corrfitter Documentation

Release 3.3

G.P. Lepage

February 12, 2013

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.

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']
[0.159791 +- 4.13311e-06 0.0542088 +- 3.06973e-06 ... ]
>>> print data['Gab']
[0.156145 +- 1.83572e-05 0.102335 +- 1.5199e-05 ... ]
```

`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 make_data('mcfile')

`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

def make_data(filename):
    dset = gvar.dataset.Dataset(filename)
    return gvar.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.GVar`s obtained by averaging over the `Gaa` data in the `'mcfile'`, and `data['Gab']` is a similar array for the `Gab` correlator.

1.2.2 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 make_prior(N)

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 lsqfit
import gvar

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

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

The priors assign an *a priori* gaussian or normal distribution to each parameter. It is possible instead to assign a log-normal distribution, which forces the parameter to be positive. This is done by choosing a label in the prior that begins with "log": for example, `'logdE'` instead of `'dE'`. The fitter implements the log-normal distribution by using the parameter's logarithm, instead of the parameter itself, as a new fit parameter; the logarithm has a gaussian/normal distribution. The original parameter is recovered by taking the exponential of the new fit parameter.

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 by changing the corresponding labels in `make_prior(N)` (the labels need not be changed in `make_models()`), and taking logarithms of the corresponding prior values:

```
from gvar import log                            # numpy.log() works too

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'] = [log(gvar.gvar(0.1, 0.5)) for i in range(N)]
    prior['b'] = [gvar.gvar(1., 5.) for i in range(N)]
    prior['logdE'] = [log(gvar.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.

1.2.4 print_results(fit,prior,data)

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()`.)

Note that only three lines in `print_results(fit,prior,data)` would change if we had used log-normal priors for `a` and `dE`, as discussed in the previous section:

```
from gvar import exp                        # numpy.exp() works too
...
a = exp(fit.p['loga'])                      # array of a[i]s
...
dE = exp(fit.p['logdE'])                    # array of dE[i]s
...
inputs = {'loga':prior['loga'], 'b':prior['b'], 'logdE':fit.prior['logdE'],
          'data':[data[k] for k in data]}
...
```

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`.

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

Here the `nterm` parameter in `corrfitter.CorrFitter` 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.)` or `gvar.gvar(0.001,1.)`. In some situations an *svd* cut (see below) can also help.

1.4 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.

An *svd* cut can be applied to the covariance matrix for the data by specifying parameters `svdcut` and/or `svdnum`. (See documentation for `lsqfit`; it is useful 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.)

1.5 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.

1.6 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 `V`s 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).

1.7 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).

1.8 New Models

Classes to describe new models are usually derived from `corrfitter.BaseModel`. These can be for fitting new types of correlators. They can also be used in other ways — for example, to add constraints. Imagine a situation where one wants to constrain the third energy (`E2`) in a fit to be `0.60(1)`. This can be accomplished by adding `E2_Constraint()` to the list of models in `corrfitter.CorrFitter` where:

```
import gvar
import corrfitter

class E2_Constraint(corrfitter.BaseModel):
    def __init__(self):
        super(E2_Constraint,self).__init__('E2-constraint') # data tag

    def fitfcn(self,x,p):
        dE = gvar.exp(p['logdE'])
        return sum(dE[:3])              # E2 formula in terms of p

    def builddata(self,d):
        return gvar.gvar(0.6,0.01)      # E2 value

    def buildprior(self, prior, nterm):
        return {}
```

Any number of constraints like this can be added to the list of models.

Note that this constraint could instead be built into the priors for `logdE` by introducing correlations between different parameters.

1.9 Implementation

`corrfitter.CorrFitter` allows models to specify how many exponentials to include in the fit function (using parameters `nterm`, `ntermа` 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, though perhaps a little less robust.

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.

1.10 Correlator Model Objects

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

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

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

class `corrfitter.`**Corr2** (*datatag*, *tdata*, *tfit*, *a*, *b*, *dE*, *s=1.0*, *tp=None*, *othertags=None*)

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

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

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

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

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

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

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

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

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

The fit parameters are specified by the keys corresponding to these parameters in a dictionary of priors supplied by `corrfitter.CorrFitter`. The keys are strings and are also used to access fit results. Any key that begins with "log" is assumed to refer to the logarithm of the parameter in question (that is, the exponential of the fit-parameter is used in the formula for `Gab(t)`.) This is useful for forcing `an`, `bn` and/or `dE` to be positive.

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

Parameters

- **datatag** (*string*) – Key used to access correlator data in the input data dictionary (see `corrfitter.CorrFitter`). `data[self.datatag]` is (1-d) array containing the correlator values (`gvar.GVars`) if `data` is the input data.
- **a** (string, or two-tuple of strings and/or `None`) – Key identifying the fit parameters for the source amplitudes `an` in the dictionary of priors provided by `corrfitter.CorrFitter`; or a two-tuple of keys for the source amplitudes `(an, ao)`. The corresponding values in the dictionary of priors are (1-d) arrays of prior values with one term for each `an[i]` or `ao[i]`. Replacing either key by `None` causes the corresponding term to be dropped from the fit function. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **b** (string, or two-tuple of strings and/or `None`) – Same as `self.a` but for the sinks `(bn, bo)` instead of the sources `(an, ao)`.
- **dE** (string, or two-tuple of strings and/or `None`) – Key identifying the fit parameters for the energy differences `dEn` in the dictionary of priors provided by `corrfitter.CorrFitter`; or a two-tuple of keys for the energy differences `(dEn, dEo)`. The corresponding values in the dictionary of priors are (1-d) arrays of prior values with one term for each `dEn[i]` or `dEo[i]`. Replacing either key by `None` causes the corresponding term to be dropped from the fit function. These keys are used to label the corresponding parameter arrays in the fit results as well as in the prior.
- **s** (*number or two-tuple of numbers*) – Overall factor `sn`, or two-tuple of overall factors `(sn, so)`.
- **tdata** (*list of integers*) – The `ts` corresponding to data entries in the input data. Note that `len(self.tdata) == len(data[self.datatag])` is required if `data` is the input data dictionary.

- **tfit** (*list of integers*) – List of `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*)

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*)
Return fit function for parameters `p`.

class `corrfitter.`**BaseModel**(*datatag*)
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.

_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.

1.11 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=None*, *svdnum=None*, *tol=1e-10*, *maxit=500*, *nterm=None*, *ratio=True*)
Nonlinear least-squares fitter for a collection of correlators.

Parameters

- **models** (*list of correlator models*) – Correlator models used to fit statistical input data.
- **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` (the default), use ratio corrections for fit data when the prior specifies more terms than are used in the fit. If `False`, use difference corrections (see implementation notes, above).

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(*prior*)

Create fit function, with support for log-normal, etc. priors.

buildprior(*prior*, *nterm=None*)

Build correctly sized prior for 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*, ***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.

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.

1.12 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`).

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX

C

INDEX