corrfitter Documentation

Release 4.1.1

G.P. Lepage

June 17, 2015

Contents:

`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.

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` can also capture any statistical correlations between different pieces of data.

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:

```python
from corrfitter import CorrFitter

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 = CorrFitter(models=models)
fit = fitter.lsqfit(data=data, prior=prior)   # do the fit
print_results(fit, prior, data)     # user-supplied routine
```

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. Imagine 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 each line. (Lines for `Gab` may be interspersed with lines for `Gaa` since every line has a tag.) The data can be analyzed using the `gvar.dataset` module:

```python
import gvar as gv

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

This reads the data from file into a dataset object (type `gvar.dataset.Dataset`) and then computes averages for each correlator and `t`, together with a covariance matrix for the set of averages. Thus `data = make_data('mcfile')` creates a dictionary where `data['Gaa']` is a 1-d array of `gvar.GVars` obtained by averaging over the `Gaa` data in the `'mcfile'`, and `data['Gab']` is a similar array for the `Gab` correlator.

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:

```python
from corrfitter import Corr2

def make_models():
    models = [ Corr2(datatag='Gaa', tdata=range(64), tfit=range(64),
                    a='a', b='a', dE='dE'),

               Corr2(datatag='Gab', tdata=range(64), tfit=range(64),
                    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 values of `t` for which results are given in the input data (`tdata`); the values of `t` to keep for fits (`tfit`, here the same as the range in the input data, but could be any subset); 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:

```python
Corr2(datatag='Gab', tdata=range(64), tfit=range(64),
      a=('a', None), b=('b', None), dE=('dE', None),
      othertags=['Gba'])
```

The additional argument `othertags` lists other data tags that correspond to the same physical quantity; the data for all equivalent data tags is averaged before fitting (using `lsqfit.wavg()`). Alternatively (and equivalently) one could add a third `Corr2` to `models` for `Gba`, but it is more efficient to combine it with `Gab` in this way if they are equivalent.

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 class `gvar.BufferDict`:

```python
import gvar as gv

def make_prior(N):
    prior = gvar.BufferDict()        # prior = {}  works too
    prior['a'] = [gv.gvar(0.1, 0.5) for i in range(N)]
    prior['b'] = [gv.gvar(1., 5.) for i in range(N)]
    prior['dE'] = [gv.gvar(0.25, 0.25) for i in range(N)]
    return prior
```

(`gvar.BufferDict` can be replaced by an ordinary Python dictionary; it is used here because it remembers the order in which the keys are added.) `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(...)`, which also prints out a summary of the fit results (this output can be suppressed if desired). Further results are reported by `print_results(fit, prior, data)`: for example,

```
def print_results(fit, prior, data):
    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 = [sum(dE[:i+1]) for i in range(len(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 `fitter.display_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 = 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 `'logdE'` or `'log(dE)'`: for example, in `make_prior(N)` use

```
prior['logdE'] = [gv.log(gv.gvar(0.25, 0.25)) for i in range(N)]
```

instead of `prior['dE'] = [gv.gvar(0.25, 0.25) for i in range(N)]`. 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 `'logdE'` 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

def make_models():                            # same as before
    models = [ Corr2(datatag='Gaa', tdata=range(64), tfit=range(64),
                a='a', b='a', dE='dE'),

               Corr2(datatag='Gab', tdata=range(64), tfit=range(64),
                a='a', b='b', dE='dE')
             ]
    return models

def make_prior(N):
```

```
    prior = gvar.BufferDict()              # prior = {}  works too
    prior['loga'] = [gv.log(gv.gvar(0.1, 0.5)) for i in range(N)]
    prior['b'] = [gv.gvar(1., 5.) for i in range(N)]
    prior['logdE'] = [gv.log(gv.gvar(0.25, 0.25)) for i in range(N)]
    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 — only the names in the prior need be changed. *corrfitter.CorrFitter* also supports "sqrt-normal" distributions, which are indicated by `'sqrt'` at the start of a parameter-name in the prior; the actual parameter in the fit function is the square of this fit- parameter, and so is again positive.

Note also that only a few lines in `print_results(fit,prior,data)`, above, would change had we used log-normal priors for `a` and `dE`:

```
...
a = fit.transformed_p['a'])                # array of a[i]s
...
dE = fit.transformed_p['dE']               # array of dE[i]s
...
inputs = {'loga':prior['loga'], 'b':prior['b'], 'logdE':fit.prior['logdE'],
          'data':[data[k] for k in data]}
...
```

Here `fit.transformed_p` contains the best-fit parameter values from the fitter, in addition to the exponentials of the `'loga'` and `'logdE'` parameters.

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

```
    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.)` rather than `gvar.gvar(0.0,1.)`. 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 never involves fitting more than a single correlator at a time. Single-correlator fits are usually much faster than simultaneous multi-correlator fits, and roundoff errors (and therefore *svd* cuts) are much less of a problem. Consequently chained fits can be more accurate in practice than conventional simultaneous fits, especially for high-statistics data.

Converting to chained fits is trivial: simply replace `fitter.lsqfit(...)` by `fitter.chained_lsqfit(...)`. The output from this function represents the results for the entire chain of fits, and so can be used in exactly the same way as the output from `fitter.lsqfit()` (and is usually quite similar, to within statistical errors). Results from the different links in the chain — that is, from the fits for individual models — can be accessed after the fit using `fitter.fit.fits[datatag]` where `datatag` is the data tag for the model of interest.

Setting parameter `parallel=True` in `fitter.chained_lsqfit(...)` makes the fits for each model independent of each other. Each correlator is fit separately, but nothing is passed from one fit to the next. In particular, each fit uses the input prior. Parallel fits can be better than chained fits in situations where different models share few or no parameters.

It is sometimes useful to combine chained and parallel fits. This is done by using a nested list of models. For example, setting

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

with `parallel=False` (the default) in `fitter.chained_lsqfit` 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`. Different levels of nesting in the list of models alternate between chained and parallel fits.

It is sometimes useful to follow a chained fit with an ordinary fit, but using the best-fit parameters from the chained fit as the prior for the ordinary fit: for example,

```
fit = fitter.chained_lsqfit(data=data, prior=prior)
fit = fitter.lsqfit(data=data, prior=fit.p)
```

The second fit should, in principle, have no effect on the results since it adds no new information. In some cases, however, it polishes the results by making small (compared to the errors) corrections that tighten up the overall fit. It is generally fairly fast since the prior (`fit.p`) is relatively narrow. It is also possible to polish fits using `fitter.chained_lsqfit`, with parameters `parallel=True` and `flat=True`, rather than using `fitter.lsqfit`. This can be faster for very large fits.

1.7 Variations

Any 2-point correlator can be 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', tdata=range(64), tfit=range(64),
      a=('a','ao'), b=('a','ao'), dE=('logdE','logdEo'), 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.

1.8 Very Fast (But Limited) Fits

At large `t`, 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. An example is:

```python
from corrfitter import fastfit

data = make_data('mcfile')     # user-supplied routine - fit data
N = 10                         # number of terms in fit functions
prior = make_prior(N)          # user-supplied routine - fit prior
model = Corr2(a=..., b=..., ...) # create model describing correlator
fit = fastfit(data=data, prior=prior, model=model)
print('E[0] =', fit.E)              # E[0]
print('a[0]*b[0] =', fit.ampl)      # a[0]*b[0]
print('chi2/dof =', fit.chi2/fit.dof)   # good fit if of order 1 or less
print('Q =', fit.Q)                 # good fit if Q bigger than about 0.1
```

fastfit estimates `E[0]` by using the prior, in effect, to remove (*marginalize*) all terms from the correlator other than the `E[0]` term: so the data `Gdata(t)` for the correlator is replaced by, for example,

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

where `a[i]`, `b[i]`, and `E[i]` for `i>0` are replaced by their values in the prior. The modified prior is then fit by a single term, `a[0] * b[0] * exp(-E[0]*t)`, which means that a fit is not necessary (since the functional form is so simple). It is important to check the `chi**2` of the fit, to make sure the fit is good. If it is not, try restricting `model.tfit` to larger `t`s (*fastfit* averages estimates from all `t`s in `model.tfit`).

The marginalization of terms with larger `E`s allows the code to use information from much smaller `t`s than otherwise, increasing precision. It also quantifies the uncertainty caused by the existence of these terms. This simple analysis is a special case of the more general marginalization strategy discussed in *Faster Fits*, above.

1.9 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:

```python
Corr3(datatag='aVbT15', T=15, tdata=range(16), tfit=range(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`), or can be periodic (include `tpa` and/or `tpb`). If there are oscillating states then additional `V`s 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.

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:

```python
models = [
    ...
    Corr3(datatag='aVbT15', T=15, tdata=range(16), tfit=range(16),
        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, tdata=range(16), tfit=range(16),
            Vnn='Vnn', Vno='Vno', Von='Von', Voo='Voo', transpose_V=True,
            a=('b','bo'), dEa=('dE','dEo'), sa=(1,-1), # b->V
            b=('a','ao'), dEb=('dE','dEo'), sb=(1,-1)  # V->a
            ),
        ...
]
```

The same `Vs` are specified for the second correlator, but setting `transpose_V=True` means that the transpose of each matrix is used in the fit for that correlator.

The second special case is for fits to `a->V->a` where source and sink are the same. 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:

```
Corr3(datatag='aVbT15', T=15, tdata=range(16), tfit=range(16),
    Vnn='Vnn', Vno='Vno', Von='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 `Vno` and `Von` are set equal to the same matrix, but specifying `symmetric_V=True` implies that the transpose will be used for `Von`. Furthermore `Vnn` and `Voo` are 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).

1.10 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_data_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:

```python
import gvar as gv
import lsqfit
import corrfitter

def main():
    dataset = gv.dataset.Dataset(...)      # from original fit code
    fitter = corrfitter.CorrFitter(        # from original fit code
        models = make_models(...),
        prior = make_prior(...),
        ...
```

```
        )
    n = 2                                     # number of simulations
    pexact = lsqfit.nonlinear_fit.load_parameters("pexact_file")
    for sdata in fitter.simulated_data_iter(n, dataset, pexact=pexact):
        # sfit = fit to the simulated data sdata
        sfit = fitter.lsqfit(data=sdata, p0=pexact, prior=prior, svdcut=..., ...)
        ... check that sfit.p values agree with pexact to within sfit.psdev ...
```

Simulated fits provide an alternative to a *bootstrap analysis* (see next section). By collecting results from many simulated fits, one can test whether or not fit results are distributed in Gaussian distributions around pexact, with widths that equal the standard deviations from the fit (fit.psdev or 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.

1.11 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:

```python
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',exp(fit.p['loga'])          # fit results for 'a' amplitudes
print 'dE',exp(fit.p['logdE'])        # 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', exp(p['loga']))    # collect bootstrap results for a[i]
    bs.append('dE', exp(p['logdE']))# 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).

1.12 Implementation

corrfitter.CorrFitter allows models to specify how many exponentials to include in the fit function (using parameters `nterm`, `nterma` and `ntermb`). If that number is less than the number of exponentials specified by the prior, the extra terms are incorporated into the fit data before fitting. The default procedure is to multiply the data by `G(t,p,N)/G(t,p,max(N,Nmax))` where: `G(p,t,N)` is the fit function with `N` terms for parameters `p` and time `t`; `N` is the number of exponentials specified in the models; `Nmax` is the number of exponentials specified in the prior; and here parameters `p` are set equal to their values in the prior (correlated `gvar.GVars`).

An alternative implementation for the data correction is to add `G(t,p,N)-G(t,p,max(N,Nmax))` to the data. This implementation is selected when parameter `ratio` in *corrfitter.CorrFitter* is set to `False`. Results are similar to the other 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, ...

1.13 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, tdata, tfit, a, b, dE, s=1.0, tp=None, othertags=None*)
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] > 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 pied 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 by *corrfitter.CorrFitter*. The keys are strings and are also used to access fit results. Any key that begins with "log" is assumed to refer to the logarithm of the parameter in question (that is, the exponential of the fit-parameter is used in the formula for `Gab(t)`.) This is 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** (*string*) – Key used to access correlator data in the input data dictionary (see *corrfitter.CorrFitter*). `data[self.datatag]` is (1-d) array containing the correlator values (`gvar.GVars`) if `data` is the input data.
>
> - **a** (string, or two-tuple of strings and/or `None`) – Key identifying the fit parameters for the source amplitudes `an` in the dictionary of priors provided by *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** (string, or two-tuple of strings and/or `None`) – Same as `self.a` but for the sinks (`bn, bo`) instead of the sources (`an, ao`).
>
> - **dE** (string, or two-tuple of strings and/or `None`) – 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** (*number or two-tuple of numbers*) – Overall factor `sn`, or two-tuple of overall factors (`sn, so`).
>
> - **tdata** (*list of integers*) – The `ts` corresponding to data entries in the input data. Note that `len(self.tdata) == len(data[self.datatag])` is required if `data` is the input data dictionary.
>
> - **tfit** (*list of integers*) – 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.
>
> - **tp** (integer or `None`) – If not `None` and 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.
>
> - **othertags** (*sequence of strings*) – List of additional data tags for data to be averaged with the `self.datatag` data before fitting.

builddata(*data*)

> Assemble fit data from dictionary `data`.

Extracts parts of array `data[self.datatag]` that are needed for the fit, as specified by `self.tp` and `self.tfit`. The entries in the (1-D) array `data[self.datatag]` are assumed to be `gvar.GVars` and correspond to the `t`''s in ``self.tdata`.

buildprior(*prior*, *nterm*)

Create fit prior by extracting relevant pieces of `prior`.

Priors for the fit parameters, as specificied by `self.a` etc., are copied from `prior` into a new dictionary for use by the fitter. If a key `"XX"` cannot be found in `prior`, the `buildprior` looks for one of `"logXX"`, `"log(XX)"`, `"sqrtXX"`, or `"sqrt(XX)"` and includes the corresponding prior instead.

The number of terms kept in each part of the fit can be specified using `nterm = (n, no)` where `n` is the number of non-oscillating terms and `no` is the number of oscillating terms. Setting `nterm = None` keeps all terms.

fitfcn(*p*, *nterm=None*, *t=None*)

Return fit function for parameters `p`.

class `corrfitter.`**Corr3**(*datatag*, *T*, *tdata*, *tfit*, *Vnn*, *a*, *b*, *dEa*, *dEb*, *sa=1.0*, *sb=1.0*, *Vno=None*, *Von=None*, *Voo=None*, *transpose_V=False*, *symmetric_V=False*, *tpa=None*, *tpb=None*, *othertags=None*)

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) + 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 `Gavb` (first term) are `Vnn[i,j]`, `an[i]`, `bn[j]`, `dEan[i]` and `dEbn[j]` where, for example:

```
dEan[0] = Ean[0] > 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** (*string*) – Tag used to label correlator in the input `gvar.dataset.Dataset`.

- **a** (string, or two-tuple of strings or `None`) – Key identifying the fit parameters for the source amplitudes `an`, for `a->V`, in the dictionary of priors provided by *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** (string, or two-tuple of strings or `None`) – Same as `self.a` except for sink amplitudes (`bn`, `bo`) for `V->b` rather than for (`an`, `ao`).

- **dEa** (string, or two-tuple of strings or `None`) – 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** (string, or two-tuple of strings or `None`) – Fit-parameter label for `V->b` intermediate-state energy differences `dEbn`, or two-tuple of labels for the differences `(dEbn,dEbo)`. 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.

- **sa** (*number, or two-tuple of numbers*) – 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.

- **sb** (*number, or two-tuple of numbers*) – Overall factor `sbn` for the non-oscillating `V->b` terms in the correlator, or two-tuple containing the overall factors `(sbn,sbo)` for the non-oscillating and oscillating terms.

- **Vnn** (string or `None`) – Fit-parameter label for the matrix of current matrix elements `Vnn[i,j]` connecting non-oscillating states. Labels that begin with "log" indicate that the corresponding matrix elements are replaced by their exponentials; these parameters are logarithms of the corresponding matrix elements, which must then be positive.

- **Vno** (string or `None`) – Fit-parameter label for the matrix of current matrix elements `Vno[i,j]` connecting non-oscillating to oscillating states. Labels that begin with "log" indicate that the corresponding matrix elements are replaced by their exponentials; these parameters are logarithms of the corresponding matrix elements, which must then be positive.

- **Von** (string or `None`) – Fit-parameter label for the matrix of current matrix elements `Von[i,j]` connecting oscillating to non-oscillating states. Labels that begin with "log" indicate that the corresponding matrix elements are replaced by their exponentials; these parameters are logarithms of the corresponding matrix elements, which must then be positive.

- **Voo** (string or `None`) – Fit-parameter label for the matrix of current matrix elements `Voo[i,j]` connecting oscillating states. Labels that begin with "log" indicate that the corresponding matrix elements are replaced by their exponentials; these parameters are logarithms of the corresponding matrix elements, which must then be positive.

- **transpose_V** (*boolean*) – If `True`, the transpose `V[j,i]` is used in place of `V[i,j]` for each current matrix element in the fit function. This is useful for doing simultaneous fits to `a->V->b` and `b->V->a`, where the current matrix elements for one are the transposes of those for the other. Default value is `False`.

- **symmetric_V** (*boolean*) – 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 with `V[i,j]=V[j,i]` and their priors are one-dimensional arrays containing only elements `V[i,j]` with `j>=i` in 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]]
```

Furthermore the matrix specified for `Von` is transposed before being used by the fitter; normally the matrix specified for `Von` is the same as the matrix specified for `Vno` when the amplitude is symmetrical. Default value is `False`.

- **tdata** (*list of integers*) – The `ts` corresponding to data entries in the input `gvar.dataset.Dataset`.

- **tfit** (*list of integers*) – 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.

- **tpa** (integer or `None`) – If not `None` and positive, the `a->V` correlator is assumed to be periodic with period `tpa`. If negative, the correlator is anti-periodic with period `-tpa`. Setting `tpa=None` implies that the correlators are not periodic.

- **tpb** (integer or `None`) – If not `None` and positive, the `V->b` correlator is assumed to be periodic with period `tpb`. If negative, the correlator is periodic with period `-tpb`. Setting `tpb=None` implies that the correlators are not periodic.

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

Extracts parts of array `data[self.datatag]` that are needed for the fit, as specified by `self.tfit`. The entries in the (1-D) array `data[self.datatag]` are assumed to be `gvar.GVars` and correspond to the `t``'s in ``self.tdata`.

buildprior(*prior*, *nterm*)
Create fit prior by extracting relevant pieces of `prior`.

Priors for the fit parameters, as specificied by `self.a` etc., are copied from `prior` into a new dictionary for use by the fitter. If a key `"XX"` cannot be found in `prior`, the `buildprior` looks for one of `"logXX"`, `"log(XX)"`, `"sqrtXX"`, or `"sqrt(XX)"` and includes the corresponding prior instead.

The number of terms kept in each part of the fit can be specified using `nterm = (n, no)` where n is the number of non-oscillating terms and `no` is the number of oscillating terms. Setting `nterm = None` keeps all terms.

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

class corrfitter.**BaseModel**(*datatag*, *othertags=[]*)
Base class for correlator models.

Derived classes must define methods `fitfcn`, `buildprior`, and `builddata`, all of which are described below. In addition they can have attributes:

datatag
corrfitter.CorrFitter builds fit data for the correlator by extracting the data in an input `gvar.dataset.Dataset` labelled by string `datatag`. This label is stored in the `BaseModel` and must be passed to its constructor.

all_datatags
Models can specify more than one set of fit data to use in fitting. The list of all the datatags used is `self.all_datatags`. The first entry is always `self.datatag`; the other entries are from `othertags`.

_abscissa
(Optional) Array of abscissa values used in plots of the data and fit corresponding to the model. Plots are not made for a model that doesn't specify this attribute.

builddata(*data*)
Construct fit data.

Format of output must be same as format for fitfcn output.

> **Parameters data** (*dictionary*) – Dataset containing correlator data (see `gvar.dataset`).

buildprior(*prior*, *nterm=None*)
Extract fit prior from `prior`; resizing as needed.

If `nterm` is not `None`, the sizes of the priors may need adjusting so that they correspond to the values specified in `nterm` (for normal and oscillating pieces).

> **Parameters**
>
> - **prior** (*dictionary*) – Dictionary containing *a priori* estimates of the fit parameters.
> - **nterm** (tuple of `None` or integers) – Restricts the number of non-oscillating terms in the fit function to `nterm[0]` and oscillating terms to `nterm[1]`. Setting either (or both) to `None` implies that all terms in the prior are used.

fitfcn(*p*, *nterm=None*)
Compute fit function fit parameters `p` using `nterm` terms. "

> **Parameters**
>
> - **p** (*dictionary*) – Dictionary of parameter values.
> - **nterm** (tuple of `None` or integers) – Restricts the number of non-oscillating terms in the fit function to `nterm[0]` and oscillating terms to `nterm[1]`. Setting either (or both) to `None` implies that all terms in the prior are used.

1.14 `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*, *svdcut=1e-15*, *tol=1e-10*, *maxit=500*, *nterm=None*, *ratio=False*, *fast=False*, *processed_data=None*)
Nonlinear least-squares fitter for a collection of correlators.

> **Parameters**
>
> - **models** (*list or other sequence*) – Sequence of correlator models, such as *corrfitter.Corr2* or *corrfitter.Corr3*, to use in fits of fit data. Individual models in the sequence can be replaced by sequences of models (and/or further sequences, recursively) for use by *corrfitter.CorrFitter.chained_lsqfit()*; such nesting is ignored by the other methods.
> - **svdcut** (number or `None`) – If svdcut is positive, eigenvalues `ev[i]` of the correlation matrix that are smaller than `svdcut*max(ev)` are replaced by `svdcut*max(ev)`. If svdcut is negative, eigenvalues less than `|svdcut|*max(ev)` are set to zero in the correlation matrix. The correlation matrix is left unchanged if `svdcut` is set equal to `None` (default).
> - **tol** (*number or tuple*) – Tolerance used in `lsqfit.nonlinear_fit()` for the least-squares fits. Use a tuple to specify separate values for the relative and absolute tolerances: `tol=(reltol, abstol)`; otherwise they are both set equal to `tol` (default=1e-10).
> - **maxit** (*integer*) – Maximum number of iterations to use in least-squares fit (default=500).
> - **nterm** (number or `None`; or two-tuple of numbers or `None`) – Number of terms fit in the non-oscillating parts of fit functions; or two-tuple of numbers indicating how many terms to

fit for each of the non-oscillating and oscillating pieces in fits. If set to `None`, the number is specified by the number of parameters in the prior.

- **ratio** (*boolean*) – If `True`, use ratio corrections for fit data when the prior specifies more terms than are used in the fit. If `False` (the default), use difference corrections (see implementation notes, above).

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`.

builddata(*data*, *prior*, *nterm=None*)
    Build fit data, corrected for marginalized terms.

buildfitfcn(*priorkeys*)
    Create fit function, with support for log-normal,... priors.

buildprior(*prior*, *nterm=None*, *fast=False*)
    Build correctly sized prior for fit from `prior`.

    Adjust the sizes of the arrays of amplitudes and energies in a copy of `prior` according to parameter `nterm`; return `prior` if both `nterm` and `self.nterm` are `None`.

chained_lsqfit(*data*, *prior*, *p0=None*, *print_fit=True*, *nterm=None*, *svdcut=None*, *tol=None*, *maxit=None*, *parallel=False*, *flat=False*, *fast=None*, ***args*)
    Compute chained least-squares fit.

    A *chained* fit fits data for each model in `self.models` sequentially, using the best-fit parameters (means and covariance matrix) of one fit to construct the prior for the fit parameters in the next fit: Correlators are fit one at a time, starting with the correlator for `self.models[0]`. The best-fit output from the fit for `self.models[i]` is fed, as a prior, into the fit for `self.models[i+1]`. The best-fit output from the last fit in the chain is the final result. Results from the individual fits can be found in dictionary `self.fit.fits`, which is indexed by the `models[i].datatags`.

    Setting parameter `parallel=True` causes parallel fits, where each model is fit separately, using the original `prior`. Parallel fits make sense when models share few or no parameters; the results from the individual fits are combined using weighted averages of the best-fit values for each parameter from every fit. Parallel fits can require larger *svd* cuts.

    Entries `self.models[i]` in the list of models can themselves be lists of models, rather than just an individual model. In such a chase, the models listed in `self.models[i]` are fit together using a parallel fit if parameter `parallel` is `False` or a chained fit otherwise. Grouping models in this ways instructs the fitter to alternate between chained and parallel fits. For example, setting

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

    with `parallel=False` causes the following chain of fits

```
m1 -> m2 -> [m3a,m3b] -> m4
```

    where: 1) the output from `m1` is used as the prior for `m2`; 2) the output from `m2` is used as the prior for for a parallel fit of `m3a` and `m3b` together; 3) the output from the parallel fit of `[m3a,m3b]` is used as the prior for `m4`; and, finally, 4) the output from `m4` is the final result of the entire chained fit.

    A slightly more complicated example is

```
models = [ m1, m2, [m3a,[m3bx,m3by]], m4]
```

    which leads to the chain of fits

> m1 -> m2 -> [m3a, m3bx -> m3by] -> m4

    where fits of `m3bx` and `m3by` are chained, in parallel with the fit to `m3a`. The fitter alternates between chained and parallel fits at each new level of grouping of models.

> **Parameters**
>
> - **data** (*dictionary*) – Input data. The `datatags` from the correlator models are used as data labels, with `data[datatag]` being a 1-d array of `gvar.GVars` corresponding to correlator values.

- **prior** (*dictionary*) – Bayesian prior for the fit parameters used in the correlator models.

- **p0** – A dictionary, indexed by parameter labels, containing initial values for the parameters in the fit. Setting `p0=None` implies that initial values are extracted from the prior. Setting `p0="filename"` causes the fitter to look in the file with name `"filename"` for initial values and to write out best-fit parameter values after the fit (for the next call to `self.lsqfit()`).

- **parallel** (*bool*) – If `True`, fit models in parallel using `prior` for each; otherwise chain the fits (default).

- **flat** (*bool*) – If `True`, flatten the list of models thereby chaining all fits (`parallel==False`) or doing them all in parallel (`parallel==True`); otherwise use `self.models` as is (default).

- **fast** – If `True`, use the smallest number of parameters needed in each fit; otherwise use all the parameters specified in `prior` in every fit. Omitting extra parameters can make fits go faster, sometimes much faster. Final results are unaffected unless `prior` contains strong correlations between different parameters, where only some of the correlated parameters are kept in individual fits. Default is `False`.

- **print_fit** – Print fit information to standard output if `True`; otherwise print nothing.

The following parameters overwrite the values specified in the [*corrfitter.CorrFitter*](#) constructor when set to anything other than `None`: nterm, svdcut, tol, and maxit. Any further keyword arguments are passed on to `lsqfit.nonlinear_fit()`, which does the fit.

collect_fitresults()
    Collect results from last fit for plots, tables etc.

    Returns

    A dictionary with one entry per correlator model, containing (`t,G,dG,Gth,dGth`) — arrays containing:

```
t       = times
G(t)    = data averages for correlator at times t
dG(t)   = uncertainties in G(t)
Gth(t)  = fit function for G(t) with best-fit parameters
dGth(t) = uncertainties in Gth(t)
```

display_plots(*save=False*)
    Show plots of data/fit-function for each correlator.

    Assumes `matplotlib` is installed (to make the plots). Plots are shown for one correlator at a time. Press key n to see the next correlator; press key p to see the previous one; press key q to quit the plot and return control to the calling program; press a digit to go directly to one of the first ten plots. Zoom, pan and save using the window controls.

    Copies of all the plots can be saved by setting parameter `save=prefix` where `prefix` is a string used to create file names: the file name for the plot corresponding to datatag k is `prefix.format(corr=k)`. It is important that the filename end with a suffix indicating the type of plot file desired: e.g., `'.pdf'`.

lsqfit(*data*, *prior*, *p0=None*, *print_fit=True*, *nterm=None*, *svdcut=None*, *tol=None*, *maxit=None*, *fast=None*, ***args*)
    Compute least-squares fit of the correlator models to data.

    Parameters

    - **data** (*dictionary*) – Input data. The `datatags` from the correlator models are used as data labels, with `data[datatag]` being a 1-d array of `gvar.GVars` corresponding to correlator values.

- **prior** (*dictionary*) – Bayesian prior for the fit parameters used in the correlator models.

- **p0** – A dictionary, indexed by parameter labels, containing initial values for the parameters in the fit. Setting `p0=None` implies that initial values are extracted from the prior. Setting `p0="filename"` causes the fitter to look in the file with name `"filename"` for initial values and to write out best-fit parameter values after the fit (for the next call to `self.lsqfit()`).

- **print_fit** – Print fit information to standard output if `True`; print nothing if `False`. Alternatively, `print_fit` can be a dictionary containing arguments for `lsqfit.nonlinear_fit.format()`.

- **fast** – If `True`, remove parameters from `prior` that are not needed by the correlator models; otherwise keep all parameters in `prior` as fit parameters (default). Ignoring extra parameters usually makes fits go faster. This has no other effect unless there are correlations between the fit parameters needed by the models and the other parameters in `prior` that are ignored.

The following parameters overwrite the values specified in the [*corrfitter.CorrFitter*](#) constructor when set to anything other than `None`: nterm, svdcut, tol, and maxit. Any further keyword arguments are passed on to `lsqfit.nonlinear_fit()`, which does the fit.

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

Simulated fit data has the same covariance matrix as `data=gvar.dataset.avg_data(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 averages of bootstrap copies of `dataset`.

The best-fit results from a fit to such simulated copies of `data` 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. More thorough testing would involve running many simulations and examining the distribution of fit parameters or functions of fit parameters around their exact values (from `pexact`). This is overkill for most problems, however.

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

Parameters

- **n** (*integer*) – Maximum number of simulated data sets made by iterator.

- **dataset** (*gvar.dataset.Dataset*) – Dataset containing Monte Carlo copies of the correlators.

- **pexact** (*dictionary of numbers*) – 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** (*positive number*) – Rescale errors in simulated data by rescale (*i.e.*, multiply covariance matrix by rescale ** 2). Default is one, which implies no rescaling.

1.15 `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.transformed_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.transformed_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.transformed_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).

1.16 Fast Fit Objects

class `corrfitter.`**`fastfit`**(*data*, *prior*, *model*, *svdcut=None*, *ratio=True*, *osc=False*)

Fast fit for the leading component of a `Corr2`.

This function class estimates `En[0]` and `an[0]*bn[0]` in a two-point correlator:

```
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 correlator is specified by `model`, and `prior` is used to remove (marginalize) all terms other than the `En[0]` term from the data. This gives a *corrected* correlator `Gc(t)` that includes uncertainties due to the terms removed. Estimates of `En[0]` 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 product of amplitutes, `an[0]*bn[0]` 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 most reliable when `Eo[0]` is smaller than `En[0]` (and so dominates at large `t`).

The results of the fast fit are stored and returned in an object of type *`corrfitter.fastfit`* with the following attributies:

E

Estimate of `En[0]` (or `Eo[0]` if `osc==True`) computed from the weighted average of `Eeff(t)` for `t`s in `model.tfit`. The prior is also included in the weighted average.

ampl

Estimate of `an[0]*bn[0]` (or `ao[0]*bo[0]` if `osc==True`) computed from the weighted average of `Aeff(t)` for `t`s in `model.tfit[1:-1]`. The prior is also included in the weighted average.

chi2

`chi[0]` is the `chi**2` for the weighted average of `Eeff(t)`s; `chi[1]` is the same for the `Aeff(t)`s.

dof

`dof[0]` is the effective number of degrees of freedom in the weighted average of `Eeff(t)`s; `dof[1]` is the same for the `Aeff(t)`s.

Q
>   `Q[0]` is the quality factor Q for the weighted average of `Eeff(t)`s; `Q[1]` is the same for the `Aeff(t)`s.

Elist
>   List of `Eeff(t)`s used in the weighted average to estimate `E`.

ampllist
>   List of `Aeff(t)`s used in the weighted average to estimate `ampl`.

> **Parameters**
> - **data** (*dictionary*) – Input data. The `datatag` from the correlator model is used as a data key, with `data[datatag]` being a 1-d array of `gvar.GVars` corresponding to the correlator values.
>
> - **prior** (*dictionary*) – Bayesian prior for the fit parameters in the correlator model.
>
> - **model** ([Corr2](#)) – Correlator model for correlator of interest. The `ts` in `model.tfit` must be consecutive.
>
> - **osc** (*Bool*) – If `True`, extract results for the leading oscillating term in the correlator (`Eo[0]`); otherwise ignore.

In addition an *svd* cut can be specified, as in *corrfitter.CorrFitter*, using parameter `svdcut`. Also the type of marginalization use can be specified with parameter `ratio` (see *corrfitter.CorrFitter*).

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:

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

import collections
import gvar as gv
import numpy as np
from corrfitter import CorrFitter, Corr2, read_dataset

def main():
    data = make_data(filename='etas-Ds.data')
    fitter = CorrFitter(models=make_models())
    p0 = None
    for N in range(2, 6):
        print(30 * '=', 'nterm =', N)
        prior = make_prior(N)
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0)
        p0 = fit.pmean
        print_results(fit)

def make_data(filename):
    """ Read data, compute averages/covariance matrix for G(t). """
```

```
    return gv.dataset.avg_data(read_dataset(filename))

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

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

def print_results(fit):
        p = fit.transformed_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-Ds.data')` from file `'etas-Ds.data'`. This file contains (among other things) 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) ]
        logdE 0   -0.87637 (28)      [ -0.7 (1.0) ]
              1     -0.330 (12)      [ -0.7 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 15/0.0)

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) ]
        logdE 0   -0.87660 (30)      [ -0.7 (1.0) ]
              1     -0.56 (28)       [ -0.7 (1.0) ]
              2     -0.92 (53)       [ -0.7 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 25/0.1)

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-07 +- 1        [  0.0 (1.0) ]
        logdE 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) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 13/0.0)

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

```
=========================== nterm = 5
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-07 +- 1      [  0.0 (1.0) ]
            4     -3e-08 +- 1      [  0.0 (1.0) ]
      logdE 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) ]
            4     -0.7 (1.0)       [ -0.7 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 14/0.1)

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

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.

ANNOTATED EXAMPLE: TRANSITION FORM FACTOR

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 could be used to calculate mixing amplitudes, such as for *B* mixing.

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

3.2 Code

The main method for this code follows the pattern described in *Basic Fits*:

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

import collections
import gvar as gv
import numpy as np
from corrfitter import CorrFitter, Corr2, Corr3, read_dataset

DISPLAYPLOTS = True          # display plots at end of fitting?
try:
    import matplotlib
except ImportError:
    DISPLAYPLOTS = False

def main():
    data = make_data('etas-Ds.data')
    fitter = CorrFitter(models=make_models())
    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)
        p0 = fit.pmean
    print_results(fit, prior, data)
    if DISPLAYPLOTS:
        fitter.display_plots()


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

The raw Monte Carlo data is in a file named ′etas-Ds.data′. 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 217 to 24. It also makes multi-term fits more stable.

The last line of main() displays plots of the fit data divided by the fit, 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.data′) reads in the Monte Carlo data, averages it, and formats it for use by *corrfitter.CorrFitter*. The data file (′eta-Ds.data′) contains 225 lines, each with 64 numbers on it, of the form:

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

Each of these lines is a single Monte Carlo estimate for the η_s correlator on a lattice with 64 lattice points in the t direction; there are 225 Monte Carlo estimates in all. The same file also contains 225 lines describing the D_s meson correlator:

```
Ds    0.230503    0.0445531    0.00895383 ...
Ds    0.230947    0.0447479    0.00904294 ...
...
```

And it contains 225 lines each giving the 3-point amplitude for $\eta_s \rightarrow D_s$ where the source and sink are separated by 15 and 16 time steps on the lattice:

```
3ptT15  4.63494e-10    1.11333e-09    2.46993e-09 ...
3ptT15  4.85637e-10    1.15445e-09    2.59419e-09 ...
...

3ptT16  1.42457e-10    3.27314e-10    7.61508e-10 ...
3ptT16  1.47582e-10    3.4255e-10     7.95205e-10 ...
...
```

The first, second, third, *etc.* lines for each label come from the first, second, third, *etc.* Monte Carlo iterations, respectively; this synchronization allows the code to compute correlations between different types of data.

Function corrfitter.read_dataset() is designed to read files in this format (among others). We use it to read the data, and gvar.dataset.avg_data() to compute the means and covariance matrix of the data:

```
def make_data(datafile):
    """ Read data from datafile and average it. """
```

This routine returns a dictionary whose keys are the strings used to label the individual lines in `etas-Ds.data`: for example,

```
>>> data = make_data('etas-Ds.data')
>>> 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 the Monte Carlo estimates (mean and covariance) for the corresponding correlator. This is the format needed by *corrfitter.CorrFitter*.

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 = [
        Corr2(
            datatag='etas',
            tp=tp, tdata=range(tp), tfit=range(tmin, tp-tmin),
            a='etas:a', b='etas:a', dE='etas:dE'
            ),

        Corr2(
            datatag='Ds',
            tp=tp, tdata=range(tp), tfit=range(tmin, tp-tmin),
            a=('Ds:a', 'Dso:a'), b=('Ds:a', 'Dso:a'),
            dE=('Ds:dE', 'Dso:dE'), s=(1., -1.)
            ),

        Corr3(
            datatag='3ptT15', tdata=range(16), T=15, tfit=range(tmin, 16-tmin),
            a='etas:a', dEa='etas:dE', tpa=tp,
            b=('Ds:a', 'Dso:a'), dEb=('Ds:dE', 'Dso:dE'), tpb=tp, sb=(1, -1.),
            Vnn='Vnn', Vno='Vno'
            ),

        Corr3(
            datatag='3ptT16', tdata=range(17), T=16, tfit=range(tmin, 17-tmin),
            a='etas:a', dEa='etas:dE', tpa=tp,
            b=('Ds:a', 'Dso:a'), dEb=('Ds:dE', 'Dso:dE'), tpb=tp, sb=(1, -1.),
            Vnn='Vnn', Vno='Vno'
            )
        ]
```

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:

```python
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)"]])
```

Parameter N specifies how many terms are kept in the fit functions. The priors are specified in a dictionary prior. Each entry is an array, of length N, with one entry for each term. Each entry is a Gaussian random variable, specified by 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,

```python
>>> 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:

```python
def print_results(fit, prior, data):
    """ Report best-fit results. """
    print('Fit results:')
    p = fit.transformed_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].fmt())
    print('  etas->V->Dso =', Vno[0, 0].fmt())

    # 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))
```

The best-fit parameter values are stored in dictionary p=fit.transformed_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: [0.41619(12) 1.007(89) 1.43(34)]
  aetas: [0.21834(16) 0.170(74) 0.30(12)]

  EDs: [1.20166(16) 1.704(17) 2.29(20)]
  aDs: [0.21466(20) 0.275(20) 0.52(20)]

  EDso: [1.442(16) 1.65(11) 2.17(44)]
  aDso: [0.0634(90) 0.080(26) 0.116(93)]

  etas->V->Ds  = 0.76725(76)
  etas->V->Dso = -0.793(92)
```

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: 0.41619(12)
               mDs: 1.20166(16)
         mDso-mDs: 0.240(16)
               Vnn: 0.76725(76)
               Vno: -0.793(92)


Partial % Errors:
                 metas      mDs  mDso-mDs      Vnn      Vno
--------------------------------------------------------------
   statistics:    0.03     0.01      4.51     0.09     8.60
 log(etas:a):     0.00     0.00      0.11     0.01     0.39
log(etas:dE):     0.00     0.00      0.06     0.01     0.38
   log(Ds:a):     0.00     0.00      0.53     0.02     0.96
   log(Ds:dE):    0.00     0.00      0.44     0.02     0.59
  log(Dso:a):     0.00     0.00      1.10     0.01     3.85
 log(Dso:dE):     0.00     0.00      1.14     0.01     5.66
         Vnn:     0.00     0.00      0.58     0.03     1.03
         Vno:     0.00     0.00      4.25     0.01     3.39
         svd:     0.00     0.00      0.00     0.00     0.00
--------------------------------------------------------------
       total:     0.03     0.01      6.46     0.10    11.61
```

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] = 7.4e+03 [69]    Q = 0    logGBF = -2.5405e+05

Parameters:
  log(etas:a) 0   -1.38766 (30)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.80364 (14)      [ -0.92 (50) ]
    log(Ds:a) 0   -1.35559 (20)      [ -1.2 (1.0) ]
   log(Ds:dE) 0   0.220836 (54)      [  0.18 (17) ]
   log(Dso:a) 0    -1.7014 (16)      [ -2.3 (1.0) ]
  log(Dso:dE) 0    0.54320 (39)      [  0.41 (24) ]
        Vnn 0,0    0.74220 (23)      [  0.0 (1.0) ]
        Vno 0,0    -1.0474 (21)      [  0.0 (1.0) ]  *

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 23/0.1)

============================== nterm = 2
Least Square Fit:
  chi2/dof [dof] = 3 [69]    Q = 6e-16    logGBF = 1531.1

Parameters:
  log(etas:a) 0   -1.52065 (64)      [ -1.2 (1.0) ]
              1    -1.300 (15)       [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87643 (26)      [ -0.92 (50) ]
              1    -0.331 (10)       [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53878 (66)      [ -1.2 (1.0) ]
              1    -1.0798 (68)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18357 (11)      [  0.18 (17) ]
              1    -0.5880 (55)      [ -0.7 (1.0) ]
   log(Dso:a) 0    -2.6014 (75)      [ -2.3 (1.0) ]
              1    -1.266 (76)       [ -2.3 (1.0) ]  *
  log(Dso:dE) 0    0.3735 (14)       [  0.41 (24) ]
              1    -0.323 (41)       [ -0.7 (1.0) ]
        Vnn 0,0    0.76314 (30)      [  0.0 (1.0) ]
            0,1    -0.4536 (52)      [  0.0 (1.0) ]
            1,0    0.0799 (73)       [  0.0 (1.0) ]
            1,1    -0.25 (76)        [  0.0 (1.0) ]
        Vno 0,0    -0.6796 (76)      [  0.0 (1.0) ]
            0,1    0.946 (66)        [  0.0 (1.0) ]
            1,0    -1.00 (13)        [  0.0 (1.0) ]
            1,1    0.06 (1.00)       [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 77/0.6)

============================== nterm = 3
Least Square Fit:
  chi2/dof [dof] = 0.7 [69]    Q = 0.97    logGBF = 1601.6

Parameters:
  log(etas:a) 0    -1.52172 (73)       [ -1.2 (1.0) ]
              1     -1.81 (47)         [ -1.2 (1.0) ]
              2     -1.13 (20)         [ -1.2 (1.0) ]
```

```
 log(etas:dE) 0      -0.87662 (28)        [ -0.92 (50) ]
              1        -0.54 (17)          [ -0.7 (1.0) ]
              2        -0.82 (44)          [ -0.7 (1.0) ]
   log(Ds:a) 0      -1.53871 (91)          [ -1.2 (1.0) ]
              1        -1.290 (73)          [ -1.2 (1.0) ]
              2        -0.58 (33)          [ -1.2 (1.0) ]
  log(Ds:dE) 0       0.18370 (13)          [  0.18 (17) ]
              1        -0.690 (34)          [ -0.7 (1.0) ]
              2        -0.49 (29)          [ -0.7 (1.0) ]
  log(Dso:a) 0        -2.76 (14)           [ -2.3 (1.0) ]
              1        -2.53 (34)           [ -2.3 (1.0) ]
              2        -2.11 (77)           [ -2.3 (1.0) ]
 log(Dso:dE) 0       0.366 (11)            [  0.41 (24) ]
              1        -1.59 (54)           [ -0.7 (1.0) ]
              2        -0.64 (76)           [ -0.7 (1.0) ]
        Vnn 0,0      0.76725 (76)          [  0.0 (1.0) ]
            0,1       -0.490 (32)          [  0.0 (1.0) ]
            0,2        0.51 (51)           [  0.0 (1.0) ]
            1,0        0.049 (37)          [  0.0 (1.0) ]
            1,1        0.24 (68)           [  0.0 (1.0) ]
            1,2      -0.005 (0.995)        [  0.0 (1.0) ]
            2,0        -0.06 (14)          [  0.0 (1.0) ]
            2,1        0.05 (1.00)         [  0.0 (1.0) ]
            2,2     0.0004 (1.0000)        [  0.0 (1.0) ]
        Vno 0,0       -0.793 (93)          [  0.0 (1.0) ]
            0,1        0.26 (33)           [  0.0 (1.0) ]
            0,2       -0.006 (841)         [  0.0 (1.0) ]
            1,0        0.39 (45)           [  0.0 (1.0) ]
            1,1        0.20 (95)           [  0.0 (1.0) ]
            1,2      -0.006 (0.999)        [  0.0 (1.0) ]
            2,0        -0.17 (92)          [  0.0 (1.0) ]
            2,1       0.006 (0.999)        [  0.0 (1.0) ]
            2,2      0.0007 (1.0000)       [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10     (itns/time = 76/0.7)


============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.7 [69]    Q = 0.97    logGBF = 1602.1

Parameters:
  log(etas:a) 0      -1.52170 (73)         [ -1.2 (1.0) ]
              1        -1.77 (43)          [ -1.2 (1.0) ]
              2        -1.22 (42)          [ -1.2 (1.0) ]
              3        -1.31 (95)          [ -1.2 (1.0) ]
 log(etas:dE) 0      -0.87661 (28)         [ -0.92 (50) ]
              1        -0.53 (15)          [ -0.7 (1.0) ]
              2        -0.85 (62)          [ -0.7 (1.0) ]
              3        -0.62 (97)          [ -0.7 (1.0) ]
   log(Ds:a) 0      -1.53869 (91)          [ -1.2 (1.0) ]
              1        -1.290 (74)          [ -1.2 (1.0) ]
              2        -0.65 (39)          [ -1.2 (1.0) ]
              3        -1.11 (99)          [ -1.2 (1.0) ]
  log(Ds:dE) 0       0.18370 (13)          [  0.18 (17) ]
              1        -0.689 (35)          [ -0.7 (1.0) ]
              2        -0.53 (32)          [ -0.7 (1.0) ]
              3        -0.77 (99)          [ -0.7 (1.0) ]
```

```
  log(Dso:a) 0          -2.76 (14)          [ -2.3 (1.0) ]
             1          -2.53 (33)          [ -2.3 (1.0) ]
             2          -2.15 (80)          [ -2.3 (1.0) ]
             3          -2.3 (1.0)          [ -2.3 (1.0) ]
 log(Dso:dE) 0          0.366 (11)          [  0.41 (24) ]
             1          -1.59 (53)          [ -0.7 (1.0) ]
             2          -0.65 (74)          [ -0.7 (1.0) ]
             3          -0.7 (1.0)          [ -0.7 (1.0) ]
      Vnn 0,0          0.76725 (76)         [  0.0 (1.0) ]
          0,1          -0.492 (33)          [  0.0 (1.0) ]
          0,2            0.50 (51)          [  0.0 (1.0) ]
          0,3           0.06 (1.00)         [  0.0 (1.0) ]
          1,0           0.050 (42)          [  0.0 (1.0) ]
          1,1            0.25 (70)          [  0.0 (1.0) ]
          1,2         -0.005 (0.995)        [  0.0 (1.0) ]
          1,3        -0.0007 (1.0000)       [  0.0 (1.0) ]
          2,0           -0.07 (19)          [  0.0 (1.0) ]
          2,1           0.05 (1.00)         [  0.0 (1.0) ]
          2,2         0.0004 (1.0000)       [  0.0 (1.0) ]
          2,3           -1e-05 +- 1         [  0.0 (1.0) ]
          3,0          0.002 (977)          [  0.0 (1.0) ]
          3,1          0.003 (1.000)        [  0.0 (1.0) ]
          3,2            3e-05 +- 1         [  0.0 (1.0) ]
          3,3            2e-07 +- 1         [  0.0 (1.0) ]
      Vno 0,0          -0.793 (92)          [  0.0 (1.0) ]
          0,1            0.25 (33)          [  0.0 (1.0) ]
          0,2          0.005 (845)          [  0.0 (1.0) ]
          0,3           0.01 (1.00)         [  0.0 (1.0) ]
          1,0            0.38 (45)          [  0.0 (1.0) ]
          1,1            0.22 (95)          [  0.0 (1.0) ]
          1,2         -0.004 (0.999)        [  0.0 (1.0) ]
          1,3        -0.0009 (1.0000)       [  0.0 (1.0) ]
          2,0           -0.17 (93)          [  0.0 (1.0) ]
          2,1          0.005 (0.999)        [  0.0 (1.0) ]
          2,2         0.0007 (1.0000)       [  0.0 (1.0) ]
          2,3           -1e-05 +- 1         [  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            1e-05 +- 1         [  0.0 (1.0) ]
          3,3            4e-07 +- 1         [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 24/0.4)

Fit results:
  Eetas: [0.41619(12) 1.007(89) 1.43(34)]
  aetas: [0.21834(16) 0.170(74) 0.30(12)]

  EDs: [1.20166(16) 1.704(17) 2.29(20)]
  aDs: [0.21466(20) 0.275(20) 0.52(20)]

  EDso: [1.442(16) 1.65(11) 2.17(44)]
  aDso: [0.0634(90) 0.080(26) 0.116(93)]

  etas->V->Ds  = 0.76725(76)
  etas->V->Dso = -0.793(92)

Values:
```

```
            metas: 0.41619(12)
              mDs: 1.20166(16)
         mDso-mDs: 0.240(16)
              Vnn: 0.76725(76)
              Vno: -0.793(92)

Partial % Errors:
                   metas      mDs  mDso-mDs      Vnn       Vno
        ---------------------------------------------------------
    statistics:     0.03     0.01      4.51     0.09      8.60
   log(etas:a):     0.00     0.00      0.11     0.01      0.39
  log(etas:dE):     0.00     0.00      0.06     0.01      0.38
     log(Ds:a):     0.00     0.00      0.53     0.02      0.96
    log(Ds:dE):     0.00     0.00      0.44     0.02      0.59
    log(Dso:a):     0.00     0.00      1.10     0.01      3.85
   log(Dso:dE):     0.00     0.00      1.14     0.01      5.66
           Vnn:     0.00     0.00      0.58     0.03      1.03
           Vno:     0.00     0.00      4.25     0.01      3.39
           svd:     0.00     0.00      0.00     0.00      0.00
        ---------------------------------------------------------
         total:     0.03     0.01      6.46     0.10     11.61
```

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/dofs` that are significantly larger than one.

- Fits with three terms work well, and adding futher terms has almost no impact. The `chi**2` 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`:
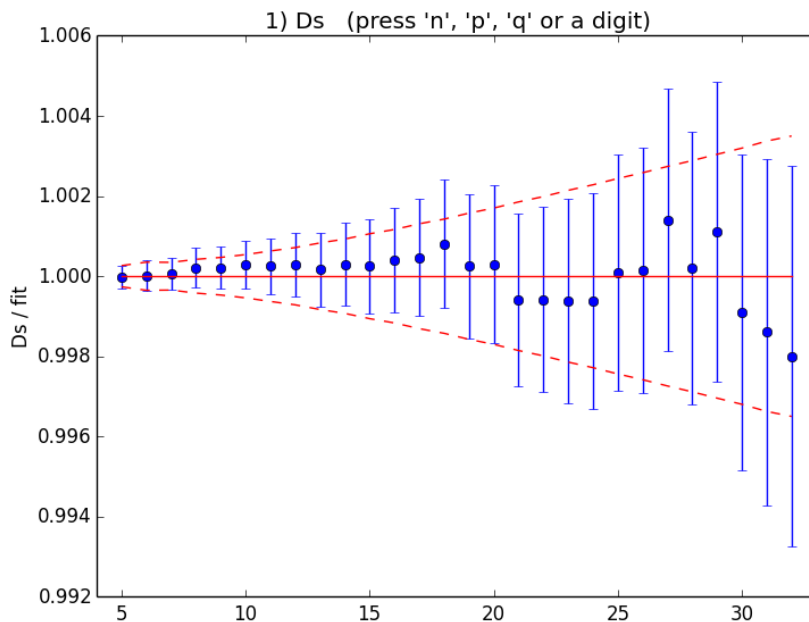
The points with error bars are the correlator data points; the fit result is 1.0 in this plot, of course, and the dashed lines show the uncertainty in the fit function evaluated with the best-fit parameters. Fit and data agree to within errors. Note how the fit-function errors (the dashed lines) track 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`.

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:

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

The first modification (#1) is in the definition of `fitter`, where we add an extra argument to tell *corrfitter.CorrFitter* what kind of marginalization to use (that is, not the ratio method). The second modification (#2) limits the fits to `N=1, 2`, because that is all that will be needed to get good values for the leading term. The third modification (#3) sets the prior to eight terms, no matter what value `N` has. The last (#4) 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 output shows that results for the leading term have converged by `N=2` (and even `N=1` isn't so bad):

```
============================ nterm = 1
Least Square Fit:
  chi2/dof [dof] = 0.98 [69]    Q = 0.53    logGBF = 1586.4

Parameters:
  log(etas:a) 0    -1.52151 (78)       [ -1.2 (1.0) ]
 log(etas:dE) 0    -0.87662 (29)       [ -0.92 (50) ]
   log(Ds:a) 0      -1.5387 (10)       [ -1.2 (1.0) ]
  log(Ds:dE) 0      0.18372 (14)       [  0.18 (17) ]
  log(Dso:a) 0       -2.628 (25)       [ -2.3 (1.0) ]
 log(Dso:dE) 0       0.3738 (32)       [  0.41 (24) ]
       Vnn 0,0      0.76533 (60)       [  0.0 (1.0) ]
       Vno 0,0       -0.710 (11)       [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 9/0.1)


============================ nterm = 2
Least Square Fit:
  chi2/dof [dof] = 0.71 [69]    Q = 0.97    logGBF = 1602.3

Parameters:
```

```
   log(etas:a) 0   -1.52169 (72)      [ -1.2  (1.0) ]
               1      -1.81 (52)      [ -1.2  (1.0) ]
  log(etas:dE) 0   -0.87660 (28)      [ -0.92 (50) ]
               1      -0.54 (17)      [ -0.7  (1.0) ]
     log(Ds:a) 0   -1.53882 (88)      [ -1.2  (1.0) ]
               1     -1.339 (75)      [ -1.2  (1.0) ]
    log(Ds:dE) 0    0.18370 (13)      [  0.18 (17) ]
               1     -0.711 (34)      [ -0.7  (1.0) ]
    log(Dso:a) 0     -2.746 (92)      [ -2.3  (1.0) ]
               1      -2.44 (10)      [ -2.3  (1.0) ]
   log(Dso:dE) 0     0.3661 (74)      [  0.41 (24) ]
               1      -1.45 (24)      [ -0.7  (1.0) ]
        Vnn 0,0     0.76759 (74)      [  0.0  (1.0) ]
            0,1      -0.488 (35)      [  0.0  (1.0) ]
            1,0       0.039 (51)      [  0.0  (1.0) ]
            1,1        0.63 (74)      [  0.0  (1.0) ]
        Vno 0,0      -0.774 (42)      [  0.0  (1.0) ]
            0,1        0.25 (16)      [  0.0  (1.0) ]
            1,0        0.34 (43)      [  0.0  (1.0) ]
            1,1        0.29 (95)      [  0.0  (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 20/0.2)

Fit results:
  Eetas: [0.41619(12) 1.00(10)]
  aetas: [0.21834(16) 0.164(85)]

  EDs: [1.20165(15) 1.693(17)]
  aDs: [0.21463(19) 0.262(20)]

  EDso: [1.442(11) 1.676(61)]
  aDso: [0.0642(59) 0.0872(90)]

  etas->V->Ds  = 0.76759(74)
  etas->V->Dso = -0.774(42)

Values:
            metas: 0.41619(12)
              mDs: 1.20165(15)
         mDso-mDs: 0.241(11)
              Vnn: 0.76759(74)
              Vno: -0.774(42)

Partial % Errors:
                metas       mDs  mDso-mDs       Vnn       Vno
  -------------------------------------------------------------
   statistics:    0.03      0.01      3.69      0.09      4.53
  log(etas:a):    0.00      0.00      0.10      0.01      0.54
 log(etas:dE):    0.00      0.00      0.06      0.00      0.42
    log(Ds:a):    0.00      0.00      0.34      0.01      0.47
   log(Ds:dE):    0.00      0.00      0.52      0.02      0.47
   log(Dso:a):    0.00      0.00      0.40      0.00      1.37
  log(Dso:dE):    0.00      0.00      0.54      0.00      1.94
         Vnn:    0.00      0.00      1.03      0.03      0.25
         Vno:    0.00      0.00      2.10      0.02      1.28
         svd:    0.00      0.00      0.00      0.00      0.00
  -------------------------------------------------------------
```

```
        total:      0.03      0.01      4.47      0.10      5.37
```

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,

```
Values:
            metas: 0.41619(12)
              mDs: 1.20156(17)
          mDso-mDs: 0.2554(41)
              Vnn: 0.7676(12)
              Vno: -0.754(26)
```

We obtain more or less the same results,

```
Values:
            metas: 0.41619(11)
              mDs: 1.20156(15)
          mDso-mDs: 0.2576(27)
              Vnn: 0.76666(67)
              Vno: -0.747(15)
```

if we polish the final results from the chained fit using a final call to `fitter.lsqfit` (see *Faster Fits — Chained Fits*):

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

Another variation is to replace the last line (`return models`) in `make_models()` by:

```
return [models[:2]] + models[2:]
```

This causes the two 2-point correlators (`models[:2]`) to be fit in parallel, which makes sense since they share no parameters. The result of the (parallel) fit of the 2-point correlators is used as a prior for the chained fits of the 3-point correlators (`models[2:]`). The fit results are mostly unchanged, although the polishing fit is significantly faster (more than 2x) in this case:

```
Values:
            metas: 0.41619(11)
              mDs: 1.20156(15)
          mDso-mDs: 0.2576(27)
              Vnn: 0.76666(67)
              Vno: -0.747(15)
```

3.6 Test the Analysis

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

```
def test_fit(fitter, datafile):
    """ Test the fit with simulated data """
    gv.ranseed((1487942813, 775399747, 906327435))
    print('\nRandom seed:', gv.ranseed.seed)
    dataset = read_dataset(datafile)
    pexact = fitter.fit.pmean
    prior = fitter.fit.prior
```

```
    for sdata in fitter.simulated_data_iter(n=2, dataset=dataset, pexact=pexact):
        print('\n=============================== simulation')
        sfit = fitter.lsqfit(data=sdata, prior=prior, p0=pexact)
        diff = []
        # check chi**2 for leading parameters
        for k in sfit.p:
            diff.append(sfit.p[k].flat[0] - pexact[k].flat[0])
        chi2diff = gv.chi2(diff)
        print(
            'Leading parameter chi2/dof [dof] = %.2f' %
            (chi2diff / chi2diff.dof),
            '[%d]' % chi2diff.dof,
            '  Q = %.1f' % chi2diff.Q
```

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 prints out each fit, and for each it computes the `chi**2` of the difference between the leading parameters and `pexact`. The output is:

```
Random seed: (1487942813, 775399747, 906327435)

=============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.54 [69]    Q = 1    logGBF = 1606.7

Parameters:
  log(etas:a) 0     -1.52148 (74)       [ -1.2 (1.0) ]
              1        -2.01 (56)        [ -1.2 (1.0) ]
              2        -1.33 (57)        [ -1.2 (1.0) ]
              3        -1.10 (67)        [ -1.2 (1.0) ]
 log(etas:dE) 0     -0.87641 (28)       [ -0.92 (50) ]
              1        -0.61 (20)        [ -0.7 (1.0) ]
              2        -1.00 (57)        [ -0.7 (1.0) ]
              3        -0.84 (90)        [ -0.7 (1.0) ]
   log(Ds:a) 0     -1.53757 (91)        [ -1.2 (1.0) ]
              1        -1.32 (10)        [ -1.2 (1.0) ]
              2        -0.81 (34)        [ -1.2 (1.0) ]
              3        -1.13 (98)        [ -1.2 (1.0) ]
  log(Ds:dE) 0      0.18390 (13)        [  0.18 (17) ]
              1       -0.696 (44)        [ -0.7 (1.0) ]
              2        -0.71 (34)        [ -0.7 (1.0) ]
              3        -0.75 (99)        [ -0.7 (1.0) ]
  log(Dso:a) 0       -2.713 (86)        [ -2.3 (1.0) ]
              1        -2.38 (23)        [ -2.3 (1.0) ]
              2        -2.39 (95)        [ -2.3 (1.0) ]
              3        -2.3 (1.0)        [ -2.3 (1.0) ]
 log(Dso:dE) 0       0.3682 (73)        [  0.41 (24) ]
              1        -1.33 (32)        [ -0.7 (1.0) ]
              2        -0.54 (91)        [ -0.7 (1.0) ]
              3        -0.7 (1.0)        [ -0.7 (1.0) ]
       Vnn 0,0      0.76817 (91)        [  0.0 (1.0) ]
           0,1       -0.516 (38)        [  0.0 (1.0) ]
           0,2         0.38 (42)        [  0.0 (1.0) ]
           0,3         0.06 (99)        [  0.0 (1.0) ]
           1,0        0.016 (77)        [  0.0 (1.0) ]
           1,1        -0.21 (74)        [  0.0 (1.0) ]
           1,2       -0.005 (990)       [  0.0 (1.0) ]
           1,3     0.0009 (0.9999)      [  0.0 (1.0) ]
           2,0         0.12 (34)        [  0.0 (1.0) ]
```

```
            2,1        0.003 (993)        [  0.0 (1.0) ]
            2,2      -0.002 (1.000)       [  0.0 (1.0) ]
            2,3         3e-06 +- 1        [  0.0 (1.0) ]
            3,0        -0.52 (88)         [  0.0 (1.0) ]
            3,1       0.003 (1.000)       [  0.0 (1.0) ]
            3,2        -4e-06 +- 1        [  0.0 (1.0) ]
            3,3        -2e-06 +- 1        [  0.0 (1.0) ]
     Vno 0,0        -0.752 (39)         [  0.0 (1.0) ]
         0,1         0.26 (20)          [  0.0 (1.0) ]
         0,2         0.12 (96)          [  0.0 (1.0) ]
         0,3        0.01 (1.00)         [  0.0 (1.0) ]
         1,0        -0.08 (49)          [  0.0 (1.0) ]
         1,1        -0.35 (95)          [  0.0 (1.0) ]
         1,2       -0.01 (1.00)         [  0.0 (1.0) ]
         1,3     -0.0005 (1.0000)       [  0.0 (1.0) ]
         2,0         0.57 (87)          [  0.0 (1.0) ]
         2,1        0.02 (1.00)         [  0.0 (1.0) ]
         2,2       -0.001 (1.000)       [  0.0 (1.0) ]
         2,3        -7e-05 +- 1         [  0.0 (1.0) ]
         3,0         0.11 (99)          [  0.0 (1.0) ]
         3,1        0.01 (1.00)         [  0.0 (1.0) ]
         3,2        -1e-05 +- 1         [  0.0 (1.0) ]
         3,3        -2e-06 +- 1         [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 54/0.7)

Leading parameter chi2/dof [dof] = 0.91 [8]   Q = 0.5

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.81 [69]   Q = 0.88   logGBF = 1596.6

Parameters:
  log(etas:a) 0    -1.52324 (74)        [ -1.2 (1.0) ]
              1       -1.84 (24)        [ -1.2 (1.0) ]
              2       -0.86 (52)        [ -1.2 (1.0) ]
              3       -1.08 (97)        [ -1.2 (1.0) ]
 log(etas:dE) 0    -0.87722 (28)        [ -0.92 (50) ]
              1       -0.60 (11)        [ -0.7 (1.0) ]
              2       -0.46 (40)        [ -0.7 (1.0) ]
              3       -0.78 (98)        [ -0.7 (1.0) ]
    log(Ds:a) 0    -1.54023 (96)        [ -1.2 (1.0) ]
              1       -1.41 (11)        [ -1.2 (1.0) ]
              2       -0.77 (31)        [ -1.2 (1.0) ]
              3       -1.10 (98)        [ -1.2 (1.0) ]
   log(Ds:dE) 0     0.18346 (13)        [  0.18 (17) ]
              1      -0.750 (51)        [ -0.7 (1.0) ]
              2       -0.70 (30)        [ -0.7 (1.0) ]
              3       -0.77 (99)        [ -0.7 (1.0) ]
  log(Dso:a) 0      -2.710 (92)         [ -2.3 (1.0) ]
              1       -2.46 (26)        [ -2.3 (1.0) ]
              2       -2.35 (90)        [ -2.3 (1.0) ]
              3       -2.3 (1.0)        [ -2.3 (1.0) ]
  log(Dso:dE) 0     0.3703 (79)         [  0.41 (24) ]
              1      -1.46 (36)         [ -0.7 (1.0) ]
              2      -0.59 (84)         [ -0.7 (1.0) ]
              3      -0.7 (1.0)         [ -0.7 (1.0) ]
```

```
        Vnn 0,0      0.76840 (85)        [   0.0 (1.0) ]
            0,1       -0.474 (38)        [   0.0 (1.0) ]
            0,2        0.18 (32)         [   0.0 (1.0) ]
            0,3        0.03 (99)         [   0.0 (1.0) ]
            1,0       0.002 (28)         [   0.0 (1.0) ]
            1,1        0.72 (58)         [   0.0 (1.0) ]
            1,2        0.12 (98)         [   0.0 (1.0) ]
            1,3     0.008 (1.000)        [   0.0 (1.0) ]
            2,0        0.13 (20)         [   0.0 (1.0) ]
            2,1       0.01 (1.00)        [   0.0 (1.0) ]
            2,2    0.0009 (1.0000)       [   0.0 (1.0) ]
            2,3       7e-05 +- 1         [   0.0 (1.0) ]
            3,0        0.07 (99)         [   0.0 (1.0) ]
            3,1      0.001 (1.000)       [   0.0 (1.0) ]
            3,2       2e-05 +- 1         [   0.0 (1.0) ]
            3,3       6e-07 +- 1         [   0.0 (1.0) ]
        Vno 0,0       -0.797 (68)        [   0.0 (1.0) ]
            0,1        0.35 (24)         [   0.0 (1.0) ]
            0,2       -0.04 (94)         [   0.0 (1.0) ]
            0,3     -0.002 (0.999)       [   0.0 (1.0) ]
            1,0        0.19 (37)         [   0.0 (1.0) ]
            1,1        0.34 (94)         [   0.0 (1.0) ]
            1,2       0.02 (1.00)        [   0.0 (1.0) ]
            1,3      0.002 (1.000)       [   0.0 (1.0) ]
            2,0       -0.21 (95)         [   0.0 (1.0) ]
            2,1      -0.02 (1.00)        [   0.0 (1.0) ]
            2,2    0.0002 (1.0000)       [   0.0 (1.0) ]
            2,3       2e-05 +- 1         [   0.0 (1.0) ]
            3,0      -0.02 (1.00)        [   0.0 (1.0) ]
            3,1     -0.002 (1.000)       [   0.0 (1.0) ]
            3,2      -4e-06 +- 1         [   0.0 (1.0) ]
            3,3       2e-07 +- 1         [   0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10     (itns/time = 48/0.7)

Leading parameter chi2/dof [dof] = 1.19 [8]   Q = 0.3
```

This shows that the fit is working well, at least for the leading parameter for each key.

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(data=sdata, prior=prior, p0=pexact, nterm=(2,2))
```

Running this code gives:

```
Random seed: (623738625, 435880512, 1745400596)

=========================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.75 [69]    Q = 0.94    logGBF = 1611.8

Parameters:
  log(etas:a) 0   -1.52071 (66)       [ -1.2 (1.0) ]
            1      -1.774 (19)        [ -1.2 (1.0) ]
 log(etas:dE) 0  -0.87629 (27)        [ -0.92 (50) ]
            1      -0.510 (16)        [ -0.7 (1.0) ]
    log(Ds:a) 0  -1.53874 (69)        [ -1.2 (1.0) ]
```

```
              1      -1.3385 (55)        [ -1.2 (1.0) ]
   log(Ds:dE) 0      0.18370 (11)        [  0.18 (17) ]
              1      -0.7106 (58)        [ -0.7 (1.0) ]
  log(Dso:a) 0        -2.776 (31)        [ -2.3 (1.0) ]
              1        -2.417 (35)        [ -2.3 (1.0) ]
 log(Dso:dE) 0        0.3619 (31)        [  0.41 (24) ]
              1        -1.435 (86)        [ -0.7 (1.0) ]
       Vnn 0,0       0.76744 (32)        [  0.0 (1.0) ]
           0,1       -0.4826 (48)        [  0.0 (1.0) ]
           1,0        0.0336 (76)        [  0.0 (1.0) ]
           1,1          1.20 (57)        [  0.0 (1.0) ]   *
       Vno 0,0        -0.747 (24)        [  0.0 (1.0) ]
           0,1         0.179 (47)        [  0.0 (1.0) ]
           1,0          0.58 (17)        [  0.0 (1.0) ]
           1,1         -0.01 (94)        [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 8/0.1)

Leading parameter chi2/dof [dof] = 0.96 [8]   Q = 0.5


============================ simulation
Least Square Fit:
  chi2/dof [dof] = 0.91 [69]    Q = 0.68    logGBF = 1606.2

Parameters:
  log(etas:a) 0   -1.52175 (66)        [ -1.2 (1.0) ]
              1      -1.791 (19)        [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87645 (27)        [ -0.92 (50) ]
              1      -0.525 (16)        [ -0.7 (1.0) ]
   log(Ds:a) 0   -1.53891 (69)        [ -1.2 (1.0) ]
              1      -1.3413 (54)        [ -1.2 (1.0) ]
  log(Ds:dE) 0      0.18376 (11)        [  0.18 (17) ]
              1      -0.7129 (58)        [ -0.7 (1.0) ]
  log(Dso:a) 0        -2.776 (32)        [ -2.3 (1.0) ]
              1        -2.425 (35)        [ -2.3 (1.0) ]
 log(Dso:dE) 0        0.3627 (32)        [  0.41 (24) ]
              1        -1.462 (87)        [ -0.7 (1.0) ]
       Vnn 0,0       0.76796 (33)        [  0.0 (1.0) ]
           0,1       -0.4868 (48)        [  0.0 (1.0) ]
           1,0        0.0335 (74)        [  0.0 (1.0) ]
           1,1          0.38 (56)        [  0.0 (1.0) ]
       Vno 0,0        -0.761 (27)        [  0.0 (1.0) ]
           0,1         0.200 (49)        [  0.0 (1.0) ]
           1,0          0.54 (17)        [  0.0 (1.0) ]
           1,1          0.16 (93)        [  0.0 (1.0) ]

Settings:
  svdcut/n = 1e-15/0    reltol/abstol = 1e-10/1e-10    (itns/time = 8/0.1)

Leading parameter chi2/dof [dof] = 0.69 [8]   Q = 0.7
```

This is also fine and confirms that `nterm=(2,2)` marginalized fits are a useful, faster substitute for full fits. Indeed the simulation suggests that the marginalized fit is somewhat more accurate than the original fit for the oscillating-state parameters (`Vno`, `log(Dso:a)`, `log(Dso:dE)` — compare the simulated results with the `nterm=4` results from the original fit, as these were used to define `pexact`).

ANNOTATED EXAMPLE: MATRIX CORRELATOR

4.1 Introduction

Matrix correlators, built from multiple sources and sinks, greatly improve results for the excited states in the correlators. Here we analyze η_b correlators using 4 sources/sinks using a prior designed by *corrfitter.EigenBasis*.

A major challenge when studying excited states with multi-source fits is the appearance in the fit of spurious states, with amplitudes that are essentially zero, between the real states in the correlator. These states contribute little to the correlators, because of their vanishing amplitudes, but they usually have a strong negative impact on the errors of states just below them and above them. They also can cause the fitter to stall, taking 1000s of iterations to change nothing other than the parameters of the spurious state. *corrfitter.EigenBasis* addresses this problem by creating a prior that discourages spurious states. It encodes the fact that only a small number of states still couple to the matrix correlator by moderate values of t, and therefore, that there exist linear combinations of the sources that couple strongly to individual low-lying states but not the others. This leaves little room for spurious low-lying states.

This example involves only two-point correlators. Priors generated by *corrfitter.EigenBasis* are also useful in fits to three-point correlators, with multiple eigen-bases if different types of hadron are involved.

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

4.2 Code

The main method follows the template in *Basic Fits*, but modified to handle the *corrfitter.EigenBasis* object basis:

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

import collections
import gvar as gv
import numpy as np
from corrfitter import CorrFitter, Corr2, EigenBasis, read_dataset

DISPLAYPLOTS = False          # display plots at end of fits?
try:
    import matplotlib
except ImportError:
    DISPLAYPLOTS = False


def main():
    data, basis = make_data('etab.data')
    fitter = CorrFitter(models=make_models(basis))
```

```
    p0 = None
    for N in range(1, 10):
        print(30 * '=', 'nterm =', N)
        prior = make_prior(N, basis)
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0, svdcut=0.0004)
        p0 = fit.pmean
    print_results(fit, basis, prior, data)
    if DISPLAYPLOTS:
        fitter.display_plots()
```

The eigen-basis is created by `make_data('etab.data')`:

```
def make_data(filename):
    data = gv.dataset.avg_data(read_dataset(filename))
    basis = EigenBasis(
        data, keyfmt='1s0.{s1}{s2}', srcs=['l', 'g', 'd', 'e'],
        t=(1,2), tdata=range(1,24),
        )
    return data, basis
```

It reads Monte Carlo data from file `'etab.data'`, which has the following format:

```
1s0.dd 0.143715 0.0588148 0.030329 ...
1s0.dd 0.129067 0.0538892 0.0426075 ...
...
1s0.de 0.120838 0.0510525 0.0306193 ...
1s0.de 0.110661 0.0610127 0.0369064 ...
...
1s0.dg 0.115676 0.0973511 0.0795044 ...
1s0.dg 0.123485 0.10629 0.0885328  ...
...
```

There are 113 lines in the file for each distinct tag `1s0.dd`, `1s0.de` ..., each with 23 numbers. Each line is a separate Monte Carlo estimate of the correlator identified by the tag for `t=1...23`. There are sixteen different correlators in all, with tags given by:

```
'1s0.{s1}{s2}'.format(s1=s1, s2=s2)
```

where `s1` and `s2` are drawn from the source list `['l', 'g', 'd', 'e']`.

The data are read in, and their means and covariance matrix computed using `gvar.dataset.avg_data()`. *corrfitter.EigenBasis* then creates an eigen-basis by solving a generalized eigenvalue problem involving the matrices of correlators at `t=1` and `t=2`. (One might want larger `t` values generally, but these data are too noisy.) The eigenanalysis constructs a set of eigen-sources that are linear combinations of the original sources chosen so that each eigen-source overlaps strongly with one of the lowest four states in the correlator, and weakly with all the others. This eigen-basis is used later to construct the prior.

A correlator fitter, called `fitter`, is created from the list of correlator models returned by `make_models(basis)`:

```
def make_models(basis):
    models = []
    for s1 in basis.srcs:
        for s2 in basis.srcs:
            tfit = basis.tdata if s1 == s2 else basis.tdata[:14]
            models.append(
                Corr2(
                    datatag=basis.keyfmt.format(s1=s1, s2=s2),
                    tdata=basis.tdata, tfit=tfit,
                    a='etab.' + s1, b='etab.' + s2, dE='etab.dE',
                    )
```

```
                )
    return models
```

There is one model for each correlator to be fit, so 16 in all. The keys (`datatag`) for the correlator data are constructed from information stored in the `basis`. Each correlator has data (`tdata`) for `t=1...23`. We fit all `t` values (`tfit`) for the diagonal elements of the matrix correlator, but only about half the `t` values for other correlators — the information at large `t` is highly correlated between different correlators, and therefore somewhat redundant. The amplitudes are labeled by `'etab.l'`, `'etab.g'`, `'etab.d'`, and `'etab.e'` in the prior. The energy differences are labeled by `'etab.dE'`.

We try fits with `N=1,2..9` terms in the fit function. The number of terms is encoded in the prior, which is constructed by `make_prior(N, basis)`:

```python
def make_prior(N, basis):
    return basis.make_prior(nterm=N, keyfmt='etab.{s1}')
```

The prior looks complicated

```
k               prior[k]
-----------     --------------------------------------------------------------
etab.l          [-0.51(17),  -0.45(16),    0.49(17),    0.103(93), -0.07(87) ...]
etab.g          [-0.88(27),   0.107(98),   0.051(93), -0.004(90), -0.13(90) ...]
etab.d          [-0.244(86), -0.50(15),   -0.274(92), -0.166(71), -0.22(60) ...]
etab.e          [-0.212(84), -0.41(13),   -0.31(10),   0.27(10),  -0.12(62) ...]

log(etab.dE) [-1.3(2.3),   -0.5(1.0),   -0.5(1.0),   -0.5(1.0),  -0.5(1.0) ...]
```

but its underlying structure becomes clear if we project it unto the eigen-basis using `prior_eig = basis.apply(prior, keyfmt='etab.{s1}')`:

```
k          prior_eig[k]
------     ----------------------------------------------------------
etab.0     [1.00(30),    .03(10),    .03(10),    .03(10),    .2(1.0) ... ]
etag.1     [ .03(10),   1.00(30),    .03(10),    .03(10),    .2(1.0) ... ]
etab.2     [ .03(10),    .03(10),   1.00(30),    .03(10),    .2(1.0) ... ]
etab.3     [ .03(10),    .03(10),    .03(10),   1.00(30),    .2(1.0) ... ]
```

The *a priori* expectation built into `prior_eig` (and therefore `prior`) is that the ground state overlaps strongly with the first source in the eigen-basis, and weakly with the other three. Similarly the first excited state overlaps strongly with the second eigen-source, but none of the others. And so on. The fifth and higher excited states can overlap with every eigen-source. The priors for the energy differences between successive levels are based upon the energies obtained from the eigenanalysis (`basis.E`): the `dE` prior for the ground state is taken to be `E0(E0)`, where `E0=basis.E[0]`, while for the other states it equals `dE1(dE1)`, where `dE1=basis.E[1]-basis.E[0]`. The prior specifies log-normal statistics for `etab.dE` and so replaces it by `log(etab.dE)`.

The fit is done by `fitter.lsqfit(...)`. An SVD cut is needed (`svdcut=0.0004`) because the data are highly correlated. The data are also over-binned, to keep down the size of the [*corrfitter*](#) distribution, and this mandates an SVD cut, as well. Fit stability is sometimes improved by applying the SVD cut to the data in the eigen-basis rather than in the original source basis. Here this could be done by replacing the last line of `make_data(...)` with:

```python
return basis.svd(data, svdcut=5e-3), basis
```

and dropping the `svdcut=0.0004` from `fitter.lsqfit(...)`. The size of the `svdcut` is usually different.

Final results are printed out by `print_results(...)` after the last fit is finished:

```python
def print_results(fit, basis, prior, data):
    print(30 * '=', 'Results\n')
    print(basis.tabulate(fit.transformed_p, keyfmt='etab.{s1}'))
```

```
    print(basis.tabulate(fit.transformed_p, keyfmt='etab.{s1}', eig_srcs=True))
    E = np.cumsum(fit.transformed_p['etab.dE'])
    outputs = collections.OrderedDict()
    outputs['a*E(2s-1s)'] = E[1] - E[0]
    outputs['a*E(3s-1s)'] = E[2] - E[0]
    outputs['E(3s-1s)/E(2s-1s)'] = (E[2] - E[0]) / (E[1] - E[0])
    inputs = collections.OrderedDict()
    inputs['prior'] = prior
    inputs['data'] = data
    inputs['svdcut'] = fit.svdcorrection
    print(gv.fmt_values(outputs))
    print(gv.fmt_errorbudget(outputs, inputs, colwidth=18))
```

This method first writes out two tables listing energies and amplitudes for the first 4 states in the correlator. The first table shows results for the original sources, while the second is for the eigen-sources. The correlators are from NRQCD so only energy differences are physical. The energy differences for each of the first two excited states relative to the ground states are stored in dictionary `outputs`. These are in lattice units. `outputs` also contains the ratio of 3s-1s difference to the 2s-1s difference, and here the lattice spacing cancels out. The code automatically handles statistical correlations between different energies as it does the arithmetic for `outputs` — the fit results are all `gvar.GVars`. The `outputs` are tabulated using `gvar.fmt_values()`. An error budget is also produced, using `gvar.fmt_errorbudget()`, showing how much error for each quantity comes from uncertainties in the prior and data, and from uncertainties introduced by the SVD cut.

Finally plots showing the data divided by the fit for each correlator are displayed (optionally).

4.3 Results

Running the code produces the following output for the last fit (N=9):

```
=============================== nterm = 9
Least Square Fit:
  chi2/dof [dof] = 0.99 [260]    Q = 0.52     logGBF = 2171

Parameters:
      etab.l 0   -0.50672 (53)       [  -0.51 (17) ]
             1    -0.3915 (61)       [  -0.45 (16) ]
             2      0.350 (31)       [   0.49 (17) ]
             3      0.145 (76)       [  0.103 (93) ]
             4     -0.49 (20)        [  -0.07 (87) ]
             5      0.25 (45)        [  -0.07 (87) ]
             6     -0.12 (40)        [  -0.07 (87) ]
             7      0.21 (38)        [  -0.07 (87) ]
             8     -0.13 (46)        [  -0.07 (87) ]
      etab.g 0   -0.87028 (87)       [  -0.88 (27) ]
             1     0.1436 (52)       [  0.107 (98) ]
             2      0.040 (11)       [  0.051 (93) ]
             3      0.033 (35)       [ -0.004 (90) ]
             4     -0.173 (82)       [  -0.13 (90) ]
             5     -0.15 (24)        [  -0.13 (90) ]
             6     -0.04 (37)        [  -0.13 (90) ]
             7      0.18 (33)        [  -0.13 (90) ]
             8     -0.14 (44)        [  -0.13 (90) ]
      etab.d 0   -0.21632 (29)       [ -0.244 (86) ]
             1    -0.4100 (72)       [  -0.50 (15) ]
             2     -0.250 (17)       [ -0.274 (92) ]
             3     -0.093 (34)       [ -0.166 (71) ]   *
```

```
              4        -0.106 (77)        [  -0.22 (60) ]
              5         0.04 (23)         [  -0.22 (60) ]
              6        -0.16 (35)         [  -0.22 (60) ]
              7        -0.33 (42)         [  -0.22 (60) ]
              8        -0.45 (46)         [  -0.22 (60) ]
     etab.e  0     -0.19704 (25)         [ -0.212 (84) ]
              1       -0.3413 (73)        [  -0.41 (13) ]
              2        -0.296 (15)        [  -0.31 (10) ]
              3         0.221 (54)        [   0.27 (10) ]
              4        -0.052 (77)        [  -0.12 (62) ]
              5         0.06 (17)         [  -0.12 (62) ]
              6         0.05 (40)         [  -0.12 (62) ]
              7        -0.30 (41)         [  -0.12 (62) ]
              8        -0.51 (43)         [  -0.12 (62) ]
 log(etab.dE) 0      -1.36212 (67)        [   -1.3 (2.3) ]
              1        -0.648 (11)        [   -0.5 (1.0) ]
              2        -1.216 (61)        [   -0.5 (1.0) ]
              3        -0.91 (36)         [   -0.5 (1.0) ]
              4        -1.06 (79)         [   -0.5 (1.0) ]
              5        -0.83 (93)         [   -0.5 (1.0) ]
              6        -0.54 (96)         [   -0.5 (1.0) ]
              7        -0.45 (95)         [   -0.5 (1.0) ]
              8        -0.46 (99)         [   -0.5 (1.0) ]

Settings:
  svdcut/n = 0.0004/172    reltol/abstol = 1e-10/1e-10    (itns/time = 143/3.6)
```

This is a good fit, with a chi-squared per degree of freedom of 0.99 for 260 degrees of freedom (the number of data points fit); the *Q* or *p*-value is 0.52. This fit required 146 iterations, but took only a few seconds on a laptop. The results are almost identical to those from N=7 and N=8.

The final energies and amplitudes for the original sources are listed as

```
       E            l            g            d            e
     ==============================================================
  0 0.25612(17) -0.50672(53) -0.87028(87) -0.21632(29) -0.19704(25)
  1 0.7792(57)  -0.3915(61)   0.1436(52)  -0.4100(72)  -0.3413(73)
  2 1.076(21)    0.350(31)    0.040(11)   -0.250(17)   -0.296(15)
  3 1.48(14)     0.145(76)    0.033(35)   -0.093(34)    0.221(54)
```

while for the eigen-sources they are

```
       E            0            1            2            3
     ==============================================================
  0 0.25612(17)  0.97956(97)  0.00644(46) -0.00238(21)  0.00164(14)
  1 0.7792(57)  -0.0337(47)   0.877(11)   -0.019(13)    0.0139(55)
  2 1.076(21)    0.036(10)    0.099(41)    0.803(32)   -0.034(22)
  3 1.48(14)    -0.025(38)   -0.037(61)    0.057(84)    0.74(14)
```

The latter shows that the eigen-sources align quite well with the first four states, as hoped. The errors, especially for the first three states, are much smaller than the prior errors, which indicates strong signals for these states.

Finally values and an error budget are presented for the 2s-1s and 3s-1s energy differences (in lattice units) and the ratio of the two:

```
Values:
        a*E(2s-1s): 0.5231(57)
        a*E(3s-1s): 0.820(21)
  E(3s-1s)/E(2s-1s): 1.567(33)
```

```
Partial % Errors:
                       a*E(2s-1s)        a*E(3s-1s) E(3s-1s)/E(2s-1s)
---------------------------------------------------------------------
            prior:          0.44              0.96              0.71
             data:          0.73              1.44              1.13
           svdcut:          0.69              1.89              1.62
---------------------------------------------------------------------
            total:          1.10              2.56              2.10
```

The first excited state is obviously more accurately determined than the second state, but the fit improves our knowledge of both. The energies for the fifth and higher states merely echo the *a priori* information in the prior — the data are not sufficiently accurate to add much new information to what was in the prior. The prior is less important for the three quantities tabulated here. The dominant source of error in each case comes from either the SVD cut or the statistical errors in the data.

4.4 Fit Stability

It is a good idea in fits like this one to test the stability of the results to significant changes in the prior. This is especially true for quantities like the `3s-1s` splitting that involve more highly excited states. The default prior in effect assigns each of the four sources in the new basis to one of the four states in the correlator with the lowest energies. Typically the actual correspondence between source and low-energy state weakens as the energy increases. So an obvious test is to rerun the fit but with a prior that associates states with only three of the sources, leaving the fourth source unconstrained. This is done by replacing

```python
def make_prior(N, basis):
    return basis.make_prior(nterm=N, keyfmt='etab.{s1}')
```

with

```python
def make_prior(N, basis):
    return basis.make_prior(nterm=N, keyfmt='etab.{s1}', states=[0, 1, 2])
```

in the code. The `states` option in the second `basis.make_prior(...)` assigns the three lowest lying states (in order of increasing energy) to the first three eigen-sources, but leaves the fourth and higher states unassigned. The prior for the amplitudes projected onto the eigen-basis then becomes

```
k         prior_eig[k]
------    ------------------------------------------------------------
etab.0    [1.00(30),    .03(10),    .03(10),  .2(1.0),  .2(1.0) ... ]
etab.1    [ .03(10),   1.00(30),    .03(10),  .2(1.0),  .2(1.0) ... ]
etab.2    [ .03(10),    .03(10),   1.00(30),  .2(1.0),  .2(1.0) ... ]
etab.3    [ .2(1.0),    .2(1.0),    .2(1.0),  .2(1.0),  .2(1.0) ... ]
```

where now no strong assumption is made about the overlaps of the first three eigen-sources with the fourth state, or about the overlap of the fourth source with any state. Running with this (more conservative) prior gives the following results for the last fit and summary:

```
============================ nterm = 9
Least Square Fit:
  chi2/dof [dof] = 0.98 [260]    Q = 0.57    logGBF = 2158.7

Parameters:
     etab.l 0   -0.50654 (57)       [ -0.49 (20) ]
            1    -0.3887 (74)       [ -0.43 (19) ]
            2      0.346 (38)       [  0.51 (21) ]
            3     -0.30 (13)        [ -0.07 (87) ]
```

```
              4       -0.15 (43)        [ -0.07 (87) ]
              5        0.47 (26)        [ -0.07 (87) ]
              6        0.23 (49)        [ -0.07 (87) ]
              7        0.09 (75)        [ -0.07 (87) ]
              8       -0.14 (47)        [ -0.07 (87) ]
     etab.g  0    -0.86999 (94)        [ -0.88 (27) ]
             1       0.1438 (53)        [ 0.110 (99) ]
             2        0.038 (13)        [ 0.054 (94) ]
             3       -0.100 (62)        [ -0.13 (90) ]
             4        -0.13 (13)        [ -0.13 (90) ]
             5        -0.03 (22)        [ -0.13 (90) ]
             6         0.23 (32)        [ -0.13 (90) ]
             7         0.09 (78)        [ -0.13 (90) ]
             8        -0.17 (49)        [ -0.13 (90) ]
     etab.d  0    -0.21624 (30)        [ -0.27 (16) ]
             1      -0.4085 (80)        [ -0.52 (20) ]
             2       -0.247 (19)        [ -0.30 (17) ]
             3        0.033 (89)        [ -0.22 (60) ]
             4       -0.169 (80)        [ -0.22 (60) ]
             5         0.11 (15)        [ -0.22 (60) ]
             6        -0.16 (36)        [ -0.22 (60) ]
             7        -0.21 (53)        [ -0.22 (60) ]
             8        -0.47 (40)        [ -0.22 (60) ]
     etab.e  0    -0.19697 (27)        [ -0.16 (31) ]
             1      -0.3389 (85)        [ -0.36 (32) ]
             2       -0.304 (21)        [ -0.26 (31) ]
             3        -0.18 (10)        [ -0.12 (62) ]
             4         0.11 (16)        [ -0.12 (62) ]
             5         0.06 (10)        [ -0.12 (62) ]
             6        -0.08 (39)        [ -0.12 (62) ]
             7        -0.21 (55)        [ -0.12 (62) ]
             8        -0.59 (40)        [ -0.12 (62) ]
 log(etab.dE) 0    -1.36215 (67)       [ -1.3 (2.3) ]
             1       -0.651 (12)        [ -0.5 (1.0) ]
             2       -1.206 (59)        [ -0.5 (1.0) ]
             3        -1.01 (45)        [ -0.5 (1.0) ]
             4        -1.17 (63)        [ -0.5 (1.0) ]
             5        -0.97 (82)        [ -0.5 (1.0) ]
             6        -0.44 (94)        [ -0.5 (1.0) ]
             7        -0.40 (96)        [ -0.5 (1.0) ]
             8        -0.45 (98)        [ -0.5 (1.0) ]

Settings:
  svdcut/n = 0.0004/172    reltol/abstol = 1e-10/1e-10     (itns/time = 500/10.9)

=========================== Results

      E            l            g            d            e
   ==================================================================
   0 0.25611(17)  -0.50654(57)  -0.86999(94)  -0.21624(30)  -0.19697(27)
   1 0.7775(65)   -0.3887(74)    0.1438(53)   -0.4085(80)   -0.3389(85)
   2 1.077(20)     0.346(38)     0.038(13)    -0.247(19)    -0.304(21)
   3 1.44(16)     -0.30(13)     -0.100(62)     0.033(89)    -0.18(10)

      E            0            1            2            3
   ==================================================================
   0 0.25611(17)   0.9792(10)   0.00641(46)  -0.00237(21)   0.00165(14)
   1 0.7775(65)   -0.0344(53)   0.873(13)    -0.018(14)     0.0166(63)
```

```
    2 1.077(20)     0.037(13)    0.100(46)     0.802(33)     -0.060(34)
    3 1.44(16)      0.108(76)    0.15(11)     -0.21(18)      -0.55(36)

Values:
        a*E(2s-1s): 0.5214(64)
        a*E(3s-1s): 0.821(20)
  E(3s-1s)/E(2s-1s): 1.574(33)


Partial % Errors:
                          a*E(2s-1s)        a*E(3s-1s) E(3s-1s)/E(2s-1s)
----------------------------------------------------------------------
            prior:           0.53              0.81              0.66
             data:           0.79              1.49              1.21
           svdcut:           0.79              1.82              1.57
----------------------------------------------------------------------
            total:           1.23              2.49              2.09
```

The energies and amplitudes for the first three states are almost unchanged, which gives us confidence in the original results. Results for the fourth and higher states have larger errors, as expected.

Note that while the chi-squared value for this last fit is almost identical to that in the original fit, the Bayes Factor (from `logGBF`) is exp(2171-2158.6)=240,000 times larger for the original fit. The Bayes Factor gives us a sense of which prior the data prefer. Specifically it says that our Monte Carlo data are 240,000 times more likely to have come from a model with the original prior than from one with the more conservative prior. This further reinforces our confidence in the original results.

4.5 Alternative Organization

Our η_b fit uses the *corrfitter.EigenBasis* to construct a special prior for the fit, but leaves the correlators unchanged. An alternative approach is to project the correlators unto the eigen-basis, and then to fit them with a fit function defined directly in terms of the eigen-basis. This approach is conceptually identical to that above, but in practice it gives somewhat different results since the SVD cut enters differently. A version of the code above that uses this approach is:

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

import collections
import gvar as gv
import numpy as np
from corrfitter import CorrFitter, Corr2, EigenBasis, read_dataset

DISPLAYPLOTS = False         # display plots at end of fits?
try:
    import matplotlib
except ImportError:
    DISPLAYPLOTS = False

def main():
    data, basis = make_data('etab.data')
    fitter = CorrFitter(models=make_models(basis))
    p0 = None
    for N in range(1, 10):
        print(30 * '=', 'nterm =', N)
        prior = make_prior(N, basis)
        fit = fitter.lsqfit(data=data, prior=prior, p0=p0, svdcut=0.005 )    #1
        p0 = fit.pmean
```

```python
        print_results(fit, basis, prior, data)
        if DISPLAYPLOTS:
            fitter.display_plots()

def make_data(filename):
    data = gv.dataset.avg_data(read_dataset(filename))
    basis = EigenBasis(
        data, keyfmt='1s0.{s1}{s2}', srcs=['l', 'g', 'd', 'e'],
        t=(1,2), tdata=range(1,24),
        )
    return basis.apply(data, keyfmt='1s0.{s1}{s2}'), basis              #2

def make_models(basis):
    models = []
    for s1 in basis.eig_srcs:                                           #3
        for s2 in basis.eig_srcs:                                       #4
            tfit = basis.tdata if s1 == s2 else basis.tdata[:14]
            models.append(
                Corr2(
                    datatag=basis.keyfmt.format(s1=s1, s2=s2),
                    tdata=basis.tdata, tfit=tfit,
                    a='etab.' + s1, b='etab.' + s2, dE='etab.dE',
                    )
                )
    return models

def make_prior(N, basis):
    return basis.make_prior(nterm=N, keyfmt='etab.{s1}', eig_srcs=True)  #5

def print_results(fit, basis, prior, data):
    print(30 * '=', 'Results\n')
    print(basis.tabulate(fit.transformed_p, keyfmt='etab.{s1}'))
    print(basis.tabulate(fit.transformed_p, keyfmt='etab.{s1}', eig_srcs=True))
    E = np.cumsum(fit.transformed_p['etab.dE'])
    outputs = collections.OrderedDict()
    outputs['a*E(2s-1s)'] = E[1] - E[0]
    outputs['a*E(3s-1s)'] = E[2] - E[0]
    outputs['E(3s-1s)/E(2s-1s)'] = (E[2] - E[0]) / (E[1] - E[0])
    inputs = collections.OrderedDict()
    inputs['prior'] = prior
    inputs['data'] = data
    inputs['svdcut'] = fit.svdcorrection
    print(gv.fmt_values(outputs))
    print(gv.fmt_errorbudget(outputs, inputs, colwidth=18))

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

Only five lines in this code differ from the original: the SVD cut is different (#1); the data are projected onto the eigen-basis (#2); the models are defined in terms of the eigen-sources (#3 and #4); and the prior is defined for the eigen-sources (#5).

The fit results from the new code are very similar to before; there is little difference between the two approaches in this case:

```
============================= nterm = 9
Least Square Fit:
  chi2/dof [dof] = 0.94 [260]    Q = 0.77    logGBF = 1945.7
```

```
Parameters:
      etab.0  0      0.9815 (11)       [  1.00 (30) ]
              1     -0.0390 (55)       [  0.03 (10) ]
              2      0.048 (10)        [  0.03 (10) ]
              3     -0.036 (32)        [  0.03 (10) ]
              4      0.222 (83)        [  0.2 (1.0) ]
              5     -0.13 (24)         [  0.2 (1.0) ]
              6      0.004 (850)       [  0.2 (1.0) ]
              7     -0.16 (41)         [  0.2 (1.0) ]
              8      0.17 (29)         [  0.2 (1.0) ]
      etab.1  0     0.00601 (52)       [  0.03 (10) ]
              1      0.883 (12)        [  1.00 (30) ]
              2      0.058 (40)        [  0.03 (10) ]
              3     -0.063 (72)        [  0.03 (10) ]
              4      0.39 (14)         [  0.2 (1.0) ]
              5      0.07 (46)         [  0.2 (1.0) ]
              6      0.13 (93)         [  0.2 (1.0) ]
              7      0.68 (51)         [  0.2 (1.0) ]
              8      0.54 (68)         [  0.2 (1.0) ]
      etab.2  0    -0.00260 (18)       [  0.03 (10) ]
              1    -0.0005 (131)       [  0.03 (10) ]
              2      0.802 (34)        [  1.00 (30) ]
              3      0.061 (86)        [  0.03 (10) ]
              4     -0.49 (18)         [  0.2 (1.0) ]
              5     -0.19 (48)         [  0.2 (1.0) ]
              6      0.13 (92)         [  0.2 (1.0) ]
              7      0.10 (69)         [  0.2 (1.0) ]
              8      1.04 (54)         [  0.2 (1.0) ]
      etab.3  0    0.001639 (60)       [  0.03 (10) ]
              1      0.0155 (50)       [  0.03 (10) ]
              2     -0.031 (20)        [  0.03 (10) ]
              3      0.66 (14)         [  1.00 (30) ]  *
              4      0.04 (28)         [  0.2 (1.0) ]
              5     -0.51 (44)         [  0.2 (1.0) ]
              6      0.21 (94)         [  0.2 (1.0) ]
              7      0.35 (69)         [  0.2 (1.0) ]
              8     -0.53 (65)         [  0.2 (1.0) ]
 log(etab.dE)  0   -1.36278 (75)       [ -1.3 (2.3) ]
              1     -0.649 (12)        [ -0.5 (1.0) ]
              2     -1.229 (73)        [ -0.5 (1.0) ]
              3     -0.93 (37)         [ -0.5 (1.0) ]
              4     -0.86 (56)         [ -0.5 (1.0) ]
              5     -0.88 (92)         [ -0.5 (1.0) ]
              6     -0.52 (93)         [ -0.5 (1.0) ]
              7     -0.53 (94)         [ -0.5 (1.0) ]
              8     -0.59 (96)         [ -0.5 (1.0) ]

Settings:
  svdcut/n = 0.005/171    reltol/abstol = 1e-10/1e-10     (itns/time = 103/3.0)

============================= Results

      E           l           g           d           e
    =======================================================================
    0 0.25595(19) -0.50766(58) -0.87211(96) -0.21649(34) -0.19720(29)
    1 0.7787(65)  -0.3819(82)   0.1506(50)  -0.4166(74)  -0.3477(73)
    2 1.071(22)    0.363(30)    0.024(11)   -0.233(19)   -0.280(17)
    3 1.46(14)     0.156(76)    0.039(28)   -0.069(43)    0.211(56)
```

```
      E               0              1              2              3
   ================================================================
   0 0.25595(19)   0.9815(11)   0.00601(52)  -0.00260(18)   0.001639(60)
   1 0.7787(65)   -0.0390(55)   0.883(12)    -0.0005(131)   0.0155(50)
   2 1.071(22)     0.048(10)    0.058(40)     0.802(34)    -0.031(20)
   3 1.46(14)     -0.036(32)   -0.063(72)     0.061(86)     0.66(14)


Values:
       a*E(2s-1s): 0.5228(65)
       a*E(3s-1s): 0.815(22)
 E(3s-1s)/E(2s-1s): 1.560(42)


Partial % Errors:
                         a*E(2s-1s)      a*E(3s-1s) E(3s-1s)/E(2s-1s)
--------------------------------------------------------------------
            prior:          0.38            1.12             1.14
             data:          0.70            1.37             1.28
           svdcut:          0.96            1.98             2.09
--------------------------------------------------------------------
            total:          1.24            2.66             2.71
```

INDICES AND TABLES

- genindex
- modindex
- search

C

Symbols

_abscissa (corrfitter.BaseModel attribute), 18

A

all_datatags (corrfitter.BaseModel attribute), 18
ampl (corrfitter.fastfit attribute), 28
ampllist (corrfitter.fastfit attribute), 29
apply() (corrfitter.EigenBasis method), 26

B

BaseModel (class in corrfitter), 18
bootstrap_fit_iter() (corrfitter.CorrFitter method), 20
bootstrap_iter() (corrfitter.CorrFitter method), 20
builddata() (corrfitter.BaseModel method), 18
builddata() (corrfitter.Corr2 method), 15
builddata() (corrfitter.Corr3 method), 18
builddata() (corrfitter.CorrFitter method), 21
buildfitfcn() (corrfitter.CorrFitter method), 21
buildprior() (corrfitter.BaseModel method), 19
buildprior() (corrfitter.Corr2 method), 16
buildprior() (corrfitter.Corr3 method), 18
buildprior() (corrfitter.CorrFitter method), 21

C

chained_lsqfit() (corrfitter.CorrFitter method), 21
chi2 (corrfitter.fastfit attribute), 28
collect_fitresults() (corrfitter.CorrFitter method), 22
Corr2 (class in corrfitter), 14
Corr3 (class in corrfitter), 16
CorrFitter (class in corrfitter), 19
corrfitter (module), 3

D

datatag (corrfitter.BaseModel attribute), 18
display_plots() (corrfitter.CorrFitter method), 22
dof (corrfitter.fastfit attribute), 28

E

E (corrfitter.EigenBasis attribute), 26
E (corrfitter.fastfit attribute), 28
eig_srcs (corrfitter.EigenBasis attribute), 26
EigenBasis (class in corrfitter), 24

Elist (corrfitter.fastfit attribute), 29

F

fastfit (class in corrfitter), 28
fitfcn() (corrfitter.BaseModel method), 19
fitfcn() (corrfitter.Corr2 method), 16
fitfcn() (corrfitter.Corr3 method), 18

L

lsqfit() (corrfitter.CorrFitter method), 22

M

make_prior() (corrfitter.EigenBasis method), 26

Q

Q (corrfitter.fastfit attribute), 28

S

simulated_data_iter() (corrfitter.CorrFitter method), 23
svd() (corrfitter.EigenBasis method), 27
svdcorrection (corrfitter.EigenBasis attribute), 26
svdn (corrfitter.EigenBasis attribute), 26

T

tabulate() (corrfitter.EigenBasis method), 27

U

unapply() (corrfitter.EigenBasis method), 27

V

v (corrfitter.EigenBasis attribute), 26
v_inv (corrfitter.EigenBasis attribute), 26