corrfitter Documentation

Release 6.0

G.P. Lepage

February 07, 2017

1 **corrfitter - Least-Squares Fit to Correlators** 3
1.1 Introduction . . . 3
1.2 Basic Fits . . . 3
1.3 Faster Fits . . . 7
1.4 Faster Fits — Postive Parameters . . . 8
1.5 Faster Fits — Marginalization . . . 9
1.6 Faster Fits — Chained Fits . . . 10
1.7 Faster Fits — Faster Fitters . . . 10
1.8 Faster Fits — Processed Datasets . . . 11
1.9 Variations . . . 11
1.10 Very Fast (But Limited) Fits . . . 12
1.11 3-Point Correlators . . . 13
1.12 Testing Fits with Simulated Data . . . 14
1.13 Bootstrap Analyses . . . 15
1.14 Implementation . . . 16
1.15 Correlator Model Objects . . . 16
1.16 `corrfitter.CorrFitter` Objects . . . 21
1.17 `corrfitter.EigenBasis` Objects . . . 24
1.18 Fast Fit Objects . . . 28

2 **Annotated Example: Two-Point Correlator** 31
2.1 Introduction . . . 31
2.2 Code . . . 31
2.3 Results . . . 33
2.4 Correlated Data? . . . 34
2.5 Fast Fit and Effective Mass . . . 35

3 **Annotated Example: Transition Form Factor and Mixing** 37
3.1 Introduction . . . 37
3.2 Code . . . 37
3.3 Results . . . 43
3.4 Variation: Marginalization . . . 46
3.5 Variation: Chained Fit . . . 48
3.6 Test the Analysis . . . 49
3.7 Mixing . . . 51

4 **Annotated Example: Matrix Correlator** 55
4.1 Introduction . . . 55
4.2 Code . . . 55
4.3 Results . . . 58
4.4 Fit Stability . . . 60

4.5 Alternative Organization 62

5 Indices and tables **67**

Python Module Index **69**

Index **71**

Contents:

CHAPTER

ONE

CORRFITTER - LEAST-SQUARES FIT TO CORRELATORS

1.1 Introduction

This module contains tools that facilitate least-squares fits, as functions of time `t`, of simulation (or other statistical) data for 2-point and 3-point correlators of the form:

```
Gab(t)     =  <b(t) a(0)>
Gavb(t,T)  =  <b(T) V(t) a(0)>
```

where `T > t > 0`. Each correlator is modeled using `corrfitter.Corr2` for 2-point correlators, or `corrfitter.Corr3` for 3-point correlators in terms of amplitudes for each source `a`, sink `b`, and vertex `V`, and the energies associated with each intermediate state. The amplitudes and energies are adjusted in the least-squares fit to reproduce the data; they are defined in a shared prior (typically a dictionary).

An object of type `corrfitter.CorrFitter` describes a collection of correlators and is used to fit multiple models to data simultaneously. Fitting multiple correlators simultaneously is important if there are statistical correlations between the correlators. Any number of correlators may be described and fit by a single `corrfitter.CorrFitter` object.

We now review the basic features of `corrfitter`. These features are also illustrated for real applications in a series of annotated examples following this section. Impatient readers may wish to jump directly to these examples.

About Printing: The examples in this tutorial use the `print` function as it is used in Python 3. Drop the outermost parenthesis in each `print` statement if using Python 2; or add

```
from __future__ import print_function
```

at the start of your file.

1.2 Basic Fits

To illustrate, consider data for two 2-point correlators: `Gaa` with the same source and sink (`a`), and `Gab` which has source `a` and (different) sink `b`. The data are contained in a dictionary `data`, where `data['Gaa']` and `data['Gab']` are one-dimensional arrays containing values for `Gaa(t)` and `Gab(t)`, respectively, with `t=0,1,2...63`. Each array element in `data['Gaa']` and `data['Gab']` is a Gaussian random variable of type `gvar.GVar`, and specifies the mean and standard deviation for the corresponding data point:

```
>>> print(data['Gaa'])
[0.1597910(41) 0.0542088(31) ... ]
>>> print(data['Gab'])
[0.156145(18) 0.102335(15) ... ]
```

`gvar.GVars` also capture statistical correlations between different pieces of data, if they exist.

We want to fit this data to the following formulas:

```
Gaa(t,N) = sum_i=0..N-1  a[i]**2 * exp(-E[i]*t)
Gab(t,N) = sum_i=0..N-1  a[i]*b[i] * exp(-E[i]*t)
```

Our goal is to find values for the amplitudes, `a[i]` and `b[i]`, and the energies, `E[i]`, so that these formulas reproduce the average values for `Gaa(t,N)` and `Gab(t,N)` that come from the data, to within the data's statistical errors. We use the same `a[i]`s and `E[i]`s in both formulas. The fit parameters used by the fitter are the `a[i]`s and `b[i]`s, as well as the differences `dE[i]=E[i]-E[i-1]` for `i>0` and `dE[0]=E[0]`. The energy differences are usually positive by construction (see below) and are easily converted back to energies using:

```
E[i] = sum_j=0..i dE[j]
```

A typical code has the following structure:

```
import corrfitter as cf

def main():
    data = make_data('mcfile')          # user-supplied routine
    models = make_models()              # user-supplied routine
    N = 4                               # number of terms in fit functions
    prior = make_prior(N)               # user-supplied routine
    fitter = cf.CorrFitter(models=models)
    fit = fitter.lsqfit(data=data, prior=prior)   # do the fit
    print(fit)
    print_results(fit, prior, data)     # user-supplied routine

...

if __name__ == '__main__':
    main()
```

We discuss each user-supplied routine in turn.

1.2.1 a) make_data

`make_data('mcfile')` creates the dictionary containing the data that is to be fit. Typically such data comes from a Monte Carlo simulation. Exactly how the data are assembled depends upon how Monte Carlo results are stored.

Imagine, for example, that the simulation creates a file called `'mcfile'` with layout

```
# first correlator: each line has Gaa(t) for t=0,1,2...63
Gaa  0.159774739530e+00 0.541793561501e-01 ...
Gaa  0.159751906801e+00 0.542054488624e-01 ...
Gaa  ...
.
.
.
# second correlator: each line has Gab(t) for t=0,1,2...63
Gab  0.155764170032e+00 0.102268808986e+00 ...
Gab  0.156248435021e+00 0.102341455176e+00 ...
Gab  ...
.
.
.
```

where each line is one Monte Carlo measurement for one or the other correlator, as indicated by the tags at the start of the line. (Lines for Gab may be interspersed with lines for Gaa since every line has a tag.) A data file in this format can be analyzed using:

```
import gvar as gv
import corrfitter as cf

def make_data(filename):
    dset = cf.read_dataset(filename)
    return gv.dataset.avg_data(dset)
```

This reads the data from the file into a dataset, which is a dictionary whose values are two-dimenional arrays where the first index labels the Monte Carlo sample, and the second index labels time: for example,

```
>>> print(dset['Gaa'])
[ [0.159774739530e+00 0.541793561501e-01 ... ],
  [0.159751906801e+00 0.542054488624e-01 ... ],
  ...]
```

Function `gvar.dataset.avg_data()` then averages over the Monte Carlo samples. Thus `data = make_data('mcfile')` creates a dictionary where `data['Gaa']` is a one-dimensional array of `gvar.GVars`, indexed by time, obtained by averaging over the `Gaa` data in the `'mcfile'`, and `data['Gab']` is a similar array for the `Gab` correlator. The correlator values for different `ts` are typically correlated with each other.

Other data formats are readily adapted to this purpose. For example, the same Monte Carlo data might be stored in an hdf5 file:

```
import h5py
import gvar as gv

def make_data(filename):
    h5file = h5py.File(filename, 'r')
    dset = dict(
        Gaa=h5file['/run5/Gaa'], Gab=hfile['/run5/Gab']
        )
    return gv.dataset.avg_data(dset)
```

Here we assume `h5file['/run5/Gaa']` and `hfile['/run5/Gab']` are hdf5 datasets that have been configured, again, as two-dimensional numpy arrays, where the first index is the Monte Carlo sample (configuration) index, and the second index is time.

Function `corrfitter.read_dataset()` can read hdf5 files, so this last example could also be handled by

```
def make_data(filename):
    dset = cf.read_dataset(filename, h5group='/run5')
    return gv.dataset.avg_data(dset)
```

provided `filename` ends in `'.h5'`. This reads in all hdf5 datasets in group `/run5`.

1.2.2 b) make_models

`make_models()` identifies which correlators in the fit data are to be fit, and specifies theoretical models (that is, fit functions) for these correlators:

```
import corrfitter as cf

def make_models():
    tdata = range(64)
```

```
    tfit = tdata[2:]
    models = [
        cf.Corr2(datatag='Gaa', tdata=tdata, tfit=tfit, a='a', b='a', dE='dE'),
        cf.Corr2(datatag='Gab', tdata=tdata, tfit=tfit, a='a', b='b', dE='dE'),
        ]
    return models
```

For each correlator, we specify: the key used in the input data dictionary `data` for that correlator (`datatag`); the `t` values, `tdata=[0,1,2...63]`, associated with each element of the fit data for the correlator; the subset of `tdata` values, `tfit=[2,3,4...63]`, to be used in the fit; and fit-parameter labels for the source (`a`) and sink (`b`) amplitudes, and for the intermediate energy-differences (`dE`). Fit-parameter labels identify the parts of the prior, discussed below, corresponding to the actual fit parameters (the labels are dictionary keys). Here the two models, for `Gaa` and `Gab`, are identical except for the data tags and the sinks. `make_models()` returns a list of models; the only parts of the input fit data that are fit are those for which a model is specified in `make_models()`.

Note that if there is data for `Gba(t,N)` in addition to `Gab(t,N)`, and `Gba = Gab`, then the (weighted) average of the two data sets will be fit if `models[1]` is replace by:

```
cf.Corr2(
    datatag='Gab', tmin=1, tmax=63, a='a', b='b', dE='dE',
    otherdata='Gba',
    )
```

Alternatively one could add a third `Corr2` to `models` for `Gba`, but it is more efficient to combine it with `Gab`, before the fit, if they are equivalent.

The arrays `tdata` and `tfit` provide more flexibility than is often needed. Here, because there is data for all `t` values starting with 0, we could have defined the correlator objects more simply, in terms of the minimum and maximum `t` values used in the fit: for example,

```
cf.Corr2(datatag='Gaa', tmin=2, tmax=63, a='a', b='a', dE='dE')
```

`corrfitter.Corr2` creates the obvious choices for `tdata` and `tfit` from the information given.

1.2.3 c) make_prior

This routine defines the fit parameters that correspond to each fit-parameter label used in `make_models()` above. It also assigns *a priori* values to each parameter, expressed in terms of Gaussian random variables (`gvar.GVars`), with a mean and standard deviation. The prior is built using a Python dictionary (we use `gvar.BufferDict` but others would work):

```
import gvar as gv

def make_prior(N):
    prior = gvar.BufferDict()
    prior['a'] = gv.gvar(N * ['0.1(5)'])
    prior['b'] = gv.gvar(N * ['1(5)'])
    prior['dE'] = gv.gvar(N * ['0.25(25)'])
    return prior
```

`make_prior(N)` associates arrays of `N` Gaussian random variables (`gvar.GVars`) with each fit-parameter label, enough for `N` terms in the fit function. These are the *a priori* values for the fit parameters, and they can be retrieved using the label: setting `prior=make_prior(N)`, for example, implies that `prior['a'][i]`, `prior['b'][i]` and `prior['dE'][i]` are the *a priori* values for `a[i]`, `b[i]` and `dE[i]` in the fit functions (see above). The *a priori* value for each `a[i]` here is set to `0.1±0.5`, while that for each `b[i]` is `1±5`:

```
>>> print(prior['a'])
[0.10(50) 0.10(50) 0.10(50) 0.10(50)]
>>> print(prior['b'])
[1.0(5.0) 1.0(5.0) 1.0(5.0) 1.0(5.0)]
```

Similarly the *a priori* value for each energy difference is `0.25±0.25`. (See the `lsqfit` documentation for further information on priors.)

1.2.4 d) print_results

The actual fit is done by `fit=fitter.lsqfit(...)`, and `print(fit)` right afterwards prints a summary of the fit results. Further results are reported by `print_results(fit, prior, data)`: for example,

```
def print_results(fit, prior, data):
    print(fit)
    a = fit.p['a']                              # array of a[i]s
    b = fit.p['b']                              # array of b[i]s
    dE = fit.p['dE']                            # array of dE[i]s
    E = np.cumsum(dE)                           # array of E[i]s
    print('Best fit values:)
    print('     a[0] =',a[0])
    print('     b[0] =',b[0])
    print('     E[0] =',E[0])
    print('b[0]/a[0] =',b[0]/a[0])
    outputs = {'E0':E[0], 'a0':a[0], 'b0':b[0], 'b0/a0':b[0]/a[0]}
    inputs = {'a'=prior['a'], 'b'=prior['b'], 'dE'=prior['dE'],
              'data'=[data[k] for k in data])
    print(fit.fmt_errorbudget(outputs, inputs))
```

The best-fit values from the fit are contained in `fit.p` and are accessed using the labels defined in the prior and the *`corrfitter.Corr2`* models. Variables like `a[0]` and `E[0]` are `gvar.GVar` objects that contain means and standard deviations, as well as information about any correlations that might exist between different variables (which is relevant for computing functions of the parameters, like `b[0]/a[0]` in this example).

The last line of `print_results(fit,prior,data)` prints an error budget for each of the best-fit results for `a[0]`, `b[0]`, `E[0]` and `b[0]/a[0]`, which are identified in the print output by the labels `'a0'`, `'b0'`, `'E0'` and `'b0/a0'`, respectively. The error for any fit result comes from uncertainties in the inputs — in particular, from the fit data and the priors. The error budget breaks the total error for a result down into the components coming from each source. Here the sources are the *a priori* errors in the priors for the `'a'` amplitudes, the `'b'` amplitudes, and the `'dE'` energy differences, as well as the errors in the fit data `data`. These sources are labeled in the print output by `'a'`, `'b'`, `'dE'`, and `'data'`, respectively. (See the `gvar`/`lsqfit` tutorial for further details on partial standard deviations and `gvar.fmt_errorbudget()`.)

Plots of the fit data divided by the fit function, for each correlator, are displayed by calling `fit.show_plots()` provided the `matplotlib` module is present.

1.3 Faster Fits

Good fits often require fit functions with several exponentials and many parameters. Such fits can be costly. One strategy that can speed things up is to use fits with fewer terms to generate estimates for the most important parameters. These estimates are then used as starting values for the full fit. The smaller fit is usually faster, because it has fewer parameters, but the fit is not adequate (because there are too few parameters). Fitting the full fit function is usually faster given reasonable starting estimates, from the smaller fit, for the most important parameters. Continuing with the example from the previous section, the code

```
data = make_data('mcfile')
fitter = cf.CorrFitter(models=make_models())
p0 = None
for N in [1,2,3,4,5,6,7,8]:
    prior = make_prior(N)
    fit = fitter.lsqfit(data=data, prior=prior, p0=p0)
    print_results(fit, prior, data)
    p0 = fit.pmean
```

does fits using fit functions with `N=1...8` terms. Parameter mean-values `fit.pmean` from the fit with `N` exponentials are used as starting values `p0` for the fit with `N+1` exponentials, hopefully reducing the time required to find the best fit for `N+1`.

1.4 Faster Fits — Postive Parameters

Priors used in *`corrfitter.CorrFitter`* assign an *a priori* Gaussian/normal distribution to each parameter. It is possible instead to assign a log-normal distribution, which forces the corresponding parameter to be positive. Consider, for example, energy parameters labeled by `'dE'` in the definition of a model (*e.g.*, `Corr2(dE='dE',...)`). To assign log-normal distributions to these parameters, include their logarithms in the prior and label the logarithms with `'log(dE)'`: for example, in `make_prior(N)` use

```
prior['log(dE)'] = gv.log(gv.gvar(N * ['0.25(25)']))
```

instead of `prior['dE'] = gv.gvar(N * ['0.25(25)'])`. The fitter then uses the logarithms as the fit parameters. The original `'dE'` parameters are recovered (automatically) inside the fit function from exponentials of the `'log(dE)'` fit parameters.

Using log-normal distributions where possible can significantly improve the stability of a fit. This is because otherwise the fit function typically has many symmetries that lead to large numbers of equivalent but different best fits. For example, the fit functions `Gaa(t,N)` and `Gab(t,N)` above are unchanged by exchanging `a[i]`, `b[i]` and `E[i]` with `a[j]`, `b[j]` and `E[j]` for any `i` and `j`. We can remove this degeneracy by using a log-normal distribution for the `dE[i]`s since this guarantees that all `dE[i]`s are positive, and therefore that `E[0],E[1],E[2]...` are ordered (in decreasing order of importance to the fit at large `t`).

Another symmetry of `Gaa` and `Gab`, which leaves both fit functions unchanged, is replacing `a[i],b[i]` by `-a[i],-b[i]`. Yet another is to add a new term to the fit functions with `a[k],b[k],dE[k]` where `a[k]=0` and the other two have arbitrary values. Both of these symmetries can be removed by using a log-normal distribution for the `a[i]` priors, thereby forcing all `a[i]>0`.

The log-normal distributions for the `a[i]` and `dE[i]` are introduced into the code example above by changing the corresponding labels in `make_prior(N)`, and taking logarithms of the corresponding prior values:

```
import gvar as gv
import corrfitter as cf

def make_models():                              # same as before
    models = [
        cf.Corr2(datatag='Gaa', tmin=2, tmax=63, a='a', b='a', dE='dE'),
        cf.Corr2(datatag='Gab', tmin=2, tmax=63, a='a', b='b', dE='dE'),
        ]
    return models

def make_prior(N):
    prior = gv.BufferDict()
    prior['log(a)'] = gv.log(gv.gvar(N * ['0.1(5)']))
    prior['b'] = gv.gvar(N * ['1(5)'])
```

```
    prior['log(dE)'] = gv.log(gv.gvar(N * ['0.25(25)']))
    return prior
```

This replaces the original fit parameters, `a[i]` and `dE[i]`, by new fit parameters, `log(a)[i]` and `log(dE)[i]`. The *a priori* distributions for the logarithms are Gaussian/normal, with priors of `log(0.1±0.5)` and `log(0.25±0.25)` for the `log(a)`s and `log(dE)`s respectively.

Note that the labels are unchanged here in `make_models()`. It is unnecessary to change labels in the models; *corrfitter.CorrFitter* will automatically connect the modified terms in the prior with the appropriate terms in the models. This allows one to switch back and forth between log-normal and normal distributions without changing the models (or any other code) — only the names in the prior need be changed. *corrfitter.CorrFitter* also supports "sqrt-normal" distributions, and other distributions, as discussed in the `lsqfit` documentation.

Finally note that another option for stabilizings fits involving many sources and sinks is to generate priors for the fit amplitudes and energies using *corrfitter.EigenBasis*.

1.5 Faster Fits — Marginalization

Often we care only about parameters in the leading term of the fit function, or just a few of the leading terms. The non-leading terms are needed for a good fit, but we are uninterested in the values of their parameters. In such cases the non-leading terms can be absorbed into the fit data, leaving behind only the leading terms to be fit (to the modified fit data) — non-leading parameters are, in effect, integrated out of the analysis, or *marginalized*. The errors in the modified data are adjusted to account for uncertainties in the marginalized terms, as specified by their priors. The resulting fit function has many fewer parameters, and so the fit can be much faster.

Continuing with the example in *Faster Fits*, imagine that `Nmax=8` terms are needed to get a good fit, but we only care about parameter values for the first couple of terms. The code from that section can be modified to fit only the leading `N` terms where `N<Nmax`, while incorporating (marginalizing) the remaining, non-leading terms as corrections to the data:

```
Nmax = 8
data = make_data('mcfile')
models = make_models()
fitter = cf.CorrFitter(models=make_models())
prior = make_prior(Nmax)        # build priors for Nmax terms
p0 = None
for N in [1,2,3]:               # fit N terms
    fit = fitter.lsqfit(data=data, prior=prior, p0=p0, nterm=N)
    print_results(fit, prior, data)
    p0 = fit.pmean
```

Here the `nterm` parameter in `fitter.lsqfit` specifies how many terms are used in the fit functions. The prior specifies `Nmax` terms in all, but only parameters in `nterm=N` terms are varied in the fit. The remaining terms specified by the prior are automatically incorporated into the fit data by *corrfitter.CorrFitter*.

Remarkably this method is usually as accurate with `N=1` or `2` as a full `Nmax`-term fit with the original fit data; but it is much faster. If this is not the case, check for singular priors, where the mean is much smaller than the standard deviation. These can lead to singularities in the covariance matrix for the corrected fit data. Such priors are easily fixed: for example, use `gvar.gvar('0.1(1.0)')` rather than `gvar.gvar('0(1)')`. In some situations an SVD cut (see below) can also help.

1.6 Faster Fits — Chained Fits

Large complicated fits, where lots of models and data are fit simultaneously, can take a very long time. This is especially true if there are strong correlations in the data. Such correlations can also cause problems from numerical roundoff errors when the inverse of the data's covariance matrix is computed for the `chi**2` function, requiring large SVD cuts which can degrade precision (see below). An alternative approach is to use *chained* fits. In a chained fit, each model is fit by itself in sequence, but with the best-fit parameters from each fit serving as priors for fit parameters in the next fit. All parameters from one fit become fit parameters in the next, including those parameters that are not explicitly needed by the next fit (since they may be correlated with the input data for the next fit or with its priors). Statistical correlations between data/priors from different models are preserved throughout (approximately).

The results from a chained fit are identical to a standard simultaneous fit in the limit of large statistics (that is, in the Gaussian limit), but a chained fit usually involves fitting only a single correlator at a time. Single-correlator fits are typically much faster than simultaneous multi-correlator fits, and roundoff errors (and therefore SVD cuts) are much less of a problem.

Converting to chained fits is trivial: simply replace `fit = fitter.lsqfit(...)` by `fit = fitter.chained_lsqfit(...)`. The output from this function comes from the last fit in the chain, whose fit results represent the cummulative results of the entire chain of fits. Results from the different links in the chain — that is, from the fits for individual models — are displayed using `print(fit.formatall())`.

There are various ways of chaining fits. For example, setting

```
models = [m1, m2, (m3a, m3b), m4]
```

causes models `m1`, `m2` and `m4` to be fit separately, but fits models `m3a` and `m3b` together in a single simultaneous fit:

```
m1 -> m2 -> (simultaneous fit of m3a, m3b) -> m4
```

Simultaneous fits make sense when there is lots of overlap between the parameters for the different models.

Another option is

```
models = [m1, m2, [m3a,m3b], m4]
```

in `fitter.chained_lsqfit` which causes the following chain of fits:

```
m1 -> m2 -> (parallel fit of m3a, m3b) -> m4
```

Here the output from `m1` is used in the prior for fit `m2`, and the output from `m2` is used as the prior for a parallel fit of `m3a` and `m3b` together — that is, `m3a` and `m3b` are not chained, but rather are fit in parallel with each using a prior from fit `m2`. The result of the parallel fit of `[m3a,m3b]` is used as the prior for `m4`. Parallel fits make sense when there is little overlap between the parameters used by the different fits.

1.7 Faster Fits — Faster Fitters

When fits take many iterations to converge (or converge to an obviously wrong result), it is worthwhile trying a different fitter. The `lsqfit` module, which is used by *`corrfitter`* for fitting, offers a variety of alternative fitting algorithms that can sometimes be much faster (2 or 3 times faster). These are deployed by adding extra directives for `lsqfit` when constructing the fitter or when doing the fit: for example,

```
import corrfitter as cf

fitter = cf.CorrFitter(
    models=make_models(),
    fitter='gsl_multifit', alg='subspace2D', solver='cholesky'
    )
```

uses the `subspace2D` algorithm for subsequent fits with `fitter`. It is also possible to reset the default algorithms for all fits:

```
import lsqfit

lsqfit.nonlinear_fit.set(
    fitter='gsl_multifit', alg='subspace2D', solver='cholesky'
    )
```

The documentation for `lsqfit` describes many more options.

1.8 Faster Fits — Processed Datasets

When fitting very large data sets, it is usually worthwhile to pare the data down to the smallest subset that is needed for the fit. Ideally this is done before the Monte Carlo data are averaged, to keep the size of the covariance matrix down. One way to do this is to process the Monte Carlo data with the models, just before averaging it, by using

```
import gvar as gv
import corrfitter as cf

def make_pdata(filename, models):
    dset = cf.read_dataset(filename)
    return cf.process_dataset(dset, models)
```

in place of `make_data(filename)`. Here `models` is the list of models used by the fitter (`fitter.models`). Function `make_pdata` returns processed data which is passed to `fitter.lsqfit` using the `pdata` keyword:

```
import corrfitter as cf

def main():
    N = 4
    models = make_models()
    pdata = make_pdata('mcfile', models)
    prior = make_prior(N)
    fitter = cf.CorrFitter(models=models)
    fit = fitter.lsqfit(pdata=pdata, prior=prior)
    print(fit)
    print_results(fit, prior, pdata)

...

if __name__ == '__main__':
    main()
```

Processed data can only be used with the models that created it, so parameters in those models should not be changed after the data is processed.

1.9 Variations

A 2-point correlator is turned into a periodic function of `t` by specifying the period through parameter `tp`. Doing so causes the replacement (for `tp>0`)

```
exp(-E[i]*t)    ->    exp(-E[i]*t) + exp(-E[i]*(tp-t))
```

in the fit function. If `tp` is negative, the function is replaced by an anti-periodic function with period `abs(tp)` and (for `tp<0`):

```
exp(-E[i]*t)   ->   exp(-E[i]*t) - exp(-E[i]*(abs(tp)-t))
```

Also (or alternatively) oscillating terms can be added to the fit by modifying parameter `s` and by specifying sources, sinks and energies for the oscillating pieces. For example, one might want to replace the sum of exponentials with two sums

```
sum_i a[i]**2 * exp(-E[i]*t) - sum_i ao[i]**2 (-1)**t * exp(-Eo[i]*t)
```

in a (nonperiodic) fit function. Then an appropriate model would be, for example,

```
Corr2(
    datatag='Gaa', tmin=2, tmax=63,
    a=('a','ao'), b=('a','ao'), dE=('dE','dEo'), s=(1,-1)
    )
```

where `ao` and `dEo` refer to additional fit parameters describing the oscillating component. In general parameters for amplitudes and energies can be tuples with two components: the first describing normal states, and the second describing oscillating states. To omit one or the other, put `None` in place of a label. Parameter `s[0]` is an overall factor multiplying the non-oscillating terms, and `s[1]` is the corresponding factor for the oscillating terms.

Highly correlated data can lead to problems from numerical roundoff errors, particularly where the fit code inverts the covariance matrix when constructing the `chi**2` function. Such problems show up as unexpectedly large `chi**2` or fits that stall and appear never to converge. Such situations are usually improved by introducing an SVD cut: for example,

```
fit = fitter.lsqfit(data=data, prior=prior, p0=p0, svdcut=1e-4)
```

Introducing an SVD cut increases the effective errors and so is a conservative move. For more information about SVD cuts see the `lsqfit` tutorial and documentation. Parameter `svdcut` is used to specify an SVD cut.

1.10 Very Fast (But Limited) Fits

At large `t`, two-point correlators are dominated by the term with the smallest `E`, and often it is only the parameters in that leading term that are needed. In such cases there is a very fast analysis that is often almost as accurate as a full fit. Assuming a non-periodic correlator, for example, we want to calculate energy `E[0]` and amplitude `A[0]` where:

```
G(t) = sum_i=0,N-1 A[i] * exp(-E[i]*t)
```

This is done using the following code

```
from corrfitter import fastfit

# Gdata = array containing G(t) for t=0,1,2...
fit = fastfit(Gdata, ampl='0(1)', dE='0.5(5)', tmin=3)
print('E[0] =', fit.E)                    # E[0]
print('A[0] =', fit.ampl)                 # A[0]
print('chi2/dof =', fit.E.chi2/fit.dof) # good fit if of order 1 or less
print('Q =', fit.E.Q)                     # good fit if Q > 0.05-0.1
```

where `G` is an array containing a two-point correlator, `ampl` is a prior for the amplitudes `A[i]`, `dE` is a prior for energy differences `E[i]-E[i-1]`, and `tmin` is the minimum time used in the analysis.

`fastfit` is fast because it does not attempt to determine any parameters in `G(t)` other than `E[0]` and `A[0]`. It does this by using the priors for the amplitudes and energy differences to remove (*marginalize*) all terms from the correlator other than the `E[0]` term: so the data `Gdata(t)` for the correlator are replaced by

```
Gdata(t) - sum_i=1..N-1  A[i] * exp(-E[i]*t)
```

where `A[i]` and `E[i]` for `i>0` are replaced by priors given by `ampl` and `(i+1) * dE`, respectively. The modified correlator is then fit by a single term, `A[0] * exp(-E[0]*t)`, which means that a fit is not actually necessary since the functional form is so simple. *`fastfit`* averages estimates for `E[0]` and `A[0]` from all `ts` larger than `tmin`. It is important to verify that these estimates agree with each other, by checking the `chi**2` of the average. Try increasing `tmin` if the `chi**2` is too large; or introduce an SVD cut.

The energies from *`fastfit`* are closely related to standard *effective masses*. The key difference is *`fastfit`*'s marginalization of terms from excited states (`i>0` above). This allows *`fastfit`* to use information from much smaller `ts` than otherwise, increasing precision. It also quantifies the uncertainty caused by the existence of excited states, and gives a simple criterion for how small `tmin` can be (the `chi**2`). Results are typically as accurate as results obtained from a full multi-exponential fit that uses the same priors for `A[i]` and `E[i]`, and the same `tmin`. *`fastfit`* can also be used for periodic and anti-periodic correlators, as well as for correlators that contain terms that oscillate in sign from one `t` to the next.

`fastfit` is a special case of the more general marginalization strategy discussed in *Faster Fits*, above.

1.11 3-Point Correlators

Correlators `Gavb(t,T) = <b(T) V(t) a(0)>` can also be included in fits as functions of `t`. In the illustration above, for example, we might consider additional Monte Carlo data describing a form factor with the same intermediate states before and after `V(t)`. Assuming the data is tagged by `aVbT15` and describes `T=15`, the corresponding entry in the collection of models might then be:

```
Corr3(datatag='aVbT15', T=15, tdata=range(16), tfit=range(1, 16),
    Vnn='Vnn',                  # parameters for V
    a='a', dEa='dE',            # parameters for a->V
    b='b', dEb='dE',            # parameters for V->b
    )
```

This models the Monte Carlo data for the 3-point function using the following formula:

```
sum_i,j a[i] * exp(-Ea[i]*t) * Vnn[i,j] * b[j] * exp(-Eb[j]*t)
```

where the `Vnn[i,j]`s are new fit parameters related to `a->V->b` form factors. Obviously multiple values of `T` can be studied by including multiple *`corrfitter.Corr3`* models, one for each value of `T`. Either or both of the initial and final states can have oscillating components (include `sa` and/or `sb`). If there are oscillating states then additional `Vs` must be specified: `Vno` connecting a normal state to an oscillating state, `Von` connecting oscillating to normal states, and `Voo` connecting oscillating to oscillating states.

Keywords `tdata` and `tfit` need not be specified when there is data for every `t=0,1...T`: for example,

```
Corr3(
    datatag='aVbT15', T=15, tmin=1,
    Vnn='Vnn', a='a', dEa='dE', b='b', dEb='dE',
    )
```

is equivalent to the definition above.

There are two cases that require special treatment. One is when simultaneous fits are made to `a->V->b` and `b->V->a`. Then the `Vnn`, `Vno`, *etc.* for `b->V->a` are the (matrix) transposes of the the same matrices for `a->V->b`. In this case the models for the two would look something like:

```
models = [
    ...
    Corr3(
```

```
        datatag='aVbT15', T=15, tmin=1,
        Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
        a=('a','ao'), dEa=('dE','dEo'), sa=(1,-1), # a->V
        b=('b','bo'), dEb=('dE','dEo'), sb=(1,-1)  # V->b
        ),
    Corr3(
        datatag='bVaT15', T=15, tmin=1, reverse=True,
        Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
        a=('a','ao'), dEa=('dE','dEo'), sa=(1,-1), # a->V
        b=('b','bo'), dEb=('dE','dEo'), sb=(1,-1)  # V->b
        ),
    ...
    ]
```

The second `Corr3` is identical to the first except for the datatag (`'bVaT15'`), and the keyword `reverse=True`, which instructs the model to time-reverse its data, interchanging `t=0` with `t=T`, before fitting. Time-reversing in effect turns `b->V->a` into `a->V->b`.

Another way to handle this last situation is to average the data from `b->V->a` with that from `a->V->b` for a single fit. This is done using one `Corr3` but with the keyword `reverseddata` to indicate the data to be time-reversed and then averaged with the `a->V->b` data:

```
models = [
    ...
    Corr3(
        datatag='aVbT15', T=15, tmin=1, reverseddata='bVaT15',
        Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
        a=('a','ao'), dEa=('dE','dEo'), sa=(1,-1), # a->V
        b=('b','bo'), dEb=('dE','dEo'), sb=(1,-1)  # V->b
        ),
    ...
    ]
```

The second special case is for fits to `a->V->a` where the initial and final particles are the same (with the same momentum). In that case, `Vnn` and `Voo` are symmetric matrices, and `Von` is the transpose of `Vno`. The model for such a case would look like, for example:

```
Corr3(
    datatag='aVbT15', T=15, tmin=1,
    Vnn='Vnn', Vno='Vno', Voo='Voo', symmetric_V=True,
    a=('a','ao'), dEa=('dE', 'dEo'), sa=(1, -1), # a->V
    b=('a','ao'), dEb=('dE', 'dEo'), sb=(1, -1)  # V->a
    )
```

Here only `Vno` is specified, since `Von` is its transpose. Furthermore `Vnn` and `Voo` are (square) symmetric matrices when `symmetric_V==True` and so only the upper part of each matrix is needed. In this case `Vnn` and `Voo` are treated as one-dimensional arrays with `N(N+1)/2` elements corresponding to the upper parts of each matrix, where `N` is the number of exponentials (that is, the number of `a[i]`s).

1.12 Testing Fits with Simulated Data

Large fits are complicated and often involve nontrivial choices about algorithms (*e.g.*, chained fits versus regular fits), priors, and SVD cuts — choices that affect the values and errors for the fit parameters. In such situations it is often a good idea to test the fit protocol that has been selected. This can be done by fitting simulated data. Simulated data looks almost identical to the original fit data but has means that have been adjusted to correspond to fluctuations around a correlator with known (before the fit) parameter values: `p=pexact`. The *`corrfitter.CorrFitter`* iterator `simulated_pdata_iter` creates any number of different simulated data sets of this kind. Fitting any of these with a particular fit protocol tests the reliability of that protocol since the fit results should agree with `pexact` to within the (simulated) fit's errors. One or two fit simulations of this sort are usually enough to establish the validity of a protocol. It is also easy to compare the performance of different fit options by applying these in fits of simulated data, again because we know the correct answers (`pexact`) ahead of time.

Typically one obtains reasonable values for `pexact` from a fit to the real data. Assuming these have been dumped into a file named `"pexact_file"` (using, for example, `fit.dump_pmean("pexact_file")`), a testing script might look something like:

```
import gvar as gv
import lsqfit
import corrfitter

def main():
    dataset = gv.dataset.Dataset(...)        # from original fit code
    prior = make_prior(...)
    fitter = corrfitter.CorrFitter(models = make_models(...))
    n = 2                                    # number of simulations
    pexact = lsqfit.nonlinear_fit.load_parameters("pexact_file")
    for spdata in fitter.simulated_pdata_iter(n, dataset, pexact=pexact):
        # sfit = fit to the simulated data sdata
        sfit = fitter.lsqfit(pdata=spdata, p0=pexact, prior=prior...)
        ... check that sfit.p values agree with pexact to within sfit.psdev ...
```

Fit simulations are particularly useful for setting SVD cuts. Given a set of approximate parameter values to use for `pexact`, it is easy to run fits with a range of SVD cuts to see how small `svdcut` can be made before the parameters of interest deviate too far from `pexact`.

1.13 Bootstrap Analyses

A *bootstrap analysis* gives more robust error estimates for fit parameters and functions of fit parameters than the conventional fit when errors are large, or fluctuations are non-Gaussian. A typical code looks something like:

```
import gvar as gv
import gvar.dataset as ds
from corrfitter import CorrFitter
# fit
dset = ds.Dataset('mcfile')
data = ds.avg_data(dset)              # create fit data
fitter = Corrfitter(models=make_models())
N = 4                                 # number of terms in fit function
prior = make_prior(N)
fit = fitter.lsqfit(prior=prior, data=data)  # do standard fit
print 'Fit results:'
print 'a', fit.p['a']                 # fit results for 'a' amplitudes
print 'dE', fit.p['dE']               # fit results for 'dE' energies
...
...
# bootstrap analysis
print 'Bootstrap fit results:'
nbootstrap = 10                       # number of bootstrap iterations
bs_datalist = (ds.avg_data(d) for d in ds.bootstrap_iter(dset, nbootstrap))
bs = ds.Dataset()                     # bootstrap output stored in bs
for bs_fit in fitter.bootstrap_iter(bs_datalist): # bs_fit = lsqfit output
    p = bs_fit.pmean    # best fit values for current bootstrap iteration
    bs.append('a', p['a'])            # collect bootstrap results for a[i]
```

```
    bs.append('dE', p['dE'])         # collect results for dE[i]
    ...                              # include other functions of p
    ...
bs = ds.avg_data(bs, bstrap=True)    # medians + error estimate
print 'a', bs['a']                   # bootstrap result for 'a' amplitudes
print 'dE', bs['dE']                 # bootstrap result for 'dE' energies
....
```

This code first prints out the standard fit results for the `'a'` amplitudes and `'dE'` energies. It then makes `10` bootstrap copies of the original input data, and fits each using the best-fit parameters from the original fit as the starting point for the bootstrap fit. The variation in the best-fit parameters from fit to fit is an indication of the uncertainty in those parameters. This example uses a `gvar.dataset.Dataset` object `bs` to accumulate the results from each bootstrap fit, which are computed using the best-fit values of the parameters (ignoring their standard deviations). Other functions of the fit parameters could be included as well. At the end `avg_data(bs, bstrap=True)` finds median values for each quantity in `bs`, as well as a robust estimate of the uncertainty (to within 30% since `nbootstrap` is only `10`).

The list of bootstrap data sets `bs_datalist` can be omitted in this example in situations where the input data has high statistics. Then the bootstrap copies are generated internally by `fitter.bootstrap_iter()` from the means and covariance matrix of the input data (assuming Gaussian statistics).

1.14 Implementation

Background information on the some of the fitting strategies used by *`corrfitter.CorrFitter`* can be found by doing a web searches for “hep-lat/0110175”, “arXiv:1111.1363”, and ”:arXiv:1406.2279” (appendix). These are papers by G.P. Lepage and collaborators whose published versions are: G.P. Lepage et al, Nucl.Phys.Proc.Suppl. 106 (2002) 12-20; K. Hornbostel et al, Phys.Rev. D85 (2012) 031504; and C. Bouchard et al, Phys.Rev. D90 (2014) 054506.

1.15 Correlator Model Objects

Correlator objects describe theoretical models that are fit to correlator data by varying the models’ parameters.

A model object’s parameters are specified through priors for the fit. A model assigns labels to each of its parameters (or arrays of related parameters), and these labels are used to identify the corresponding parameters in the prior. Parameters can be shared by more than one model object.

A model object also specifies the data that it is to model. The data is identified by the data tag that labels it in the input file or `gvar.dataset.Dataset`.

class `corrfitter.`**`Corr2`**(*datatag*, *a*, *b*, *dE*, *s=1.0*, *tp=None*, *tmin=None*, *tmax=None*, *tdata=None*, *tfit=None*, *reverse=False*, *reverseddata=[]*, *otherdata=[]*)

Two-point correlators `Gab(t) = <b(t) a(0)>`.

`corrfitter.Corr2` models the `t` dependence of a 2-point correlator `Gab(t)` using

```
Gab(t) = sn * sum_i an[i] * bn[i] * fn(En[i], t)
       + so * sum_i ao[i] * bo[i] * fo(Eo[i], t)
```

where `sn` and `so` are typically `-1`, `0`, or `1` and

```
fn(E, t) =  exp(-E*t) + exp(-E*(tp-t)) # tp>0 -- periodic
       or   exp(-E*t) - exp(-E*(-tp-t))# tp<0 -- anti-periodic
       or   exp(-E*t)                  # if tp is None (nonperiodic)
```

```
fo(E, t) = (-1)**t * fn(E, t)
```

The fit parameters for the non-oscillating piece of `Gab` (first term) are `an[i]`, `bn[i]`, and `dEn[i]` where:

```
dEn[0] = En[0]
dEn[i] = En[i]-En[i-1] > 0     (for i>0)
```

and therefore `En[i] = sum_j=0..i dEn[j]`. The fit parameters for the oscillating piece are defined analogously: `ao[i]`, `bo[i]`, and `dEo[i]`.

The fit parameters are specified by the keys corresponding to these parameters in a dictionary of priors supplied to *corrfitter.CorrFitter*. The keys are strings and are also used to access fit results. A log-normal prior can be specified for a parameter by including an entry for `log(c)` in the prior, rather than for `c` itself. See the `lsqfit` documentation for information about other distributions that are available. Values for both `log(c)` and `c` are included in the parameter dictionary. Log-normal distributions are useful for forcing `an`, `bn` and/or `dE` to be positive.

When `tp is not None` and positive, the correlator is assumed to be symmetrical about `tp/2`, with `Gab(t)=Gab(tp-t)`. Data from `t>tp/2` is averaged with the corresponding data from `t<tp/2` before fitting. When `tp` is negative, the correlator is assumed to be anti-symetrical about `-tp/2`.

Parameters

- **datatag** (*str*) – Key used to access correlator data in the input data dictionary (see *corrfitter.CorrFitter*): `data[self.datatag]` is a (1-d) array containing the correlator values (`gvar.GVars`).
- **a** (*str or tuple*) – Key identifying the fit parameters for the source amplitudes `an` in the dictionary of priors provided to *corrfitter.CorrFitter*; or a two-tuple of keys for the source amplitudes `(an, ao)`. The corresponding values in the dictionary of priors are (1-d) arrays of prior values with one term for each `an[i]` or `ao[i]`. Replacing either key by `None` causes the corresponding term to be dropped from the fit function. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **b** (*str or tuple*) – Same as `self.a` but for the sinks `(bn, bo)` instead of the sources `(an, ao)`.
- **dE** (*str*) – Key identifying the fit parameters for the energy differences `dEn` in the dictionary of priors provided by *corrfitter.CorrFitter*; or a two-tuple of keys for the energy differences `(dEn, dEo)`. The corresponding values in the dictionary of priors are (1-d) arrays of prior values with one term for each `dEn[i]` or `dEo[i]`. Replacing either key by `None` causes the corresponding term to be dropped from the fit function. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **s** (*float or tuple*) – Overall factor `sn` for non-oscillating part of fit function, or two-tuple of overall factors `(sn, so)` for both pieces.
- **tdata** (*list of ints*) – The `ts` corresponding to data entries in the input data. Note that `len(self.tdata)` should equal `len(data[self.datatag])`. If `tdata` is omitted, `tdata=numpy.arange(tp)` is assumed, or `tdata=numpy.arange(tmax)` if `tp` is not specified.
- **tfit** (*list of ints*) – List of `ts` to use in the fit. Only data with these `ts` (all of which should be in `tdata`) is used in the fit. If `tfit` is omitted, it is assumed to be all `t` values from `tdata` that are larger than or equal to `tmin` (if specified) and smaller than or equal to `tmax` (if specified).
- **tp** (int or `None`) – If `tp` is positive, the correlator is assumed to be periodic with `Gab(t)=Gab(tp-t)`. If negative, the correlator is assumed to be anti-periodic with

`Gab(t)=-Gab(-tp-t)`. Setting `tp=None` implies that the correlator is not periodic, but rather continues to fall exponentially as `t` is increased indefinitely.

- **tmin** (int or `None`) – If `tfit` is omitted, it is assumed to be all `t` values from `tdata` that are larger than or equal to `tmin` and smaller than or equal to `tmax` (if specified). `tmin` is ignored if `tfit` is specified.
- **tmax** (int or `None`) – If `tfit` is omitted, it is assumed to be all `t` values from `tdata` that are larger than or equal to `tmin` (if specified) and smaller than or equal to `tmax`. `tmin` is ignored if `tfit` and `tdata` are specified.
- **ncg** (*int*) – Width of bins used to coarse-grain the correlator before fitting. Each bin of `ncg` correlator values is replaced by its average. Default is `ncg=1` (ie, no coarse-graining).
- **reverse** (*bool*) – If `True`, the data associated with `self.datatag` is time-reversed (`data -> [data[0], data[-1], data[-2]...data[1]]`). Ignored otherwise.
- **otherdata** (str or list or `None`) – Data tag or list of data tags for additional data that are averaged with the `self.datatag` data before fitting. This is useful including data from correlators with the source and sink interchanged. Default is `None`.
- **reverseddata** (str or list or `None`) – Data tag or list of data tags for data that is time-reversed and then averaged with the `self.datatag` data before fitting. Default is `None`.

builddata(*data*)
Assemble fit data from dictionary `data`.

builddataset(*dataset*)
Assemble fit data from data set dictionary `dataset`.

buildprior(*prior*, *nterm=None*, *mopt=None*, *extend=None*)
Create fit prior by extracting relevant pieces from `prior`.

This routine selects the entries in dictionary `prior` corresponding to the model's fit parameters. If `nterm` is not `None`, it also adjusts the number of terms that are retained.

Parameters

- **prior** (*dictionary*) – Dictionary containing priors for fit parameters.
- **nterm** (`None` or int or two-tuple) – Setting `nterm=(n,no)` restricts the number of terms to `n` in the non-oscillating part and `no` in the oscillating part of the fit function. Replacing either or both by `None` keeps all terms, as does setting `nterm=None`. This optional argument is used to implement marginalization.

fitfcn(*p*, *t=None*)
Return fit function for parameters `p`.

class `corrfitter.`**Corr3**(*datatag*, *T*, *Vnn*, *a*, *b*, *dEa*, *dEb*, *sa=1.0*, *sb=1.0*, *Vno=None*, *Von=None*, *Voo=None*, *tdata=None*, *tfit=None*, *tmin=None*. *reverse=False*, *symmetric_V=False*, *reverseddata=[]*, *otherdata=[]*)
Three-point correlators `Gavb(t, T) = <b(T) V(t) a(0)>`.

`corrfitter.Corr3` models the `t` dependence of a 3-point correlator `Gavb(t, T)` using

```
Gavb(t, T) =
 sum_i,j san * an[i] * fn(Ean[i],t) * Vnn[i,j] * sbn * bn[j] * fn(Ebn[j],T-t)
+sum_i,j san * an[i] * fn(Ean[i],t) * Vno[i,j] * sbo * bo[j] * fo(Ebo[j],T-t)
+sum_i,j sao * ao[i] * fo(Eao[i],t) * Von[i,j] * sbn * bn[j] * fn(Ebn[j],T-t)
+sum_i,j sao * ao[i] * fo(Eao[i],t) * Voo[i,j] * sbo * bo[j] * fo(Ebo[j],T-t)
```

where

```
fn(E, t) =  exp(-E*t)
fo(E, t) = (-1)**t * exp(-E*t)
```

The fit parameters for the non-oscillating piece of `Gavb` (first term) are `Vnn[i,j]`, `an[i]`, `bn[j]`, `dEan[i]` and `dEbn[j]` where, for example:

```
dEan[0] = Ean[0]
dEan[i] = Ean[i] - Ean[i-1] > 0     (for i>0)
```

and therefore `Ean[i] = sum_j=0..i dEan[j]`. The parameters for the other terms are similarly defined.

Parameters

- **datatag** (*str*) – Tag used to label correlator in the input data.
- **a** (*str or tuple*) – Key identifying the fit parameters for the source amplitudes `an`, for `a->V`, in the dictionary of priors provided to `corrfitter.CorrFitter`; or a two-tuple of keys for the source amplitudes `(an, ao)`. The corresponding values in the dictionary of priors are (1-d) arrays of prior values with one term for each `an[i]` or `ao[i]`. Replacing either key by `None` causes the corresponding term to be dropped from the fit function. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **b** (*str or tuple*) – Same as `self.a` but for the `V->b` sink amplitudes `(bn, bo)`.
- **dEa** (*str or tuple*) – Fit-parameter label for `a->V` intermediate-state energy differences `dEan`, or two-tuple of labels for the differences `(dEan, dEao)`. Each label represents an array of energy differences. Replacing either label by `None` causes the corresponding term in the correlator function to be dropped. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **dEb** (*str or tuple*) – Same as `self.dEa` but for `V->b` sink energies `(dEbn, dEbo)`.
- **sa** (*float or tuple*) – Overall factor `san` for the non-oscillating `a->V` terms in the correlator, or two-tuple containing the overall factors `(san, sao)` for the non-oscillating and oscillating terms. Default is `(1,-1)`.
- **sb** (*float or tuple*) – Same as `self.sa` but for `V->b` sink overall factors `(sbn, sbo)`.
- **Vnn** (str or `None`) – Fit-parameter label for the matrix of current matrix elements `Vnn[i,j]` connecting non-oscillating states. The matrix must be square and symmetric if `symmetric_V=True`, and only the elements `V[i,j]` for `j>=i` are specified, using a 1-d array `V_sym` with the following layout:

```
[V[0,0],V[0,1],V[0,2]...V[0,N],
        V[1,1],V[1,2]...V[1,N],
               V[2,2]...V[2,N],
                     .
                      .
                       .
                        V[N,N]]
```

  Note that `V[i,j] = V_symm[i*N + j - i * (i+1) / 2]` for `j>=i`. Set `Vnn=None` to omit it.

- **Vno** (str or `None`) – Fit-parameter label for the matrix of current matrix elements `Vno[i,j]` connecting non-oscillating to oscillating states. Only one of `Von` and `Vno` can be specified if `symmetric_V=True`; the other is defined to be its transform. Set `Vno=None` to omit it.

- **Von** (str or `None`) – Fit-parameter label for the matrix of current matrix elements `Vno[i,j]` connecting oscillating to non- oscillating states. Only one of `Von` and `Vno` can be specified if `symmetric_V=True`; the other is defined to be its transform. Set `Von=None` to omit it.
- **Voo** (str or `None`) – Fit-parameter label for the matrix of current matrix elements `Voo[i,j]` connecting oscillating states. The matrix must be square and symmetric if `symmetric_V=True`, and only the elements `V[i,j]` for `j>=i` are specified, using a 1-d array `V_sym` with the following layout:

```
[V[0,0],V[0,1],V[0,2]...V[0,N],
        V[1,1],V[1,2]...V[1,N],
               V[2,2]...V[2,N],
                     .
                      .
                       .
                        V[N,N]]
```

  Note that `V[i,j] = V_symm[i*N + j - i * (i+1) / 2]` for `j>=i`. Set `Voo=None` to omit it.

- **reverse** (*bool*) – If `True`, the data associated with `self.datatag` is time-reversed before fitting (interchanging `t=0` with `t=T`). This is useful for doing simultaneous fits to `a->V->b` and `b->V->a`, where one is time-reversed relative to the other: *e.g.*,

```
models = [ ...
    Corr3(
        datatag='a->V->b', tmin=3, T=15,
        a=('a', 'ao'), dEa=('dEa', 'dEao'),
        b=('b', 'bo'), dEb=('dEb', 'dEbo'),
        Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
        ),
    Corr3(
        datatag='b->V->a', tmin=3, T=15,
        a=('a', 'ao'), dEa=('dEa', 'dEao'),
        b=('b', 'bo'), dEb=('dEb', 'dEbo'),
        Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
        reverse=True,
        ),
    ...
    ]
```

  Another (faster) strategy for such situations is to average data from the second process with that from the first, before fitting, using keyword `reverseddata`. Default is `False`.

- **symmetric_V** (*bool*) – If `True`, the fit function for `a->V->b` is unchanged (symmetrical) under the the interchange of `a` and `b`. Then `Vnn` and `Voo` are square, symmetric matrices and their priors are one-dimensional arrays containing only elements `V[i,j]` with `j>=i`, as discussed above. Only one of `Von` and `Vno` can be specified if `symmetric_V=True`; the other is defined to be its transform.
- **T** (*int*) – Separation between source and sink.
- **tdata** (*list of ints*) – The `ts` corresponding to data entries in the input data. If omitted, is assumed equal to `numpy.arange(T + 1)`.
- **tfit** (*list of ints*) – List of `ts` to use in the fit. Only data with these `ts` (all of which should be in `tdata`) is used in the fit. If omitted, is assumed equal to `numpy.arange(tmin, T - tmin + 1)`.

- **tmin** (int or None) – If tfit is omitted, it is set equal to numpy.arange(tmin, T - tmin + 1). tmin is ignored if tfit is specified.
- **ncg** (*int*) – Width of bins used to coarse-grain the correlator before fitting. Each bin of ncg correlator values is replaced by its average. Default is ncg=1 (ie, no coarse-graining).
- **reverseddata** (str or list or None) – Data tag or list of data tags for additional data that are time-reversed and then averaged with the self.datatag data before fitting. This is useful for folding data from b->V->a into a fit for a->V->b: *e.g.*,

```
Corr3(
    datatag='a->V->b',
    a=('a', 'ao'), dEa=('dEa', 'dEao'),
    b=('b', 'bo'), dEb=('dEb', 'dEbo'),
    Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo',
    tmin=3, T=15, reverseddata='b->V->a'
    ),
```

  This is faster than using a separate model with transpose_V=True. Default is None.
- **otherdata** (str or list or None) – Data tag or list of data tags for additional data that are averaged with the self.datatag data before fitting. Default is None.

1.16 corrfitter.CorrFitter Objects

corrfitter.CorrFitter objects are wrappers for lsqfit.nonlinear_fit() which is used to fit a collection of models to a collection of Monte Carlo data.

class corrfitter.**CorrFitter**(*models*, *nterm=None*, *ratio=False*, *fast=True*, ***fitterargs*)
Nonlinear least-squares fitter for a collection of correlator models.

Parameters

- **models** – List of models, derived from lsqfit.MultiFitterModel, to be fit to the data. Individual models in the list can be replaced by lists of models or tuples of models; see below.
- **nterm** (*tuple or int or None*) – Number of terms fit in the non-oscillating part of fit functions; or a two-tuple of numbers indicating how many terms to fit in each of the non-oscillating and oscillating parts. Terms omitted from the fit are marginalized (*i.e.*, included as corrections to the fit data). If set to None, all parameters in the prior are fit, and none are marginalized.
- **ratio** (*bool*) – If True, implement marginalization using ratios: data_marg = data * fitfcn(prior_marg) / fitfcn(prior). If False (default), implement using differences: data_marg = data + (fitfcn(prior_marg) - fitfcn(prior)).
- **fast** (*bool*) – Setting fast=True (default) strips any variable not required by the fit from the prior. This speeds fits but loses information about correlations between variables in the fit and those that are not. The information can be restored using lsqfit.wavg after the fit.
- **fitterargs** – Additional arguments for the lsqfit.nonlinear_fit, such as tol, maxit, svdcut, fitter, etc., as needed.

bootstrap_fit_iter (*datalist=None*, *n=None*)
Iterator that creates bootstrap copies of a *corrfitter.CorrFitter* fit using bootstrap data from list data_list.

A bootstrap analysis is a robust technique for estimating means and standard deviations of arbitrary functions of the fit parameters. This method creates an interator that implements such an analysis of list (or iterator) datalist, which contains bootstrap copies of the original data set. Each data_list[i] is a different data input for self.lsqfit() (that is, a dictionary containing fit data). The iterator works its way through the data sets in data_list, fitting the next data set on each iteration and returning the resulting lsqfit.LSQFit fit object. Typical usage, for an *corrfitter.CorrFitter* object named fitter, would be:

```
for fit in fitter.bootstrap_iter(datalist):
    ... analyze fit parameters in fit.p ...
```

Parameters

- **data_list** (sequence or iterator or None) – Collection of bootstrap data sets for fitter. If None, the data_list is generated internally using the means and standard deviations of the fit data (assuming gaussian statistics).
- **n** (*integer*) – Maximum number of iterations if n is not None; otherwise there is no maximum.

Returns Iterator that returns a lsqfit.LSQFit object containing results from the fit to the next data set in data_list.

bootstrap_iter (*datalist=None*, *n=None*)
Iterator that creates bootstrap copies of a *corrfitter.CorrFitter* fit using bootstrap data from list data_list.

A bootstrap analysis is a robust technique for estimating means and standard deviations of arbitrary functions of the fit parameters. This method creates an interator that implements such an analysis of list (or iterator) datalist, which contains bootstrap copies of the original data set. Each data_list[i] is a different data input for self.lsqfit() (that is, a dictionary containing fit data). The iterator works its way through the data sets in data_list, fitting the next data set on each iteration and returning the resulting lsqfit.LSQFit fit object. Typical usage, for an *corrfitter.CorrFitter* object named fitter, would be:

```
for fit in fitter.bootstrap_iter(datalist):
    ... analyze fit parameters in fit.p ...
```

Parameters

- **data_list** (sequence or iterator or None) – Collection of bootstrap data sets for fitter. If None, the data_list is generated internally using the means and standard deviations of the fit data (assuming gaussian statistics).
- **n** (*integer*) – Maximum number of iterations if n is not None; otherwise there is no maximum.

Returns Iterator that returns a lsqfit.LSQFit object containing results from the fit to the next data set in data_list.

static **read_dataset** (*inputfiles*, *grep=None*, *keys=None*, *h5group='/'*, *binsize=1*, *tcol=0*, *Gcol=1*)
Read correlator Monte Carlo data from files into a gvar.dataset.Dataset.

Three files formats are supported by *read_dataset()*, depending upon inputfiles.

If `inputfiles` is a string ending in `'.h5'`, it is assumed to be the name of a file in hpf5 format. The file is opened as `h5file` and all hpf5 datasets in `h5file[h5group]` are collected into a dictionary and returned.

The second file format is the text-file format supported by `gvar.dataset.Dataset`: each line consists of a tag or key identifying a correlator followed by data corresponding to a single Monte Carlo measurement of the correlator. This format is assumed if `inputfiles` is a filename or a list of filenames. It allows a single file to contain an arbitrary number of measurements for an arbitrary number of different correlators. The data can also be spread over multiple files. A typical file might look like

```
# this is a comment; it is ignored
aa 1.237 0.912 0.471
bb 3.214 0.535 0.125
aa 1.035 0.851 0.426
bb 2.951 0.625 0.091
...
```

which describes two correlators, `aa` and `bb`, each having three different `t` values.

The third file format is assumed when `inputfiles` is a dictionary. The dictionary's keys and values identify the (one-dimensional) correlators and the files containing their Monte Carlo data, respectively. So the data for correlators `aa` and `bb` above are in separate files:

```
fileinputs = dict(aa='aafile', bb='bbfile')
```

Each line in these data files consists of an index `t` value followed by the corresponding value for correlator `G(t)`. The `ts` increase from line to line up to their maximum value, at which point they repeat. The `aafile` file for correlator `aa` above would look like:

```
# this is a comment; it is ignored
1 1.237
2 0.912
3 0.471
1 1.035
2 0.851
3 0.426
...
```

The columns in these files containing `t` and `G(t)` are assumed to be columns 0 and 1, respectively. These can be changed by setting arguments `tcol` and `Gcol`, respectively.

`corrfitter.process_dataset` supports keywords `binsize`, `grep` and `keys`. If `binsize` is greater than one, random samples are binned with bins of size `binsize`. If `grep` is not `None`, only keys that match or partially match regular expression `grep` are retained; others are ignored. If `keys` is not `None`, only keys that are in list `keys` are retained; others are discarded.

simulated_pdata_iter(*n*, *dataset*, *pexact=None*, *rescale=1.0*)
Create iterator that returns simulated fit pdata from `dataset`.

Simulated fit data has the same covariance matrix as `pdata=self.process_dataset(dataset)`, but mean values that fluctuate randomly, from copy to copy, around the value of the fitter's fit function evaluated at `p=pexact`. The fluctuations are generated from bootstrap copies of `dataset`.

The best-fit results from a fit to such simulated copies of `pdata` should agree with the numbers in `pexact` to within the errors specified by the fits (to the simulated data) — `pexact` gives the "correct" values for the parameters. Knowing the correct value for each fit parameter ahead of a fit allows us to test the reliability of the fit's error estimates and to explore the impact of various fit options (*e.g.*, `fitter.chained_fit` versus `fitter.lsqfit`, choice of SVD cuts, omission of select models, etc.)

Typically one need examine only a few simulated fits in order to evaluate fit reliability, since we know the correct values for the parameters ahead of time. Consequently this method is much faster than traditional bootstrap analyses.

`pexact` is usually taken from the last fit done by the fitter (`self.fit.pmean`) unless overridden in the function call. Typical usage is as follows:

```
dataset = gvar.dataset.Dataset(...)
data = gvar.dataset.avg_data(dataset)
...
fit = fitter.lsqfit(data=data, ...)
...
for spdata in fitter.simulated_pdata_iter(n=4, dataset):
    # redo fit 4 times with different simulated data each time
    # here pexact=fit.pmean is set implicitly
    sfit = fitter.lsqfit(pdata=spdata, ...)
    ... check that sfit.p (or functions of it) agrees ...
    ... with pexact=fit.pmean to within sfit.p's errors      ...
```

Parameters

- **`n`** (*int*) – Maximum number of simulated data sets made by iterator.
- **`dataset`** (*dictionary*) – Dataset containing Monte Carlo copies of the correlators. `dataset[datatag]` is a two-dimensional array for the correlator corresponding to `datatag`, where the first index labels the Monte Carlo copy and the second index labels time.
- **`pexact`** (dictionary or `None`) – Correct parameter values for fits to the simulated data — fit results should agree with `pexact` to within errors. If `None`, uses `self.fit.pmean` from the last fit.
- **`rescale`** (*float*) – Rescale errors in simulated data by `rescale` (*i.e.*, multiply covariance matrix by `rescale ** 2`). Default is one, which implies no rescaling.

1.17 `corrfitter.EigenBasis` Objects

`corrfitter.EigenBasis` objects are useful for analyzing two-point and three-point correlators with multiplle sources and sinks. The current interface for *`EigenBasis`* is experimental. It may change in the near future, as experience accumulates from its use.

class `corrfitter.`**`EigenBasis`**(*data*, *srcs*, *t*, *keyfmt='{s1}.{s2}'*, *tdata=None*)

Eigen-basis of correlator sources/sinks.

Given N sources/sinks and the $N \times N$ matrix $G_{ij}(t)$ of 2-point correlators created from every combination of source and sink, we can define a new basis of sources that makes the matrix correlator approximately diagonal. Each source in the new basis is associated with an eigenvector $v^{(a)}$ defined by the matrix equation

$$G(t_1)v^{(a)} = \lambda^{(a)}(t_1 - t_0)G(t_0)v^{(a)},$$

for some t_0, t_1. As t_0, t_1 increase, fewer and fewer states couple to $G(t)$. In the limit where only N states couple, the correlator

$$\overline{G}_{ab}(t) \equiv v^{(a)T}G(t)v^{(b)}$$

becomes diagonal, and each diagonal element couples to only a single state.

In practice, this condition is only approximate: that is, $\overline{G}(t)$ is approximately diagonal, with diagonal elements that overlap strongly with the lowest lying states, but somewhat with other states. These new sources are nevertheless useful for fits because there is an obvious prior for their amplitudes: `prior[a][b]` approximately equal to one when `b==a`, approximately zero when `b!=a` and `b<N`, and order one otherwise.

Such a prior can significantly enhance the stability of a multi-source fit, making it easier to extract reliable results for excited states. It encodes the fact that only a small number of states couple strongly to $G(t)$ by time t_0, without being overly prescriptive about what their energies are. We can easilty project our correlator onto the new eigen-basis (using `EigenBasis.apply()`) in order to use this prior, but this is unnecessary. `EigenBasis.make_prior()` creates a prior of this type in the eigen-basis and then transforms it back to the original basis, thereby creating an equivalent prior for the amplitudes corresponding to the original sources.

Typical usage is straightforward. For example,

```
basis = EigenBasis(
    data,                           # data dictionary
    keyfmt='G.{s1}.{s2}',           # key format for dictionary entries
    srcs=['local', 'smeared'],      # names of sources/sinks
    t=(5, 7),                       # t0, t1 used for diagonalization
    )
prior = basis.make_prior(nterm=4, keyfmt='m.{s1}')
```

creates an *eigen-prior* that is optimized for fitting the 2-by-2 matrix correlator given by

```
[[data['G.local.local'],     data['G.local.smeared']]
 [data['G.smeared.local'],   data['G.smeared.smeared']]]
```

where `data` is a dictionary containing all the correlators. Parameter `t` specifies the times used for the diagonalization: $t_0 = 5$ and $t_1 = 7$. Parameter `nterm` specifies the number of terms in the fit. `basis.make_prior(...)` creates priors `prior['m.local']` and `prior['m.smeared']` for the amplitudes corresponding to the local and smeared source, and a prior `prior[log(m.dE)]` for the logarithm of the energy differences between successive levels.

The amplitudes `prior['m.local']` and `prior['m.smeared']` are complicated, with strong correlations between local and smeared entries for the same state. Projecting the prior unto the eigen-basis, however, reveals its underlying structure:

```
p_eig = basis.apply(prior)
```

implies

```
p_eig['m.0'] = [1.0(3), 0.0(1), 0(1), 0(1)]
p_eig['m.1'] = [0.0(1), 1.0(3), 0(1), 0(1)]
```

where the different entries are now uncorrelated. This structure registers our expectation that the `'m.0'` source in the eigen-basis overlaps strongly with the ground state, but almost not at all with the first excited state; and vice versa for the `'m.1'` source. Amplitude `p_eig` is noncommittal about higher states. This structure is built into `prior['m.local']` and `prior['smeared']`.

It is easy to check that fit results are consistent with the underlying prior. This can be done by projecting the best-fit parameters unto the eigen-basis using `p_eig = basis.apply(fit.p)`. Alternatively, a table listing the amplitudes in the new eigen-basis, together with the energies, is printed by:

```
print(basis.tabulate(fit.p, keyfmt='m.{s1}', eig_srcs=True))
```

The prior can be adjusted, if needed, using the `dEfac`, `ampl`, and `states` arguments in `EigenBasis.make_prior()`.

`EigenBasis.tabulate()` is also useful for printing the amplitudes for the original sources:

```
print(basis.tabulate(fit.p, keyfmt='m.{s1}'))
```

corrfitter.EigenBasis requires the scipy library in Python.

The parameters for creating an eigen-basis are:

Parameters

- **data** – Dictionary containing the matrix correlator using the original basis of sources and sinks.
- **keyfmt** – Format string used to generate the keys in dictionary `data` corresponding to different components of the matrix of correlators. The key for G_{ij} is assumed to be `keyfmt.format(s1=i, s2=j)` where `i` and `j` are drawn from the list of sources, `srcs`.
- **srcs** – List of source names used with `keyfmt` to create the keys for finding correlator components G_{ij} in the data dictionary.
- **t** – `t=(t0, t1)` specifies the `t` values used to diagonalize the correlation function. Larger `t` values are better than smaller ones, but only if the statistics are adequate. When fitting staggered-quark correlators, with oscillating components, choose `t` values where the oscillating pieces are positive (typically odd `t`). If only one `t` is given, `t=t0`, then `t1=t0+2` is used with it. Fits that use *corrfitter.EigenBasis* typically depend only weakly on the choice of `t`.
- **tdata** – Array containing the times for which there is correlator data. `tdata` is set equal to `numpy.arange(len(G_ij))` if it is not specified (or equals `None`).

The interface for *EigenBasis* is experimental. It may change in the near future, as experience accumulates from its use.

In addition to `keyfmt`, `srcs`, `t` and `tdata` above, the main attributes are:

E
Array of approximate energies obtained from the eigenanalysis.

eig_srcs
List of labels for the sources in the eigen-basis: `'0'`, `'1'` ...

svdcorrection
The sum of the SVD corrections added to the data by the last call to *EigenBasis.svd()*.

svdn
The number of degrees of freedom modified by the SVD correction in the last call to *EigenBasis.svd()*.

v
`v[a]` is the eigenvector corresponding to source `a` in the new basis, where `a=0,1....`

v_inv
`v_inv[i]` is the inverse-eigenvector for transforming from the new basis back to the original basis.

The main methods are:

apply(*data*, *keyfmt='{s1}'*)
Transform `data` to the eigen-basis.

The data to be transformed is `data[k]` where key `k` equals `keyfmt.format(s1=s1)` for vector data, or `keyfmt.format(s1=s1, s2=s2)` for matrix data with sources `s1` and `s2` drawn from `self.srcs`. A dictionary containing the transformed data is returned using the same keys but with the sources replaced by `'0', '1' ...` (from `basis.eig_srcs`).

If `keyfmt` is an array of formats, the transformation is applied for each format and a dictionary containing all of the results is returned. This is useful when the same sources and sinks are used for different types of correlators (e.g., in both two-point and three-point correlators).

make_prior(*nterm*, *keyfmt='{s1}'*, *dEfac='1(1)'*, *ampl=('1.0(3)', '0.03(10)', '0.2(1.0)')*, *states=None*, *eig_srcs=False*)

Create prior from eigen-basis.

Parameters

- **keyfmt** – Format string usded to generate keys for amplitudes and energies in the prior (a dictionary): keys are obtained from `keyfmt.format(s1=a)` where `a` is one of the original sources, `self.srcs`, if `eig_srcs=False` (default), or one of the eigen-sources, `self.eig_srcs`, if `eig_srcs=True`. The key for the energy differences is generated by `'log({})'.format(keyfmt.format(s1='dE'))`. The default is `keyfmt={s1}`.
- **dEfac** (string or `gvar.GVar`) – A string or `gvar.GVar` from which the priors for energy differences `dE[i]` are constructed. The mean value for `dE[0]` is set equal to the lowest energy obtained from the diagonalization. The mean values for the other `dE[i]`s are set equal to the difference between the lowest two energies from the diagonalization (or to the lowest energy if there is only one). These central values are then multiplied by `gvar.gvar(dEfac)`. The default value, *1(1)*, sets the width equal to the mean value. The prior is the logarithm of the resulting values.
- **ampl** – A 3-tuple of strings or `gvar.GVar`s from which priors are contructed for amplitudes corresponding to the eigen-sources. `gvar.gvar(ampl[0])` is used for for source components where the overlap with a particular state is expected to be large; `1.0(3)` is the default value. `gvar.gvar(ampl[1])` is used for states that are expected to have little overlap with the source; `0.03(10)` is the default value. `gvar.gvar(ampl[2])` is used where there is nothing known about the overlap of a state with the source; `0(1)` is the default value.
- **states** – A list of the states in the correlator corresponding to successive eigen-sources, where `states[i]` is the state corresponding to `i`-th source. The correspondence between sources and states is strong for the first sources, but can decay for subsequent sources, depending upon the quality of the data being used and the `t` values used in the diagonalization. In such situations one might specify fewer states than there are sources by making the length of `states` smaller than the number of sources. Setting `states=[]` assigns broad priors to the every component of every source. Parameter `states` can also be used to deal with situations where the order of later sources is not aligned with that of the actual states: for example, `states=[0,1,3]` connects the eigen-sources with the first, second and fourth states in the correlator. The default value, `states=[0, 1 ... N-1]` where `N` is the number of sources, assumes that sources and states are aligned.

svd(*data*, *keyfmt=None*, *svdcut=1e-15*)

Apply SVD cut to data in the eigen-basis.

The SVD cut is applied to `data[k]` where key `k` equals `keyfmt.format(s1=s1)` for vector data, or `keyfmt.format(s1=s1, s2=s2)` for matrix data with sources `s1` and `s2` drawn from `self.srcs`. The data are transformed to the eigen-basis of sources/sinks before the cut is applied and then transformed back to the original basis of sources. Results are returned in a dictionary containing the modified correlators.

If `keyfmt` is a list of formats, the SVD cut is applied to the collection of data formed from each format. The defaul value for `keyfmt` is `self.keyfmt`.

tabulate(*p*, *keyfmt='{s1}'*, *nterm=None*, *nsrcs=None*, *eig_srcs=False*, *indent=' '*)

Create table containing energies and amplitudes for `nterm` states.

Given a correlator-fit result `fit` and a corresponding *EigenBasis* object `basis`, a table listing the energies and amplitudes for the first `N` states in correlators can be printed using

```
print basis.tabulate(fit.p)
```

where `N` is the number of sources and `basis` is an *EigenBasis* object. The amplitudes are tabulated for the original sources unless parameter `eig_srcs=True`, in which case the amplitudes are projected onto the the eigen-basis defined by `basis`.

Parameters

- **p** – Dictionary containing parameters values.
- **keyfmt** – Parameters are `p[k]` where keys `k` are obtained from `keyfmt.format(s1=s)` where `s` is one of the original sources (`basis.srcs`) or one of the eigen-sources (`basis.eig_srcs`). The default definition is `'{s1}'`.
- **nterm** – The number of states from the fit tabulated. The default sets `nterm` equal to the number of sources in the basis.
- **nsrcs** – The number of sources tabulated. The default causes all sources to be tabulated.
- **eig_srcs** – Amplitudes for the eigen-sources are tabulated if `eigen_srcs=True`; otherwise amplitudes for the original basis of sources are tabulated (default).
- **indent** – A string prepended to each line of the table. Default is `4 * ' '`.

unapply (*data*, *keyfmt='{s1}'*)

Transform `data` from the eigen-basis to the original basis.

The data to be transformed is `data[k]` where key `k` equals `keyfmt.format(s1=s1)` for vector data, or `keyfmt.format(s1=s1, s2=s2)` for matrix data with sources `s1` and `s2` drawn from `self.eig_srcs`. A dictionary containing the transformed data is returned using the same keys but with the original sources (from `self.srcs`).

If `keyfmt` is an array of formats, the transformation is applied for each format and a dictionary containing all of the results is returned. This is useful when the same sources and sinks are used for different types of correlators (e.g., in both two-point and three-point correlators).

1.18 Fast Fit Objects

class corrfitter.**fastfit** (*G*, *ampl='0(1)'*, *dE='1(1)'*, *E=None*, *s=(1, -1)*, *tp=None*, *tmin=6*, *svdcut=1e-06*, *osc=False*, *nterm=10*)

Fast fit of a two-point correlator.

This function class estimates `E=En[0]` and `ampl=an[0]*bn[0]` for a two-point correlator modeled by

```
Gab(t) = sn * sum_i an[i]*bn[i] * fn(En[i], t)
       + so * sum_i ao[i]*bo[i] * fo(Eo[i], t)
```

where `(sn, so)` is typically `(1, -1)` and

```
fn(E, t) =  exp(-E*t) + exp(-E*(tp-t)) # tp>0 -- periodic
       or   exp(-E*t) - exp(-E*(-tp-t))# tp<0 -- anti-periodic
       or   exp(-E*t)                   # if tp is None (nonperiodic)

fo(E, t) = (-1)**t * fn(E, t)
```

Prior estimates for the amplitudes and energies of excited states are used to remove (that is, marginalize) their contributions to give a *corrected* correlator `Gc(t)` that includes uncertainties due to the terms removed. Estimates of `E` are given by:

```
Eeff(t) = arccosh(0.5 * (Gc(t+1) + Gc(t-1)) / Gc(t)),
```

The final estimate is the weighted average `Eeff_avg` of the `Eeff(t)`s for different `t`s. Similarly, an estimate for the amplitude `ampl` is obtained from the weighted average of

```
Aeff(t) = Gc(t) / fn(Eeff_avg, t).
```

If `osc=True`, an estimate is returned for `Eo[0]` rather than `En[0]`, and `ao[0]*bo[0]` rather than `an[0]*bn[0]`. These estimates are reliable when `Eo[0]` is smaller than `En[0]` (and so dominates at large `t`), but probably not otherwise.

Examples

The following code examines a periodic correlator (period 64) at large times (`t >= tmin`), where estimates for excited states don't matter much:

```
>>> import corrfitter as cf
>>> print(G)
[0.305808(29) 0.079613(24) ... ]
>>> fit = cf.fastfit(G, tmin=24, tp=64)
>>> print('E =', fit.E, ' ampl =', fit.ampl)
E = 0.41618(13)  ampl = 0.047686(95)
```

Smaller `tmin` values can be used if (somewhat) realistic priors are provided for the amplitudes and energy gaps:

```
>>> fit = cf.fastfit(G, ampl='0(1)', dE='0.5(5)', tmin=3, tp=64)
>>> print('E =', fit.E, ' ampl =', fit.ampl)
E = 0.41624(11)  ampl = 0.047704(71)
```

The result here is roughly the same as from the larger `tmin`, but this would not be true for a correlator whose signal to noise ratio falls quickly with increasing time.

`corrfitter.fastfit` estimates the amplitude and energy at all times larger than `tmin` and then averages to get its final results. The chi-squared of the average (*e.g.*, `fit.E.chi2`) gives an indication of the consistency of the estimates from different times. The chi-squared per degree of freedom is printed out for both the energy and the amplitude using

```
>>> print(fit)
E: 0.41624(11) ampl: 0.047704(71) chi2/dof [dof]: 0.9 0.8 [57] Q: 0.8 0.9
```

Large values for `chi2/dof` indicate an unreliable results. In such cases the priors should be adjusted, and/or `tmin` increased, and/or an SVD cut introduced. The averages in the example above have good values for `chi2/dof`.

Parameters

- **G** – An array of `gvar.GVar`s containing the two-point correlator. `G[j]` is assumed to correspond to time `t=j`, where `j=0...`.
- **ampl** – A `gvar.GVar` or its string representation giving an estimate for the amplitudes of the ground state and the excited states. Use `ampl=(ampln, amplo)` when the correlator contains oscillating states; `ampln` is the estimate for non-oscillating states, and `amplo` for oscillating states; setting one or the other to `None` causes the corresponding terms to be dropped. Default value is `'0(1)'`.

- **dE** – A `gvar.GVar` or its string representation giving an estimate for the energy separation between successive states. This estimate is also used to provide an estimate for the lowest energy when parameter `E` is not specified. Use `dE=(dEn, dEo)` when the correlator contains oscillating states: `dEn` is the estimate for non-oscillating states, and `dEo` for oscillating states; setting one or the other to `None` causes the corresponding terms to be dropped. Default value is `'1(1)'`.
- **E** – A `gvar.GVar` or its string representation giving an estimate for the energy of the lowest-lying state. Use `E=(En, Eo)` when the correlator contains oscillating states: `En` is the estimate for the lowest non-oscillating state, and `Eo` for lowest oscillating state. Setting `E=None` causes `E` to be set equal to `dE`. Default value is `None`.
- **s** – A tuple containing overall factors `(sn, so)` multiplying contributions from the normal and oscillating states. Default is `(1,-1)`.
- **tp** (*int or None*) – When not `None`, the correlator is periodic with period `tp` when `tp>0`, or anti-periodic with period `-tp` when `tp<0`. Setting `tp=None` implies that the correlator is neither periodic nor anti-periodic. Default is `None`.
- **tmin** (*int*) – Only `G(t)` with `t >= tmin` are used. Default value is `6`.
- **svdcut** (*float or None*) – SVD cut used in the weighted average of results from different times. (See the *corrfitter.CorrFitter* documentation for a discussion of SVD cuts.) Default is `1e-6`.
- **osc** (*bool*) – Set `osc=True` if the lowest-lying state is an oscillating state. Default is `False`.

Note that specifying a single `gvar.GVar g` (as opposed to a tuple) for any of parameters `ampl`, `dE`, or `E` is equivalent to specifying the tuple `(g, None)` when `osc=False`, or the tuple `(None, g)` when `osc=True`. A similar rule applies to parameter `s`.

corrfitter.fastfit objects have the following attributes:

E
: Energy of the lowest-lying state (`gvar.GVar`).

ampl
: Amplitude of the lowest-lying state (`gvar.GVar`).

Both `E` and `ampl` are obtained by averaging results calculated for each time larger than `tmin`. These are averaged to produce a final result. The consistency among results from different times is measured by the chi-squared of the average. Each of `E` and `ampl` has the following extra attributes:

chi2
: chi-squared for the weighted average.

dof
: The effective number of degrees of freedom in the weighted average.

Q
: The probability that the chi-squared could have been larger, by chance, assuming that the data are all Gaussain and consistent with each other. Values smaller than 0.05 or 0.1 suggest inconsistency. (Also called the *p-factor*.)

An easy way to inspect these attributes is to print the fit object `fit` using `print(fit)`, which lists the values of the energy and amplitude, the `chi2/dof` for each of these, the number of degrees of freedom, and the `Q` for each.

CHAPTER

TWO

ANNOTATED EXAMPLE: TWO-POINT CORRELATOR

2.1 Introduction

The simplest use of *corrfitter* is calculating the amplitude and energy of the ground state in a single two-point correlator. Here we analyze an η_s propagator where the source and sink are the same.

The one slightly non-obvious aspect of this fit is its use of log-normal priors for the energy differences `dE` between successive states in the correlator. As discussed in *Faster Fits — Postive Parameters*, this choice imposes an order on the states in relation to the fit parameters by forcing all `dE` values to be positive. Any such restriction helps stabilize a fit, improving both efficiency and the final results.

Another design option that helps stabilize the fit is to do a series of fits, with increasing number `N` of states in the fit function, where the results from the `N-1` fit are used by the fitter as the starting point (`p0`) for the `N` fit. The initial fits are bad, but this procedure helps guide the fit parameters towards sensible values as the number of states increases. See *Faster Fits* for more discussion.

The source code (`etas.py`) and data file (`etas-Ds.data`) are included with the *corrfitter* distribution, in the `examples/` directory. The data are from the HPQCD collaboration.

2.2 Code

Following the template outlined in *Basic Fits*, the entire code is:

```
from __future__ import print_function   # makes this work for python2 and 3

import collections
import gvar as gv
import numpy as np
import corrfitter as cf

def main():
    data = make_data(filename='etas.data')
    fitter = cf.CorrFitter(models=make_models())
    p0 = None
    for N in [2, 3, 4]:
        print(30 * '=', 'nterm =', N)
        prior = make_prior(N)
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0)
        print(fit)
        p0 = fit.pmean
        print_results(fit)
    fastfit = cf.fastfit(G=data['etas'], ampl='0(1)', dE='0.5(5)', tmin=3, tp=64)
    print(fastfit)
```

```
def make_data(filename):
    """ Read data, compute averages/covariance matrix for G(t). """
    return gv.dataset.avg_data(cf.read_dataset(filename))

def make_models():
    """ Create corrfitter model for G(t). """
    return [cf.Corr2(datatag='etas', tp=64, tmin=5, a='a', b='a', dE='dE')]

def make_prior(N):
    """ Create prior for N-state fit. """
    prior = collections.OrderedDict()
    prior['a'] = gv.gvar(N * ['0(1)'])
    prior['log(dE)'] = gv.log(gv.gvar(N * ['0.5(5)']))
    return prior

def print_results(fit):
    p = fit.p
    E = np.cumsum(p['dE'])
    a = p['a']
    print('{:2}  {:15}  {:15}'.format('E', E[0], E[1]))
    print('{:2}  {:15}  {:15}\n'.format('a', a[0], a[1]))

if __name__ == '__main__':
    main()
```

Here the Monte Carlo data are read by `make_data('etas.data')` from file `etas.data`. This file contains 225 lines, each with 64 numbers, of the form:

```
etas    0.305044    0.0789607   0.0331313 ...
etas    0.306573    0.0802435   0.0340765 ...
...
```

Each line is a different Monte Carlo estimate of the η_s correlator for t=0...63. The mean values and covariance matrix are computed for the 64 elements of the correlator using `gvar.dataset.avg_data()`, and the result is stored in `data['etas']`, which is an array of Gaussian random variables (objects of type `gvar.GVar`).

A *`corrfitter.CorrFitter`* object, `fitter`, is created for a single two-point correlator from a list of models created by `make_models()`. There is only one model in the list because there is only one correlator. It is a `Corr2` object which specifies that: the key (`datatag`) for extracting the correlator from the data dictionary is `'etas'`; the propagator is periodic with period 64; each correlator contains data for t values ranging from 0 to 63; only values greater than or equal to 5 and less than 64-5 are fit; the source and sink amplitudes are the same and labeled by `'a'` in the prior; and the energy differences between successive states are labeled `'dE'` in the prior.

Fits are tried with `N` states in the fit function, where `N` varies from 2 to 5. Usually `N=2` is too small, resulting in a poor fit. Here we will find that results have converged by `N=3`.

A prior, containing *a priori* estimates for the fit parameters, is contructed for each `N` by `make_prior(N)`. The amplitude priors, `prior['a'][i]`, are assumed to be 0±1, while the differences between successive energies are taken to be, roughly, 0.5±0.5. These are broad priors, based upon preliminary fits of the data. We want to use log-normal statistics for the energy differences, to guarantee that they are positive (and the states ordered, in order of increasing energy), so we use `prior['logdE']` for the logarithms of the differences — instead of `prior['dE']` for the differences themselves — and take the logarithm of the prior.

The fit is done by `fitter.lsqfit()` and `print_results(fit)` prints results for the first two states after each fit (that is, for each `N`). Note how results from the fit to `N` terms is used as the starting point for the fit with `N+1` terms, via parameter `p0`. As mentioned above, this speeds up the larger fits and also helps to stabilize them.

2.3 Results

The output from this fit code is:

```
============================== nterm = 2
Least Square Fit:
  chi2/dof [dof] = 0.98 [28]    Q = 0.49    logGBF = 481.95

Parameters:
            a 0    0.21854 (15)      [  0.0 (1.0) ]
              1     0.2721 (46)      [  0.0 (1.0) ]
      log(dE) 0   -0.87637 (28)      [ -0.7 (1.0) ]
              1     -0.330 (12)      [ -0.7 (1.0) ]
---------------------------------------------------
           dE 0    0.41629 (11)      [  0.50 (50) ]
              1     0.7191 (83)      [  0.50 (50) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 21/0.0)

E   0.41629(11)      1.1354(83)
a   0.21854(15)      0.2721(46)

============================== nterm = 3
Least Square Fit:
  chi2/dof [dof] = 0.68 [28]    Q = 0.89    logGBF = 483.08

Parameters:
            a 0    0.21836 (18)      [  0.0 (1.0) ]
              1       0.15 (12)      [  0.0 (1.0) ]
              2      0.308 (51)      [  0.0 (1.0) ]
      log(dE) 0   -0.87660 (30)      [ -0.7 (1.0) ]
              1      -0.56 (28)      [ -0.7 (1.0) ]
              2      -0.92 (53)      [ -0.7 (1.0) ]
---------------------------------------------------
           dE 0    0.41620 (12)      [  0.50 (50) ]
              1       0.57 (16)      [  0.50 (50) ]
              2       0.40 (21)      [  0.50 (50) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 18/0.0)

E   0.41620(12)      0.99(16)
a   0.21836(18)      0.15(12)

============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.68 [28]    Q = 0.89    logGBF = 483.08

Parameters:
            a 0    0.21836 (18)      [  0.0 (1.0) ]
              1       0.15 (12)      [  0.0 (1.0) ]
              2      0.308 (51)      [  0.0 (1.0) ]
              3     2e-06 +- 1       [  0.0 (1.0) ]
      log(dE) 0   -0.87660 (30)      [ -0.7 (1.0) ]
              1      -0.56 (28)      [ -0.7 (1.0) ]
              2      -0.92 (53)      [ -0.7 (1.0) ]
              3      -0.7 (1.0)      [ -0.7 (1.0) ]
```

```
---------------------------------------------------
          dE 0    0.41620 (12)      [  0.50 (50) ]
             1       0.57 (16)      [  0.50 (50) ]
             2       0.40 (21)      [  0.50 (50) ]
             3       0.50 (50)      [  0.50 (50) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 36/0.1)

E   0.41620(12)      0.99(16)
a   0.21836(18)      0.15(12)

E: 0.41624(11) ampl: 0.047704(71) chi2/dof [dof]: 0.9 0.8 [57] Q: 0.8 0.9
```

These fits are very fast — a small fraction of a second each on a laptop. Fit results converge by `N=3` states. The amplitudes and energy differences for states above the first three are essentially identical to the prior values; the Monte Carlo data are not sufficiently accurate to add any new information about these levels. The fits for `N>=3` are excellent, with chi-square per degree of freedom (`chi2/dof`) of 0.68. There are only 28 degrees of freedom here because the fitter, taking advantage of the periodicity, folded the data about the midpoint in `t` and averaged, before fitting. The ground state energy and amplitude are determined to a part in 1,000 or better.

2.4 Correlated Data?

It is worth checking whether the initial Monte Carlo data has correlations from sample to sample, since such correlations lead to underestimated fit errors. One approach is verify that results are unchanged when the input data are binned. To bin the data we use

```python
def make_data(filename):
    """ Read data, compute averages/covariance matrix for G(t). """
    return gv.dataset.avg_data(cf.read_dataset(filename, binsize=2))
```

which averages successive samples (bins of 2). Binned data give the following results from the last iteration and summary:

```
============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.94 [28]    Q = 0.55    logGBF = 477.28

Parameters:
            a 0    0.21839 (17)      [  0.0 (1.0) ]
              1      0.175 (80)      [  0.0 (1.0) ]
              2       0.36 (16)      [  0.0 (1.0) ]
              3      2e-07 +- 1      [  0.0 (1.0) ]
      log(dE) 0   -0.87651 (29)      [ -0.7 (1.0) ]
              1      -0.52 (17)      [ -0.7 (1.0) ]
              2      -0.67 (67)      [ -0.7 (1.0) ]
              3      -0.7 (1.0)      [ -0.7 (1.0) ]
---------------------------------------------------
          dE 0    0.41623 (12)      [  0.50 (50) ]
             1       0.59 (10)      [  0.50 (50) ]
             2       0.51 (34)      [  0.50 (50) ]
             3       0.50 (50)      [  0.50 (50) ]

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 9/0.0)
```

```
E   0.41623(12)      1.01(10)
a   0.21839(17)      0.175(80)
```

These agree pretty well with the previous results, suggesting that correlations are not a problem.

Binning should have no significant effect on results if there are no correlations, provided the total number of samples after binning is sufficiently large (*e.g.*, more than 100—200). Strong correlations cause error estimates to grow with increased bin size (like the square root of `binsize`). Binning reduces correlations; data should be binned with increasing bin sizes until fit error estimates stop growing.

2.5 Fast Fit and Effective Mass

The last two lines in the `main()` function of the code illustrate the use of *corrfitter.fastfit* to get a very fast results for the lowest-energy state. As discussed in *Very Fast (But Limited) Fits*, *corrfitter.fastfit* provides an alternative to the multi-exponential fits discussed above when only the lowest-energy parameters are needed. The method used is similar to a traditional effective mass analysis except that estimates for contributions from excited states are generated from priors and removed from the correlator before determining the effective mass. This allows the code to use much smaller `t` values than in the traditional approach, thereby obtaining results that rival the multi-exponential fits.

In this example, *corrfitter.fastfit* is used to analyze the two-point correlator stored in array `data['etas']`. The amplitudes for different states are estimated to have size 0 ± 1, while the spacings between energies (and between the first state and 0) are estimated to be 0.5 ± 0.5. The code averages results form all `t` values down to `tmin=3`. Setting `tp=64` indicates that the correlator is periodic with period 64.

The last line of the output summarizes the results of the fast fit. The energy and amplitude are almost identical to what was obtained from the multi-exponential fits (note that `fastfit.ampl` is the same as `fit.a[0]**2`, which has value 0.047681(79)). *corrfitter.fastfit* estimates the energy and amplitude for each `t` greater than `tmin`, and then averages the results. The consistency of results from different `t`s is measured by the chi-squared of the averages. The chi-squared per degree of freedom is reported here to be 0.8 for the `E` average and 0.9 for the `ampl` average, indicating that there is good agreement between different `t`s.

While a fast fit is easier to set up, multi-exponential fits are usually more robust, and provide more detailed information about the fit. One use for fast fits is to estimate the sizes of parameters for use in designing the priors for a multi-exponential fit. There are often situations where *a priori* knowledge about fit parameters is sketchy, especially for amplitudes. A fast fit to data at large `t` can quickly generate estimates for both amplitudes and energies, from which it is then easy to construct priors. In the code above, for example, we could replace `make_prior(N)` by `alt_make_prior(N, data['etas'])` where:

```
def alt_make_prior(N, G):
    fastfit = cf.fastfit(G=G, tmin=24, tp=64)
    da = 2 * fastfit.ampl.mean ** 0.5
    dE = 2 * fastfit.E.mean
    prior = collections.OrderedDict()
    prior['a'] = gv.gvar([gv.gvar(0, da) for i in range(N)])
    prior['log(dE)'] = gv.log(gv.gvar([gv.gvar(dE, dE) for i in range(N)]))
    return prior
```

This code does a fast fit using data from very large `t`, where priors for the excited states are unimportant. It then uses the results to create priors for the amplitudes and energy differences for all states, assuming that the ground state values are either larger, or smaller by no more than roughly a factor of two. This customized prior gives results that are almost identical to what was obtained using the original prior, above (in part because the original prior is pretty sensible to begin with).

Designing a prior using *corrfitter.fastfit* would be even more useful when multiple sources and sinks are involved, as in a matrix fit.

CHAPTER

THREE

ANNOTATED EXAMPLE: TRANSITION FORM FACTOR AND MIXING

3.1 Introduction

Here we describe a complete Python code that uses *corrfitter* to calculate the transition matrix element or form factor from an η_s meson to a D_s meson, together with the masses and amplitudes of these mesons. A very similar code, for (speculative) D_s-D_s mixing, is described at the end.

The form factor example combines data from two-point correlators, for the amplitudes and energies, with data from three-point correlators, for the transition matrix element. We fit all of the correlators together, in a single fit, in order to capture correlations between the various output parameters. The correlations are built into the output parameters and consequently are reflected in any arithmetic combination of parameters — no bootstrap is needed to calculate correlations or their impact on quantities derived from the fit parameters. The best-fit parameters (in `fit.p`) are objects of type `gvar.GVar`.

Staggered quarks are used in this simulation, so the D_s has oscillating components as well as normal components in its correlators.

The source codes (`etas-Ds.py`, `Ds-Ds.py`) and data files (`etas-Ds.h5`, `Ds-Ds.h5`) are included with the *corrfitter* distribution, in the `examples/` directory. The data are from the HPQCD collaboration.

3.2 Code

The `main` method for the form-factor code follows the pattern described in *Basic Fits*:

```
from __future__ import print_function   # makes this work for python2 and 3

import collections
import sys
import h5py
import gvar as gv
import numpy as np
import corrfitter as cf

SHOWPLOTS = True

def main():
    data = make_data('etas-Ds.h5')
    fitter = cf.CorrFitter(models=make_models(), svdcut=1e-5)
    p0 = None
    for N in [1, 2, 3, 4]:
        print(30 * '=', 'nterm =', N)
        prior = make_prior(N)
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0)
```

```
        print(fit.format(pstyle=None if N < 4 else 'm'))
        p0 = fit.pmean
    print_results(fit, prior, data)
    if SHOWPLOTS:
        fit.show_plots()
```

We include an SVD cut (`svdcut=1e-5`) to ameliorate roundoff errors in the highly correlated data. The Monte Carlo data are in a file named `'etas-Ds.h5'`. We are doing four fits, with 1, 2, 3, and 4 terms in the fit function. Each fit starts its minimization at point `p0`, which is set equal to the mean values of the best-fit parameters from the previous fit (`p0 = fit.pmean`). This reduces the number of iterations needed for convergence in the `N = 4` fit, for example, from 162 to 45. It also makes multi-term fits more stable.

After the fit, plots of the fit data divided by the fit are displayed by `fit.show_plots()`, provided `matplotlib` is installed. A plot is made for each correlator, and the ratios should equal one to within errors. To move from one plot to the next press "n" on the keyboard; to move to a previous plot press "p"; to quit the plots press "q".

We now look at each other major routine in turn.

3.2.1 a) make_data

Method `make_data('etas-Ds.h5')` reads in the Monte Carlo data, averages it, and formats it for use by *corrfitter.CorrFitter*:

```
def make_data(datafile):
    """ Read data from datafile and average it. """
    dset = cf.read_dataset(datafile)
    return gv.dataset.avg_data(dset)
```

The data file `etas-Ds.h5` is in hdf5 format. It contains four datasets:

```
>>> for v in dset.values():
...     print(v)
<HDF5 dataset "3ptT15": shape (225, 16), type "<f8">
<HDF5 dataset "3ptT16": shape (225, 17), type "<f8">
<HDF5 dataset "Ds": shape (225, 64), type "<f8">
<HDF5 dataset "etas": shape (225, 64), type "<f8">
```

Each corresponds to Monte Carlo data for a single correlator, which is packaged as a two-dimensional `numpy` array whose first index labels the Monte Carlo sample, and whose second index labels time. For example,

```
>>> print(dset['etas'][:, :])
[[ 0.305044   0.0789607  0.0331313 ...,  0.0164646  0.0332153  0.0791385]
 [ 0.306573   0.0802435  0.0340765 ...,  0.0170088  0.034013   0.0801528]
 [ 0.306194   0.0800234  0.0338007 ...,  0.0168862  0.0337728  0.0799462]
 ...,
 [ 0.305955   0.0797565  0.0335741 ...,  0.0167847  0.0336077  0.0796961]
 [ 0.305661   0.0793606  0.0333133 ...,  0.0165365  0.0333934  0.0792943]
 [ 0.305365   0.079379   0.033445  ...,  0.0164506  0.0332284  0.0792884]]
```

is data for a two-point correlator describing the η_s meson. Each of the 225 lines is a different Monte Carlo sample for the correlator, and has 64 entries corresponding to `t=0,1...63`. Note the periodicity in this data.

Function `gv.dataset.avg_data(dset)` averages over the Monte Carlo samples for all the correlators to compute their means and covariance matrix. The end result is a dictionary whose keys are the keys used to label the hdf5 datasets: for example,

```
>>> data = make_data('etas-Ds.h5')
>>> print(data['etas'])
```

```
[0.305808(29) 0.079613(24) 0.033539(17) ... 0.079621(24)]
>>> print(data['Ds'])
[0.2307150(73) 0.0446523(32) 0.0089923(15) ... 0.0446527(32)]
>>> print(data['3ptT16'])
[1.4583(21)e-10 3.3639(44)e-10 ... 0.000023155(30)]
```

Here each entry in `data` is an array of `gvar.GVars` representing Monte Carlo averages for the corresponding correlator at different times. This is the format needed by *corrfitter.CorrFitter*. Note that the different correlators are correlated with each other: for example,

```
>>> print(gv.evalcorr([data['etas'][0], data['Ds'][0]]))
[[ 1.          0.96432174]
 [ 0.96432174  1.        ]]
```

shows a 96% correlation between the `t=0` values in the η_s and D_s correlators.

3.2.2 b) make_models

Method `make_models()` specifies the theoretical models that will be used to fit the data:

```
def make_models():
    """ Create models to fit data. """
    tmin = 5
    tp = 64
    models = [
        cf.Corr2(
            datatag='etas', tp=tp,  tmin=tmin,
            a='etas:a',  b='etas:a',  dE='etas:dE',
            ),

        cf.Corr2(
            datatag='Ds', tp=tp,  tmin=tmin,
            a=('Ds:a', 'Dso:a'), b=('Ds:a', 'Dso:a'), dE=('Ds:dE', 'Dso:dE'),
            ),

        cf.Corr3(
            datatag='3ptT15', T=15, tmin=tmin, a='etas:a', dEa='etas:dE',
            b=('Ds:a', 'Dso:a'), dEb=('Ds:dE', 'Dso:dE'),
            Vnn='Vnn', Vno='Vno',
            ),

        cf.Corr3(
            datatag='3ptT16', T=16, tmin=tmin, a='etas:a', dEa='etas:dE',
            b=('Ds:a', 'Dso:a'), dEb=('Ds:dE', 'Dso:dE'), tpb=tp,
            Vnn='Vnn', Vno='Vno',
            )
        ]
    return models
```

Four models are specified, one for each correlator to be fit. The first two are for the η_s and D_s two-point correlators, corresponding to entries in the data dictionary with keys `'etas'` and `'Ds'`, respectively. These are periodic propagators, with period 64 (`tp`), and we want to omit the first and last 5 (`tmin`) time steps in the correlator. The `ts` to be fit are listed in `tfit`, while the `ts` contained in the data are in `tdata`. Labels for the fit parameters corresponding to the sources (and sinks) are specified for each, `'etas:a'` and `'Ds:a'`, as are labels for the energy differences, `'etas:dE'` and `'Ds:dE'`. The D_s propagator also has an oscillating piece because this data comes from a staggered-quark analysis. Sources/sinks and energy differences are specified for these as well: `'Dso:a'` and `'Dso:dE'`.

Finally three-point models are specified for the data corresponding to data-dictionary keys '3ptT15' and '3ptT16'. These share several parameters with the two-point correlators, but introduce new parameters for the transition matrix elements: 'Vnn' connecting normal states, and 'Vno' connecting normal states with oscillating states.

3.2.3 c) make_prior

Method make_prior(N) creates *a priori* estimates for each fit parameter, to be used as priors in the fitter:

```
def make_prior(N):
    """ Create priors for fit parameters. """
    prior = gv.BufferDict()
    # etas
    metas = gv.gvar('0.4(2)')
    prior['log(etas:a)'] = gv.log(gv.gvar(N * ['0.3(3)']))
    prior['log(etas:dE)'] = gv.log(gv.gvar(N * ['0.5(5)']))
    prior['log(etas:dE)'][0] = gv.log(metas)

    # Ds
    mDs = gv.gvar('1.2(2)')
    prior['log(Ds:a)'] = gv.log(gv.gvar(N * ['0.3(3)']))
    prior['log(Ds:dE)'] = gv.log(gv.gvar(N * ['0.5(5)']))
    prior['log(Ds:dE)'][0] = gv.log(mDs)

    # Ds -- oscillating part
    prior['log(Dso:a)'] = gv.log(gv.gvar(N * ['0.1(1)']))
    prior['log(Dso:dE)'] = gv.log(gv.gvar(N * ['0.5(5)']))
    prior['log(Dso:dE)'][0] = gv.log(mDs + gv.gvar('0.3(3)'))

    # V
    prior['Vnn'] = gv.gvar(N * [N * ['0(1)']])
    prior['Vno'] = gv.gvar(N * [N * ['0(1)']])
    return prior
```

Parameter N specifies how many terms are kept in the fit functions. The priors are stored in a dictionary prior. Each entry is an array, of length N, with one entry for each term in the fit function. Each entry is a Gaussian random variable, an object of type gvar.GVar. Here we use the fact that gvar.gvar() can make a list of gvar.GVars from a list of strings of the form '0.1(1)': for example,

```
>>> print(gv.gvar(['1(2)', '3(2)']))
[1.0(2.0) 3.0(2.0)]
```

In this particular fit, we can assume that all the sinks/sources are positive, and we can require that the energy differences be positive. To force positivity, we use log-normal distributions for these parameters by defining priors for 'log(etas:a)', 'log(etas:dE)' ... rather than 'etas:a', 'etas:dE' ... (see *Faster Fits — Postive Parameters*). The *a priori* values for these fit parameters are the logarithms of the values for the parameters themselves: for example, each 'etas:a' has prior 0.3(3), while the actual fit parameters, log(etas:a), have priors log(0.3(3)) = -1.2(1.0).

We override the default priors for the ground-state energies in each case. This is not unusual since dE[0], unlike the other dEs, is an energy, not an energy difference. For the oscillating D_s state, we require that its mass be 0.3(3) larger than the D_s mass. One could put more precise information into the priors if that made sense given the goals of the simulation. For example, if the main objective is a value for Vnn, one might include fairly exact information about the D_s and η_s masses in the prior, using results from experiment or from earlier simulations. This would make no sense, however, if the goal is to verify that simulations gives correct masses.

Note, finally, that a statement like

```
prior['Vnn'] = gv.gvar(N * [N* ['0(1)']])        # correct
```

is *not* the same as

```
prior['Vnn'] = N * [N * [gv.gvar('0(1)')]]       # wrong
```

The former creates `N ** 2` independent `gvar.GVars`, with one for each element of `Vnn`; it is one of the most succinct ways of creating a large number of `gvar.GVars`. The latter creates only a single `gvar.GVar` and uses it repeatedly for every element `Vnn`, thereby forcing every element of `Vnn` to be equal to every other element when fitting (since the difference between any two of their priors is 0 ± 0); it is almost certainly not what is desired. Usually one wants to create the array of strings first, and then convert it to `gvar.GVars` using `gvar.gvar()`.

3.2.4 d) print_results

Method `print_results(fit, prior, data)` reports on the best-fit values for the fit parameters from the last fit:

```
def print_results(fit, prior, data):
    """ Report best-fit results. """
    print('Fit results:')
    p = fit.p                       # best-fit parameters

    # etas
    E_etas = np.cumsum(p['etas:dE'])
    a_etas = p['etas:a']
    print('  Eetas:', E_etas[:3])
    print('  aetas:', a_etas[:3])

    # Ds
    E_Ds = np.cumsum(p['Ds:dE'])
    a_Ds = p['Ds:a']
    print('\n  EDs:', E_Ds[:3])
    print(  '  aDs:', a_Ds[:3])

    # Dso -- oscillating piece
    E_Dso = np.cumsum(p['Dso:dE'])
    a_Dso = p['Dso:a']
    print('\n  EDso:', E_Dso[:3])
    print(  '  aDso:', a_Dso[:3])

    # V
    Vnn = p['Vnn']
    Vno = p['Vno']
    print('\n  etas->V->Ds  =', Vnn[0, 0])
    print('  etas->V->Dso =', Vno[0, 0])

    # error budget
    outputs = collections.OrderedDict()
    outputs['metas'] = E_etas[0]
    outputs['mDs'] = E_Ds[0]
    outputs['mDso-mDs'] = E_Dso[0] - E_Ds[0]
    outputs['Vnn'] = Vnn[0, 0]
    outputs['Vno'] = Vno[0, 0]

    inputs = collections.OrderedDict()
    inputs['statistics'] = data             # statistical errors in data
    inputs.update(prior)                    # all entries in prior
```

```
    inputs['svd'] = fit.svdcorrection           # svd cut (if present)

    print('\n' + gv.fmt_values(outputs))
    print(gv.fmt_errorbudget(outputs, inputs))
    print('\n')
```

The best-fit parameter values are stored in dictionary `p=fit.p`, as are the exponentials of the log-normal parameters. We also turn energy differences into energies using `numpy`'s cummulative sum function `numpy.cumsum()`. The final output is:

```
Fit results:
  Eetas: [0.41620(12) 1.011(87) 1.44(35)]
  aetas: [0.21835(16) 0.174(72) 0.30(13)]

  EDs: [1.20163(16) 1.692(17) 2.17(17)]
  aDs: [0.21461(19) 0.261(21) 0.42(16)]

  EDso: [1.440(18) 1.64(14) 2.18(43)]
  aDso: [0.063(12) 0.079(25) 0.13(10)]

  etas->V->Ds  = 0.76736(79)
  etas->V->Dso = -0.767(68)
```

Finally we create an error budget for the η_s and D_s masses, for the mass difference between the D_s and its opposite-parity partner, and for the ground-state transition amplitudes `Vnn` and `Vno`. The quantities of interest are specified in dictionary `outputs`. For the error budget, we need another dictionary, `inputs`, specifying various inputs to the calculation: the Monte Carlo data, the priors, and the results from any *svd* cuts (none here). Each of these inputs contributes to the errors in the final results, as detailed in the error budget:

```
Values:
              metas: 0.41620(12)
                mDs: 1.20163(16)
           mDso-mDs: 0.239(18)
                Vnn: 0.76736(79)
                Vno: -0.767(68)

Partial % Errors:
                  metas        mDs   mDso-mDs        Vnn        Vno
---------------------------------------------------------------
  statistics:      0.03       0.01       5.21       0.09       6.69
 log(etas:a):      0.00       0.00       0.13       0.01       0.36
log(etas:dE):      0.00       0.00       0.11       0.01       0.34
   log(Ds:a):      0.00       0.00       0.41       0.01       1.03
  log(Ds:dE):      0.00       0.00       0.48       0.02       1.07
  log(Dso:a):      0.00       0.00       1.63       0.00       3.18
 log(Dso:dE):      0.00       0.00       2.75       0.00       3.00
         Vnn:      0.00       0.00       0.97       0.03       0.79
         Vno:      0.00       0.00       4.26       0.02       3.48
         svd:      0.00       0.00       0.40       0.01       0.48
---------------------------------------------------------------
       total:      0.03       0.01       7.55       0.10       8.91
```

The error budget shows, for example, that the largest sources of uncertainty in every quantity are the statistical errors in the input data.

3.3 Results

The output from running the code is as follows:

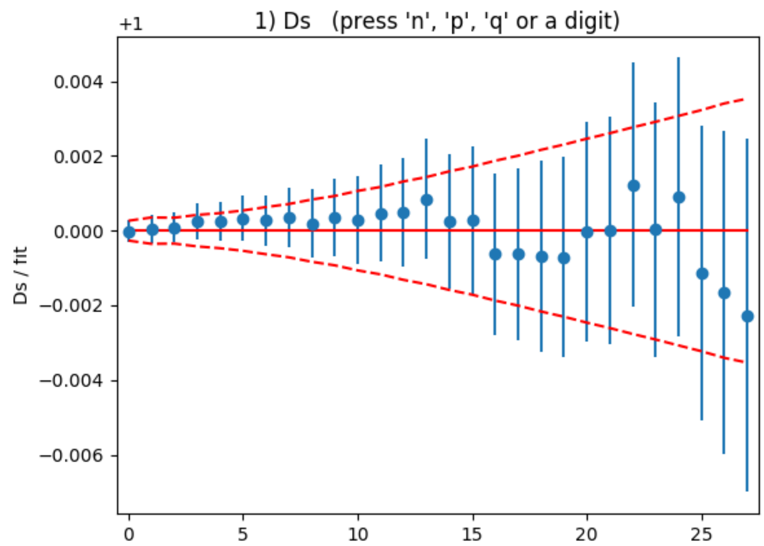
```
============================== nterm = 1
Least Square Fit:
  chi2/dof [dof] = 1.4e+04 [71]    Q = 0    logGBF = -4.9122e+05

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 30/0.2)

============================== nterm = 2
Least Square Fit:
  chi2/dof [dof] = 8.6 [71]    Q = 3.6e-87    logGBF = 1365.7

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 14/0.1)

============================== nterm = 3
Least Square Fit:
  chi2/dof [dof] = 0.68 [71]    Q = 0.98    logGBF = 1636.9

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 77/0.9)

============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.68 [71]    Q = 0.98    logGBF = 1637.6

Parameters:
  log(etas:a) 0      -1.52164 (73)        [ -1.2 (1.0) ]
              1         -1.75 (42)        [ -1.2 (1.0) ]
              2         -1.21 (44)        [ -1.2 (1.0) ]
              3         -1.30 (95)        [ -1.2 (1.0) ]
 log(etas:dE) 0      -0.87659 (28)        [ -0.92 (50) ]
              1         -0.52 (15)        [ -0.7 (1.0) ]
              2         -0.83 (64)        [ -0.7 (1.0) ]
              3         -0.63 (97)        [ -0.7 (1.0) ]
    log(Ds:a) 0      -1.53891 (90)        [ -1.2 (1.0) ]
              1        -1.343 (81)        [ -1.2 (1.0) ]
              2         -0.86 (38)        [ -1.2 (1.0) ]
              3         -1.03 (96)        [ -1.2 (1.0) ]
   log(Ds:dE) 0       0.18368 (13)        [  0.18 (17) ]
              1        -0.712 (35)        [ -0.7 (1.0) ]
              2         -0.74 (33)        [ -0.7 (1.0) ]
              3         -0.80 (98)        [ -0.7 (1.0) ]
   log(Dso:a) 0         -2.77 (19)        [ -2.3 (1.0) ]
              1         -2.54 (31)        [ -2.3 (1.0) ]
              2         -2.02 (76)        [ -2.3 (1.0) ]
  log(Dso:dE) 0         0.365 (13)        [  0.41 (24) ]
              1         -1.62 (66)        [ -0.7 (1.0) ]
              2         -0.61 (65)        [ -0.7 (1.0) ]
        Vnn 0,0       0.76736 (79)        [  0.0 (1.0) ]
            0,1        -0.478 (32)        [  0.0 (1.0) ]
            0,2          0.22 (27)        [  0.0 (1.0) ]
            0,3         -0.12 (95)        [  0.0 (1.0) ]
            1,0         0.048 (39)        [  0.0 (1.0) ]
            1,1          0.50 (69)        [  0.0 (1.0) ]
```

```
            1,2          0.15 (98)        [  0.0 (1.0) ]
            1,3        0.02 (1.00)        [  0.0 (1.0) ]
            2,0         -0.07 (20)        [  0.0 (1.0) ]
            2,1        0.04 (1.00)        [  0.0 (1.0) ]
            2,2      0.002 (1.000)        [  0.0 (1.0) ]
            2,3    0.0004 (1.0000)        [  0.0 (1.0) ]
            3,0         -0.04 (98)        [  0.0 (1.0) ]
            3,1      0.003 (1.000)        [  0.0 (1.0) ]
            3,2         6e-05 +- 1        [  0.0 (1.0) ]
            3,3         1e-06 +- 1        [  0.0 (1.0) ]
        Vno 0,0        -0.767 (68)        [  0.0 (1.0) ]
            0,1          0.15 (34)        [  0.0 (1.0) ]
            0,2          0.50 (51)        [  0.0 (1.0) ]
            0,3          0.12 (99)        [  0.0 (1.0) ]
            1,0          0.24 (43)        [  0.0 (1.0) ]
            1,1          0.07 (93)        [  0.0 (1.0) ]
            1,2       -0.06 (1.00)        [  0.0 (1.0) ]
            1,3     -0.007 (1.000)        [  0.0 (1.0) ]
            2,0         -0.15 (93)        [  0.0 (1.0) ]
            2,1      0.009 (0.999)        [  0.0 (1.0) ]
            2,2        -9e-05 +- 1        [  0.0 (1.0) ]
            2,3   -0.0001 (1.0000)        [  0.0 (1.0) ]
            3,0       -0.03 (1.00)        [  0.0 (1.0) ]
            3,1     -0.003 (1.000)        [  0.0 (1.0) ]
            3,2         2e-05 +- 1        [  0.0 (1.0) ]
            3,3         1e-07 +- 1        [  0.0 (1.0) ]
--------------------------------------------------------
       etas:a 0       0.21835 (16)        [  0.30 (30) ]
              1         0.174 (72)        [  0.30 (30) ]
              2          0.30 (13)        [  0.30 (30) ]
              3          0.27 (26)        [  0.30 (30) ]
      etas:dE 0       0.41620 (12)        [  0.40 (20) ]
              1         0.595 (87)        [  0.50 (50) ]
              2          0.43 (28)        [  0.50 (50) ]
              3          0.53 (51)        [  0.50 (50) ]
         Ds:a 0       0.21461 (19)        [  0.30 (30) ]
              1         0.261 (21)        [  0.30 (30) ]
              2          0.42 (16)        [  0.30 (30) ]
              3          0.36 (34)        [  0.30 (30) ]
        Ds:dE 0       1.20163 (16)        [  1.20 (20) ]
              1         0.491 (17)        [  0.50 (50) ]
              2          0.48 (16)        [  0.50 (50) ]
              3          0.45 (44)        [  0.50 (50) ]
        Dso:a 0         0.063 (12)        [  0.10 (10) ]
              1         0.079 (25)        [  0.10 (10) ]
              2          0.13 (10)        [  0.10 (10) ]
       Dso:dE 0         1.440 (18)        [  1.50 (36) ]
              1          0.20 (13)        [  0.50 (50) ]
              2          0.54 (35)        [  0.50 (50) ]
              3          0.48 (48)        [  0.50 (50) ]

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 40/0.8)

Fit results:
  Eetas: [0.41620(12) 1.011(87) 1.44(35)]
  aetas: [0.21835(16) 0.174(72) 0.30(13)]
```

```
  EDs: [1.20163(16) 1.692(17) 2.17(17)]
  aDs: [0.21461(19) 0.261(21) 0.42(16)]

  EDso: [1.440(18) 1.64(14) 2.18(43)]
  aDso: [0.063(12) 0.079(25) 0.13(10)]

  etas->V->Ds  = 0.76736(79)
  etas->V->Dso = -0.767(68)

Values:
              metas: 0.41620(12)
                mDs: 1.20163(16)
           mDso-mDs: 0.239(18)
                Vnn: 0.76736(79)
                Vno: -0.767(68)

Partial % Errors:
                  metas       mDs  mDso-mDs       Vnn       Vno
---------------------------------------------------------------
  statistics:      0.03      0.01      5.21      0.09      6.69
 log(etas:a):      0.00      0.00      0.13      0.01      0.36
log(etas:dE):      0.00      0.00      0.11      0.01      0.34
   log(Ds:a):      0.00      0.00      0.41      0.01      1.03
  log(Ds:dE):      0.00      0.00      0.48      0.02      1.07
  log(Dso:a):      0.00      0.00      1.63      0.00      3.18
 log(Dso:dE):      0.00      0.00      2.75      0.00      3.00
         Vnn:      0.00      0.00      0.97      0.03      0.79
         Vno:      0.00      0.00      4.26      0.02      3.48
         svd:      0.00      0.00      0.40      0.01      0.48
---------------------------------------------------------------
       total:      0.03      0.01      7.55      0.10      8.91
```

Note:

- This is a relatively simple fit, taking only a couple of seconds on a laptop.
- Fits with only one or two terms in the fit function are poor, with `chi2/dof`s that are significantly larger than one.
- Fits with three terms work well, and adding futher terms has almost no impact. The chi-squared does not improve and parameters for the added terms differ little from their prior values (since the data are not sufficiently accurate to add new information).
- The quality of the fit is confirmed by the fit plots displayed at the end (press the 'n' and 'p' keys to cycle through the various plots, and the 'q' key to quit the plot). The plot for the D_s correlator, for example, shows correlator data divided by fit result as a function of `t`:



The points with error bars are the correlator data points; the fit result is 1.0 in this plot, of course, and the dashed lines show the uncertainty in the fit function evaluated with the best-fit parameters. Fit and data agree to within errors. Note how the fit-function errors (the dashed lines) track the data errors. In general the fit function is at least as accurate as the data. It can be much more accurate, for example, when the data errors grow rapidly with `t`.

- In many applications precision can be improved by factors of 2—3 or more by using multiple sources and sinks for the correlators. The code here is easily generalized to handle such a situation: each *corrfitter.Corr2* and *corrfitter.Corr3* in `make_models()` is replicated with various different combinations of sources and sinks (one entry for each combination).

3.4 Variation: Marginalization

Marginalization (see *Faster Fits — Marginalization*) can speed up fits like this one. To use an 8-term fit function, while tuning parameters for only `N` terms, we change only four lines in the main program:

```
def main():
    data = make_data('etas-Ds.h5')
    models = make_models()
    prior = make_prior(8)
    fitter = CorrFitter(models=make_models(), svdcut=1e-5)
    p0 = None
    for N in [1, 2]:                                                  # 1
        print(30 * '=', 'nterm =', N)
        prior = make_prior(8)                                         # 2
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0, nterm=(N, N))    # 3
        print(fit)                                                    # 4
        p0 = fit.pmean
    print_results(fit, prior, data)
    if DISPLAYPLOTS:
        fitter.display_plots()
```

The first modification (`#1`) limits the fits to `N=1,2`, because that is all that will be needed to get good values for the leading term. The second modification (`#2`) sets the prior to eight terms, no matter what value `N` has. The third (`#3`) tells `fitter.lsqfit` to fit parameters from only the first `N` terms in the fit function; parts of the prior that are not being fit are incorporated (*marginalized*) into the fit data. The last modification (`#4`) changes what is printed out. The output shows that results for the leading term have converged by `N=2` (and even `N=1` is pretty good):

```
============================== nterm = 1
Least Square Fit:
  chi2/dof [dof] = 0.59 [71]    Q = 1    logGBF = 1586.7

Parameters:
  log(etas:a) 0   -1.52164 (85)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87662 (30)      [ -0.92 (50) ]
    log(Ds:a) 0    -1.5386 (13)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18370 (16)      [  0.18 (17) ]
   log(Dso:a) 0     -2.622 (94)      [ -2.3 (1.0) ]
  log(Dso:dE) 0     0.3753 (98)      [  0.41 (24) ]
        Vnn 0,0     0.7674 (35)      [  0.0 (1.0) ]
        Vno 0,0     -0.708 (30)      [  0.0 (1.0) ]
---------------------------------------------------
       etas:a 0    0.21835 (18)      [  0.30 (30) ]
      etas:dE 0    0.41619 (12)      [  0.40 (20) ]
         Ds:a 0    0.21469 (28)      [  0.30 (30) ]
        Ds:dE 0    1.20165 (19)      [  1.20 (20) ]
        Dso:a 0     0.0726 (68)      [  0.10 (10) ]
       Dso:dE 0      1.455 (14)      [  1.50 (36) ]

Settings:
  svdcut/n = 1e-05/27    tol = (1e-08*,1e-10,1e-10)    (itns/time = 8/0.1)

============================== nterm = 2
Least Square Fit:
  chi2/dof [dof] = 0.66 [71]    Q = 0.99    logGBF = 1633.3

Parameters:
  log(etas:a) 0   -1.52170 (74)      [ -1.2 (1.0) ]
              1      -1.74 (53)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87662 (28)      [ -0.92 (50) ]
              1      -0.53 (17)      [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53905 (94)      [ -1.2 (1.0) ]
              1     -1.367 (84)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18367 (13)      [  0.18 (17) ]
              1     -0.724 (40)      [ -0.7 (1.0) ]
   log(Dso:a) 0     -2.714 (73)      [ -2.3 (1.0) ]
              1      -2.43 (12)      [ -2.3 (1.0) ]
  log(Dso:dE) 0     0.3688 (63)      [  0.41 (24) ]
              1      -1.40 (23)      [ -0.7 (1.0) ]
        Vnn 0,0    0.76716 (82)      [  0.0 (1.0) ]
            0,1     -0.461 (30)      [  0.0 (1.0) ]
            1,0      0.072 (55)      [  0.0 (1.0) ]
            1,1       0.17 (85)      [  0.0 (1.0) ]
        Vno 0,0     -0.776 (41)      [  0.0 (1.0) ]
            0,1       0.31 (16)      [  0.0 (1.0) ]
            1,0       0.27 (41)      [  0.0 (1.0) ]
            1,1       0.20 (97)      [  0.0 (1.0) ]
---------------------------------------------------
       etas:a 0    0.21834 (16)      [  0.30 (30) ]
              1      0.175 (92)      [  0.30 (30) ]
      etas:dE 0    0.41619 (12)      [  0.40 (20) ]
              1       0.59 (10)      [  0.50 (50) ]
         Ds:a 0    0.21458 (20)      [  0.30 (30) ]
```

```
             1      0.255 (21)      [  0.30 (30) ]
       Ds:dE 0    1.20161 (16)      [  1.20 (20) ]
             1      0.485 (19)      [  0.50 (50) ]
       Dso:a 0     0.0663 (48)      [  0.10 (10) ]
             1      0.088 (10)      [  0.10 (10) ]
      Dso:dE 0     1.4459 (91)      [  1.50 (36) ]
             1      0.248 (57)      [  0.50 (50) ]

Settings:
  svdcut/n = 1e-05/12    tol = (1e-08*,1e-10,1e-10)    (itns/time = 25/0.2)

Fit results:
  Eetas: [0.41619(12) 1.01(10)]
  aetas: [0.21834(16) 0.175(92)]

  EDs: [1.20161(16) 1.687(19)]
  aDs: [0.21458(20) 0.255(21)]

  EDso: [1.4459(91) 1.693(60)]
  aDso: [0.0663(48) 0.088(10)]

  etas->V->Ds  = 0.76716(82)
  etas->V->Dso = -0.776(41)

Values:
              metas: 0.41619(12)
                mDs: 1.20161(16)
           mDso-mDs: 0.2443(91)
                Vnn: 0.76716(82)
                Vno: -0.776(41)

Partial % Errors:
                  metas       mDs  mDso-mDs       Vnn       Vno
---------------------------------------------------------------
  statistics:      0.03      0.01      2.66      0.09      4.57
 log(etas:a):      0.00      0.00      0.04      0.01      0.43
log(etas:dE):      0.00      0.00      0.02      0.01      0.16
   log(Ds:a):      0.00      0.00      0.07      0.02      0.23
  log(Ds:dE):      0.00      0.00      0.10      0.03      0.16
  log(Dso:a):      0.00      0.00      0.36      0.00      1.19
 log(Dso:dE):      0.00      0.00      0.47      0.00      1.69
         Vnn:      0.00      0.00      0.53      0.05      0.52
         Vno:      0.00      0.00      2.44      0.01      0.81
         svd:      0.00      0.00      0.44      0.02      1.39
---------------------------------------------------------------
       total:      0.03      0.01      3.72      0.11      5.32
```

3.5 Variation: Chained Fit

Chained fits (see *Faster Fits — Chained Fits*) are used if `fitter.lsqfit(...)` is replaced by `fitter.chained_lsqfit(...)` in `main()`. The results are about the same: for example,

```
Fit results:
  Eetas: [0.41619(11) 1.012(29) 1.439(58)]
  aetas: [0.21835(15) 0.174(17) 0.295(99)]
```

```
  EDs: [1.20162(17) 1.688(15) 2.205(71)]
  aDs: [0.21460(22) 0.258(14) 0.481(99)]

  EDso: [1.4531(39) 1.728(50) 2.02(15)]
  aDso: [0.0717(14) 0.070(17) 0.090(77)]

  etas->V->Ds  = 0.76772(98)
  etas->V->Dso = -0.759(26)

Values:
              metas: 0.41619(11)
                mDs: 1.20162(17)
           mDso-mDs: 0.2515(39)
                Vnn: 0.76772(98)
                Vno: -0.759(26)

Partial % Errors:
                  metas       mDs  mDso-mDs       Vnn       Vno
---------------------------------------------------------------
  statistics:      0.03      0.01      1.40      0.11      2.95
 log(etas:a):      0.00      0.00      0.06      0.02      0.25
log(etas:dE):      0.00      0.00      0.03      0.01      0.21
   log(Ds:a):      0.00      0.00      0.11      0.01      0.23
  log(Ds:dE):      0.00      0.00      0.07      0.02      0.37
  log(Dso:a):      0.00      0.00      0.32      0.00      0.83
 log(Dso:dE):      0.00      0.00      0.24      0.00      0.49
         Vnn:      0.00      0.00      0.25      0.05      0.87
         Vno:      0.00      0.00      0.42      0.01      0.78
         svd:      0.00      0.00      0.14      0.00      0.11
---------------------------------------------------------------
       total:      0.03      0.01      1.55      0.13      3.36
```

Chained fits are particularly useful for very large data sets (much larger than this one).

3.6 Test the Analysis

We can test our analysis by adding `test_fit(fitter, 'etas-Ds.h5')` to the `main` program, where:

```
def test_fit(fitter, datafile):
    """ Test the fit with simulated data """
    gv.ranseed(98)
    print('\nRandom seed:', gv.ranseed.seed)
    dataset = h5py.File(datafile)
    pexact = fitter.fit.pmean
    prior = fitter.fit.prior
    for spdata in fitter.simulated_pdata_iter(n=2, dataset=dataset, pexact=pexact):
        print('\n============================== simulation')
        sfit = fitter.lsqfit(pdata=spdata, prior=prior, p0=pexact)
        print(sfit.format(pstyle=None))
        # check chi**2 for key parameters
        diff = {}
        for k in ['etas:a', 'etas:dE', 'Ds:a', 'Ds:dE', 'Vnn']:
            p_k = sfit.p[k].flat[0]
            pex_k = pexact[k].flat[0]
            print(
                '{:>10}:  fit = {}    exact = {:<9.5}    diff = {}'
```

```
                    .format(k, p_k, pex_k, p_k - pex_k)
                )

            diff[k] = p_k - pex_k
        print('\nAccuracy of key parameters: ' + gv.fmt_chi2(gv.chi2(diff)))
```

This code does n=2 simulations of the full fit, using the means of fit results from the last fit done by `fitter` as `pexact`. The code compares fit results woth `pexact` in each case, and computes the chi-squared of the difference between the leading parameters and `pexact`. The output is:

```
Random seed: 98

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.8 [71]    Q = 0.89    logGBF = 1633.6

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 61/1.2)

    etas:a:  fit = 0.21816(16)    exact = 0.21835      diff = -0.00019(16)
   etas:dE:  fit = 0.41610(11)    exact = 0.4162       diff = -0.00010(11)
      Ds:a:  fit = 0.21449(19)    exact = 0.21461      diff = -0.00013(19)
     Ds:dE:  fit = 1.20153(16)    exact = 1.2016       diff = -0.00010(16)
       Vnn:  fit = 0.76585(98)    exact = 0.76736      diff = -0.00151(98)

Accuracy of key parameters: chi2/dof = 0.86 [5]    Q = 0.51

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.49 [71]    Q = 1    logGBF = 1647.9

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 81/1.7)

    etas:a:  fit = 0.21835(15)    exact = 0.21835      diff = 5(1497)e-07
   etas:dE:  fit = 0.41620(11)    exact = 0.4162       diff = -6(1110)e-07
      Ds:a:  fit = 0.21460(19)    exact = 0.21461      diff = -0.00001(19)
     Ds:dE:  fit = 1.20161(16)    exact = 1.2016       diff = -0.00003(16)
       Vnn:  fit = 0.76633(78)    exact = 0.76736      diff = -0.00102(78)

Accuracy of key parameters: chi2/dof = 0.35 [5]    Q = 0.88
```

This shows that the fit is working well.

Other options are easily checked. For example, only one line need be changed in `test_fit` in order to test a marginalized fit:

```
sfit = fitter.lsqfit(pdata=spdata, prior=prior, p0=pexact, nterm=(2,2))
```

Running this code gives:

```
Random seed: 98

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.84 [71]    Q = 0.82    logGBF = 1647.2

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 11/0.1)
```

```
    etas:a:  fit = 0.21821(14)    exact = 0.21834      diff = -0.00013(14)
   etas:dE:  fit = 0.41611(11)    exact = 0.41619      diff = -0.00007(11)
      Ds:a:  fit = 0.21447(13)    exact = 0.21458      diff = -0.00011(13)
     Ds:dE:  fit = 1.20150(13)    exact = 1.2016       diff = -0.00011(13)
       Vnn:  fit = 0.76639(42)    exact = 0.76716      diff = -0.00077(42)

Accuracy of key parameters: chi2/dof = 0.86 [5]    Q = 0.51

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.58 [71]    Q = 1    logGBF = 1656.7

Settings:
  svdcut/n = 1e-05/7    tol = (1e-08*,1e-10,1e-10)    (itns/time = 14/0.1)

    etas:a:  fit = 0.21830(14)    exact = 0.21834      diff = -0.00004(14)
   etas:dE:  fit = 0.41617(11)    exact = 0.41619      diff = -0.00002(11)
      Ds:a:  fit = 0.21458(13)    exact = 0.21458      diff = 7(1319)e-07
     Ds:dE:  fit = 1.20161(13)    exact = 1.2016       diff = -9(131)e-06
       Vnn:  fit = 0.76668(42)    exact = 0.76716      diff = -0.00048(42)

Accuracy of key parameters: chi2/dof = 0.3 [5]    Q = 0.91
```

This is also fine and confirms that `nterm=(2,2)` marginalized fits are a useful, faster substitute for full fits in this case.

3.7 Mixing

Code to analyze D_s-D_s mixing is very similar to the code above for a transition form factor. The `main()` and `make_data()` functions are identical, except that here data is read from file `'Ds-Ds.h5'`. We need models for the two-point D_s correlator, and for two three-point correlators describing the D_s to D_s transition:

```
def make_models():
    """ Create models to fit data. """
    tmin = 3
    tp = 64
    models = [
        cf.Corr2(
            datatag='Ds', tp=tp, tmin=tmin,
            a=('a', 'ao'), b=('a', 'ao'), dE=('dE', 'dEo'), s=(1., -1.),
            ),
        cf.Corr3(
            datatag='DsDsT18', T=18, tmin=tmin,
            a=('a', 'ao'), dEa=('dE', 'dEo'), tpa=tp, sa=(1., -1),
            b=('a', 'ao'), dEb=('dE', 'dEo'), tpb=tp, sb=(1., -1.),
            Vnn='Vnn', Voo='Voo', Vno='Vno', symmetric_V=True,
            ),
        cf.Corr3(
            datatag='DsDsT15', T=15, tmin=tmin,
            a=('a', 'ao'), dEa=('dE', 'dEo'), tpa=tp, sa=(1., -1),
            b=('a', 'ao'), dEb=('dE', 'dEo'), tpb=tp, sb=(1., -1.),
            Vnn='Vnn', Voo='Voo', Vno='Vno', symmetric_V=True,
            )
        ]
    return models
```

The initial and final states in the three-point correlators are the same here so we set parameter `symmetricV=True` in *corrfitter.Corr3*.

The prior is also similar to the previous case:

```
def make_prior(N):
    """ Create priors for fit parameters. """
    prior = gv.BufferDict()
    # Ds
    mDs = gv.gvar('1.2(2)')
    prior['log(a)'] = gv.log(gv.gvar(N * ['0.3(3)']))
    prior['log(dE)'] =  gv.log(gv.gvar(N * ['0.5(5)']))
    prior['log(dE)'][0] = gv.log(mDs)

    # Ds -- oscillating part
    prior['log(ao)'] = gv.log(gv.gvar(N * ['0.1(1)']))
    prior['log(dEo)'] = gv.log(gv.gvar(N * ['0.5(5)']))
    prior['log(dEo)'][0] = gv.log(mDs + gv.gvar('0.3(3)'))

    # V
    nV = int((N * (N + 1)) / 2)
    prior['Vnn'] = gv.gvar(nV * ['0.0(5)'])
    prior['Voo'] = gv.gvar(nV * ['0.0(5)'])
    prior['Vno'] = gv.gvar(N * [N * ['0.0(5)']])
    return prior
```

We use log-normal distributions for the energy differences, and amplitudes. We store only the upper triangular parts of the `Vnn` and `Voo` matrices since they are symmetrical (because `symmetricV=True` is set).

A minimal `print_results()` function is:

```
def print_results(fit, prior, data):
    """ Print results of fit. """
    outputs = collections.OrderedDict()
    outputs['mDs'] = fit.p['dE'][0]
    outputs['Vnn'] = fit.p['Vnn'][0]

    inputs = collections.OrderedDict()
    inputs['statistics'] = data              # statistical errors in data
    inputs['Ds priors'] = {
        k:prior[k] for k in ['log(a)', 'log(dE)', 'log(ao)', 'log(dEo)']
        }
    inputs['V priors'] = {
        k:prior[k] for k in ['Vnn', 'Vno', 'Voo']
        }
    inputs['svd'] = fit.svdcorrection        # errors from svd cut (if present)

    print('\n' + gv.fmt_values(outputs))
    print(gv.fmt_errorbudget(outputs, inputs))
```

Running the mixing code gives the following output:

```
============================== nterm = 1
Least Square Fit:
  chi2/dof [dof] = 1.6e+04 [55]    Q = 0    logGBF = -4.447e+05

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 36/0.2)

============================== nterm = 2
```

```
Least Square Fit:
  chi2/dof [dof] = 6.7 [55]    Q = 1e-47    logGBF = 1469.7

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 51/0.4)

============================== nterm = 3
Least Square Fit:
  chi2/dof [dof] = 0.72 [55]    Q = 0.94    logGBF = 1618

Settings:
  svdcut/n = 1e-12/0    tol = (1e-08*,1e-10,1e-10)    (itns/time = 118/0.9)

============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.72 [55]    Q = 0.94    logGBF = 1616.5

Parameters:
       log(a) 0      -1.5553 (59)       [ -1.2 (1.0) ]
              1        -1.67 (33)       [ -1.2 (1.0) ]
              2        -0.73 (13)       [ -1.2 (1.0) ]
              3        -1.50 (94)       [ -1.2 (1.0) ]
      log(dE) 0      0.27132 (56)       [  0.18 (17) ]
              1        -0.95 (22)       [ -0.7 (1.0) ]
              2        -0.86 (13)       [ -0.7 (1.0) ]
              3        -0.81 (99)       [ -0.7 (1.0) ]
      log(ao) 0        -2.76 (12)       [ -2.3 (1.0) ]
              1        -2.15 (13)       [ -2.3 (1.0) ]
              2        -2.60 (70)       [ -2.3 (1.0) ]
              3        -2.46 (95)       [ -2.3 (1.0) ]
     log(dEo) 0        0.341 (10)       [  0.41 (24) ]
              1        -1.27 (21)       [ -0.7 (1.0) ]
              2        -0.76 (93)       [ -0.7 (1.0) ]
              3        -0.58 (97)       [ -0.7 (1.0) ]
          Vnn 0       0.1065 (16)       [  0.00 (50) ]
              1       -0.002 (21)       [  0.00 (50) ]
              2       -0.014 (49)       [  0.00 (50) ]
              3        -0.05 (20)       [  0.00 (50) ]
              4       0.004 (500)       [  0.00 (50) ]
              5    -0.0005 (4991)       [  0.00 (50) ]
              6     0.0008 (4998)       [  0.00 (50) ]
              7   0.00006 (50000)       [  0.00 (50) ]
              8     -1e-06 +- 0.5       [  0.00 (50) ]
              9      2e-08 +- 0.5       [  0.00 (50) ]
        Vno 0,0      -0.2117 (90)       [  0.00 (50) ]
            0,1       -0.003 (46)       [  0.00 (50) ]
            0,2         0.05 (18)       [  0.00 (50) ]
            0,3        -0.13 (39)       [  0.00 (50) ]
            1,0        0.024 (85)       [  0.00 (50) ]
            1,1       0.001 (500)       [  0.00 (50) ]
            1,2      2e-06 +- 0.5       [  0.00 (50) ]
            1,3     0.0001 (5000)       [  0.00 (50) ]
            2,0         0.02 (11)       [  0.00 (50) ]
            2,1       0.002 (500)       [  0.00 (50) ]
            2,2     -9e-07 +- 0.5       [  0.00 (50) ]
            2,3      3e-07 +- 0.5       [  0.00 (50) ]
            3,0        -0.05 (39)       [  0.00 (50) ]
            3,1    -0.0006 (5000)       [  0.00 (50) ]
```