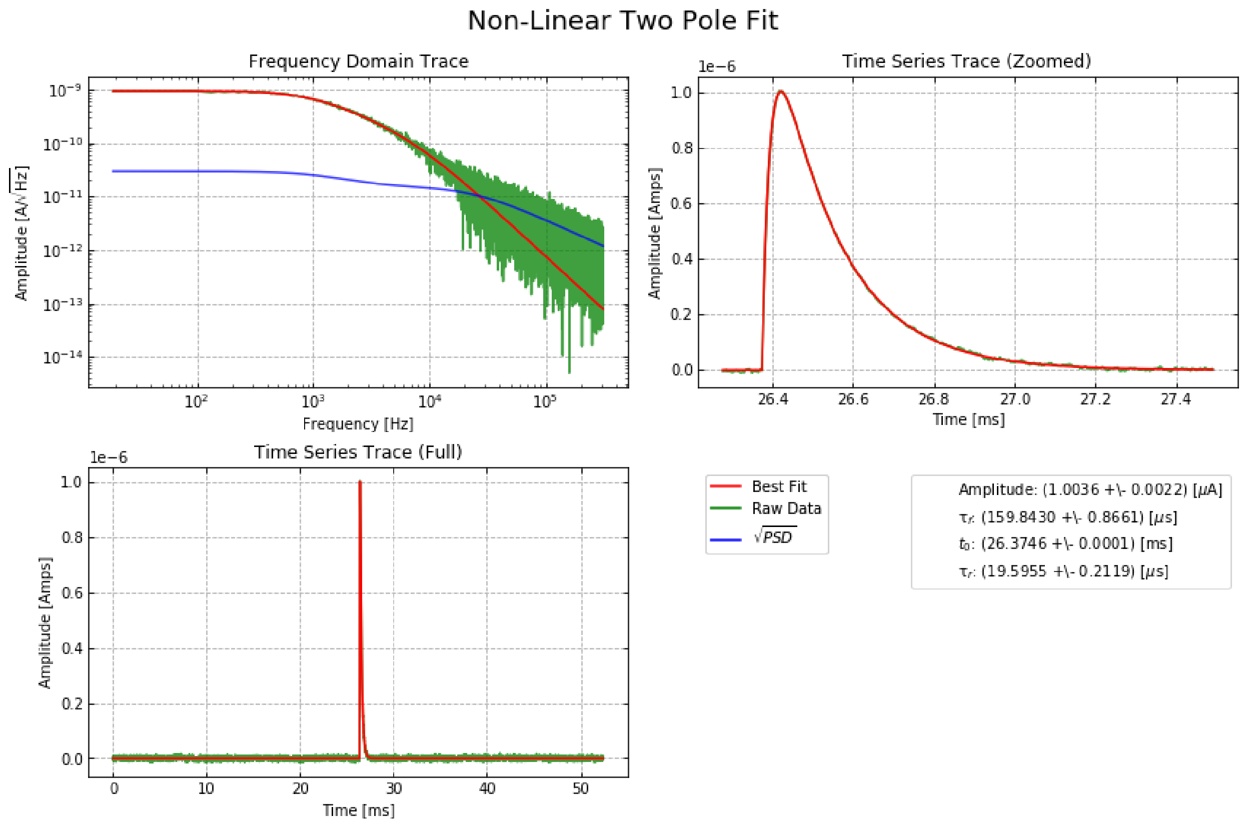
```
With Delay Fit: amp = 1.07 A, t_0 = 163.20 s, ^2 = 72524.20
```

Let's use `qetpy.OFnonlin` to do the fit. To help visualize the fit, we will use the parameter `lgcplot=True` to plot the fit in frequency domain and time domain

```
qp.OFnonlin?
```

```
qp.OFnonlin.fit_falltimes?
```

```
nonlinof = qp.OFnonlin(psd_sim, fs, template=None)
params, error, _, chi2_nonlin, success = nonlinof.fit_falltimes(signal, npolefit=2,␣
↪lgcfullrtn=True, lgcplot=True)
```



```
print(f"Nonlinear fit: ^2 = {chi2_nonlin * (len(nonlinof.data)-nonlinof.dof):.2f}")
```

```
Nonlinear fit: ^2 = 32719.95
```

And we see that the fit using `qetpy.OFnonlin` is great! The chi-squared is near the number of degrees of freedom (32768), which indicates that we have a good fit.

For further documentation on the different fitting functions, please visit https://qetpy.readthedocs.io/en/latest/qetpy.core.html#module-qetpy.core._fitting.

7.10 NSMB Optimal Filter

7.10.1 ### Multi-template OF Fit with One Time Shift Degree of Freedom Between Templates

Bill Page

Date: 190205

This fit is called the "NSMB Optimal Filter" because it fits amplitudes for:

- N signal templates that are allowed to time shift anywhere in the trace
- M background templates that are fixed in time.

It was first written by Matt Pyle to subtract off effects of muons from test facility data. It has been resurrected for a different application: fitting background pileup events in the presence of laser-triggered events with known start times. This new application has required some tweaks to the fit. This notebook describes the general algorithm and the things I've added to it.

```
import numpy as np
import qetpy as qp
import matplotlib.pyplot as plt

from helpers import create_example_pulseplusmuontail, create_example_ttl_leakage_pulses
```

Before getting into how this fit is performed, we motivate it with a few use cases:

1) Fitting out contamination from muons in surface test facility data:

At a test facility this fit will help remove pulses from background particles and long thermal muon tails from the data. In the example below we simulate example data with a pulse randomly spaced in time as well as a muon tail (a long fall time exponential feature across the trace).

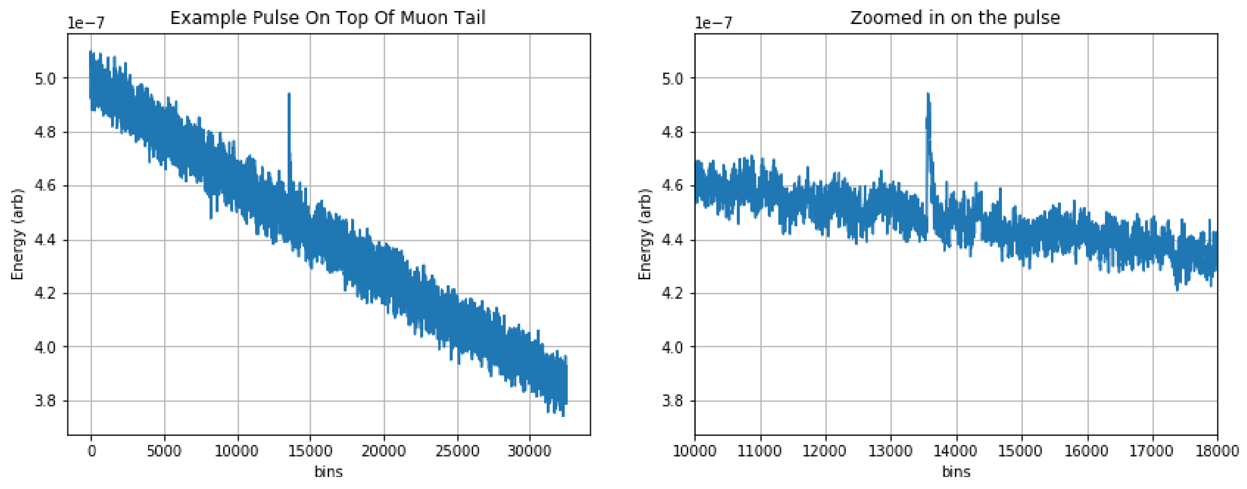
Then we perform the NSMB fit where the free parameters in the are: * the amplitude of the pulse * the time delay of the pulse * the amplitude of the DC component * the amplitude of the muon tail (slope) component

```
# create noise + fake pulse + muon tail, also outputting template for pulse and psd for␣
↪noise
signal, template, psd = create_example_pulseplusmuontail()
signal = (-1)*signal # flip polarity for positive going pulse

plt.figure(figsize=(14,5));
plt.subplot(1,2,1)
plt.plot(signal, '-');
plt.xlabel('bins')
plt.ylabel('Energy (arb)')
plt.title('Example Pulse On Top Of Muon Tail')
```

(continues on next page)

(continued from previous page)

```
plt.grid()
plt.subplot(1,2,2)
plt.plot(signal,'-')
plt.xlim([10000, 18000])
plt.xlabel('bins')
plt.ylabel('Energy (arb)')
plt.title('Zoomed in on the pulse')
plt.grid()
```

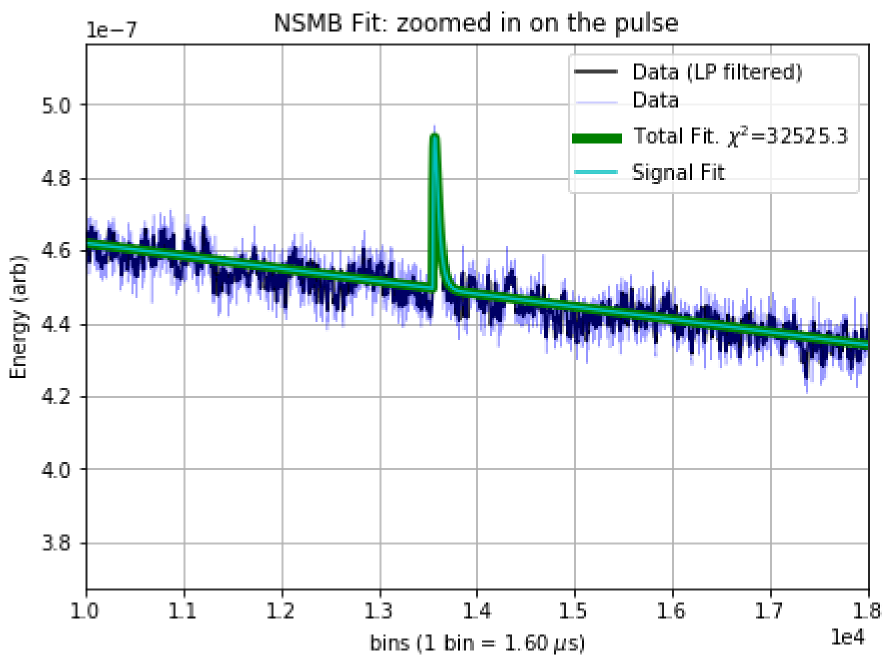


```
nbin = len(signal)

# construct the background templates which are just a slope and a baseline
backgroundtemplates, backgroundtemplatesshifts = qp.get_slope_dc_template_nsmb(nbin)

# setup the NSMB
fs = 625e3 # example sample rate
(psddnu,phi,Pfs, P, sbtemplatef,
 sbtemplatet,iB,B,ns,nb,bitcomb,lfindex) = qp.of_nsmb_setup(template,
↪backgroundtemplates, psd, fs)

# invert the P matrix -- see below if interested in this
iP = qp.of_nsmb_getiP(P)

# give bin window over which to look for signal
indwindow = np.arange(0,len(template))

# put the window into a list (can have multiple list elements)
indwindow_nsmb = [indwindow[:,None].T]

lgcplotnsmb = True
(amps_nsmb, t0_s_nsmb, chi2_nsmb,
chi2_nsmb_lf,resid) = qp.of_nsmb(signal,
                                 phi,
                                 sbtemplatef.T,
                                 sbtemplatet, iP,
                                 psddnu.T, fs,
```

(continues on next page)

(continued from previous page)

```
                                    indwindow_nsmb, ns, nb,
                                    bitcomb, lfindex,
                                    lgcplot=lgcplotnsmb, lgcsaveplots=False)
fig = plt.gcf()
fig.set_size_inches(7,5)
plt.xlim([10000, 18000]);
plt.ylabel('Energy (arb)');
plt.title('NSMB Fit: zoomed in on the pulse');
```



By eye, the 4 free parameters (the amplitude of the pulse, the time delay of the pulse, the amplitude of the DC component, the amplitude of the slope component) have been fit well. One could imagine cutting on the amplitude of the pulse to obtain a better selection of noise, or subtracting the slope fit from the data to remove the contamination from muons, thereby recovering a noise trace that is a closer realization of the fundamental electronic noise.

On to the next use case:

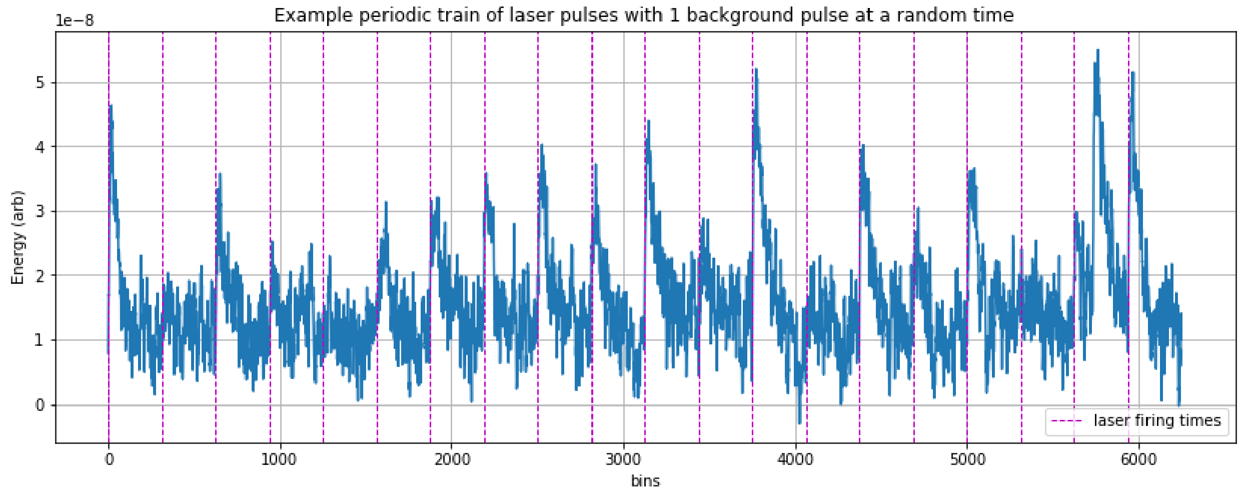
2) Fitting pileup background events in the presence of laser-triggered events with known start times:

```
fs = 625e3
ttlrate = 2e3

# create noise + fake laser trigger pulses at certain start times + a background pulse␣
↪at a random time
# also outputting template for the pulse shapes and psd for noise
signal, template, psd = create_example_ttl_leakage_pulses(fs,ttlrate)
signal = (-1)*signal # flip polarity for positive going pulse

nbin = len(signal)

# get templates for background, as well as
```

(continues on next page)

(continued from previous page)

```
(backgroundtemplates,
backgroundtemplateshifts,
backgroundpolarityconstraint,
indwindow_nsmb) = qp.maketemplate_ttlfit_nsmb(template,
                                              fs,
                                              ttlrate,
                                              lgcconstrainpolarity=True,
                                              lgcpositivepolarity=True,
                                              notch_window_size=1)


plt.figure(figsize=(14,5));
plt.plot(signal, '-');
plt.xlabel('bins')
plt.ylabel('Energy (arb)')
plt.title('Example periodic train of laser pulses with 1 background pulse at a random␣
↪time')
plt.grid()
plt.axvline(x=backgroundtemplateshifts[0],linestyle='--', color='m', linewidth=1, label=
↪'laser firing times')
for ii in range(1, len(backgroundtemplateshifts)):
    plt.axvline(x=backgroundtemplateshifts[ii],linestyle='--', color='m', linewidth=1)
plt.legend();
```



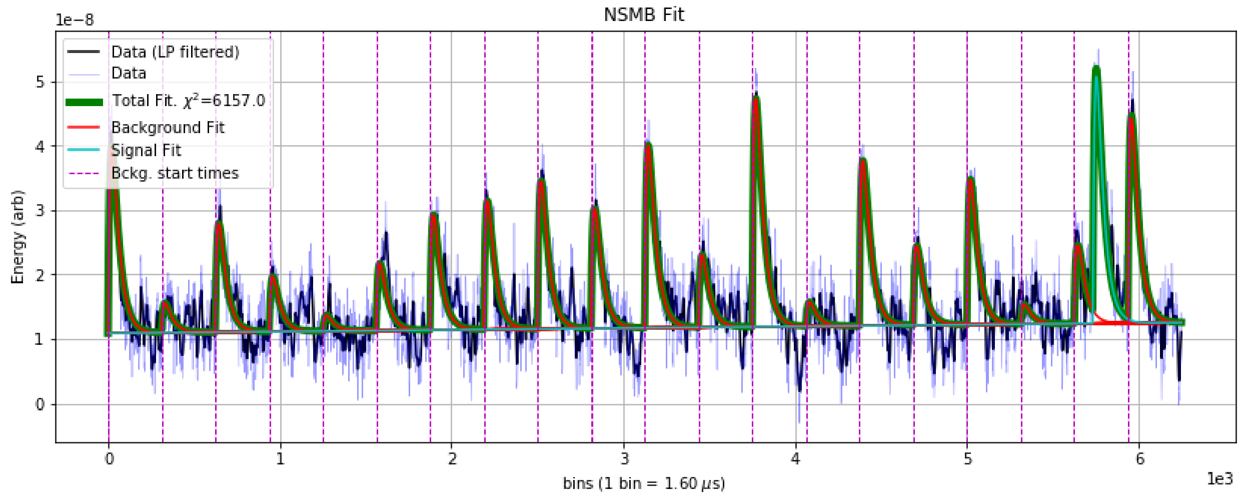
```
# concatenate signal and background template matrices and take FFT
sbtemplatef, sbtemplatet = qp.of_nsmb_ffttemplate(np.expand_dims(template,1),␣
↪backgroundtemplates)

(psddnu, phi, Pfs, P,
sbtemplatef, sbtemplatet, iB,
B, ns, nb, bitcomb, lfindex) = qp.of_nsmb_setup(template, backgroundtemplates, psd, fs)

sigpolarityconstraint = np.ones(1)



lgcplotnsmb = True
```

(continues on next page)

(continued from previous page)

```
(amps_nsmb,t0_s_nsmb,
 chi2_nsmb,chi2_nsmb_lf,
 resid,amps_sig_nsmb_cwindow,
 chi2_nsmb_cwindow,
 t0_s_nsmb_cwindow,
 amp_s_nsmb_int,
 t0_s_nsmb_int,
 chi2_nsmb_int,
 amps_sig_nsmb_cwindow_int,
 chi2_nsmb_cwindow_int,
 t0_s_nsmb_cwindow_int) = qp.of_nsmb_con(signal, phi, Pfs,
                                         P, sbtemplatef.T, sbtemplatet,
                                         psddnu.T, fs, indwindow_nsmb, ns,nb, bitcomb,␣
↪lfindex,
                                         background_templates_
↪shifts=backgroundtemplateshifts,
                                                                              ␣
↪bkgpolarityconstraint=backgroundpolarityconstraint,
                                         sigpolarityconstraint=sigpolarityconstraint,
                                         lgcplot=lgcplotnsmb, lgcsaveplots=False)

fig = plt.gcf()
fig.set_size_inches(14,5)
plt.ylabel('Energy (arb)');
plt.title('NSMB Fit');
```

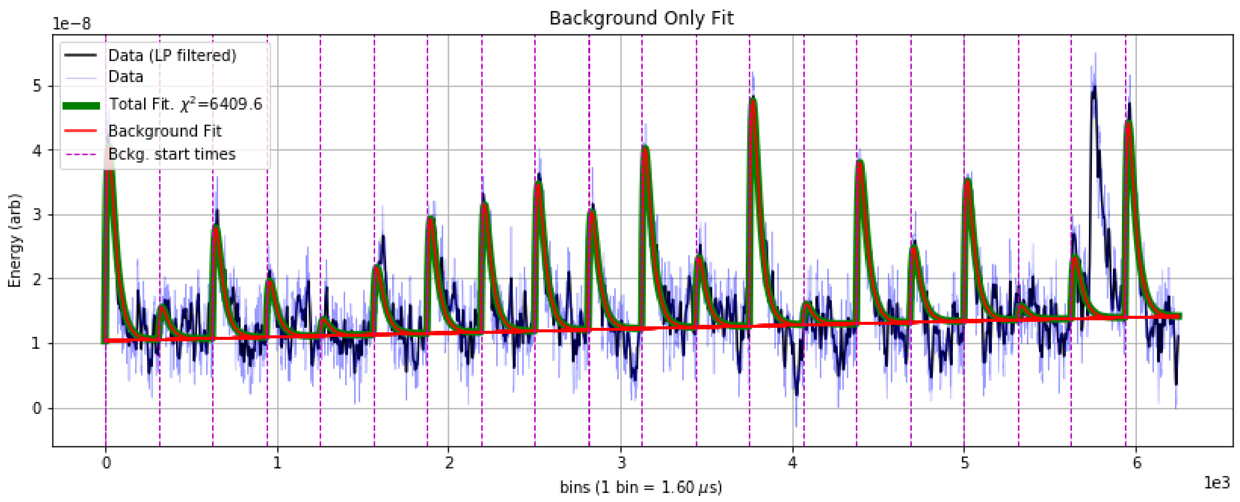
```
all good
```



There is a large background pileup pulse at bin ~ 5700, which is fit as the signal component because it occurs outside of the known times that the laser fires at the detector (given by the vertical lines). One might want to construct a discrimination parameter to determine, in a statistical sense, the likelihood that a background pileup pulse exists in the data.

To do this, we rerun the fit without the signal component and look at the $\Delta\chi^2$ between the two fits:

```
(ampsbonly_nsmb, chi2bonly_nsmb,
 chi2bonly_nsmb_lf) = qp.of_mb(signal, phi, sbtemplatef.T, sbtemplatet,
                                iB, B, psddnu.T, fs, ns, nb, lfindex,
                                background_templates_shifts=backgroundtemplateshifts,
                                bkgpolarityconstraint=backgroundpolarityconstraint,
                                sigpolarityconstraint=sigpolarityconstraint,
                                lgcplot=True, lgcsaveplots=False)
fig = plt.gcf()
fig.set_size_inches(14, 5)
plt.ylabel('Energy (arb)');
plt.title('Background Only Fit');
```



The difference in χ^2 between the fits is 6409.6 - 6157 = 252.6. For noise to produce this $\Delta\chi^2$, it would be a $\sigma = \sqrt{\Delta\chi^2} \sim 16$ fluctuation — so we can say with high confidnce there's background pileup here.

Derivation

The χ^2 for the fit is

:raw-latex:`\begin{equation} \chi^2(\textbf{a}, t_0) = \sum_{k} \frac{|\tilde{S}_n - \sum_{n=1}^{N} a_n e^{-2\pi i t_0 f_k} \tilde{A}_{n,k} - \sum_{m=N}^{N+M}a_m \tilde{A}_{m,k}|^2}{J_k} \end{equation}`

where we have defined

- t_0: the time offset of the N signal templates
- J: the noise PSD
- S: the trace
- A: the templates
- N/M: the number of signal/background templates
- n/m: the index over signal/background templates
- k: the index over frequencies

We analytically solve for the χ^2 minimum and speed up the calculation by using the inverse Fourier transform wherever possible.

Below we derive the solution for 1 signal template and 2 background templates (N=1, M=2), though the algorithm is written for arbitrary values of N and M. The χ^2 for this specific case is given by:

:raw-latex:`\begin{equation} \chi^2(\textbf{a}, t_0) = \sum_{k} \frac{|\tilde{S}_n - a_1 e^{-2\pi i t_0 f_k} \tilde{A}_{1,k} - a_2 \tilde{A}_{2,k} - a_3 \tilde{A}_{3,k}|^2}{J_k}. \end{equation}`

Minimizing the χ^2 with respect to the three amplitudes, by solving $\frac{\partial\chi^2}{\partial a_1} = \frac{\partial\chi^2}{\partial a_2} = \frac{\partial\chi^2}{\partial a_3} = 0$, gives the following equation for the best fit amplitudes, $\mathbf{a}$:

:raw-latex:`begin{equation} textbf{a}= textbf{P}^{-1} cdot textbf{q} tag{1}

end{equation}`

where the **P** matrix is

:raw-latex:`begin{equation} textbf{P} = left(begin{array}{ccc}

sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{1,k}}{J_{k}} & sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}e^{2pi i t_0 f_k}}{J_{k}} & sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}e^{2pi i t_0 f_k}}{J_{k}} \ sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}e^{2pi i t_0 f_k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{2,k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} \ sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}e^{2pi i t_0 f_k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} & sum_{k}frac{tilde{A}_{3,k}^{*}tilde{A}_{3,k}}{J_{k}} \ end{array} right) end{equation}`

and the **q** vector is

:raw-latex:`begin{equation} textbf{q} = left(begin{array}{ccc}

sum_{k}frac{tilde{S}_{k}tilde{A}_{1,k}^{*}e^{2pi i t_0 f_k}}{J_{k}} \ sum_{k}frac{tilde{S}_{k}tilde{A}_{2,k}^{*}}{J_{k}} \ sum_{k}frac{tilde{S}_{k}tilde{A}_{3,k}^{*}}{J_{k}} \ end{array} right) end{equation}`

Equation 1 hides some of the complexity of the calculation because both **P** and **q** depend on t_0, the time delay offset of the signal template. In the algorithm, the amplitudes are calculated for each time delay–**a**(t_0)–and for the global minimum we pick the time delay that gives the lowest χ^2. This initially seems to be getting nasty computationally, where if your data has 4096 time bins you have to do 4096 matrix inversions, but we can speed up the calculation in a couple ways:

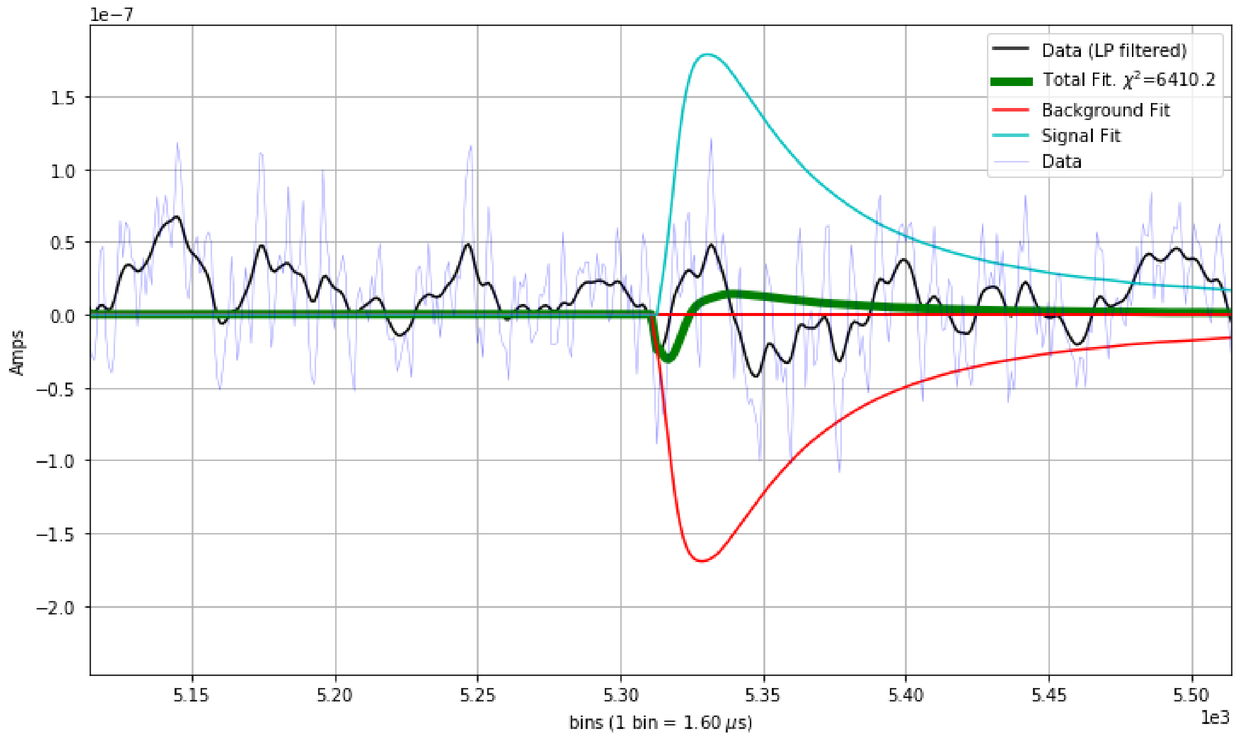
1. The **P** matrix does not depend on the data S and therefore it can be precomputed and inverted for every t_0 before looping over the traces

   - We will see below that to impose amplitude polarity constraints, we will not be able to precompute **P** as it would require too much RAM, and so the algorithm is much slower if amplitude polarity constraints are turned on

2. The first row of **q** and the first row and column of **P**, except for the (1,1) element, are inverse Fourier transforms. Using the same old trick that is used in the stanard optimal filter algorithm with a time delay, these elements can be computed quickly in $\mathcal{O}(n \log n)$ time.

Polarity Constraint

The location in the parameter space where there is an absolute miniumum of the χ^2, given by Equation 1, can be non-physical. What is an example of a non-physical best fit result? Here you go:



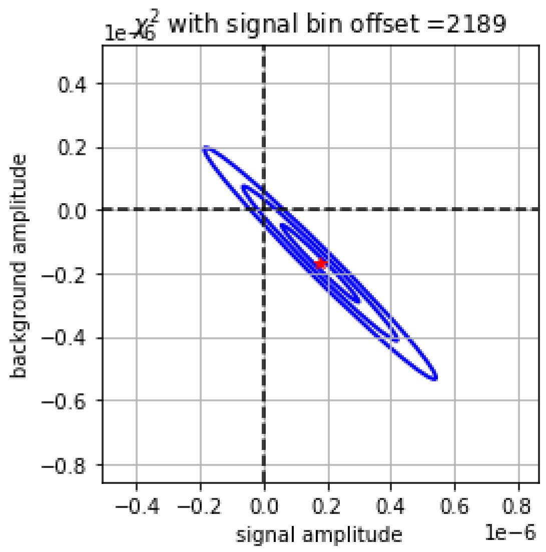
What is going on above? * If there is not a background pileup pulse, the minimum of the χ^2 can have the signal template interfere with the background template in order to fit noise.

- To fix this, we can impose physical constraints on the amplitude values in the fit, by which we mean that if the detector has been biased so that pulses go in the positive direction, we disallow negative fit amplitudes.

The procedure for imposing an amplitude polarity constraint is more involved for this multi-dimensional amplitude space than for the simple 1D optimal filter. For the problematic fit above, for constraining the amplitudes to be positive, what is a robust but fast way to find the minimum in the allowed region (the region with no negative amplitudes)?

- A flawed procedure would be to constrain the negative background amplitude to 0
  - It could very well be that then, on the second minimization, the signal amplitude would be fit negative
    - Do you then set the signal amplitude to 0 as well?
      - Sequential approaches are a bad idea when doing highly correlated fits!
- A robust and fast procedure is to use the gradient to determine which amplitudes to constrain to 0

The below plot shows the χ^2 gradient, determined with the covariance matrix between signal and background amplitudes, for the fit above, with the absolute minimum in red:



We can calculate the gradient at different points in the parameter space quickly because the covariance matrix (**E**) and the Hessian(**H**) comes for free from unconstrained calculation, from their relation to the P matrix:

:raw-latex:`\begin{equation} \textbf{E} = \left(\begin{array}{ccc} \sigma_{11}^{2} & \sigma_{12}^{2} & \sigma_{13}^{2} \\ \sigma_{12}^{2} & \sigma_{22}^{2} & \sigma_{23}^{2} \\ \sigma_{13}^{2} & \sigma_{23}^{2} & \sigma_{33}^{2} \\ \end{array} \right) = \textbf{P}^{-1} = \left(\begin{array}{ccc} \sum_{k}\frac{\tilde{A}_{1,k}^{*}\tilde{A}_{1,k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{1,k}^{*}\tilde{A}_{2,k}e^{2\pi i t_0 f_k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{1,k}^{*}\tilde{A}_{3,k}e^{2\pi i t_0 f_k}}{J_{k}} \\ \sum_{k}\frac{\tilde{A}_{1,k}^{*}\tilde{A}_{2,k}e^{2\pi i t_0 f_k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{2,k}^{*}\tilde{A}_{2,k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{2,k}^{*}\tilde{A}_{3,k}}{J_{k}} \\ \sum_{k}\frac{\tilde{A}_{1,k}^{*}\tilde{A}_{3,k}e^{2\pi i t_0 f_k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{2,k}^{*}\tilde{A}_{3,k}}{J_{k}} & \sum_{k}\frac{\tilde{A}_{3,k}^{*}\tilde{A}_{3,k}}{J_{k}} \\ \end{array} \right)^{-1}. \end{equation}`

This correspondence might be more familiar by remembering that the standard 1D optimal filter amplitude estimate variance is

:raw-latex:`\begin{equation} \sigma_{\hat{a}}^{2} = \left( \sum_{k}\frac{|\tilde{A}_{1,k}|^2}{J_{k}} \right)^{-1} \end{equation}`

The gradient anywhere in the parameter space can be done with the Hessian, where $\mathbf{H} = \frac{1}{2}\mathbf{E}^{-1} = \frac{1}{2}\mathbf{P}$ and

$$\mathbf{H}_{i,j} = \frac{\partial^2\chi^2}{\partial a_i \partial a_j}.$$

The gradient at a point **a** in the parameter sapce is just given by

:raw-latex:`\begin{equation} \nabla{\chi^2} = \textbf{H} \cdot (\textbf{a} - \textbf{a}_{abs}) \end{equation}`

In practice, here is how we use gradient information:

- we find the abolute minimum. if some amplitudes are in a disallowed region of the space we:
  - we check the gradient at the closest point in the allowed parameter space
  - for those amplitudes in the disallowed region:
    * if the gradient points into the disallowed region then those amplitudes are set to zero
    * if the gradient points into the allowed region then those amplitudes are not set to zero (allowed to float)

        * we refit with the certain amplitudes taken out of the fit

    – we redo this process until, for all the amplitudes that are set to zero, the gradient points into the disallowed region

        * if this check fails then we output a warning flag in the fit

            · have never seen this failure

        * typically this only takes one iteration! much faster than a gradient descent!

In the algorithm a few quantities are precomputed to speed up calculations. In the code, these quantities are named and defined as below:

- **PF**, a 3 by 3 by k matrix

:raw-latex:`begin{equation} textbf{PF} = left(begin{array}{ccc}

frac{tilde{A}_{1,k}^{*}tilde{A}_{1,k}}{J_{k}} & frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}}{J_{k}} & frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}}{J_{k}} \ frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}}{J_{k}} & frac{tilde{A}_{2,k}^{*}tilde{A}_{2,k}}{J_{k}} & frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} \ frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}}{J_{k}} & frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} & frac{tilde{A}_{3,k}^{*}tilde{A}_{3,k}}{J_{k}} \ end{array} right) end{equation}`

- **PFS**, a 3 by 3 matrix

:raw-latex:`begin{equation} textbf{PFS} = left(begin{array}{ccc}

sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{1,k}}{J_{k}} & sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}}{J_{k}} & sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}}{J_{k}} \ sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{2,k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{2,k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} \ sum_{k}frac{tilde{A}_{1,k}^{*}tilde{A}_{3,k}}{J_{k}} & sum_{k}frac{tilde{A}_{2,k}^{*}tilde{A}_{3,k}}{J_{k}} & sum_{k}frac{tilde{A}_{3,k}^{*}tilde{A}_{3,k}}{J_{k}} \ end{array} right) end{equation}`

- ϕ, a 3 by k matrix

:raw-latex:`begin{equation} phi = left(begin{array}{ccc}

frac{tilde{A}_{1,k}^{*}}{J_{k}} \ frac{tilde{A}_{2,k}^{*}}{J_{k}} \ frac{tilde{A}_{3,k}^{*}}{J_{k}} \ end{array} right) end{equation}`

CHAPTER EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

  Developed and maintained by Caleb Fink (https://github.com/cwfink) and Sam Watkins (https://github.com/slwatkins)