

# Computational Materials Science (計算材料学特論)

## Lecture materials updated

[http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/D2MatE\\_programs.html?page=cms](http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/D2MatE_programs.html?page=cms)

### COMPUTATIONAL MATERIALS SCIENCE 2025 Q2

### 2025年度Q2 計算材料学特論 (資料: 英語 + 日本語版)

Lecture materials for numerical analyses (by Kamiya)  
数値解析に関する講義資料・pythonプログラム (神谷担当分)

#### Update News:

- June 20, 08:05 lecture materials on June 20 have been updated ([20250620InterporlateSmoothing.zip](#))
- June 19, 10:09 lecture materials on June 20 have been uploaded ([20250619InterporlateSmoothing.zip](#))

**We will wait for five minutes.**

**In the meantime, please make sure to download the lecture materials**

[▶ detailed history](#)

#### Getting Started with python (pythonプログラミングを始める前に)

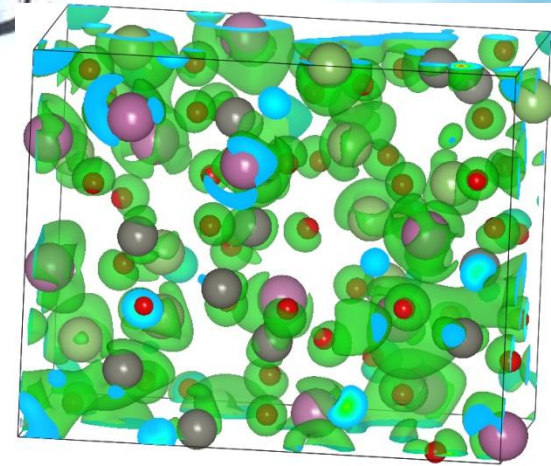
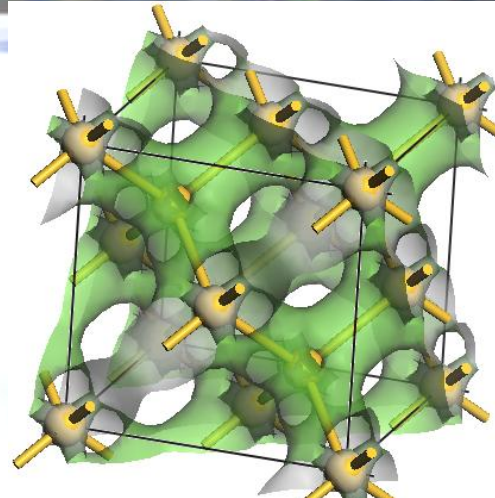
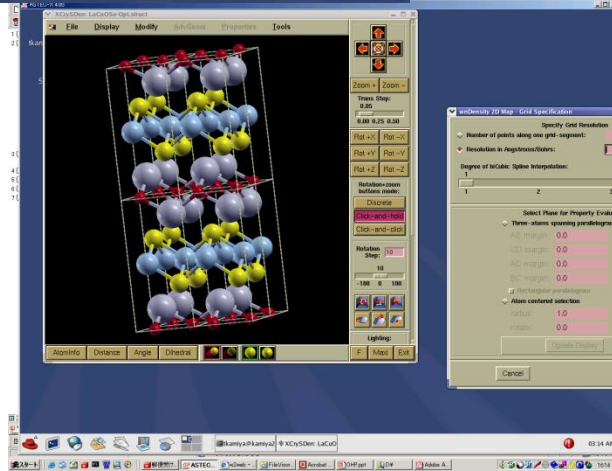
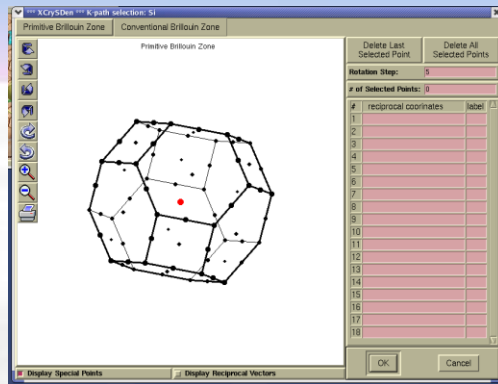
本講義では、pythonは必須ではありませんが、アルゴリズムの理解と今後の研究に役に立ちますので、余裕のある人は試してみてください。  
python is not a requirement for this class, but it will help your understanding about the algorithms to be learned and also assist your future research.

- [python programming \(Japanese\)](#)

# Computational Materials Science

## 計算材料学特論

Toshio Kamiya  
神谷利夫



# Class Schedule

Lecture materials (Kamiya's part): [http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/D2MatE\\_programs.html?page=cms](http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/D2MatE_programs.html?page=cms)

- #01 June 10 (Tue) Kamiya (Fundamentals of computer, Sources of errors (コンピュータの基礎、誤差))
- #02 June 13 (Fri) Kamiya (Numerical differentiation/integration (数値微分/積分))
- #03 June 17 (Tue) Kamiya (Differential equation (微分方程式), Molecular dynamics (分子動力学法),  
Interpolation (補間), Smoothing (平滑化))
- #04 June 20 (Fri) Kamiya (Smoothing (平滑化), Linear least-squares method (線形最小二乗法),  
Numerical solutions of equations (方程式の数値解法))
- #05 June 24 (Tue) Canceled
- #06 June 27 (Fri) Kamiya (Nonlinear optimization (非線形最適化), Fourier transformation (フーリエ変換))
- #07 July 1 (Tue) Kamiya, Matrix (行列)
- #08 July 4 (Fri) Sasagawa (Review of quantum theory 1: 量子論おさらい1)
- #09 July 8 (Tue) Sasagawa (Review of quantum theory 2: 量子論おさらい2)
- #10 July 11 (Fri) Sasagawa (First principles calculations: basics 1 第一原理計算: 基礎1)
- #11 July 15 (Tue) Sasagawa (First principles calculations: basics 2 第一原理計算: 基礎2)
- #12 July 18 (Fri) Sasagawa (First principles calc.: applications 1 第一原理計算: 応用1)
- #13 July 22 (Tue) Sasagawa (First principles calc.: applications 2 第一原理計算: 応用2)
- #14 July 25 (Fri) Sasagawa (Classical and Quantum Computers 古典および量子コンピュータ)

# Evaluation (Kamiya)

- **Small quiz**

**Do not evaluate correctness of the answers**

**but consider **how you think** and answer them**

Example of bad case: answering only resulting values in Excel

Good case: Answer with Excel equations / python program etc  
so that I can check how you solve the assignments

- **Term-end assignment**

**Problems will be given at the end of Q2 from LMS**

# **Explanation of the answers**

**課題解答の解説**

# PROBLEM, June 17

## PROBLEM:

- (i) By filling the  $dx/dt$  and the  $x(t)$  columns in diffeq.xlsx, solve  $dx(t) / dt = -x(t)\sin(\pi t)$  using the Euler method.

## Conditions:

$t$  starts from 0 and ends at 3.0 with the time step of 0.1.

$$x(0) = 1.0$$

**Euler formula:**  $\frac{dx(t)}{dt} = f(x(t), t) = -x(t)\sin(\pi t)$

$$x(t + \Delta t) = x(t) + \Delta t \cdot f(x(t), t)$$

## Typical mistake:

$$x(t + \Delta t) = x(t) + \Delta t \cdot f(x(t + \Delta t), t + \Delta t)$$

This calculation is not possible if  $f(x, t)$  includes  $x$  explicitly.

See diffeq2\_answer.xlsx



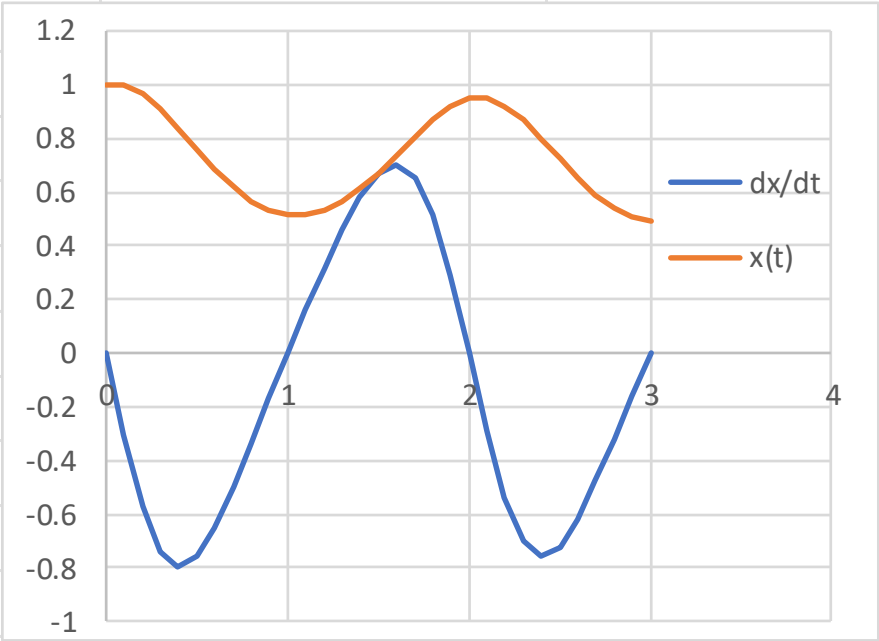
# PROBLEM, June 17

## PROBLEM:

(i) By filling the  $dx/dt$  and the  $x(t)$  columns in diffeq.xlsx, solve  $dx(t) / dt = -x(t)\sin(\pi t)$  using the Euler method.

Condition:  $x(0) = 1.0$

t	dx/dt	x(t)		dx/dt	x(t)
0	0	1		=-C2*SIN(3.14*A2)	1
0.1	-0.30887	1		=-C3*SIN(3.14*A3)	=C2+(A3-A2)*B2
0.2	-0.56938	0.969113			
0.3	-0.73771	0.912175			
0.4	-0.7972	0.838404			
0.5	-0.75868	0.758684			
0.6	-0.6496	0.682816			
0.7	-0.50026	0.617856			
0.8	-0.33435	0.56783			
0.9	-0.16587	0.534395			
1	-0.00082	0.517809			
1.1	0.159123	0.517726			
1.2	0.312839	0.533638			



Euler formula:  $\frac{dx(t)}{dt} = -x(t)\sin(\pi t)$   
 $x(t + \Delta t) = x(t) + \Delta t \cdot f(x(t), t)$

# PROBLEM, June 20

- Submit electronic file(s) via LMS until the midnight of June 22nd  
(If LMS doesn't work, send the files to [kamiya.t.aa@m.titech.ac.jp](mailto:kamiya.t.aa@m.titech.ac.jp).  
In this case, file name must include your STUDENT ID and FULL NAME)

## PROBLEM:

Smoothen the data DOS(E) in dos.xlsx  
by simple moving average method and polynomial fit method.

Add them and plot the raw DOS(E) and the smoothed data in an Excel file.

You can choose smoothing parameters as you like, but explicitly describe them.  
Submit the excel file.



# Smoothing

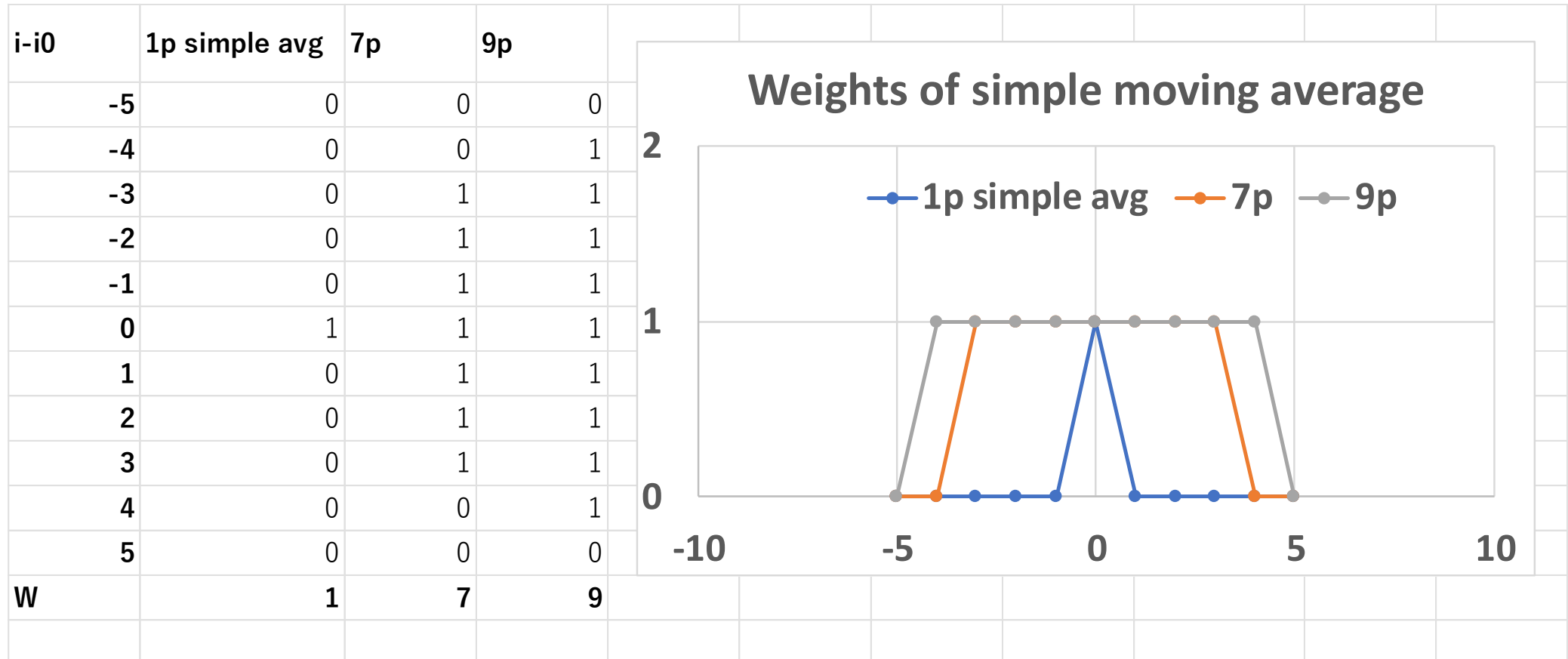
## 平滑化

# Smoothing

## Simple moving average (2m+1 points)

$$y_{i,smoothed} = \frac{1}{2m+1} \sum_{j=i-m}^{i+m} y_j$$

### Weight

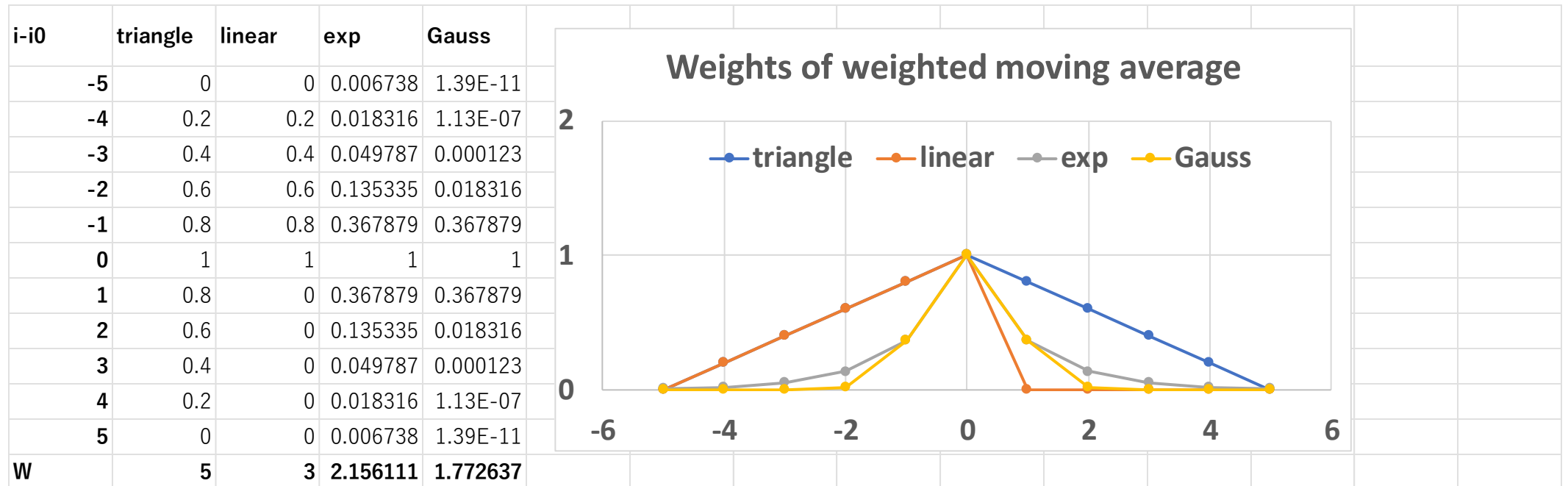


# Smoothing

## Weighted moving average (2m+1 points)

$$y_{i,smoothed} = \sum_{j=i-m}^{i+m} w_j y_j / \sum_{j=i-m}^{i+m} w_j$$

$w_i$



# Weights of polynomial fit (Savitzky-Golay method)

## 多項式適合法 (Savitzky-Golay法) の重み

南茂夫, 科学計測のための波形データ処理, CQ出版社 (1986)

**Table 5.1 Weights for  
order 2 and 3 polynomial fit**

# of points N	25	23	21	19	17	15	13	11	9	7	5
m=int(N/2)	12	11	10	9	8	7	6	5	4	3	2
-12	-253										
-11	-138	-210									
-10	-33	-105	-171								
-9	62	-10	-76	-136							
-8	147	75	9	-51	-105						
-7	222	150	84	24	-30	-78					
-6	287	215	149	89	35	-13	-55				
-5	342	270	204	144	90	42	0	-36			
-4	387	315	249	189	135	87	45	9	-21		
-3	422	350	284	224	170	122	80	44	14	-10	
-2	447	375	309	249	195	147	105	69	39	15	-3
-1	462	390	324	264	210	162	120	84	54	30	12
0	467	395	329	269	215	167	125	89	59	35	17
1	462	390	324	264	210	162	120	84	54	30	12
2	447	375	309	249	195	147	105	69	39	15	-3
3	422	350	284	224	170	122	80	44	14	-10	
4	387	315	249	189	135	87	45	9	-21		
5	342	270	204	144	90	42	0	-36			
6	287	215	149	89	35	-13	-55				
7	222	150	84	24	-30	-78					
8	147	75	9	-51	-105						
9	62	-10	-76	-136							
10	-33	-105	-171								
11	-138	-210									
12	-253										
Normalization factor	5175	4025	3059	2261	1615	1105	715	429	231	105	35

**Order 1: Simple moving average**

**Orders 2 and 3 have the same weights**

**Weights for order 2 and 3 using (2m+1) points ((2m+1)点を用いた2,3次多項式適合の重み)**

$$w_{23}(j) = 3m(m+1) - 1 - 5j^2$$

$$j = -m, \dots, -1, 0, 1, \dots, m$$

$$W_{23} = (4m^2 - 1)(2m + 3)/3$$

# Smoothing

## Order 2 and 3 polynomial fit using $(2m+1)$ points

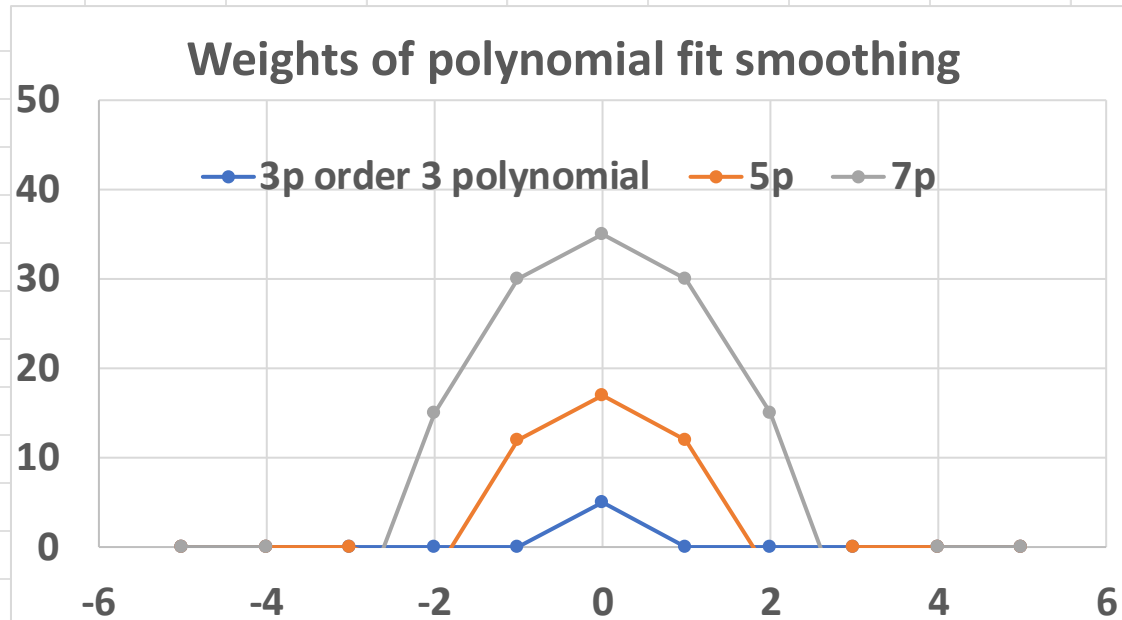
$$w_{23}(j) = 3m(m+1) - 1 - 5j^2 \quad j = -m, \dots, -1, 0, 1, \dots, m$$

$$W_{23} = (4m^2 - 1)(2m + 3)/3$$

$$y_{i,smoothed} = \frac{1}{W_{23}} \sum_{j=i-m}^{i+m} w_{23}(j) y_j$$

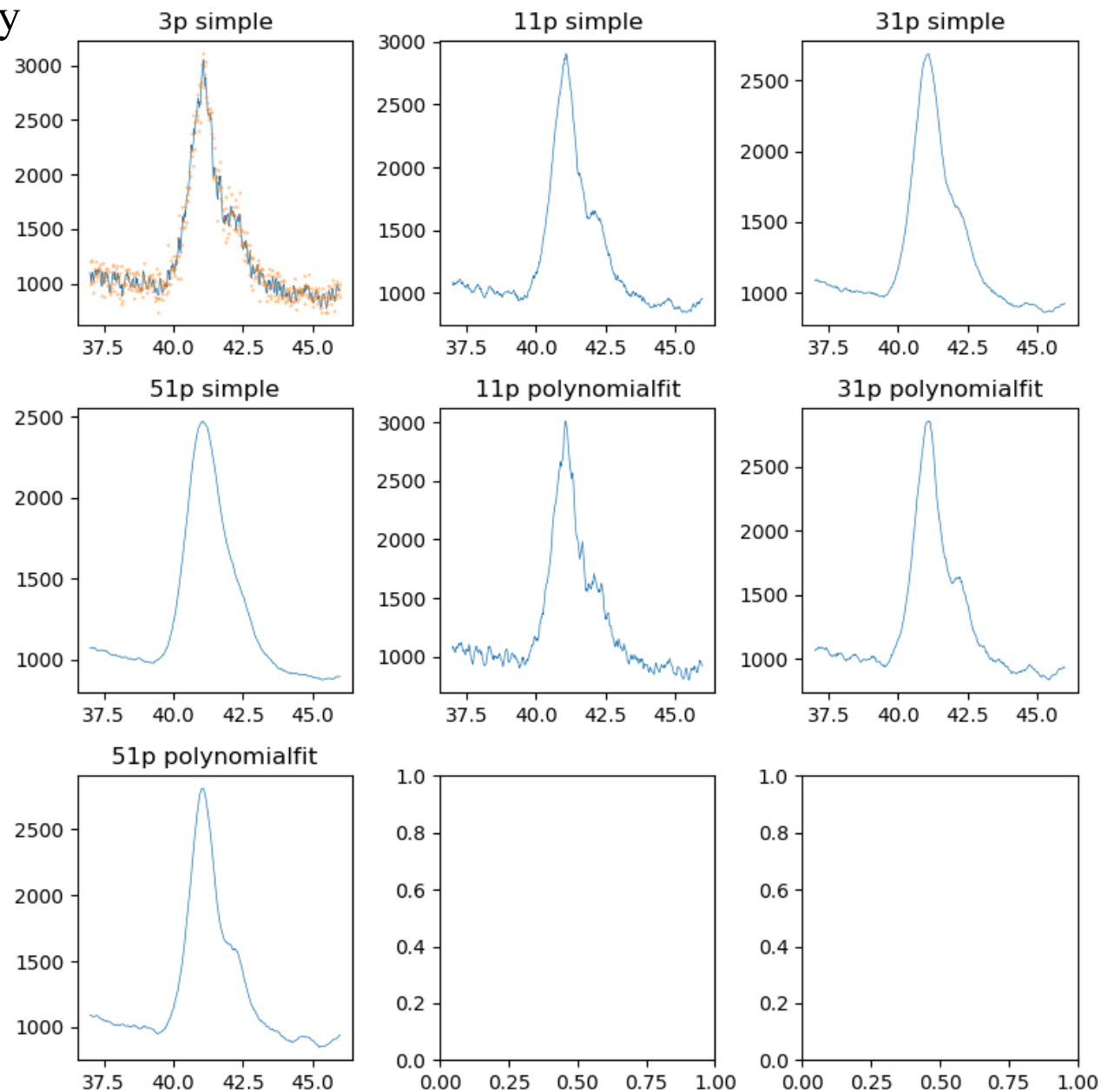
$w_{23}(j)$

i-i0	3p order 3 polynomial	5p	7p
-5	0	0	0
-4	0	0	0
-3	0	0	-10
-2	0	-3	15
-1	0	12	30
0	5	17	35
1	0	12	30
2	0	-3	15
3	0	0	-10
4	0	0	0
5	0	0	0
W	5	35	105



# Program: smoothing.py

Usage: python smoothing.py



# Fourier transformation (フーリエ変換)

## Different definitions

$$\left\{ \begin{array}{ll} \text{FT (フーリエ変換)} & F(\omega) = \int_{-\infty}^{\infty} f(t) \exp(i\omega t) dt \\ \text{IFT (逆フーリエ変換)} & f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \exp(-i\omega t) d\omega \end{array} \right.$$
$$\left\{ \begin{array}{ll} \text{FT} & F(\omega) = \int_{-\infty}^{\infty} f(t) \exp(i2\pi ft) dt \\ \text{IFT} & f(t) = \int_{-\infty}^{\infty} F(\omega) \exp(-i2\pi ft) d\omega \end{array} \right.$$

## Features of Fourier transformation

- Convert time-dependent data to frequency data
- Convert position-dependent data to wavenumber data
- Origin of original data is converted to whole range of FT data
- Whole range of original data is converted to origin of FT data
- **IFT of FTed data recovers the original data**

Fourier変換したデータをFourier逆変換すると元のデータに戻る

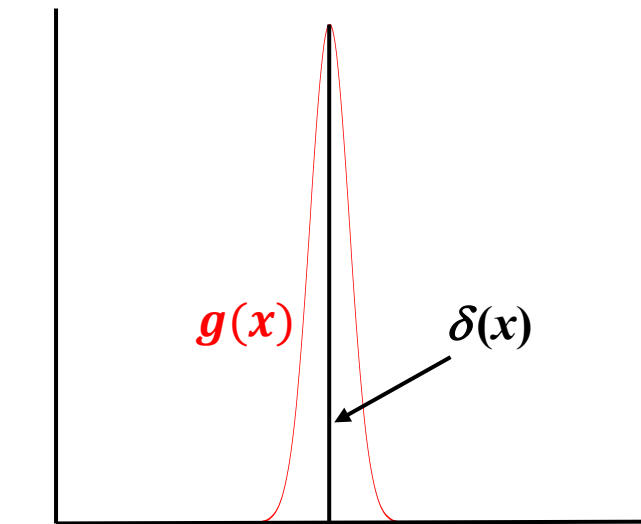


# Convolution (畳み込み)

$$(f * g)(x) = f^*(x) = \int_{-\infty}^{\infty} f(x')g(x - x')dx'$$

Observed peak has a finite width  
originating **from apparatus function  $g(x)$**   
Even if the intrinsic peak has zero width  
(delta function  $\delta(x)$ )

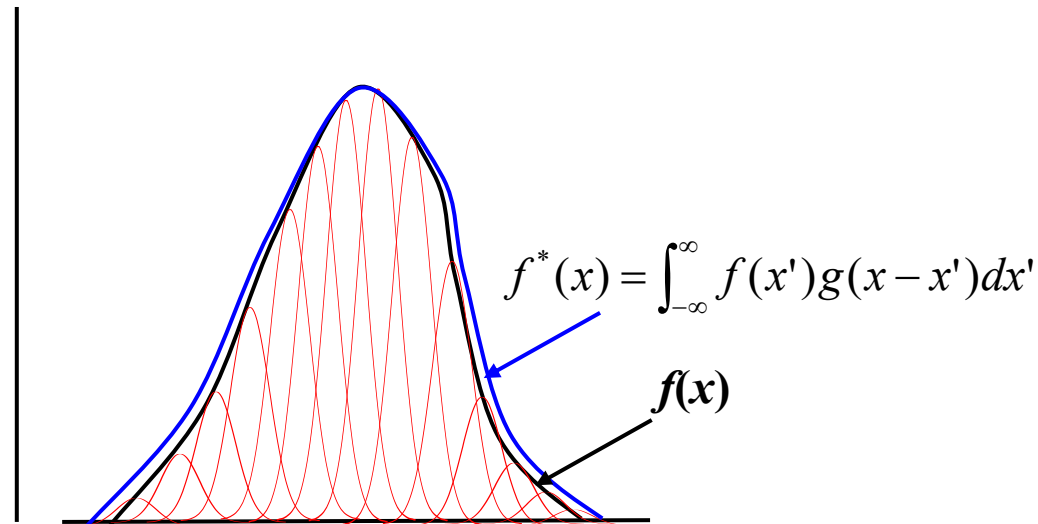
試料本来のデータは線幅ゼロ ( $\delta$ 関数) でも、  
測定値は**装置関数  $g(x)$**  の広がりを持つ



$$\int_{-\infty}^{\infty} g(x)dx = 1$$

For a real case a sample has an intrinsic peak  
 $f(x)$ , the observed peak will be **a convolution**  
**of  $f(x)$  and apparatus function  $g(x)$ ,  $f^*(x)$ .**

試料本来のデータは  $f(x)$  でも、測定されるのは  
装置関数  $g(x)$  の畳み込みをした  $f^*(x)$



# Convolution: Matrix representation (行列表示)

南茂夫 編著、科学計測のための波形データ処理、CQ出版 (1986年)

$$f^*(x_i) = \int_{-\infty}^{\infty} f(x')g(x_i - x')dx' = N^{-1} \sum_{j=1}^N f(x_j)g(x_i - x_j)$$

$$\begin{pmatrix} f_1^* \\ f_2^* \\ f_3^* \\ \vdots \\ f_{N-1}^* \\ f_N^* \end{pmatrix} = \begin{pmatrix} g_0 & g_{-1} & \cdots & g_{-(N-3)} & g_{-(N-2)} & g_{-(N-1)} \\ g_1 & g_0 & \cdots & g_{-(N-4)} & g_{-(N-3)} & g_{-(N-2)} \\ g_2 & g_1 & \ddots & \vdots & g_{-(N-4)} & g_{-(N-3)} \\ \vdots & \vdots & \cdots & g_0 & g_{-1} & \vdots \\ g_{N-2} & g_{N-3} & \cdots & g_1 & g_0 & g_{-1} \\ g_{N-1} & g_{N-2} & \cdots & g_2 & g_1 & g_0 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}$$

$f^*(x)$   
Observed signal

$g(x_i - x_j)$   
Apparatus function

$f(x)$   
Intrinsic signal

**Very often, matrix  $g_{ij}$  is a band matrix with maxima at diagonal**

(行列 $g_{ij}$ は対角要素に最大値を持つ帯行列になることが多い)

# Smoothing by convolution (smearing)

畳み込みによる平滑化 (ぼかし)

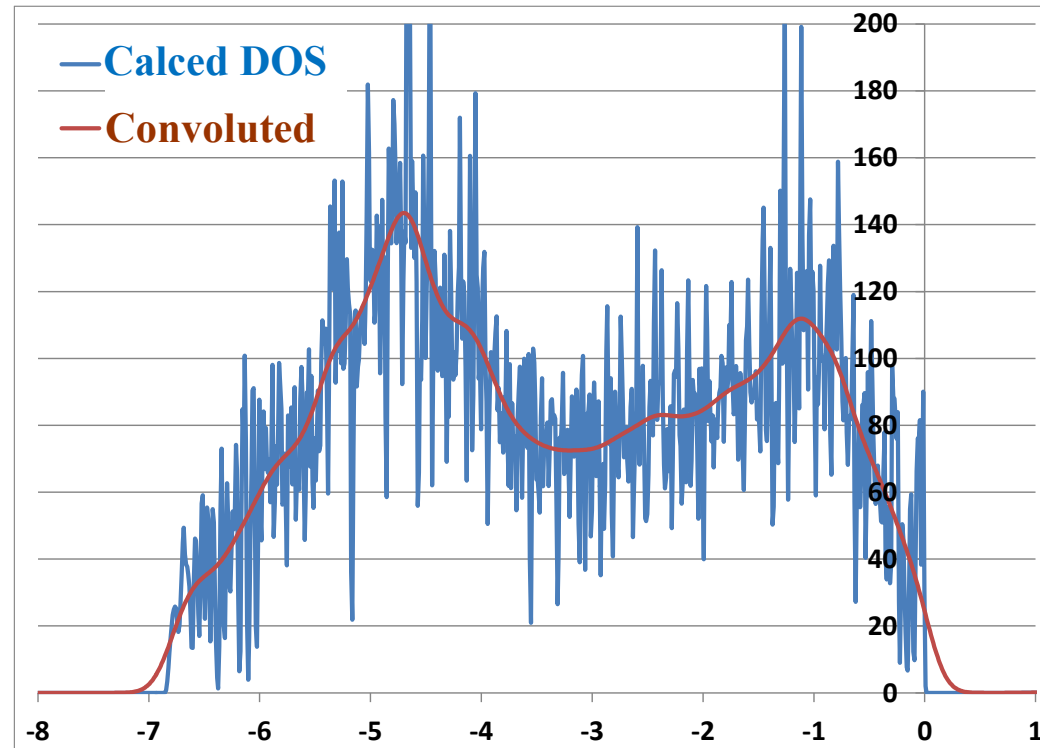
**Density of state (DOS) function calculated by density functional theory**

密度汎関数計算で得たa-InGaZnO<sub>4</sub>の状態密度

**Problem: Many noise, difficult to read**

**Add finite-width Gauss function to each data** (それぞれのデータにGauss関数の広がり)

$$G(E) = \exp(-[(E - E_0)/w]^2) \quad (w = 0.2 \text{ eV})$$



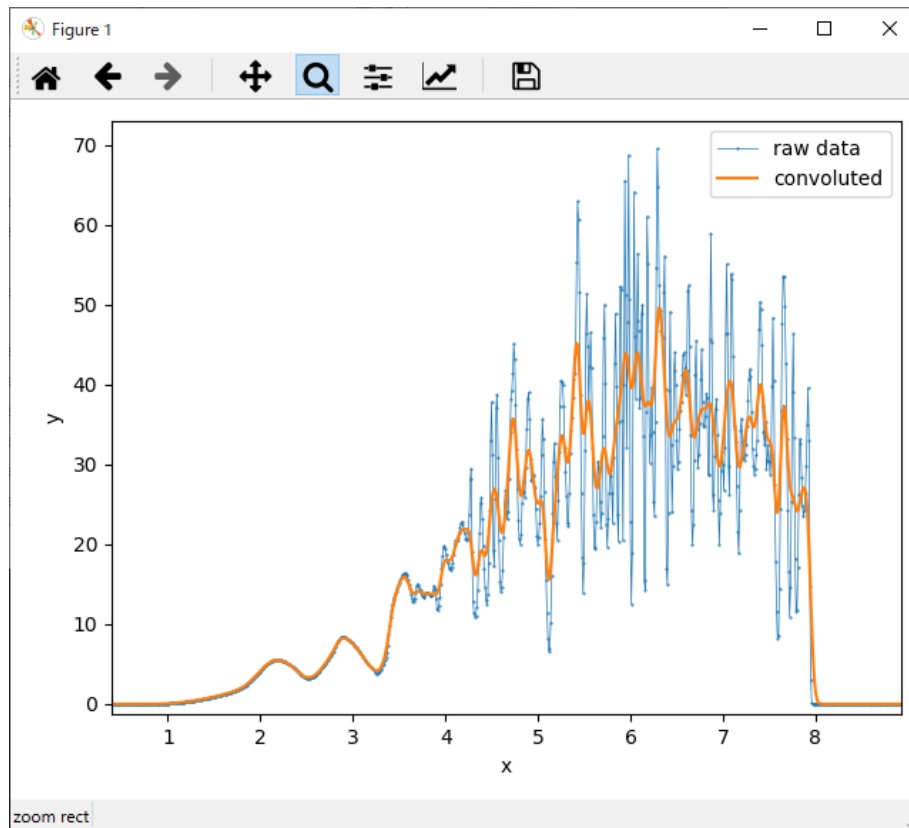
**Note: Estimation of band, edge energies will have the errors originating from the smearing width  $w$**

# Program: convolution.py

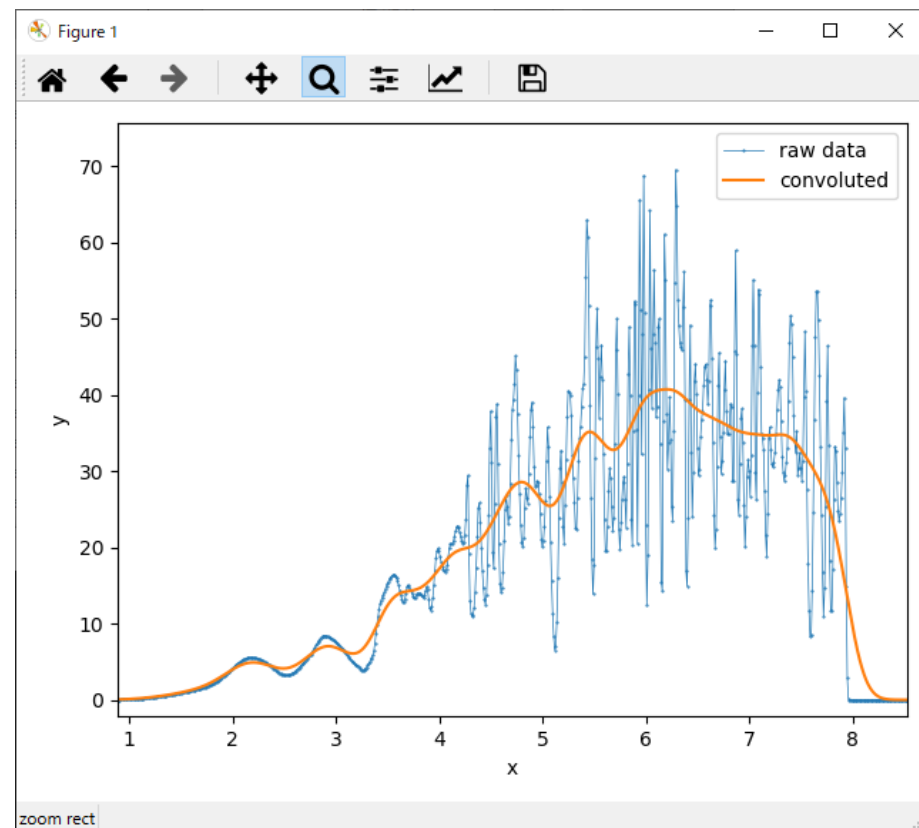
Usage: python convolution.py width

width: width of Gaussian function to convolute

python convolution.py 0.05



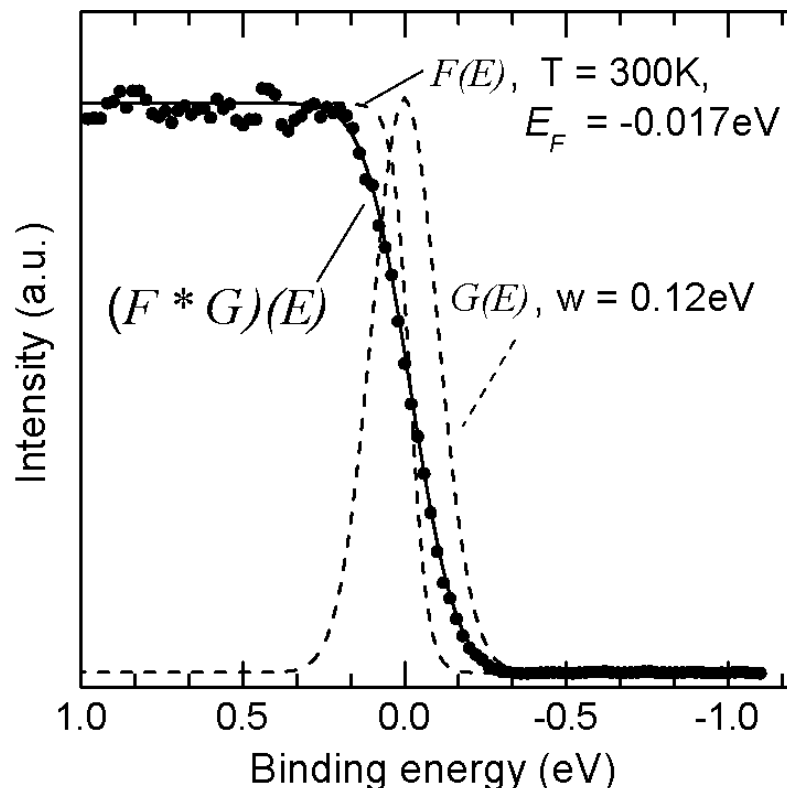
python convolution.py 0.2



# Convolution (畳み込み)

$$(f * g)(x) = f^*(x) = \int_{-\infty}^{\infty} f(x')g(x - x')dx'$$

## Example: UPS spectrum of Au Fermi edge



Intrinsic sample spectrum

$S(E)$

Apparatus function

$$G(E) = G_0 \exp(-[(E - E_0)/aw]^2)$$

Fermi-Dirac distribution

$$f(E) = 1/(1 + \exp[(E - E_F)/k_B T]) \quad \text{eq. (1)}$$

Observed spectrum

$$I(x) = \int_{-\infty}^{\infty} S(E')G(E - E')f(E - E')dE'$$

Assuming constant  $S(E)$  for Au reference,  
 $G(E)$  is determined by fitting eq. (1) to  $I(x)$

$w = 0.12 \text{ eV}$

**$S(E)$  for different sample is obtained by deconvolution using the  $G(E)$  obtained by the reference spectrum**

$G(E)$ がわかると、他の実測スペクトルから 逆畳み込みで  $S(E)$  がわかる

# Filter and convolution

First differential

Convolution:  
Matrix product of  
data vector and filter

$$\left. \frac{dy}{dx} \right|_2 = \frac{1}{2h} \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Filter

$$\left. \frac{d^2y}{dx^2} \right|_2 = \frac{1}{2h^2} \begin{pmatrix} 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$\left. \frac{dy}{dx} \right|_0 \sim \frac{y_1 - y_{-1}}{2h}$$

$$\left. \frac{d^2y}{dx^2} \right|_0 \sim \frac{y_1 - 2y_0 + y_{-1}}{h^2}$$

$$y_{2,s} \sim \frac{y_1 + y_2 + y_3}{3}$$

Second differential

Simple moving average (3 points)

$$y_{2,s} = \frac{1}{3} \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Weighted moving average (3p one-side triangle)

$$y_{2,s} = \frac{1}{3} \begin{pmatrix} 0 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Weighted moving average (3p double-side triangle)

$$y_{2,s} = \frac{1}{4} \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Polynomial fit smoothing ( $2m+1$  points)

$$y_{3,s} = \frac{1}{35} \begin{pmatrix} -3 & 12 & 17 & 12 & -3 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix}$$

Differentiation, smoothing, convolution may be performed with the same convolution program by adopting appropriate filters.

微分、平滑化、コンボリューションは、 フィルターを変えるだけで 同じコンボリューションプログラムを流用できる

# Weights for various smoothing

Weighted moving average (2m+1 points)

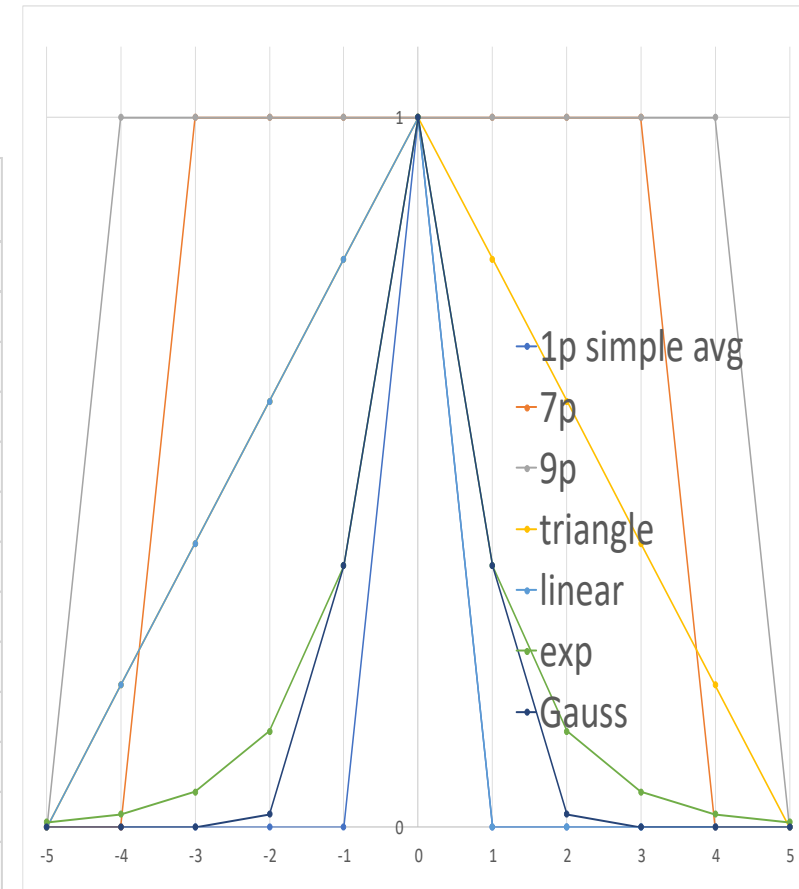
$$y_{i,smoothed} = \frac{1}{\sum_{j=i-m}^{i+m} w_j} \sum_{j=i-m}^{i+m} w_j y_j$$

e.g., one-side triangle:  $(w_{-1}, w_0, w_1) = \frac{1}{3}(1, 2, 0)$

triangle :  $(w_{-1}, w_0, w_1) = \frac{1}{4}(1, 2, 1)$

Gaussian :  $w_i = \frac{1}{\sum e^{(ai)^2}} e^{(ai)^2}$

i-i0	1p simple avg	7p	9p	triangle	linear	exp	Gauss	3p order 3 polynomial	5p	7p
-5	0	0	0	0	0	0.006738	1.39E-11	0	0	0
-4	0	0	1	0.2	0.2	0.018316	1.13E-07	0	0	0
-3	0	1	1	0.4	0.4	0.049787	0.000123	0	0	-10
-2	0	1	1	0.6	0.6	0.135335	0.018316	0	-3	15
-1	0	1	1	0.8	0.8	0.367879	0.367879	0	12	30
0	1	1	1	1	1	1	1	5	17	35
1	0	1	1	0.8	0	0.367879	0.367879	0	12	30
2	0	1	1	0.6	0	0.135335	0.018316	0	-3	15
3	0	1	1	0.4	0	0.049787	0.000123	0	0	-10
4	0	0	1	0.2	0	0.018316	1.13E-07	0	0	0
5	0	0	0	0	0	0.006738	1.39E-11	0	0	0
W	1	7	9	5	3	2.15611	1.77264	5	35	105
							m=	1	2	3



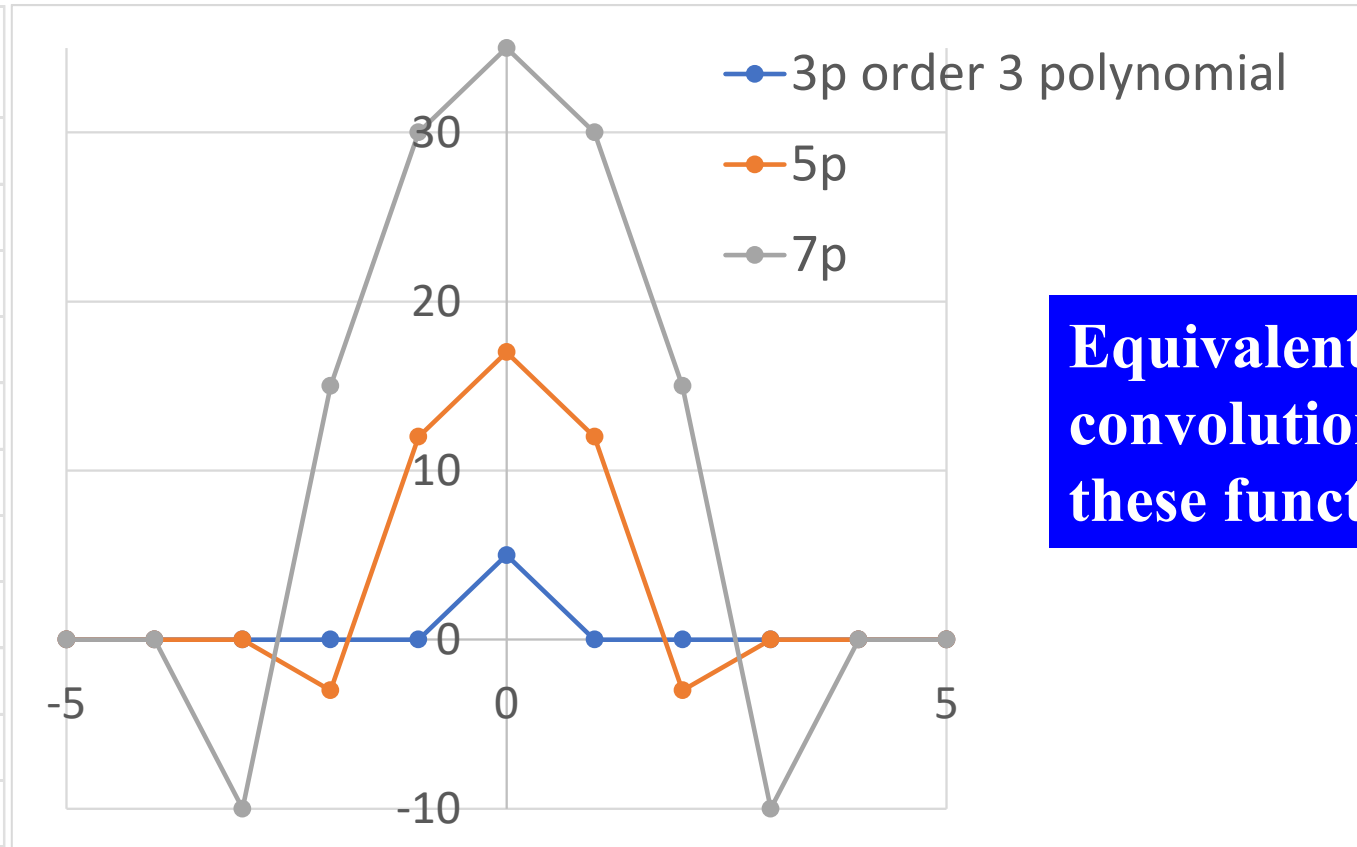
Equivalent to convolution  
using these functions



# Weights

Order 2 and 3 polynomial fit using  $(2m+1)$  points

i-i0	3p order 3 polynomial	5p	7p
-5	0	0	0
-4	0	0	0
-3	0	0	-10
-2	0	-3	15
-1	0	12	30
0	5	17	35
1	0	12	30
2	0	-3	15
3	0	0	-10
4	0	0	0
5	0	0	0
W	5	35	105
	1	2	3



Equivalent to convolution using these functions

# Smoothing by python library

## Smoothing.py

Simple moving average

# make a constant filter with weight of  $1/N$

`w = np.ones(nsmooth) / nsmooth`

# convolution

`ys = np.convolve(y, w, mode = 'same')`

Polynomial smoothing: **Savizky-Golay filter**

`ys = scipy.signal.savgol_filter(y, nsmooth, norder, deriv = 0)`

nsmooth: number of smoothing points

norder : order of polynomial

deriv : order of differentiation

**注意:** `savgol_filter()` takes differentiation to specify `deriv = 1`,  
but its absolute values are different from the first differentials  
because `savgol_filter()` does not know x-axis values.

Divide the results by  $h^{\text{deriv}}$  after smoothing if you need absolute values

*Note: check the final results by yourself*

# Multi-dimensional filter, convolution, image processing

$$\text{filter: } \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad y'_{22} = \sum_{i,j=1,2,3} a_{ij} y_{ij}$$

e.g., the convolution of  $y_{22}$  is given by a sum of each product of

$$\text{data } \begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix} \text{ and } \text{filter } \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

**NOTE: different from convolution in mathematics**

**X differential filter (edge detection)**

$$\frac{1}{2h} \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

**Y differential filter (edge detection)**

$$\frac{1}{2h} \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

**Diagonal differential filter (edge detection)**

$$\frac{1}{2h} \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$$

**Smoothing filter (smearing/blur)**

⇔ sharpening can be done by deconvolution

$$\frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

**Convolved Neural Network (畳み込みニューラルネットワーク)**

Insert a convolution layer in multilayer neural network to learn the elements of the filter

We may know how the filter works by analyzing the values of the filter elements

# Deconvolution (逆畳み込み)

$$(f * g)(x) = f^*(x) = \int_{-\infty}^{\infty} f(x')g(x - x')dx'$$

**Fourier transformation (FT)**  $F^*(k) = \int_{-\infty}^{\infty} f^*(x) \exp(ikx)dx$

**Inverse Fourier transformation (IFT)**  $f(x) = \int_{-\infty}^{\infty} F(k) \exp(-ikx)dk$   
 $g(x) = \int_{-\infty}^{\infty} G(k') \exp(-ik'x)dk'$

$$\begin{aligned} F^*(k) &= \int_{-\infty}^{\infty} f(x)g(x - x') \exp(ikx)dx dx' \\ &= \int_{-\infty}^{\infty} f(x) \left( \int g(x - x') \exp(ikx)dx \right) dx' \\ &= \int_{-\infty}^{\infty} f(x) \left( \int g(x) \exp(ik(x + x'))dx \right) dx' \\ &= \int_{-\infty}^{\infty} f(x)G(k) \exp(ikx')dx' \\ &= F(k)G(k) \end{aligned}$$

**$f(x)$  can be obtained by IFT of  $F(k) = F^*(k) / G(k)$ , but usually is vulnerable against small perturbations like noise**

$F(k) = F^*(k) / G(k)$ を計算して逆フーリエ変換で  $f(x)$  が得られる  
=> ノイズなどがあると不安定で解が発散しやすい

# Convolution: Matrix representation (行列表示)

南茂夫 編著、科学計測のための波形データ処理、CQ出版 (1986年)

$$f^*(x_i) = \int_{-\infty}^{\infty} f(x')g(x_i - x')dx' = N^{-1} \sum_{j=1}^N f(x_j)g(x_i - x_j)$$

$$\begin{pmatrix} f_1^* \\ f_2^* \\ f_3^* \\ \vdots \\ f_{N-1}^* \\ f_N^* \end{pmatrix} = \begin{pmatrix} g_0 & g_{-1} & \cdots & g_{-(N-3)} & g_{-(N-2)} & g_{-(N-1)} \\ g_1 & g_0 & \cdots & g_{-(N-4)} & g_{-(N-3)} & g_{-(N-2)} \\ g_2 & g_1 & \ddots & \vdots & g_{-(N-4)} & g_{-(N-3)} \\ \vdots & \vdots & \cdots & g_0 & g_{-1} & \vdots \\ g_{N-2} & g_{N-3} & \cdots & g_1 & g_0 & g_{-1} \\ g_{N-1} & g_{N-2} & \cdots & g_2 & g_1 & g_0 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ f_N \end{pmatrix}$$

$f^*(x)$   
Observed signal

$g(x_i - x_j)$   
Apparatus function

$f(x)$   
Intrinsic signal

**Very often, matrix  $g_{ij}$  is band matrix with maxima at diagonal**

(行列 $g_{ij}$ は対角要素に最大値を持つ帯行列になることが多い)

# Deconvolution (逆畳み込み)

南茂夫 編著、科学計測のための波形データ処理、CQ出版 (1986年)

$$f^*(x_i) = N^{-1} \sum_{j=1}^N f(x_j) g(x_i - x_j)$$

**Deconvolution is carried out by solving the linear simultaneous equations,**

$$\begin{pmatrix} f_1^* \\ f_2^* \\ \vdots \\ f_N^* \end{pmatrix} = \begin{pmatrix} g_0 & g_{-1} & & g_{-(N-1)} \\ g_1 & g_0 & & \\ \vdots & & \ddots & \vdots \\ g_{N-1} & & \cdots & g_0 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{pmatrix}$$

**However, similar to the FFT method, usually vulnerable against noise**  
(フーリエ変換法と同様、ノイズなどがあると不安定で解が発散しやすい)

**Better way:**

- 1. Remove noise effects (smoothing etc) before deconvolution**
- 2. Use an iterative method (e.g., Jacobi method and Gauss-Seidel method) to solve the simultaneous equation, where noise-compensation process is included during the iteration process.**

# Jacobi / Gauss-Seidel method

Solve large-size simultaneous linear equations:

$$\begin{pmatrix} a_{11} & a_{12} & & a_{1N} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \vdots \\ a_{N1} & & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

**For  $(k+1)$ -th iteration,  $x_i^{(k+1)}$  is estimated from  $x_i^{(k)}$**   
(initial data may be chosen as  $x_i^{(0)} = b_i$ , uniform value  $x_i^{(0)} = 1$ , etc):

**(i) Jacobi method:**  $x_i^{(k+1)} = (b_i - \sum_{j \neq i}^N a_{ij} x_j^{(k)}) / a_{ii}$

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}) / a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}) / a_{22}$$

**(ii) Gauss-Seidel method:**  $x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^N a_{ij} x_j^{(k)}) / a_{ii}$

Using the known  $x_j^{(k+1)}$  values enhances convergence.

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}) / a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}) / a_{22}$$

Convergence is better for the Gauss-Seidel method,  
While parallelization is more easy for the Jacobi method.



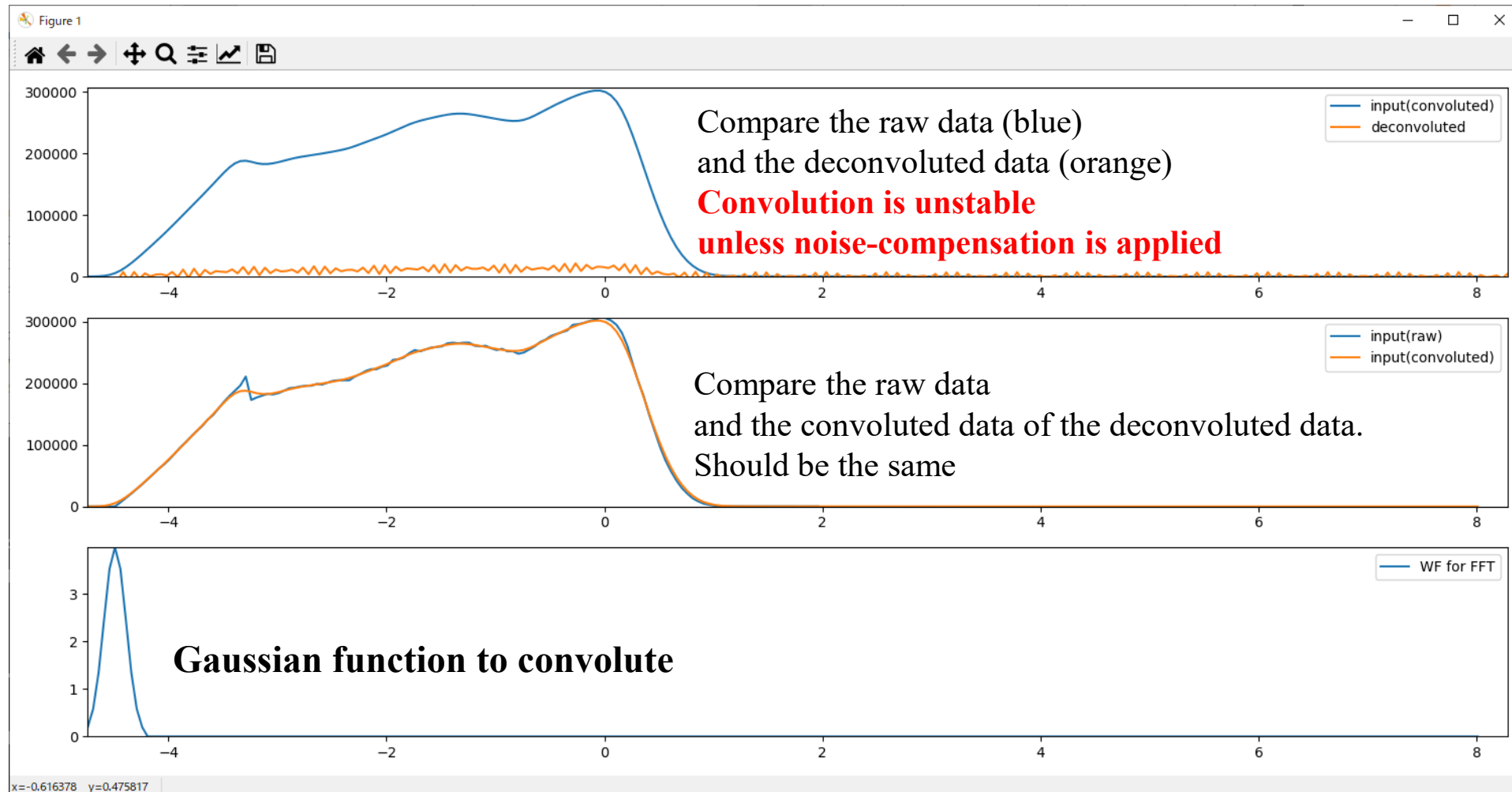
# Program: deconvolution.py

Usage: `python deconvolution.py file mode convmode smoothmode xmin xmax Wa Grange kzero klin`

see usage of the program output

`python deconvolution.py pes.csv fft full convolve+extend -4.5 2.0 0.12 2.0 5 5`

Use **FFT and iFFT** without smoothing



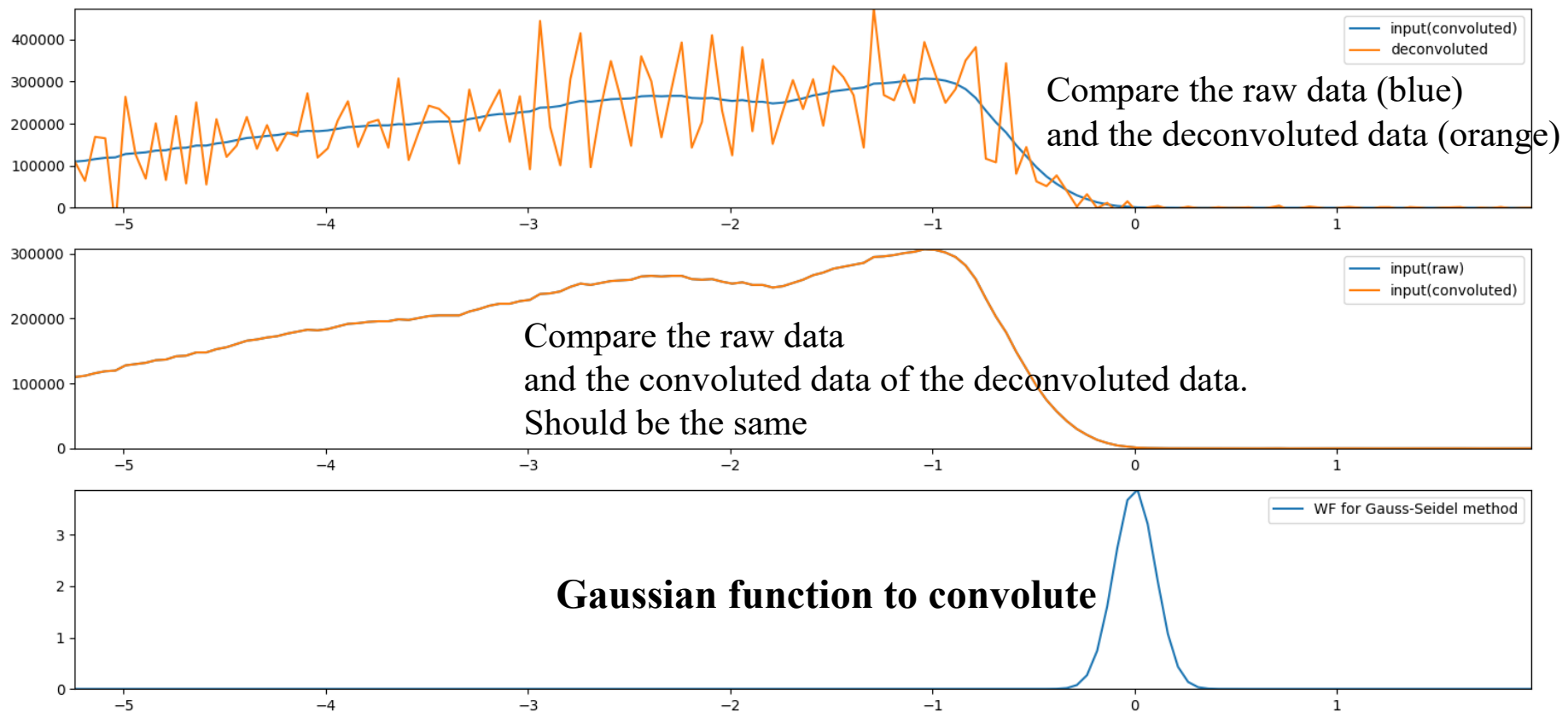
# Deconvolution: Gauss-Seidel method w/o smooting

Usage: `python deconvolution.py file mode xmin xmax Wa dump nmaxiter eps nsmooth zeroc`  
see usage of the program output

`python deconvolution.py pes.csv gs -6.0 2.0 0.12 1.0 300 1.0e-4 1 0`

Use **Gauss-Seidel (gs) method** with the width of the Gaussian function of 0.12 eV.

**No smoothing** is applied for each iteration.



# Program: Gauss-Seidel method with smoothing

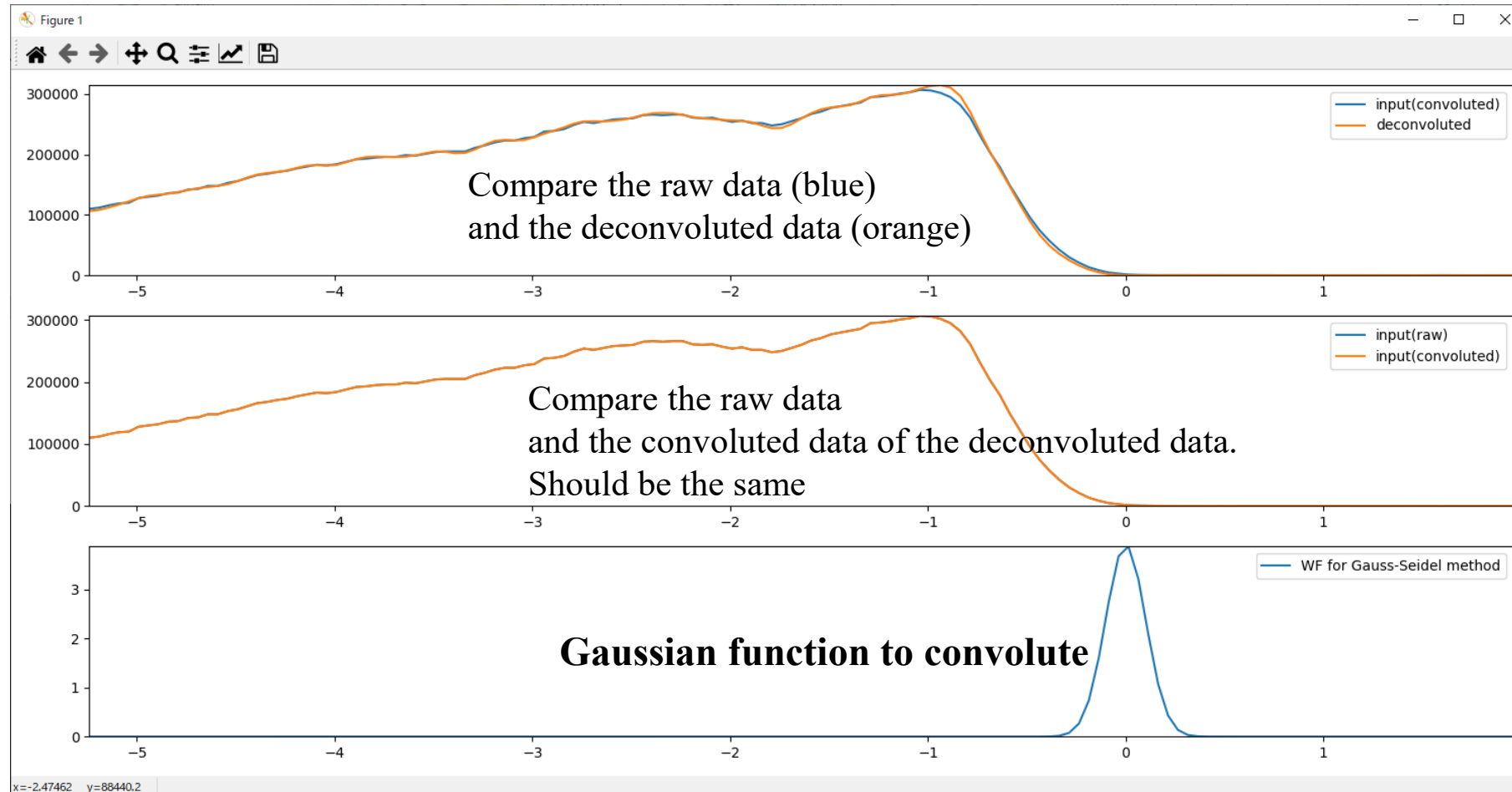
Usage: `python deconvolution.py file mode xmin xmax Wa dump nmaxiter eps nsMOOTH zeroC`

see usage of the program output

`python deconvolution.py pes.csv gs -6.0 2.0 0.12 1.0 300 1.0e-4 5 0`

Use **Gauss-Seidel (gs) method** with the width of the Gaussian function of 0.12 eV.

**5-point polynomial-fit average** is applied for each iteration.



**Linear least squares method (LSQ)**

線形最小自乘法

# Approximation of many sample points: Minimization (Optimization)

(多数の標本点の近似: 最小化問題)

How to determine most plausible parameters  $a$  and  $b$   
if observed data  $(x_1, y_1), \dots, (x_n, y_n)$  follow  $f(x) = a + bx$ ,  
※ Error  $\varepsilon_i$  should be considered:  $y_i = f(x_i) + \varepsilon_i$

Fundamental idea: Determine  $a$  and  $b$  so as to minimize (maximize)  
a target function  $S$  (e.g., error residual function (残差関数))

Mini max approximation: minimize  $\max_{a \leq x \leq b} |f(x_i) - y_i|$

$L_n$  norm:

$$S_n = \sum_i |f(x_i) - y_i|^n$$

$L_0$  norm:  $S_0 = 0$

Minimize  $L_1$  norm

$$: S = \sum |f(x_i) - y_i|$$

Least-squares (LSQ) method (最小自乗法) ( $L_2$  norm)

$$: S = \sum (f(x_i) - y_i)^2$$

$$S = \sum (a + bx_i - y_i)^2$$

$$dS/da = 2\sum (a + bx_i - y_i) = 2an + 2b\sum x_i - 2\sum y_i = 0$$

$$dS/db = 2\sum x_i (a + bx_i - y_i) = 2a\sum x_i + 2b\sum x_i^2 - 2\sum x_i y_i = 0$$

$$\begin{pmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix}$$

Even for  $f(x) = a + bx + cx^2 + \dots$ , only one matrix operation can give a final solution

# Mini-max approximation

入門 数値計算

Minimize  $\max_{a \leq x \leq b} |f(x_i) - y_i|$

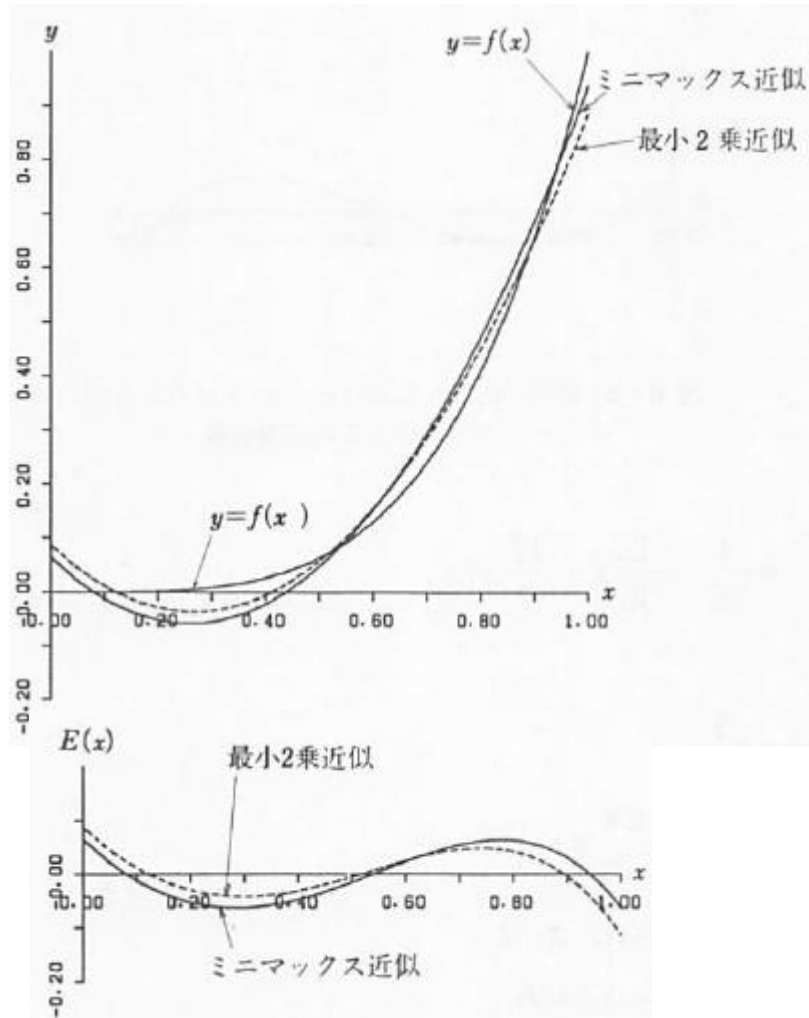


図 6・3 ミニマックス近似と最小2乗近似

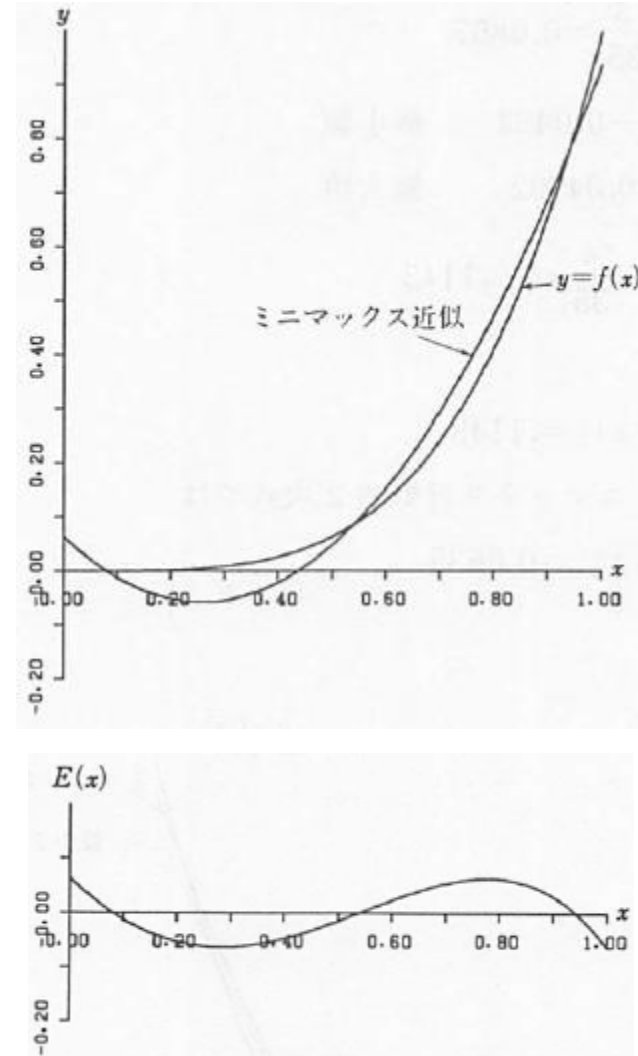


図 6・2 区間  $[0, 1]$  における  $f(x)=x^4$  の2次のミニマックス近似とその誤差曲線

# LSQ: Polynomial

線形最小二乘法: 多項式

$$f(x) = \sum_{k=0}^n a_k x^k \quad S = \sum_{i=1}^N (y_i - \sum_{k=0}^n a_k x_i^k)^2$$
$$\frac{dS}{da_l} = -2 \sum_{i=1}^N x_i^l (y_i - \sum_{k=0}^n a_k x_i^k) = 0$$

$$\sum_{k=0}^n \sum_{i=1}^N a_k x_i^{k+l} = \sum_{i=1}^N y_i x_i^l \quad (l = 0, 1, \dots, N)$$

$$\begin{pmatrix} n & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^N \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & & \sum x_i^{N+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & & \sum x_i^{N+2} \\ \vdots & & & \ddots & \\ \sum x_i^N & \sum x_i^{N+1} & \sum x_i^{N+2} & & \sum x_i^{2N} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i x_i^2 \\ \vdots \\ \sum y_i x_i^N \end{pmatrix}$$

$|x_i| > 1$  might cause overflow,

$|x_i| < 1$  might cause underflow errors.

=> Normalize the x range e.g. to  $[-1, 1]$ :  $x'_i = 2 \frac{x_i - x_{\text{mid}}}{x_{\text{max}} - x_{\text{min}}}$

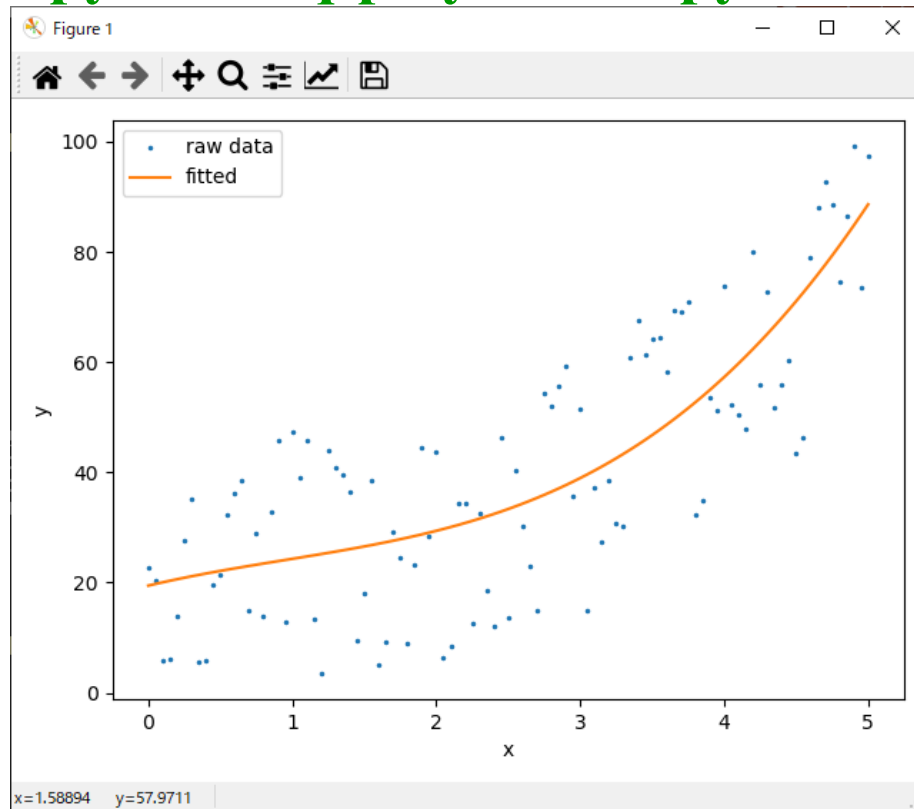
by average and standard deviation:  $x'_i = 2 \frac{x_i - x_{\text{average}}}{\sigma_x}$



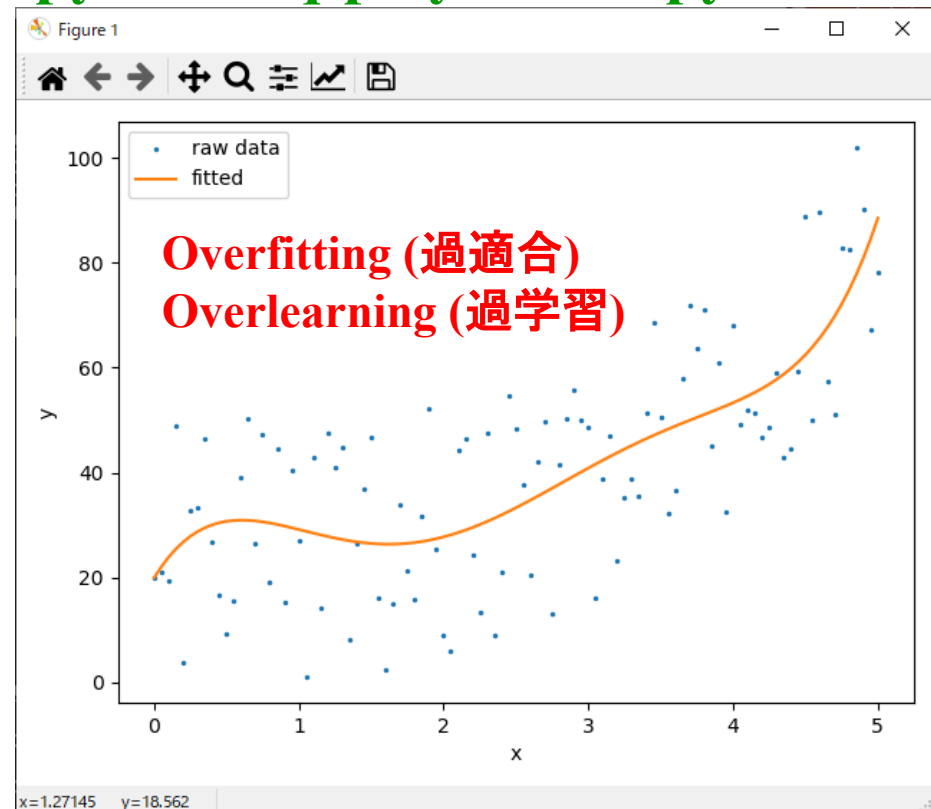
# Program: lsq-polynomial.py

Usage: `python lsq-polynomial.py norder`

`python lsq-polynomial.py 3`



`python lsq-polynomial.py 6`



# LSQ: General functions

## 線形最小二乗法: 一般関数の場合

$$f(x) = \sum_{k=1}^n a_k f_k(x) \quad S = \sum_{i=1}^N \left( y_i - \sum_{k=1}^n a_k f_k(x_i) \right)^2$$
$$\frac{dS}{da_l} = -2 \sum_{i=1}^N f_l(x_i) \left( y_i - \sum_{k=1}^n a_k f_k(x_i) \right) = 0$$

$$\begin{pmatrix} \sum f_1(x_i)f_1(x_i) & \sum f_1(x_i)f_2(x_i) & \sum f_1(x_i)f_3(x_i) & \cdots & \sum f_1(x_i)f_N(x_i) \\ \sum f_2(x_i)f_1(x_i) & \sum f_2(x_i)f_2(x_i) & \sum f_2(x_i)f_3(x_i) & & \sum f_2(x_i)f_N(x_i) \\ \sum f_3(x_i)f_1(x_i) & \sum f_3(x_i)f_2(x_i) & \sum f_3(x_i)f_3(x_i) & & \sum f_3(x_i)f_N(x_i) \\ \vdots & & & \ddots & \\ \sum f_N(x_i)f_1(x_i) & \sum f_N(x_i)f_2(x_i) & \sum f_N(x_i)f_3(x_i) & & \sum f_N(x_i)f_N(x_i) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{pmatrix} = \begin{pmatrix} \sum y_i f_1(x_i) \\ \sum y_i f_2(x_i) \\ \sum y_i f_3(x_i) \\ \vdots \\ \sum y_i f_N(x_i) \end{pmatrix}$$

**If  $f(x)$  is linear with respect to fitting parameters,  
final solution is obtained by one matrix operation**

係数に関して線形であれば、1度の行列計算で最終解が得られる

**ex.**  $f(x) = a + b \log x + c / x$

$$f(x, y) = a + bxy + cy / x$$

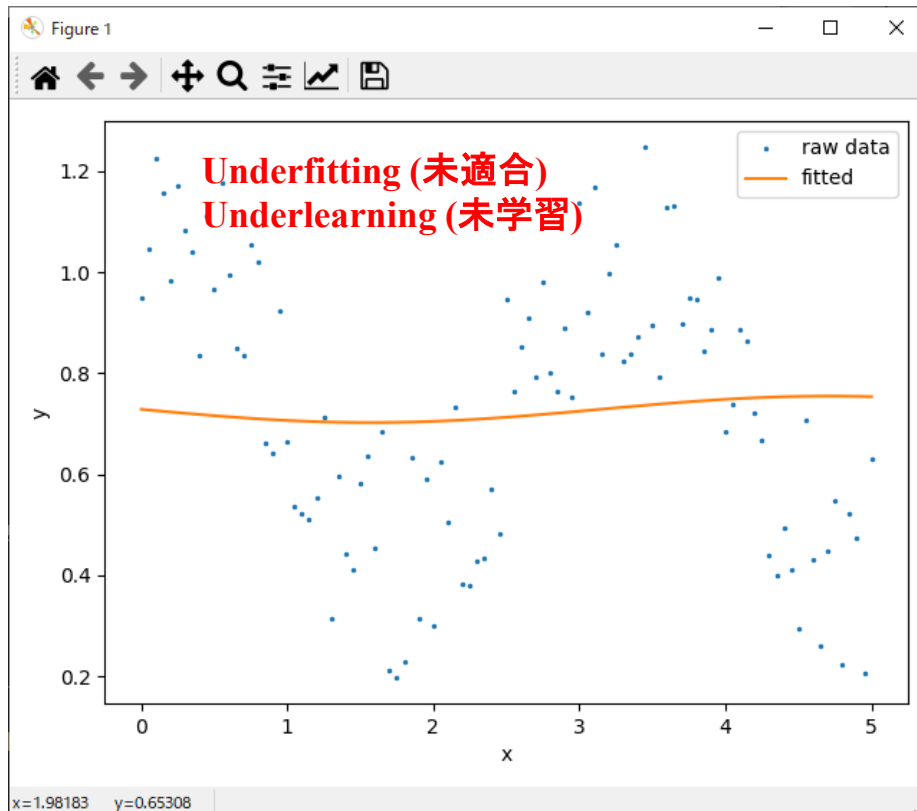
# Program: lsq-general.py

Usage: `python lsq-general.py nfunc`

fit to  $y = c_0 + c_1 \sin x + c_2 \cos x + c_3 \sin 2x + c_4 \cos 2x + c_5 \sin 3x + c_6 \cos 3x$

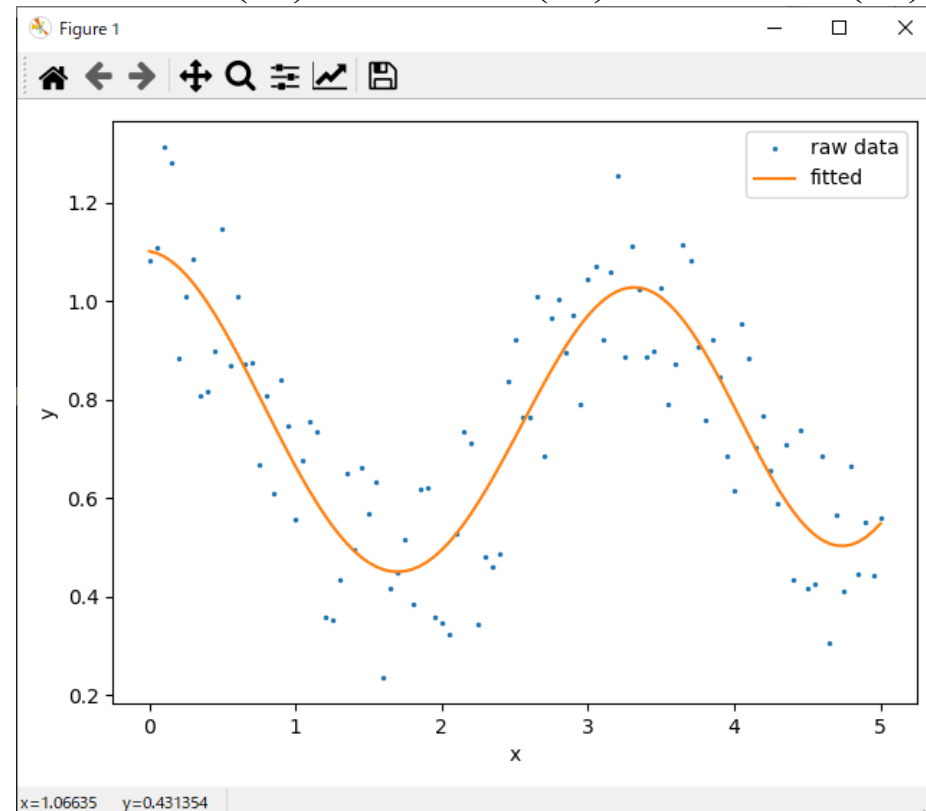
`python lsq-general.py 2`

$y = 0.740 + 0.000432 \sin(x)$



`python lsq-polynomial.py 6`

$y = 0.753 + 0.0064 \sin(x) + 0.00358 \cos(x) + 0.125 \sin(2x) + 0.303 \cos(2x) + 0.0119 \sin(3x)$



# Ex of ILSQ: Lattice spacing of triclinic lattice

(三斜晶結晶の面間隔)

$$d_{hkl}^{-2} = |\mathbf{G}_{hkl}|^2 = |h\mathbf{a}^* + k\mathbf{b}^* + l\mathbf{c}^*|^2$$

$$\frac{1}{d_{hkl}^2} = S_{11}h^2 + S_{22}k^2 + S_{33}l^2 + 2S_{12}hk + 2S_{23}kl + 2S_{31}lh$$

$$S_{11} = \mathbf{a}^* \cdot \mathbf{a}^* = b^2 c^2 \sin^2 \alpha / V^2$$

$$S_{22} = c^2 a^2 \sin^2 \beta / V^2$$

$$S_{33} = a^2 b^2 \sin^2 \gamma / V^2$$

$$S_{12} = \mathbf{a}^* \cdot \mathbf{b}^* = abc^2 (\cos \alpha \cos \beta - \cos \gamma) / V^2$$

$$S_{23} = a^2 bc (\cos \beta \cos \gamma - \cos \alpha) / V^2$$

$$S_{31} = ab^2 c (\cos \gamma \cos \alpha - \cos \beta) / V^2$$

$$V = abc \sqrt{1 - \cos^2 \alpha - \cos^2 \beta - \cos^2 \gamma + 2 \cos \alpha \cos \beta \cos \gamma}$$

The form of  $d_{hkl}^{-2}$  is a linear function with respect to  $S_{ij}$ .

1.  $S_{ij}$  is obtained by ILSQ
2.  $S_{ij} \Rightarrow$  Reciprocal lattice parameters ( $a^*, b^*, c^*, \alpha^*, \beta^*, \gamma^*$ )
3.  $\Rightarrow$  Lattice parameters ( $a, b, c, \alpha, \beta, \gamma$ )

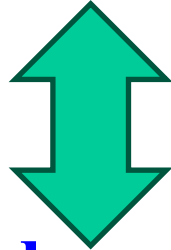
**How to solve equations?**

**Self-consistent method**

自己無撞着法

**Final unique solution is obtained just by one step calculation**

- **Linear least-squares method**
- **Up to 4<sup>th</sup> order polynomial eq.**



**Five or higher order polynomial,  
Transcendental equation (超越方程式)**

- **Difficult to have an analytical solution**
- **Even numerical analysis cannot give final solution by one-cycle calculation**  
**=> Iterative calculation (反復計算)**

# Simplest method: Self-consistent (SC) method

**A simple case: Solve  $g(x) = 0$**

**SC method is applicable by converting to  $x = g(x) + x = f(x)$**

*Note: not efficient nor stable for many cases*

**Simple procedure:**

**Initial value  $x_0$**

**1st iteration :  $x_1 = f(x_0)$**

**2nd iteration:  $x_2 = f(x_1) \dots$**

**Difficult to converge: Diverge, Oscillation**

(収束しにくい: 発散、振動)

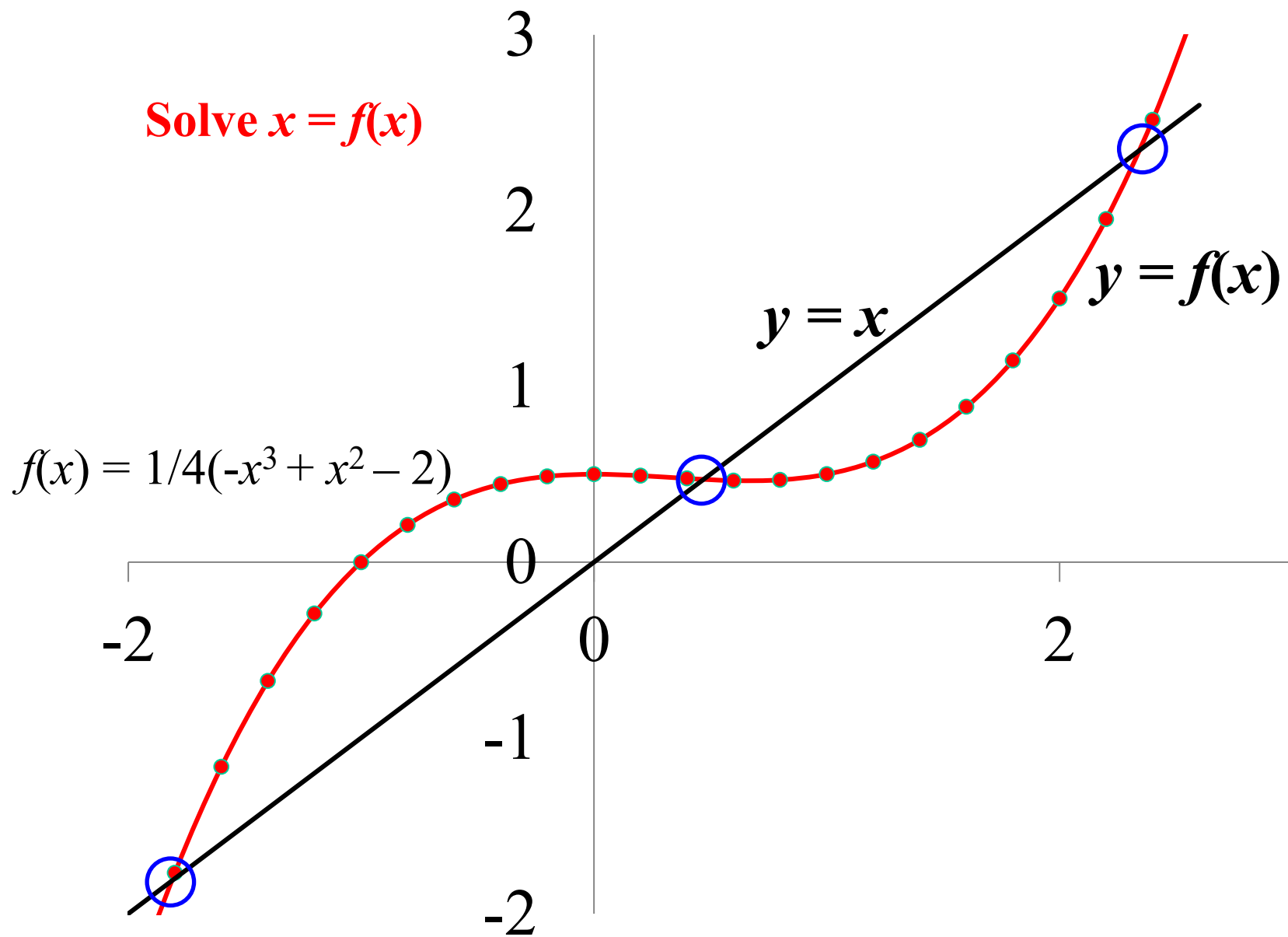
**Mixing factor (混合係数)  $k_{\text{mix}}$ : Stabilize convergence**

**Initial value  $x_0$**

**1st iteration :  $x_1 = f(x_0) \Rightarrow x_1' = (1 - k_{\text{mix}}) x_0 + k_{\text{mix}} x_1$**

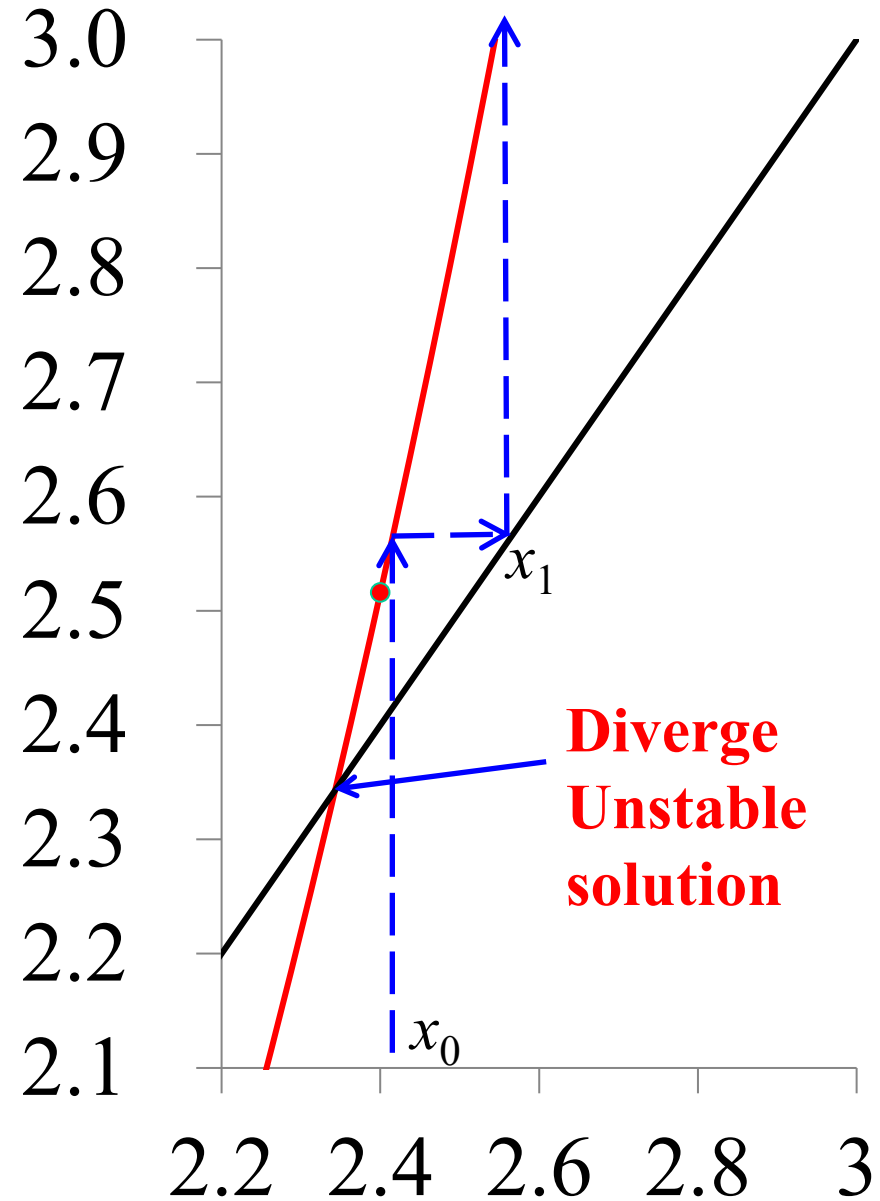
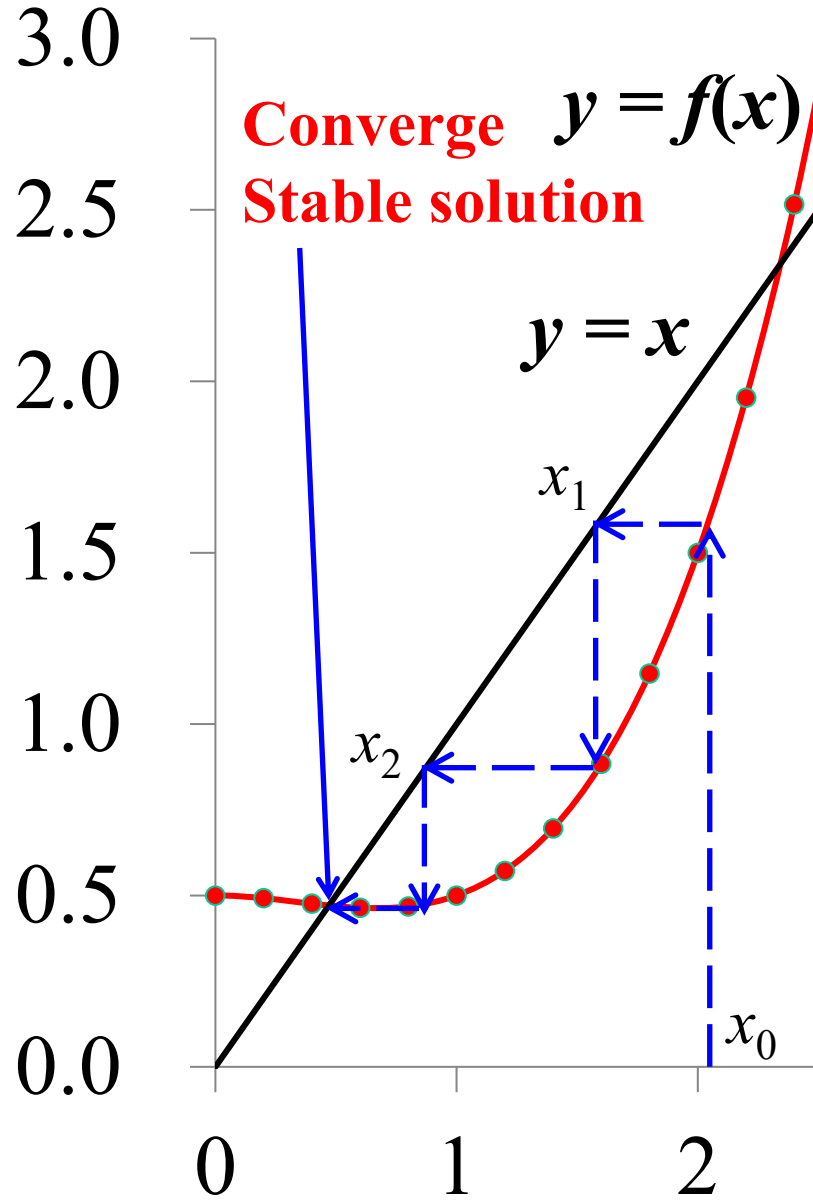
**2nd iteration:  $x_2 = f(x_1') \dots$**

# Illustrative explanation of SC





# SC: Convergence process



$f'(x) < 1$  must be satisfied for convergence

# Example of SC: Diode with series resistance

$$I = I_0 \left[ \exp \left( \frac{e}{nkT} (V - RI) \right) - 1 \right]$$

**Repeat**

$$I_i = I_0 \left[ \exp \left( \frac{e}{nkT} (V - RI_{i-1}) \right) - 1 \right]$$

**until  $\text{abs}(I_i - I_{i-1}) < \text{EPS}$  is achieved**

- E.g., initial voltages would be chosen as  $V/2$  for the diode and the R
- This SC is not so stable; mixing factor  $k$  should be adjusted

For sequential calculations of  $I - V$  characteristic, e.g.,  $V$  from 0.0 to 1.0, using a preconverged result for the initial value of the next  $V$  will enhance convergence.

例えば  $V$  を順次変えて  $I - V$  特性を計算するような場合、すでに収束した値を次の  $V$  における初期値として利用すると早く収束できる。

**SC-Diode.xlsx**

i	I	Ical	error	I0=	1.E-12	A
0	2	-1E-12	2	n=	1	
1	1.8	-1E-12	1.8	T=	300	K
2	1.62	-1E-12	1.62	R=	1	ohm
3	1.458	-1E-12	1.458	V=	1	
4	1.3122	-1E-12	1.3122			
5	1.18098	-1E-12	1.18098	k=	0.1	
6	1.062882	-9.1E-13	1.06288			
7	0.956594	4.31E-12	0.95659			
8	0.860934	2.09E-10	0.86093			
9	0.774841	5.77E-09	0.77484			
10	0.697357	1.14E-07	0.69736			
11	0.627621	1.66E-06	0.62762			
12	0.564859	1.86E-05	0.56484			
13	0.508375	0.000163	0.50821			
14	0.457554	0.00115	0.4564			
15	0.411914	0.006655	0.40526			
16	0.371388	0.031631	0.33976			
17	0.337412	0.116849	0.22056			
18	0.315356	0.272927	0.04243			
19	0.311113	0.321305	0.01019			
20	0.312132	0.308953	0.00318			
21	0.311814	0.312754	0.00094			
22	0.311908	0.311626	0.00028			
23	0.31188	0.311965	8.5E-05			
24	0.311888	0.311863	2.5E-05			
25	0.311886	0.311893	7.6E-06			
26	0.311887	0.311884	2.3E-06			

# First-principles calculation:

## Self-consistent field (SCF, 自己無撞着) calculation

- Hamiltonian of one-electron quantum equation includes wave functions

$$\left\{ -\frac{1}{2} \nabla_l^2 - \sum_m \frac{Z_m}{r_{lm}} + \sum_m \int \frac{\rho_m(\mathbf{r}_m)}{r_{lm}} d\mathbf{r}_m + V_{xl}(\mathbf{r}_l) \right\} \phi_l(\mathbf{r}_l) = \varepsilon_l \phi_l(\mathbf{r}_l)$$

- First-step calculation requires electron density guessed / assumed  $\rho_{\text{ini}}$ :  
e.g., by uniform density, sum of atomic electron density,,



- Electron density  $\rho_{\text{fin}}$  is calculated the solved wave functions, but  $\rho_{\text{fin}}$  would be different from  $\rho_{\text{ini}}$



$\rho_{\text{ini}}$  must be equal to  $\rho_{\text{fin}}$ , otherwise  
these loss physical meaning

- More appropriate  $\rho_{\text{new}}$  is guessed from  $\rho_{\text{fin}}$  and  $\rho_{\text{ini}}$ ,  
and repete the above calculations

$$\text{ex. : } \rho_{\text{new}} = \rho_{\text{ini}} + k_{\text{mix}}(\rho_{\text{fin}} + \rho_{\text{ini}})$$

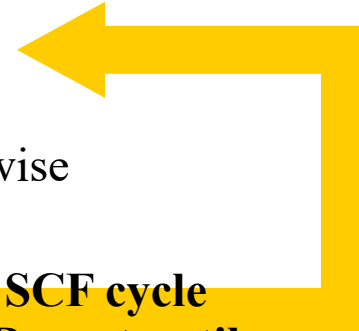
$k_{\text{mix}}$  : Mixing factor

A parameter to suppress divergence of the SCF calculation

close to 1 would be easily diverged, close to 0 causes slow convergence

**SCF cycle**

**Repeat until  $\rho_{\text{fin}} = \rho_{\text{ini}}$**



# Example: SCF/structure relaxation by VASP

```
tkamiya@csrv0:~/Work/LaCrAsO/SpinPolarized
ファイル(E) 編集(E) 表示(V) 端末(T) タブ(B) ヘルプ(H)
1 F= -.24922201E+03 E0= -.24922201E+03 d E =-.249222E+03 mag= 17.6753
curvature: 0.00 expect dE= 0.000E+00 dE for cont linesearch 0.000E+00
trial: gam= 0.00000 g(F)= 0.620E+00 g(S)= 0.305E-01 ort = 0.000E+00 (trialstep = 0.100E+01
)
search vector abs. value= 0.650E+00
bond charge predicted
      N      E      dE      d eps      ncg      rms      rms(c)
DAV: 1 -0.249256423264E+03 -0.24926E+03 -0.54781E+01 3528 0.200E+01 0.196E+00
DAV: 2 -0.249670978228E+03 -0.41455E+00 -0.52988E+00 4416 0.955E+00 0.161E+00
DAV: 3 -0.249672461360E+03 -0.14831E-02 -0.53814E-01 4640 0.336E+00 0.153E+00
DAV: 4 -0.249667045995E+03 0.54154E-02 -0.45192E-01 4632 0.183E+00 0.129E+00
DAV: 5 -0.249662986402E+03 0.40596E-02 -0.16171E-01 4664 0.134E+00 0.113E+00
DAV: 6 -0.249664501455E+03 -0.15151E-02 -0.86520E-02 4520 0.152E+00 0.943E-01
DAV: 7 -0.249658663938E+03 0.58375E-02 -0.36669E-02 4626 0.103E+00 0.315E-01
DAV: 8 -0.249657255947E+03 0.14080E-02 -0.11030E-02 4432 0.529E-01 0.406E-01
DAV: 9 -0.249656661683E+03 0.59426E-03 -0.64937E-03 3424 0.480E-01 0.219E-01
DAV: 10 -0.249654538004E+03 0.21237E-02 -0.11755E-03 2528 0.225E-01 0.151E-01
DAV: 11 -0.249654612437E+03 -0.74432E-04 -0.11566E-03 2520 0.213E-01
2 F= -.24965461E+03 E0= -.24965461E+03 d E =-.432599E+00 mag= 18.2912
trial-energy change: -0.432599 1 .order -0.416777 -0.650072 -0.183481
step: 1.3105(harm= 1.3932) dis= 0.06748 next Energy= -249.683568 (dE=-0.462E+00)
bond charge predicted
      N      E      dE      d eps      ncg      rms      rms(c)
DAV: 1 -0.249658788237E+03 -0.24966E+03 -0.53760E+00 3536 0.623E+00 0.599E-01
DAV: 2 -0.249698102900E+03 -0.39315E-01 -0.48908E-01 4528 0.303E+00 0.671E-01
```

# Typical iteration of SC calculation

Find the solution of  $f(x, \rho(x)) = 0$ :

Case this is easily done if  $\rho(x)$  is provided

1. Assume  $\rho(x)$  and solve  $f(x, \rho(x)) = 0$  to get approximate  $x_i$
2. Calculate  $\rho(x_i)$  with the obtained  $x_i$ , solve  $f(x, \rho(x_i)) = 0$ , and get improved approximation  $x_{i+1}$
3. Repeat 1 – 2 so as to decrease  $|\rho(x_{i+1}) - \rho(x_i)|$ ,  $|x_{i+1} - x_i|$  to required accuracy

**Self-consistent approach** (自己無動着計算)

May be diverged if the obtained  $x_i'$  is used for  $x_{i+1}$

=> **Stabilize convergece using mixing factor** (混合係数)  $k_{\text{mix}}$

Initial  $x_0$

First iteration:  $x_1 = f(x_0)$  =>  $x_1' = (1 - k_{\text{mix}}) x_0 + k_{\text{mix}} x_1$

Next iteration:  $x_2 = f(x_1')$  ....

# Problems of SC calculations

- **Some solutions would not be obtained** (収束しない解があり得る)  
 $f'(x) < 1$  must be satisfied at the solution to obtain the solution of  $x = f(x)$   
=> Conversion of the equation may help, but not always
- **Convergence is not stable**  
mixing factor may improve

**For many cases, use another method such as Newton method**

- **Cases SC method is effective**  
Initial values close to the solution  
Effect of SC parameters is small to the equation  
(自己無撞着変数の方程式への影響が小さい)  
SC parameters have good convergence  
(自己無撞着変数の収束特性が良く、予測できる場合)

**How to solve equations?**

**More sophisticated algorithms**

# Newton-Raphson method

**Solve  $f(x) = 0$**

Start from initial guess:  $x_0$

$x_0 + dx$  is supposed to be a solution

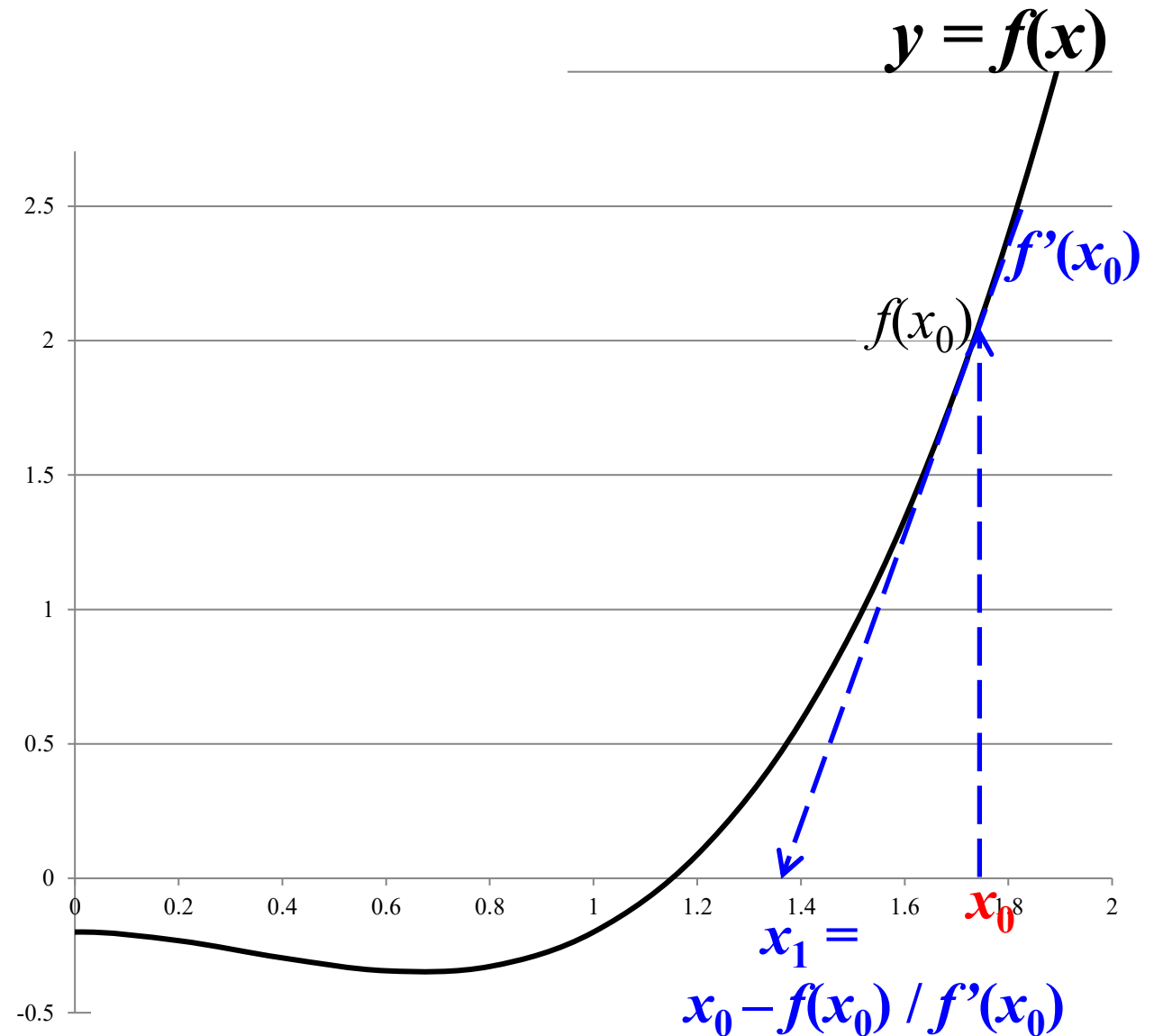
$$f(x_0 + dx) = f(x_0) + dx f'(x_0) \sim 0$$

$$\Rightarrow x_1 = x_0 + dx = x_0 - f(x_0) / f'(x_0)$$

Stabilize convergence:

$$x_{k+1} = x_k - f(x_k) / f'(x_k) / (1 + \lambda)$$

$\lambda$ : Dumping Factor





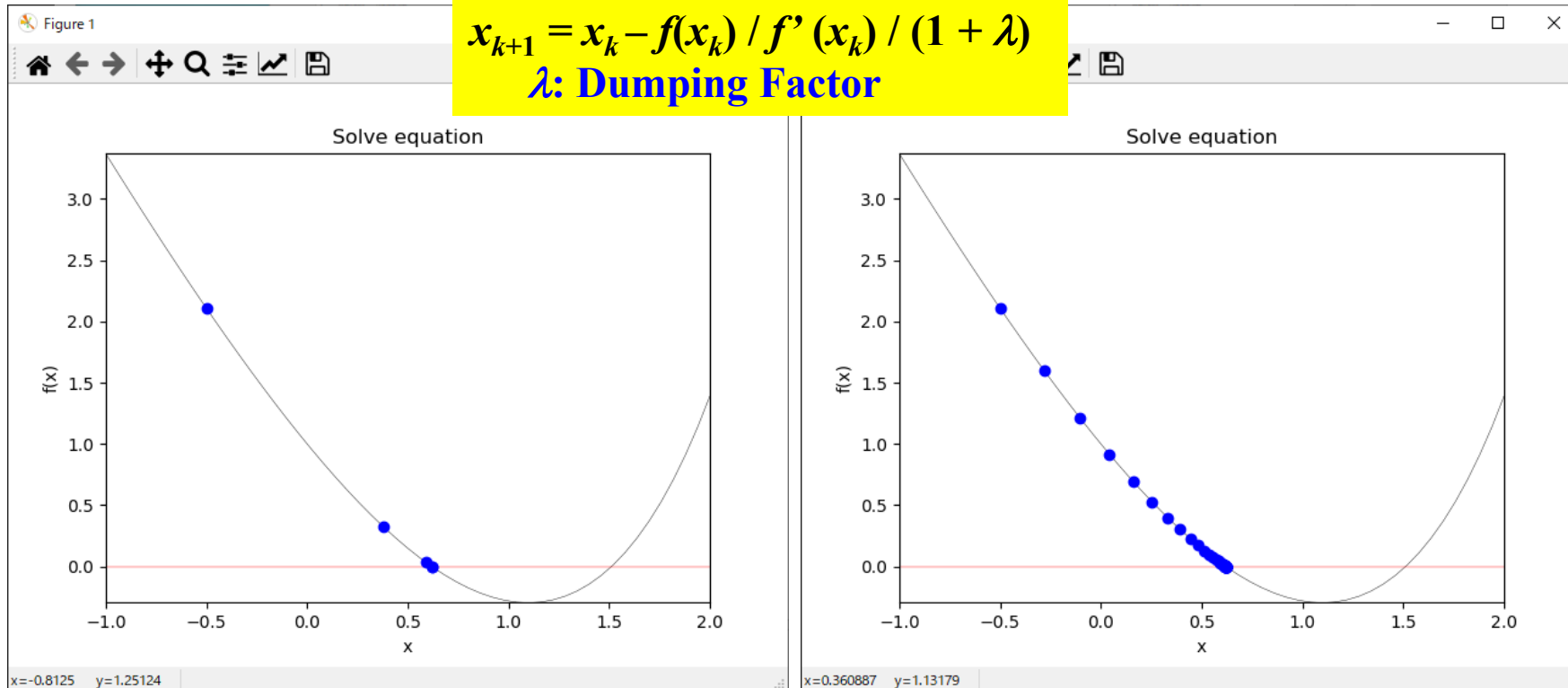
# Program: equation-newton-Raphson.py

Usage: python equation-newton-raphson.py x0 dump t<sub>sleep</sub>

$$f(x) = \exp(x) - 3.0x$$

python equation-newton-raphson.py -0.5 0

python equation-newton-raphson.py -0.5 3



# Effect of dumping factor (収束過程の比較)

$$f(x) = \exp(x) - 3x = 0 \text{ (initial } x = 0) \quad \text{Exact } 0.619061$$

Newton-Raphson (Dumping factor = 0)

Iter.	$x$	$ x_i - x_{i-1} $
1	0.5	
2	0.610059654958962	0.110059654958962
3	0.61899677974154	0.00893712478257794
4	0.619061283355313	6.4503613773092e-005
5	0.619061286735945	3.38063244722622e-009
6	0.619061286735945	-1.94296000199483e-016

Newton-Raphson (Dumping factor = 0.1)

1	0.476190476190476	
2	0.597901649246081	0.121711173055605
3	0.617090542717403	0.0191888934713221
4	0.618900291486661	0.00180974876925825
5	0.619048316423879	0.000148024937217564
6	0.619060243007723	1.19265838440254e-005
7	0.619061202754359	9.59746635487409e-007
8	0.619061279978579	7.72242198569211e-008
9	0.619061286192231	6.21365241490959e-009
10	0.619061286692197	4.99965669237101e-010
11	0.619061286732425	4.0228535713285e-011

Newton-Raphson (Dumping factor = 1.0)

1	0.333333333333333	
2	0.485235618882813	0.15190228554948
3	0.556317491275292	0.0710818723924794
4	0.589692022113926	0.0333745308386341
5	0.605333177012923	0.0156411548989961
6	0.612649553494255	0.00731637648133212
7	0.616067929129785	0.00341837563553035
8	0.617664103982484	0.00159617485269905
9	0.618409199563502	0.00074509558101794
10	0.618756961315507	0.000347761752005284
11	0.618919262817103	0.000162301501596124

# Newton-Raphson method

Solve  $f(x) = 0$

Start from initial guess:  $x_0$

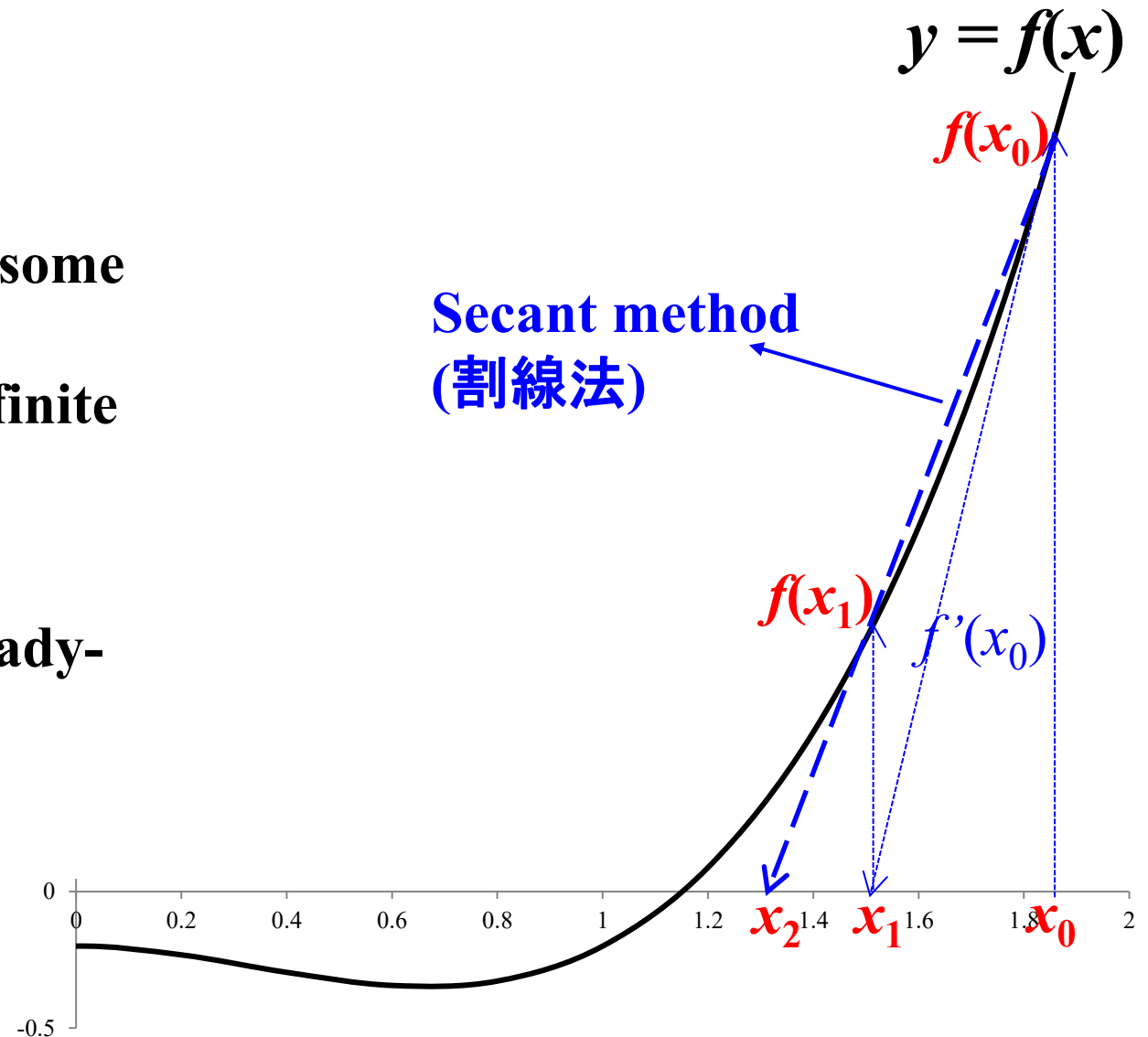
repeat:  $x_1 = x_0 + dx = x_0 - f(x_0) / f'(x_0)$

- $f'(x_0)$  can be analytical calculated for some cases
- $f'(x_0)$  is more often approximated by finite difference

$$\frac{f(x_0+h) - f(x_0-h)}{2h}$$

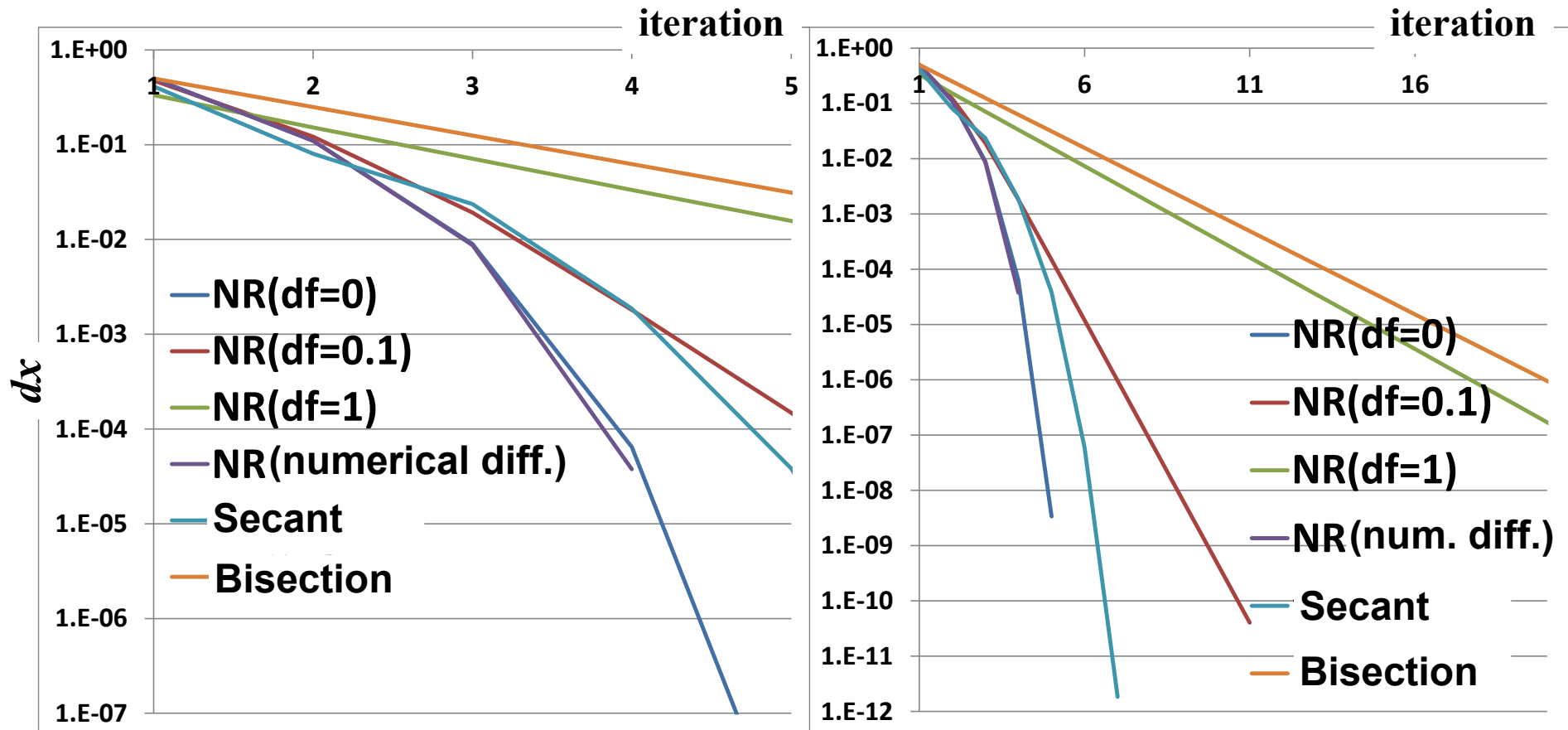
- $f'(x_0)$  can be approximated using already-calculated values

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad \text{Secant method (割線法)}$$



# Effect of dumping factor: Convergence process

$f(x) = \exp(x) - 3x = 0$  (initial  $x = 0$ )      Exact 0.619061



$$x_{k+1} = x_k - f(x_k) / (f'(x_k) + \lambda)$$
  
 $\lambda$ : Dumping Factor

NR: Newton-Raphson method  
df: Dumping Factor

# APPENDIX

related to linear least-squares methods

# 最尤推定法

尤度関数とは:

事象  $(x_k)$  が起こる確率を、既知のパラメータ  $(a_k)$  の確率密度関数

$$P(X = x_i | a_k) = \prod_i \left\{ \frac{1}{\sqrt{2\pi}\sigma_i} \exp \left[ -\frac{\varepsilon_i(a_k)^2}{2\sigma_i^2} \right] \right\} = \prod_i \left( \frac{1}{\sqrt{2\pi}\sigma_i} \right) \cdot \exp \left[ -\sum_i \frac{\varepsilon_i(a_k)^2}{2\sigma_i^2} \right]$$

などとする、逆に  $X = (x_i)$  がわかっていると、

パラメータ  $(a_k)$  がどれだけ尤もらしいか (尤度) を表す確率密度関数とみなし、

上記の確率密度関数を変数  $(a_k)$  の関数として

尤度関数  $P(a_i) = P(x_i | a_i)$  という。

## 最尤推定法

誤差  $\varepsilon_i = f(x_i, a_i) - y_i$  が分散  $\sigma_i$  の正規分布に従うとする。データ  $(x_i, y_i)$  に対するパラメータ  $(a_i)$  の尤度関数は

$$P(a_i) = \prod_i \left( \frac{1}{\sqrt{2\pi}\sigma_i} \right) \cdot \exp \left[ -\sum_i \frac{\varepsilon_i^2}{2\sigma_i^2} \right]$$

尤度を最大化するパラメータを求めるのが「最尤推定法」。

$$\max P(a_i) = \max \ln P(a_i) = \min \sum_i \frac{\varepsilon_i^2}{\sigma_i^2}: \text{最小二乗法に一致する}$$

# FET特性の解析: 飽和領域

$$I_{DS} = \frac{W}{L} \mu C_{OX} \left[ (V_{GS} - V_{th}) V_{DS} - \frac{V_{DS}^2}{2} \right]$$

$$V_{DS} > V_p = V_{GS} - V_{th}$$

$$I_{DS} = \frac{W}{2L} \mu C_{OX} (V_{GS} - V_{th})^2$$

$$I_{DS}^{1/2} = \sqrt{\frac{W}{2L} \mu C_{OX} (V_{GS} - V_{th})}$$

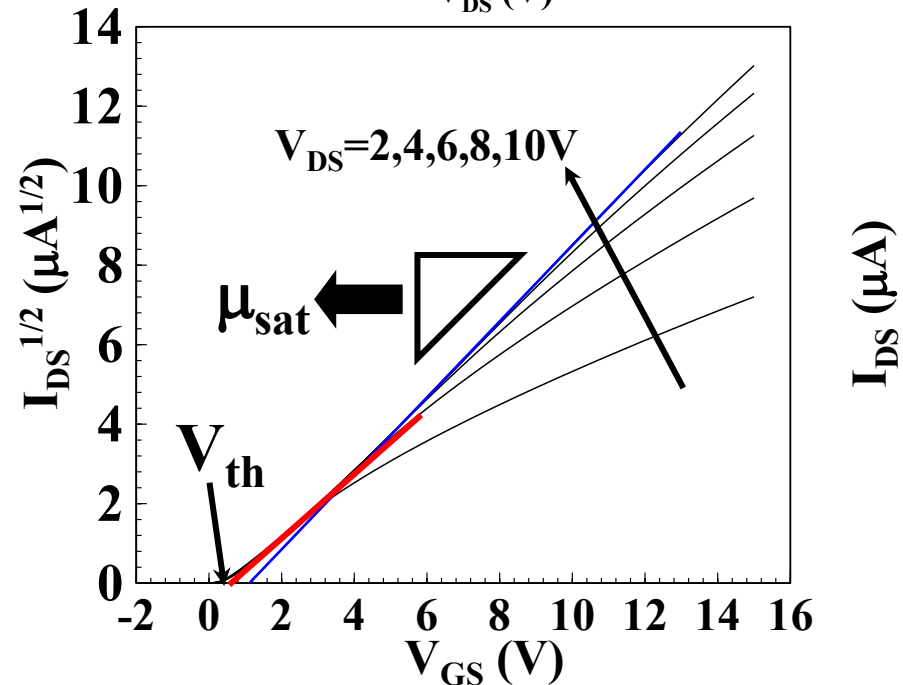
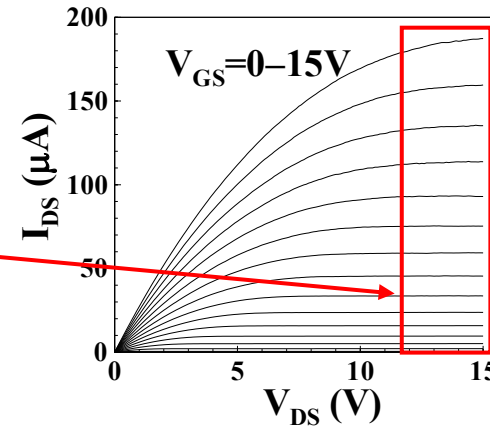
$I_{DS}^{1/2}$  vs.  $V_{GS}$ をプロット

$V_{GS}$ 軸切片:  $V_{th}$

傾き: 飽和移動度

Saturation mobility,  $\mu_{sat}$

a-IGZO TFT



# ゲート容量 $C_{OX}$ の計算

ゲート絶縁体: アモルファス  $\text{SiO}_2$

誘電率:  $11.9\epsilon_0$  ( $\epsilon_0 = 8.854418782\text{e-}12 \text{ C}^2\text{N}^{-1}\text{m}^{-2}$ )

厚さ 100 nm

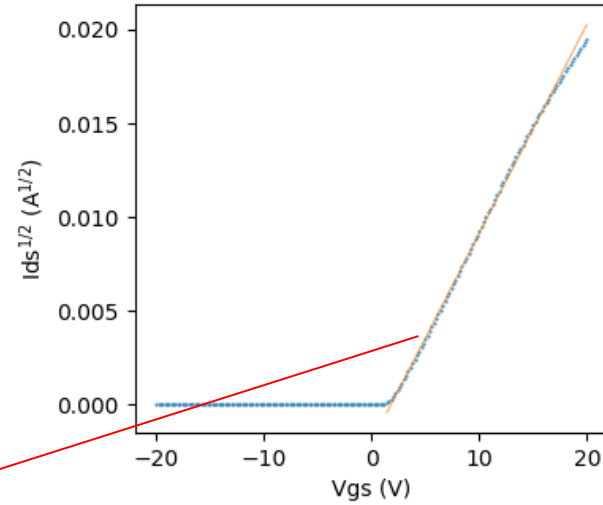
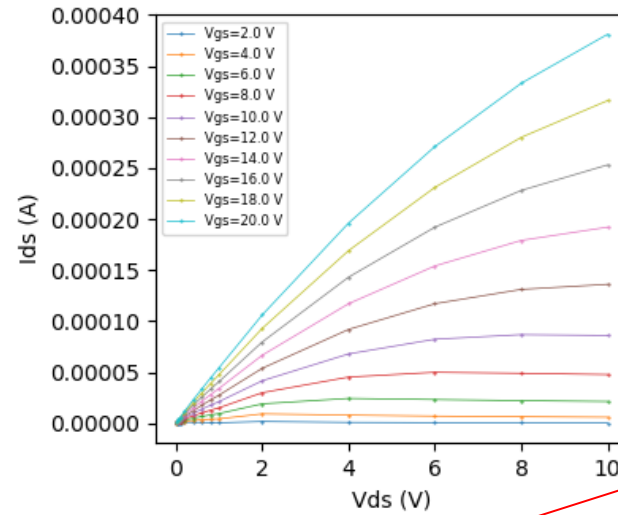
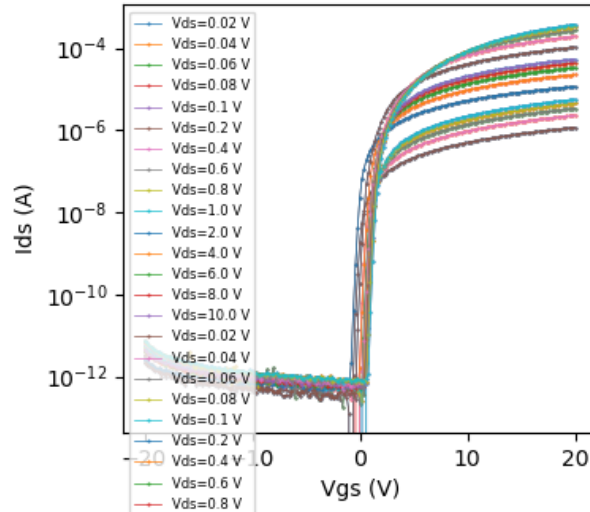
1 m<sup>2</sup> 当たり静電容量

$$C_{OX} = 11.9\epsilon_0 / 100\text{e-}9 [\text{m}] = 0.00105 \text{ F/m}^2$$



# プログラム

TFTiv.py



直線に見える 1.9 ~ 10.0 V で、numpy.polyfit() で一次多項式にフィッティング

```
ai = np.polyfit(xfit, yfit, 1)
```

```
# y = ai[1] + ai[0]x
```

```
Vth = -ai[1] / ai[0] = 1.794 V
```

```
 $\frac{dI_{gs}^{1/2}}{dV_{gs}} = ai[0] = 0.001114 \text{ A}^{1/2}/\text{V}$ 
```

$$I_{DS}^{1/2} = \sqrt{\frac{W}{2L} \mu C_{OX} (V_{GS} - V_{th})}$$

$$\mu_{sat} = \frac{(dI_{gs}^{1/2}/dV_{gs})^2}{\frac{W}{2L} C_{OX}} = 0.000392 \text{ m}^2/\text{Vs} = 3.92 \text{ cm}^2/\text{Vs}$$

# プログラム (抜粋)

TFTiv.py

```
import re    # 正規表現モジュールを読み込む
```

```
#=====
```

```
# parameters
```

```
#=====
```

```
infile = 'TransferCurve.csv'
```

```
dg = 100e-9 #m
```

```
erg = 11.9
```

```
W = 300.0e-6 #m
```

```
L = 50.0e-6
```

```
#  $I_{ds}^{1/2}$ - $V_{gs}$ プロットをする $V_{ds}$ 
```

```
 $V_{ds0} = 10.0$ 
```

```
# 余計な文字が含まれている文字列から、  
# 浮動小数点に変換できる最初の文字列を切り出し、  
# 浮動小数点に変換して返す
```

```
def pfloat(str, defval = None):
```

```
# 文字列から、浮動小数点に使える文字が連続している部分を切り  
出す
```

```
    m = re.search(r'([+¥-eE¥d¥.]+)', str)
```

```
# 一致した文字列を取得
```

```
    valstr = m.group()
```

```
    try:
```

```
        return float(valstr)
```

```
    except:
```

```
        return defval
```

```
def read_csv(fname):
```

```
    print("")
```

```
    with open(fname) as f:
```

```
        fin = csv.reader(f)
```

```
        labels = next(fin)
```

```
        xlabel = labels[0]
```

```
# label行が 空文字 の場合、データとしては読み込まない
```

```
        ylabels = []
```

```
        for i in range(1, len(labels)):
```

```
            if labels[i] == ":
```

```
                break
```

```
            ylabels.append(labels[i])
```

```
        ny = len(ylabels)
```

```
        x    = []
```

```
        ylist = []
```

```
        for i in range(ny):
```

```
            ylist.append([])
```

```
        for row in fin:
```

```
            x.append(pfloat(row[0]))
```

```
            for i in range(1, ny+1):
```

```
                v = pfloat(row[i])
```

```
                if v is not None:
```

```
                    ylist[i-1].append(v)
```

```
                else:
```

```
                    ylist[i-1].append(None)
```

```
    return xlabel, ylabels, x, ylist
```

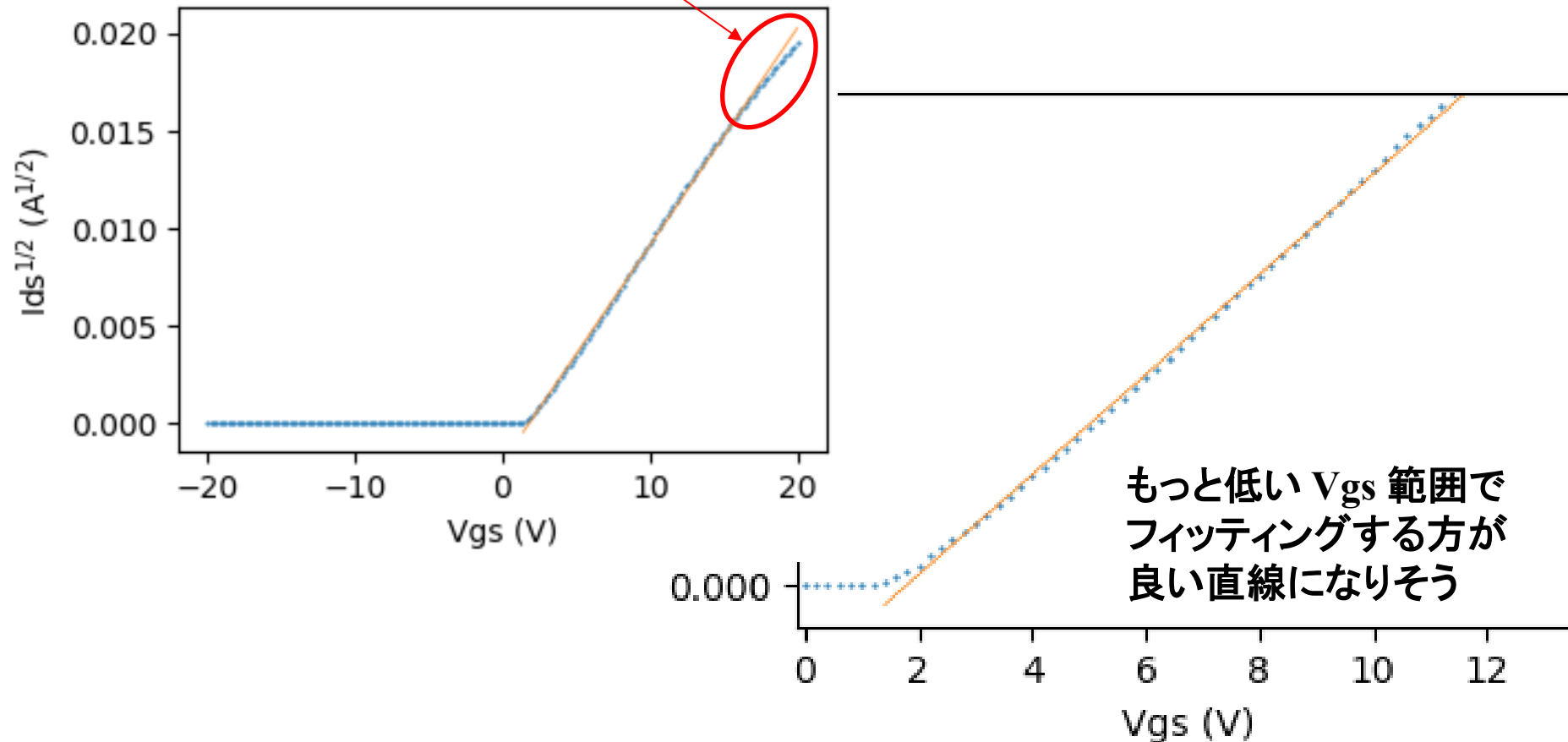
# フィッティング範囲の妥当性

TFTiv\_errorcheck.py

フィッティング範囲 2.0 ~ 10.0V

高  $V_{gs}$  では  $V_{ds} < V_p = V_{gs} - V_{th}$  となり、  
直線から外れる

最小二乗に入れてはいけない



# 線形最小二乗法 $y = a + bx$ の誤差

酒井英行訳、M.C.Barford著、実験精度と誤差、丸善

$$f(x_i) = a + bx_i$$

$$\varepsilon_i = f(x_i) - (a + bx_i) \quad \text{目的関数 } S = \sum \varepsilon_i^2 \text{ を最小化}$$

$$\begin{pmatrix} n & s_x \\ s_x & s_{xx} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} s_y \\ s_{xy} \end{pmatrix} \quad s_x = \sum x_i, s_y = \sum y_i, s_{xx} = \sum x_i^2, s_{xy} = \sum x_i y_i$$

$\langle A \rangle$  は  $A$  の平均

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\Delta} \begin{pmatrix} s_{xx} & -s_x \\ -s_x & n \end{pmatrix} \begin{pmatrix} s_y \\ s_{xy} \end{pmatrix} \quad \Delta = ns_{xx} - s_x^2 = n \sum (x_i - \langle x \rangle)^2$$

$$b = \frac{ns_{xy} - s_x s_y}{\Delta} = \frac{\sum (x_i - \langle x \rangle)(y_i - \langle y \rangle)}{\sum (x_i - \langle x \rangle)^2} \quad a = \frac{s_{xx} s_y - s_x s_{xy}}{\Delta} = \langle y \rangle - b \langle x \rangle$$

$$\sigma_y^2 = \langle y^2 \rangle - \langle y \rangle^2 - \frac{(\langle xy \rangle - \langle x \rangle \langle y \rangle)^2}{\langle x^2 \rangle - \langle x \rangle^2} = \frac{1}{n^2} \left[ ns_{yy} - s_y^2 - \frac{(ns_{xy} - s_x s_y)^2}{ns_{xx} - s_x^2} \right]$$

$$\text{標準誤差: } S_a = \frac{\sigma_y \sqrt{\langle x^2 \rangle}}{\sqrt{(n-2)(\langle x^2 \rangle - \langle x \rangle^2)}}$$

$$S_b = \frac{\sigma_y}{\sqrt{(n-2)(\langle x^2 \rangle - \langle x \rangle^2)}}$$

$$\text{相関係数: } r = \frac{s_{xy}}{\sqrt{s_{xx} s_{yy}}}$$

# 誤差の計算

## 誤差の伝播則

変数  $a, b$  の標準誤差  $\sigma_a, \sigma_b$  が既知の場合、 $f(a, b)$  の誤差は

$$\delta f(a, b) = \left( \frac{\partial f}{\partial a} \right)_b \delta a + \left( \frac{\partial f}{\partial b} \right)_a \delta b$$

$\delta a, \delta b$  が正規分布に従う場合、 $\delta f(a, b)$  の標準偏差  $\sigma_f$  は

$$\sigma_f = \sqrt{\left[ \left( \frac{\partial f}{\partial a} \right)_b \sigma_a \right]^2 + \left[ \left( \frac{\partial f}{\partial b} \right)_a \sigma_b \right]^2}$$

最小自乗法で  $y = a + bx$  の標準誤差  $\sigma_a, \sigma_b$  が得られたら・・・

$$\cdot V_{th}(a, b) = -a/b$$

$$\delta V_{th} = -\delta a/b + a\delta b/b^2$$

$$\sigma_{V_{th}} = \sqrt{\left[ \left( \frac{1}{b} \right) \sigma_a \right]^2 + \left[ \left( \frac{a}{b^2} \right) \sigma_b \right]^2}$$

$$\delta \log V_{th} = \frac{\delta V_{th}}{V_{th}} = \delta a/a - \delta b/b$$

$$\sigma_{V_{th}} = V_{th} \sqrt{\left[ \left( \frac{1}{a} \right) \sigma_a \right]^2 + \left[ \left( \frac{1}{b} \right) \sigma_b \right]^2}$$

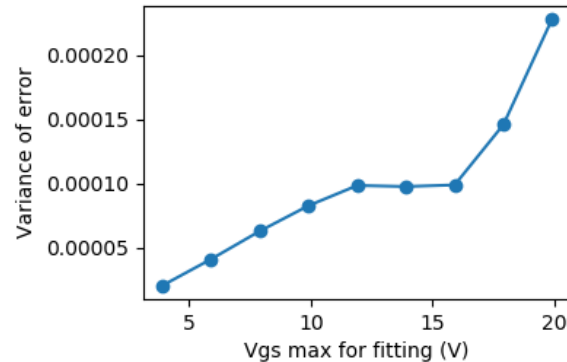
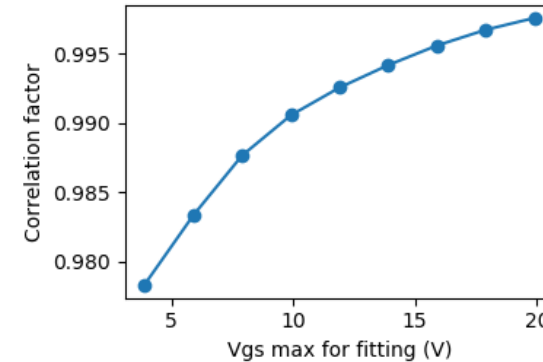
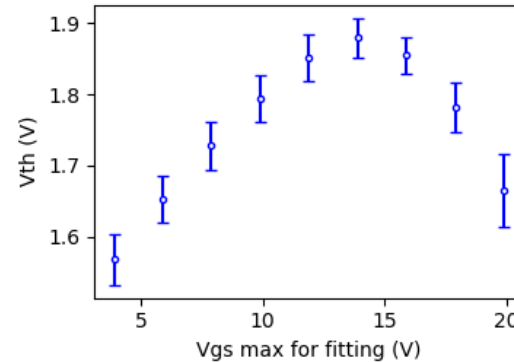
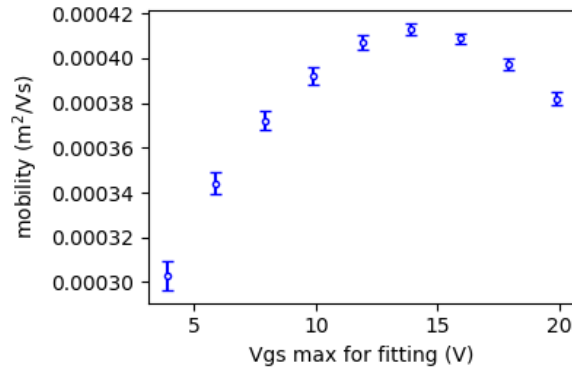
$$\cdot \mu(a, b) = b^2 / \sqrt{\frac{C_{ox}}{2L}}$$

$$\delta \log \mu = \frac{\delta \mu}{\mu} = \delta [\text{const} + 2 \ln b] = 2 \frac{\delta b}{b}$$

$$\sigma_\mu = 2\mu\sigma_b/b$$

# 最小二乗法の誤差と相関係数

TFTiv\_errorcheck.py



- フィッティング範囲を広げると**相関係数**は 1.0 に近づく  
x 範囲が広いと相関係数は大きくなりやすい。  
**必ずしも良い直線範囲の指標にはなっていない**
- **誤差  $\varepsilon_i$  の標準偏差  $\sigma_{\varepsilon_i}$**  は  $V_{gs} > 16V$  で明確に大きくなる:  
 **$V_{gs} > 16V$  のデータを入れてはいけない**
- 移動度、 $V_{th}$  は 14 V までのデータを入れると最大値をとる  
**意味はない。フィッティング範囲の妥当性の検証が重要**
- **標準誤差** は  $1\sigma$  で表示していることに注意。  
 **$3\sigma$  を見込むのが普通**

# プログラム (抜粋)

TFTiv\_errorcheck.py

# 誤差、相関係数計算付き 一次多項式線形最小二乗

```
def lsq1(x, y, iPrint = 0):
```

```
    n = len(x)
```

# 統計量の計算

# si: 変数 i の和                      avi: i の平均

# sij: 変数 i と j の積の和        sij: i\*j の平均

```
    sx = sum(x)
```

```
    avx = sx / n
```

```
    sy = sum(y)
```

```
    avy = sy / n
```

```
    sxx = sum([x[i] * x[i] for i in range(n)])
```

```
    avxx = sxx / n
```

```
    sxy = sum([x[i] * y[i] for i in range(n)])
```

```
    avxy = sxy / n
```

```
    syy = sum([y[i] * y[i] for i in range(n)])
```

```
    avyy = syy / n
```

```
    delta = n * sxx - sx * sx
```

#  $y = a + bx$

```
    b = (n * sxy - sx * sy) / delta
```

```
    a = avy - b * avx
```

# 残差の二乗和  $\sum \varepsilon_i^2$ 、誤差の標準偏差  $\sigma(\varepsilon_i)$

```
    sum_ei2 = sum([pow(y[i] - a - b * x[i], 2) for i in range(n)])
```

```
    sigma_ei = sqrt(sum_ei2 / (n - 1))
```

```
    sigma_y2 = avyy - avy * avy - pow(avxy - avx * avy, 2) / (avxx - avx * avx)
```

```
    sigma_y = sqrt(sigma_y2)
```

# パラメータ a, b の標準誤差と相関係数

```
    Sa = sigma_y * sqrt(avxx) / sqrt((n - 2) * (avxx - avx * avx))
```

```
    Sb = sigma_y / sqrt((n - 2) * (avxx - avx * avx))
```

```
    r = sxy / sqrt(sxx * syy)
```

# 戻り値が多いので、統計量、標準誤差、相関係数は

# 辞書変数 (ハッシュ、連想配列) で返す

# 辞書変数の値は、{key:val}

```
res = {'sx': sx, 'sy': sy, 'sxx': sxx, 'sxy': sxy, 'syy': syy,
```

```
      'Sa': Sa, 'Sb': Sb, 'r': r, 'sigma_ei': sigma_ei, 'residual': sum_ei2}
```

```
return a, b, res
```

```
def main():
```

# 最小二乗を実行

```
    ai = lsq1(xfit, yfit, 0)
```

# 辞書変数は ai[2] に入る

```
    res = ai[2]
```

# a, b の標準誤差、相関係数

# 辞書変数の要素は 変数名[key]で受け取る

```
    Sa = res['Sa']
```

```
    Sb = res['Sb']
```

```
    r = res['r']
```

```
    Vth = -ai[0] / ai[1]
```

```
    grad = ai[1]
```

```
    mu = grad * grad / (W * Cox / 2.0 / L)
```

# 誤差伝播則から Vth と  $\mu$  の標準誤差を計算

```
    SVth = sqrt(pow(Sa / ai[1], 2) + pow(Sb * ai[0] / ai[1] / ai[1], 2))
```

```
    Smu = 2.0 * mu * Sb / ai[1]
```

# エラーバー付きグラフのプロット

```
ax4.errorbar(xecheck, ymu, yerr = ySmu,
```

```
            capsize = 3.0, fmt = 'o', markersize = 3.0, ecolor = 'b',
```

```
            markeredgecolor = 'b', color = 'w')
```