第十一回　matplotlib, numpy, scipy

今回の目的

Python の有用なモジュール、matplotlib と numpy, scipy のさわりを理解する。
matplotlib は様々なグラフを描画するモジュール、numpy, scipy は様々な数値計算を行うモジュールである。

1　array 型

ここで、matplotlib, scipy の両方で使う便利な array 型をまず勉強しよう。array 型は、matplotlib と scipy の両方に共通して含まれる numpy モジュールの中で定義されている。array 型は、
array([3, 5, 6, 7])
のように、数値が入ったリストを array()で囲んだ形をしている。

このような array を作成するには、numpy.array(リスト)とする。以下に例を示す。
```
>>>
>>> import numpy   #numpy のインポート。すでにインポートしていれば不要。
>>> a=numpy.array([3,5,6,7])     #リスト[3,5,6,7]から array を作る。
>>> a
array([3, 5, 6, 7])  #確かに array ができている。
```

リストは変数に入っていても良い。
```
>>> b=[3,5,6,7]
>>> a=numpy.array(b)
```
でも前の例と同じことになる。

array 型はリスト型のための関数のほとんどを使えるが、リストのためのメソッドは使えない。たとえば、
```
>>> len(a)
4     #a の要素数
```
はできるが、a.append(2), a.revome(1)などはエラーになり使えない。そのかわり、非常に強力な数値演算ができる。array 同士の 4 則演算は、二つの array からインデックスごとに一つずつデータをとりだし、その二つのデータに対して演算した結果を array に格納したものが結果になる。たとえば、以下の例では、

```
>>> a=numpy.array([3,4,5,6])
>>> b=numpy.array([1,2,3,4])
>>> c=a*b
>>> c
array([ 3,  8, 15, 24])
```
c[0]は、a[0]*b[0]の結果であり、同様に、c[i]の結果は a[i]*b[i]の結果であることがわかる。ほかの演算でも同じである。

```
>>> a/b
array([3, 2, 1, 1])
>>> a+b
array([ 4,  6,  8, 10])
>>> a-b
array([2, 2, 2, 2])
```

また、array と数値の四則演算ができる。この場合、すべての要素と数値のあいだの計算が行われる。
```
>>> a-1
array([2, 3, 4, 5]) #すべての要素から 1 がひかれている。
>>> a*2
array([ 6,  8, 10, 12])
```

numpy モジュールのなかでほとんどの数学関数が array 用に再定義されている。これらの関数は array を与えると、入力 array の各要素に対しその関数による演算が行われ、その結果を格納した出力 array を返す。

例：
```
>>> a=numpy.array([3,5,6,7])
>>> numpy.exp(a)
array([  20.08553692,   54.59815003,  148.4131591 ,  403.42879349])
# e3, e5, e6, e7 が答えの array に格納されている。
>> numpy.log(a)
array([ 1.09861229,  1.38629436,  1.60943791,  1.79175947])
# log(3), log(5), log(6), log(7)が答えの array に格納されている。
```

また、前回使った range と同様に arange を用いて、数列をつくることができる。arange は range と違って実数の数列が作れる。書式は range のときと同じ。
例:
```
>>> numpy.arange(0,1.2,0.1)   #0 から 1.2 の手前までの増分 0.1 の数列を作る
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1])
>>> numpy.arange(0.5,-0.5,-0.2)    #0.5 から-0.5 の手前までの増分-0.2 の数列を作る。
array([ 0.5,  0.3,  0.1, -0.1, -0.3])
```

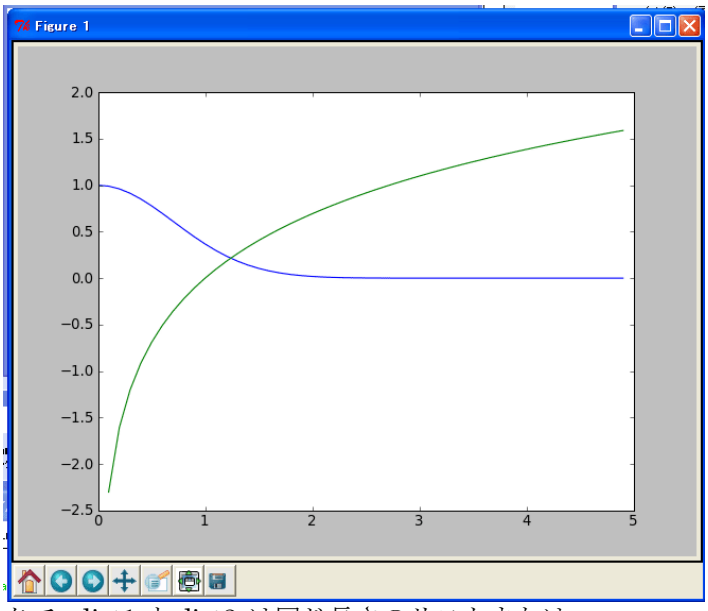
さらに、平均値、標準偏差などを非常に簡単に計算できる。
```
>>> numpy.average(b) :b (array([1,2,3,4])) の平均値
2.5
>>> numpy.std(b) : b (array([1,2,3,4])) の標準偏差
1.1180339887498949
```

2　Matplotlib によるグラフ描画

研究や勉強をしていると、データをプロットしたり、様々な関数のグラフを見てみたくなることがある。Python はそのために非常に強力なライブラリ matplotlib を持っている。matplotlib はモジュール pylab としてインポートするが、使用前に少しだけ設定が必要である。設定には、第三回課題 1 でホームディレクトリにコピーした、matplotlibrc を用いる。この matplotlibrc は、matplotlib がグラフを描画するための設定ファイルである。MacOSX の場合は、emacs を用いて、matplotlibrc を開き、検索を用いて以下の行を探す。
```
backend      : Agg
```
この行の Agg を、以下のように MacOSX に変えて、
```
backend      : MacOSX
```
保存すれば良い。Pylabにはnumpyが含まれているので、numpy.array, numpy.arangeなど、numpyの関数はpylab.array, pylab.arangeなど、pylabの関数としても呼び出せる。
これが済めば、まず python を起動し、以下のように入力してみよう。

```
>>> import pylab
```

```
>>> x=pylab.arange(0.1,5,0.1)
>>> y=pylab.exp(-x**2)
>>> y2=pylab.log(x)
>>> pylab.plot(x,y)
>>> pylab.plot(x,y2)
```

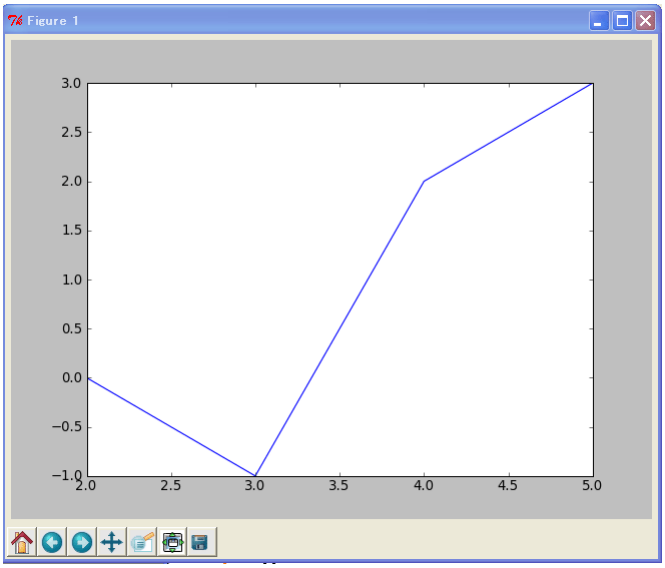


で右のような window が出現するはずである。
ただし、Windows か linux では、このあとに
>>> pylab.show()
を入力することが必要である。

pylab モジュールの plot の基本

pylab.plot(list1, list2)　プロットの作成。
pylab.show()　プロットの表示。
list1 は x 座標、list2 は y 座標のデータで、list1 と list2 は同じ長さのリストまたは array。
MacOSX の対話モードに限っては pylab.show()は不要。

例:




```
>>> x=[2,3,4,5]　#x 座標のリストを入力
>>> y=[0,-1,2,3]　#y 座標のリストを入力。x 座標のリストと要素数をそろえる。
>>> pylab.plot(x,y)
```

(2,0), (3,-1), (4,2), (5,3)の 4 点が直線で結ばれているのがわかる。

pylab.plot を実行するたびにグラフが重なっていく。
グラフをクリアするときには、
>>> pylab.clf()
とすればよい。

最初の例をもう一度

```
>>> import pylab　#モジュール pylab の import
>>> x=pylab.arange(0.1,5,0.1)　#0.1 から 5 まで 0.1 刻みの array を作成
>>> y=pylab.exp(-x**2)　#array の各要素ごとにに e^(-x^2) を計算して、結果を y に代入
>>> y2=pylab.log(x)　# array の各要素ごとにに log を計算して、結果を y2 に代入
>>> pylab.plot(x,y)　#x と y についてプロット
>>> pylab.plot(x,y2) #x と y2 についてプロット
>>> pylab.show()　#グラフを表示
```

グラフの表示設定

plot は、標準状態では点の間を直線で結ぶが、ほかにも様々な表示をすることができる。直線以外を使う場合、次のようにする。

pylab.plot(data1, data2, ‘character’)
data1, data2 は list または array

character は marker と style、色を同時指定できる。指定の順番は何でも良い。なにも指定しない場合は’-‘とおなじになる。

character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	pylabus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

- b : blue
- g : green
- r : red
- c : cyan

- m : magenta
- y : yellow
- k : black
- w : white

例。各自他の組み合わせも試して欲しい
```
>>> import pylab
>>> x=pylab.arange(-2,2,0.2)
>>> y=pylab.sin(x)
>>> pylab.plot(x,y,'-')    #pylab.plot(x,y)と同じ。クオーテーションの中はハイフン。
>>> pylab.clf()   #グラフクリア
>>> pylab.plot(x,y,'.')   #点で表示
>>> pylab.clf()
>>> pylab.plot(x,y,'yo')   #黄色円で表示
>>> pylab.clf()
>>> pylab.plot(x,y,'D—g') #緑の点線とダイアモンドで表示
```
表示範囲の変更をしたい場合は、
pylab.axis([xmin,xmax,ymin,ymax])
を用いる。たとえば、
```
>>> import pylab
>>> x=pylab.arange(-2,2,0.2)
>>> y=pylab.sin(x)
>>> pylab.plot(x,y)
>>> pylab.axis([0,0.5,0.2,1])
```
とすると、x が 0 から 0.5、 y が 0.2 から 1 の領域だけが表示される。
x 軸、y 軸だけ変更したければ
pylab.xlim(xmin,xmax)　#x 軸領域の設定
pylab.ylim(ymin,ymax)　#y 軸領域の設定

pylab.axis([0,0.5,0,2.1])
と
pylab.xlin([0,0.5])
pylab.ylim([0.2,1])
は同じ効果である。

課題１　関数のプロット１

$f(x)=e^{2\pi ix}$　を-1 から 1 の範囲で実部と虚部にわけてプロットせよ。
ヒント:
$e^{2\pi ix}$　は、x を arange で作成した array とすると、モジュール math をインポート(import math を実行)した上で、
pylab.exp(2*math.pi*1j*x)
で計算できる。また、array の実部と虚部はそれぞれ
array 名.real, array 名.imag で得られる。たとえば、
```
>>> a = pylab.array([1+2j, 3+4j, 5+6j])
>>> a.imag #a の虚部
array([ 2.,  4.,  6.])
>>> a.real #a の実部
array([ 1.,  3.,  5.])
```

プロットした結果(画像)とそれを得るのに必要な全ての情報をレポートせよ。

ファイルを読み込んでグラフを書く。

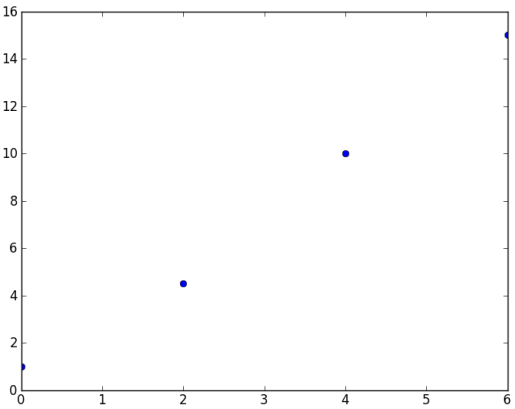
当たり前であるが、プログラムを書いて、ファイルを読み込んでグラフを書くこともできる。ここでは、ごく簡単なプロットプログラム plot.py を書いてみよう。

plot.py

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

import pylab,sys
infile=sys.argv[1]
fin=open(infile,'r')
x=[]
y=[]
for line in fin:
    linedata=line.split()
    x.append(float(linedata[0]))
    y.append(float(linedata[1]))
pylab.plot(x,y,'o')
pylab.show()
```

ここで、pylab.show()は、グラフを表示させるコマンドである。MacOSX の対話モードにおいては必要なかったが、プログラム内では、これを明示しないとグラフを表示しない。

たとえば、以下のようなデータ linear.dat を表示すると、右図のようになる。



linear.dat:

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

3　scipy を用いたフィッティング

Scipy, numpy には行列、ベクトル演算、最小化、フーリエ変換、信号処理など様々な機能があるが、その中から、関数フィッティングを紹介する。より詳しくは http://docs.scipy.org/doc/を参照のこと。

さて、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

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

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

である。

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)。

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。linear.dat に対して線形フィッティングをしてみる。
第一列が x 座標、第二列が y 座標である。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。

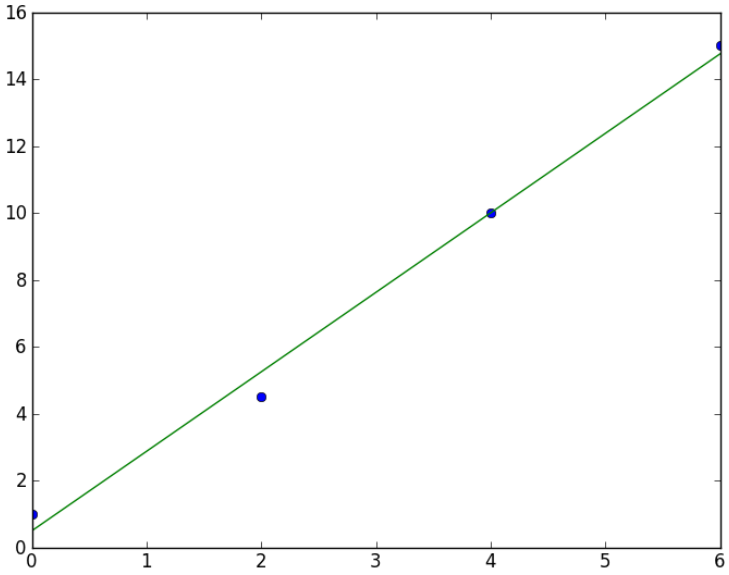
linearfit.py:

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

import sys,math,numpy,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):   #フィッティング残差の計算
    res = y - modelfunc(x, p)   # ri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xiを格納するリスト
ymeas=[]  #測定値yiを格納するリスト

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

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

param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray),
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.plot(xarray,ymarray,'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p0=0, p1=0でフィッティングする。
./linearfit.py linear.dat 0 0
```

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
[0.5　　2.375]
[[0.7　-0.15]
 [-0.15　0.05]]
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。



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 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、

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

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。linefit.py はデータが二次元であった。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。

4　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 にすれ

ば良い。linearfit.py においては、 関数 residue(p, y, x)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 がデータ array であり、二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]
    return (func)

def residue (p, y, x):    #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)
    return (res)
```

次に初期パラメータp0を決定し

```
p0=[0.0,0.0]
infile=sys.argv[1]
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
```

x, y の二つのデータ array をつくる。

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

xarray=numpy.array(x)
ymarray=numpy.array(ymeas)
```

あとは、フィッティングをして、

```
param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray),
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.plot(xarray,ymarray,'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。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。

5 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

```
[ 0.5     2.375]
[[ 0.7   -0.15]
 [-0.15   0.05]]
```

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 = 2.375x + 0.5

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 0.7 の平方根で約 0.8 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
[ 0.5    2.375]
[[ 0.7  -0.15]
 [-0.15  0.05]]
y = Ax + B
A=  2.375 +- 0.223606797751
B=  0.5 +- 0.836660026538
```

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、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。

ある物質が一定時間あたり A の確率で不可逆的に変化するとき、ある時間 t において残っている物質の量 f(t)は、以下の微分方程式に従う。

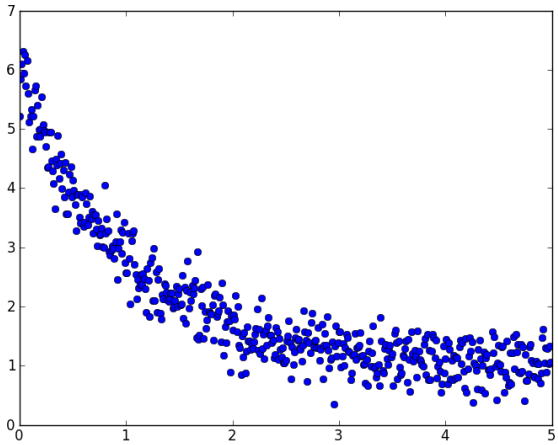
$$\frac{df}{dt} = -Af(t)$$

t=0 のときの物質の量を C とすると、この微分方程式の解は、

$$f(t) = Ce^{-At}$$

実際に計測されるデータは、多くの場合ベースラインの上に乗っており、またノイズも足される。そのため以下のようになる。

$$f(t) = B + Ce^{-At} + noise(t)$$



B は定数でベースラインを表し、第三項はノイズである。このようなデータの例として、exptest.dat をホームページ上においた。これを plot.py でプロットすると、右図のようになる。これに対してフィッティングをしてみよう。

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。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。

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。

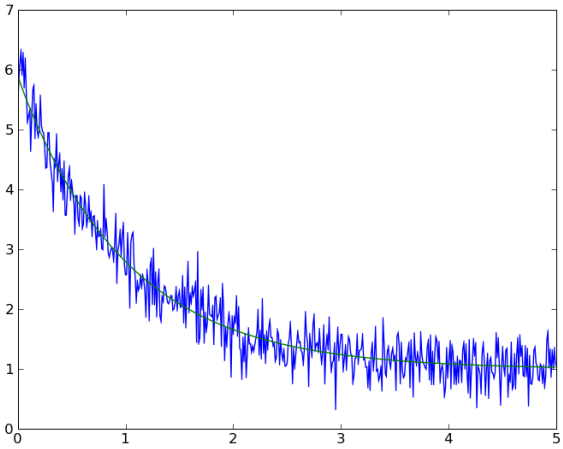
最後に expfit.py をまとめて書いておこう。関数を変えたときに変更が必要な行には、”#変更が必要”と書いた。
expfit.py

```
#!/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):
    res = y - modelfunc(x, p)
    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=[]

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

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


param_output = scipy.optimize.leastsq(residue, p0, args=(ymarray,xarray),
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。exptest.datに対する実行例を以下に示す。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 2:ガウシアンフィッティング

ホームページ上にある gausstest.dat（plot.pyの出力は右図）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gaussfit.pyとして書け。フィッティング関数は以下のようになる。また、実際に gausstest.datに対するフィッティングをレポートせよ。

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

$p_0 + p_1 x$ がベースラインの直線であり、

$p_2 e^{-(\frac{x-p_3}{p_4})^2}$ は中心p_3，幅p_4のガウシアンである。