分子生物第３講座/構造生物学研究センター　　2009 年 2 年生コンピュータ実習
文責　成田哲博

# 第十回 データフィッティング

## 今回の目的

実験データに任意の関数をフィッティングできる。

## 1　scipy の設定

データのフィッティングには scipy というモジュールを用いる。scipy は自然科学用のライブラリパッケージであり、多くの便利な機能を持っている。残念ながら実習用コンピュータにはインストールされていないので、講義資料のホームページから二年生コンピュータ実習をクリック、scipy 実習用を選び、scipy.tar.gz をダウンロードし、環境変数 PYTHONPATH に書かれているディレクトリにコピーする。本実習では、第７回に ~/pythonlib を PYTHONPATH に設定したはずである。従って、~/pythonlib にコピーする。
コマンドラインから、
% tar xvzf scipy.tar.gz
と入力すると、pythonlib の下に scipy が展開され、使えるようになる。python の対話モードで、
>>> import scipy
と入力して、エラーがでなければ正常にインストールされている。

## 2 フィッティングの基礎

x に関する任意の関数を実験値に対してフィッティングすることを考える。フィッティングするべきパラメータ列を$\vec{p} = (p_0, p_1, p_2, \cdots, p_n)$ とし、フィッティング関数を $f(x, \vec{p})$ とする。たとえば、フィッティング関数が一次関数なら
$\vec{p}=( p_0, p_1)$
$f(x, \vec{p})=p_1x + p_0$
である。前回の expsim.py で作ったような、指数関数でベースラインがついている場合では、
$\vec{p}=( p_0, p_1, p_2)$
$f(x, \vec{p})=p_1\exp(-p_2x) + p_0$
となる。

測定値($x_i$, $y_i$)が i 番目の測定値として、これらの値に f をフィッティングするためには、各点のフィッティング残差 (測定値と、フィッティング関数からの予測値の差)

$r_i = y_i - f(x_i, \vec{p})$

の二乗和$\sum_i r_i^2$ を最小化するパラメータ列$\vec{p}$を探せば良い(最小二乗法によるフィッティング)。もし、各点の測定誤差(エラー)がわかっていて、i 番目の測定点の誤差が $e_i$ だった場合には、フィッティング各点のには、$r_i$ をエラー補正付きフィッティング残差 $r_{i,err}$

$r_{i,err} = (y_i - f(x_i, \vec{p})) / e_i$

に変更して、同様に$\sum_i r_{i,err}^2$ を最小化するパラメータ列$\vec{p}$を探せば良い。フィッティング残差を $e_i$ でわることによって、二乗和$\sum_i r_{i,err}^2$において誤差が小さい点の比重を上げているだけである。

## 3　フィッティングプログラムの例

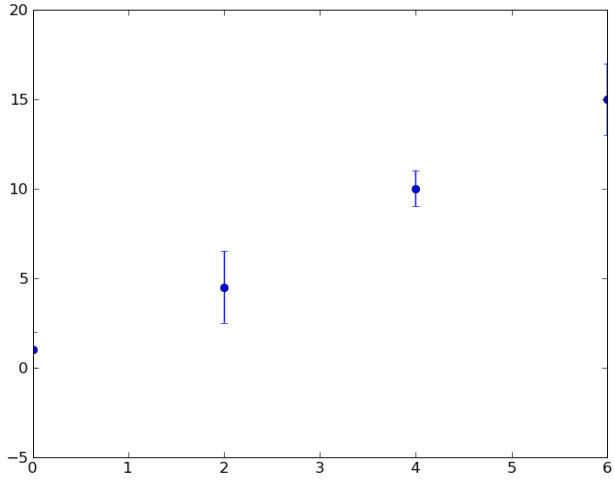
このようなフィッティングを行うプログラムを一から書くのは大変だが、幸いなことに python の scipy のサブモジュール scipy.optimize の中に、強力なフィッティング関数 leastsq が用意されている。

まず例を見てみよう。以下のようなデータ linear.dat に対して線形フィッティングをしてみる。

linear.dat:

```
0 1 1
2 4.5 2
4 10 1
6 15 2
```

第一列が x 座標、第二列が y 座標、第三列がエラーとする。これを第六回の課題で作成した plot3.py で表示すると右のようになる。これに線形フィットをするプログラムは下に示す linearfit.py である。




linearfit.py:

```
#!/usr/bin/env python
```

```
import sys,math,pylab   #モジュールのインポート
```

```
import scipy.optimize  #scipy.optimizeのインポート。これでleastsqが使える。

def modelfunc (x, p):  #フィッティングする関数の定義
    func = p[1]*x + p[0]  # リストpがパラメータ列。p[1]が直線の傾き、p[0]が切片
    return (func)

def residue (p, y, x, err):  #フィッティング残差の計算
    res = ((y - modelfunc(x, p))/err)  # r_i,errの定義と同じ。
    return (res)

p0=[0.0,0.0]  #p0はパラメータ列の初期値
infile=sys.argv[1]  #データファイル名をコマンドライン引数から入力
p0[0]=float(sys.argv[2])  #切片の初期値をコマンドライン引数から入力
p0[1]=float(sys.argv[3])  #傾きの初期値をコマンドライン引数から入力

x=[]  #測定値x_iを格納するリスト
ymeas=[]  #測定値y_iを格納するリスト
yerr=[] #測定点のエラーe_iを格納するリスト

fin=open(infile,'r')    #データファイルから各測定点データを読み込む
for line in fin:
        linedata=line.split()
        x.append(float(linedata[0]))
        ymeas.append(float(linedata[1]))
        yerr.append(float(linedata[2]))

xarray=pylab.array(x)  #各測定点データのリストをarray型に変換
ymarray=pylab.array(ymeas) #leastsqは、データのリストはarray型しか受け付けない。
yerrarray=pylab.array(yerr)

param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray,yerrarray),
full_output=True)  # フィッティングの実行。param_outputに結果を格納。
print param_output[0]  #param_output[0]は、フィッティングパラメータのリスト。この場合、
param_output[0][0]がフィットされた切片、param_output[0][1]が直線の傾き。
print param_output[1] #param_output[1]は誤差行列。その対角成分の平方根が、それぞれの
パラメータのフィッティング誤差。
```

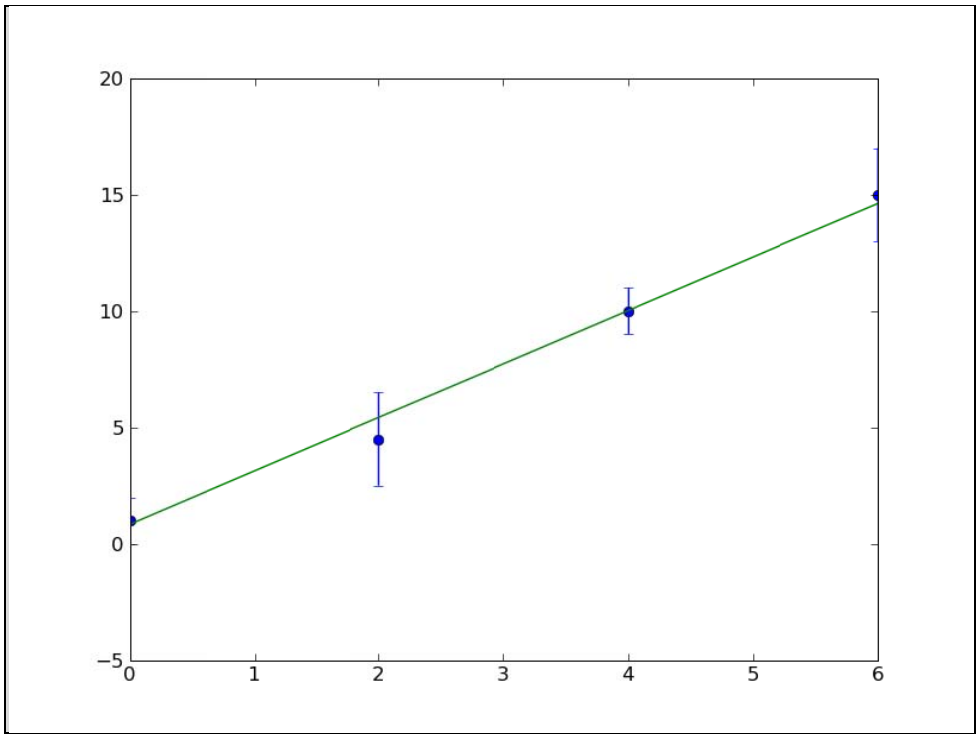
```
y=modelfunc(xarray,param_output[0])
```
#ここから先は、各データのプロットと、そのフィッティング直線のプロット。

```
pylab.errorbar(xarray,ymarray,yerrarray,fmt='o')
pylab.plot(xarray,y)
pylab.show()
```

このプログラムは第一引数がデータファイル、第二引数が直線の傾きの初期値、第三引数が切片の初期値である。フィットしたパラメータとグラフを出力する。たとえば、先ほどのlinear.datに対してフィッティングした例は以下のようになる。

% ./linearfit.py linear.dat 0 0 #初期値$p_0$=0, $p_1$=0でフィッティングする。

```
./linearfit.py linear.dat 0 0
[ 0.8362069    2.29741379]
[[ 0.89655173 -0.20689656]
 [-0.20689656  0.0862069 ]]
```




緑色の直線がフィッティングした直線で、ターミナル上に出力された

```
[ 2.29741379  0.8362069 ]
[[ 0.0862069  -0.20689656]
 [-0.20689656  0.89655173]]
```

がフィッティングパラメータとエラーを表す。では、このプログラムを詳しく見てみよう。

## 4 scipy.optimize.leastsq

このプログラムは長く見えるが、本質的には、scipy.optimize.leastsq を使うための、関数、データの準備と、その結果の出力が大半を占めており、scipy.optimize.leastsq の一行だけが本質である。書式は、

scipy.optimize.leastsq(関数名, 初期値列, args=(dataarray0, dataarray1, dataarray2‥), full_output=True)

となる。最後の full_output=True は、エラー行列を出力するという意味で、常に指定しておいた方が良い。

たとえば、ここで用いる関数名を f とし、

def f(p,x,y,z)

‥

と定義し、p がパラメータ列だったとする。leastsq に用いる関数では、パラメータ列は常に最初の引数でなければならない。またデータは x,y,z という三つの array(第三回を参照)に格納されているものとし、それぞれの i 番目の要素を $x_i,y_i,z_i$ とする。x, y, z の array の長さは同じでなくてはならない。p0 には初期パラメータ列を格納しているとする。このとき、

param_output=scipy.optimize.leastsq(f, p0, args=(x,y,z),full_output=True)

とすると、この関数は、

$$\sum_i (f(p, x_i, y_i, z_i))^2$$

を最小化するようなパラメータ列 p を初期パラメータ列 p0 のまわりで見つけてくれる。その結果は、フィッティングパラメータ列がリスト param_output[0], エラー行列が二次元リスト param_output[1]に出力される。この場合はデータが x,y,z の三次元だが、二次元でも n 次元でも同じように最小化ができる。注意点としては、以下の三つがある。

1　フィッティング対象のデータ(上の例では x,y,z)は、常に array 型しか許されない。

2　フィッティング対象のデータ(上の例では、x,y, z)は、leastsq の中では、関数(上の例では f)に対して array 型のままで渡される。従って、f の中で数学関数を使う場合は、array 型に対応した関数(モジュール math ではなく、モジュール pylab の中の関数、たとえば sin なら math.sin ではなく、pylab.sin)を使う必要がある。

3　初期パラメータ列があまり真の値からずれていると、正しいパラメータは見つからない。

## 5 scipy.optimize.leastsq を使った関数フィッティング

関数フィッティングの場合は、scipy.optimize.leastsq にわたす関数を、3 で述べた $r_{i,err}$

にすれば良い。linearfit.py においては、 関数 residue(p, y, x, err)がそれにあたる。p がフィッティングパラメータ(直線の傾きと切片) であり、y, x, err がデータ array であり、三次元のデータフィッティングになる。以下で先ほどの linearfit.py をもう一度見てみよう。

まず、residue(p,y,x,err)の定義

```
def modelfunc (x, p):  #フィッティングする関数の定義
    func = p[1]*x + p[0]  # リストpがパラメータ列。p[1]が直線の傾き、p[0]が切片
    return (func)

def residue (p, y, x, err):  #フィッティング残差の計算。これを最小化関数として、
                                scipy.optimize.leastsqにわたす。
    res = ((y - modelfunc(x, p))/err)  # r_i,errの定義と同じ。
    return (res)
```

次に初期パラメータ p0 を決定し、

```
p0=[0.0,0.0]  #p0はパラメータ列の初期値
infile=sys.argv[1]  #データファイル名をコマンドライン引数から入力
p0[0]=float(sys.argv[2])  #切片の初期値をコマンドライン引数から入力
p0[1]=float(sys.argv[3])  #傾きの初期値をコマンドライン引数から入力
```

x, y, err の三つのデータ array をつくる。

```
fin=open(infile,'r')    #データファイルから各測定点データを読み込む
for line in fin:
        linedata=line.split()
        x.append(float(linedata[0]))
        ymeas.append(float(linedata[1]))
        yerr.append(float(linedata[2]))

xarray=pylab.array(x)  #各測定点データのリストをarray型に変換
ymarray=pylab.array(ymeas)
yerrarray=pylab.array(yerr)
```

あとは、フィッティングをして、

```
param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray,yerrarray),
full_output=True)
```

以下でその結果を出力しているだけである。

```
 print param_output[0]
print param_output[1]
y=modelfunc(xarray,param_output[0])
pylab.errorbar(xarray,ymarray,yerrarray,fmt='o')
pylab.plot(xarray,y)
pylab.show()
```

ここで、

```
y=modelfunc(xarray,param_output[0])
```

において、xarray が array 型である。moodelfunc の中で、

```
func = p[1]*x + p[0]
```

の計算が行われ、func が出力されるが、x が array 型であるので、この計算の結果も array 型になる。第三回の内容を思い出して欲しい。Array 型に対して定数を掛けたり足したりすると、array の全ての要素に対してその計算を行い、結果は、それぞれの要素に対する結果を格納した array になる。後述するが、param_output[0]には、フィッティングパラメータが格納されているので、フィッティングされたパラメータに従って、xarray の各要素に対してフィッティング関数の計算が行われる。その結果を最後から二行目の pylab.plot(xarray,y)で描画することで、フィッティング直線を描画できるのである。

## 6 出力パラメータ

scipy.optimize.leastsq の出力 param_output は、二つの要素を持つリストで、param_output[0]は、フィッティングパラメータが格納されているリスト、param_output[1]は、フィッティングのエラーを示す誤差行列である。linear.py においては、モデル関数 modelfunc の定義は、

```
def modelfunc (x, p):
    func = p[1]*x + p[0]
    return (func)
```

であったから、パラメータ列の最初の要素(p[0])が直線の y 切片で、次の要素が(p[1])直線の傾きを示している。したがって、param_output[0]の最初の要素、param_output[0][0]がフィッティング直線の y 切片を表し、param_output[0][1]が傾きを表す。

linear.pyの実行時にターミナル上に表示された二つのリスト

```
[ 2.29741379  0.8362069 ]
[[ 0.0862069  -0.20689656]
 [-0.20689656  0.89655173]]
```

のうち、上が param_output[0]、下が param_output[1]であるが、param_output[0]を読む

と、フィッティングされた直線は、

y = 2.29741379 + 0.8362069x

であるとわかる。また、param_output[1]の行列の対角成分の平方根がフィッティングされたパラメータのエラーを示している。たとえば、y 切片のエラーは、param_output[1][0][0]である 0.0862069 の平方根で約 0.3 となる。一般に i 番目のパラメータのエラーは、param_output[1][i][i]の平方根で表される。これをわかりやすく表示するようにプログラムの出力部分に変更を加えたのが、以下の linearfit2.py である。出力部分の三行加えただけである。

linearfit2.py:

```
#!/usr/bin/env python

import sys,math,pylab
import scipy.optimize

def modelfunc (x, p):
    func = p[1]*x + p[0]
    return (func)

def residue (p, y, x, err):
    res = ((y - modelfunc(x, p))/err)
    return (res)

p0=[0.0,0.0]
infile=sys.argv[1]
p0[0]=float(sys.argv[2])
p0[1]=float(sys.argv[3])

x=[]
ymeas=[]
yerr=[]

fin=open(infile,'r')
for line in fin:
        linedata=line.split()
        x.append(float(linedata[0]))
```

```
        ymeas.append(float(linedata[1]))
        yerr.append(float(linedata[2]))

xarray=pylab.array(x)
ymarray=pylab.array(ymeas)
yerrarray=pylab.array(yerr)

param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray,yerrarray),
full_output=True)
print param_output[0]
print param_output[1]
print "y = Ax + B" #ここから下三行をlinearfit.pyに追加
print "A= ",param_output[0][1],"+-",math.sqrt(param_output[1][1][1])
print "B= ",param_output[0][0],"+-",math.sqrt(param_output[1][0][0])

y=modelfunc(xarray,param_output[0])
pylab.errorbar(xarray,ymarray,yerrarray,fmt='o')
pylab.plot(xarray,y)
pylab.show()
```

出力例:

```
% ./linearfit2.py linear.dat 0 0
 2.29741379  0.8362069 ]
[[ 0.0862069  -0.20689656]
 [-0.20689656  0.89655173]]
y = Ax + B
A=  2.29741379311 +- 0.293610112987
B=  0.836206896554 +- 0.946864158095
```

このようにすると、フィッティングの結果がわかりやすくなるだろう。

## 6 指数関数に対するフィッティング

linearfit2.py を改造して、前回作成した expsim.py で生成されるような指数関数のデータに対するフィッティングプログラム expfit.py を作ってみよう。指数関数のフィッティング

では、

$\vec{p}=(p_0, p_1, p_2)$

$f(x,\vec{p})=p_1\exp(-p_2x)+p_0$

のようになり、フィッティングするパラメータは三つになる。また、フィッティング対象のデータには誤差は設定されていない。このような場合は、全ての点の誤差を一定値にしておけばよい。他の変更場所は、modelfunc の中と、初期値の設定、結果の出力の部分だけである。

### modelfunc の変更

linearfit2.py

```
def modelfunc (x, p):
    func = p[1]*x + p[0]
    return (func)
```

を、expfit.py では、指数関数 $f(x,\vec{p})=p_1\exp(-p_2x)+p_0$ を表すように、

```
def modelfunc (x, p):
    func = p[1]+p[2]*pylab.exp(-1*p[0]*x)
    return (func)
```

に変更する。

### 初期値入力

次に初期値の入力を変更する。初期値はリストp0に入力する。今回の指数関数フィッティングの場合パラメータが三つになる（線形フィッティングでは二つだった)。入力する値が多くなると、コマンドライン引数からの入力はわかりにくくなる。そのため、raw_input を使って入力しよう。

linearfit2.py

```
p0=[0.0,0.0]
infile=sys.argv[1]
p0[0]=float(sys.argv[2])
p0[1]=float(sys.argv[3])
```

を、expfit.pyでは、

```
p0=[0,0,0]
print "f(t)=B+Cexp(-At)+noise(t)¥n"
p0[0]=float(raw_input('initA? '))
p0[1]=float(raw_input('initB? '))
p0[2]=float(raw_input('initC? '))
infile=raw_input('data file? ')
```

に変更する。linearfit2.pyのときのようにsys.argvを使っても良いが、このように

raw_inputを使って説明分を入れると、よりユーザーフレンドリーになる。やっていることは、初期値リストp0[0], p0[1], p0[2]にデータを入れているだけで、linearfit2.pyと余り変わらない。

### エラーの扱い

エラーがデータに含まれていない場合は、単純にエラーデータとして１を入力すれば良い。linearfit2.py では、ファイルデータの二列目を代入していたが、

```
        yerr.append(float(linedata[2]))
```

expfit.pyでは、全て1.0を代入する。

```
        yerr.append(1.0)
```

### フィットしたパラメータの出力

パラメータの出力を変える。

linearfit2.py

```
print "y = Ax + B"
print "A= ",param_output[0][1],"+-",math.sqrt(param_output[1][1][1])
print "B= ",param_output[0][0],"+-",math.sqrt(param_output[1][0][0])
```

を、expfit.py では、

```
print "y = B+Cexp(-At)"
print "A= ",param_output[0][0],"+-",math.sqrt(param_output[1][0][0])
print "B= ",param_output[0][1],"+-",math.sqrt(param_output[1][1][1])
print "C= ",param_output[0][2],"+-",math.sqrt(param_output[1][2][2])
```

にすれば良い。これだけで、線形フィットのプログラムが指数関数フィットのプログラムに変わる。以上と同じ手順を踏めば、どんな関数でもデータにフィッティングできる。

### グラフのプロット

最後に、expfit.py では入力データにエラーがないので、pylab.errorbar ではなく、pylab.plot を用いる。linearfit2.py では、

```
pylab.errorbar(xarray,ymarray,yerrarray,fmt='o')
```

だったのを、expfit.py では、

```
pylab.plot(xarray,ymarray)
```

に変えればよい。

最後に expfit.py をまとめて書いておこう。関数を変えたときに変更が必要な行には、”#変更が必要”と書いた。

expfit.py

```
#!/usr/bin/env python

import sys,math,pylab
import scipy.optimize

def modelfunc (x, p):
    func = p[1]+p[2]*pylab.exp(-1*p[0]*x)      #変更が必要
    return (func)

def residue (p, y, x, err):
    res = ((y - modelfunc(x, p))/err)
    return (res)

p0=[0,0,0]    #変更が必要
print "f(t)=B+Cexp(-At)+noise(t)¥n"  #変更が必要
p0[0]=float(raw_input('initA? '))    #変更が必要
p0[1]=float(raw_input('initB? '))    #変更が必要
p0[2]=float(raw_input('initC? '))    #変更が必要
infile=raw_input('data file? ')    #変更が必要

x=[]
ymeas=[]
yerr=[]

fin=open(infile,'r')
for line in fin:
        linedata=line.split()
        x.append(float(linedata[0]))
        ymeas.append(float(linedata[1]))
        yerr.append(1.0)

xarray=pylab.array(x)
ymarray=pylab.array(ymeas)
yerrarray=pylab.array(yerr)
```

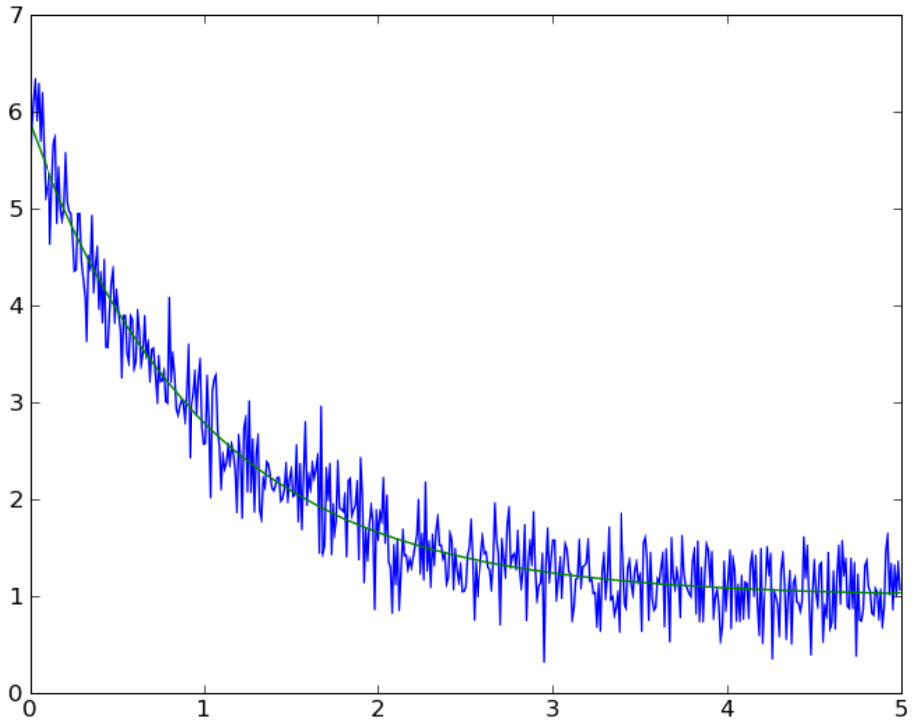
```
param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray,yerrarray),
full_output=True)
print param_output[0]
print param_output[1]
print "y = B+Cexp(-At)"    #変更が必要
print "A= ",param_output[0][0],"+-",math.sqrt(param_output[1][0][0])    #変更が必要
print "B= ",param_output[0][1],"+-",math.sqrt(param_output[1][1][1])    #変更が必要
print "C= ",param_output[0][2],"+-",math.sqrt(param_output[1][2][2])    #変更が必要

y=modelfunc(xarray,param_output[0])
pylab.plot(xarray,ymarray)
pylab.plot(xarray,y)
pylab.show()
```

関数が変わることによって変更が必要なところは、modelfuncの中の一行以外は、データの入出力との部分だけである。これを見れば、容易に他の関数のフィッティングのためにプログラムを変更することができることがわかるだろう。実行例を以下に示す。入力データexptest.datは、前回のexpsim.pyで作成したもの。緑がフィットされたカーブ、青が元データ。

```
% ./expfit.py
f(t)=B+Cexp(-At)+noise(t)

initA? 1
initB? 1
initC? 1
data file? exptest.dat
[ 0.99965573  0.99701046  4.89119424]
[[ 0.00772382  0.00581466  0.00720532]
 [ 0.00581466  0.007682   -0.0011079 ]
 [ 0.00720532 -0.0011079   0.03942911]]
y = B+Cexp(-At)
A=  0.99965572572 +- 0.0878852519936
B=  0.997010456747 +- 0.0876470096235
C=  4.89119424454 +- 0.198567638516
```




課題1 ：前回の課題２で作ったgausssim.pyの出力に対して傾いたベースライン付きガウシアンをフィッティングするプログラムを書け。まずガウシアン一つの場合をgaussfit1.pyとして書くこと。フィッティング関数は以下のようになる。

$$f(x) = p_0 + p_1 x + p_2 e^{-(\frac{x-p_3}{p_4})^2}$$

また、実際にgausssim.pyで、ベースラインに一つのガウシアンが乗った状態のデータを作り、その出力に対してフィッティングをし、その結果をレポートせよ。

課題2: ガウシアンが二つの場合について同様にプログラムgaussfit2.pyを書け。また、実際にgausssim.pyで、ベースラインに二つのガウシアンが乗った状態のデータを作り、その出力に対してフィッティングをし、その結果をレポートせよ。

ヒント:gaussfit2.py の実行例

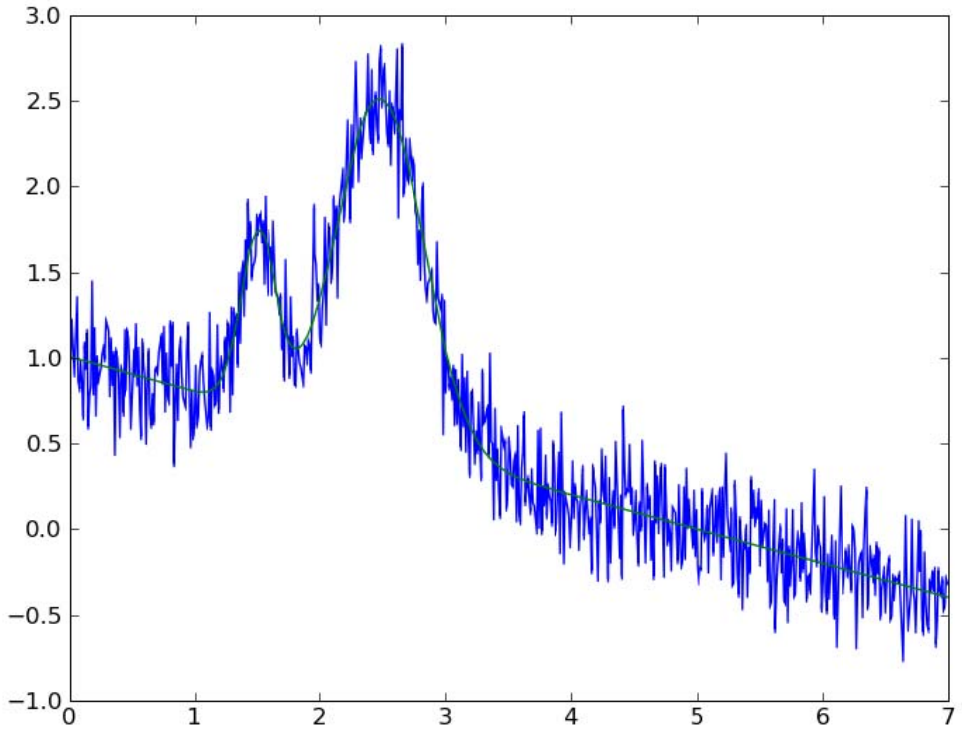
```
% ./gaussfit2.py

f(x)=A+Bx+Cexp(-(x-D)**2 / (E**2)) + Fexp(-(x-G)**2 / (H**2))+noise(t)

initA? 0
```

```
initB? 0
initC? 1
initD? 1.5
initE? 0.5
initF? 1
initG? 2.5
initH? 0.5
data file? gausstest2.dat
[ 1.00517209 -0.20058348  1.00164566  1.50773802  0.20423266  2.00094977
  2.4821588   0.48700508]
[[  9.45956858e-03  -1.80413990e-03  -4.62708988e-03   4.35747372e-04
   -1.38872701e-03  -4.37352090e-03   1.11085213e-04  -1.17382516e-03]
 [ -1.80413990e-03   4.42184683e-04   7.89752657e-04  -7.24580716e-05
    2.49242335e-04   6.51096087e-04  -3.98783961e-05   1.46770547e-04]
 [ -4.62708988e-03   7.89752657e-04   6.10662830e-02  -1.34707502e-04
   -7.13216790e-03   3.53083250e-03  -5.19016333e-05   1.63556540e-04]
 [  4.35747372e-04  -7.24580716e-05  -1.34707502e-04   1.76617715e-03
    1.32242500e-04   4.83437022e-04   1.84143733e-04  -5.30338663e-04]
 [ -1.38872701e-03   2.49242335e-04  -7.13216790e-03   1.32242500e-04
    3.77836492e-03   1.72847692e-03   3.00504950e-04  -5.62630213e-04]
 [ -4.37352090e-03   6.51096087e-04   3.53083250e-03   4.83437022e-04
    1.72847692e-03   2.77616282e-02   1.87664224e-04  -3.84181294e-03]
 [  1.11085213e-04  -3.98783961e-05  -5.19016333e-05   1.84143733e-04
    3.00504950e-04   1.87664224e-04   1.02780362e-03  -1.34977647e-04]
 [ -1.17382516e-03   1.46770547e-04   1.63556540e-04  -5.30338663e-04
   -5.62630213e-04  -3.84181294e-03  -1.34977647e-04   2.50440188e-03]]
f(x)=A+Bx+Cexp(-(x-D)**2 / (E**2)) + Fexp(-(x-G)**2 / (H**2))

A=  1.00517209389 +- 0.0972603135063
B=  -0.200583476544 +- 0.0210281878181
C=  1.00164565665 +- 0.247115930186
D=  1.50773801514 +- 0.0420259105144
E=  0.204232661317 +- 0.0614684058991
F=  2.00094976709 +- 0.166618210959
G=  2.48215880459 +- 0.0320593764067
H=  0.487005081249 +- 0.0500439994028
```




少しわかりにくいが、緑がフィットデータ。よく一致している。