High Performance Python

Week 4: Fitting Functions to Data

Overview

- There are several data fitting utilities available
- We will focus on two:
  - `scipy.optimize`
  - `lmfit.minimize`
- You can fit any arbitrary function that you define
- It is possible to constrain given parameters during the fit
- The canned packages come with useful diagnostics

Important Disclaimers

- We have focused on optimizing code speed in past sessions. Now, however, we are ignoring speed considerations.
- You sometimes must be careful to ensure you are getting physical results from fits.

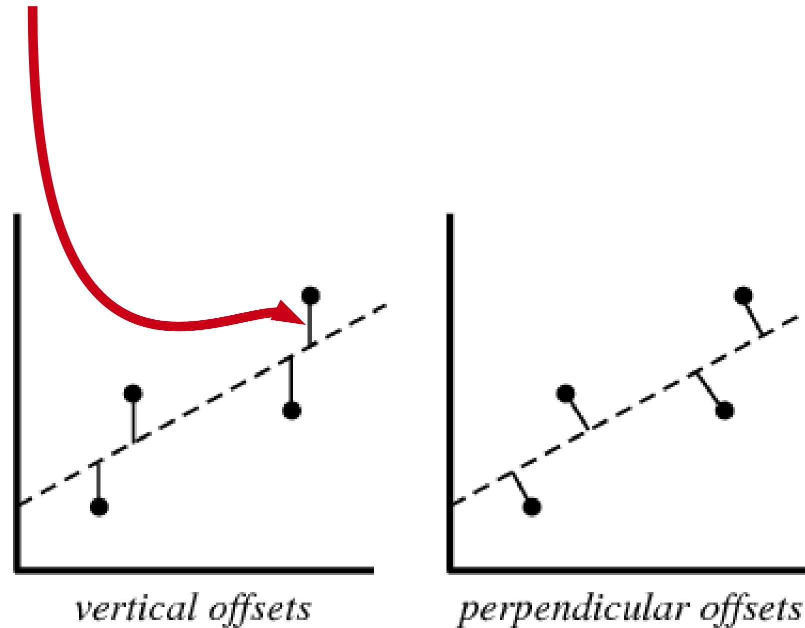
Fitting Basics

- The 'fitting' we discuss here is an iterative process.
- First, we define our desired function, and calculate values given certain parameters
- Calculate the difference between the initial and the new values
- We want to minimize this difference
  - Specifically, we generally minimize the sum of the squares of these differences
- Several examples at http://www.scipy.org/Cookbook/FittingData

Fitting Basics

- Minimization is usually done by the method of least squares fitting.
- In math-speak, we minimize:

$$\sum difference^2 = \sum [y_i(actual) - y_i(model)]^2$$



Fitting Basics

- There are several algorithms available for this minimization.
- The most common is the Levenberg-Marquardt
  - Susceptible to finding local minima instead of global
  - Fast
  - Usually well-behaved for most functions, etc.
  - By far the most tested of methods, with many accompanying statistics implemented
- Other methods include the Nelder-Mead, L-BFGS-B, and Simulated Annealing algorithms
- See, e.g., http://cars9.uchicago.edu/software/python/lmfit/fitting.html

Goodness-of-Fit

- There are several statistics that can help you determine the goodness-of-fit
- Most people use
  - reduced chi-squared, χ^2_ν
  - Standard error
  - Etc...
- You get these for free with `lmfit.minimize`
- This subtopic is much too large to discuss in detail here. Review your Bevington or other good data analysis text.

Example 1

No constraints. Fit a quadratic.

Start Simple

- First, let's try fitting a simple quadratic to some fake data.

$$y = a \cdot x^2 + b \cdot x + c$$

- What we will do:
  - Generate some data for the example
  - Define the function we wish to fit
  - Use `scipy.optimize` to do the actual optimization

Quadratic Fit

- Generate the example data.
  - The x-data is an array from -3 to 10
  - The y-data is x^2, with some random noise added.
  - Let's put our initial guesses for the coefficients a,b,c into a list called p0 (for fit parameters)

```
import numpy as np

#Generate the arrays
xarray1=np.arange(-3,10,.2)
yarray1=xarray1**2

#Adding noise
yarray1+=np.random.randn(yarray1.shape[0])*2

p0=[2,2,2] #Our initial guesses for our fit parameters
```

Quadratic Fit

- Cheap & easy method for polynomials (only):
  - `scipy.polyfit()`
- This method involves the least amount of setup
- It simply outputs an array of the coefficients that best fit the data to the specified polynomial order

```
from scipy import polyfit

fitcoeffs=polyfit(xarray1,yarray1,2)

print fitcoeffs
# --> Returns array
#     (<coeff. order 2>, <coeff. order 1>, <coeff. order 0>)
```

Quadratic Fit

- Define the function you want to fit.
  - Remember, p will be our array of initial guesses to the fit parameters, the coefficients a, b, c

```
def quadratic(p,x):
    y_out=p[0]*(x**2)+p[1]*x+p[2]
    return y_out
```

Quadratic Fit

- Alternatively, define with a lambda function
  - These are basically one-line (compact) definitions
  - The two definitions below are equivalent:

```
def quadratic(p,x):
    y_out=p[0]*(x**2)+p[1]*x+p[2]
    return y_out

#Is the same as

quadratic = lambda p,x: p[0]*(x**2)+p[1]*x+p[2]
```

Quadratic Fit

- Here we define a function that returns the difference between the fit iteration value and the initial data

```
quadraticerr = lambda p,x,y: quadratic(p,x)-y
```

- This difference or residual is the quantity that we will minimize with `scipy.optimize`

Quadratic Fit

- Call the least-squares optimization routine with scipy.optimize.leastsq()
- The parameters you fit are stored in the zeroth element of the output

```
fitout=leastsq(quadraticerr,p0[:],args=(xarray1,yarray1))

paramsout=fitout[0] #These are the fitted coefficients
covar=out[1] #This is the covariance matrix output

print('Fitted Parameters:\na = %.2f , b = %.2f , c = %.2f'
       % (paramsout[0],paramsout[1],paramsout[2]))
```

Quadratic Fit

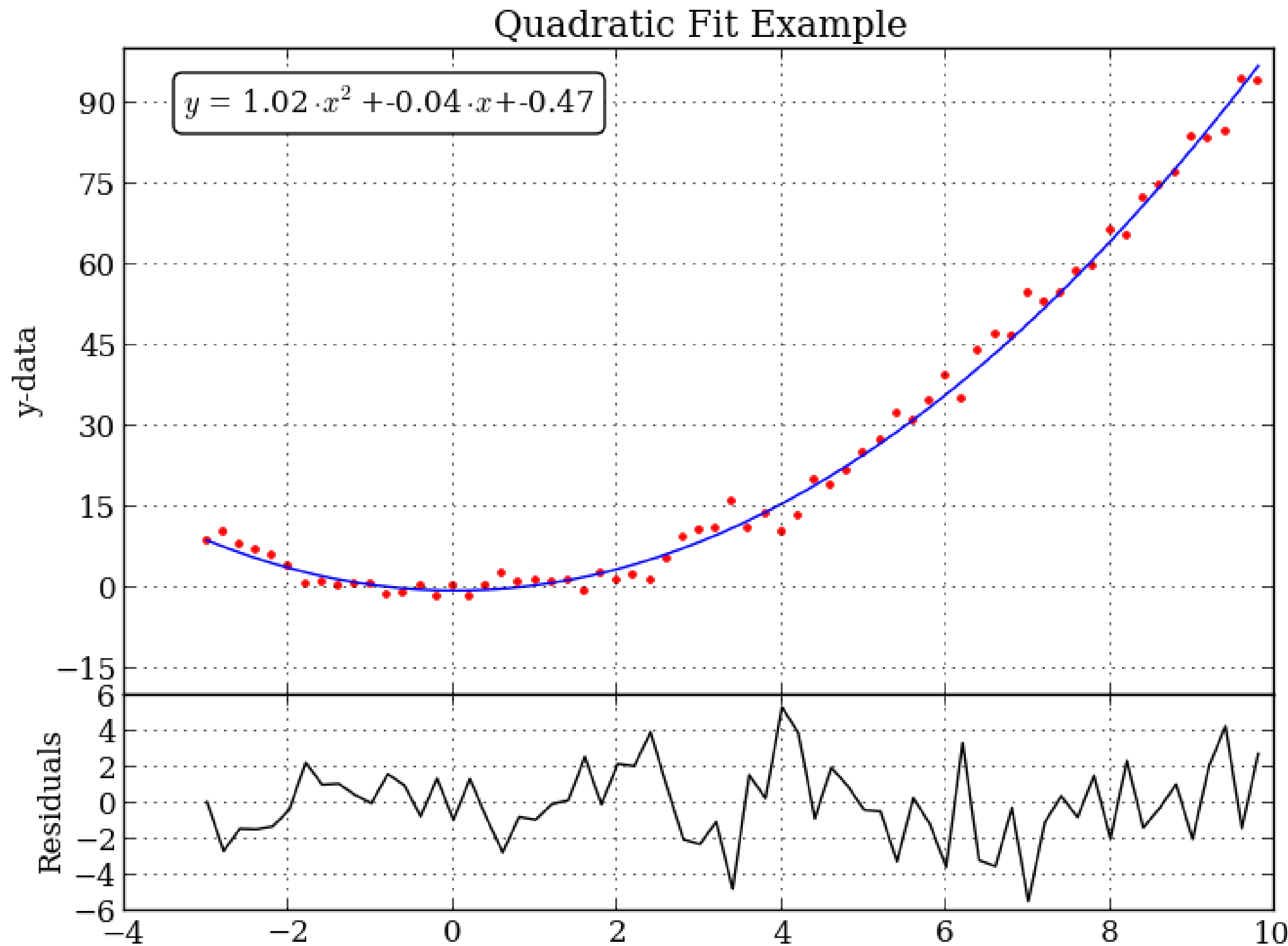
- Now to get an array values for the results, just call your function definition with the fitted parameters
- Residuals, of course, will just be that minus the original data.

```
fitarray1=quadratic(paramsout,xarray1)

residualarray1=fitarray1-yarray1
```

Quadratic Fit

- The results for our example will look like this:



Example 2

Introduce constraints

Constraining Parameters

- Often we want to set limits on the values that our fitted parameters can have
  - For example, if you know that something can't be negative, etc.
- To do this, we can use `scipy.optimize.minimize()`
- Another useful package is `lmfit.minimize()`
  - We create an `lmfit.Parameters()` object
  - We can set limits for the parameters to be fit
  - We can even tell some params not to vary at all
- The `Parameters()` object updates with every iteration

Power Law Example

- For this example, let's use some real data.
- The M-σ relation is a well-known empirical correlation between the stellar velocity dispersion of a galaxy bulge (σ) and the galaxy's central black hole (M)
- The data here comes from Gultekin et al. 2009 – http://arxiv.org/abs/0903.4897

```
dat=np.genfromtxt('M-SigmaData.csv',skiprows=2)

M=dat[:,0]     #The y-data
Sigma=dat[:,1] #The x-data
```

Power Law Example

- The correlation is known to follow a power law distribution, which has the form

$$y = a \cdot x^{b}$$

- This is easy enough to fit as is, but let's be clever and fancy, note that this is essentially just fitting a *line* in log-space.
  - Why? It's faster to fit a line than an exponent.

$$\log(y) = \log(a) + \log(x^{b}) = \alpha + \beta \cdot \log(x)$$

Power Law Example

- The correlation is known to follow a power law distribution, which has the form

$$y = a \cdot x^{b}$$

- This is easy enough to fit as is, but let's be clever and fancy, note that this is essentially just fitting a *line* in log-space.
  - Why? It's faster to fit a line than an exponent.

$$\log(y) = \log(a) + \log(x^{b}) = \alpha + \beta \cdot \log(x)$$

Power Law Example

- Set up the `lmfit.Parameters()` object

```
from lmfit import minimize, Parameters

params=Parameters()

params.add('alpha',value=8,vary=True)
params.add('beta',value=4,vary=True,max=5.0) #,min=...
```

- We can access a particular parameter's current value with, e.g., `params['alpha'].value`

Power Law Example

- Now define our function to fit and its residual

```
power=lambda p,x: params['alpha'].value+params['beta'].value*x

powererr=lambda p,x,y: power(p,x)-y
```

- Now the fit only requires one line of code:

```
fitout2=minimize(powererr,params,args=(np.log10(Sigma),np.log10(M)))
```

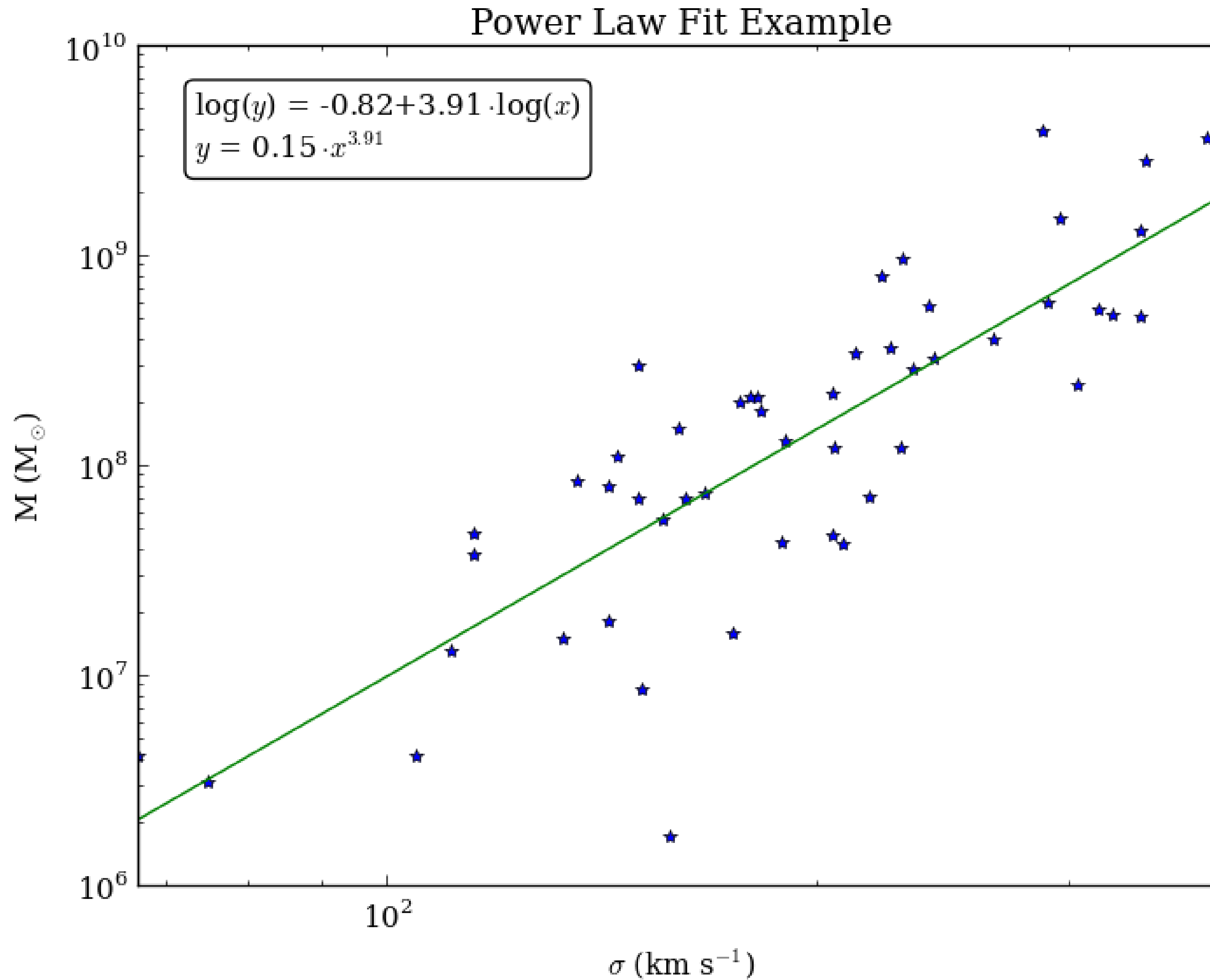
Power Law Example

- Now if we want an array of the fitted data values:

```
fitarray2=10**params['alpha'].value* Sigma**params['beta'].value
```

Power Law Example

- If we plot the results, we get this:



Some Freebies with lmfit.minimize()

- `lmfit.minimize()` gives you some free goodness-of-fit diagnostics:
  - output.nfev → number of function evaluations
  - output.lmdif_message → message from scipy.optimize.leastsq
  - output.nvarys → number of variables in fit
  - output.ndata → number of data points
  - output.nfree → degrees of freedom in fit
  - output.residual → residual array
  - output.chisqr → chi-square
  - output.redchi → reduced chi-square
- Again, see

  http://cars9.uchicago.edu/software/python/lmfit/

Example 3

More complex functions, with constraints

More Complex Example

- Let's use more real data for a typical real-world application: fitting a profile to spectral data.
- The data: stacked velocity-amplitude spectra from a VLA observation
- The functions:
  - A modified Gaussian to include Hermite polynomials (approximations to skew and kurtosis)
  - A double gaussian (gaus1 + gaus2 = gausTot)
- Data available on this workshop's website.

More Complex Example

- Let's use more real data for a typical real-world application: fitting a profile to spectral data.
- The data: stacked velocity-amplitude spectra from a VLA observation
- The functions:
  - A modified Gaussian to include Hermite polynomials (approximations to skew and kurtosis)
  - A double gaussian (gaus1 + gaus2 = gausTot)
- Data available on this workshop's website.
- DON'T PANIC – this last example is just meant to showcase some complex things you can do.

Gaussian Functions

- Standard Gaussian:

$$f(x) = A\, e^{-\frac{g^2}{2}} \qquad g = \frac{x - x_c}{\sigma}$$

- Multiple Gaussians:

$$F(x) = \sum_i f_i(x) = A_1\, e^{\frac{-{g_1}^2}{2}} + A_2\, e^{\frac{-{g_2}^2}{2}} + \ldots$$

Gaussian Functions

- Gauss-Hermite Polynomial:

$$f(x) = A\, e^{-\frac{g^2}{2}} \left[1 + h_3 \left(-\sqrt{3}\, g + \frac{2}{\sqrt{3}}\, g^3 \right) + h_4 \left(\frac{\sqrt{6}}{4} - \sqrt{6}\, g^2 + \frac{\sqrt{6}}{3}\, g^4 \right) \right]$$

- H_3 → (Fisher) Skew: asymmetric component

$$\xi_1 \approx 4\sqrt{3}\, h_3$$

- H_4 → (Fisher) Kurtosis: how 'fat' the tails are

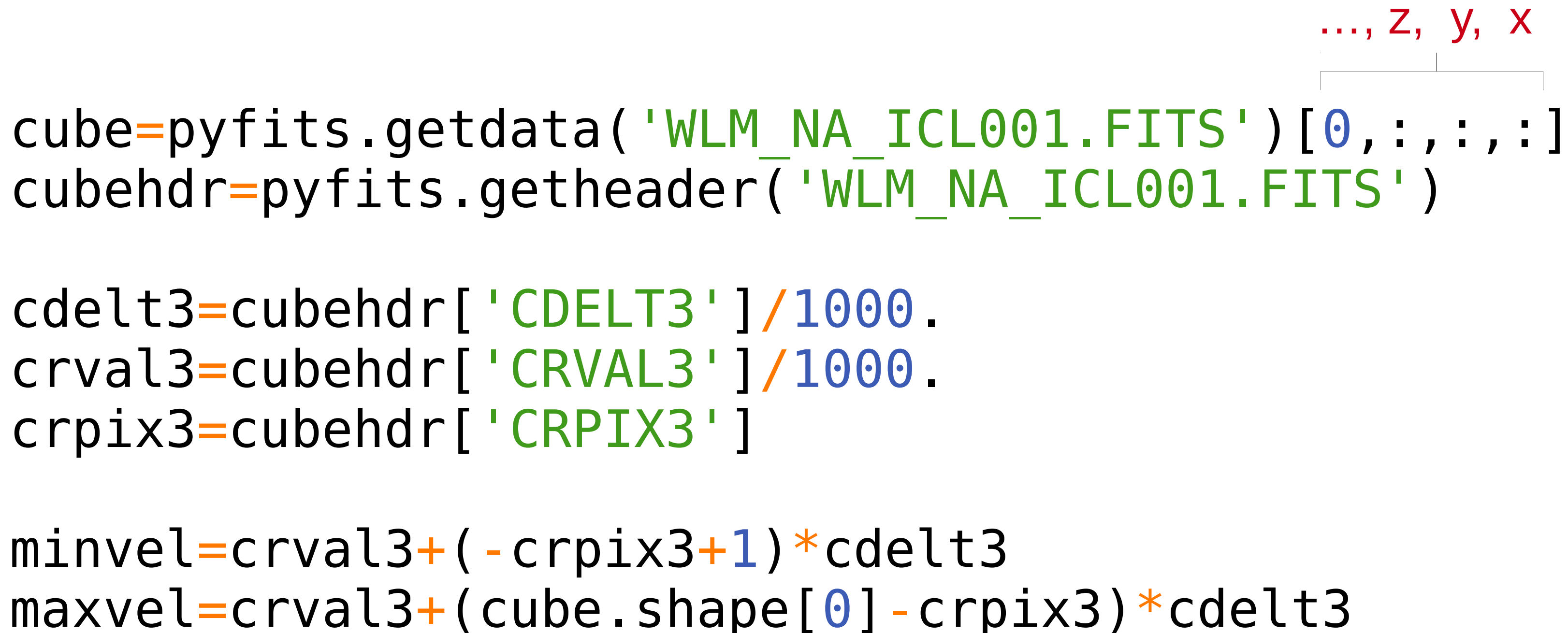
$$\xi_2 \approx 3 + 8\sqrt{6}\, h_4$$

$$\xi_f = \xi_2 - 3$$

$$\xi_f \approx 8\sqrt{6}\, h_4$$

Aside: pyfits

- Astronomy folks often work with .fits files
- Access them with pyfits
- Here we use the pyfits 'convenience functions'



Aside: pyfits

- We want to stack all the spectra in the image for this particular example:

```
stackspec=np.sum(np.sum(cube,axis=2),axis=1)

vels=np.arange(minvel,maxvel+int(cdelt3),cdelt3)
```

Gaussian Fits

- If you don't care about the .fits file business, you can just use the csv file on the website
  - Velocity values in column 0
  - Stacked spectra values at each velocity in column 1

```
dat=np.genfromtxt('GausFitData.csv',skiprows=1)

vels=dat[:,0]
stackspec=dat[:,1]
```

Gaussian Fits

- Set up the `lmfit.Parameters()` and define the Gauss-Hermite function:

```
p_gh=Parameters()
p_gh.add('amp',value=np.max(stackspec),vary=True);
p_gh.add('center',value=vels[50],min=np.min(vels),max=np.max(vels));
p_gh.add('sig',value=3*abs(cdelt3),min=abs(cdelt3),max=abs(maxvel-minvel));
p_gh.add('skew',value=0,vary=True,min=None,max=None);
p_gh.add('kurt',value=0,vary=True,min=None,max=None);

def gaussfunc_gh(paramsin,x):
    amp=paramsin['amp'].value
    center=paramsin['center'].value
    sig=paramsin['sig'].value
    c1=-np.sqrt(3); c2=-np.sqrt(6); c3=2/np.sqrt(3);
    c4=np.sqrt(6)/3; c5=np.sqrt(6)/4
    skew=paramsin['skew'].value
    kurt=paramsin['kurt'].value

    gaustot_gh=amp*np.exp(-.5*((x-center)/sig)**2)* \
               (1+skew*(c1*((x-center)/sig+c3*((x-center)/sig)**3)+ \
               kurt*(c5+c2*((x-center)/sig)**2+c4*((x-center)/sig)**4))
    return gaustot_gh
```

Gaussian Fits

- Now do the same for the double gaussian:

```
p_2g=Parameters()
p_2g.add('amp1',value=np.max(stackspec)/2.,
         min=.1*np.max(stackspec),max=np.max(stackspec));
p_2g.add('center1',value=vels[50+10],min=np.min(vels),max=np.max(vels));
p_2g.add('sig1',value=2*abs(cdelt3),min=abs(cdelt3),max=abs(maxvel-minvel));
p_2g.add('amp2',value=np.max(stackspec)/2.,
         min=.1*np.max(stackspec),max=np.max(stackspec));
p_2g.add('center2',value=vels[50-10],min=np.min(vels),max=np.max(vels));
p_2g.add('sig2',value=3*abs(cdelt3),min=abs(cdelt3),max=abs(maxvel-minvel));

def gaussfunc_2g(paramsin,x):
    amp1=paramsin['amp1'].value; amp2=paramsin['amp2'].value;
    center1=paramsin['center1'].value; center2=paramsin['center2'].value;
    sig1=paramsin['sig1'].value; sig2=paramsin['sig2'].value;
    gaus1=amp1*np.exp(-.5*((x-center1)/sig1)**2)
    gaus2=amp2*np.exp(-.5*((x-center2)/sig2)**2)
    gaustot_2g=(gaus1+gaus2)
    return gaustot_2g
```

Gaussian Fits

- Now define the residual functions, as well as a single Gaussian function

```
gausserr_gh = lambda p,x,y: gaussfunc_gh(p,x)-y
gausserr_2g = lambda p,x,y: gaussfunc_2g(p,x)-y

gausssingle = lambda a,c,sig,x: a*np.exp(-.5*((x-c)/sig)**2)
```

Gaussian Fits

- I want to constrain the fit parameters again
  - For example, I don't want to allow negative-amplitude Gaussians...
- Use `lmfit.minimize` again

```
fitout_gh=minimize(gausserr_gh,p_gh,args=(vels,stackspec))

fitout_2g=minimize(gausserr_2g,p_2g,args=(vels,stackspec))
```

Gaussian Fits

- For fits with many parameters, making shorthand definitions can help keep things tidy:

```
pars_gh=[p_gh['amp'].value,
         p_gh['center'].value,
         p_gh['sig'].value,
         p_gh['skew'].value,
         p_gh['kurt'].value]

pars_2g=[p_2g['amp1'].value,
         p_2g['center1'].value,
         p_2g['sig1'].value,
         p_2g['amp2'].value,
         p_2g['center2'].value,
         p_2g['sig2'].value]
```

Gaussian Fits

- Finally, if you want to create arrays and residuals of the final fit values:

```
fit_gh=gaussfunc_gh(pars_gh,vels)
fit_2g=gaussfunc_2g(pars_2g,vels)

resid_gh=fit_gh-stackspec
resid_2g=fit_2g-stackspec
```

Gaussian Fits

- The results look like this: