

Table of Contents

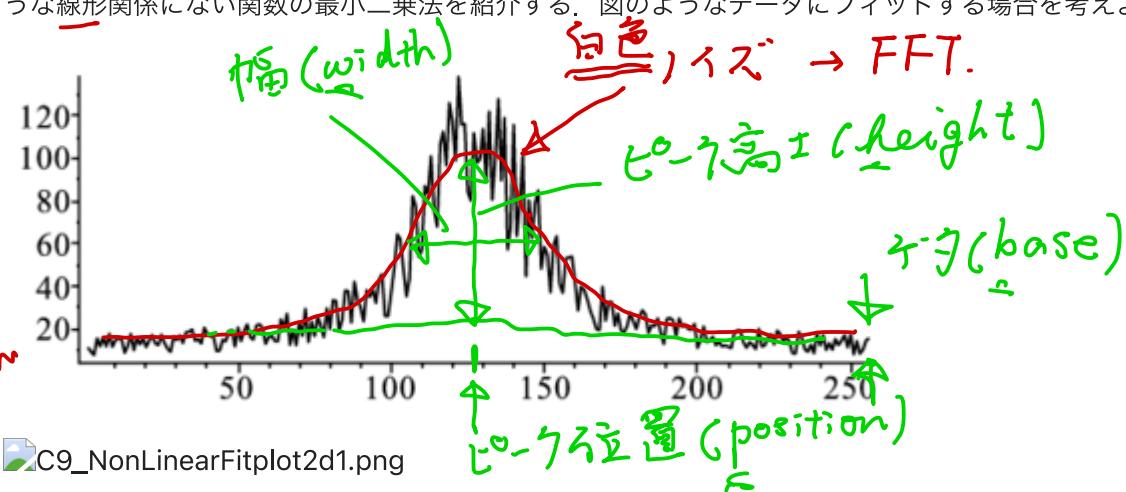
- 1 非線形最小2乗法の原理
- 2 python code
- 3 具体的な手順
- 4 pythonによる解法の指針
- 5 Gauss-Newton法に関するメモ
- 6 課題
 - 6.1 Gaussian(正規分布)へのフィット

非線形最小2乗法 (NonLinearFit)

file:/Users/bob/Github/TeamNishitani/jupyter_num_calc/nonlinearfit
https://github.com/daddygongon/jupyter_num_calc/tree/master/notebooks_p
cc by Shigeto R. Nishitani 2017-8

非線形最小2乗法の原理

前章では、データに近似的にフィットする最小二乗法を紹介した。ここでは、フィット式が多項式のような線形関係にない関数の最小二乗法を紹介する。図のようなデータにフィットする場合を考えよう。



このデータにあてはめるのはローレンツ関数,

$$F(x; \mathbf{a}) = a_1 + \frac{a_2}{a_3 + (x - a_4)^2}$$

$$= b + \frac{h}{\omega + (x - p)^2}$$

$$\frac{1}{x^2 + 1} + 0$$

走査 9565

である。この関数の特徴は、今まで見てきた関数と違いパラメータが線形関係になっていない。誤差関数は、今までと同様に

$$F = \sum a_i x_i(\alpha)$$

$$\chi^2(\mathbf{a}) = \sum_i^N d_i^2 = \sum_i^N (F(x_i; \mathbf{a}) - y_i)^2$$

で、 $\mathbf{a} = \{a_0, a_1, \dots\}$ をパラメータとして変えた時に最小となる値を求める点もかわらない。しかし、線形の最小二乗法のように微分しても一元の方程式にならず連立方程式を単に解くだけでは求められない。

そこで図のような2次関数の最小値を求める場合を考える。最小値の点 a_0 のまわりで、Taylor展開すると、 d, D をそれぞれの係数とすると、

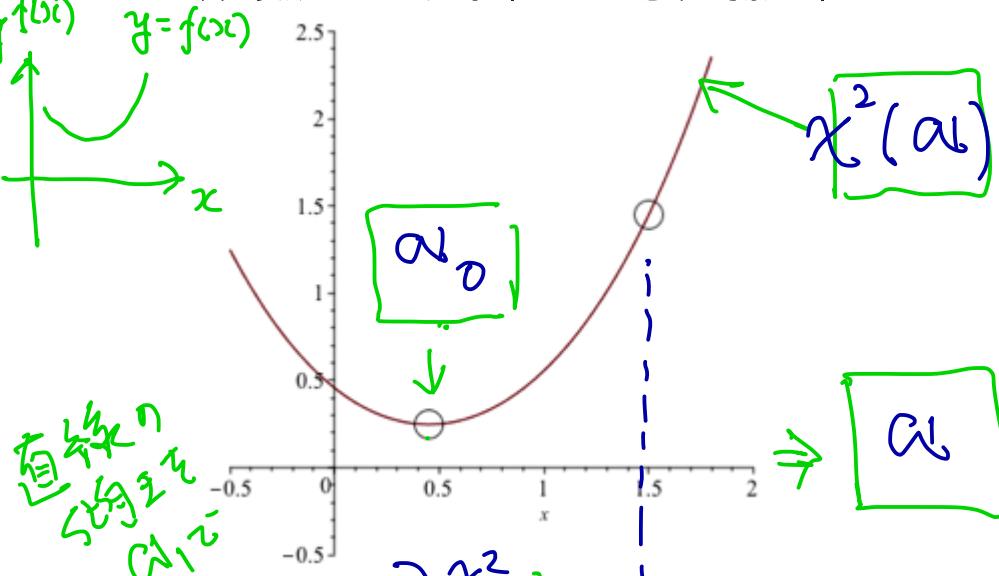
$$\chi^2(\mathbf{a}) = \chi^2(a_0) - d(a - a_0) + \frac{1}{2}D(a - a_0)^2 \approx f$$

である。最小の点 a_0 は、微分が0になるので、

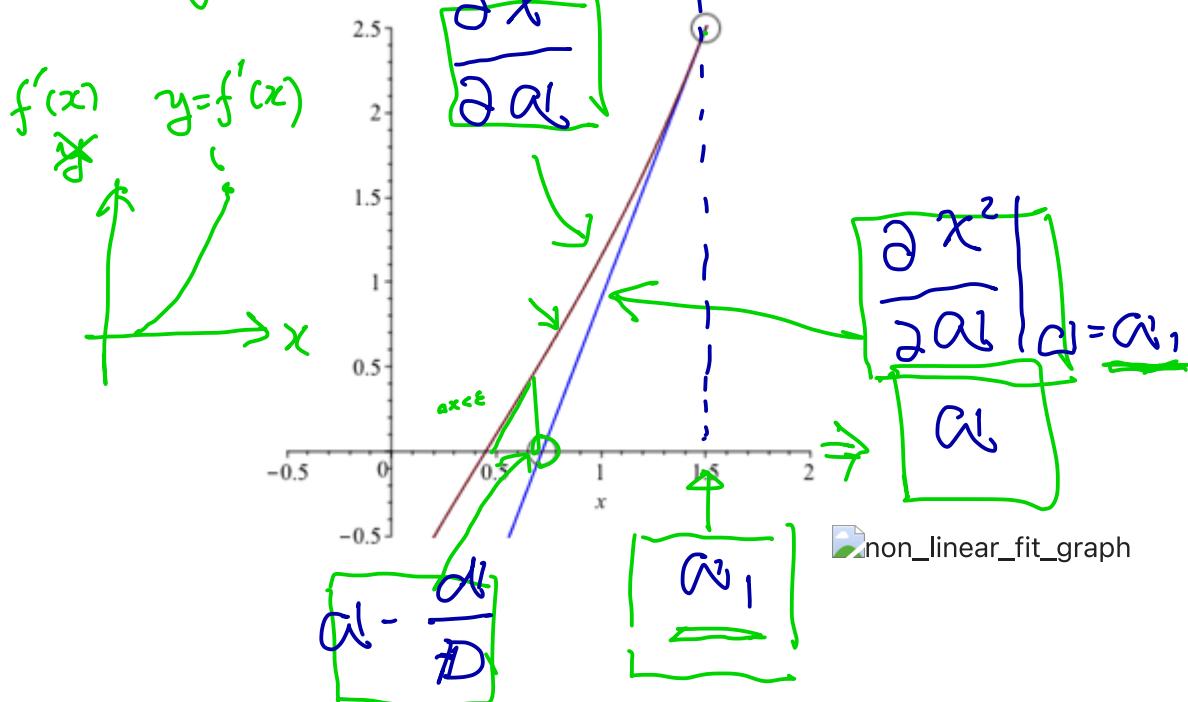
$$a_0 = a + D^{-1} \times (-d) \quad \frac{\partial f}{\partial a} = -d + D(a - a_0)$$

と予測される。

図を参照して上の式を導け、またその意味を考察せよ。



$$\begin{aligned} \frac{\partial f}{\partial a} &= 0 \\ -d + D(a - a_0) &= 0 \\ \frac{d}{D} &= a - a_0 \\ a_0 &= a_1 - \frac{d}{D} \end{aligned}$$



$$\begin{aligned} \frac{\partial \chi^2}{\partial a_1} &= a_1 - a_0 \\ a_1 &= a_0 + \frac{d}{D} \end{aligned}$$

non_linear_fit_graph

現実には高次項の影響で計算通りにはいかず、単に最小値の近似値を求めるだけである。これは、 $\chi(\mathbf{a})^2$ の微分関数の解を Newton 法で求める操作に対応する。つまり、この操作を何度も繰り返せばいずれ解がある精度で求まるはず。

python code

幾つもの関数が用意されている。

- curve_fit
- curve_fit with bounds
- least square fit

全部を理解する必要はないが、manualを見ながら使うことができるといいね。boundsとかparamsの初期値が重要。

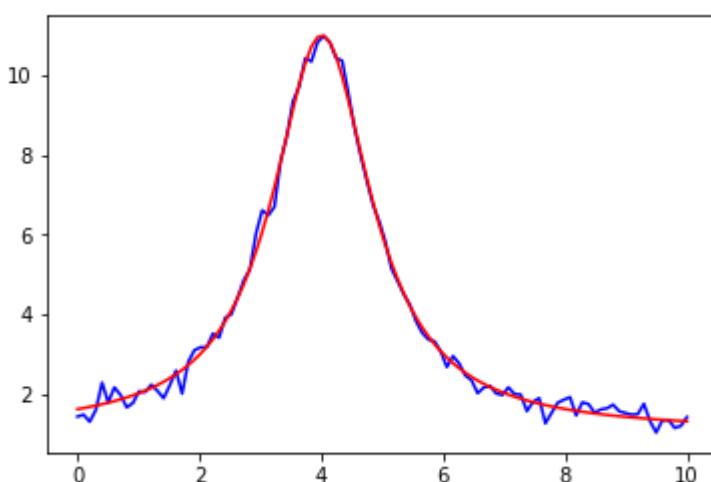
```
In [7]: import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def func(t, a1, a2, a3, a4):
    return a1+a2/(a3+(t-a4)**2)

xdata = np.linspace(0, 10, 100)
y = func(xdata, 1, 10, 1, 4)
y_noise = 0.2 * np.random.normal(size=xdata.size)
ydata = y + y_noise
plt.plot(xdata, ydata, 'b-', label='data')

popt, pcov = curve_fit(func, xdata, ydata)
print(popt)
plt.plot(xdata, func(xdata, *popt), 'r-', label='fit')
plt.show()
```

```
[ 1.04022586  9.67489309  0.96979944  4.00792049]
```



```
In [3]: import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def func(t, a1, a2, a3, a4, a5):
    return a1+a2*1000/(a3+(t-a4)**2)+a2*1000/(a3+(t-a5)**2)

xdata = np.linspace(0, 256, 256)
```

```

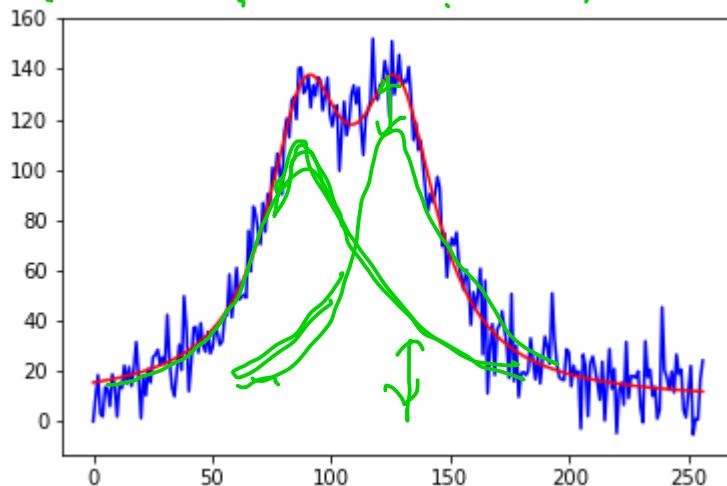
y = func(xdata, 10, 40, 380, 90, 128)
y_noise = 10 * np.random.normal(size=xdata.size)
ydata = y + y_noise
plt.plot(xdata, ydata, 'b-', label='data')

popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [15,50,400,100,150]))
plt.plot(xdata, func(xdata, *popt), 'r-', label='fit')

print(popt)
plt.show()

```

[7.92623836 42.57537952 400. 89.17411616 127.86538647]



```

In [4]: import scipy.optimize
from numpy import *

params0=[15,50,400,100,150]

def fit_func(params,t,y):
    a1,a2,a3,a4,a5=params
    residual=y-(a1+a2*t**1000/(a3+(t-a4)**2)+a2*t**1000/(a3+(t-a5)**2))
    return residual

params, cov=scipy.optimize.leastsq(fit_func,params0,args=(xdata, ydata))
print(params)

```

[7.16505033 44.91568512 427.72186145 89.189773 127.90437592]

In [8]: ?curve_fit

具体的な手順

パラメータの初期値を $a_1 = \underline{d}a$

$$\underline{a_0 + \Delta a} = \{a_0 + \Delta a, b_0 + \Delta b, c_0 + \Delta c, d_0 + \Delta d\}$$

とする。このとき関数 f を真値 a_0, b_0, c_0, d_0 のまわりでテイラー展開し、高次項を無視すると

$$\begin{aligned}\underline{\Delta f} &= f(a_0 + \Delta a_1, b_0 + \Delta b_1, c_0 + \Delta c_1, d_0 + \Delta d_1) - f(a_0, b_0, c_0, d_0) \\ &= \left(\frac{\partial}{\partial a} f \right)_0 \Delta a_1 + \left(\frac{\partial}{\partial b} f \right)_0 \Delta b_1 + \left(\frac{\partial}{\partial c} f \right)_0 \Delta c_1 + \left(\frac{\partial}{\partial d} f \right)_0 \Delta d_1\end{aligned}$$

となる。

課題でつくったデータは $t = 1$ から $t = 256$ までの時刻に対応したデータ点 f_1, f_2, \dots, f_{256} とする。各測定値とモデル関数から予想される値との差 $\Delta f_1, \Delta f_2, \dots, \Delta f_{256}$ は、

$$\Delta f_i = J \begin{pmatrix} \Delta a_1 \\ \Delta b_1 \\ \Delta c_1 \\ \Delta d_1 \end{pmatrix}$$

となる。ここで J はヤコビ行列と呼ばれる行列で、4列256行

$$J = \begin{pmatrix} \left(\frac{\partial}{\partial a} f\right)_1 & \left(\frac{\partial}{\partial b} f\right)_1 & \left(\frac{\partial}{\partial c} f\right)_1 & \left(\frac{\partial}{\partial d} f\right)_1 \\ \vdots & \vdots & \vdots & \vdots \\ \left(\frac{\partial}{\partial a} f\right)_{256} & \left(\frac{\partial}{\partial b} f\right)_{256} & \left(\frac{\partial}{\partial c} f\right)_{256} & \left(\frac{\partial}{\partial d} f\right)_{256} \end{pmatrix}$$

である。このような矩形行列の逆行列は転置行列 J^T を用いて、

$$J^{-1} = (J^T J)^{-1} J^T$$

と表わされる。したがって、真値からのずれは

$$\Delta a = J^{-1} J^T \Delta f$$

$$\begin{pmatrix} \Delta a_1 \\ \Delta b_1 \\ \Delta c_1 \\ \Delta d_1 \end{pmatrix} = (J^T J)^{-1} J^T \begin{pmatrix} \Delta f_1 \\ \Delta f_2 \\ \vdots \\ \Delta f_{256} \end{pmatrix}$$

で求められる。理想的には $(\Delta a_1, \Delta b_1, \Delta c_1, \Delta d_1)$ は $(\Delta a, \Delta b, \Delta c, \Delta d)$ に一致するはずだが、測定誤差と高次項のために一致しない。初期値に比べ、より真値に近づくだけ。そこで、新たに得られたパラメータの組を新たな初期値に用いて、より良いパラメータに近付けていくという操作を繰り返す。新たに得られたパラメータと前のパラメータとの差がある誤差以下になったところで計算を打ち切り、フィッティングの終了となる。

pythonによる解法の指針

まずは、お任せで `curve_fit` を試しましょう。

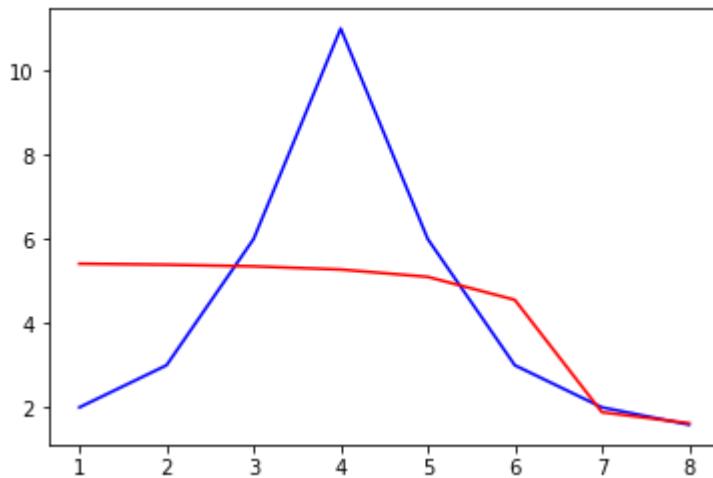
```
In [14]: import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def func(t, a1, a2, a3, a4):
    return a1+a2/(a3+(t-a4)**2)

ndata = 8
nparam = 4
xdata = np.linspace(1, 8, ndata)
y = func(xdata, 1, 10, 1, 4)
#y = func(xdata, 5, -2, 0, 7)
ydata = y
plt.plot(xdata, ydata, 'b-', label='data')

popt, pcov = curve_fit(func, xdata, ydata)
```

```
plt.plot(xdata, func(xdata, *popt), 'r-', label='fit')
plt.show()
```



In [10]: `print(popt)`

[5.4714478 -2.5260854 0.4296826 7.52394083]

うまくいってません。curve_fitの失敗の原因は、ほとんどが初期値の取り方のせいです。

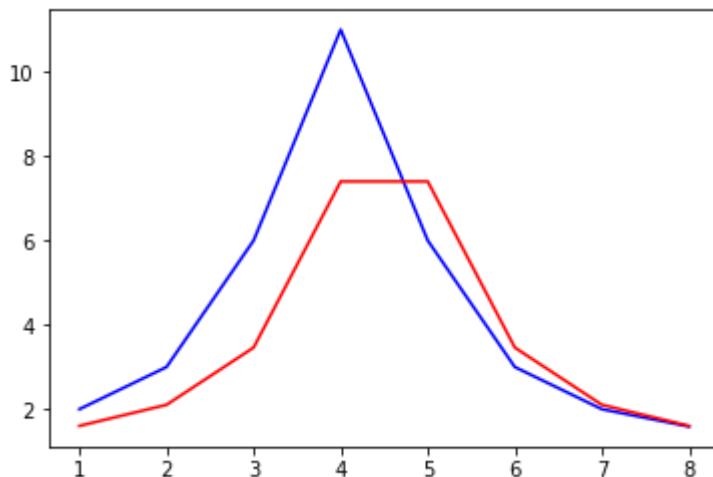
では、手計算でどうなるかを観て行きましょう。まずは初期値として適当な値を取ります。さらに、numpyと線形代数計算のためにscipy.linalg as linalgを呼びだしておきます。サンプルデータydataと初期値で予測される関数を同時にplotして観ます。

Q1,

In [15]: `guess1 = [1, 8, 1, 4.5]`

In [16]:
`from pprint import pprint
import scipy.linalg as linalg
plt.plot(xdata, ydata, 'b-', label='data')

plt.plot(xdata, func(xdata, *guess1), 'r-', label='fit')
plt.show()`



ydataと予測した関数との差をdfに入れます。

見やすいように、小数点以下を3桁表示に制限しています。

In [18]: `np.set_printoptions(precision=3, suppress=True)`

→ `df=np.zeros([ndata])`
`for i in range(0,ndata):`

Δf_i

```
df[i] = ydata[i]-func(xdata[i], *guess1)
```

```
pprint(df)
```

```
array([ 0.396, 0.897, 2.538, 3.6 , -1.4 , -0.462, -0.103, -0.016])
```

ローレンツ型の関数を仮定し、関数として定義。

```
def func(t, a1, a2, a3, a4):  
    return a1+a2/(a3+(t-a4)**2)
```

$$f = a_1 + \frac{a_2}{a_3 + (x - a_4)^2}$$

ヤコビアンの中の微分を新たな関数として定義します。

Δf

- $dfda1 := x \mapsto 1$
- $dfda2 := x \mapsto \frac{a_2}{a_3 + (x - a_4)^2}^{-1}$
- $dfda3 := x \mapsto -\frac{a_2}{(a_3 + (x - a_4)^2)^2}$
- $dfda4 := x \mapsto -\frac{a_2(-2x + 2a_4)}{(a_3 + (x - a_4)^2)^2}$

$$\frac{\partial f}{\partial a_2} = \frac{1}{a_3 + (x - a_4)^2}$$

$$\frac{\partial f}{\partial a_3} =$$

```
In [19]: def dfda1(x, a1, a2, a3, a4):  
    return 1  
def dfda2(x, a1, a2, a3, a4):  
    return (a3+(x-a4)**2)**(-1)  
def dfda3(x, a1, a2, a3, a4):  
    return -a2/(a3+(x-a4)**2)**2  
def dfda4(x, a1, a2, a3, a4):  
    return -a2*(-2*x+2*a4)/(a3+(x-a4)**2)**2
```

Jacobian行列を作ります。

```
In [20]: Jac=np.zeros([ndata,nparam])  
for i in range(0,ndata):  
    Jac[i,0] = dfda1(xdata[i], *guess1)  
    Jac[i,1] = dfda2(xdata[i], *guess1)  
    Jac[i,2] = dfda3(xdata[i], *guess1)  
    Jac[i,3] = dfda4(xdata[i], *guess1)  
pprint(Jac)
```

$J =$

```
array([[ 1. ,  0.075, -0.046, -0.319],  
       [ 1. ,  0.138, -0.152, -0.761],  
       [ 1. ,  0.308, -0.757, -2.272],  
       [ 1. ,  0.8 , -5.12 , -5.12 ],  
       [ 1. ,  0.8 , -5.12 ,  5.12 ],  
       [ 1. ,  0.308, -0.757,  2.272],  
       [ 1. ,  0.138, -0.152,  0.761],  
       [ 1. ,  0.075, -0.046,  0.319]])
```

$$\underline{J^{-1}} = (J^T J)^{-1}$$

を求めます。

```
In [21]: iJac = linalg.inv(np.dot(np.transpose(Jac),Jac))  
print(iJac)
```

```
[[ 1.017 -6.476 -0.821  0. ]  
 [-6.476  50.763  6.775 -0. ]
```

4

```
[ -0.821  6.775  0.933 -0. ]
[ 0.   -0.   -0.   0.016]]
```

```
In [22]: Jdf = np.dot(np.transpose(Jac), df)
pprint(Jdf)
```

$$\Delta \alpha_1 = \underbrace{i_j \cdot j^T}_{Jdf} \underbrace{\Delta f}_{\Delta df}$$

```
array([ 5.451, 2.537, -12.975, -33.309])
```

```
In [24]: np.dot(iJac, Jdf)
```

$$\Delta \alpha_1$$

```
Out[24]: array([-0.235, 5.592, 0.613, -0.52])
```

これをまたもとの近似値(guess)に入れ直して表示させると以下のようになる。カーブがデータに近づいているのが確認できるでしょう。

$$\alpha_2 = \alpha_1 + \Delta \alpha_1$$

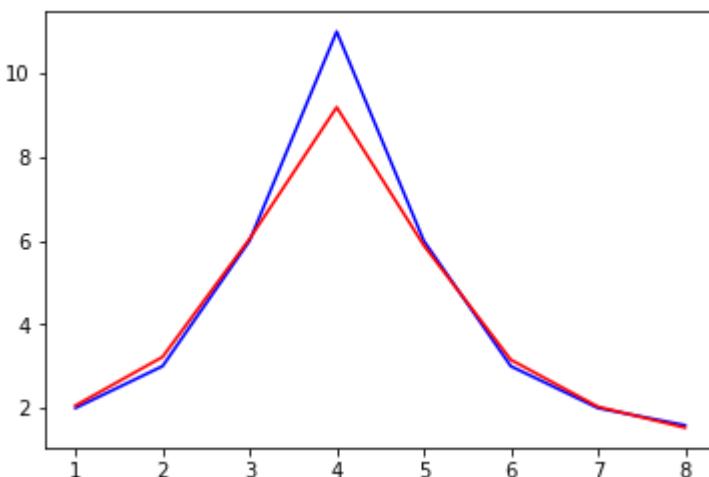
```
In [25]: guess1 = guess1 + np.dot(iJac, Jdf)
pprint(guess1)
```

$$\underbrace{1}_{\alpha_1} \underbrace{8}_{\Delta \alpha_1} \underbrace{1}_{+} \underbrace{4.5}_{\alpha_2}$$

```
array([ 0.765, 13.592, 1.613, 3.98])
```

```
In [26]: plt.plot(xdata, ydata, 'b-', label='data')
```

```
popt, pcov = curve_fit(func, xdata, ydata)
plt.plot(xdata, func(xdata, *guess1), 'r-', label='fit')
plt.show()
```



この操作をずれが十分小さくなるまで繰り返します。

```
df=np.zeros([ndata])
for i in range(0,ndata):
    dy = ydata[i]-func(xdata[i], *guess1)
    df[i]=dy
#pprint(df)
Jac=np.zeros([ndata,np.param])
for i in range(0,ndata):
    Jac[i,0] = dfda1(xdata[i], *guess1)
    Jac[i,1] = dfda2(xdata[i], *guess1)
    Jac[i,2] = dfda3(xdata[i], *guess1)
    Jac[i,3] = dfda4(xdata[i], *guess1)
# pprint(Jac)
iJac = linalg.inv(np.dot(np.transpose(Jac),Jac))
# print(iJac)
Jdf = np.dot(np.transpose(Jac),df)
# pprint(Jdf)
guess1 = guess1 + np.dot(iJac, Jdf)
pprint(guess1)
plt.plot(xdata, ydata, 'b-', label='data')
```

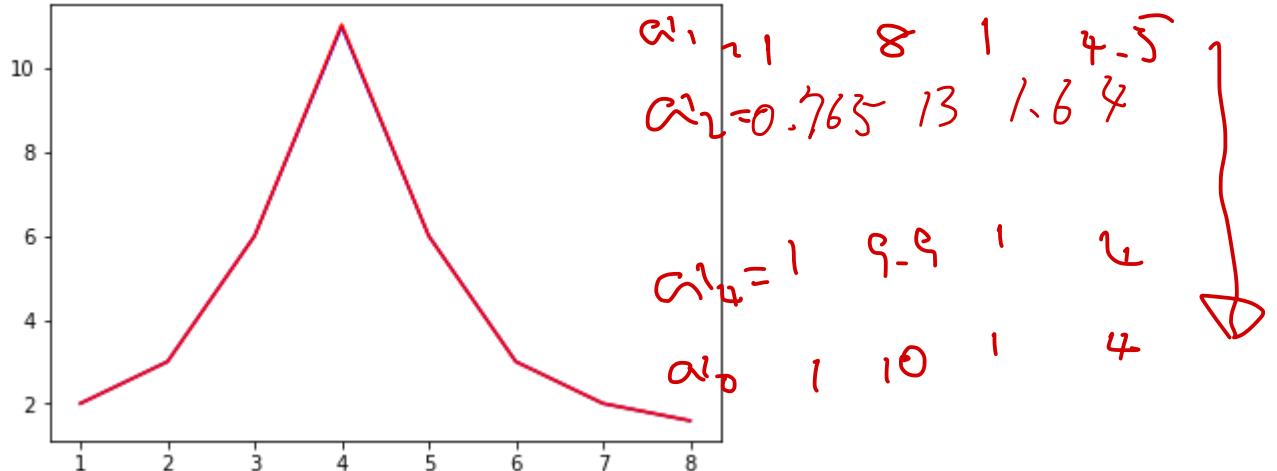
手で
計算
する。

```

popt, pcov = curve_fit(func, xdata, ydata)
plt.plot(xdata, func(xdata, *guess1), 'r-', label='fit')
plt.show()

```

```
array([ 1.006, 9.926, 0.989, 4. ])
```



4回ほど繰り返すと以上の通り、いい値に収束します。

Gauss-Newton法に関するメモ

このGauss-Newton法と呼ばれる非線形最小二乗法は線形問題から拡張した方法として論理的に簡明であり、広く使われている。しかし、収束性は高くなく、むしろ発散しやすいので注意が必要。2次の項を無視するのではなく、うまく見積もる方法を用いたのがLevenberg-Marquardt法である。明快な解説がNumerical Recipes in C(C言語による数値計算のレシピ) WilliamH.Press 他著、技術評論社1993にある。

課題

Gaussian(正規分布)へのフィット

正規分布で知られる、ガウス関数

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

でフィットをやってみましょう。

例えば、平均値(μ)が60点、偏差値(σ)が15点、ピークの人数が20人としましょう。

```

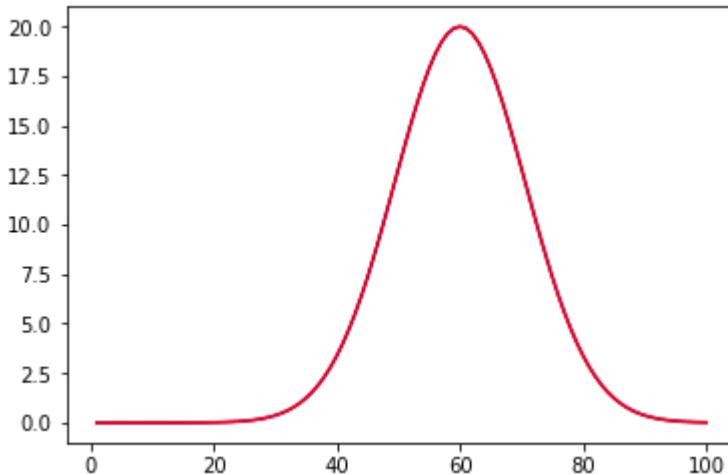
In [28]: import numpy as np
         import matplotlib.pyplot as plt

         def func(x, a1, a2, a3):
             return a1*np.exp(-(x-a2)**2/a3**2)

         ndata = 100
         xdata = np.linspace(1, ndata, ndata)
         y = func(xdata, 20, 60, 15)
         ydata = y
         plt.plot(xdata, ydata, 'b-', label='data')

```

```
popt, pcov = curve_fit(func, xdata, ydata)
plt.plot(xdata, func(xdata, *popt), 'r-', label='fit')
plt.show()
```



In [29]: `print(popt)`

[20. 60. 15.]

`guess1 = [10, 50, 10]`

から初めてGauss-Newton法でfittingしなさい。

ただし、Gauss関数

$$f(x) = a_1 \exp\left(-1/2 \frac{(x - a_2)^2}{a_3^2}\right)$$

それぞれのパラメータでの微分は、

$$\frac{\partial f}{\partial a_1} = \exp\left(-\frac{(x - a_2)^2}{2a_3^2}\right)$$

$$\frac{\partial f}{\partial a_2} = \frac{a_1 (x - a_2)}{a_3^2} \exp\left(-\frac{(x - a_2)^2}{2a_3^2}\right)$$

$$\frac{\partial f}{\partial a_3} = \frac{a_1 (x - a_2)^2}{a_3^3} \exp\left(-\frac{(x - a_2)^2}{2a_3^2}\right)$$

これらの関数は次の通り定義される。

In [30]: `from pprint import pprint
import scipy.linalg as linalg`

```
def dfda1(x,a1,a2,a3):
    return np.exp(-(x - a2)**2 / a3**2 / 2)
def dfda2(x,a1,a2,a3):
    return a1 * (x - a2) / a3**2 * np.exp(-(x - a2)**2 / a3**2 / 2)
def dfda3(x,a1,a2,a3):
    return a1 * (x - a2)**2 / a3**3 * np.exp(-(x - a2)**2 / a3**2 / 2)
```

以下の初期条件からfittingをおこなえ.

In [31]: nparam = 3
guess1 = [10, 50, 10]

In []: