Introduction to fitting and optimization using Python libraries

Zachary Morgan, Oak Ridge National Laboratory

2023 NXS: Data Science for Neutron Scattering
August 11, 2023

This research used resources at the Spallation Neutron Source, a DOE Office of Science User Facility operated by the Oak Ridge National Laboratory

Non-linear least squares

- Set of m observations

$$(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_m, y_m)$$

- Model function

$$\hat{y} = f(x, \beta_1, \beta_2, \beta_3, \ldots, \beta_n) = f(x, \boldsymbol{\beta})$$

- Vector of $n < m$ fitting parameters

$$\boldsymbol{\beta} = (\beta_1, \beta_2, \beta_3, \ldots, \beta_n)$$

- Residuals

$$r_i = y_i - f(x_i, \boldsymbol{\beta})$$

- Minimization sum of squared residuals

$$S = \sum_{i=1}^{m} r_i^2$$

- Gradient of objective function

$$\begin{aligned} \frac{\partial S}{\partial \beta_j} &= 2 \sum_{i=1}^{m} \frac{\partial r_i}{\partial \beta_j} r_i \\ &= -2 \sum_{i=1}^{m} J_{ij} r_i = 0 \end{aligned} \qquad j = 0, 1, 2, \ldots, n$$

Strang, G., *Computational science and engineering*, Wellesley-Cambridge Press: Wellesley, MA, 2007.

Non-linear least squares

- Change in residual

$$\Delta y_i = y_i - f(x_i, \boldsymbol{\beta}^k)$$

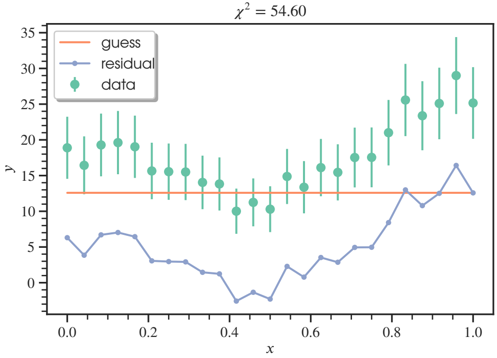
- Residual approximation

$$r_i \approx \Delta y_i - \sum_{s=1}^{n} J_{is} \Delta\beta_s$$

- Normal equations

$$\sum_{i=1}^{m} \sum_{s=1}^{n} J_{ij} J_{is} \Delta\beta = \sum_{i=1}^{m} J_{ij} \Delta y_i \quad j = 0, 1, 2, \ldots, n$$

$$(\boldsymbol{J}^\top \boldsymbol{J}) \Delta\boldsymbol{\beta} = \boldsymbol{J}^\top \Delta\boldsymbol{y}$$



Fitting a one-dimensional function with uncertainties: initial guess

---

*If Jacobian function cannot be analytically derived, finite difference methods can be used as approximation. Accuracy (e.g. step size or higher order differences) typically needs to be balanced with the cost of evaluating difference equations.

Non-linear least squares

- Jacobian matrix *

$$J_{ij} = -\frac{\partial r_i}{\partial \beta_j} = \frac{\partial f(x_i, \boldsymbol{\beta})}{\partial \beta_j} \qquad \boldsymbol{J}_i = \frac{\partial f(x_i, \boldsymbol{\beta})}{\partial \boldsymbol{\beta}}$$

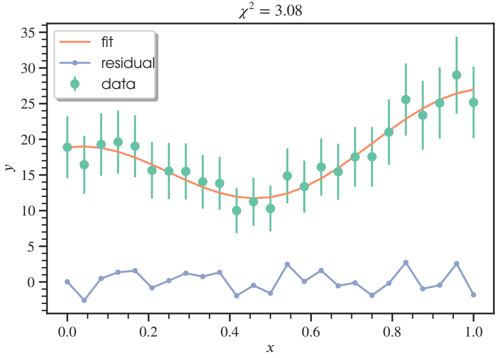
- Iterative refinement shifting parameter $\Delta\beta_j$ from step k to step $k+1$

$$\beta_j \approx \beta_j^{k+1} = \beta_j^k + \Delta\beta_j$$

- First-order Taylor expansion

$$f(x_i, \boldsymbol{\beta}) \approx f(x_i, \boldsymbol{\beta}^k) + \sum_{j=1}^{n} \frac{\partial f(x_i, \boldsymbol{\beta}^k)}{\partial \beta_j} \Delta\beta_j$$

$$= f(x_i, \boldsymbol{\beta}^k) + \sum_{j=1}^{n} J_{ij} \Delta\beta_j$$



Fitting a one-dimensional function with uncertainties: final fit

Non-linear weighted least squares

- Set of m observations with uncertainties

$$(x_1, y_1, \sigma_1), (x_2, y_2, \sigma_2), \ldots, (x_m, y_m, \sigma_m)$$

- Mean squared weighted deviation

$$\chi^2 = \sum_{i=1}^{m} \frac{[y_i - f(x_i, \boldsymbol{\beta})]^2}{\sigma_i^2}$$

- Diagonal weight matrix †

$$W_{ii} = \frac{1}{\sigma_i^2}$$

- Weighted normal equations

$$(\boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J})\Delta\boldsymbol{\beta} = \boldsymbol{J}^\intercal \boldsymbol{W} \Delta \boldsymbol{y}$$

- Covariance matrix

$$\boldsymbol{C} = (\boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J})$$

- Parameter uncertainties $\beta_j \pm \epsilon_j$

$$\epsilon_j = \sqrt{C_{jj}}$$

- Reduced chi-squared statistic $\nu = m - n$

$$\chi_\nu^2 = \frac{\chi^2}{\nu} = \frac{1}{m-n} \sum_{i=1}^{m} \frac{[y_i - f(x_i, \boldsymbol{\beta})]^2}{\sigma_i^2}$$

- Standard errors ‡ $\chi^2(\beta_j \pm \hat{\epsilon}_j) = \chi^2(\beta_j) + 1$

$$\hat{\epsilon}_j = \sqrt{\chi_\nu^2 C_{jj}} = \frac{\chi \epsilon_j}{\sqrt{m-n}}$$

---

† Assumes uncertainties are independent and uncorrelated.

‡ Useful when the original weights are unreliable.

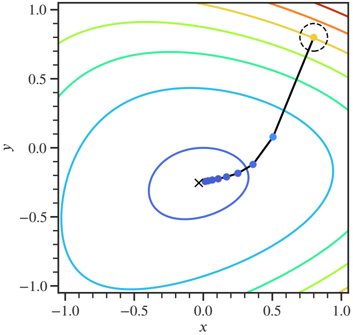
Non-linear weighted least squares

Various methods
- Gauss-Newton method
- Levenberg–Marquardt algorithm

Matrix methods
- QR decomposition
- Singular Value decomposition

Gradient methods
- Newton's method
- Davidon–Fletcher–Powell method
- Steepest descent
- Conjugate gradient search

Direct search methods
- Alternating variable search
- Nelder–Mead (simplex) search



Gauss-Newton method for minimizing a two-dimensional function

Levenberg–Marquardt algorithm

Interpolation between Gauss–Newton algorithm and steepest descent

- Weighted normal equations

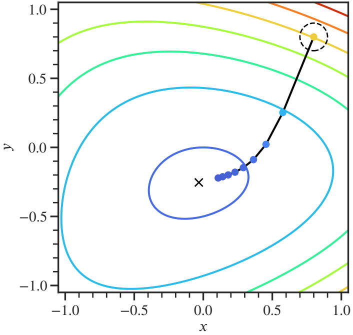
$$(\boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J} + \lambda \boldsymbol{I})\Delta\boldsymbol{\beta} = \boldsymbol{J}^\intercal \boldsymbol{W} \Delta\boldsymbol{y}$$

- Steepest descent §

$$\lambda \boldsymbol{I} \gg \boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J} \quad \Delta\boldsymbol{\beta} = \frac{1}{\lambda} \boldsymbol{J}^\intercal \boldsymbol{W} \Delta\boldsymbol{y}$$

- Gauss–Newton method ¶

$$\boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J} \gg \lambda \boldsymbol{I} \quad (\boldsymbol{J}^\intercal \boldsymbol{W} \boldsymbol{J})\Delta\boldsymbol{\beta} = \boldsymbol{J}^\intercal \boldsymbol{W} \Delta\boldsymbol{y}$$



Steepest descent method for minimizing a two-dimensional function

§ Converges slowly, but solution will converge with appropriate step size.

¶ Converges rapidly in close proximity to minima, but solution may diverge.

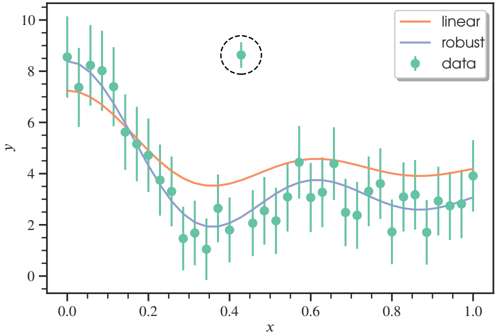
Robust non-linear optimization

Robust optimization using loss functions

$$\sum_{i=1}^{m} \rho(W_{ii} r_i^2))$$

Loss functions

- Maximum likelihood (least squares)
  $\rho(z) = z$
- Soft approximation of L_1 norm
  $\rho(z) = 2(\sqrt{1+z} - 1)$
- Huber $\rho(z) = \begin{cases} z & z < 1 \\ \sqrt{z} - 1 & z > 1 \end{cases}$
- Cauchy $\rho(z) = \ln\left(1+z\right)$
- Arctan $\rho(z) = \arctan z$



Robust optimization using soft approximation of L_1 norm improves the fit when data contains outliers

Constrained optimization

Minimize a function $\chi^2(\boldsymbol{\beta})$ subject to

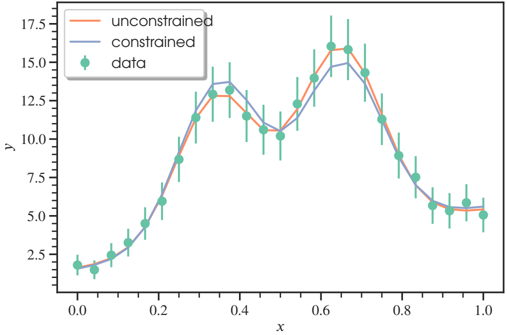
$$
\begin{aligned}
g_k(\boldsymbol{\beta}) &= c_k \qquad k = 1, 2, \ldots \\
h_l(\boldsymbol{\beta}) &\geq d_l \qquad l = 1, 2, \ldots
\end{aligned}
$$

Equality constraints

- Substitution
- Lagrange multipliers

Inequality constraints

- Penalty methods
- Linear and nonlinear programming
- Quadratic programming
- Branch and bound
- First-choice bounding functions
- Bucket elimination



Fitting of two overlapping Gaussian peaks with equal widths, equal amplitudes, and peak overlap limited to two standard deviations between centers

Python libraries

Many fitting Python libraries are available

SciPy ‖

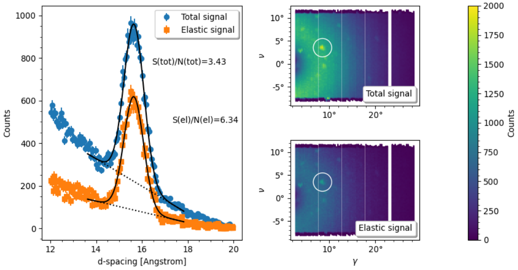
- `scipy.optimize.curve_fit`
- `scipy.optimize.least_squares`
- `scipy.optimize.leastsq`
- `scipy.optimize.minimize`

LMfit **

- `lmfit.minimize`

Mantid †† (GNU scientific library)

- `Fit`



Fitting single crystal peaks from a protein collected at CORELLI using Mantid

‖ `https://docs.scipy.org/doc/scipy/reference/optimize.html`

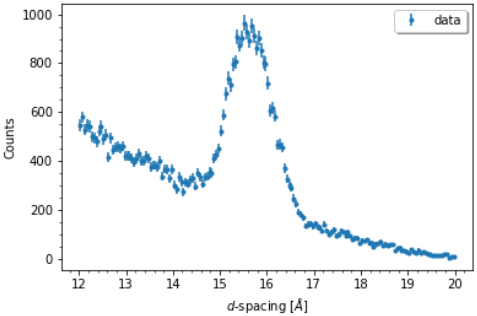
** `https://lmfit.github.io/lmfit-py/intro.html`

†† `https://www.gnu.org/software/gsl/`

Arnold, O. et al. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **2014**, *764*, 156–166.

Loading and plotting data

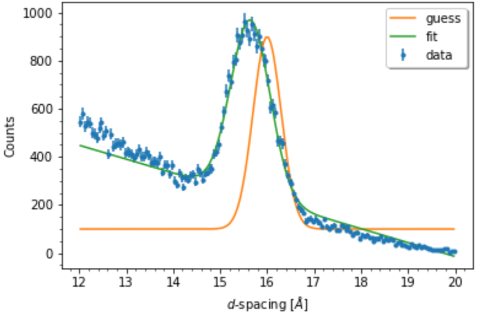
```
import numpy as np
import matplotlib.pyplot as plt

import scipy.optimize
import lmfit
import h5py

# --- load data ---

f = h5py.File('../data/total.nxs', mode='r')

ws = f['mantid_workspace_1/workspace/']
d_spacing_bin_edges = ws['axis1'][()]

counts  = ws['values'][()].flatten()
errors  = ws['errors'][()].flatten()

f.close()

# --- plot data ---

d_spacing = 0.5*(d_spacing_bin_edges[1:]+d_spacing_bin_edges[:-1])

fig, ax = plt.subplots(1,1)
ax.errorbar(d_spacing, counts, yerr=errors, fmt='.', label='data')
ax.legend(shadow=True)
ax.minorticks_on()
```



Plotted peak

Initializing and fitting model

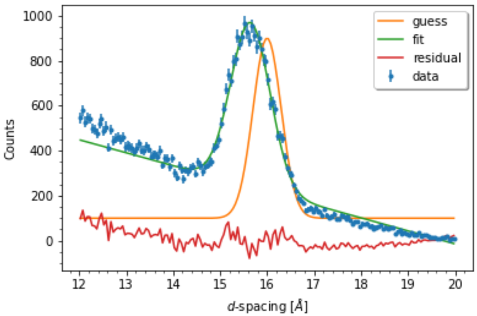
```
# --- define model ---

def model(d, A, mu, sigma, B, c):

    return A*np.exp(-0.5*(d-mu)**2/sigma**2)+B+c*d

A, mu, sigma, B, c = 800, 16, 0.3, 100, 0

p0 = (A, mu, sigma, B, c)

ax.plot(d_spacing, model(d_spacing, *p0), label='guess')
ax.legend(shadow=True)

# --- fit model ---

popt, pcov = scipy.optimize.curve_fit(model, d_spacing, counts, p0=p0,
                                      sigma=errors, absolute_sigma=True,
                                      method='lm')

ax.plot(d_spacing, model(d_spacing, *popt), label='fit')
ax.legend(shadow=True)
```



Final fit

Plotting residuals and calculating uncertanties

```
# --- plot residuals ---

def residual(d, counts, A, mu, sigma, B, c):

    return counts-model(d, A, mu, sigma, B, c)

ax.plot(d_spacing, residual(d_spacing, counts, *popt), label='residual')
ax.legend(shadow=True)
fig.savefig('scd-residual-total.png')

# --- calculate errors ---

perr = np.sqrt(np.diag(pcov))

print('Fitted uncertanties as 1-std using curve_fit')
print('A     = {:8.3f} ± {:5.3f}'.format(popt[0],perr[0]))
print('mu    = {:8.3f} ± {:5.3f}'.format(popt[1],perr[1]))
print('sigma = {:8.3f} ± {:5.3f}'.format(popt[2],perr[2]))
print('B     = {:8.3f} ± {:5.3f}'.format(popt[3],perr[3]))
print('c     = {:8.3f} ± {:5.3f}'.format(popt[4],perr[4]))
```



Final residuals

More examples

```python
# --- define weighted least squares problem ---

def weighted_deviations(x, d, counts, errors):

    A, mu, sigma, B, c = x

    return residual(d, counts, A, mu, sigma, B, c)/errors

sol = scipy.optimize.least_squares(weighted_deviations, x0=p0,
                                   args=(d_spacing, counts, errors),
                                   method='lm')

vals = sol.x

J = sol.jac
cov = np.linalg.inv(J.T.dot(J))

err = np.sqrt(np.diag(cov))

print('Fitted uncertanties as 1-std using least_squares')
print('A     = {:8.3f} ± {:5.3f}'.format(vals[0],err[0]))
print('mu    = {:8.3f} ± {:5.3f}'.format(vals[1],err[1]))
print('sigma = {:8.3f} ± {:5.3f}'.format(vals[2],err[2]))
print('B     = {:8.3f} ± {:5.3f}'.format(vals[3],err[3]))
print('c     = {:8.3f} ± {:5.3f}'.format(vals[4],err[4]))

chi2dof = np.sum(sol.fun**2)/(sol.fun.size-sol.x.size)
cov *= chi2dof

stderr = np.sqrt(np.diag(cov))

print('Fitted uncertanties as 1-stderr using least_squares')
print('A     = {:8.3f} ± {:5.3f}'.format(vals[0],stderr[0]))
print('mu    = {:8.3f} ± {:5.3f}'.format(vals[1],stderr[1]))
print('sigma = {:8.3f} ± {:5.3f}'.format(vals[2],stderr[2]))
print('B     = {:8.3f} ± {:5.3f}'.format(vals[3],stderr[3]))
print('c     = {:8.3f} ± {:5.3f}'.format(vals[4],stderr[4]))
```

```python
# --- use constranied optimization ---

def weighted_residual(params, d, counts, errors):

    A = params['A']
    mu = params['mu']
    sigma = params['sigma']
    B = params['B']
    c = params['c']

    return residual(d, counts, A, mu, sigma, B, c)/errors

params = lmfit.Parameters()
params.add('A', value=A, min=0, max=np.inf)
params.add('mu', value=mu, min=-np.inf, max=np.inf)
params.add('sigma', value=sigma, min=0, max=np.inf)
params.add('B', value=B, min=0, max=np.inf)
params.add('c', value=c, min=-np.inf, max=np.inf)

result = lmfit.minimize(weighted_residual, params,
                        args=(d_spacing, counts, errors))

print('Fitted uncertanties as 1-stderr using lmfit')
for key in params.keys():
    value, stderr = result.params[key].value, result.params[key].stderr
    print('{:7} = {:8.3f} ± {:5.3f}'.format(key,value,stderr))
```