corrfitter Documentation

Release 3.6.3

G.P. Lepage

February 02, 2014

CONTENTS

Contents:

CHAPTER

ONE

CORRFITTER - LEAST-SQUARES FIT TO CORRELATORS

1.1 Introduction

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

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

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

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

We now review the basic features of `corrfitter`. These features are also illustrated in the context of a real application in an *Annotated Example*, at the end.

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.GVar`s 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:

```
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:

```
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:

```
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:

```
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`:

```
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.

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. Parameters `svdcut` and `svdnum` are used to specify an *svd* cut. (It is often useful to to set `svdnum` equal to the number of measurements used to determine the covariance matrix for `G(t)` since that is the largest number of eigenmodes possible in the covariance matrix.)

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:

```
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:

```
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:

```
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:

```
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:

```
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 dset.bootstrap_iter(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 web searches for "hep-lat/0110175" and "arXiv:1111.1363". These are two papers by G.P. Lepage and collaborators whose published versions are: G.P. Lepage et al, Nucl.Phys.Proc.Suppl. 106 (2002) 12-20; and K. Hornbostel et al, Phys.Rev. D85 (2012) 031504.

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 `t`s 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 `t`s to use in the fit. Only data with these `t`s (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.GVar`s 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, 1e-15)*, *svdnum=None*, *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` or 2-tuple) – If `svdcut` is positive, eigenvalues `ev[i]` of the (rescaled) data covariance matrix that are smaller than `svdcut*max(ev)` are replaced by `svdcut*max(ev)` in the covariance matrix. If `svdcut` is negative, eigenvalues less than `|svdcut|*max(ev)` are set to zero in the covariance matrix. The covariance matrix is

left unchanged if `svdcut` is set equal to `None` (default). If `svdcut` is a 2-tuple, *svd* cuts are applied to both the correlator data (`svdcut[0]`) and to the prior (`svdcut[1]`).

- **svdnum** (integer or `None` or 2-tuple) – At most `svdnum` eigenmodes are retained in the (rescaled) data covariance matrix; the modes with the smallest eigenvalues are discarded. `svdnum` is ignored if it is set to `None`. If `svdnum` is a 2-tuple, *svd* cuts are applied to both the correlator data (`svdnum[0]`) and to the prior (`svdnum[1]`).
- **tol** (*positive number less than 1*) – Tolerance used in `lsqfit.nonlinear_fit()` for the least-squares fits (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*, *svdnum=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].datatag`s.

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 `datatag`s from the correlator models are used as data labels, with `data[datatag]` being a 1-d array of `gvar.GVar`s 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`, `svdnum`, `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()

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.

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

Parameters

- **data** (*dictionary*) – Input data. The `datatag`s from the correlator models are used as data labels, with `data[datatag]` being a 1-d array of `gvar.GVar`s 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`; otherwise print nothing.
- **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`, `svdnum`, `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 Fast Fit Objects

class `corrfitter.`**`fastfit`**(*data*, *prior*, *model*, *svdcut=None*, *svdnum=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 amplitudes, `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.GVar`s 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 `t`s 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 parameters `svdcut` and `svdnum`. Also the type of marginalization use can be specified with parameter `ratio` (see `corrfitter.CorrFitter`).

1.16 Annotated Example

In this section we describe a complete script that uses `corrfitter` to extract energies, amplitudes, and transition matrix elements for the η_s and D_s mesons. The source code (`example.py`) and data file (`example.data`) are included with the `corrfitter` distribution, in the `examples/` directory.

The `main` method follows the pattern described in *Faster Fits*:

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

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

def main():
    data = make_data('example.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)
    fitter.display_plots()
```

The raw Monte Carlo data is in a file named `'example.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 23. It also makes multi-term fits more stable.

The last line 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.

1.16.1 a) make_data

Method `make_data('example.data')` reads in the Monte Carlo data, averages it, and formats it for use by `corrfitter.CorrFitter`. The data file (`'example.data'`) contains 225 lines, each with 64 numbers on it, of the form:

```
etas    0.3045088594E+00    0.7846334531E-01    0.3307295938E-01 ...
etas    0.3058093438E+00    0.7949004688E-01    0.3344648906E-01 ...
...
```

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    0.2303351094E+00    0.4445243750E-01    0.8941437344E-02 ...
Ds    0.2306766563E+00    0.4460026875E-01    0.8991960781E-02 ...
...
```

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

```
3ptT15    0.4679643906E-09    0.1079643844E-08    0.2422032031E-08 ...
3ptT15    0.4927106406E-09    0.1162639109E-08    0.2596277812E-08 ...
...

3ptT16     0.1420718453E-09    0.3205214219E-09    0.7382921875E-09 ...
3ptT16     0.1501385469E-09    0.3478552344E-09    0.8107883594E-09 ...
...
```

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

We use the tools in `gvar.dataset` designed for reading files in this format:

```
def make_data(datafile):
    """ Read data from datafile and average it. """
    return gv.dataset.avg_data(gv.dataset.Dataset(datafile))
```

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

```
>>> data = make_data('example.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.GVar`s representing the Monte Carlo estimates (mean and covariance) for the corresponding correlator. This is the format needed by `corrfitter.CorrFitter`.

1.16.2 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(tmin),  tfit=range(tmin, tp-tmin),
            a='etas:a',  b='etas:a',  dE='etas:dE'
            ),

        Corr2(
            datatag='Ds',
            tp=tp,  tdata=range(tmin),  tfit=range(tmin, tp-tmin),
            a=('Ds:a', 'Ds:ao'), b=('Ds:a', 'Ds:ao'),
            dE=('Ds:dE', 'Ds:dEo'), 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', 'Ds:ao'), dEb=('Ds:dE', 'Ds:dEo'), 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', 'Ds:ao'), dEb=('Ds:dE', 'Ds:dEo'), tpb=tp, sb=(1, -1.),
            Vnn='Vnn', Vno='Vno'
            )
        ]
    return models
```

Four models are specified, one for each correlator to be fit. The first two are for the η_s and D_s two-point correlators, corresponding to entries in the data dictionary with keys `'etas'` and `'Ds'`, respectively. These are periodic propagators, with period 64 (`tp`), and we want to omit the first and last 5 (`tmin`) time steps in the correlator. The `t`s to be fit are listed in `tfit`, while the `t`s 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: `'Ds:ao'` and `'Ds:dEo'`.

Finally three-point models are specifies 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 elements: `'Vnn'` connecting normal states, and `'Vno'` connecting normal states with oscillating states.

1.16.3 c) make_prior

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

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

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

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

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

    return prior
```

Parameter `N` specifies how many terms are kept in the fit functions. The priors are 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.GVar`s from a list of strings of the form `'0.1(1)'`: for example,

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

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

We override the default priors for the ground-state energies in each case. This is not unusual since `dE[0]`, unlike the other `dE`s, 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.GVar`s, with one for each element of `Vnn`; it is one of the most succinct ways of creating a large number of `gvar.GVar`s. 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.GVar`s using `gvar.gvar()`.

1.16.4 d) print_results

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

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

    # 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['Ds:dEo'])
    a_Dso = p['Ds:ao']
    print('\n  EDso:', E_Dso[:3])
    print(  '  aDso:', a_Dso[:3])

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

    # error budget
    outputs = gv.BufferDict()
    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()      # can use dict() instead
    inputs['statistics'] = data             # statistical errors in data
    inputs.update(prior)                    # all entries in prior
    inputs['svd'] = fit.svdcorrection       # svd cut (if present)
```

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

    print('\n')
```

The best-fit parameter values are stored in dictionary `p=fit.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(Ds:ao):      0.00      0.00      1.10      0.01      3.85
        log(Ds:dEo):      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.

1.16.5 e) Final Results

The output from running this code is as follows:

```
============================== nterm = 1
Least Square Fit:
  chi2/dof [dof] = 7.4e+03 [69]    Q = 0    logGBF = -2.5405e+05    itns = 26

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(Ds:ao) 0    -1.7014 (16)      [ -2.3 (1.0) ]
  log(Ds:dEo) 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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

============================== nterm = 2
Least Square Fit:
  chi2/dof [dof] = 3 [69]    Q = 6e-16    logGBF = 1531.1    itns = 78

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(Ds:ao) 0    -2.6014 (75)      [ -2.3 (1.0) ]
              1     -1.266 (76)      [ -2.3 (1.0) ]  *
  log(Ds:dEo) 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       0.1 (1.0)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

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

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(Ds:ao) 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(Ds:dEo) 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        0.0 (1.0)      [  0.0 (1.0) ]
            2,0      -0.06 (14)       [  0.0 (1.0) ]
            2,1        0.1 (1.0)      [  0.0 (1.0) ]
            2,2        0.0 (1.0)      [  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       0.00 (84)       [  0.0 (1.0) ]
            1,0       0.39 (45)       [  0.0 (1.0) ]
            1,1       0.20 (95)       [  0.0 (1.0) ]
            1,2        0.0 (1.0)      [  0.0 (1.0) ]
            2,0      -0.17 (92)       [  0.0 (1.0) ]
            2,1        0.0 (1.0)      [  0.0 (1.0) ]
            2,2        0.0 (1.0)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

============================== nterm = 4
Least Square Fit:
  chi2/dof [dof] = 0.7 [69]    Q = 0.97    logGBF = 1602.1    itns = 23

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(Ds:ao) 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(Ds:dEo) 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       0.1 (1.0)      [  0.0 (1.0) ]
            1,0      0.050 (42)      [  0.0 (1.0) ]
            1,1       0.25 (70)      [  0.0 (1.0) ]
            1,2       0.0 (1.0)      [  0.0 (1.0) ]
            1,3       0.0 (1.0)      [  0.0 (1.0) ]
            2,0      -0.07 (19)      [  0.0 (1.0) ]
            2,1       0.1 (1.0)      [  0.0 (1.0) ]
            2,2       0.0 (1.0)      [  0.0 (1.0) ]
            2,3       0.0 (1.0)      [  0.0 (1.0) ]
            3,0       0.00 (98)      [  0.0 (1.0) ]
            3,1       0.0 (1.0)      [  0.0 (1.0) ]
            3,2       0.0 (1.0)      [  0.0 (1.0) ]
            3,3       0.0 (1.0)      [  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       0.00 (85)      [  0.0 (1.0) ]
            0,3       0.0 (1.0)      [  0.0 (1.0) ]
            1,0       0.38 (45)      [  0.0 (1.0) ]
            1,1       0.22 (95)      [  0.0 (1.0) ]
            1,2       0.0 (1.0)      [  0.0 (1.0) ]
            1,3       0.0 (1.0)      [  0.0 (1.0) ]
            2,0      -0.17 (93)      [  0.0 (1.0) ]
            2,1       0.0 (1.0)      [  0.0 (1.0) ]
            2,2       0.0 (1.0)      [  0.0 (1.0) ]
            2,3       0.0 (1.0)      [  0.0 (1.0) ]
            3,0      -0.0 (1.0)      [  0.0 (1.0) ]
            3,1       0.0 (1.0)      [  0.0 (1.0) ]
            3,2       0.0 (1.0)      [  0.0 (1.0) ]
            3,3       0.0 (1.0)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

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(Ds:ao):       0.00      0.00      1.10      0.01      3.85
        log(Ds:dEo):       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/dof`s 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).
- 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.chained_lsqfit(data=data, prior=prior, p0=p0)
  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: 0.41620(11)
                mDs: 1.20154(15)
          mDso-mDs: 0.2557(29)
               Vnn: 0.76718(60)
               Vno: -0.746(15)
```

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

```
def main():
    data = make_data('example.data')
    models = make_models()
    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)
    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 (input data correlated with prior):
  chi2/dof [dof] = 0.98 [69]    Q = 0.53    logGBF = 1586.4    itns = 8

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(Ds:ao) 0     -2.628 (25)      [ -2.3 (1.0) ]
  log(Ds:dEo) 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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

============================== nterm = 2
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.71 [69]    Q = 0.97    logGBF = 1602.3    itns = 17

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(Ds:ao)  0     -2.746 (92)       [ -2.3 (1.0) ]
                1      -2.44 (10)       [ -2.3 (1.0) ]
   log(Ds:dEo)  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 = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

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(Ds:ao):      0.00      0.00      0.40      0.00      1.37
        log(Ds:dEo):      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
```

- Test the code by adding `test_fit(fitter, 'example.data')` to the `main` program, where:

```
def test_fit(fitter, datafile):
    gv.ranseed((5339893179535759510, 4088224360017966188, 7597275990505476522))
    print('\nRandom seed:', gv.ranseed.seed)
    dataset = gv.dataset.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])
        print(
            'Leading parameter chi2/dof [dof] = %.2f' %
            (gv.chi2(diff) / gv.chi2.dof),
            '[%d]' % gv.chi2.dof,
            '  Q = %.1f' % gv.chi2.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: (5339893179535759510, 4088224360017966188, 7597275990505476522)

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.68 [69]    Q = 0.98    logGBF = 1602.5    itns = 21

Parameters:
  log(etas:a) 0   -1.52103 (72)      [ -1.2 (1.0) ]
              1      -1.76 (32)      [ -1.2 (1.0) ]
              2      -1.13 (48)      [ -1.2 (1.0) ]
              3      -1.22 (95)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87635 (28)      [ -0.92 (50) ]
              1      -0.54 (12)      [ -0.7 (1.0) ]
              2      -0.72 (53)      [ -0.7 (1.0) ]
              3      -0.70 (97)      [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53847 (93)      [ -1.2 (1.0) ]
              1      -1.32 (10)      [ -1.2 (1.0) ]
              2      -0.83 (38)      [ -1.2 (1.0) ]
              3      -1.13 (98)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18379 (13)      [  0.18 (17) ]
              1     -0.701 (45)      [ -0.7 (1.0) ]
              2      -0.69 (39)      [ -0.7 (1.0) ]
              3      -0.75 (99)      [ -0.7 (1.0) ]
   log(Ds:ao) 0     -2.709 (97)      [ -2.3 (1.0) ]
              1      -2.46 (38)      [ -2.3 (1.0) ]
              2      -2.16 (82)      [ -2.3 (1.0) ]
              3      -2.3 (1.0)      [ -2.3 (1.0) ]
  log(Ds:dEo) 0     0.3691 (82)      [  0.41 (24) ]
              1      -1.40 (48)      [ -0.7 (1.0) ]
              2      -0.69 (80)      [ -0.7 (1.0) ]
              3      -0.7 (1.0)      [ -0.7 (1.0) ]
        Vnn 0,0    0.76731 (83)      [  0.0 (1.0) ]
            0,1     -0.487 (38)      [  0.0 (1.0) ]
            0,2       0.32 (43)      [  0.0 (1.0) ]
```

```
              0,3       0.07 (99)       [  0.0 (1.0) ]
              1,0      0.077 (41)       [  0.0 (1.0) ]
              1,1       0.60 (70)       [  0.0 (1.0) ]
              1,2       0.12 (99)       [  0.0 (1.0) ]
              1,3        0.0 (1.0)      [  0.0 (1.0) ]
              2,0      -0.24 (30)       [  0.0 (1.0) ]
              2,1        0.0 (1.0)      [  0.0 (1.0) ]
              2,2        0.0 (1.0)      [  0.0 (1.0) ]
              2,3        0.0 (1.0)      [  0.0 (1.0) ]
              3,0      -0.13 (98)       [  0.0 (1.0) ]
              3,1        0.0 (1.0)      [  0.0 (1.0) ]
              3,2        0.0 (1.0)      [  0.0 (1.0) ]
              3,3        0.0 (1.0)      [  0.0 (1.0) ]
          Vno 0,0     -0.766 (61)       [  0.0 (1.0) ]
              0,1       0.29 (28)       [  0.0 (1.0) ]
              0,2       0.11 (89)       [  0.0 (1.0) ]
              0,3        0.0 (1.0)      [  0.0 (1.0) ]
              1,0       0.08 (40)       [  0.0 (1.0) ]
              1,1       0.21 (96)       [  0.0 (1.0) ]
              1,2        0.0 (1.0)      [  0.0 (1.0) ]
              1,3        0.0 (1.0)      [  0.0 (1.0) ]
              2,0      -0.07 (94)       [  0.0 (1.0) ]
              2,1        0.0 (1.0)      [  0.0 (1.0) ]
              2,2        0.0 (1.0)      [  0.0 (1.0) ]
              2,3        0.0 (1.0)      [  0.0 (1.0) ]
              3,0      -0.0 (1.0)       [  0.0 (1.0) ]
              3,1        0.0 (1.0)      [  0.0 (1.0) ]
              3,2        0.0 (1.0)      [  0.0 (1.0) ]
              3,3        0.0 (1.0)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

Leading parameter chi2/dof [dof] = 0.27 [8]   Q = 1.0

============================== simulation
Least Square Fit:
  chi2/dof [dof] = 0.63 [69]    Q = 0.99    logGBF = 1604.2    itns = 41

Parameters:
  log(etas:a) 0   -1.52248 (71)       [ -1.2 (1.0) ]
              1      -1.60 (18)       [ -1.2 (1.0) ]
              2      -1.03 (69)       [ -1.2 (1.0) ]
              3      -1.05 (97)       [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87681 (28)       [ -0.92 (50) ]
              1     -0.474 (72)       [ -0.7 (1.0) ]
              2      -0.48 (59)       [ -0.7 (1.0) ]
              3      -0.78 (98)       [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53993 (98)       [ -1.2 (1.0) ]
              1      -1.45 (15)       [ -1.2 (1.0) ]
              2      -0.81 (28)       [ -1.2 (1.0) ]
              3      -1.18 (98)       [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18356 (13)       [  0.18 (17) ]
              1     -0.769 (64)       [ -0.7 (1.0) ]
              2      -0.78 (30)       [ -0.7 (1.0) ]
              3      -0.74 (99)       [ -0.7 (1.0) ]
   log(Ds:ao) 0     -2.709 (76)       [ -2.3 (1.0) ]
              1      -2.38 (16)       [ -2.3 (1.0) ]
```

```
              2      -2.46 (98)       [ -2.3 (1.0) ]
              3      -2.3 (1.0)       [ -2.3 (1.0) ]
  log(Ds:dEo) 0     0.3681 (66)       [  0.41 (24) ]
              1      -1.33 (25)       [ -0.7 (1.0) ]
              2      -0.43 (94)       [ -0.7 (1.0) ]
              3      -0.7 (1.0)       [ -0.7 (1.0) ]
        Vnn 0,0    0.76785 (82)       [  0.0 (1.0) ]
            0,1     -0.443 (44)       [  0.0 (1.0) ]
            0,2       0.02 (26)       [  0.0 (1.0) ]
            0,3      -0.02 (99)       [  0.0 (1.0) ]
            1,0      0.047 (32)       [  0.0 (1.0) ]
            1,1       0.23 (70)       [  0.0 (1.0) ]
            1,2       0.09 (99)       [  0.0 (1.0) ]
            1,3        0.0 (1.0)      [  0.0 (1.0) ]
            2,0      -0.21 (36)       [  0.0 (1.0) ]
            2,1       -0.0 (1.0)      [  0.0 (1.0) ]
            2,2        0.0 (1.0)      [  0.0 (1.0) ]
            2,3        0.0 (1.0)      [  0.0 (1.0) ]
            3,0      -0.04 (99)       [  0.0 (1.0) ]
            3,1        0.0 (1.0)      [  0.0 (1.0) ]
            3,2        0.0 (1.0)      [  0.0 (1.0) ]
            3,3        0.0 (1.0)      [  0.0 (1.0) ]
        Vno 0,0     -0.760 (37)       [  0.0 (1.0) ]
            0,1       0.31 (17)       [  0.0 (1.0) ]
            0,2       0.00 (98)       [  0.0 (1.0) ]
            0,3        0.0 (1.0)      [  0.0 (1.0) ]
            1,0       0.15 (38)       [  0.0 (1.0) ]
            1,1      -0.18 (96)       [  0.0 (1.0) ]
            1,2        0.0 (1.0)      [  0.0 (1.0) ]
            1,3        0.0 (1.0)      [  0.0 (1.0) ]
            2,0       0.14 (98)       [  0.0 (1.0) ]
            2,1        0.0 (1.0)      [  0.0 (1.0) ]
            2,2        0.0 (1.0)      [  0.0 (1.0) ]
            2,3        0.0 (1.0)      [  0.0 (1.0) ]
            3,0        0.0 (1.0)      [  0.0 (1.0) ]
            3,1        0.0 (1.0)      [  0.0 (1.0) ]
            3,2        0.0 (1.0)      [  0.0 (1.0) ]
            3,3        0.0 (1.0)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

Leading parameter chi2/dof [dof] = 0.46 [8]   Q = 0.9
```

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: (5339893179535759510, 4088224360017966188, 7597275990505476522)

============================== simulation
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.7 [69]    Q = 0.97    logGBF = 1602.4    itns = 23
```

```
Parameters:
  log(etas:a) 0   -1.52098 (72)      [ -1.2 (1.0) ]
              1      -1.71 (49)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87634 (28)      [ -0.92 (50) ]
              1      -0.52 (16)      [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53883 (91)      [ -1.2 (1.0) ]
              1     -1.390 (78)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18377 (13)      [  0.18 (17) ]
              1     -0.735 (37)      [ -0.7 (1.0) ]
   log(Ds:ao) 0     -2.670 (58)      [ -2.3 (1.0) ]
              1      -2.34 (14)      [ -2.3 (1.0) ]
  log(Ds:dEo) 0     0.3719 (55)      [  0.41 (24) ]
              1      -1.20 (21)      [ -0.7 (1.0) ]
        Vnn 0,0    0.76751 (76)      [  0.0 (1.0) ]
            0,1     -0.459 (35)      [  0.0 (1.0) ]
            1,0      0.080 (53)      [  0.0 (1.0) ]
            1,1       0.72 (73)      [  0.0 (1.0) ]
        Vno 0,0     -0.755 (29)      [  0.0 (1.0) ]
            0,1       0.37 (16)      [  0.0 (1.0) ]
            1,0    -0.067 (398)      [  0.0 (1.0) ]
            1,1       0.27 (96)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

Leading parameter chi2/dof [dof] = 0.82 [8]   Q = 0.6

============================== simulation
Least Square Fit (input data correlated with prior):
  chi2/dof [dof] = 0.63 [69]    Q = 0.99    logGBF = 1604.6    itns = 12

Parameters:
  log(etas:a) 0   -1.52243 (70)      [ -1.2 (1.0) ]
              1      -1.34 (50)      [ -1.2 (1.0) ]
 log(etas:dE) 0   -0.87680 (27)      [ -0.92 (50) ]
              1      -0.41 (13)      [ -0.7 (1.0) ]
    log(Ds:a) 0   -1.53999 (93)      [ -1.2 (1.0) ]
              1     -1.455 (75)      [ -1.2 (1.0) ]
   log(Ds:dE) 0    0.18355 (13)      [  0.18 (17) ]
              1     -0.771 (38)      [ -0.7 (1.0) ]
   log(Ds:ao) 0     -2.700 (72)      [ -2.3 (1.0) ]
              1      -2.39 (12)      [ -2.3 (1.0) ]
  log(Ds:dEo) 0     0.3688 (64)      [  0.41 (24) ]
              1      -1.32 (22)      [ -0.7 (1.0) ]
        Vnn 0,0    0.76780 (76)      [  0.0 (1.0) ]
            0,1     -0.437 (33)      [  0.0 (1.0) ]
            1,0      0.065 (62)      [  0.0 (1.0) ]
            1,1       0.15 (77)      [  0.0 (1.0) ]
        Vno 0,0     -0.761 (34)      [  0.0 (1.0) ]
            0,1       0.33 (17)      [  0.0 (1.0) ]
            1,0     0.071 (399)      [  0.0 (1.0) ]
            1,1      -0.14 (97)      [  0.0 (1.0) ]

Settings:
  svdcut = (1e-15,1e-15)   svdnum = (None,None)    reltol/abstol = 1e-10/1e-10

Leading parameter chi2/dof [dof] = 0.56 [8]   Q = 0.8
```

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(Ds:ao)`, `log(Ds:dEo)` — compare the simulated results with the `nterm=4` results from the original fit, as these were used to define `pexact`).

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

Symbols

A

B

C

D

E

F

L

Q

S