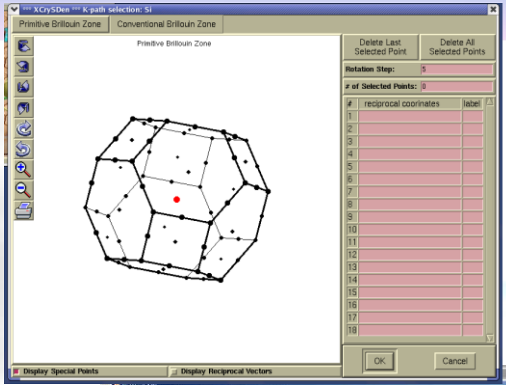
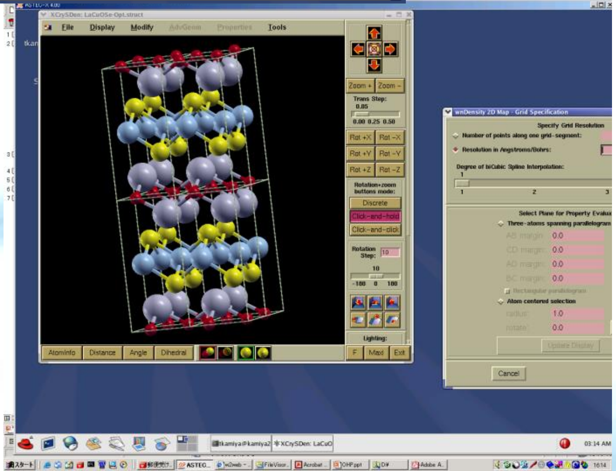
Computational Materials Science
計算材料学特論

Toshio Kamiya
神谷利夫


*** XCrySDen *** K-path selection: Si
Primitive Brillouin Zone
Conventional Brillouin Zone
Primitive Brillouin Zone
Delete Last Selected Point
Delete All Selected Points
reciprocal coorinates
label
OK
Cancel
Display Special Points
Display Reciprocal Vectors

Rotation Step: 5

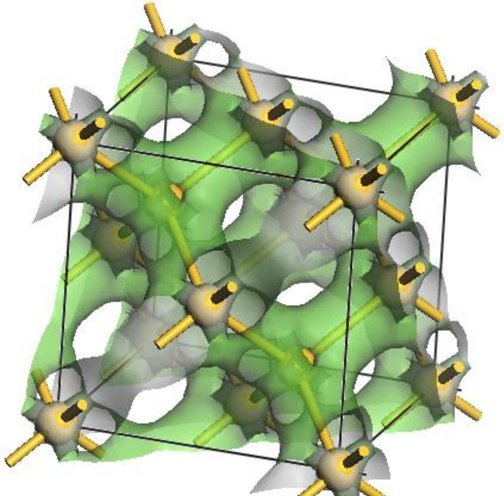
# of Selected Points: 0

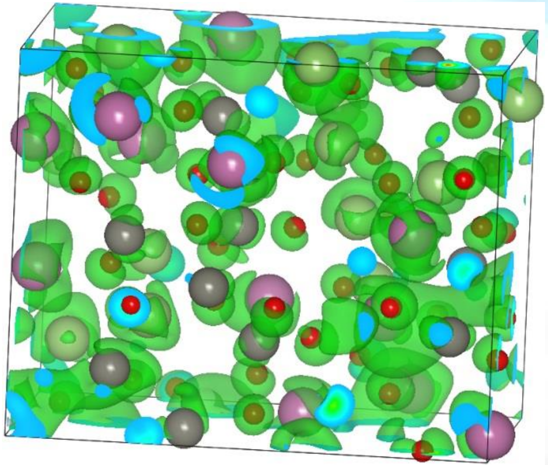


Trans Step: 0.05

Rotation+zoom buttons mode: Click-and-hold

Rotation Step: 10





Class Schedule

Lecture materials (Kamiya’s part): http://conf.msl.titech.ac.jp/Lecture/

http://conf.msl.titech.ac.jp/Lecture/ComputationalMaterialsScience/index-numericalanalysis.html

#01 June 13 (Tue)	Kamiya (Fundamental of computer, Sources of errors (コンピュータの基礎、誤差))
#02 June 16 (Fri)	Kamiya (Numerical differentiation/integration (数値微分/積分)
#03 June 20 (Tue)	Kamiya (Differential equation (微分方程式), Molecular dynamics (分子動力学法))
#04 June 23 (Fri)	Kamiya (Interpolation (補間), Smoothing (平滑化))
#05 June 27 (Tue)	Kamiya (Linear least-squares method (線形最小二乗法), Optimization (最適化), Numerical solutions of equations (方程式の数値解法))
#06 June 30 (Fri)	Kamiya (Nonlinear optimization (非線形最適化))
#07 July 4 (Tue)	Kamiya (Fourier transformation (フーリエ変換), Matrix, Applications)
#08 July 7 (Fri)	Sasagawa (Review of quantum theory 1: 量子論おさらい1)
#09 July 11 (Tue)	Sasagawa (Review of quantum theory 2: 量子論おさらい2)
#10 July 14 (Fri)	Sasagawa (First principles calculations: basics 1 第一原理計算:基礎1)
#11 July 18 (Tue)	Sasagawa (First principles calculations: basics 2 第一原理計算:基礎2)
#12 July 21 (Fri)	Sasagawa (First principles calc.: applications 1 第一原理計算:応用1)
#13 July 25 (Tue)	Sasagawa (First principles calc.: applications 2 第一原理計算:応用2)
#14 July 28 (Fri)	Sasagawa (Classical and Quantum Computers 古典および量子コンピュータ)

English textbooks

Search by ‘numerical analysis’, ‘numerical simulation’, ‘数値解析’ etc.

1. ***Introduction to Applied Numerical Analysis***
   Richard W. Hamming
   Dover publications, inc., New York (1989)
   ~340 pages

2. ***A First Course in Numerical Analysis***
   Anthony Ralston and Philip Rabinowitz
   Dover publications, inc., New York (1978)
   ~600 pages

For practical programming: Numerical Recipes series

1. **Numerical Recipes in C**
2. **Numerical Recipes Example Book (FORTRAN)**
3. **Numerical Recipes Source Code**
   Second Edition: C, Fortran77, Fortran 90
   Third Edition: C++

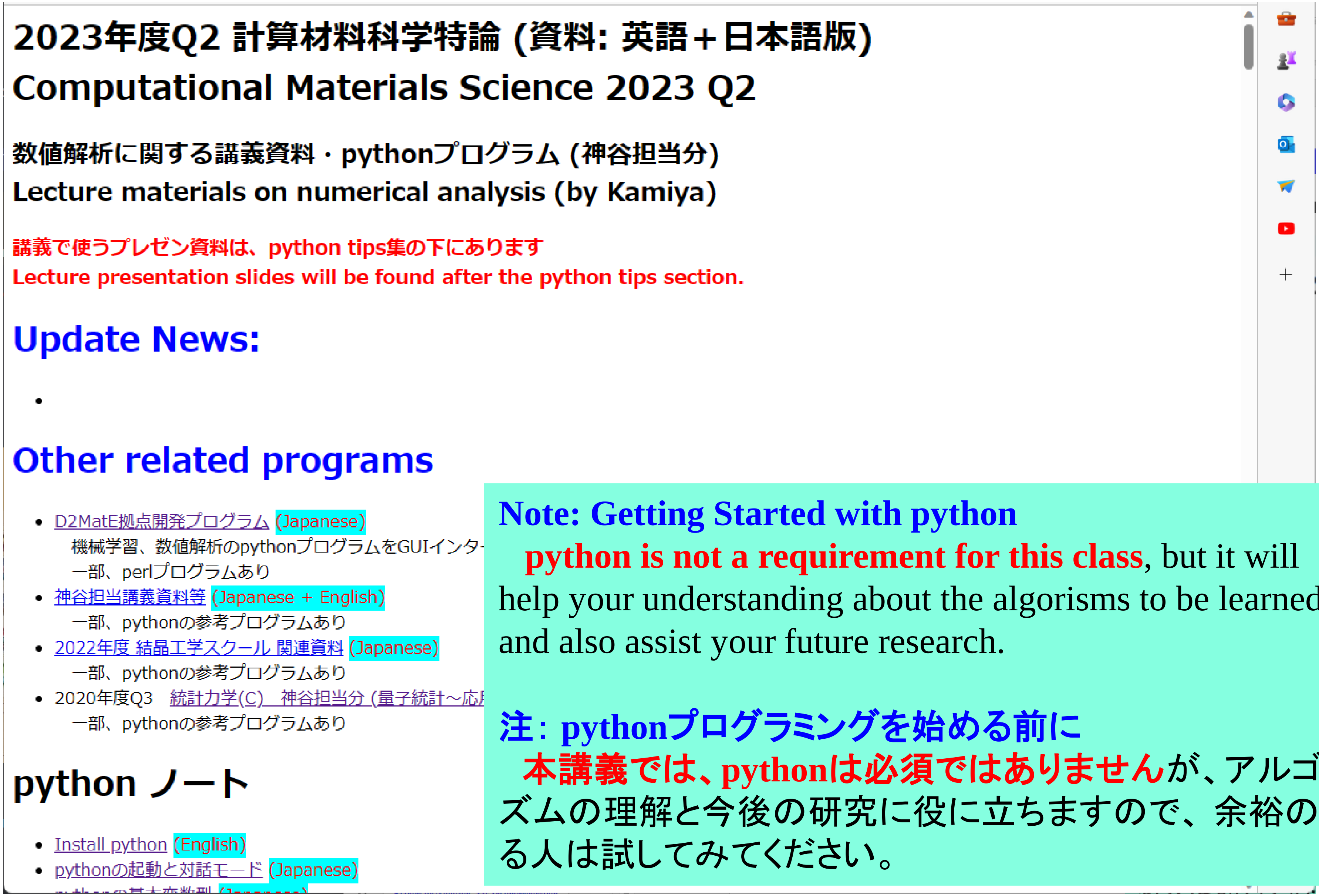
Evaluation (Kamiya)

- Small quiz
  Not evaluate correctness of the answers
  but consider how you answered them

- Term-end paper
  Problems will be given at the end of Q2
  from T2SCHOLAR

Numerical analysis web

http://conf.msl.titech.ac.jp/Lecture/ComputationalMaterialsScience/index-numericalanalysis.html



2023年度Q2 計算材料科学特論 (資料: 英語＋日本語版)

Computational Materials Science 2023 Q2

数値解析に関する講義資料・pythonプログラム (神谷担当分)

Lecture materials on numerical analysis (by Kamiya)

講義で使うプレゼン資料は、python tips集の下にあります

Lecture presentation slides will be found after the python tips section.

Update News:

-

Other related programs

- D2MatE拠点開発プログラム (Japanese)
  機械学習、数値解析のpythonプログラムをGUIインター
  一部、perlプログラムあり
- 神谷担当講義資料等 (Japanese + English)
  一部、pythonの参考プログラムあり
- 2022年度 結晶工学スクール 関連資料 (Japanese)
  一部、pythonの参考プログラムあり
- 2020年度Q3　統計力学(C)　神谷担当分 (量子統計～応
  一部、pythonの参考プログラムあり

python ノート

- Install python (English)
- pythonの起動と対話モード (Japanese)
- pythonの基本変数型 (Japanese)

Note: Getting Started with python

python is not a requirement for this class, but it will help your understanding about the algorisms to be learned and also assist your future research.

注：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: A Light Weight Language (LWL)

Install: http://conf.msl.titech.ac.jp/Lecture/InstallPython/InstallPython.html

- **Interpreter language** (インタプリタ言語 – 逐次解釈)

  ⇔ Compiled language (コンパイル言語 – 機械語翻訳)

  Slower execution, but faster development
- Only **interpreter** and **editor** are required
- Free or public domain versions available
- Grammar similar to C, C++, perl, php, …
- Native **Object-Oriented** (オブジェクト指向) language
- Efficient functions and libraries

  Text processing: Regular expression (正規表現),

  csv, html, xml, json etc

  Science: numpy, scipy, scikit-learn etc

  Network: …

  Graph plotting: matplotlib etc

  GUI: tkinter, pygtk etc

Python distribution: My recommendation

Distribution: Same main software may be combined with different sets of supporting programs / files

ex. Linux distribution: CentOS, Ubuntu, SUSE, …

For python

Linux / Mac OS X pre-installed: Basic python

you may need to install numpy, scipy, etc by the command:

```
pip install {module_name}
```

Active python: Commercial base, multi-platform

Free distribution is available as ‘Community Edition’

Anaconda: Basic python + major libralities (modules)

including numpy, scipy, scikit-learn, etc

https://www.anaconda.com/products/individual

For installation, see

http://conf.msl.titech.ac.jp/Lecture/InstallPython/InstallPython.html

Anaconda: License condition changed

Apr, 2020

Free Anaconda Individual Edition

For solo practitioners, students, and researchers.

For others (200名以上の営利団体による利用を有償化)

Commercial Edition @ $14.95/month, etc

Some ideas to adopt this change (有償化への対応策例)

https://qiita.com/c60evaporator/items/ba41cef4b37465c39948

https://blog.neko-ni-naritai.com/entry/installing-intel-channel-numpy

Editor vs Word processor

	Editor	Word processor
Startup time (起動時間)	Shorter	Longer
Processing speed (実行速度)	Faster	Slower
Memory	Light	Heavy
Text style / format	Usually none	Required
File format	**Basically text-based**	**Application specific**
Others	**Specialized for specific program languages.** Macro (small program languages)	Print (WYSIWYG): What You See is What You Get
Examples	Linux : vi, emax Windows: TeraPad, Sakura Edtior Multi : Visual Studio Code, Sublime text, Atom	MS-Word

Recommendation:

Microsoft Visual Studio Code: https://code.visualstudio.com/

- Multiplatform (Windows, MacOS, Linux)
- Multilanguage
- Integrated Development Editor (IDE)

Q: Methfessel-Paxton and tetrahedron method

These are used to integrate / smear E(k) obtained by band calculations

purposes of smearing used in band calculations.

1. **Increasing the accuracy of 1st BZ integration (interpolation)**

   Tetrahedron method

2. **Stabilize convergence of SCF (distribution)**

   Gauss smearing, Fermi smearing

3. **Make the DOS display easier to read (smoothing)**

   Polynomial fitting may help

   but convolution (smearing) and tetrahedron method are commonly used

Problem of smearing by convolution

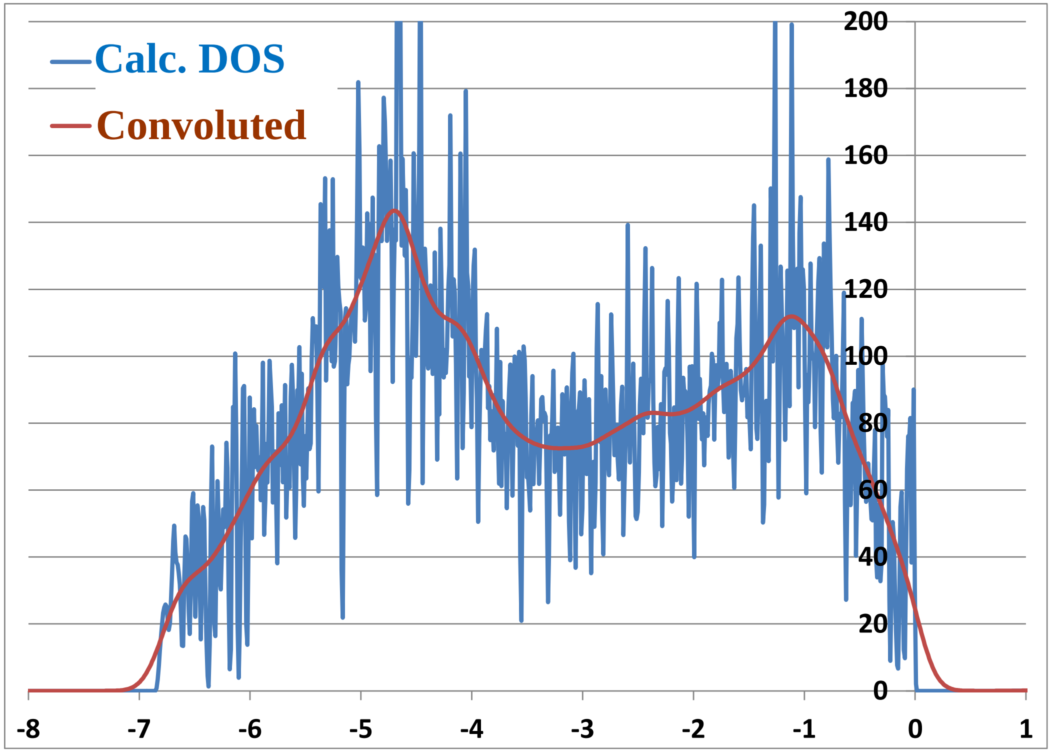
Density of state (DOS) function calculated by density functional theory

密度汎関数計算で得たa-$InGaZnO_4$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)

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

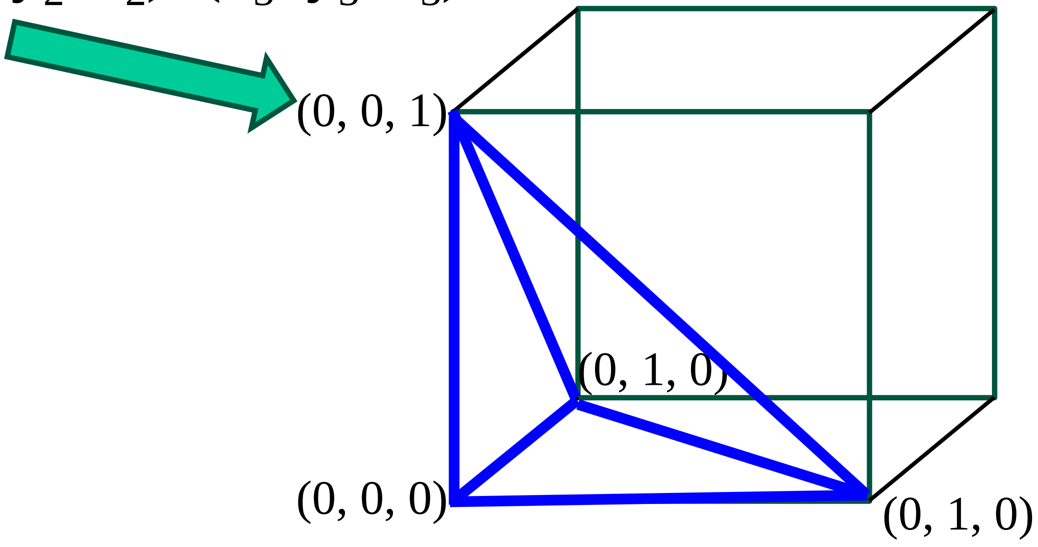


Note: Estimation of band, edge energies will have the errors originating from the smearing width *w*

A: Tetrahedron method

1. Divide the first Brillouin zone to tetrahedrons

2. Choose one tetrahedron with the vertexes (x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), normalize the vertexes to



3. Interpolate by

$$E(\boldsymbol{k}) = E_{000} + (E_{100} - E_{000})k_x + (E_{010} - E_{000})k_y + (E_{001} - E_{000})k_z$$

, where E_{ijk} is $E(\mathbf{k})$ at a vertex (i, j, k)

4. Integrate $E(\mathbf{k})$

No smearing of $D(E)$ => **Exact estimation of HOMO and LUMO are possible**

Q: Determination of E_V (HOMO), E_C (LUMO), E_g

1. Calculate $E(k)$ by band calculation throughout the first Brillouin zone.
2. Sort $E(k)$ from the lowest to the highest
3. Find HOMO (or E_F) level from the number of electrons (N_{tot}) and the number of orbitals included in the calculation
4. Find LUMO level as the next upper $E(k)$ from HOMO

If you have Density-Of-States data $D(E)$,

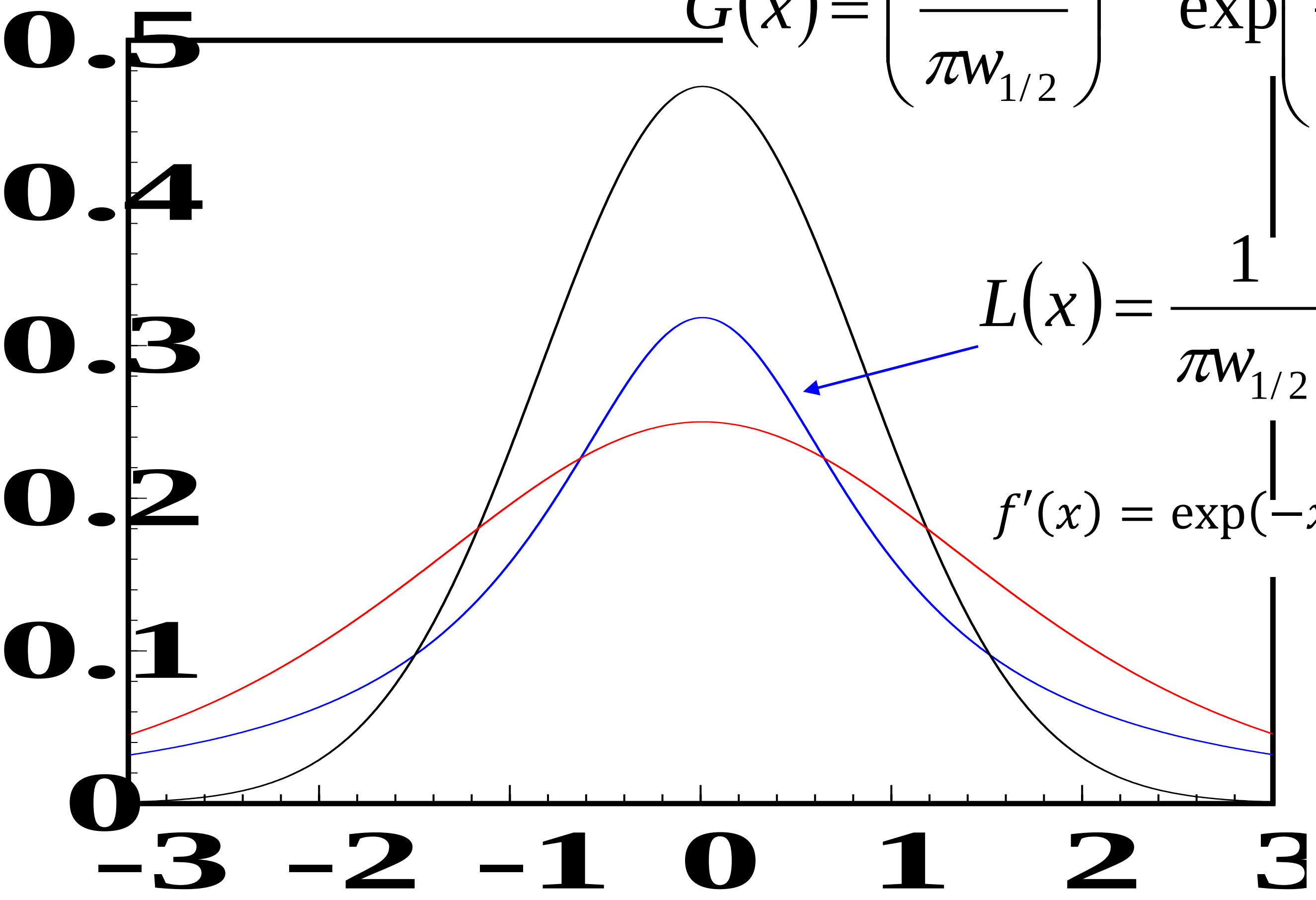
1. Integrate $D(E)$ from the lowest energy to get an integrated electron number function $N(E)$
2. Find HOMO (E_F) as the energy satisfying $N(E_F) = N_{tot}$
3. Find LUMO level as the next upper $E(k)$ from HOMO

NOTE: If $D(E)$ is smeared, it is difficult to find exact HOMO and LUMO.
Use non-smeared $D(E)$ or tetrahedron-method

For VASP, see [tkProg_Root]¥tkprog_base¥VASP¥gbandedges

A: Smearing functions

$w_{1/2}$ = 1 eV



$$G(x) = \left(\frac{\ln 2}{\pi w_{1/2}} \right)^{1/2} \exp\left(-\ln 2 \left(\frac{x}{w_{1/2}} \right)^2 \right)$$

$$L(x) = \frac{1}{\pi w_{1/2}} \frac{1}{1 + \left(x / w_{1/2} \right)^2}$$

$$f'(x) = \exp(-x^2) \,/ (1 + \exp(x))^2$$

A: Methfessel Paxton function

M. Methfessel and A.T. Paxton, High-precision sampling
for Brillouin-zone integration in metals, Phys. Rev. B **40** (1989) 3616

Expand the delta function with Hermitian polynomials

$$\delta(x) = \sum_{n=0}^{\infty} A_n H_{2n}(x) \exp\left(-x^2\right) \qquad A_n = \frac{(-1)^n}{n! 4^n \sqrt{\pi}}$$

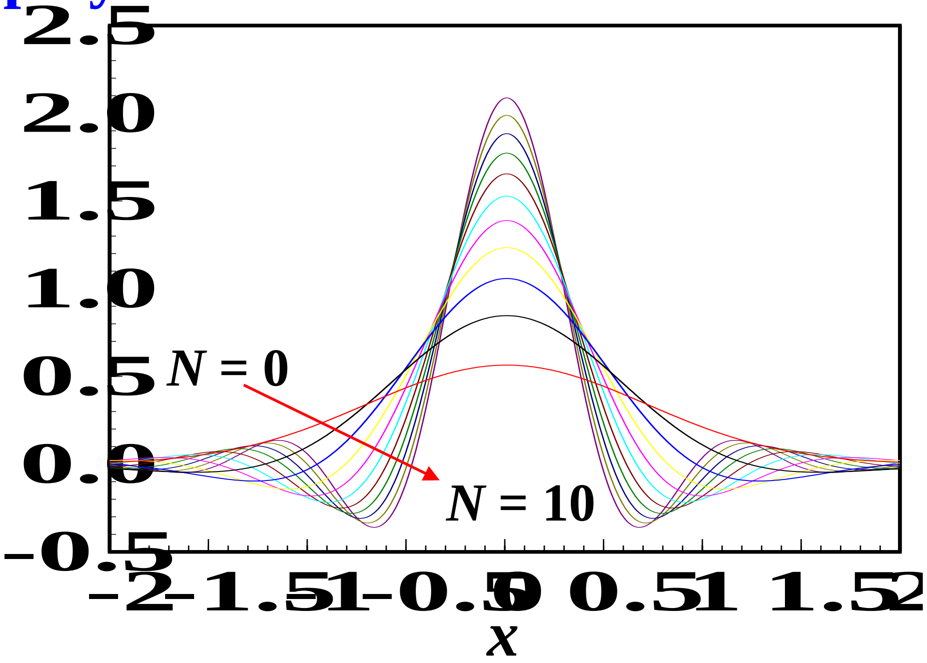
$$D_N(x) = \sum_{n=0}^{N} A_n H_{2n}(x) \exp\left(-x^2\right)$$

***D*$_N$ (*x*) is a (2*N*+1)-order polynomial,**
orthogonal to a 2*N* or less order polynomial

Approximation of the Step Function

$$S_N(x) = 1 - \int_{-\infty}^{x} D_N(t)\,dt$$

$$S_0(x) = (1/2)(1 - erf(x))$$



Hermitian polynomial

$\left(\frac{d^2}{dx^2} - 2x\frac{d}{dx} + 2n\right)H_n(x) = 0$ **solution of a problem**

$$H_n(x) = n!\sum_{m=0}^{\mathrm{int}(n/2)} \frac{(-1)^m}{m!\,(n-2m)!}(2x)^{n-2m}$$

$$H_0(x) = 1, H_1(x) = 2x$$
$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$$
$$H_n{}'(x) = 2nH_{n-1}(x) = 2xH_n(x) - H_{n+1}(x)$$

H

$\boldsymbol{H_n\,(x)\exp(-x^2\,/2)}$ **is an orthonormal basis**

$$\int_{-\infty}^{\infty} H_n(x)H_m(x)\exp(-x^2)\,dx = \delta_{mn}2^n\sqrt{\pi}n!$$

Wavefunction of harmonic oscillator model:

$$\Psi_n(x) = (2^n\sqrt{\pi}n!)^{1/2}H_n(x)\exp(-x^2/2)$$

A: Characteristics of Methfessel Paxton Functions

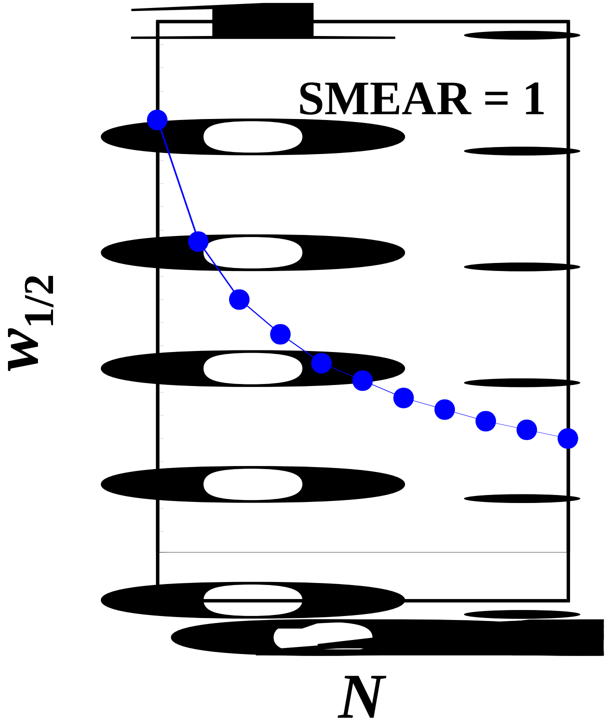
1. **If the band structure and DOS can be approximated by polynomials of the *2N-th* order or less, smearing with an N-th-order MP function will not produce an integration error.**

2. **In the case of a simple band structure or DOS, integration error will be zero even if the SMEAR width is quite large.**

3. **When the band structure is complex (e.g., d-system), the optimal SMEAR width is comparable to Gaussian**

4. **The actual smearing width $w_{1/2}$ depends on the value of the order *N* and SMEAR width.**

5. **Negative values or values over 1.0 for occupancy**

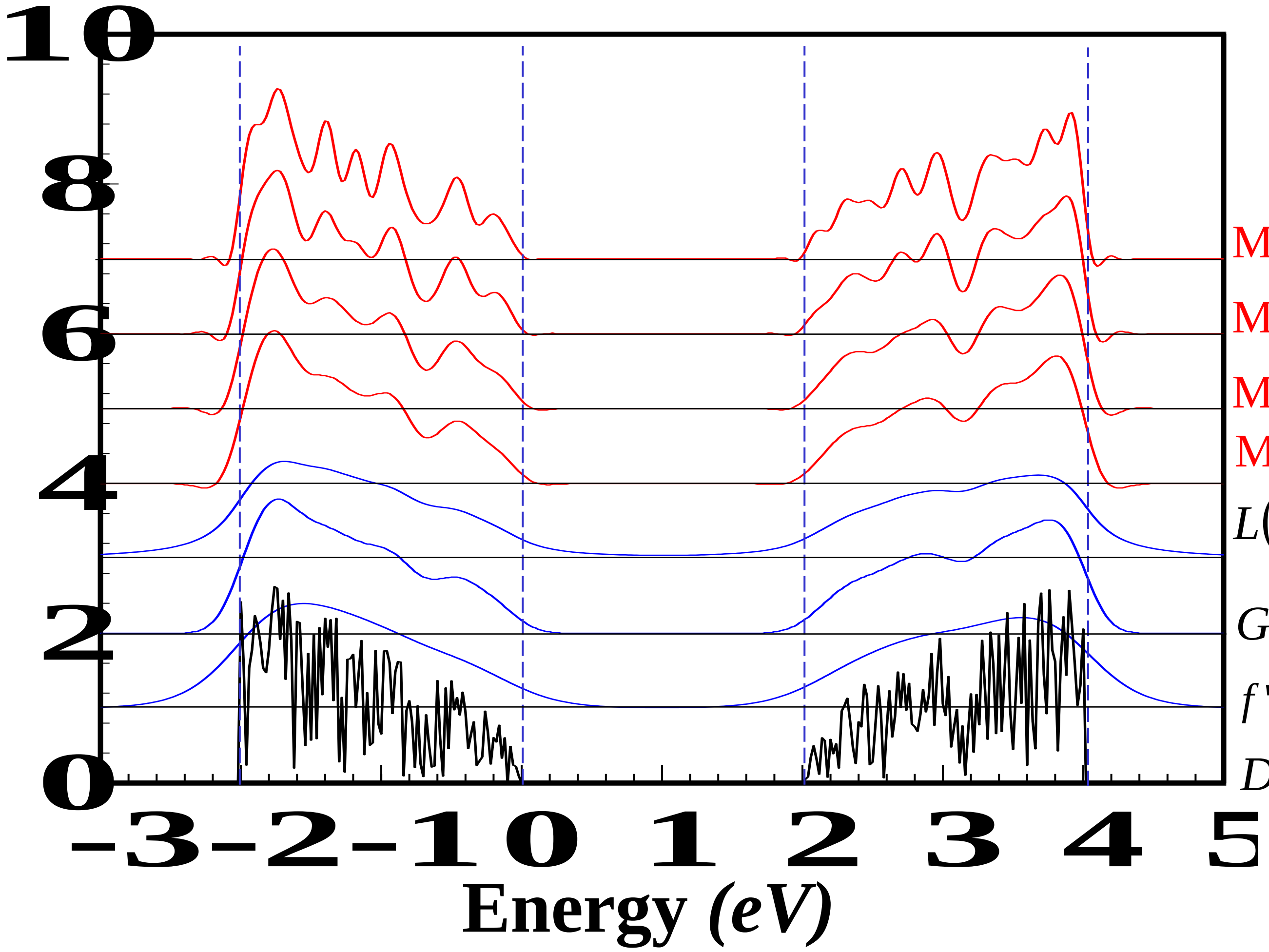


A: Smearing of density of states $D(E)*f_S(E)$

$$D(E) = D_{V0}(E_V - E)^{1/2} + D_{C0}(E - E_C)^{1/2} * (1 + \mathrm{rand}[-0.5, 0.5])$$

Smearing: $D(E) * f_s(E) = \int D(E')f_s(E' - E)dE'$

with w = 0.2 eV



Methfessel Paxton Order 10

Methfessel Paxton Order 5

Methfessel Paxton Order 2

Methfessel Paxton Order 1

$$L(x) = \frac{1}{\pi w}\frac{1}{1+(x/w)^2}$$

$$G(x) = (1/\pi w)^{1/2}\exp\left(-(x/w)^2\right)$$

$$f'(x) = \exp(x/w)/(1+\exp(x/w))^2$$

$$D(E) = D_V(E) + D_C(E)$$

Q: Avoid local minimum issues

Avoid local minimum issues in non-linear optimization in particular for crystal structure determination for unknown materials

- **Examine different initial values**
- **Use algorisms that has better robustness for converging region for the initial search.**
  - **Simplex method**
  - **Simulated annealing, Dumped molecular dynamics**
    Try to exit from the current minimum to examine another possible minimum
- **Use first-principles structure relaxation calculations to verify the stability of the analyzed structures**
- **Use structure-search programs such as**
  USPEX (genetic algorism) and
  CARYPSO (particle swarm optimization)

Q: Search in tree-structured data

以下の方法 に 基づいて 探索木 から 探索を 行う プログラム:
breadth-first search, best-first search, and A-star algorism

Can’t give deep instructions, but for tree-structured problems:

- **Recursive programming may help to make a logic simple (but may need more memory and calculation time)**
- **Build a class that has pointers to child and parent trees**
- **For speed up, parallel search is easily applied for tree-structured problem**
  But python is not good for parallel computing due to ‘Global Interpreter Lock’
  https://kosuke-space.com/python-parallel-processing

Fundamental of computer

コンピュータの基礎

Numeric representation

(数の表現)

Base 10
(decimal)
(10進数)

$$1975 = 1\times1000 + 9\times100 + 7\times10 + 5\times1$$
$$= 1\times10^{3} + 9\times10^{2} + 7\times10^{1} + 5\times10^{0}$$

the 1000’s place
(1000の位)

All data in computer are represented by 0 or 1 (binary) : bit (b)

Base 2
(binary)
(2進数)

$$(11011)_2 = 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$
$$= 1\times(16)_{10} + 1\times(8)_{10} + 0\times(4)_{10} + 1\times(2)_{10} + 1\times(1)_{10}$$
$$= (27)_{10}$$

Base *r*
(*r*進数)

$$N = a_n r^{n} + a_{n\text{-}1} r^{n\text{-}1} + \cdots\cdots + a_3 r^{3} + a_2 r^{2} + a_1 r^{1} + a_0 r^{0}$$
$$= (a_n a_{n\text{-}1} \cdots\cdots a_3 a_2 a_1 a_0)_r$$

Numeric representation

(数の表現)

Base 8 (octal) (8進数) **(01234567)**

2 digits: $0 \sim 8^2 - 1 = 63$

00: $0\times8^1 + 0\times8^0 = 0$

53: $5\times8^1 + 3\times8^0 = 43$

77: $7\times8^1 + 7\times8^0 = 63$

Base 16 (hexadecimal) (16進数) **(0123456789ABCDEF) = (0 ~ 15)**

2 digits: $0 \sim 16^2 - 1 = 255$

00: $0\times16^1 + 0\times16^0 = 0$

9F: $9\times16^1 + 15\times16^0 = 159$

FF: $15\times16^1 + 15\times16^0 = 255$

(ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789+/) = (0 ~ 63)

Correspondence relations (対応関係)

Base 10	Base 2	Base 8	Base 16
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Convert Base (基数の変換)

Base *r* to Base 10

$N_r = (a_n a_{n-1} \cdots\cdots a_3 a_2 a_1 a_0)_r$

$N_{10} = a_0 r^0 + a_1 r^1 + a_2 r^2 + a_3 r^3 + \cdots\cdots + a_{n-1} r^{n-1} + a_n r^n$

Ex. $1101_2 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 13_{10}$

Base 10 to Base *r*

$N_{10} = (b_n b_{n-1} \cdots\cdots b_3 b_2 b_1 b_0)_{10} = (c_{n'} c_{n'-1} \cdots\cdots c_2 c_1 c_0)_r$

$= \boldsymbol{c_0} r^0 + c_1 r^1 + c_2 r^2 + \cdots + c_{n-1} r^{n-1} + c_n r^n$

$= \boldsymbol{c_0} + r(\boldsymbol{c_1} + c_2 r^1 + \cdots + c_{n-1} r^{n-2} + c_n r^{n-1})$

$= \boldsymbol{c_0} + r(\boldsymbol{c_1} + r(\boldsymbol{c_2} + c_3 r \cdots + c_{n-1} r^{n-3} + c_n r^{n-2}))$

(1) $N_{10}^{(0)} = N_{10} = N_{10}^{(1)} * r + \boldsymbol{c_0}$ where $0 \le c_0 < r$

(2) $N_{10}^{(1)} = N_{10}^{(2)} * r + \boldsymbol{c_1}$ where $0 \le c_1 < r$

… repeat until $N_{10}^{(n+1)} = 0$

$=> N_r = (c_n c_{n-1} \cdots\cdots c_2 c_1 c_0)_r$

Ex. Base 10 to Base 8

$302_{10} = 8 \times 37 + 6$

$37_{10} = 8 \times 4 + 5$

$4_{10} = 8 \times 0 + 4$

$300_{10} = 456_8$

Python program: base.py

Program: base.py

Usage: python base.py value base_source base_target

Ex.

COMMAND:

```
python base.py FA 16 8
```

Convert FA in base 16 to base 8

OUTPUT:

Convert FA in base 16 to base 10

1st digit = 10: + 10 * 16^0 => + 10_{10} => 10_{10}

2nd digit = 15: + 15 * 16^1 => + 240_{10} => 250_{10}

Convert 250 in base 10 to base 8

250_{10} = 31 * 8 + 2: base_8 => 2

31_{10} = 3 * 8 + 7: base_8 => 72

3_{10} = 0 * 8 + 3: base_8 => 372_8 result

Units of data processed in computers

(コンピュータ内のデータ単位)

bit (b): binary: 0 or 1

In computer: 8 bits data is treated as a fundamental unit

byte (B): 0 ~ $2^8 - 1 = 255$

1 kB = 2^{10} B = 1,024 B

1 MB = 1024 kB = 1,048,576 B

1 TB = 1024 GB = 1024^2 MB = 1024^3 kB = 1024^4 B

Numeric representation: Integer (整数型)

Integer type：Based on the CPU bit (CPUのbit数が基本)

16bit for 16bit CPU

unsigned int (符号無し整数型)　0 ~ 2^{16} – 1 = 65,535

signed int　(符号付き整数型)　–32,768 ~ +32,767

32bit for 32bit CPU

unsigned int (符号無し整数型)　0 ~ 4,294,967,295

signed int　(符号付き整数型)　–2,147,483,648 ~ + 2,147,483,647

For all CPUs:

int　　　　: depends on CPU bits

short int　 : 16 bit

long int　　: 32 bit

long long int: 64 bit

Numeric representation: Floating point, Real

(浮動小数点型, 実数)

Floating point type: Minimum 32bit (except half precision)

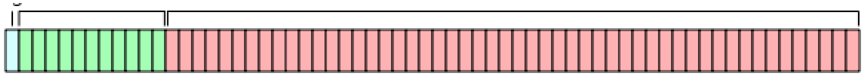
The range of available value depends on computer architectures, programming language etc.

$$-1.011101_2 \times 2^{-0101_2}$$

Sign (符号)

Fraction (仮数部)

Exponent (指数部)



Sign (1bit) Exponent (11bit) Fraction (52bit)

C language (C言語)

float : 32 bit 3.4E-38 ~ 3.4E+38

double : 64 bit 1.7E-308 ~ 1.7E+308

long double: 64 bit

Fortran

Single precision (単精度) FP (REAL) : 32 bit

Double precision (倍精度) FP (DOUBLE) : 64 bit, 16 digits (桁) in decimal

Quadruple precision (4倍精度) FP (REAL*16) : 128 bit

Definition of IEEE 754 (binary32, binary64):

Sign : 1 bit

Exponent: 8 bits (REAL, –128 ~ +127) 11 bits (DOUBLE, –1024 ~ +1023)

Fraction : 23 bits (REAL) 52 bits (DOUBLE)

8,388,608: 7 digits 4,503,599,627,370,495: 16 digits

Required sizes: Integer types

unsigned int (16 bit): 65,536

16 bit CPU can handle only 64 kB of memory

(アドレスバスが16bitだと、64 kBのメモリーしか扱えない)

unsigned int (32 bit): 4,294,967,295

32bit CPU can handle 4 GB memory

(アドレスバスが32bitだと、4 GBのメモリーを扱える)

GDP of Japan: ~5 trillion US$ = 500,000,000,000,000 JYen

(requires 16 digits)

cf. unsigned long long int (64 bit): ~1.8E+19 (18 digits)

The ratio of the circumference of a circle (円周率):

Significant figure: 50 trillion digits (as of Jan, 2020)

Need to use multi-fold calculation (多倍長計算)

Implemented based on software

Required sizes: FP types for quantum calc.

1s orbital energy level:

H atom : 13.6 eV
heavy atoms: >> keV

Energies related to physical properties

Thermal energy at room temperature: 26 meV
Magnetism: several meV

Quantum simulations of physical properties require the precision for the meV – MeV range (over 9 digits precision)

Definition of standard FP: IEEE 754

Fraction: 23 bit (single)	8,388,608	7 digits
Fraction: 52bit (double)	4,503,599,627,370,495	16 digits

Required sizes: FP types for semiconductor simulation

Boltzmann factor: $\exp(-E_g / k_B T)$

E_g = 1.1 eV

$k_B T$ = 0.026 eV (T = 300 K) => exp(-42) ~ $\mathbf{10^{-19}}$

E_g = 4.0 eV

$k_B T$ = 0.026 eV (T = 300 K) => exp(-154) ~ $\mathbf{10^{-67}}$

$k_B T$ = 0.00026 eV (T = 3 K) => exp(-15400) ~ $\mathbf{10^{-5141}}$

Double precision (64bit): Fraction: 16 digits

Exponent: –1024 ~ +1023 (2^{-1024} ~ 10^{-308})

Quad precision : 128 bit

Octuple precision (8倍精度): 256 bit

Error of floating point (浮動小数点の誤差)

Representation of floating point in computer:

$-1.011101_2 \times 2^{-015_{10}}$ **(in binary)**

Errors arise from converting Base 10 to Base 2.

- **Some values do not have errors between Base 10 and Base 2 if fraction equals to 2^n**

$1.0 = (1.0)_2 \times 2^0$

$0.5 = (1.0)_2 \times 2^{-1}$

$0.125 = (1.0)_2 \times 2^{-3}$

$0.0390625 = 1.25 \times 2^{-5} = (1.01)_2 \times 2^{-5}$

$1.75 = 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = (1.11)_2$

$0.65625 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = (0.10101)_2$

$100.0 = 1.5625 \times 64 = (1+2^{-1}+2^{-4}) \times 2^6 = (1.1001)_2 \times 2^4$

- **Other values have errors even if it is represented by a simple figure in Base 10:**

$0.1 = (1.1001100110011001\cdots)_2 \times 2^{-3}$

Program (roundoff error): sum_error.py

Usage: python sum_error.py *h n iPrintStep*

Summing up *h* for n times with different precision interger types. Output every *iPrintStep* steps.

python sum_error.py 0.1 100 20

```
exact:   sum16 (error)                          sum32 (error)                          sum64 (error)
0.1000: 0.099975585937500000 (+2.44e-05)  0.100000001490116119 (-1.49e-09)  0.100000000000000006 (+0.00e+00)
2.1000: 2.095703125000000000 (+4.30e-03)  2.100000143051147461 (-1.43e-07)  2.100000000000000533 (-4.44e-16)
4.1000: 4.089843750000000000 (+1.02e-02)  4.099998474121093750 (+1.53e-06)  4.100000000000001421 (-8.88e-16)
6.1000: 6.121093750000000000 (-2.11e-02)  6.099996566772460938 (+3.43e-06)  6.099999999999994316 (+6.22e-15)
8.1000: 8.148437500000000000 (-4.84e-02)  8.099994659423828125 (+5.34e-06)  8.099999999999987210 (+1.24e-14)
```

python sum_error.py 0.125 100 20

```
exact:   sum16 (error)                          sum32 (error)                          sum64 (error)
0.1250: 0.125000000000000000 (+0.00e+00)  0.125000000000000000 (+0.00e+00)  0.125000000000000000 (+0.00e+00)
2.6250: 2.625000000000000000 (+0.00e+00)  2.625000000000000000 (+0.00e+00)  2.625000000000000000 (+0.00e+00)
5.1250: 5.125000000000000000 (+0.00e+00)  5.125000000000000000 (+0.00e+00)  5.125000000000000000 (+0.00e+00)
7.6250: 7.625000000000000000 (+0.00e+00)  7.625000000000000000 (+0.00e+00)  7.625000000000000000 (+0.00e+00)
10.125: 10.125000000000000000 (+0.00e+00)  10.125000000000000000 (+0.00e+00)  10.125000000000000000 (+0.00e+00)
```

python sum_error.py 0.0390625 100 20

```
exact:    sum16 (error)                          sum32 (error)                          sum64 (error)
0.0391: 0.039062500000000000 (+0.00e+00)  0.039062500000000000 (+0.00e+00)  0.039062500000000000 (+0.00e+00)
0.8203: 0.820312500000000000 (+0.00e+00)  0.820312500000000000 (+0.00e+00)  0.820312500000000000 (+0.00e+00)
1.6016: 1.601562500000000000 (+0.00e+00)  1.601562500000000000 (+0.00e+00)  1.601562500000000000 (+0.00e+00)
2.3828: 2.382812500000000000 (+0.00e+00)  2.382812500000000000 (+0.00e+00)  2.382812500000000000 (+0.00e+00)
3.1641: 3.164062500000000000 (+0.00e+00)  3.164062500000000000 (+0.00e+00)  3.164062500000000000 (+0.00e+00)
```

Roundoff error (桁落ち誤差)

Summing small value *h* for many times *N*

Calc by summation

```
x = 0.0;
for i in range(N)
    x = x + h
```

Error is accumulated by each summation

Calc by multiplication

```
x0 = 0.0;
for i in range (N)
    x = x0 + i * h
```

Typically multiplication is slower than summation, but **the total error originates from only one multiplication operation**

Result of sum_error.py (compare different precision FP types): *h* = 0.01, *N* = 101

Program: sum_error.py

```
        float16: half precision, 16 bit           float32: single precision, 32 bit           float64: half precision, 64 bit
Exact:        float16 (error)                              float32 (error)                              float64 (error)
0.0100: 0.0100021362304687500 (-2.14e-06)  0.009999999776482582 (+2.24e-10)  0.0100000000000000000 (+0.00e+00)
0.1100: 0.1100463867187500000 (-4.64e-05)  0.109999984502792358 (+1.55e-08)  0.1099999999999999987 (+1.39e-17)
0.2100: 0.2100830078125000000 (-8.30e-05)  0.210000023245811462 (-2.32e-08)  0.2100000000000000048 (-5.55e-17)
0.3100: 0.3100585937500000000 (-5.86e-05)  0.309999972581863403 (+2.74e-08)  0.3100000000000000109 (-1.11e-16)
0.4100: 0.4101562500000000000 (-1.56e-04)  0.409999877214431763 (+1.23e-07)  0.4100000000000000198 (-1.67e-16)
0.5100: 0.5097656250000000000 (+2.34e-04)  0.509999811649322510 (+1.88e-07)  0.5100000000000000231 (-2.22e-16)
…
0.8100: 0.8027343750000000000 (+7.27e-03)  0.809999525547027588 (+4.74e-07)  0.8100000000000000497 (-4.44e-16)
0.9100: 0.9003906250000000000 (+9.61e-03)  0.909999430179595947 (+5.70e-07)  0.9100000000000000586 (-5.55e-16)
1.0100: 0.9980468750000000000 (+1.20e-02)  1.009999394416809082 (+6.06e-07)  1.0100000000000000675 (-6.66e-16)
```

Python program: sum.py

Program: sum.py
Usage: python sum.py h N
Summing small value h for many times N

Example command: `python sum.py 0.1 10`

OUTPUT:

```
0: 0.0 + 0.1 => 0.1
1: 0.1 + 0.1 => 0.2
2: 0.2 + 0.1 => 0.30000000000000004
3: 0.30000000000000004 + 0.1 => 0.4
4: 0.4 + 0.1 => 0.5
…
7: 0.7 + 0.1 => 0.7999999999999999
…
9: 0.8999999999999999 + 0.1 => 0.9999999999999999
```

Caution for conditional branch

(条件分岐における注意)

Calculation of integers does not produce errors.

=> Conditional judgement only using integers works properly

整数変数のみでの条件判断は誤差が出ないので問題ない

```
if i * 10 == 30:
    print(“i == 30”)    # executed if i == 30
```

Calculation of floating points can produce roundoff error.

=> Strict conditional judgement often does not work properly: **Must not be used!!**

実数計算では丸め誤差が発生するので、厳格な条件判断は使ってはいけない

```
if x * 10.0 == 30.0:
    print(“x == 3.0”)   # expected to execute if x == 3.0, but may be not
```

[Important!!] Consider possible errors:

【重要】起こりうる誤差 (epsilon: eps) を考慮した条件判断をする

```
eps = 1.0e-30    # epsilon: small, but a bit larger value than possible error
if abs(x * 10.0 - 30.0) < eps:
    print(“x == 3.0”)   # executed if x is practically equal to 3.0
```

Program: bad_if.py

Usage: python bad_if.py h n answer

  Check the condition h * n == answer

python bad_if.py 0.1 1 0.1 **Confirm $\sum_{i=1}^{1} 0.1 == 0.1$**

Summing up 0.1 for 1 times: v = 0.1

v == 0.1?: True

|v – 0.1| < 1e-10?: True

python bad_if.py 0.1 2 0.2 **Confirm $\sum_{i=1}^{2} 0.1 == 0.2$**

Summing up 0.1 for 2 times: v = 0.2

v == 0.2?: True

|v – 0.2| < 1e-10?: True

python bad_if.py 0.1 3 0.3 **Confirm $\sum_{i=1}^{3} 0.1 == 0.3$: Failed**

Summing up 0.1 for 3 times: v = 0.30000000000000004

v == 0.3?: **False** **$\sum_{i=1}^{3} 0.1 == 0.3$ では判断を間違える**

|v – 0.3| < 1e-10?: **True** **$\left|\sum_{i=1}^{3} 0.1 - 0.3\right| <$ eps (eps = 10^{-10}) では正しく判断できる**

How to use conditional branch, if (条件分岐の判断)

Bad (悪い例):

```
if x * 10.0 == 30.0:
```

DO NOT use the strict comparison ‘==‘ for floating values

(浮動小数点の比較には、厳密な比較 == は使わない)

Good (良い例):

```
eps = 1.0e-30    # epsilon:
```

A value satisfactory smaller than minimum expected value

(想定される誤差よりも十分大きいが、なるべく小さい値を設定する)

```
if abs(x * 10.0 - 30.0) < eps
```

Program: bad_int.py

Usage: python bad_int.py *h* *n*
Check interger conversion of the summation of *h* for *n* times

python bad_int.py 0.1 100
Summing up 0.1 for 100 times: v = 9.99999999999998
int(9.99999999999998) = 9 10でなければいけない
int(9.99999999999998 + 1e-10) = 10 **eps** (10^{-10}) を加えてから int() を取ることで正しい解

python bad_int.py 0.4 20
Summing up 0.4 for 20 times: v = 8.000000000000002
int(8.000000000000002) = 8 正しい解だが、v には誤差があることに注意
int(8.000000000000002 + 1e-10) = 8 **eps** (10^{-10}) を加えてから int() を取っても正しい解

python bad_int.py 1.2 20
Summing up 1.2 for 20 times: v = 23.999999999999993
int(23.999999999999993) = 23 24でなければいけない
int(23.999999999999993 + 1e-10) = 24 **eps** (10^{-10}) を加えてから int() を取ることで正しい解

Case for floating point to integer conversion (浮動小数点 => 整数変換):

Bad (悪い例):

```
n = int(v)
```

Good (良い例):

```
eps = 1.0e-6
n = int(v + eps)
```

Typical cases for FP calculations with care

Evaluate possible errors every time for FP floating point) calculations

- Error originates from the limited length of FP type: underflow, overflow

  Representation range of 64bit FP (IEEE 754 standard)

  Exponent: 11 bit -1024 ~ +1023

  Fraction : 23 bit 4,503,599,627,370,495: 16 digits

- **FP type in computer cannot represent accurate values for most of integers**

  $100.0_{10} = 1.5625\times64 = (1+2^{-1}+2^{-4}) \times 2^{6} = (1.1001)_{2}\times2^{4}$

- **Most of FP values in computer include errors**

  1.0/3.0 ＝ 0.3333…333 (16 digits) Error ~ 10^{-16} should be included

Conditional branch:

Bad: if x * 10.0 == 30.0: **No guarantee to get the correct judge 'true'** even if x = 3.0

Good: eps = 1.0e-30 # epsilon: A value satisfactory smaller than expected values

if **abs(x * 10.0 - 30.0) < eps**: **Gives the correct judge within the error of eps**

FP => integer conversion:

How to calculate the number of division in the range xmin – xmax at xstep step

Bad: n = int((xmax – xmin) / xstep): The value in int() can include error.

Even if the correct value is n = 4,

you will get n = 3 if int() becomes 3.99999… due to error,.

Good:

eps = 1.0e-6

n = int((xmax – xmin) / xstep + eps)

Even if (xmax – xmin) / xstep becomes smaller than the expected integer due to erro,

you can receive the correct value as long as the error is smaller than eps.

数値演算プログラムの一般的な注意

浮動小数点型の演算では、常に誤差を意識すること

- 変数長の制限による誤差: underflow, overflow

  IEEE 754の標準で、64bit浮動小数点の範囲は

  指数部: 11 bit -1024 ~ +1023

  仮数部: 23 bit 4,503,599,627,370,495: 16桁

- **浮動小数点では、整数を "正確に" 表現できない**

  $100.0_{10} = 1.5625\times64 = (1+2^{-1}+2^{-4}) \times 2^6 = (1.1001)_2\times2^4$

- **有限の桁数の浮動小数点の表現は、ほぼすべての場合に誤差を含む**

  1.0/3.0 ＝ 0.3333…333 (小数点以下16桁) 10^{-16} 程度の誤差が発生する

条件分岐の判断:

悪い例: if x * 10.0 == 30.0: x = 3.0 であっても、**true と判断される保証はない**

良い例: eps = 1.0e-30 # epsilon: 想定される誤差よりも十分大きいが、なるべく小さい値を設定する。

if **abs(x * 10.0 - 30.0)** < **eps**: **誤差 eps 以内で必ず実行される**

浮動小数点 => 整数変換: xmin ~ xmax の範囲を xstep 毎の幅で分割したときの分点の数

悪い例:

```
n = int( (xmax – xmin) / xstep):
```

(xmax – xmin) / xstepが誤差により 3.99999… となった場合、

本来はint() = 4 となって欲しいのに、3 となってしまう

良い例:

```
eps = 1.0e-6
n = int( (xmax – xmin) / xstep + eps):
```

(xmax – xmin) / xstepが誤差により期待する整数値より小さくなっても、

誤差がepsより小さければ、本来期待している整数値が得られる

Precision and errors in computer

Data bit width (データ長): Determine the upper limit of precision
=> Roundoff (rounding) error (丸め誤差)

Other error sources

- **Overflow (積み残し誤差, 桁あふれ):**

  e.g. by summation between large integers (有効桁数を超える整数の和･積)

  ⬄ **underflow**

  (overflow and underflow can be detected by CPU / software
  but may deteriorate calculation speed)

- **Roundoff error (桁落ち誤差)：By subtracting very similar values**

  ex: for 4 digits calculation:

  $5\sqrt{41} - 32 \sim 5*6.403 - 32.00 = 32.015 - 32.00 = 32.02 - 32.00 = 0.02$

  The given values have 4 significant digits
  but the result has only 1 significant digits

  Avoid subtraction between similar values

- **Loss of trailing digits (情報落ち)：**

  by summing / subtracting between largely-different values

  ex: 1000 + 1.456 = 1001 (The initial significant value of .456 is lost)

Errors in calculation process

- Overflow (summing large values)
- Underflow (huge numbers of summing up small values)
- Rounding error
- **Information buried (情報埋没)**
- **Truncation error (打切り誤差)**

  To sum up values slowly approaching to zero:

  - **Tailor expansion (テーラー展開)**
  - **Summation of Coulomb energy (Coulombエネルギーの和)**

  Need to terminate the summation if calculation time has limitation or the result reaches the required precision

  (計算時間と必要な精度に応じて、どこかで計算を打ち切る)

- **Convergence error (収束誤差)**

  The required precision (often expressed EPS) is given to judge the termination of iterative convergence calculations
- **Errors originating from physical model (物理モデルの誤差)**

Information buried (情報埋没)

Program: python information_buried.py

***e.g.*, calculate exp(-40) by $\exp(x) = \sum_{n=0} x^n/n!$**

Summing up large values with opposite signs results in significant errors (正負が交番する大きな数の和を取るために誤差が大きくなる)

Better to add positive values only

$\mathbf{A} = \sum_{n=0}^{N}(-x)^n/n!$ (if $x < 0$)

, and take $\frac{1}{A} = \exp(-40)$

Exact value 4.24835425529159×10^{-18}

N :	$\sum_{n=0}^{N} x^n/n!$	$1.0/\sum_{n=0}^{N}(-x)^n/n!$
0 :	1	1
1 :	-39	0.024390244
2 :	761	0.0011890606
18 :	7.3620174e+12	5.290335e-14
19 :	-1.5234693e+13	2.4096905e-14
20 :	2.9958728e+13	1.153502e-14
21 :	-5.6123978e+13	5.7878667e-15
22 :	1.0039003e+14	3.0368438e-15
23 :	-1.7180825e+14	1.6625449e-15
	Sum up large +/- values	
79 :	-1.3651644e+09	**4.2483543e-18**
115:	5.8811462	4.2483543e-18
116:	5.8811665	4.2483543e-18

Well converged,

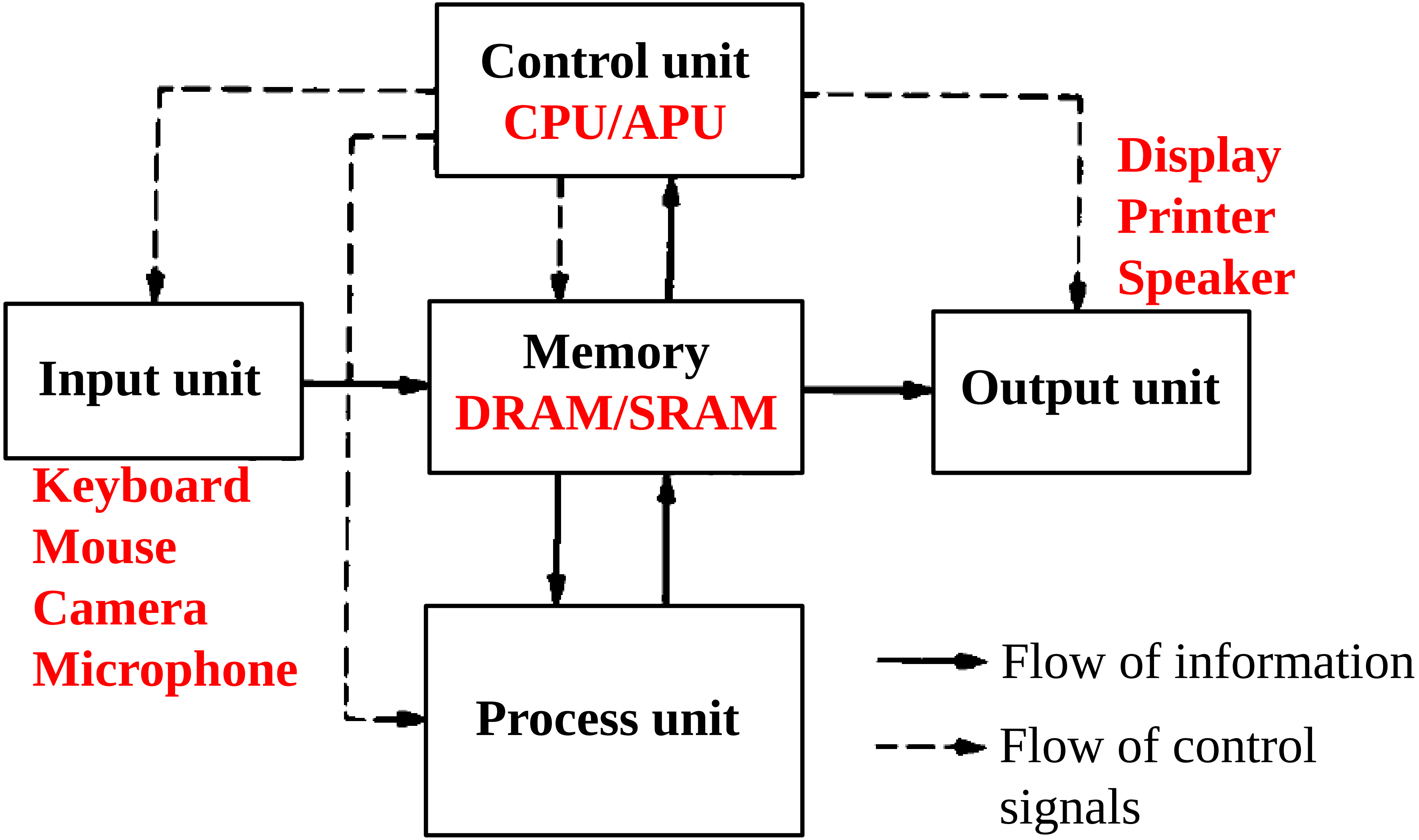
but **18 digits of error!!**

Supplementary materials

Structure of typical computer

(基本的な計算機の構成)

大河内他、基礎 電子計算機、実教出版



Computer architectures

4bit CPU:	Intel 4004	(1971)	data 4bit, address 12bit
8bit CPU:	8008	(1972)	data 8bit, address 14bit
16bit CPU:	8086	(1978)	data 16bit, address 20bit
32bit CPU:	80386SX	(1985)	External data/address 32bit Internal data 16bit, address 24bit
	80486	(1989)	data 32bit, address 32bit
	Pentium,,,		

64bit CPU: Pentium Pro(?), Itanium, Core i,, …

Pentium Pro:

Processor 32bit: Operation (命令)・Process (データ処理) in CPU

External data bus (外部データバス)

64bit: Data transfer with memory / external units

Floating point operation (浮動小数点演算)

80bit

Logical operations (bitwise operations)
(論理演算, ビット演算)

Logical NOT (Bitwise inversion) (論理否定, ビット反転)

NOT 0 = 1; NOT 1 = 0

Logical AND (論理積)

0 AND 0 = 0; 1 AND 0 = 0

0 AND 1 = 0; 1 AND 1 = 1

Logical OR (論理和)

0 OR 0 = 0; 1 OR 0 = 1

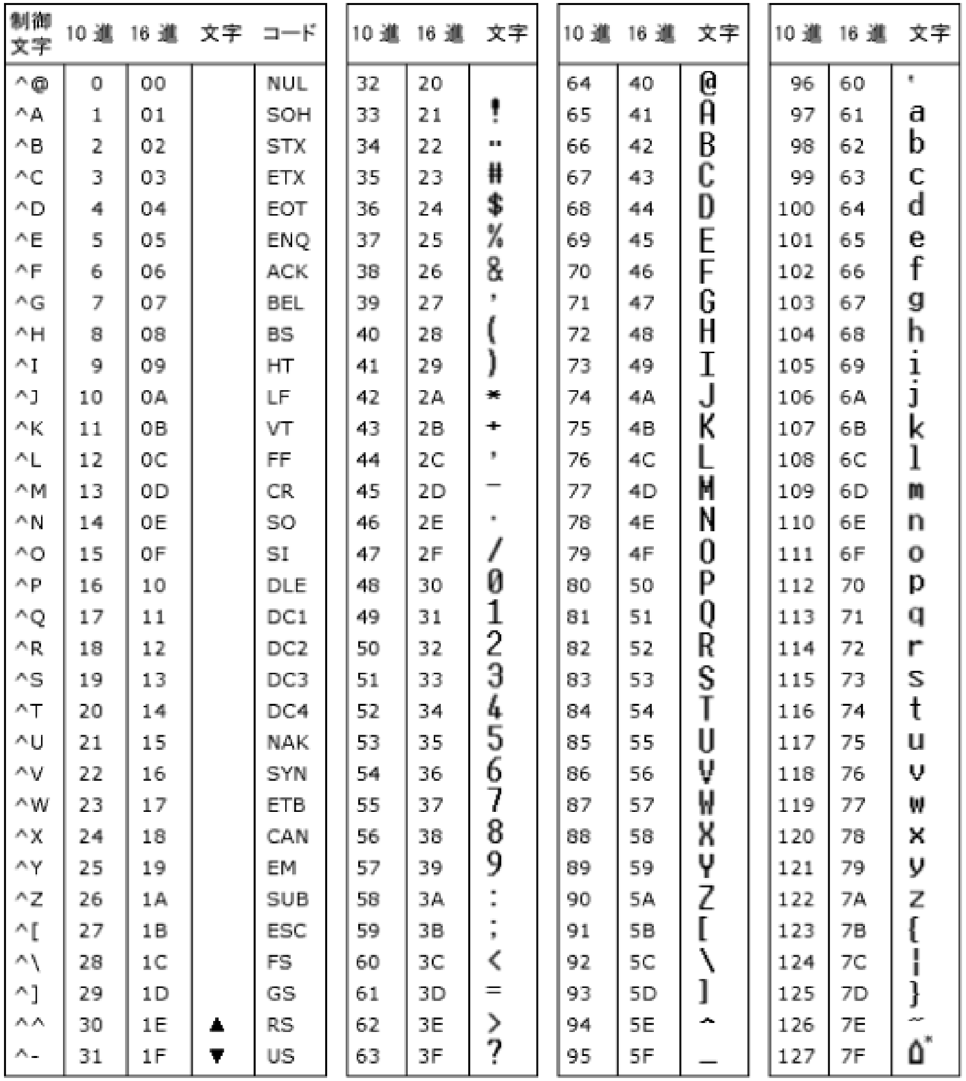
0 OR 1 = 1; 1 OR 1 = 1

Logical Exclusive OR (排他的論理和)

0 XOR 0 = 0; 1 XOR 0 = 1

0 XOR 1 = 1; 1 XOR 1 = 0

Required data size: Character type



制御文字	10 進	16 進	文字	コード	10 進	16 進	文字	10 進	16 進	文字	10 進	16 進	文字
^@	0	00		NUL	32	20		64	40	@	96	60	`
^A	1	01		SOH	33	21	!	65	41	A	97	61	a
^B	2	02		STX	34	22	"	66	42	B	98	62	b
^C	3	03		ETX	35	23	#	67	43	C	99	63	c
^D	4	04		EOT	36	24	$	68	44	D	100	64	d
^E	5	05		ENQ	37	25	%	69	45	E	101	65	e
^F	6	06		ACK	38	26	&	70	46	F	102	66	f
^G	7	07		BEL	39	27	'	71	47	G	103	67	g
^H	8	08		BS	40	28	(	72	48	H	104	68	h
^I	9	09		HT	41	29	)	73	49	I	105	69	i
^J	10	0A		LF	42	2A	*	74	4A	J	106	6A	j
^K	11	0B		VT	43	2B	+	75	4B	K	107	6B	k
^L	12	0C		FF	44	2C	,	76	4C	L	108	6C	l
^M	13	0D		CR	45	2D	-	77	4D	M	109	6D	m
^N	14	0E		SO	46	2E	.	78	4E	N	110	6E	n
^O	15	0F		SI	47	2F	/	79	4F	O	111	6F	o
^P	16	10		DLE	48	30	0	80	50	P	112	70	p
^Q	17	11		DC1	49	31	1	81	51	Q	113	71	q
^R	18	12		DC2	50	32	2	82	52	R	114	72	r
^S	19	13		DC3	51	33	3	83	53	S	115	73	s
^T	20	14		DC4	52	34	4	84	54	T	116	74	t
^U	21	15		NAK	53	35	5	85	55	U	117	75	u
^V	22	16		SYN	54	36	6	86	56	V	118	76	v
^W	23	17		ETB	55	37	7	87	57	W	119	77	w
^X	24	18		CAN	56	38	8	88	58	X	120	78	x
^Y	25	19		EM	57	39	9	89	59	Y	121	79	y
^Z	26	1A		SUB	58	3A	:	90	5A	Z	122	7A	z
^[	27	1B		ESC	59	3B	;	91	5B	[	123	7B	{
^\	28	1C		FS	60	3C	<	92	5C	\	124	7C	|
^]	29	1D		GS	61	3D	=	93	5D	]	125	7D	}
^^	30	1E	▲	RS	62	3E	>	94	5E	^	126	7E	~
^-	31	1F	▼	US	63	3F	?	95	5F	_	127	7F	⌂*

Alphanumeric (英数字文字):

0~9, A~Z, a~z,

Control chars (制御文字) etc

ASCII code: 7 bit (0 ~ 127)

Extended ASCII code

Add non-English chars,

symbols etc: 8 bit

Japanese

ASCII+half-width Kana (半角カナ): 8bit

Kanji･Kana (Full-width Kana, 全角文字): 16 bit

Shift-JIS (SJIS), JIS, EUC-JP

Universal character codes

(全世界共通文字コード)

Unicode: Started from 2 Bytes (Ver1.0.0)

Extended to 1 – 4 Bytes (UCS, Unicode / UTF-7/8/16 etc)

Numerical differentiation

数値微分

Fundamental of numerical analysis:
Differential => Difference (差分法)

Differential (微分) $\frac{dy}{dx}$ => approximated by difference (差分) $\frac{\Delta y}{\Delta x}$

The following terms will often appear.

Difference (差分) : $\Delta x = x_i - x_j, \Delta y = y_i - y_j$

Divided difference (差分商) : $\frac{\Delta y}{\Delta x}$

Forward difference (前進差分): $\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$ $(x_i < x_{i+1})$

Backward difference (後退差分): $\frac{y_i - y_{i-1}}{x_i - x_{i-1}}$ $(x_{i-1} < x_i)$

Central difference (中心差分): $\frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$ $(x_{i+1} - x_i = x_i - x_{i-1} = h > 0)$

Numerical differentiation (数値微分)

To calculate $\frac{dy}{dx}$ by computer,

replace the differential '*d*' with finite difference 'Δ' (for small Δ*x*)

$$\frac{df(x)}{dx} = \lim_{h\to 0}\frac{f(x+h)-f(x)}{(x+h)-x} \sim \frac{\Delta f(x)}{\Delta x} = \frac{f(x+h)-f(x)}{(x+h)-x} = \frac{f(x+h)-f(x)}{h}$$

Accuracy can be improved by decreasing *h*

⇔ But limited by the error of cancellation of significant digits

at least *h* > 0.01*v* (*v*: a representative value to be handled)

32bit floating point (~7 digits) : *h* > 10^{-5}*v* (should be much larger)

64bit floating point (~16 digits): *h* > 10^{-14}*v* (should be much larger)

$$f(x+h) = f(x) + \frac{df(x)}{dx}h + \frac{1}{2}\frac{d^2f(x)}{dx^2}h^2 + O(h^3)$$

$$\frac{f(x+h)-f(x)}{h} = \frac{df(x)}{dx} + \boxed{\frac{1}{2}\frac{d^2f(x)}{dx^2}\underline{h} + O(h^2)}$$

Error ∝ *h*^{1}

(Difference error, 差分誤差)

Numerical differentiation: Effect of *h*

$f(x) = x^3 \qquad df/dx = 3x^2$

		h=	1	0.1	0.01	0.001	1.00E-06
x	**f(x)**	**df(x)/dx**	**Δf(x)/Δx**				
0	0	**0**	**1**	**0.01**	**0.0001**	**0.000001**	**1E-12**
0.1	0.001	**0.03**	**1.33**	**0.07**	**0.0331**	**0.030301**	**0.0300003**
0.2	0.008	**0.12**	**1.72**	**0.19**	**0.1261**	**0.120601**	**0.1200006**
0.3	0.027	**0.27**	**2.17**	**0.37**	**0.2791**	**0.270901**	**0.2700009**
0.4	0.064	**0.48**	**2.68**	**0.61**	**0.4921**	**0.481201**	**0.4800012**
0.5	0.125	**0.75**	**3.25**	**0.91**	**0.7651**	**0.751501**	**0.7500015**
0.6	0.216	**1.08**	**3.88**	**1.27**	**1.0981**	**1.081801**	**1.0800018**
0.7	0.343	**1.47**	**4.57**	**1.69**	**1.4911**	**1.472101**	**1.4700021**
0.8	0.512	**1.92**	**5.32**	**2.17**	**1.9441**	**1.922401**	**1.9200024**
0.9	0.729	**2.43**	**6.13**	**2.71**	**2.4571**	**2.432701**	**2.4300027**
1	1	**3**	**7**	**3.31**	**3.0301**	**3.003001**	**3.000003**
1.1	1.331	**3.63**	**7.93**	**3.97**	**3.6631**	**3.633301**	**3.6300033**
1.2	1.728	**4.32**	**8.92**	**4.69**	**4.3561**	**4.323601**	**4.3200036**
1.3	2.197	**5.07**	**9.97**	**5.47**	**5.1091**	**5.073901**	**5.070003899**
1.4	2.744	**5.88**	**11.08**	**6.31**	**5.9221**	**5.884201**	**5.8800042**
1.5	3.375	**6.75**	**12.25**	**7.21**	**6.7951**	**6.754501**	**6.750004499**
1.6	4.096	**7.68**	**13.48**	**8.17**	**7.7281**	**7.684801**	**7.680004799**
1.7	4.913	**8.67**	**14.77**	**9.19**	**8.7211**	**8.675101**	**8.6700051**
1.8	5.832	**9.72**	**16.12**	**10.27**	**9.7741**	**9.725401**	**9.720005399**
1.9	6.859	**10.83**	**17.53**	**11.41**	**10.8871**	**10.8357**	**10.8300057**
2	8	**12**	**19**	**12.61**	**12.0601**	**12.006**	**12.000006**

How to improve accuracy?: Average

$$\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x)}{h}$$ **Asymmetric equation with respect to '*x*'**

Error: $$\frac{f(x+h)-f(x)}{h} = \frac{df(x)}{dx} \boxed{+\frac{1}{2}\frac{d^2f(x)}{dx^2}h + \frac{1}{3!}\frac{d^3f(x)}{dx^3}h^2 + O(h^3)}$$

1st order error (hに関して一次の誤差)

Average => Symmetric formula: Three-point formula (3点公式, 中点則)

$$\frac{df(x)}{dx} \sim \left[\frac{f(x+h)-f(x)}{h} + \frac{f(x)-f(x-h)}{h}\right]/2 = \boldsymbol{\frac{f(x+h)-f(x-h)}{2h}}$$

$$f(x+h) = f(x) + \frac{df(x)}{dx}h + \frac{1}{2}\frac{d^2f(x)}{dx^2}h^2 + \frac{1}{3!}\frac{d^3f(x)}{dx^3}h^3 + O(h^4)$$

$$f(x-h) = f(x) - \frac{df(x)}{dx}h + \frac{1}{2}\frac{d^2f(x)}{dx^2}h^2 - \frac{1}{3!}\frac{d^3f(x)}{dx^3}h^3 + O(h^4)$$

Error: $$\frac{f(x+h)-f(x-h)}{2h} = \frac{df(x)}{dx} + \boxed{\frac{1}{3!}\frac{d^3f(x)}{dx^3}\underline{h^2} + O(h^3)}$$

2nd order error

$\propto \boldsymbol{h^2}$ (二次の誤差)

How to improve accuracy?: Take average

$$\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x)}{h} \qquad \frac{df(x)}{dx} \sim \frac{f(x)-f(x-h)}{h}$$

Asymmetric equations with respect to '*x*'

$$\frac{df(x)}{dx} \sim \left[\frac{f(x+h)-f(x)}{h} + \frac{f(x)-f(x-h)}{h}\right]/2 = \frac{f(x+h)-f(x-h)}{2h}$$

Symmetric equation, better

***f*(*x*) = x^3 *df*(*x*)/*dx* = 3x^2**

		h=	1	1	0.01	0.01
x	**f(x)**	**df(x)/dx**	**(f(x+h)-f(x))/h**	**(f(x+h)-f(x-h))/(2h)**	**(f(x+h)-f(x))/h**	**(f(x+h)-f(x-h))/(2h)**
0	0	**0**	**1**	**1**	**0.0001**	**0.0001**
0.2	0.008	**0.12**	**1.72**	**1.12**	**0.1261**	**0.1201**
0.4	0.064	**0.48**	**2.68**	**1.48**	**0.4921**	**0.4801**
0.6	0.216	**1.08**	**3.88**	**2.08**	**1.0981**	**1.0801**
0.8	0.512	**1.92**	**5.32**	**2.92**	**1.9441**	**1.9201**
1	1	**3**	**7**	**4**	**3.0301**	**3.0001**
1.2	1.728	**4.32**	**8.92**	**5.32**	**4.3561**	**4.3201**
1.4	2.744	**5.88**	**11.08**	**6.88**	**5.9221**	**5.8801**
1.6	4.096	**7.68**	**13.48**	**8.68**	**7.7281**	**7.6801**
1.8	5.832	**9.72**	**16.12**	**10.72**	**9.7741**	**9.7201**
2	8	**12**	**19**	**13**	**12.0601**	**12.0001**

Higher order formula

Three-point formula (3点公式)

$$\begin{aligned} f'(a) &= \frac{1}{h}\left\{\frac{1}{2}f(a+h)-\frac{1}{2}f(a-h)\right\} \\ &+ \frac{1}{6}f^{(3)}(a)\underline{h^{2}}+\cdots \end{aligned}$$

Five-point formula (5点公式)

$$\begin{aligned} f'(a) &= \frac{1}{h}\left\{-\frac{1}{12}f(a+2h)+\frac{2}{3}f(a+h)-\frac{2}{3}f(a-h)+\frac{1}{12}f(a-2h)\right\} \\ &+ \frac{1}{30}f^{(5)}(a)\underline{h^{4}}+\cdots \end{aligned}$$

Seven-point formula (7点公式)

$$\begin{aligned} f'(a) &= \frac{1}{h}\left\{\frac{1}{60}f(a+3h)-\frac{3}{20}f(a+2h)+\frac{3}{4}f(a+h)-\frac{3}{4}f(a-h)\right. \\ &+ \left.\frac{3}{20}f(a-2h)-\frac{1}{60}f(a-3h)\right\} \\ &+ \frac{1}{140}f^{(7)}(a)\underline{h^{6}}+\cdots \end{aligned}$$

$$\frac{d}{dx}\exp(x)\Big|_{x=1}$$

Numerical error

Analytic solution (解析解)**:**

exp(1.0) = 2.71828182845905

N_{div}	*h*	2-point	3-point	5-point	7-point
1	**0.5**	8.09E-01	1.15E-01	-5.83E-03	3.18E-04
2	**0.25**	3.70E-01	2.84E-02	-3.57E-04	4.80E-06
3	**0.125**	1.77E-01	7.08E-03	-2.22E-05	7.43E-08
4	**0.0625**	8.67E-02	1.77E-03	-1.38E-06	1.16E-09
5	**0.03125**	4.29E-02	4.42E-04	-8.64E-08	1.81E-11
6	**0.015625**	2.13E-02	1.11E-04	-5.40E-09	2.64E-13
7	**0.007813**	1.06E-02	2.77E-05	-3.38E-10	4.44E-15
8	**0.003906**	5.32E-03	6.91E-06	-2.11E-11	-7.90E-14
9	**0.001953**	2.66E-03	1.73E-06	-1.37E-12	-3.51E-14
10	**0.000977**	1.33E-03	4.32E-07	-1.23E-13	-3.65E-13
11	**0.000488**	6.64E-04	1.08E-07	-8.42E-13	-5.70E-13
12	**0.000244**	3.32E-04	2.70E-08	-2.36E-13	7.04E-13
13	**0.000122**	1.66E-04	6.75E-09	1.28E-12	5.52E-13
14	**6.1E-05**	8.30E-05	1.69E-09	-2.36E-13	-1.93E-12
15	**3.05E-05**	4.15E-05	4.19E-10	-5.09E-12	-1.69E-12
16	**1.53E-05**	2.07E-05	1.06E-10	-7.51E-12	1.63E-11
17	**7.63E-06**	1.04E-05	1.92E-11	-1.48E-11	3.64E-12
18	**3.81E-06**	5.18E-06	-9.94E-12	-4.87E-11	-9.94E-12
19	**1.91E-06**	2.59E-06	-9.94E-12	-2.93E-11	-2.18E-12

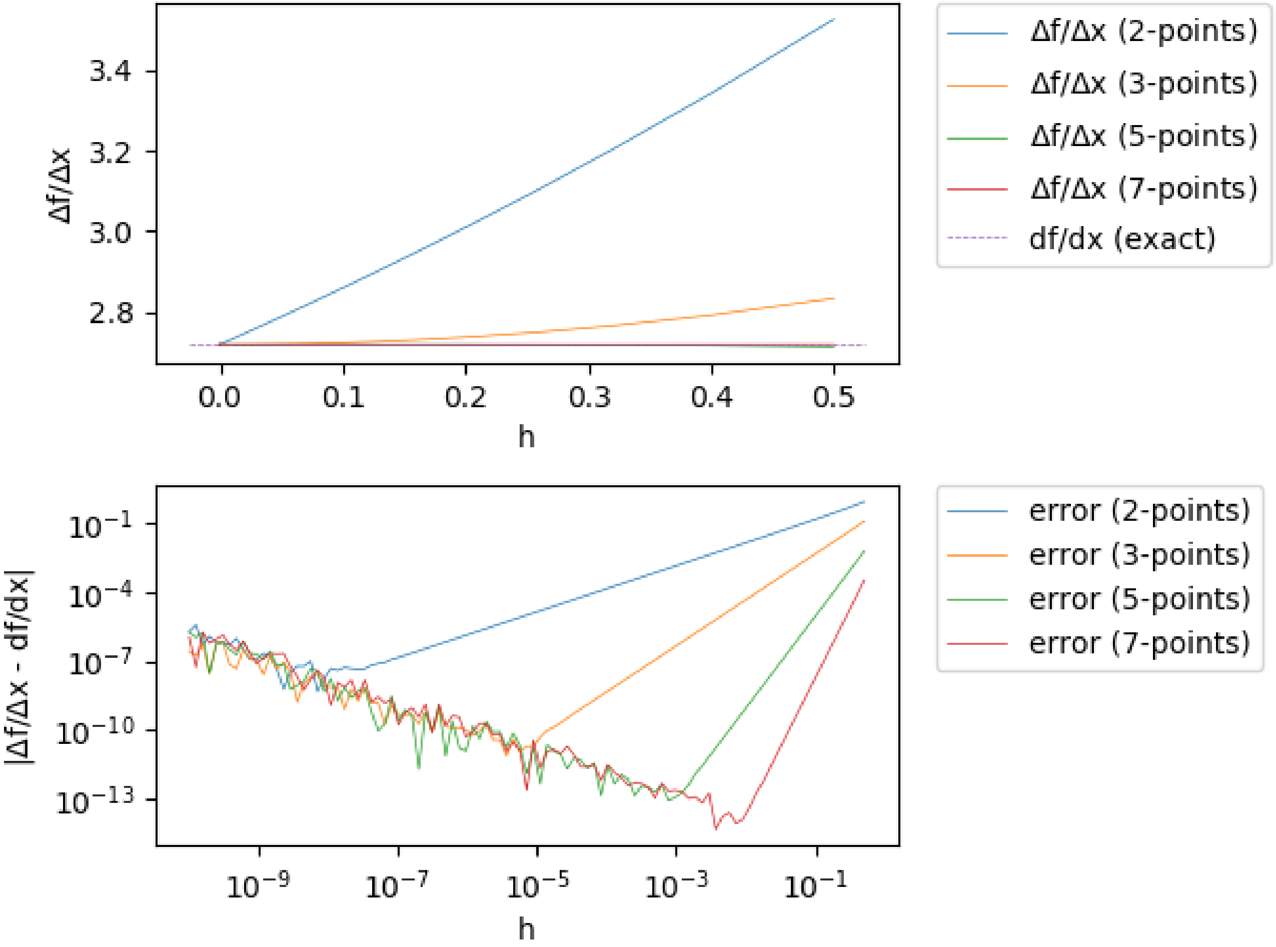
Program: diff_order.py

$$\frac{d}{dx}\exp(x)\bigg|_{x=1}$$

Analytic solution (解析解)**:**

$$\frac{d}{dx}\exp(x)\Big|_{x=1} = \exp(1.0) = 2.71828182845905$$

run: python diff_order.py



Richardson extrapolation differentiation

(リチャードソン補外)

森正武，FORTRAN 77 数値計算プログラミング、岩波書店 (1987年増補版)

- **Start from the three-point formula (中点則), and then iteratively repeat the following formula that updates the calculation precision until a required precision will be satisfied.**

(中点則から出発し、高次の微分に相当する公式を自動的に適用し、
要求精度を満たすまで繰り返す)

1. **Calc by three-point formula $D_0^{(0)} = (f(x+h) - f(x\text{-}h)) / (2h)$ at the x mesh $h = h_0$.**
2. **Reduce the mesh to a half $h_\mathrm{k} = (1/2)^\mathrm{k} h$, and the calculate $D_0^{(\mathrm{k})}$ by the three-point fomula.**
3. **Calculate next quantity**

$$D_m^{(k)} = \frac{4^m D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1}$$

4. **Iteration will be terminated if $|D_\mathrm{m}^{(0)} - D_\mathrm{m\text{-}1}^{(0)}|$ becomes smaller than the required precision.**

Numerial error

$$\left.\frac{d}{dx}\exp(x)\right|_{x=1}$$

Analytic solution (解析解)**:**

exp(1) = 2.71828182845905

N_{div}	*h*	2-point	3-point	5-point	7-point	Richardson extrapolation		
1	**0.5**	8.09E-01	1.15E-01	-5.83E-03	3.18E-04			
2	**0.25**	3.70E-01	2.84E-02	-3.57E-04	4.80E-06	-3.57E-04		
3	**0.125**	1.77E-01	7.08E-03	-2.22E-05	7.43E-08			
4	**0.0625**	8.67E-02	1.77E-03	-1.38E-06	1.16E-09	2.06E-09		
5	**0.03125**	4.29E-02	4.42E-04	-8.64E-08	1.81E-11			
6	**0.015625**	2.13E-02	1.11E-04	-5.40E-09	2.64E-13			
7	**0.007813**	1.06E-02	2.77E-05	-3.38E-10	4.44E-15			
8	**0.003906**	5.32E-03	6.91E-06	-2.11E-11	-7.90E-14	-1.38E-14		
9	**0.001953**	2.66E-03	1.73E-06	-1.37E-12	-3.51E-14			
10	**0.000977**	1.33E-03	4.32E-07	-1.23E-13	-3.65E-13			
11	**0.000488**	6.64E-04	1.08E-07	-8.42E-13	-5.70E-13			
12	**0.000244**	3.32E-04	2.70E-08	-2.36E-13	7.04E-13			
13	**0.000122**	1.66E-04	6.75E-09	1.28E-12	5.52E-13			
14	**6.1E-05**	8.30E-05	1.69E-09	-2.36E-13	-1.93E-12			
15	**3.05E-05**	4.15E-05	4.19E-10	-5.09E-12	-1.69E-12			
16	**1.53E-05**	2.07E-05	1.06E-10	-7.51E-12	1.63E-11	-3.11E-15		
17	**7.63E-06**	1.04E-05	1.92E-11	-1.48E-11	3.64E-12			
18	**3.81E-06**	5.18E-06	-9.94E-12	-4.87E-11	-9.94E-12	4.52E-13		
19	**1.91E-06**	2.59E-06	-9.94E-12	-2.93E-11	-2.18E-12	1.69E-12		

For non-constant $h_i = x_{i+1} - x_i$

x	y
x_{-1}	y_{-1}
x_0	y_0
x_1	y_1

Rough method: Take average

(maybe good but not best)

$$y'(x_0) = \frac{1}{2}\left[\frac{y_1 - y_0}{x_1 - x_0} + \frac{y_0 - y_{-1}}{x_0 - x_{-1}}\right]$$

Polynomial method: Lagrange polynomial (ラングランジュ多項式)

$$P_{n-1}(x) = f(x_0)\phi_0(x) + f(x_1)\phi_1(x) + \cdots f(x_{n-1})\phi_{n-1}(x)$$

$$\phi_i(x) = \frac{\prod_{k\neq i}^{n-1}(x - x_k)}{\prod_{k\neq i}^{n-1}(x_i - x_k)} = \prod_{k\neq i}^{n-1}\frac{(x - x_k)}{(x_i - x_k)}$$

$$y(x) = y_{-1}\frac{(x - x_0)(x - x_1)}{(x_{-1} - x_0)(x_{-1} - x_1)} + y_0\frac{(x - x_{-1})(x - x_1)}{(x_0 - x_{-1})(x_0 - x_1)} + y_1\frac{(x - x_{-1})(x - x_0)}{(x_1 - x_{-1})(x_1 - x_0)}$$

$$y'(x) = y_{-1}\frac{2x - (x_0 + x_1)}{(x_{-1} - x_0)(x_{-1} - x_1)} + y_0\frac{2x - (x_{-1} + x_1)}{(x_0 - x_{-1})(x_0 - x_1)} + y_1\frac{2x - (x_{-1} + x_0)}{(x_1 - x_{-1})(x_1 - x_0)}$$

Second differential (二階微分)

If calculate 2nd differential using forward differences both for the 1st and the 2nd differentials …

(一階微分を前進差分で計算してから二階微分を前進差分で計算すると･･･)

$$\frac{d^2x(t)}{dt^2} = \frac{\frac{dx}{dt}(t+\Delta t) - \frac{dx}{dt}(t)}{\Delta t}$$

$$\sim \frac{\frac{x(t+2\Delta t)-x(t+\Delta t)}{\Delta t} - \frac{x(t+\Delta t)-x(t)}{\Delta t}}{\Delta t} = \frac{x(t+2\Delta t)-2x(t+\Delta t)+x(t)}{\Delta t^2} \quad (1)$$

If use backward differentials only for the 1st differentials (but logically inconsistent):

$$\frac{d^2x(t)}{dt^2} \sim \frac{\frac{x(t+\Delta t)-x(t)}{\Delta t} - \frac{x(t)-x(t-\Delta t)}{\Delta t}}{\Delta t}$$

$$\boldsymbol{\frac{d^2x(t)}{dt^2} \sim \frac{x(t+\Delta t)-2x(t)+x(t-\Delta t)}{\Delta t^2}} \quad \mathbf{(2)}$$

Symmetric formula w.r.t. *t* + Δ*t* and *t* – Δ*t* is obtained

(*t* + Δ*t*, *t* – Δ*t* について対称になる式が取れ、精度が上がる)

Note: *x* value of eq. (1) is shifted by one *Δt* from eq. (2)

(eq.(1)では、横軸が$\boldsymbol{\Delta t}$ひとつ分ずれているために精度が落ちる)

Second differential by central differences

$$\frac{d^2x(t)}{dt^2} = \frac{\frac{dx}{dt}(t+\Delta t) - \frac{dx}{dt}(t-\Delta t)}{2\Delta t}$$

$$\sim \frac{\frac{x(t+2\Delta t)-x(t)}{2\Delta t} - \frac{x(t)-x(t-2\Delta t)}{2\Delta t}}{2\Delta t} = \frac{x(t+2\Delta t)-2x(t)+x(t-2\Delta t)}{(2\Delta t)^2}$$

$$\boldsymbol{\frac{d^2x(t)}{dt^2} = \frac{x(t+\Delta t')-2x(t)+x(t-\Delta t')}{\Delta t'^2}}$$

Symmetric formula w.r.t. $t+\Delta t$ and $t-\Delta t$ is obtained

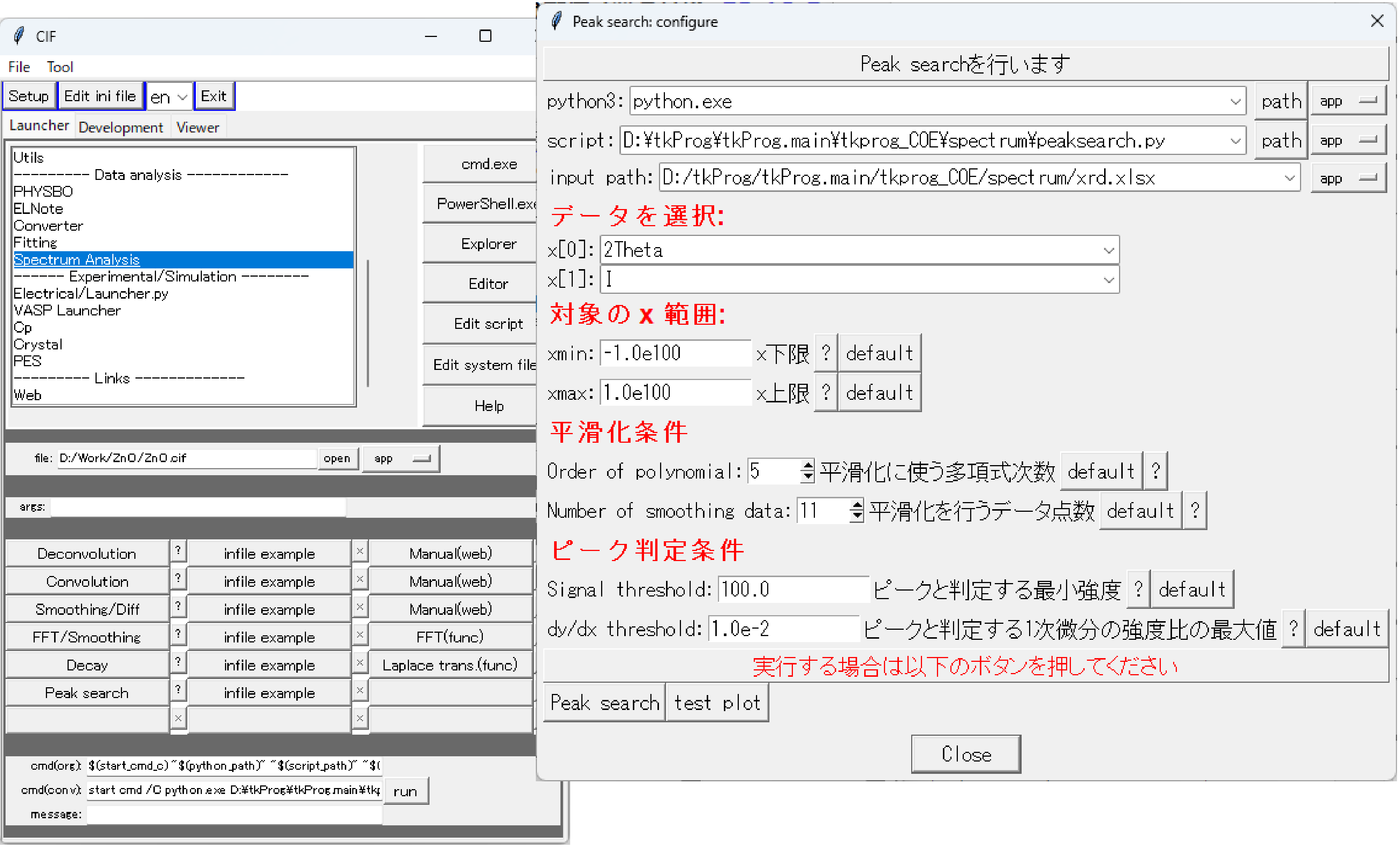
($t+\Delta t$, $t-\Delta t$ について対称になる式が取れ、精度が上がる)

Note: *x* value of eq. (1) is shifted by one Δt from eq. (2)

(eq.(1)では、横軸がΔtひとつ分ずれているために精度が落ちる)

Q: Peak search program

- http://conf.msl.titech.ac.jp/D2MatE/PeakSearch/PeakSearch.html



CIF

File Tool

Launcher / Development / Viewer

```
Utils
--------- Data analysis ------------
PHYSBO
ELNote
Converter
Fitting
Spectrum Analysis
------ Experimental/Simulation --------
Electrical/Launcher.py
VASP Launcher
Cp
Crystal
PES
--------- Links -------------
Web
```

file: D:/Work/ZnO/ZnO.cif

args:

Deconvolution	?	infile example	×	Manual(web)
Convolution	?	infile example	×	Manual(web)
Smoothing/Diff	?	infile example	×	Manual(web)
FFT/Smoothing	?	infile example	×	FFT(func)
Decay	?	infile example	×	Laplace trans.(func)
Peak search	?	infile example	×	
	×		×	

cmd(org): $(start_cmd_c) "$(python_path)" "$(script_path)" "$(

cmd(conv): start cmd /C python.exe D:¥tkProg¥tkProg.main¥tkp

message:

Peak search: configure

Peak searchを行います

python3: python.exe

script: D:¥tkProg¥tkProg.main¥tkprog_COE¥spectrum¥peaksearch.py

input path: D:/tkProg/tkProg.main/tkprog_COE/spectrum/xrd.xlsx

データを選択:

x[0]: 2Theta

x[1]: I

対象の x 範囲:

xmin: -1.0e100 x下限

xmax: 1.0e100 x上限

平滑化条件

Order of polynomial: 5 平滑化に使う多項式次数

Number of smoothing data: 11 平滑化を行うデータ点数

ピーク判定条件

Signal threshold: 100.0 ピークと判定する最小強度

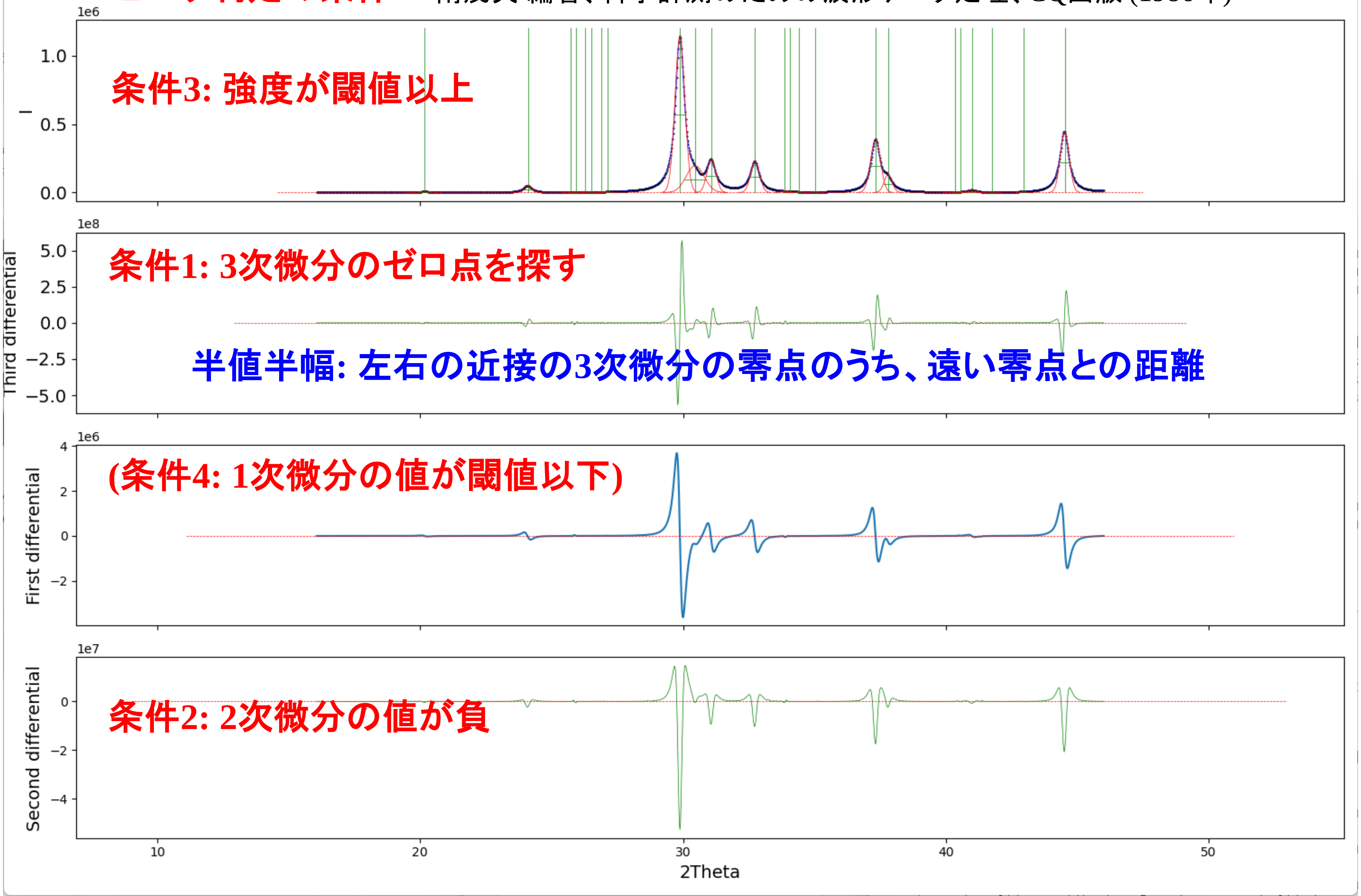
dy/dx threshold: 1.0e-2 ピークと判定する1次微分の強度比の最大値

実行する場合は以下のボタンを押してください

A: Peak search program

[tkProg]¥tkprog_base¥spectrum¥peak_search. py

ピーク判定の条件　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)



条件3: 強度が閾値以上

条件1: 3次微分のゼロ点を探す

半値半幅: 左右の近接の3次微分の零点のうち、遠い零点との距離

(条件4: 1次微分の値が閾値以下)

条件2: 2次微分の値が負

Numeral integration (quadrature)

数値積分 (求積)

Numerical integration (数値積分)

How to calculate $F(x) = \int_{x_0}^{x} g(x')dx'$ by computer

Replace integral with summation of small mesh area

(積分 を和 で置き換える)

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{x_i=x} g(x_i)h$$

Derivation from difference approximation (差分式からの導出):

$$\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x)}{h} \quad \Rightarrow \quad g(x) \sim \frac{F(x+h)-F(x)}{h}$$

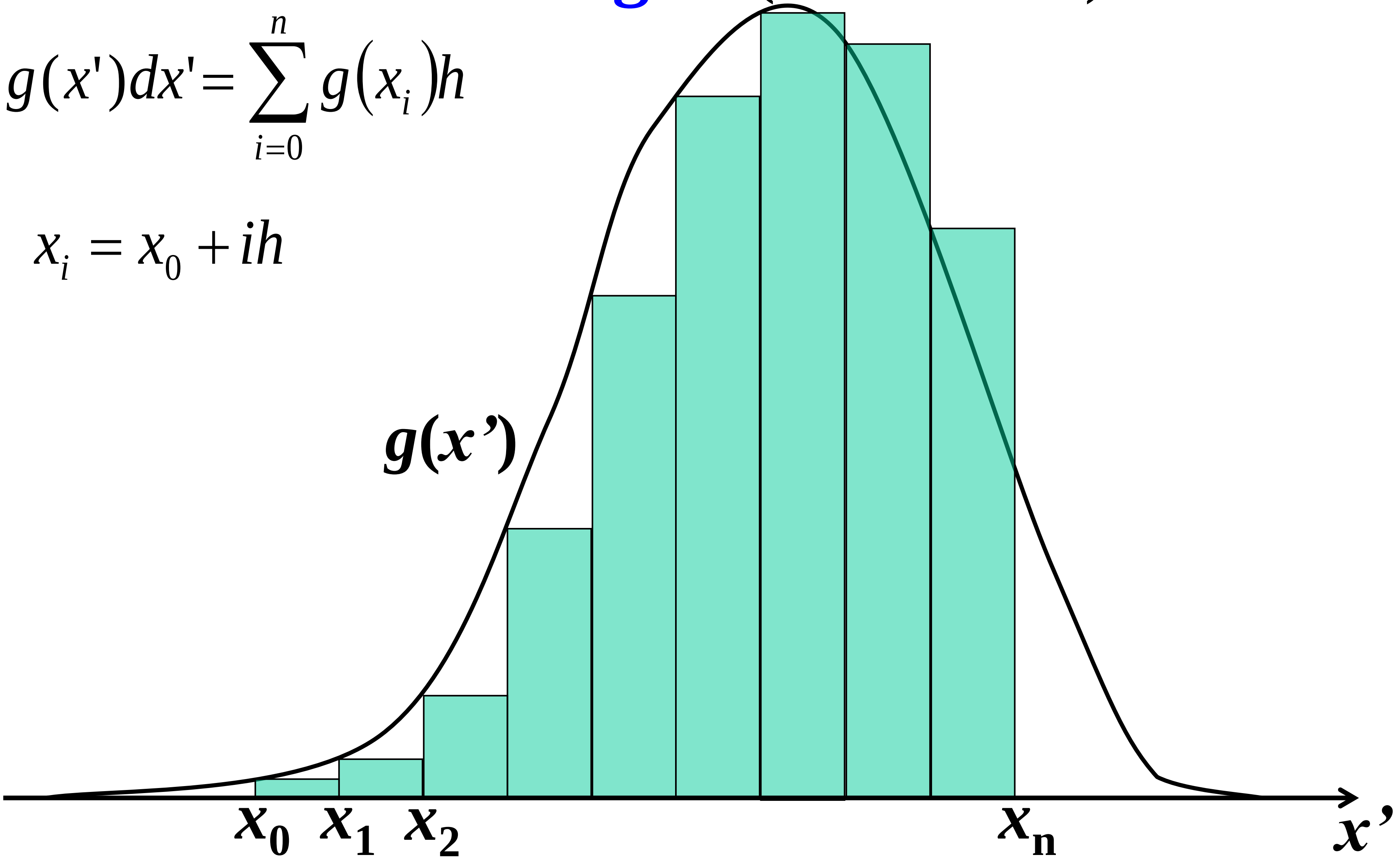
$$F(x+h) = F(x) + g(x)h = F(x-h) + \left[g(x) + g(x-h)\right]h$$

$$= \sum_{i=0}^{x_i=x} g(x_i)h$$

Rieman integral (Rieman積分)

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{n} g(x_i)h$$

$$x_i = x_0 + ih$$



Asymmetric formula:

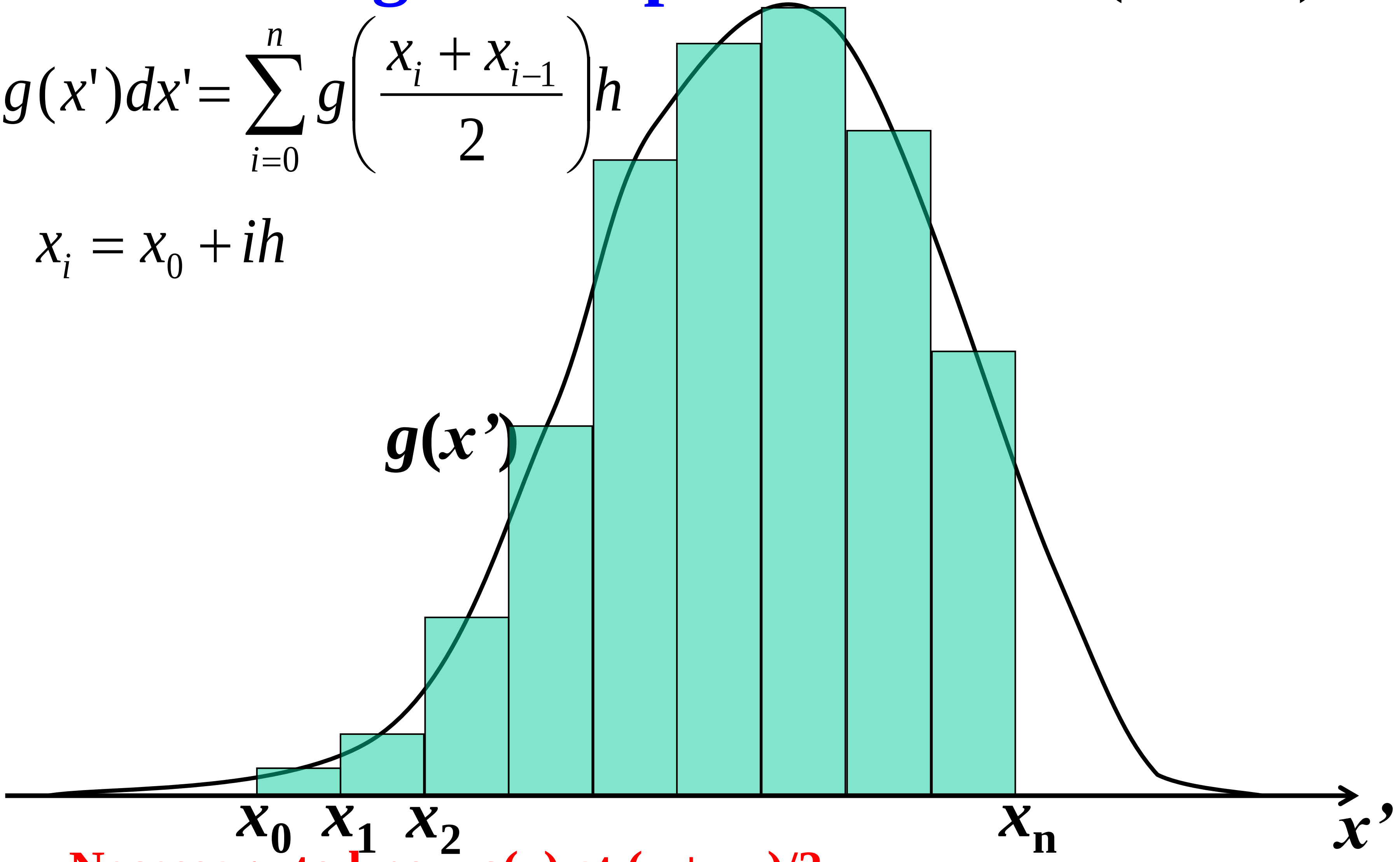
monotone increasing $g(x)$ => Underestimation (過小評価)

monotone decreasing $g(x)$ => Overestimation (過大評価)

Take average: Mid-point formula (中点則)

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{n} g\left(\frac{x_i + x_{i-1}}{2}\right)h$$

$$x_i = x_0 + ih$$



Necessary to know $g(x)$ at $(x_i+x_{i-1})/2$.

=> Unavailable for $g(x)$ given only by numerical data

($g(x)$が数値データで与えられている場合は使えない)

Trapezoid formula (台形公式)

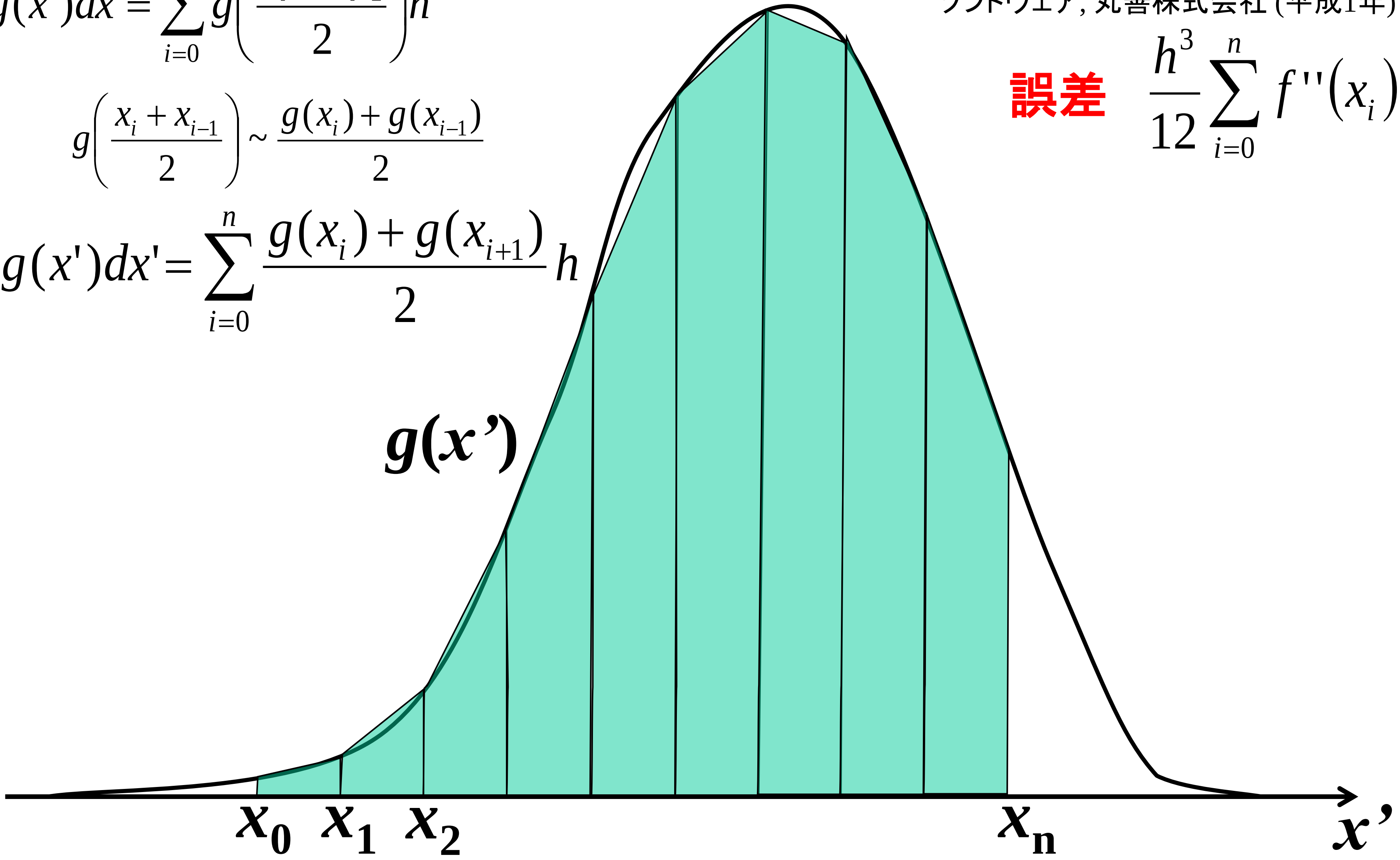
渡部力ら監修、Fortran77による数値計算
ソフトウェア, 丸善株式会社 (平成1年)

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{n} g\left(\frac{x_i + x_{i-1}}{2}\right)h$$

$$g\left(\frac{x_i + x_{i-1}}{2}\right) \sim \frac{g(x_i) + g(x_{i-1})}{2}$$

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{n} \frac{g(x_i) + g(x_{i+1})}{2}h$$

誤差 $$\frac{h^3}{12}\sum_{i=0}^{n} f''(x_i)$$



Simpson formula

1. Approximate by $g(x_i) \sim g(x_1) + a_1(x_i - x_1) + a_2(x_i - x_1)^2$**,**
and determine a_i so as to reproduce $f(x_0)$, $f(x_1)$, and $f(x_2)$.
($x_i = x_1 - h$, x_1, $x_1 + h$)

2. Integrate the above approximation analytically for a range $x = x_0 \sim x_0 + 2h$:

$$\int_{x_0}^{x_2} g(x')dx' \sim \frac{1}{3}h\left[g(x_0) + 4g(x_1) + g(x_2)\right]$$

3. For multiply divided range ($x = x_0 \sim x_n = x_0 + nh$):

$$\int_{x_0}^{x_n} g(x')dx' \sim \frac{h}{3}\left[g(x_0) + 4g(x_1) + 2g(x_2) + 4g(x_3) + 2g(x_4) + \cdots + g(x_n)\right]$$

Derivation of the Simpson formula

1. Approximate by $g(x_i) \sim g(x_1) + a_1(x_i - x_1) + a_2(x_i - x_1)^2$**,**
and determine a_i so as to reproduce $f(x_0)$, $f(x_1)$, and $f(x_2)$.
($x_i = x_1 - h$, x_1, $x_1 + h$)

$$g(x_0) \sim g(x_1) - a_1 h + a_2 h^2$$
$$g(x_2) \sim g(x_1) + a_1 h + a_2 h^2$$

$$\Longrightarrow \quad a_1 = \frac{g(x_2) - g(x_0)}{2h} \qquad a_2 = \frac{g(x_2) - 2g(x_1) + g(x_0)}{2h^2}$$

$$\int_{x_0}^{x_2} g(x')dx' \sim g(x_1)x_2 + \frac{1}{2}\frac{g(x_2) - g(x_0)}{2h}(x_2 - x_1)^2 + \frac{1}{3}\left[\frac{g(x_2) - 2g(x_1) + g(x_0)}{2h^2}\right](x_2 - x_1)^3$$
$$- \left\{ g(x_1)x_0 + \frac{1}{2}\frac{g(x_2) - g(x_0)}{2h}(x_0 - x_1)^2 + \frac{1}{3}\left[\frac{g(x_2) - 2g(x_1) + g(x_0)}{2h^2}\right](x_0 - x_1)^3 \right\}$$
$$= 2g(x_1)h + 2\left[\frac{g(x_2) - 2g(x_1) + g(x_0)}{6}\right]h$$
$$= \frac{1}{3}\left[g(x_2) + 4g(x_1) + g(x_0)\right]$$

片岡勲 他、数値解析入門, コロナ社

Error $\leq \dfrac{nh^5}{180}\left|f^{(4)}(x_i)\right|$

Comparison of numerical integration

$g(x) = x^2$

$$\int_0^x g(x')dx' = \frac{1}{3}x^3$$

x	g(x)	Exact	Rieman	Trapezoid	Simpson
0	0	0	0	0	0
0.2	0.04	0.0027	0	0.004	
0.4	0.16	0.0213	0.008	0.024	0.021333

Series of Newton-Cotes formula

- **Trapezoid formula** **(台形則)**

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1+\frac{1}{2}f_2\right]+O(\underline{h^3}f'')$$

- **Simpson formula** **(Simpson則)**

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1+\frac{4}{3}f_2+\frac{1}{3}f_3\right]+O(\underline{h^5}f^{(4)})$$

- **Simpson’s 3/8 formula** **(Simpsonの3/8則)**

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1+\frac{9}{8}f_2+\frac{9}{8}f_3+\frac{3}{8}f_4\right]+O(\underline{h^5}f^{(4)})$$

- **Bode/Boole-Vilarceau formula** **(Bode/Boole則)**

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1+\frac{64}{45}f_2+\frac{24}{45}f_3+\frac{64}{45}f_4+\frac{14}{45}f_5\right]+O(\underline{h^7}f^{(6)})$$

Rieman/Trapezoid formula are better than Simpson formula for infinite-range integration

Simpson則より単純和/台形則の方が良い

$$\int_{-\infty}^{\infty} g(x')dx' \sim \frac{h}{3}[g(x_0) + 4g(x_1) + 2g(x_2) + 4g(x_3) + 2g(x_4) + \cdots + g(x_n)]$$

For infinite-range integration (-∞ ~ ∞), x_0 and x_n are not essential.

$$\int_{-\infty}^{\infty} g(x')dx' \sim \frac{h}{3}[g(x_{-1}) + 4g(x_0) + 2g(x_1) + 4g(x_2) + 2g(x_3) + \cdots + g(x_{n-1})]$$

$$\int_{-\infty}^{\infty} g(x')dx' \sim \frac{h}{3}[\qquad\qquad g(x_0) + 4g(x_1) + 2g(x_2) + 4g(x_3) + \cdots \qquad\qquad + g(x_n)]$$

also provides the essentially the same result.

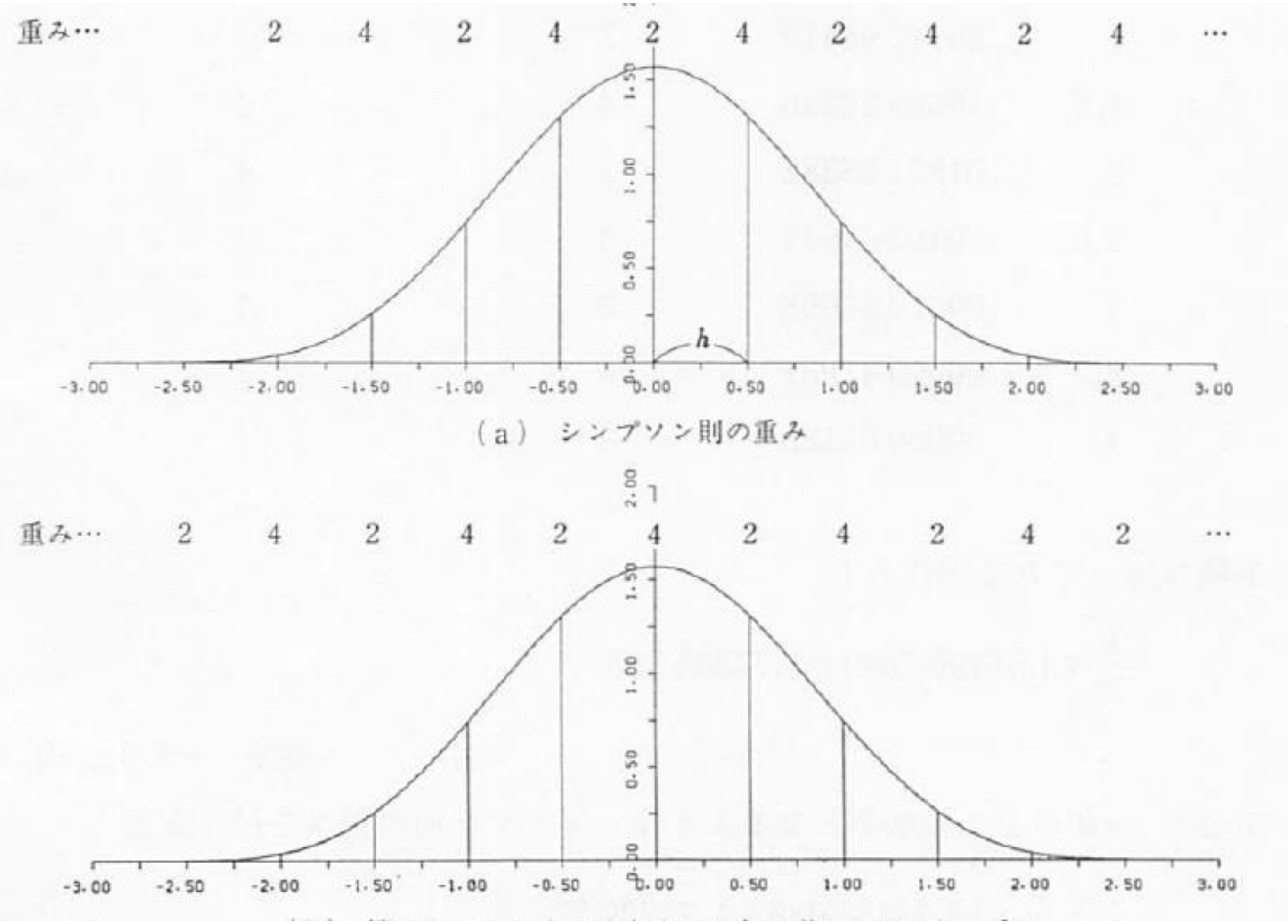
$$\int_{x_0}^{x_n} g(x')dx' \sim \frac{h}{3}\left[0.5g(x_{-1}) + 2.5g(x_0) + 3g(x_1) + 3g(x_2) + 3g(x_3) + 3g(x_4) + \cdots + 0.5g(x_n)\right]$$

Considering g(x_{-1}) and $g(x_n)$ are negligible for infinite integration leads to

$$\int_{x_0}^{x_n} g(x')dx' \sim h\left[g(x_1) + g(x_2) + g(x_3) + g(x_4) + \cdots + g(x_{n-2})\right]$$

, which is the same as the Rieman sum and the trapezoid formula.

Simpson法より単純和(台形則)の方が良い



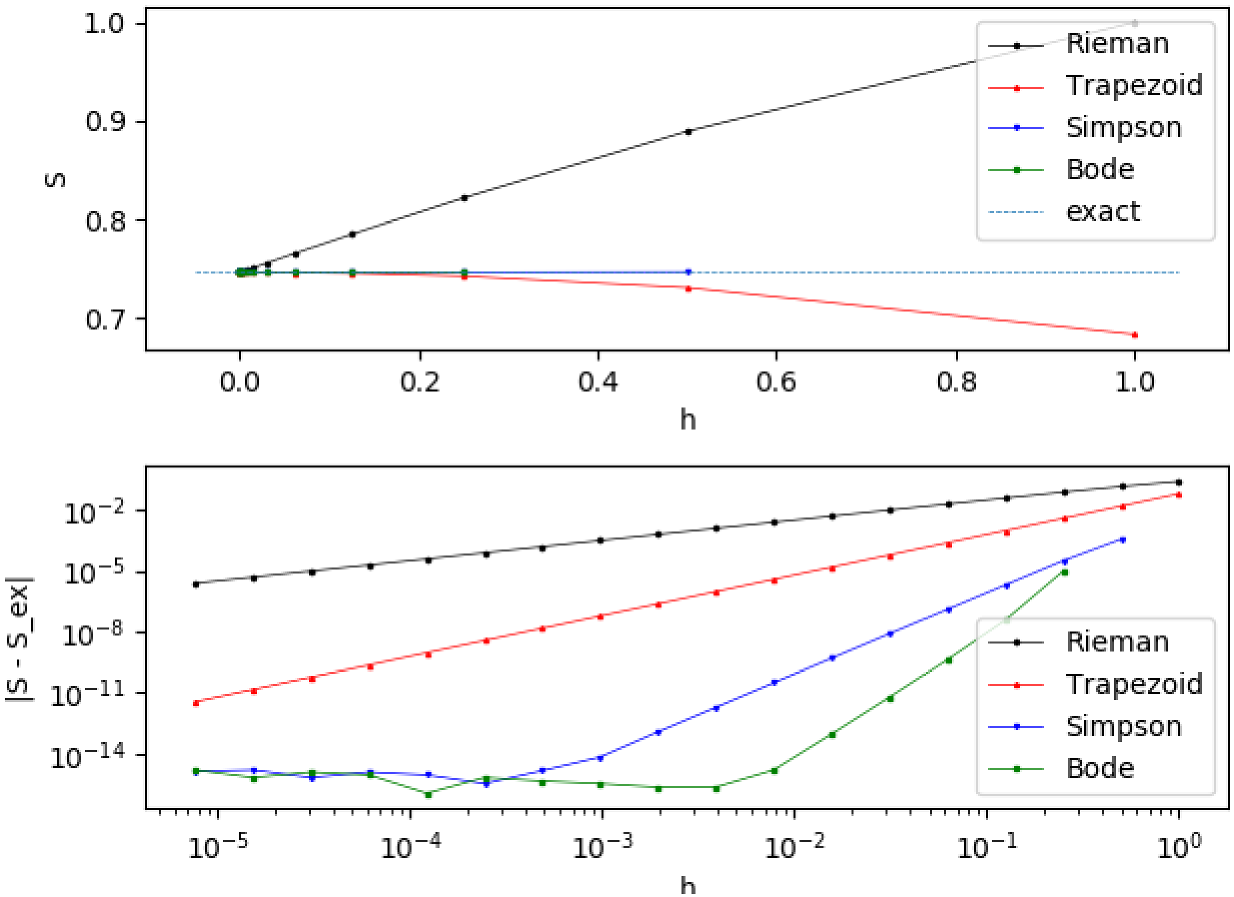
（a） シンプソン則の重み

Program: integ_order_h.py

$g(x') = \exp(-x^2), \int_{x0}^{x1} g(x')dx' = \text{erf}(x_1) \; - \text{erf}(x_0)$

$[x_0, x_1] = [0, 1.0]$, exact = 0.746824132812427

Run: **python integ_order_h.py 0 1 18 gauss**



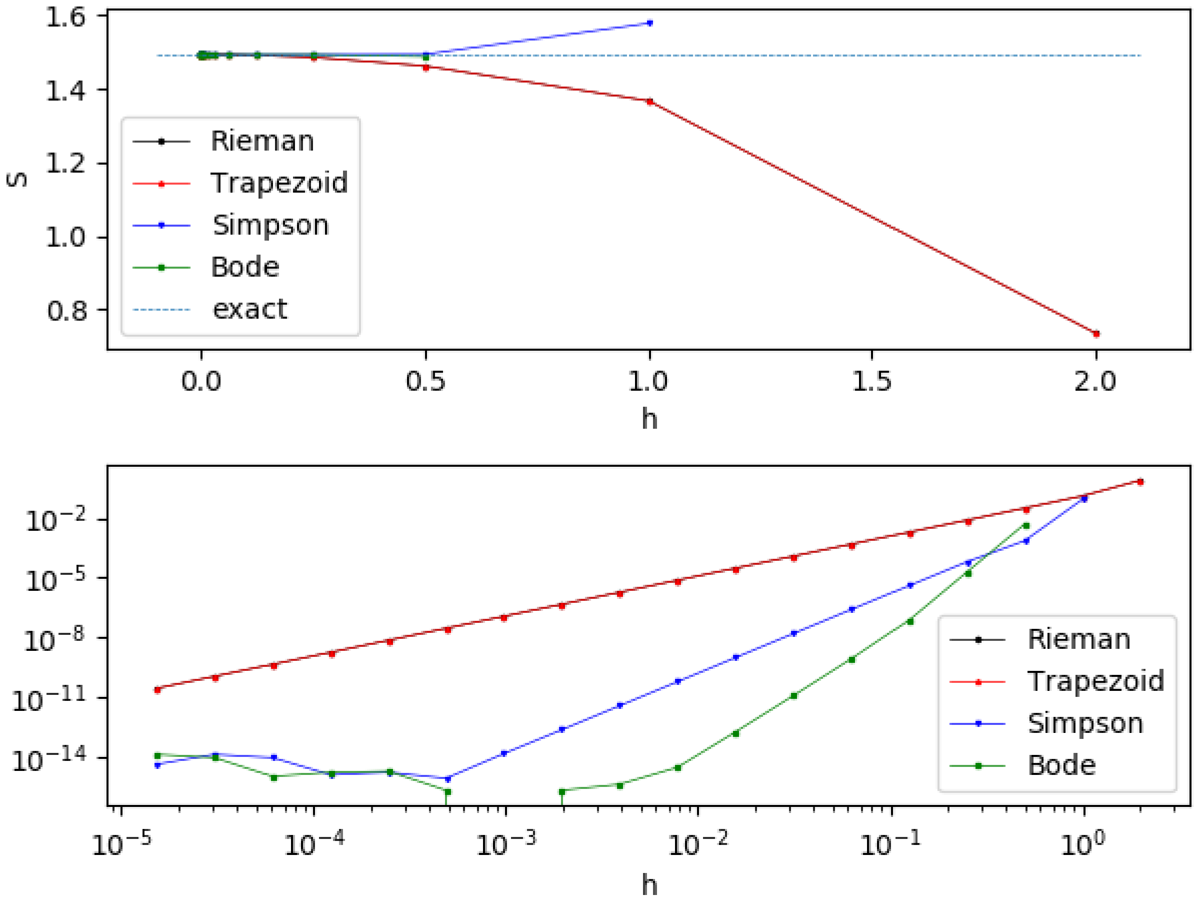
Trapezoid approx. is better than Rieman sum for asymmetric function over **finite range**

Program: integ_order_h.py

$g(x') = \exp(-x^2), \int_{x0}^{x1} g(x')dx' = \mathrm{erf}(x_1) \; - \mathrm{erf}(x_0)$

$[x_0, x_1]$ = [-1.0, 1.0], exact = 1.493648265624854

Run: **python integ_order_h.py -1 1 18 gauss**



Trapezoid approx. is better than Rieman sum also for symmetric integration over **finite range**

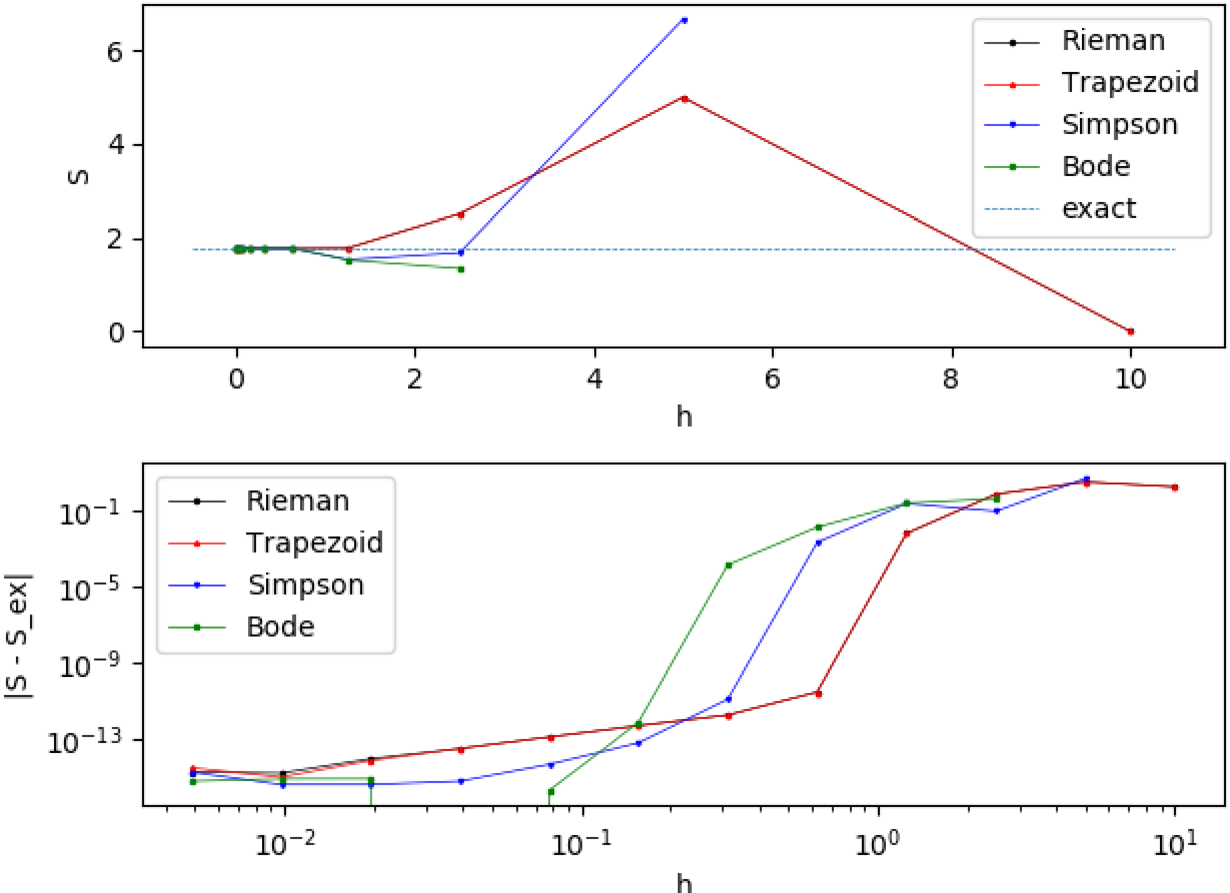
Program: integ_order_h.py

$g(x') = \exp(-x^2), \int_{x0}^{x1} g(x')dx' = \mathrm{erf}(x_1) \ - \mathrm{erf}(x_0)$

$[x_0, x_1]$ = [-5, 5], exact = 1.772453850902791 ($\sim\sqrt{\pi}$)

Note: The range [-5, 5] is virtually equivalent to infinite integration range as exp(-25) can be negligible

Run: **python integ_order_h.py -5 5 12 gauss**



Simposon method looses accuracy for integration over **infinite range**

Features of other numerical integrations

Newton-Cotes formula: Analytically integrate approximated polynomial that exactly takes *g*(*x*) with uniform integration points.

(積分範囲を等分割し、各積分点を通る多項式で近似して解析的に積分する)

- **Trapezoid formula (first order)** (台形則, 一次式)
- **Simpson formula (second/third order)** (Simpson則, 二次式、三次式)
- **Bode/Boole formula (fourth order)** (Bode/Bool則, 四次式)

Maximize precision by optimize both weights and integration points

(計算点位置も含めて精度が最大になるようにする)

(High precision, Non-uniform points (精度は高い、積分点が等間隔でない)**)**

- **Gauss-Legendre formula**
- **Gauss-Chebyshev formula**

Interpolation type (補間型) **(Better precision?)**

- **Spline integration** (スプライン積分)

Extrapolation type (補外型) **(Controlled precision)**

- **Romberg integration** (ロンバーグ積分)

Variable conversion type (変数変換型) **(better for infinite integration, anomaly points** 無限積分や特異点を含む積分に有利**)**

Gauss-Legendre method

戸田英雄, 小野令美，入門 数値計算, オーム社 (昭和58年)

- **Choose *n* integration points x_i and weights w_i so as to minimize the integration error by (2n-1) order polynomial.**
  積分区間に n個の積分点を選ぶ際、積分点と重みの 2*n*個のパラメータを*f*(*x*) が (2*n*-1)次の多項式に一致するように　決める。
- **Can integrate a function with anomaly points.**
  端点を含まないので、積分区間端に特異点があっても計算できる
- **Best accuracy for good functions in finite integration range.**
  有限区間で解析的な関数の積分では最も精度が高い

- **Integration points x_i are given as the zero points of Legendre polynomial.**
  分点はLegendre多項式の零点

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}\left(x^2 - 1\right)^n = 0$$

$$w_i = \frac{2(1 - x^2)}{(n+1)^2\left[P_{n+1}(x_i)\right]^2}$$

$$\Rightarrow$$

$$S = \sum_{i=1}^{n} f(x_i) w_i$$

Gauss-Legendre method: fractional coordinates and weights (分点と重み)

Fractional coordinates (分点)	Weight (重み係数)
Four points formula (4 点公式)	
−0.861136311594052575223946488892	0.347854845137453857373063949221
−0.339981043584856264802665759103	0.652145154862546142626936050778
+0.339981043584856264802665759103	0.652145154862546142626936050778
+0.861136311594052575223946488892	0.347854845137453857373063949221
Five points (5点公式)	
−0.906179845938663992797626878299	0.236926885056189087514264040719
−0.538469310105683091036314420700	0.478628670499366468041291514835
0	0.568888888888888888888888888888
+0.538469310105683091036314420700	0.478628670499366468041291514835
+0.906179845938663992797626878299	0.236926885056189087514264040719
Six points (6 点公式)	
−0.932469514203152027812301554493	0.171324492379170345040296142172
−0.661209386466264513661399595019	0.360761573048438607569833513837
−0.238619186093196908630501721680	0.467913934572691047389870343989
+0.238619186093196908630501721680	0.467913934572691047389870343989
+0.661209386466264513661399595019	0.360761573048438607569833513837
+0.932469514203152027812301554493	0.171324492379170345040296142172
Seven points (7 点公式)	
−0.949107912342758524526189684047	0.129484966168869693270611432679
−0.741531185599394439863864773280	0.279705391489276667901467771423
−0.405845151377397166906606412076	0.381830050505118944950369775488
0	0.417959183673469387755102040816
+0.405845151377397166906606412076	0.381830050505118944950369775488
+0.741531185599394439863864773280	0.279705391489276667901467771423
+0.949107912342758524526189684047	0.129484966168869693270611432679

Extrapolation method: Romberg integration

戸田英雄, 小野令美，入門 数値計算, オーム社 (昭和58年)

Good for finite range integration without anomaly points

・Start from the Trapezoid formula, and sequentially apply higher order Newton-Cotes precision formula.

(台形則から出発し、高次のニュートン・コーツ型に相当する公式を自動的に適用し、要求精度を満たすまで続ける)

1. **Integrate by the Trapezoid formula in [*a*, *b*] with the mesh h_0 => $S_{0,0}$**
2. **Decrease mesh to $h_1 = (1/2)h_0$ and integrate all the range => $S_{1,0}$**
3. **Decrease mesh to $h_k = (1/2)h_{k-1}$ and integrate all the range => $S_{k,0}$, and calculate $S_{k,d}$ (d = 1, 2, ･･･, k) by**

$$S_{k,d} = \frac{4^d S_{k,d-1} - S_{k-1,d-1}}{4^d - 1}$$

4. **$S_{k,k}$ will be the approximated integration values. Stop if $|S_{k,k} - S_{k-1,k-1}|$ becomes smaller than the required accuracy.**

Error of numerical integration: Monotone increasing function

$$S = \int_{-1}^{1} \exp(x)dx$$

Exact: exp(1) - exp(-1) = 2.3504023872876

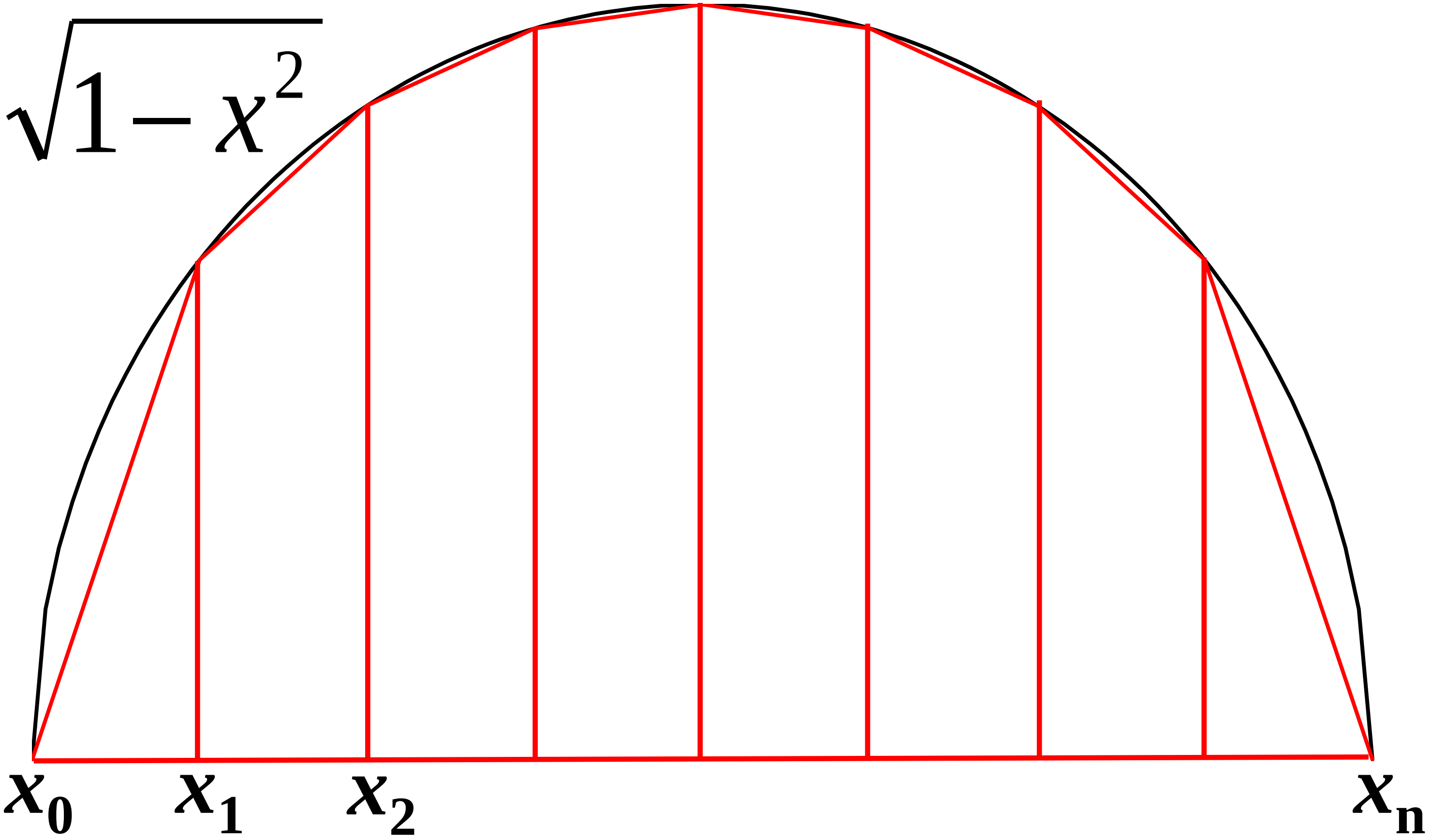
nDivide	Rieman	Trapezoid	Simpson	Simpson 3/8	Bode	Romberg	Cubic Spline	Order 3 Gauss-Legendre
1	1.61E+00	-7.36E-01				-7.36E-01		
2	9.83E-01	-1.93E-01	-1.17E-02			-1.17E-02		6.55E-05
3	6.97E-01	-8.64E-02		-5.25E-03				
4	5.39E-01	-4.88E-02	-7.92E-04		-6.85E-05	-6.85E-05	7.19E-03	1.13E-06
5	4.39E-01	-3.13E-02					3.75E-03	
6	3.70E-01	-2.17E-02	-1.59E-04	-3.53E-04			2.35E-03	1.01E-07
7	3.20E-01	-1.60E-02					1.54E-03	
8	2.82E-01	-1.22E-02	-5.06E-05		-1.18E-06	-1.07E-07	1.07E-03	1.81E-08
9	2.51E-01	-9.66E-03		-7.08E-05			7.73E-04	
10	2.27E-01	-7.83E-03	-2.08E-05				5.77E-04	4.75E-09
11	2.07E-01	-6.47E-03					4.41E-04	
12	1.90E-01	-5.44E-03	-1.00E-05	-2.25E-05	-1.05E-07		3.45E-04	1.59E-09
13	1.76E-01	-4.63E-03					2.75E-04	
14	1.64E-01	-4.00E-03	-5.43E-06				2.23E-04	6.32E-10
15	1.53E-01	-3.48E-03		-9.25E-06			1.83E-04	
16	1.44E-01	-3.06E-03	-3.18E-06		-1.88E-08	-4.21E-11	1.52E-04	2.84E-10
17	1.36E-01	-2.71E-03					1.28E-04	
18	1.28E-01	-2.42E-03	-1.99E-06	-4.46E-06			1.08E-04	1.40E-10
19	1.22E-01	-2.17E-03					9.27E-05	
20	1.16E-01	-1.96E-03	-1.30E-06		-4.95E-09		7.99E-05	7.45E-11
32						-3.55E-15		

Problem for integration with anomaly points

(特異点を含む場合の問題)

$$F(x) = \int_{x_0}^{x} g(x')dx'$$

$$g(x) = \sqrt{1 - x^2}$$



Very large errors for large |*f* ’(*x*)| / |*f* ”(*x*)|

Variable conversion type: Double exponential type formula (変数変換型: 二重指数関数型公式)

戸田英雄, 小野令美, 入門 数値計算, オーム社 (昭和58年)

Good for integral including anomaly points at the ends and for infinite range

端点に特異点のある積分や、無限積分に有効

Finite range integral is converted to the infinite range (-∞, ∞) by variable conversion

有限区間積分の場合は、変数変換により無限積分にする

Calculate by the Trapezoid formula

$$S = h \sum_{n=-\infty}^{\infty} f\left(\phi(nh)\right)\phi'\left(nh\right)$$

Iri-Moriguchi-Takasawa (IMT) formula

伊理･森口･高沢(IMT)の公式

戸田英雄, 小野令美, 入門 数値計算, オーム社 (昭和58年)

Good for finite range integral including anomaly points at the ends and for infinite range

By variable conversion (変数変換)

$$x = \phi(u) = \frac{1}{Q}\int_0^u \exp\left(-\frac{1}{t} - \frac{1}{1-t}\right)dt \qquad \phi'(u) = \frac{1}{Q}\exp\left(-\frac{1}{t} - \frac{1}{1-t}\right)$$

$$Q = \int_0^1 \exp\left(-\frac{1}{t} - \frac{1}{1-t}\right)dt \quad = 0.00702985841$$

an integral of *f*(*x*) is converted to

$$\int_0^1 f(x)dx = \int_0^1 f(\phi(u))\phi'(u)du$$

, and then calculate the integral by the Trapezoid formula

1. **Convert the range to [0, 1] by *x* = (*x*'- *a*) / (*b* - *a*)**

$$\int_a^b f(x')dx' = (b-a)\int_0^1 f(x)dx$$

2. **Calculate integ. Points x_k = ϕ(k/n) and weights w_k = ϕ'(k/n)**

3. **Calculate** $I = h\sum_{k=1}^{n-1} f(x_k)w_k$ **(*h* = (*b* – *a*)/n)**

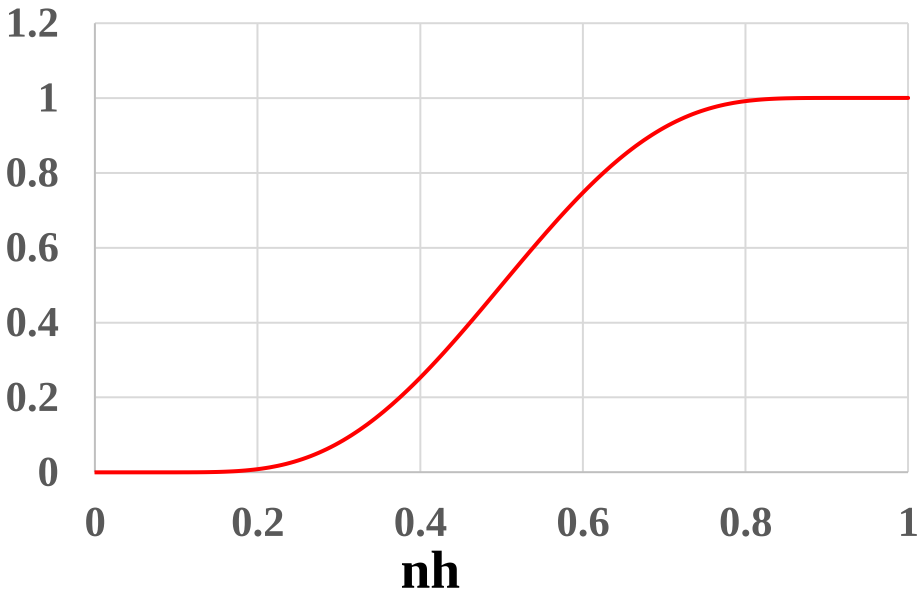
Iri-Moriguchi-Takasawa (IMT) formula

伊理･森口･高沢(IMT)の公式

$$x_n = \phi(nh) = \frac{1}{Q}\int_0^{nh} \exp\left(-\frac{1}{t} - \frac{1}{1-t}\right) dt$$

Q = 0.00702985841

戸田英雄, 小野令美，入門 数値計算, オーム社 (昭和58年)



Variable conversion type: Double exponential type formula (変数変換型: 二重指数関数型公式)

戸田英雄, 小野令美，入門 数値計算, オーム社 (昭和58年)

For $\int_{-1}^{1} f(x)dx$

$$x_n=\phi(nh)=\tanh\left[\frac{\pi}{2}\sinh(nh)\right] \qquad \phi'(nh)=\frac{\pi}{2}\frac{\cosh nh}{\cosh^2\left(\left(\pi/2\right)\sinh nh\right)}$$

For $\int_{0}^{\infty} f(x)dx$

$$x_n=\phi(nh)=\exp\left[\frac{\pi}{2}\sinh(nh)\right] \qquad \phi'(nh)=\frac{\pi}{2}\cosh nh\exp\left(\frac{\pi}{2}\sinh nh\right)$$

For $\int_{0}^{\infty} f(x)dx$ **where *f*(*x*) includes exp(-*x*) type factor**

$$x_n=\phi(nh)=\exp\left[\frac{\pi}{2}\left(nh-\exp(-nh)\right)\right] \qquad \phi'(nh)=\frac{\pi}{2}\left(1+\exp(-nh)\right)\exp\left(\frac{\pi}{2}(nh-\exp(-nh)\right)$$

For $\int_{-\infty}^{\infty} f(x)dx$

$$x_n=\phi(nh)=\sinh\left[\frac{\pi}{2}\sinh\left(nh\right)\right] \qquad \phi'(nh)=\frac{\pi}{2}\cosh nh\cosh\left(\frac{\pi}{2}\sinh(nh)\right)$$

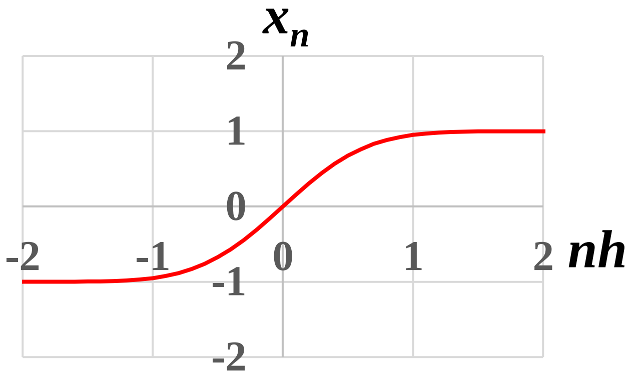
Integ. Points of double exp formula

二重指数関数型公式の積分点

戸田英雄, 小野令美, 入門 数値計算, オーム社 (昭和58年)

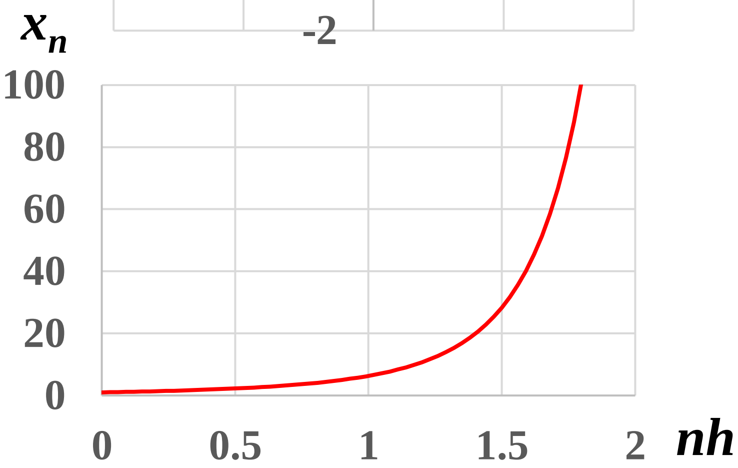
For $\int_{-1}^{1} f(x)dx$

$$x_n = \phi(nh) = \tanh\left[\frac{\pi}{2}\sinh(nh)\right]$$



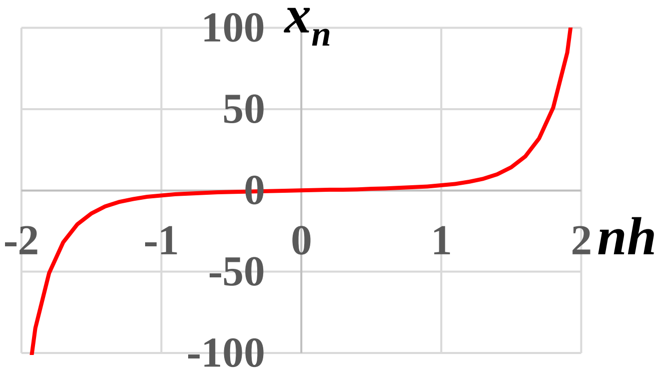
For $\int_{0}^{\infty} f(x)dx$

$$x_n = \phi(nh) = \exp\left[\frac{\pi}{2}\sinh(nh)\right]$$



For $\int_{-\infty}^{\infty} f(x)dx$

$$x_n = \phi(nh) = \sinh\left[\frac{\pi}{2}\sinh(nh)\right]$$



Error for integration with anomaly points

$$S = \int_{-1}^{1} \sqrt{1 - x^2}\, dx$$ **Exact: π/2 = 1.5707963**

nDivided	Rieman	Trapezoid	Simpson	Simpson 3/8	Bode	Romberg	Cubic Spline	Order 3 Gauss-Legendre	IMT	Double exp*
2	5.71E-01	5.71E-01	2.37E-01			2.37E-01		2.08E-02	1.03E+00	1.5708035
3	3.14E-01	3.14E-01		1.57E-01					1.74E-01	-0.52993
4	2.05E-01	2.05E-01	8.28E-02		7.24E-02	7.24E-02	6.93E-02	7.24E-03	2.72E-02	0.1417235
5	1.47E-01	1.47E-01					5.26E-02		3.40E-03	-0.0288253
6	1.12E-01	1.12E-01	4.48E-02	5.47E-02			3.97E-02	3.92E-03	2.99E-03	0.0050382
7	8.90E-02	8.90E-02					3.17E-02		8.70E-04	-0.0007911
8	7.29E-02	7.29E-02	2.90E-02		2.54E-02	2.47E-02	2.60E-02	2.54E-03	2.37E-05	0.0001138
9	6.12E-02	6.12E-02		2.96E-02			2.18E-02		7.98E-05	-1.55E-05
10	5.23E-02	5.23E-02	2.07E-02				1.87E-02	1.81E-03	4.90E-05	1.99E-06
11	4.53E-02	4.53E-02					1.62E-02		1.32E-05	-2.49E-07
12	3.98E-02	3.98E-02	1.57E-02	1.92E-02	1.38E-02		1.42E-02	1.38E-03	4.53E-06	2.82E-08
13	3.53E-02	3.53E-02					1.26E-02		8.86E-06	-6.05E-09
14	3.16E-02	3.16E-02	1.25E-02				1.13E-02	1.09E-03	6.87E-06	-3.54E-09
15	2.85E-02	2.85E-02		1.37E-02			1.02E-02		2.03E-06	-5.65E-09
16	2.59E-02	2.59E-02	1.02E-02		8.95E-03	8.62E-03	9.25E-03	8.93E-04	1.23E-05	-7.57E-09
17	2.36E-02	2.36E-02					8.45E-03		2.22E-06	-9.79E-09
18	2.17E-02	2.17E-02	8.54E-03	1.04E-02			7.76E-03	7.48E-04	1.05E-05	-1.22E-08
19	2.00E-02	2.00E-02					7.15E-03		1.21E-05	-1.48E-08
20	1.85E-02	1.85E-02	7.29E-03		6.40E-03		6.63E-03	6.38E-04	1.12E-05	-1.75E-08
32						3.04E-03				-5.18E-08

* 変換積分範囲は u = [-2.0, 2.0]

有限温度での粒子数、エネルギー

Fermi-Dirac分布関数

$$f(e) = \frac{1}{\exp(\beta(e - E_F)) + 1}$$

状態密度関数

$$\boldsymbol{N(e) = (2S+1)V\frac{2\pi(2m)^{3/2}}{h^3}\sqrt{e}}$$

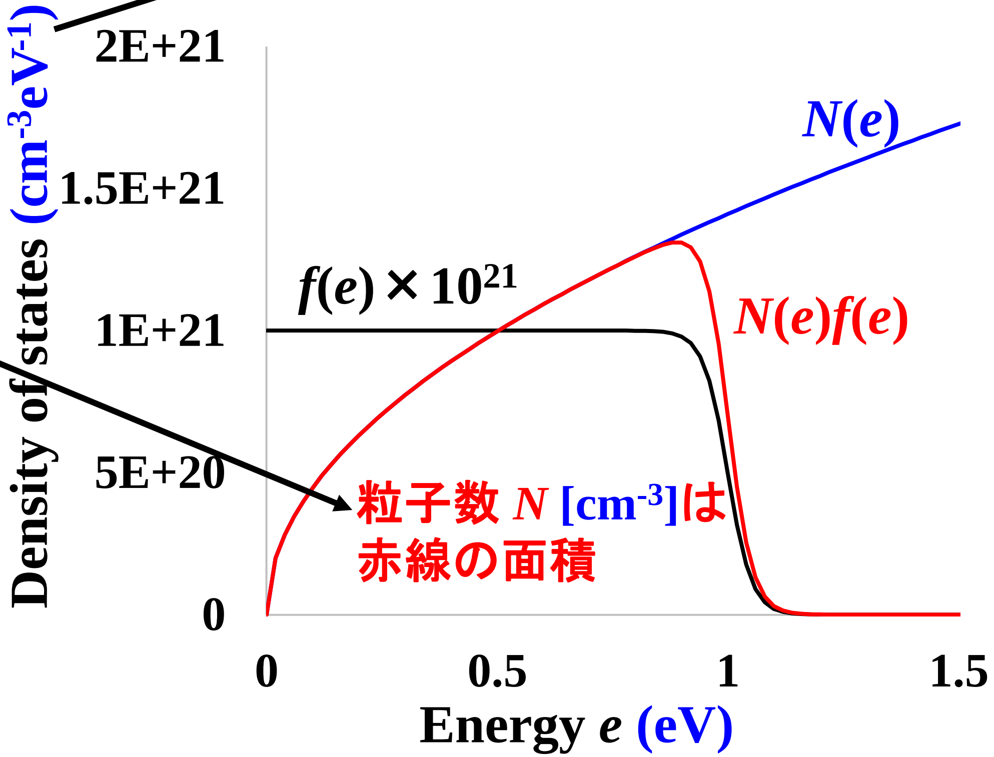
伝導帯中の電子数

$$\boldsymbol{N = \int_0^\infty N(e)f(e)\,de}$$

電子系の内部エネルギー

$$\boldsymbol{U = \int_0^\infty e(k)N(e)f(e)\,de}$$

状態密度の「密度」は体積当たり状態数のエネルギー密度



粒子数 N [cm^{-3}]は赤線の面積

Program: Calculate Ne in metal

Issue: How to integrate *N*(*e*)*f*(*e*) efficiently

- Wide integration range $E = 0 \sim E_F + \alpha k_B T \sim$ several eV (if precision is ~exp(-α))
- The range that needs precise calc is only around E_F with a range $\alpha k_B T \sim 0.1$ eV
- Function changes sharply around E_F, so integration mesh ΔE should be fine enough (e.g., $\Delta E < \alpha k_B T / 100$, 1 meV)

=> We should not the same ΔE throughout the entire integration range $E = 0 \sim E_F + \alpha k_B T$

=> Divide integration range

(We can use the analytical form for the range $0 \sim E_F - \alpha k_B T$)

Usage: python N-integration-metal.py 300 5.0

Temperature at 300K, $E_F = 5.0$ eVで

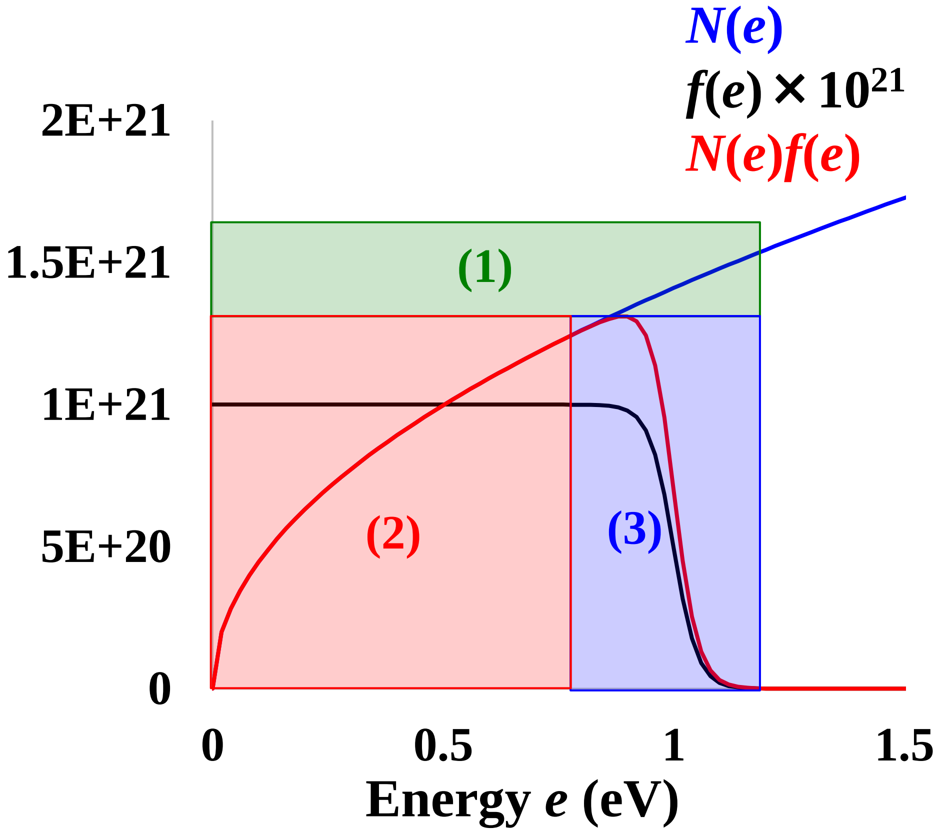
Measure time by repeating for 300 times

Precision 8digits (epsrel = 1e-8), α = 6:

	Integ. range	**Time for 300 repetition**
(1)	**$0 \sim E_F + \alpha k_B T$**	**0.109 秒**
(2)	**$0 \sim E_F - \alpha k_B T$**	**0.063 秒**
(3)	**$E_F - \alpha k_B T \sim E_F + \alpha k_B T$**	**0.016 秒**

30% faster for (2) + (3)

Using analytic form for (2) is 10 times faster



Program: Debye model of heat capacity

$$C_V = 3Rf_D\left(\frac{\Theta_D}{T}\right)$$ **Debye eq**

$$f_D(y) = \frac{3}{y^3}\int_0^y \frac{x^4 e^x}{\left(e^x - 1\right)^2}dx$$ **Debye function**

数値積分を使って計算: python の scipyモジュールの quad 関数 (適応積分法) を使ってみる

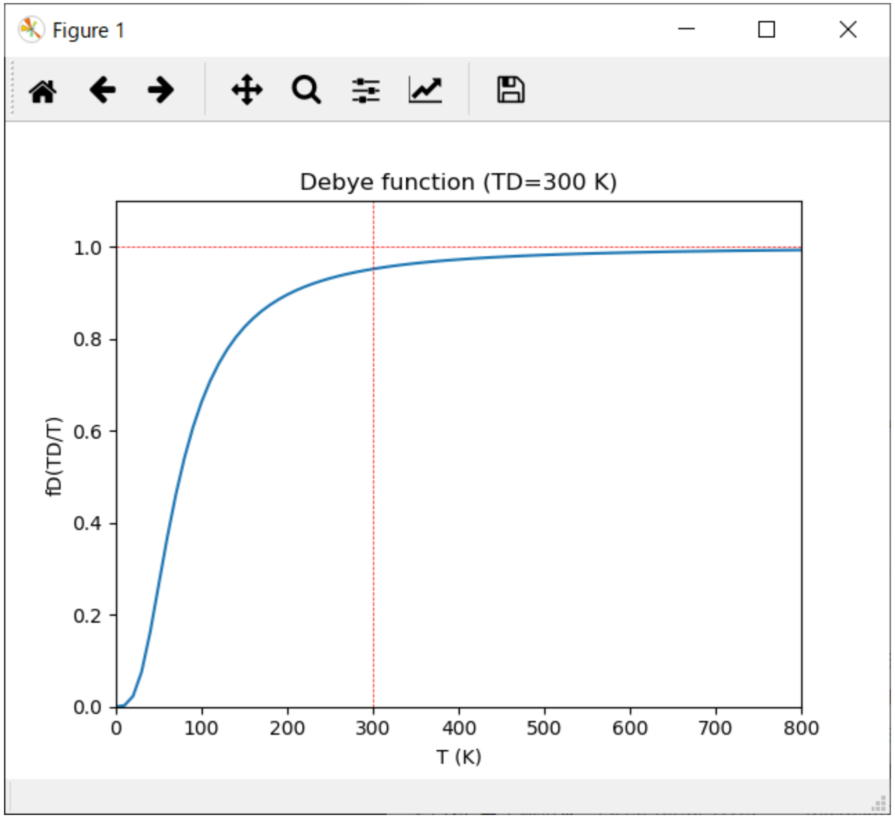
参考例 : https://org-technology.com/posts/integrate-function.html

数値積分の講義資料: http://conf.msl.titech.ac.jp/Lecture/python/index-numericalanalysis.html

python debye_function.py 300 0 500 10

Debye temperature 300 K

Temperate range 0 – 500 K, 10 K step



Numerical solutions of differential equations

微分方程式の数値解法

Motion of planets – Analytical solution

(惑星の運動 – 解析解)

$$m\frac{d^2\mathbf{r}}{dt^2} = -G\frac{mM}{r^2}\frac{\mathbf{r}}{r}$$

$$mr^2\frac{d\theta}{dt} = l$$

l:a constant, conservation of angular momentum

$$\frac{1}{2}m\left(\frac{dr}{dt}\right)^2 + m\left(\frac{l^2}{2m^2r^2} - \frac{GM}{r}\right) = E$$

$$r(\theta) = \frac{b}{1+\varepsilon\cos(\theta-\delta)}$$

$$b = \frac{l^2}{mc}$$

$$\varepsilon = \sqrt{1+2El^2/mc^2}$$

Elliptic equations (楕円方程式)

Long radius of ellipse $a' = 2b/\left(1-\varepsilon^2\right)$

Short radius of ellipse $b' = 2b/\sqrt{1-\varepsilon^2}$

Eccentricity (離心率) 焦点間の距離/長径 $\varepsilon = \sqrt{1+2El^2/mc^2}$

Close distance point (近点距離) $q = a'(1\text{-}e) = b/(1+\varepsilon)$

Long distance point (遠点距離) $Q = a'(1+e) = b/(1\text{-}\varepsilon)$

Period (周期) $T = 2\pi\sqrt{ma^3/c}$

Normalization of equation

(方程式の規格化/無次元化)

$$m\frac{d^2\mathbf{r}}{dt^2} = -G\frac{mM}{r^2}\frac{\mathbf{r}}{r}$$

Convert variables to T and R by representative constants τ$_0$ and l_0

$$t = \tau_0 T \qquad r = l_0 R$$

τ$_0$, l_0: Time and length specific to the system
Chose so that T and R will the the order of 1.0

$$m\frac{l_0}{\tau_0{}^2}\frac{d^2\mathbf{R}}{dT^2} = -G\frac{1}{l_0{}^2}\frac{mM}{R^2}\frac{\mathbf{R}}{R}$$

E.g., for planet simulation
τ$_0$ = Revolution / Rotation period
(公転 / 自転周期)
l_0 = Revolution radius, Astronomy unit
for molecular dynamics (MD)
τ$_0$ = MD time step
l_0 = Bohr radius (atomic unit)

$$\frac{d^2\mathbf{R}}{dT^2} = -G'\frac{mM}{R^2}\frac{\mathbf{R}}{R} \qquad G' = \frac{G\tau_0{}^2}{l_0{}^3}$$

First-order diff. eq. : Euler formula (オイラー法)

$$\frac{dx}{dt} = f(x,t)$$

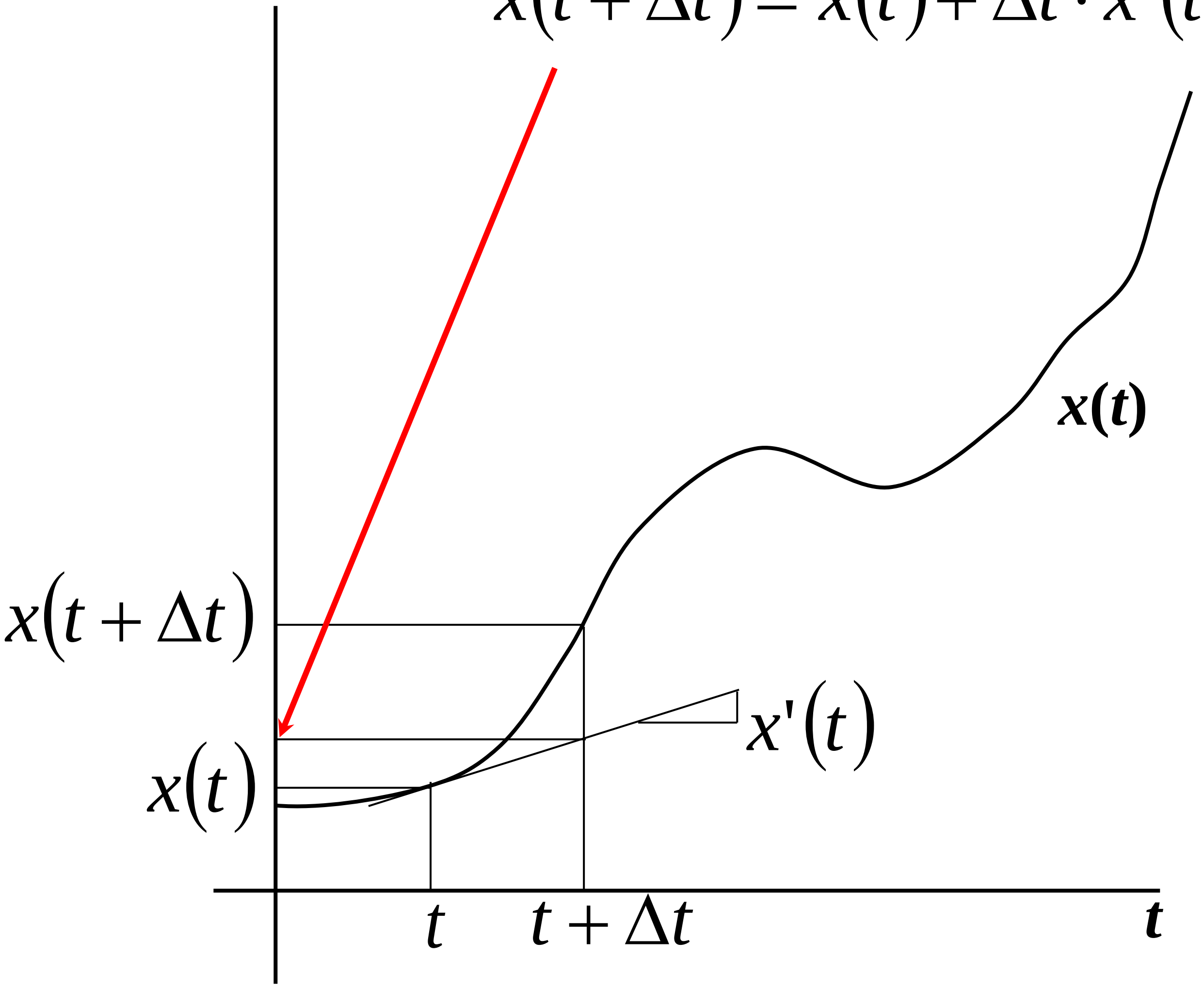
$$\frac{x(t+\Delta t) - x(t)}{\Delta t} = f(t, x(t))$$

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

- **Accuracy not good**
- **Asymmetric with respect to *t*, *t*+Δ*t***

Illustrative image of Euler method

$$x(t+\Delta t) = x(t) + \Delta t \cdot x'(t)$$



First-order diff. eq. : Heun formula (ホイン法)

$$\frac{dx}{dt} = f\left(t, x(t)\right)$$

- **Average the Euler formula at *t* and *t*+Δ*t***

$$x(t+\Delta t) = x(t) + \frac{1}{2}\Delta t[f(t, x(t)) + \boldsymbol{f(t+\Delta t, x(t+\Delta t))}]$$

Problem: *x*(*t*+Δ*t*) and *f*(*t*+Δ*t*, *x*(*t*+Δ*t*)) are unknown

=> Use *x*(*t*+Δ*t*) by Euler formula

$$x(t+\Delta t) \sim x(t) + \Delta t f(t) = x(t) + k_0$$

$$k_0 = \Delta t \cdot f\left(t, x(t)\right)$$

$$k_1 = \Delta t \cdot f\left(t+\Delta t, x(t+\Delta t)\right) \sim \Delta t \cdot f\left(t+\Delta t, x(t)+k_0\right)$$

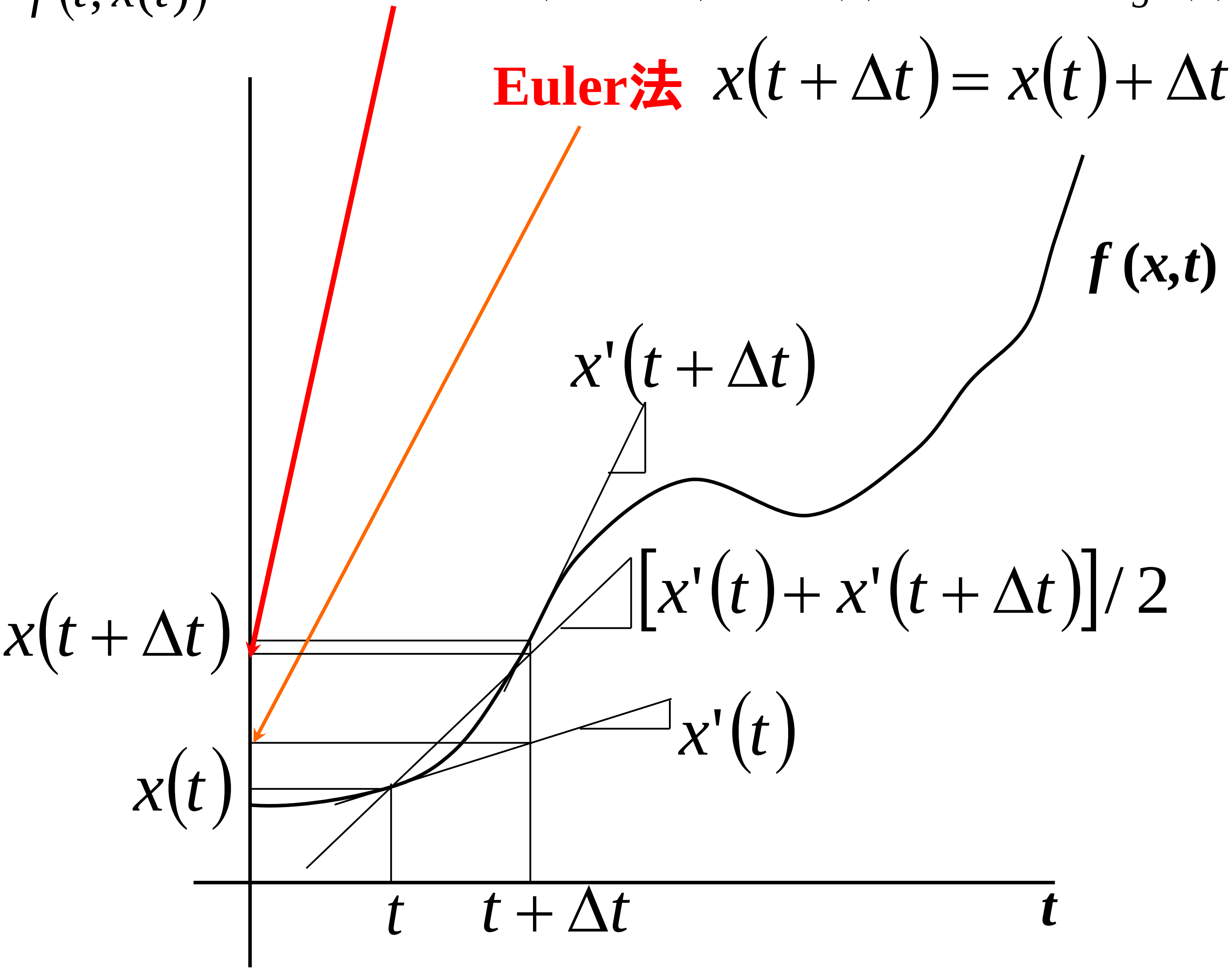
$$\boxed{x(t+\Delta t) = x(t) + \frac{k_0 + k_1}{2}}$$

Illustrative image of Heun method

$$\frac{dx}{dt} = x'(T) = f\left(t, x(t)\right)$$

Heun法 $x\left(t+\Delta t\right) = x\left(t\right) + \Delta t \cdot x'_{avg}\left(t\right)$

Euler法 $x\left(t+\Delta t\right) = x\left(t\right) + \Delta t \cdot x'\left(t\right)$



$f\,(x,t)$
$x'\left(t+\Delta t\right)$
$\left[x'\left(t\right) + x'\left(t+\Delta t\right)\right]/\,2$
$x'\left(t\right)$
$x\left(t+\Delta t\right)$
$x\left(t\right)$
t
$t+\Delta t$
t

First-order differential equation

$$\frac{dx}{dt} = f(x, t)$$

Euler formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$

$$\boldsymbol{x(t + \Delta t) = x(t) + k_0}$$

Heun formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$

$$k_1 = \Delta t \cdot f(x(t) + k_0, t + \Delta t)$$

$$\boldsymbol{x(t + \Delta t) = x(t) + \frac{k_0 + k_1}{2}}$$

Outline of program

```
dt  = 0.01
t0 = 0.0
x0 = 1.0

# dx/dt = dxdt(t, x)
def dxdt(t, x):
    return -x*x
```

```
# Solve by the Euler formula
def diffeq_euler(diff1func, t0, x0, dt):
   k0 = dt * diff1func(t0, x0)
   x1 = x0 + k0
   return x1

x1 = diffeq_euler(dxdt, t0, x0, dt)
```

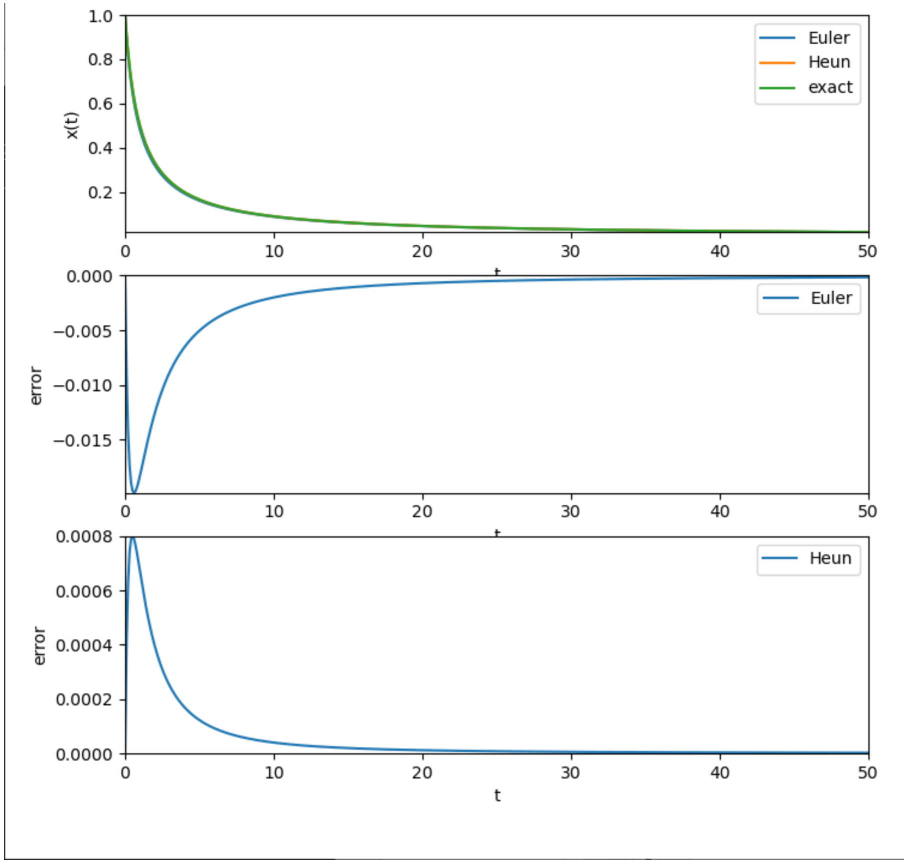
```
# Solve by the Heun formula
def diffeq_heun(diff1func, t0, x0, dt):
   k0 = dt * diff1func(t0, x0)
   k1 = dt * diff1func(t0+dt, x0+k0)
   x1 = x0 + (k0 + k1) / 2.0
   return x1

x1 = diffeq_heun(dxdt, t0, x0, dt)
```

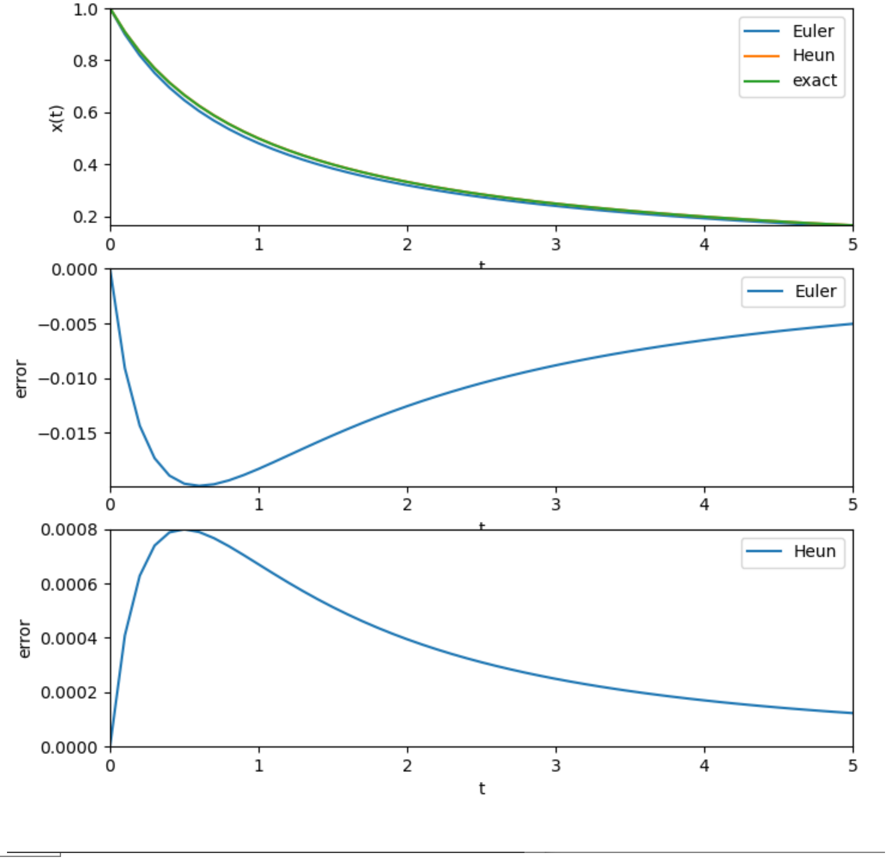
Program: Euler vs. Heun methods

Usage: python diffeq_euler_heun.py x0 dt nt iprint_interval

```
python diffeq_euler_heun.py
```

$$\frac{dx}{dt} = -x^2 \text{ for } x_0 = 1.0, \Delta t = 0.1, n_t = 501$$





First-order diff. eq. : Simpson formula (シンプソン則)

$$\int_{x_0}^{x_2} g(x')dx' \sim \frac{1}{3}h\left[g(x_0)+4g(x_1)+g(x_2)\right] = f(x_2)-f(x_0)$$

Solution of $\frac{df(x)}{dx} = g(x)$ => $\frac{dx}{dt} = f(t,x)$

$$x(t+2\Delta t) = x(t) + \frac{1}{3}\Delta t\left[f(t)+4f(t+\Delta t)+f(t+2\Delta t)\right]$$

Problem: $x(t+\Delta t)$ and $x(t+2\Delta t)$ are unknown

=> Use $x(t+\Delta t)$ by Euler or Heun formula

$$x(t+2\Delta t) = x(t) + \frac{k_0+4k_1+k_2}{3}$$

$$k_0 = \Delta t \cdot f(t,x)$$
$$k_1 = \Delta t \cdot f(t+\Delta t, x+k_0)$$
$$k_2 = \Delta t \cdot f(t+2\Delta t, x+k_0+k_1)$$

Convert Δt to a half

$$x(t+\Delta t) = x(t) + \frac{k_0+4k_1+k_2}{6}$$

$$k_1 = \Delta t \cdot f(t+\Delta t/2, x+k_0/2)$$
$$k_2 = \Delta t \cdot f(t+\Delta t, x+(k_0+k_1)/2)$$

=> Runge-Kutta formula

First-order diff. eq. : Runge-Kutta formula

(ルンゲークッタ公式)

$$\frac{dx}{dt} = f(t, x)$$

$$x(t+\Delta t) = x(t) + \frac{dx}{dt}\Delta t + \frac{1}{2!}\frac{d^2x}{dt^2}\Delta t^2 + \frac{1}{3!}\frac{d^3x}{dt^3}\Delta t^3 + \cdots$$

$$x(t+\Delta t) = x(t) + \mu_1 k_1 + \mu_2 k_2 + \mu_3 k_3 + \cdots$$

$$k_1 = \Delta t \cdot f(t, x)$$

$$k_2 = \Delta t \cdot f(t + \alpha_1 \Delta t, x + \beta_1 k_1)$$

$$k_3 = \Delta t \cdot f(t + \alpha_2 \Delta t, x + \beta_2 k_1 + \beta_3 k_2)$$

Determine μ_i and k_i so as to get minimum error

Number of k_i n => n-stage formula

Formula of $O(\Delta t^p) = 0$ is called ‘order p formula’

3-stage 3-order Runge-Kutta formula

(3段3次のRunge-Kutta公式)

$$x(t+\Delta t) = x(t) + \frac{k_0 + 4k_1 + k_2}{6} + O(h^4)$$

$$k_0 = \Delta t \cdot f(t, x)$$

$$k_1 = \Delta t \cdot f(t + \Delta t/2, x + k_0/2)$$

$$k_2 = \Delta t \cdot f(t + \Delta t, x \boxed{+2k_1 - k_0})$$

Different from Simpson formula

$(k_0 + k_1)/2$

Different μ_i and k_i can provide the same accuracy

(同じ精度で違う取り方もできる)

$$k^* = \Delta t \cdot f(t + \Delta t/4, x + \Delta x/4)$$

$$k_0 = \Delta t \cdot f(t, x)$$

$$k_1 = \Delta t \cdot f(t + \Delta t/2, x + k^*/2)$$

$$k_2 = \Delta t \cdot f(t + \Delta t, x + k_1)$$

4-stage 4-order Runge-Kutta formula

(4段4次のRunge-Kutta公式)

$$x(t+\Delta t) = x(t) + \frac{k_0 + 2k_1 + 2k_2 + k_3}{6}$$

$$k_0 = \Delta t \cdot f(t, x)$$

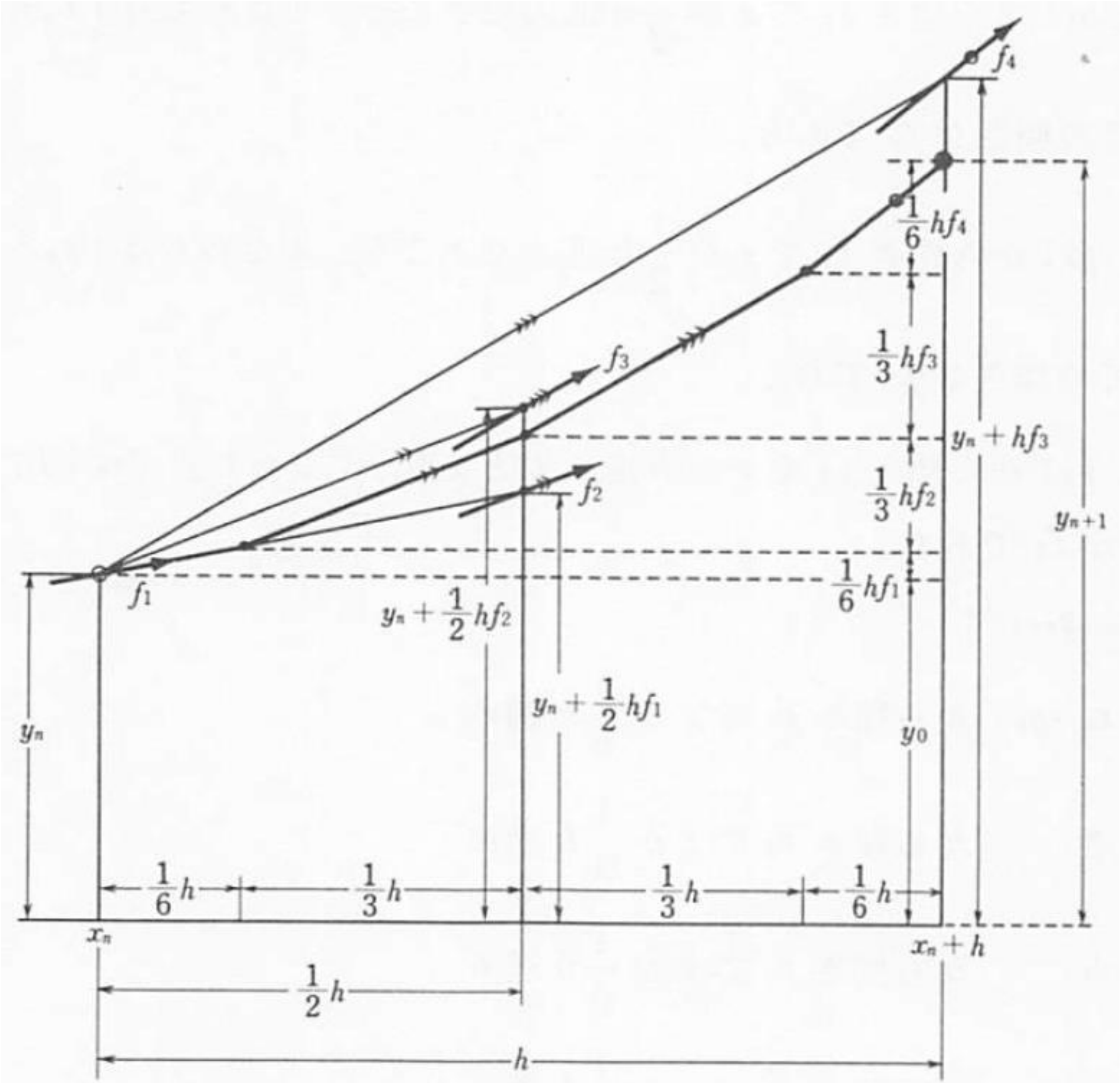
$$k_1 = \Delta t \cdot f(t + \Delta t/2, x + k_1/2)$$

$$k_2 = \Delta t \cdot f(t + \Delta t/2, x + k_2/2)$$

$$k_3 = \Delta t \cdot f(t + \Delta t, x + k_3)$$

Illustrative image of Runge-Kutta formula

戸田英雄, 小野令美，入門 数値計算, オーム社 (昭和58年)



First-order differential equation

$$\frac{dx}{dt} = f(x,t)$$

Euler formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$
$$\boldsymbol{x(t+\Delta t) = x(t) + k_0}$$

Heun formula:

$$k_0 = \Delta t \cdot f(x(t), \quad t)$$
$$k_1 = \Delta t \cdot f(x(t) + k_0, t + \Delta t)$$
$$\boldsymbol{x(t+\Delta t) = x(t) + \frac{1}{2}(k_0 + k_1)}$$

Simson formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$
$$k_1 = \Delta t \cdot f(x(t) + k_0/2, \quad t + \Delta t/2)$$
$$k_2 = \Delta t \cdot f(x(t) + (k_0 + k_1)/2, t + \Delta t)$$
$$\boldsymbol{x(t+\Delta t) = x(t) + \frac{1}{6}(k_0 + 4k_1 + k_2)}$$

3-stage 3-order Runge-Kutta formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$
$$k_1 = \Delta t \cdot f(x(t) + k_0/2, \quad t + \Delta t/2)$$
$$k_2 = \Delta t \cdot f(x(t) + 2k_1 - k_0, t + \Delta t)$$
$$\boldsymbol{x(t+\Delta t) = x(t) + \frac{1}{6}(k_0 + 4k_1 + k_2)}$$

4-stage 4-order Runge-Kutta formula:

$$k_0 = \Delta t \cdot f(x(t), t)$$
$$k_1 = \Delta t \cdot f(x(t) + k_1/2, t + \Delta t/2)$$
$$k_2 = \Delta t \cdot f(x(t) + k_2/2, t + \Delta t/2)$$
$$k_3 = \Delta t \cdot f(x(t) + k_3, \quad t + \Delta t)$$
$$\boldsymbol{x(t+\Delta t) = x(t) + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3)}$$

Second-order diff. eq. (二階微分方程式)

$$\frac{d^2\mathbf{r}_i}{dt^2} = \mathbf{F}_i / m_i$$

- **2nd-order diff eq is divided to two simultaneous 1st-order eqs**

(二階微分方程式の場合、一階微分方程式に分解するのが良い)

$$\frac{d^2x}{dt^2} = f(x, v, t)$$

$$\frac{dv}{dt} = f(x, v, t) \qquad \frac{dx}{dt} = v$$

Euler formula: $v(t+\Delta t) \sim v(t) + \Delta t \cdot \frac{dv}{dt}$

$$\boldsymbol{v(t+\Delta t) = v(t) + \Delta t \cdot f(x(t), v(t), t)}$$

$$\boldsymbol{x(t+\Delta t) = x(t) + \Delta t \cdot v(t)}$$

Second-order diff. eq. : Heun formula

(二階微分方程式の解法: ホイン法)

$$\frac{d^2x}{dt^2} = f(x, v, t)$$

$$\frac{dv}{dt} = f(x, v, t)$$

(1) $k_0 = \Delta t \cdot f(x(t), v(t), t)$

(3) $k_1 = \Delta t \cdot f(\boldsymbol{x(t) + k_0'}, \boldsymbol{v(t) + k_0}, t + \Delta t)$

(4) $v(t + \Delta t) = v(t) + \frac{1}{2}(k_0 + k_1)$

Each step needs to calculate k_0 and k_1: time-consuming for MD

$$\frac{dx}{dt} = v(x, v, t)$$

(2) $k_0{}' = \Delta t \cdot v(t)$

(5) $k_1{}' = \Delta t \cdot v(t + \Delta t)$

(6) $x(t + \Delta t) = x(t) + \frac{1}{2}(k_0{}' + k_1{}')$

Second-order diff. eq. : Verlet formula

(二階微分方程式の解法: ベルレ法)

$$\frac{d^2x}{dt^2} = f(x, v, t)$$

$$f(x, v, t) = \frac{d^2x(t)}{dt^2} \sim \frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}$$

$$x(t+\Delta t) = 2x(t) - x(t-\Delta t) + \Delta t^2 f(\boldsymbol{x}(\boldsymbol{t}), \boldsymbol{v}(\boldsymbol{t}), \boldsymbol{t})$$

$$v(t) = \frac{1}{2\Delta t}\{x(t+\Delta t) - x(t-\Delta t)\}$$

Each step only needs to calculate one $\boldsymbol{f(x(t), v(t), t)}$

- **Better accuracy than Euler formula, equivalent to Heun formula**
- **Directly solve 2nd-order differential equation**
- **Drawback:**

  The subtraction of similar values, *x*(*t*+*n⊿t*), may cause roundoff error.

velocity Verlet formula

$$\frac{d^2x}{dt^2} = f\left(t, x, v\right)$$

$$\frac{d^2x(t+\Delta t)}{dt^2} \sim \frac{x(t+2\Delta t) - 2x(t+\Delta t) + x(t)}{\Delta t^2}$$

$$x(t+2\Delta t) = 2x(t+\Delta t) - x(t) + \Delta t^2 f(t+\Delta t)$$

$$x\left(t+\Delta t\right) = 2x\left(t\right) - x\left(t-\Delta t\right) + \Delta t^2 f\left(t, x\left(t\right), v\left(t\right)\right)$$

$$v\left(t+\Delta t\right) = v\left(t\right) + \frac{1}{2}\left\{f\left(t+\Delta t\right) + f\left(t\right)\right\}$$

- **Better accuracy than Verlet formula**

Program: diffeq2nd_verlet.py

Usage: python diffeq2nd_verlet.py

```
 t           x(cal)       x(exact)     v(cal)
t= 0.00    0.000000    0.000000    1.000000
t= 0.01    0.010000    0.010000    0.999950
t= 0.20    0.198673    0.198669    0.980066
t= 0.40    0.389425    0.389418    0.921060
t= 0.60    0.564652    0.564642    0.825334
t= 0.80    0.717367    0.717356    0.696704
t= 1.00    0.841484    0.841471    0.540299
t= 1.20    0.932053    0.932039    0.362353
t= 1.40    0.985463    0.985450    0.169961
t= 1.60    0.999586    0.999574   -0.029206
t= 1.80    0.973858    0.973848   -0.227209
t= 2.00    0.909305    0.909297   -0.416154
t= 2.20    0.808501    0.808496   -0.588509
t= 2.40    0.675464    0.675463   -0.737400
t= 2.60    0.515499    0.515501   -0.856894
t= 2.80    0.334981    0.334988   -0.942226
t= 3.00    0.141109    0.141120   -0.989994
t= 3.20   -0.058388   -0.058374    -0.998294
t= 3.40   -0.255558   -0.255541    -0.966795
t= 3.60   -0.442539   -0.442520    -0.896752
t= 3.80   -0.611878   -0.611858    -0.790958
t= 4.00   -0.756823   -0.756802    -0.653631
t= 4.20   -0.871595   -0.871576    -0.490246
t= 4.40   -0.951620   -0.951602    -0.307315
t= 4.60   -0.993706   -0.993691    -0.112133
```

Second-order diff. eq. : Leap Flog formula

(二階微分方程式の解法: かえる跳び法)

Essentially the same as the Verlet formula.

However, Verlet formula

$$x(t+\Delta t) = 2x(t) - x(t-\Delta t) + \Delta t^2 f(t, x(t), v(t))$$

includes the subtraction of

***x*(*t*) terms and may cause roundoff error.**

Converting the equation to

$$v(t+\Delta t) = v(t-\Delta t) + 2\Delta t \cdot f(t)$$

$$x(t+2\Delta t) = x(t) + 2\Delta t \cdot v(t+\Delta t)$$

Can reduce the roundoff errors.

Note: Time calculated for *x*(*t*) and *v*(*t*) are shifted by Δt

Leap Flog vs. Verlet

Confirm the Leap Flog formula is identical to the Verlet formula

Leap Flog $x(t+2\Delta t)=x(t)+2\Delta t\cdot v(t+\Delta t)$

$$v(t-\Delta t)=\frac{x(t)-x(t-\Delta t)}{\Delta t}$$

$$\frac{x(t+\Delta t)-x(t)}{\Delta t}=\frac{x(t)-x(t-\Delta t)}{\Delta t}+2\Delta t\cdot f(t)$$

$$x(t+\Delta t)=2x(t)-x(t-\Delta t)+2\Delta t\cdot f(t)$$

Verlet

Program: diffeq2nd_2d_euler.py

Usage: python diffeq2nd_2d_euler.py

```
 t          x(cal)     x(exact)    y(cal)      y(exact)
t= 0.00   0.000000   0.000000   2.000000    2.000000
t= 0.01   0.010000   0.010000   2.000000    1.999900
t= 0.20   0.198862   0.198669   1.962097    1.960133
t= 0.40   0.390186   0.389418   1.845820    1.842122
t= 0.60   0.566322   0.564642   1.655653    1.650671
t= 0.80   0.720212   0.717356   1.399036    1.393413
t= 1.00   0.845671   0.841471   1.086077    1.080605
t= 1.20   0.937633   0.932039   0.729152    0.724716
t= 1.40   0.992364   0.985450   0.342415    0.339934
t= 1.60   1.007603   0.999574  -0.058761   -0.058399
t= 1.80   0.982665   0.973848  -0.458394   -0.454404
t= 2.00   0.918464   0.909297  -0.840535   -0.832294
t= 2.20   0.817482   0.808496  -1.189900   -1.177002
t= 2.40   0.683677   0.675463  -1.492481   -1.474787
t= 2.60   0.522322   0.515501  -1.736110   -1.713778
t= 2.80   0.339800   0.334988  -1.910948   -1.884445
t= 3.00   0.143353   0.141120  -2.009878   -1.979985
t= 3.20  -0.059207  -0.058374   -2.028803   -1.996590
t= 3.40  -0.259811  -0.255541   -1.966806   -1.933596
t= 3.60  -0.450448  -0.442520   -1.826199   -1.793517
t= 3.80  -0.623492  -0.611858   -1.612436   -1.581935
t= 4.00  -0.772001  -0.756802   -1.333901   -1.307287
t= 4.20  -0.890001  -0.871576   -1.001578   -0.980522
t= 4.40  -0.972722  -0.951602   -0.628623   -0.614666
t= 4.60  -1.016792  -0.993691   -0.229835   -0.224305
```

Program: diffeq2nd_2d_verlet.py

Usage: python diffeq2nd_2d_verlet.py

```
 t           x(cal)      x(exact)    y(cal)      y(exact)
t= 0.00    0.000000    0.000000    2.000000    2.000000
t= 0.01    0.010050    0.010000    1.999950    1.999900
t= 0.20    0.199666    0.198669    1.961126    1.960133
t= 0.40    0.391372    0.389418    1.844068    1.842122
t= 0.60    0.567475    0.564642    1.653492    1.650671
t= 0.80    0.720954    0.717356    1.396995    1.393413
t= 1.00    0.845691    0.841471    1.084805    1.080605
t= 1.20    0.936713    0.932039    0.729366    0.724716
t= 1.40    0.990390    0.985450    0.344850    0.339934
t= 1.60    1.004584    0.999574   -0.053414   -0.058399
t= 1.80    0.978727    0.973848   -0.449550   -0.454404
t= 2.00    0.913852    0.909297   -0.827762   -0.832294
t= 2.20    0.812544    0.808496   -1.172975   -1.177002
t= 2.40    0.678842    0.675463   -1.471424   -1.474787
t= 2.60    0.518076    0.515501   -1.711211   -1.713778
t= 2.80    0.336656    0.334988   -1.882778   -1.884445
t= 3.00    0.141815    0.141120   -1.979283   -1.979985
t= 3.20   -0.058680   -0.058374   -1.996880   -1.996590
t= 3.40   -0.256836   -0.255541   -1.934867   -1.933596
t= 3.60   -0.444752   -0.442520   -1.795716   -1.793517
t= 3.80   -0.614937   -0.611858   -1.584975   -1.581935
t= 4.00   -0.760607   -0.756802   -1.311046   -1.307287
t= 4.20   -0.875953   -0.871576   -0.984849   -0.980522
t= 4.40   -0.956378   -0.951602   -0.619389   -0.614666
t= 4.60   -0.998674   -0.993691   -0.229235   -0.224305
```

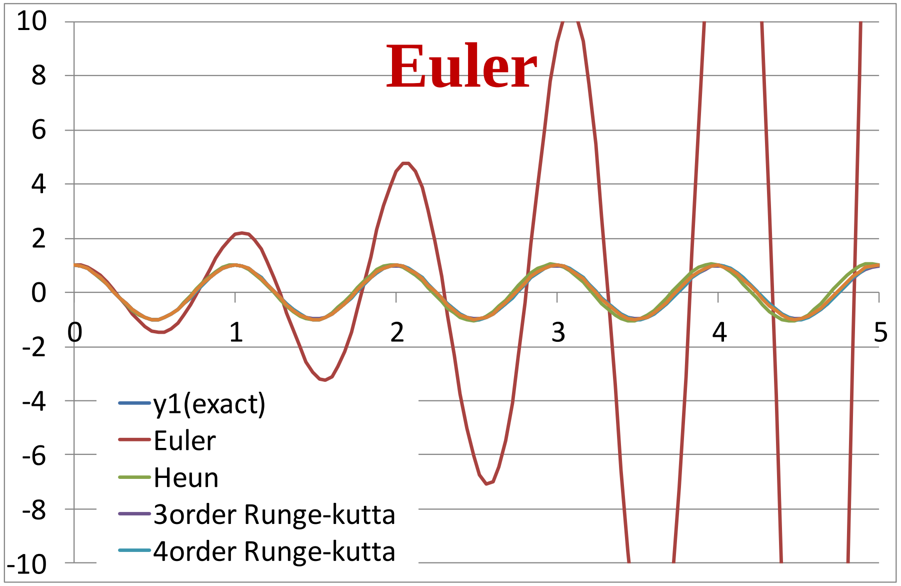
Accuracy of numerical solusions: Diff. eq.

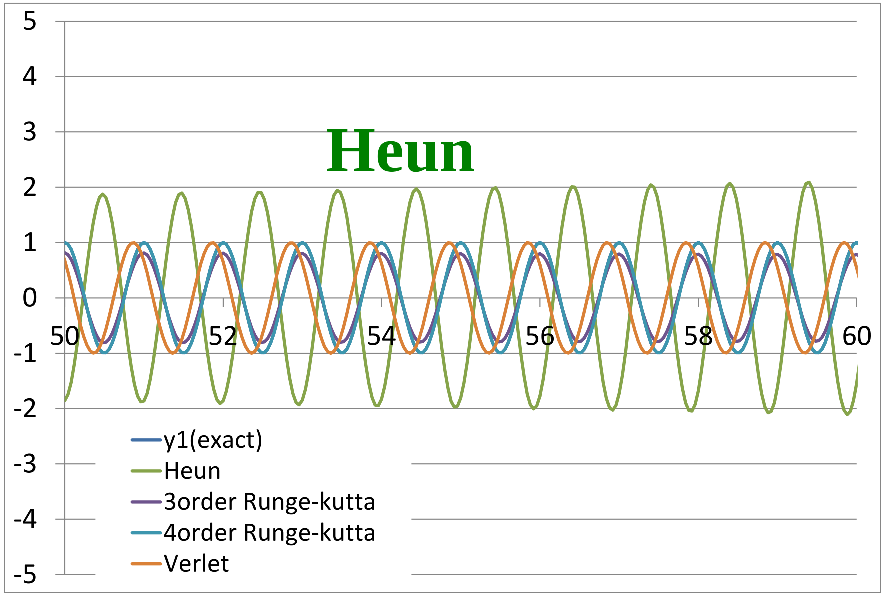
$$\frac{d^2x}{dt^2} = -4\pi^2 x \qquad \left(\frac{dx}{dt} = v, \quad \frac{dv}{dt} = -4\pi^2 x \right)$$

Exact (*t* = 0: *x* = 1.0, *v* = 0.0)

$$x = \cos(2\pi t) \quad v = -2\pi \sin(2\pi t)$$

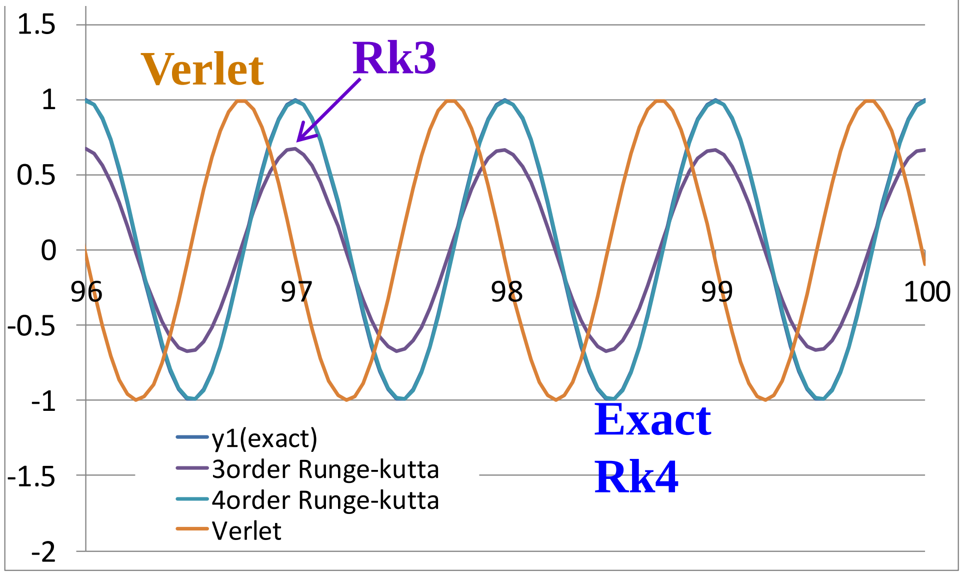
Δ*t* = 0.04

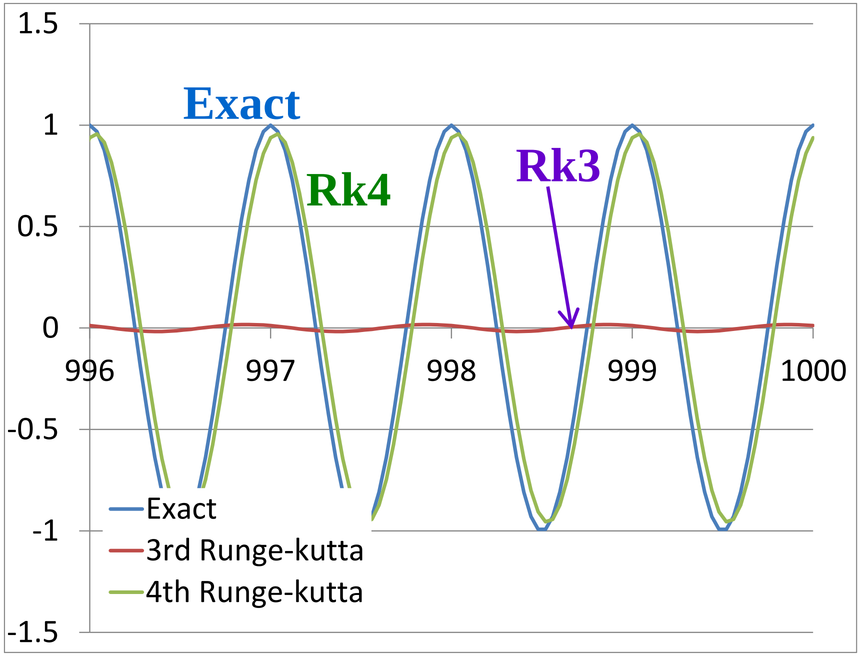




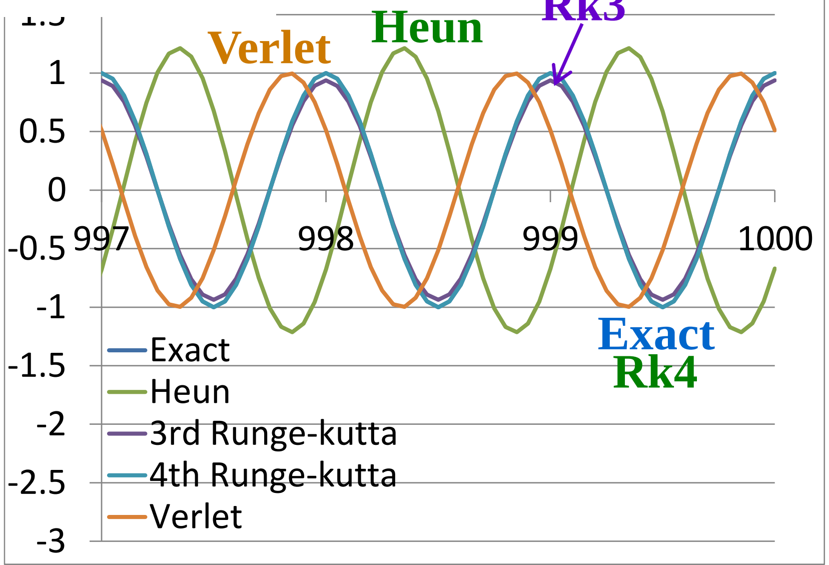
Accuracy of numerical solusions: Diff. eq.

Δt = 0.04



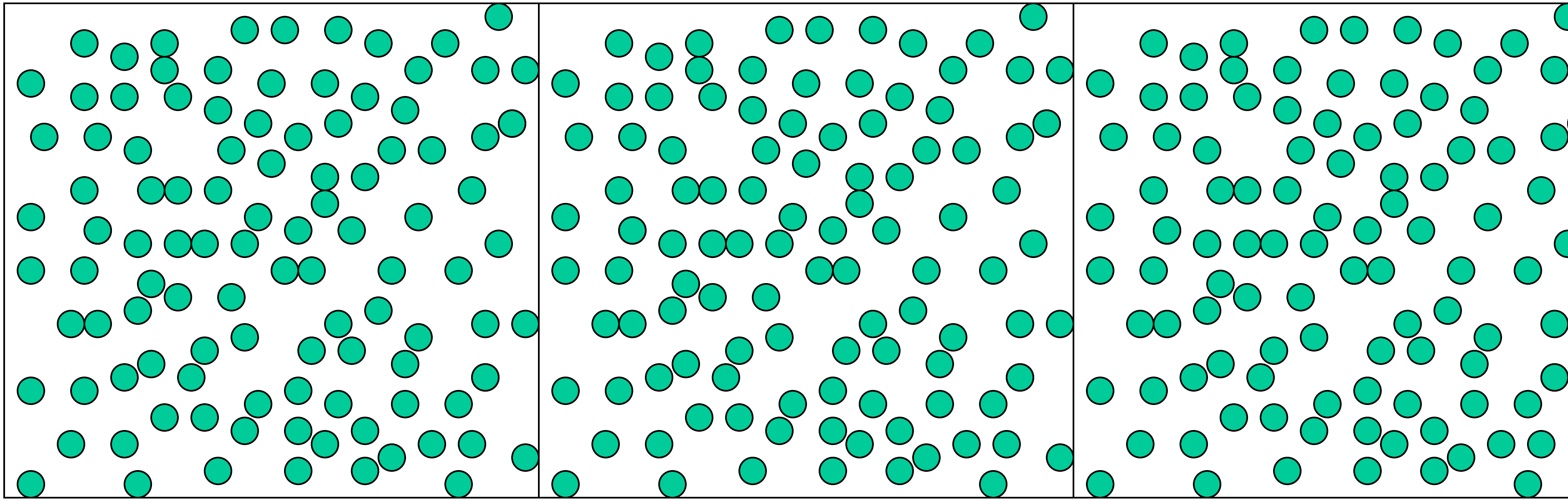


Δt = 0.01



Molecular dynamics (MD) (分子動力学法)

3D periodic condition: MD cell



$$\mathbf{F}_i = m_i \frac{d^2\mathbf{r}_i}{dt^2}$$

$$\mathbf{v}_i\left(t+\Delta t\right) = \mathbf{v}_i\left(t\right) + \Delta t \frac{\mathbf{F}_i}{m_i}$$

$$\mathbf{r}_i\left(t+\Delta t\right) = \mathbf{r}_i\left(t\right) + \Delta t \cdot \mathbf{v}_i\left(t\right)$$

Empirical interatomic potential

(経験的原子間ポテンシャル)

Hard core potential

ハードコア（剛体）ポテンシャル

$$\phi(r) = \infty \qquad r \le \sigma$$
$$= 0 \qquad r > \sigma$$

Lennard-Jones (LJ) potential

レナードージョーンズポテンシャル

$$\phi_{ij}(r) = 4\varepsilon_{ij}\left\{\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right\}$$

Born-Mayer-Huggins (BMH) potential

ボルンーメイヤーヒュギンズ

$$\phi_{ij}(r) = \frac{z_i z_j e^2}{r} + A_{ij} b \cdot \exp\left(\frac{\sigma_i + \sigma_j - r}{\rho}\right) - \frac{C_{ij}}{r^6} - \frac{D_{ij}}{r^8}$$

Kawamura potential（MXDOrto/MXDTricl）

河村ポテンシャル

$$\phi_{ij} = \frac{z_i z_j}{r_{ij}} + f_0(b_i + b_j)\exp\left(\frac{a_i + a_j - r_{ij}}{b_i + b_j}\right) + \frac{c_i c_j}{r_{ij}{}^6}$$

$$\phi_{ij}(r) = \frac{z_i z_j e^2}{r} + f_0(b_i + b_j)\exp\left(\frac{a_i + a_j - r}{b_i + b_j}\right)$$
$$+ D_{ij}\left(\exp\left[-2\beta_{ij}(r - r^*)\right] - 2\exp\left[-\beta_{ij}(r - r^*)\right]\right)$$

Morse potential

Empirical interatomic potential

$$U_{ij}(r_{ij}) = \frac{z_i z_j e^2}{4\pi\varepsilon_0}\frac{1}{r_{ij}} + f_0(b_i + b_j)\exp\left[\frac{a_i + a_j - r_{ij}}{b_i + b_j}\right] + \frac{c_i c_j}{r_{ij}{}^6}$$

Coulomb potential **Repulsion term** **Dispersion (London interaction)**

Example of Parameters for an ion

Ion charge	**:** z_i	**Fixed to ion formal charge**
~Ion radius	**:** a_i	**Adjust to crystal structure**
~Ion hardness:	b_i	**Adjust to elastic constant**
Dispersion	**:** c_i	**Fixed**

Potentials and forces for the ion *i* at r_i

$$U_i(\boldsymbol{r_i}, t) = \sum_j U_{ij}\left(\boldsymbol{r_j}(t) - \boldsymbol{r_i}(t)\right), \boldsymbol{F_i}(\boldsymbol{r_i}, t) = -\sum_j \frac{\partial}{\partial \boldsymbol{r_i}} U_{ij}\left(\boldsymbol{r_j}(t) - \boldsymbol{r_i}(t)\right)$$

Most time-consuming term

Better to re-use previous steps,

$\boldsymbol{F_i}(\boldsymbol{r_i}, \boldsymbol{t} - \Delta\boldsymbol{t}), \boldsymbol{F_i}(\boldsymbol{r_i}, \boldsymbol{t} - 2\Delta\boldsymbol{t})$ **etc**

=> Verlet formula is better than Heun and Runge-Kutta formula

Requirements of algorisms used for MD

Requirements

- Enough accuracy (can be checked by energy / momentum conservation laws)
- Fast calculations (note the most time-consuming process is the force calculations, better to re-use the previous results)

Runge-Kutta formula: not suitable for MD

High accuracy, but high cost

It cannot re-use the previous results

Each step requires three/four new force calculations, high cost

Frequently used formula:

- Verlet formula (Leap Flog formula)
- Beeman formula
- Predictor-Corrector method (予測子ー修正子法)

  Rahman predictor-corrector method
  (ラーマンの予測子ー修正子法)

  Gear predictor-corrector method (ギアの予測子ー修正子法)

Program: Planet simulation

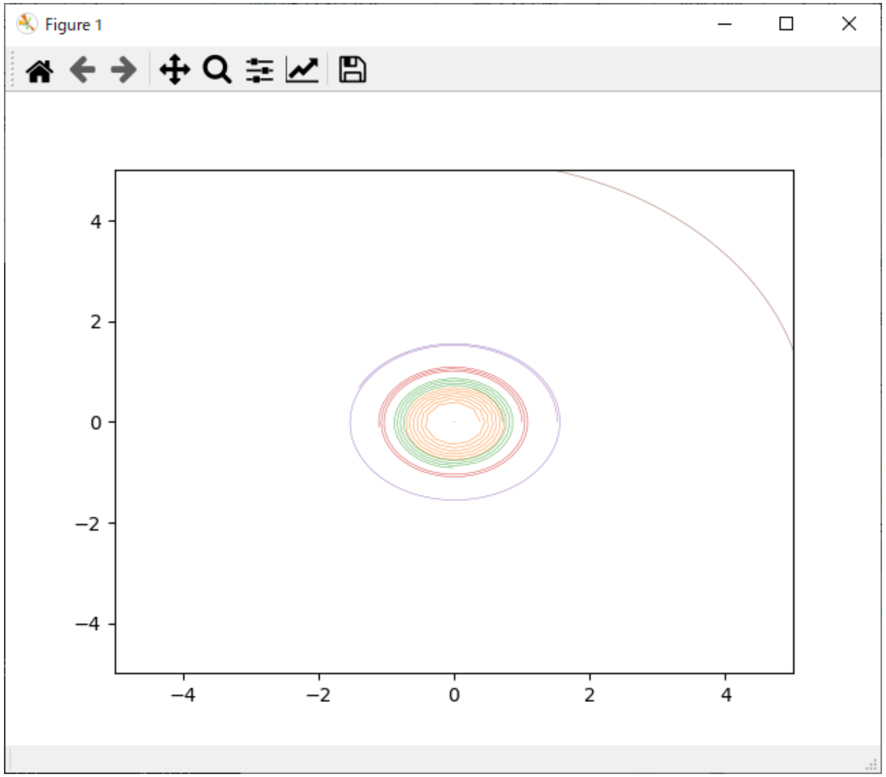
Usage: python diffeq2nd_planet.py solver dt nt

solver: ‘Euler’ or ‘Verlet’

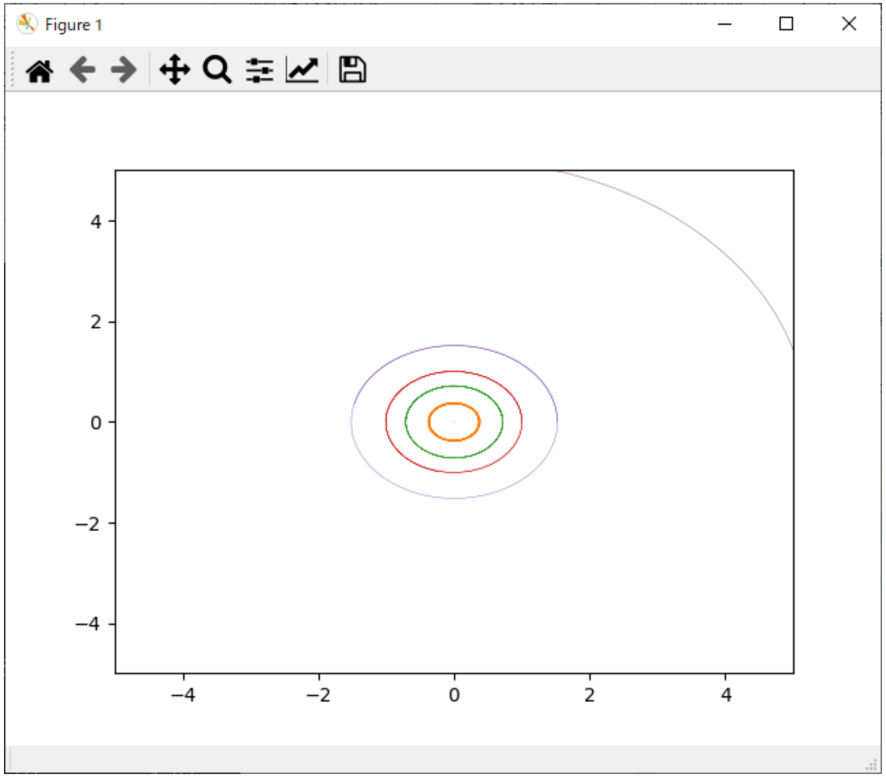
dt: time step in day (time is normalized by a day)

nt: number of steps

python diffeq2nd_planet.py **Euler** 0.2 5000

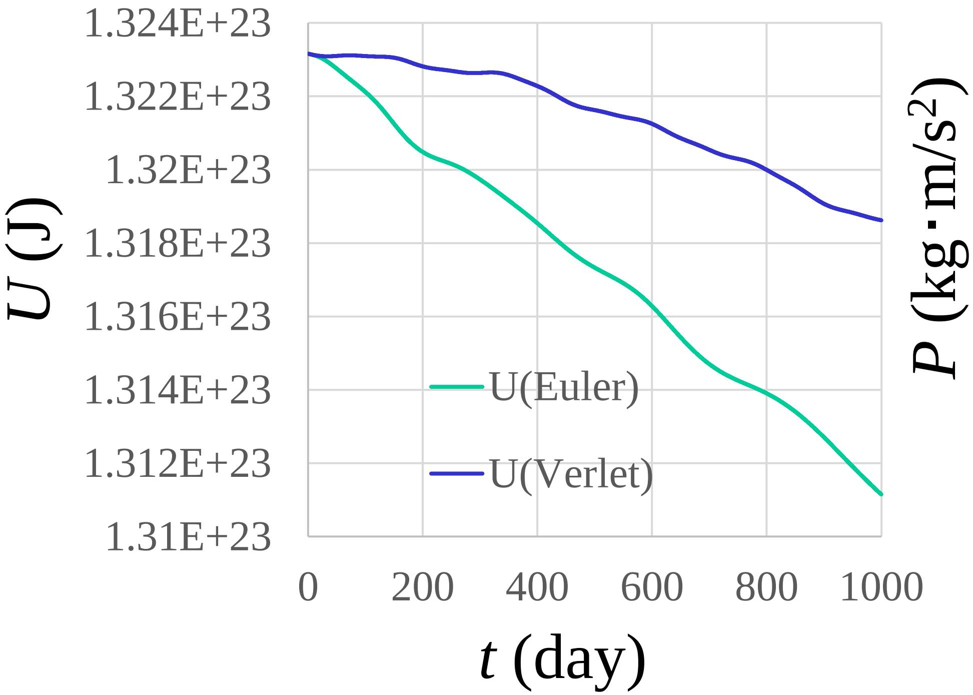


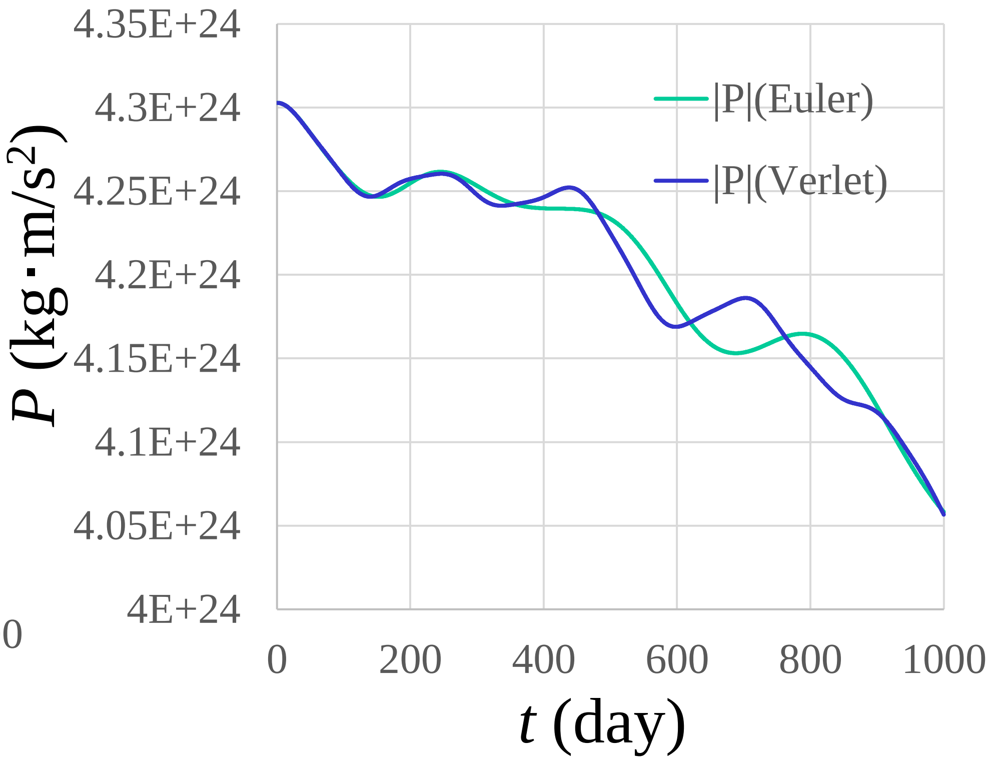
python diffeq2nd_planet.py **Verlet** 0.2 5000



Program: Check by conservation law

```
python diffeq2nd_planet.py Euler 0.2 5000
python diffeq2nd_planet.py Verlet 0.2 5000
```





Approximation of discrete data: Interpolation/Extrapolation

(離散データの近似: 補間/補外）

Interpolation

Pattern 1: Reproduce all sample points (標本点を必ず通る)

n sample points are reproduced by (*n* – 1) order polynomial.

・Interpolated data might be scattered largely in particular for orders higher than 3 (Runge's phenomenon/oscillation **ルンゲの現象**).

補間点が大きく振動する問題がでる。特に3次以上の多項式

=> To suppress the Runge's phenomenon:

Make the *n*-th order differentiations continuous at the boundaries between neighboring regions

=> Spline function

n sample points are reproduced by (*n* + *N* – 1) order polynomial.

Pattern 2: Smoothing (平滑化)

Scattering of data will be reduced

Pattern 3: Does not reproduce sample points exactly, but the deviation will be minimized

(標本点を通らないが、補間データは標本点から大きく外れない)

・Least-squares method (LSQ, 最小二乗法)

・Minimax approximation (ミニマックス近似)

Polynomial that reproduces sample points

(標本点を通る多項式)

n sample points (x_i, y_i) (i = 1, ･･･, n) are reproduced by (n-1) order polynomial.

$$y_i = \sum_{k=0}^{n-1} a_k {x_i}^k \qquad (i = 1, \cdots, n)$$

$$\begin{pmatrix} 1 & x_1 & {x_1}^2 & \cdots & {x_1}^{n-1} \\ 1 & x_2 & {x_2}^2 & & {x_2}^{n-1} \\ 1 & x_3 & {x_3}^2 & & {x_3}^{n-1} \\ \vdots & & & \ddots & \\ 1 & x_n & {x_n}^2 & & {x_n}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

$|x_i| > 1$ might cause overflow,

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

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

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

Lagrange interpolation formula

(ラグランジの補間公式)

戸田英雄, 小野令美, 入門 数値計算, オーム社 (昭和58年)

(n – 1) order polynomial that reproduces n sample points (x_{i}, y_{i}) (i = 0, ･･･, n – 1) is determined uniquely.

Lagrange interpolation formula

$$P_{n-1}(x) = f(x_0)\phi_0(x) + f(x_1)\phi_1(x) + \cdots f(x_{n-1})\phi_{n-1}(x)$$

$$\phi_i(x) = \frac{\prod_{k\neq i}^{n-1}(x-x_k)}{\prod_{k\neq i}^{n-1}(x_i-x_k)} = \prod_{k\neq i}^{n-1}\frac{(x-x_k)}{(x_i-x_k)}$$

n = 2:

$$P_1(x) = f(x_0)\frac{(x-x_1)}{(x_0-x_1)} + f(x_1)\frac{(x-x_0)}{(x_1-x_0)}$$

n = 3:

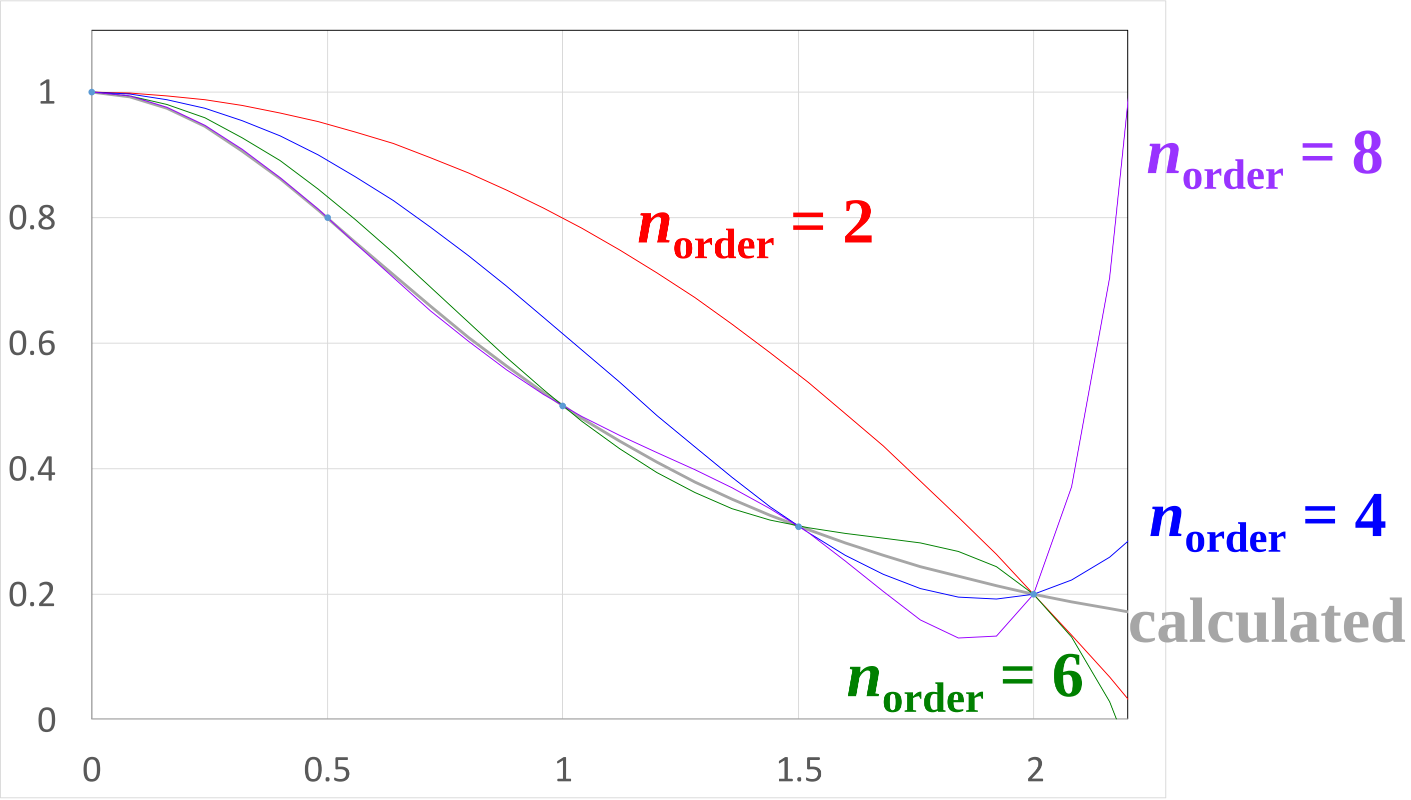
$$P_2(x) = f(x_0)\frac{(x-x_1)}{(x_0-x_1)}\frac{(x-x_2)}{(x_0-x_2)} + f(x_1)\frac{(x-x_0)}{(x_1-x_0)}\frac{(x-x_2)}{(x_1-x_2)} + f(x_2)\frac{(x-x_0)}{(x_2-x_0)}\frac{(x-x_1)}{(x_2-x_1)}$$

Problem of such polynomials

- Increasing the sample points will change the coefficients of polynomial completely.
- **Runge's phenomenon / oscillation (ルンゲの現象)**

  High order (e.g. >3) polynomial will cause large oscillations at points other than the sample points (高次の多項式では標本点以外で大きく振動することがある)

***Ex.* Interpolate $f(x) = 1 / (1 + x^2)$ for (n_{order}+1) points in the range x = [-2, 2]**



In the machine learning (機械学習):

Overfitting (過適合), Overlearning (過学習)

Interpolation: Piecewise polynomial interpolation

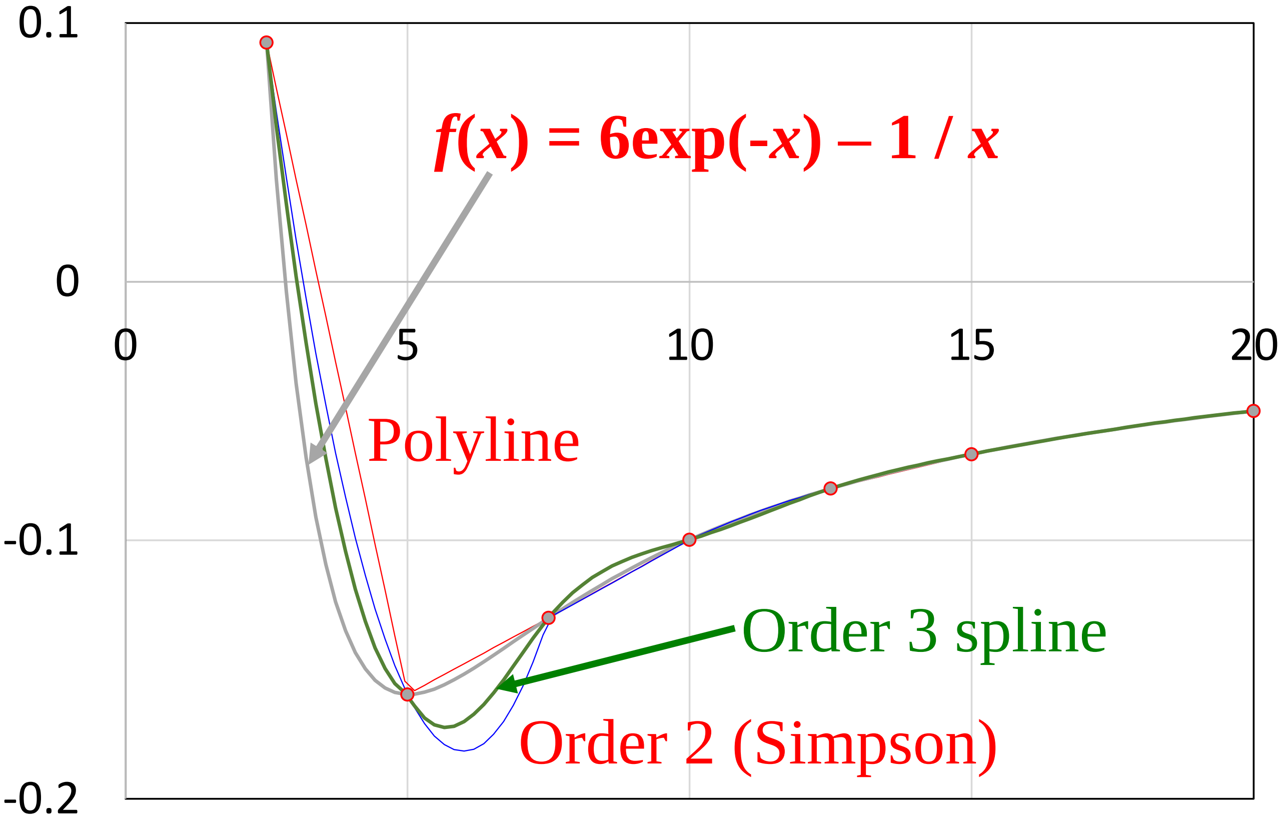
(区分多項式補間)

Connect divided sections by polylines (折れ線)

=> First derivatives will be discontinuous at the boundaries

=> (n – 1)-th derivatives are continuous for whole range:

Order *n* spline functions (n次のスプライン関数)



Smoothing

平滑化

Smoothing (平滑化)

Take some average for sample points

- **Moving average** (移動平均)
  - **Simple moving average** (単純移動平均)**:**
    Average of sequential data with the uniform weight
  - **Weighted moving average** (加重移動平均)**:**
    Average of sequential data with weight
    Weight：Linear, Triangular, Exponetnial, Gauss, etc…

Approximate sample points by some function

- **Polynomial smoothing** (多項式による平滑化)
- **Smoothing spline** (スプライン平滑化)
- **Least-squares method** (最小二乗法)

Other

- **Fourier transformation** (フーリエ変換)

Calculation

Simple moving average (2m+1 points)

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

Weighted moving average (2m+1 points)

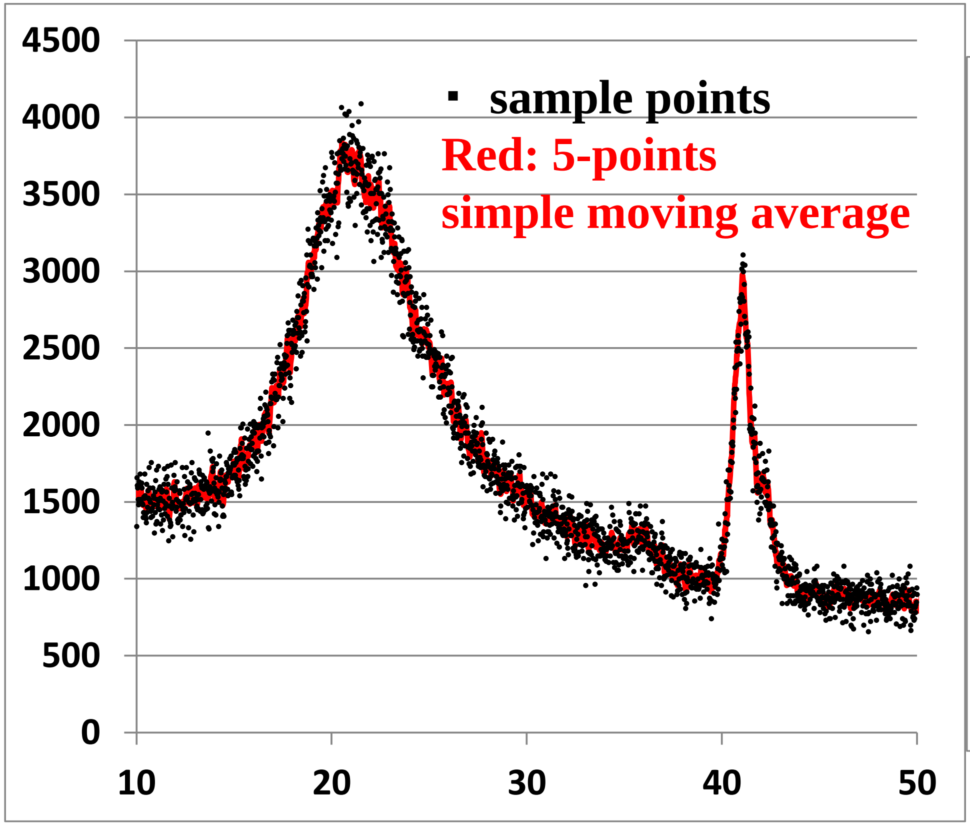
$$y_{i,smoothed} = \sum_{j=i-m}^{i+m} w_j y_j \,/ \sum_{j=i-m}^{i+m} w_i$$

Smoothing (平滑化)

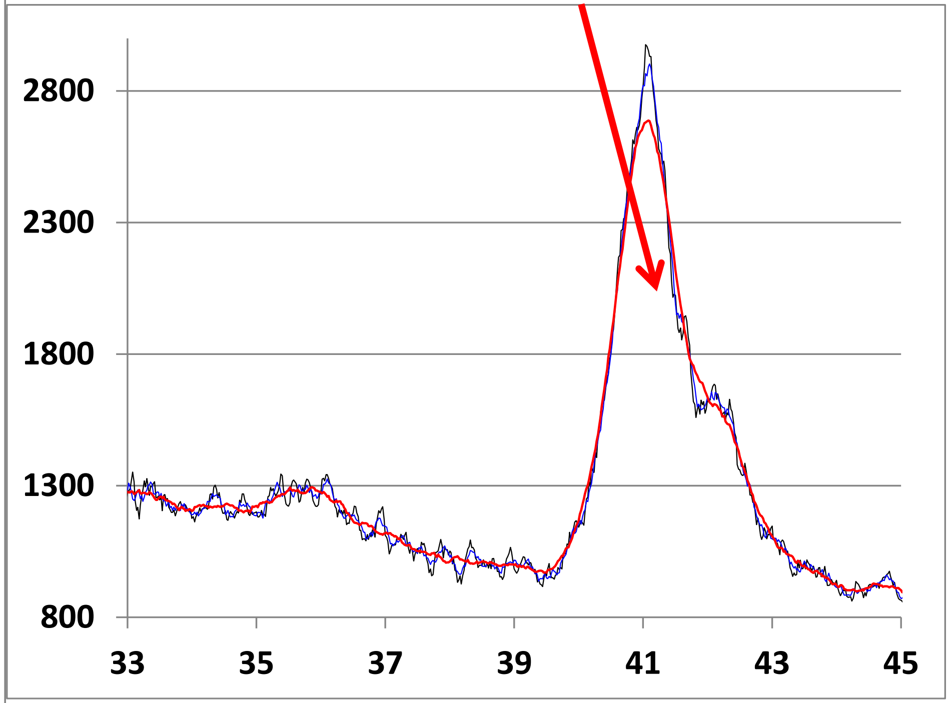
- **Moving average** (移動平均法)

  More smooth with more sample points for average,
  but would alter the function shape if the function is not monotonic.
  => Affect peak height, valley depth, peak width etc…
  The range of averaged sample points larger than the peak width
  => split peaks might become difficult to be separated.

Poor S/N ratio XRD pattern



Simple moving average: 5,11,31点

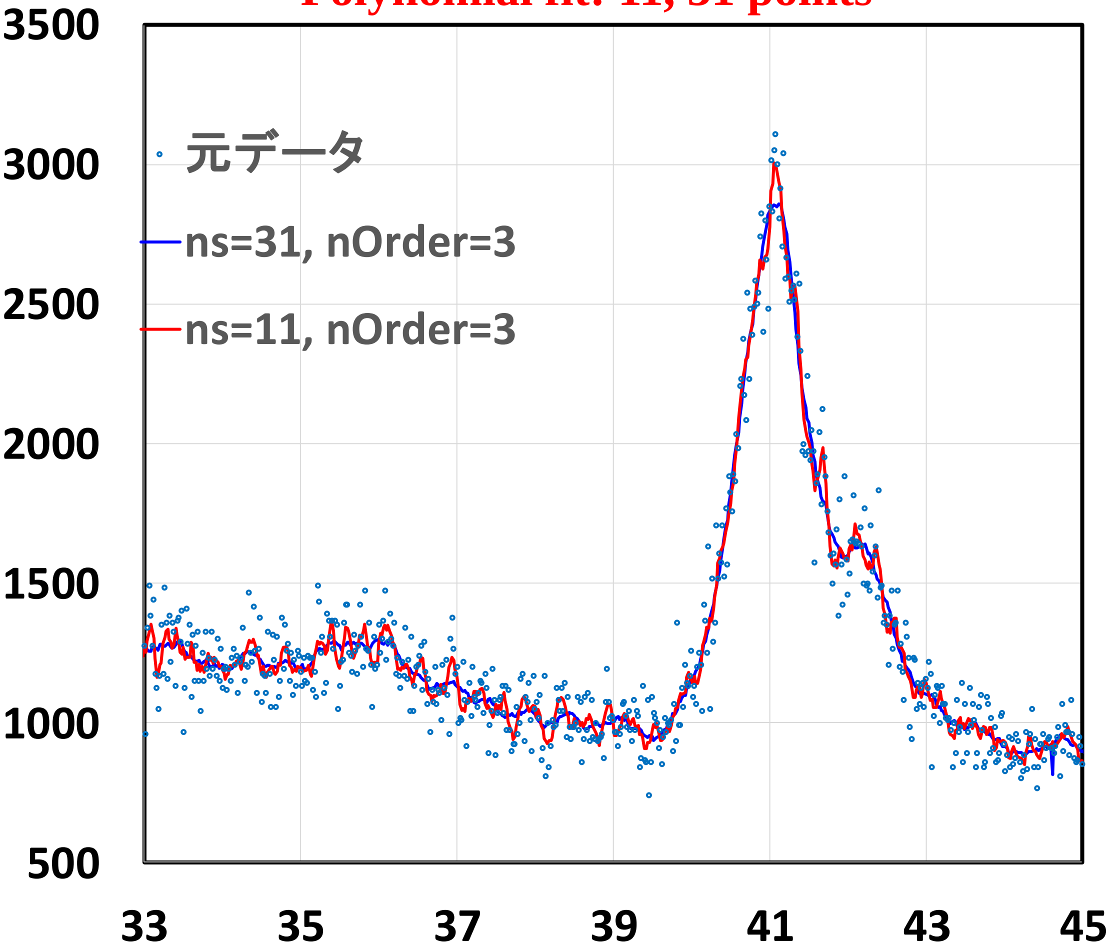


Smoothing: Polynomial fit method (多項式適合法)

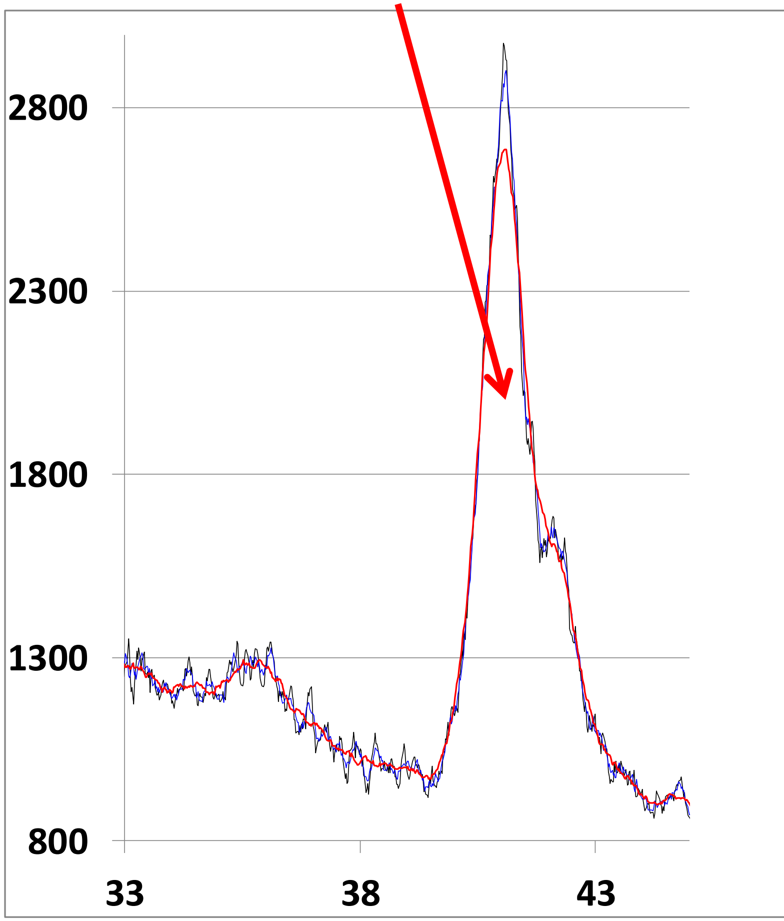
Adopt n_{order} order polynomial to n_s sample points among the given n sample points, determined by LSQ

データに n_{order}次多項式を最小自乗法で求め、標本点の値を内挿する

Polynomial fit: 11, 31 points



Simple moving average: 5, 11, 31 points



Weights of polynomial fit (Savizky-Golay method)

多項式適合法 (Savizky-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 (2*m*+1) points ((2*m*+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, ･･･, -1, 0, 1, ･･･, m

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

Calculation

Simple moving average (2m+1 points)

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

Weighted moving average (2m+1 points)

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

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

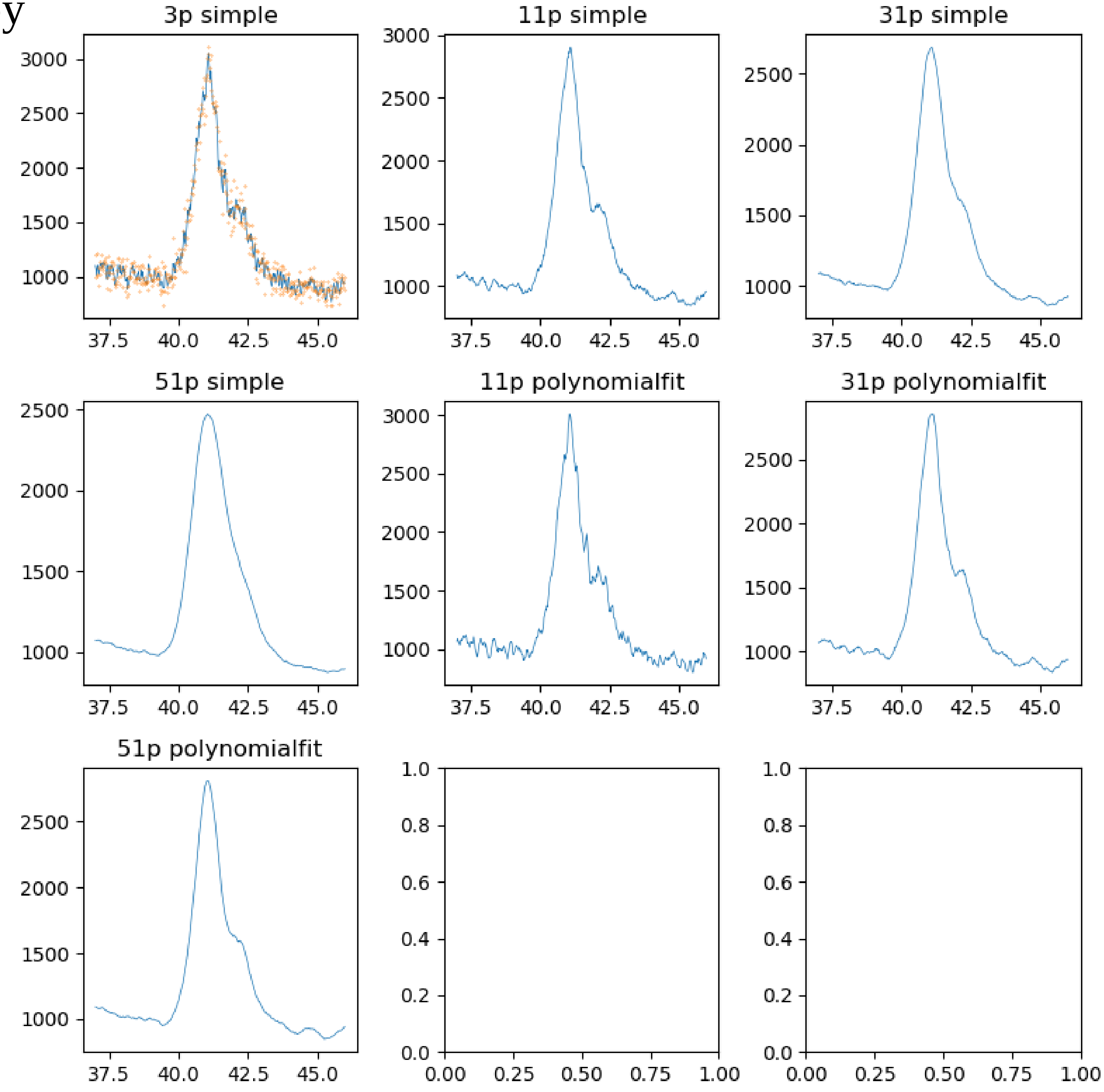
$w_{23}(j) = 3m(m+1) - 1 - 5j^2$ j = -m, ⋯, -1, 0, 1, ⋯, m

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

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

Program: smoothing.py

Usage: python smoothing.py



Fourier transformation (フーリエ変換)

Different definitions

FT (フーリエ変換) $F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(i\omega t)dt$

IFT (逆フーリエ変換) $f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\exp(-i\omega t)d\omega$

FT $F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(i2\pi ft)dt$

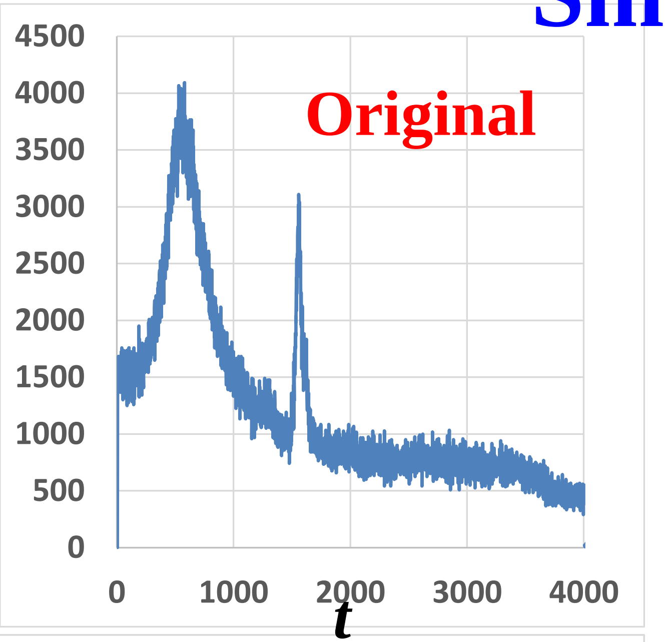
IFT $f(t) = \int_{-\infty}^{\infty} F(\omega)\exp(-i2\pi ft)d\omega$

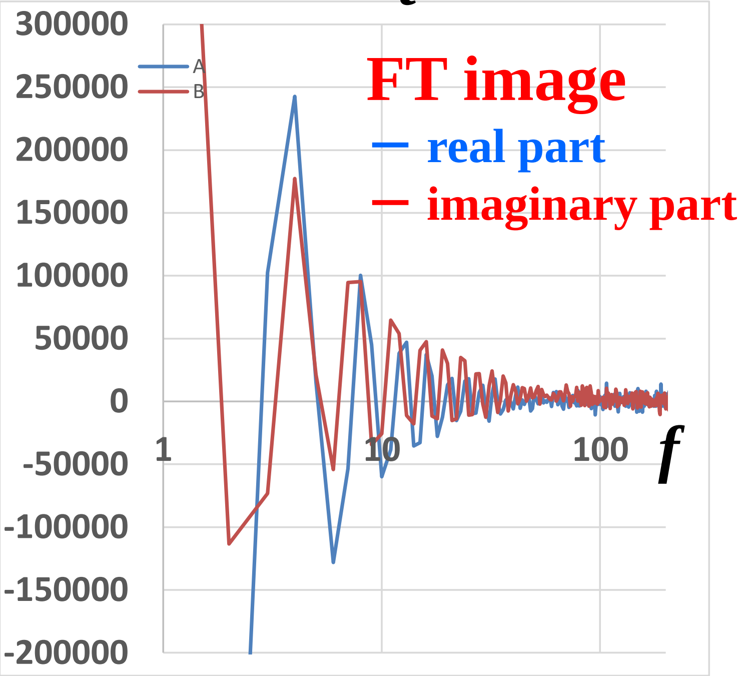
Features of Fourier transformation

- **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

Smoothing: FT





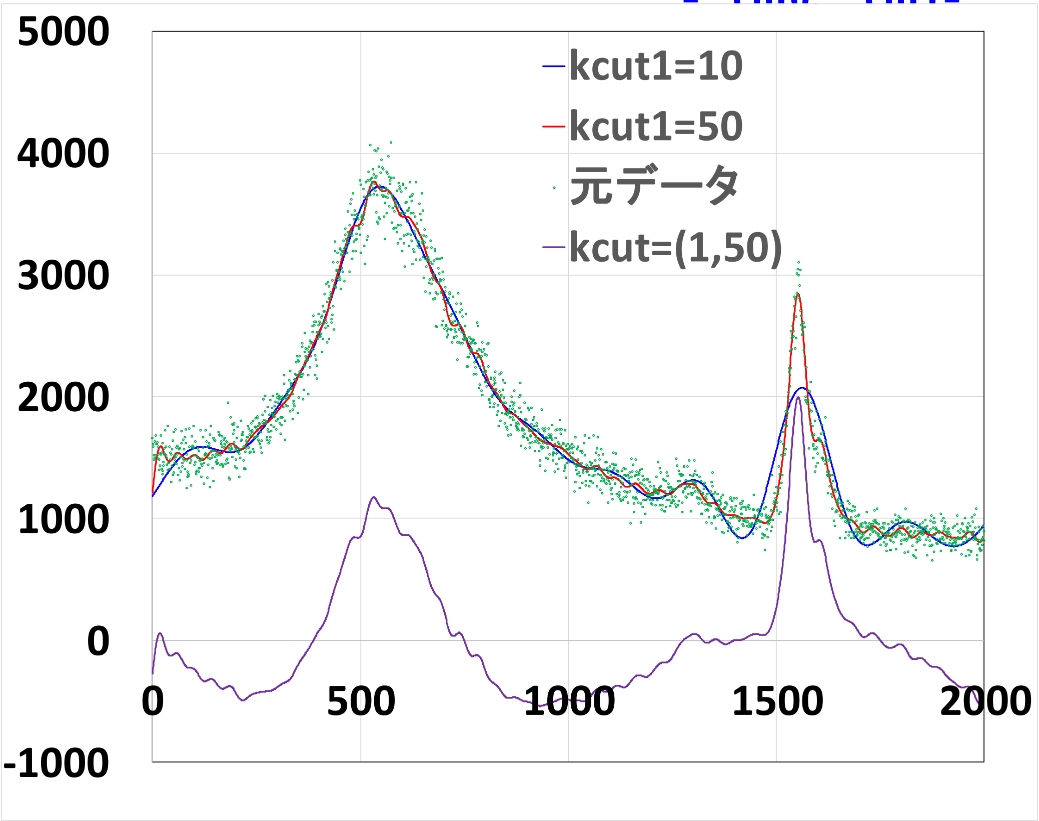
Remove high-frequency FT data: Smoothing

Low-pass filter

Remove low-frequency FT data: Cut drift

High-pass filter

***Ex*. Cut FT data outside [k_{cut0} k_{cut1}]**

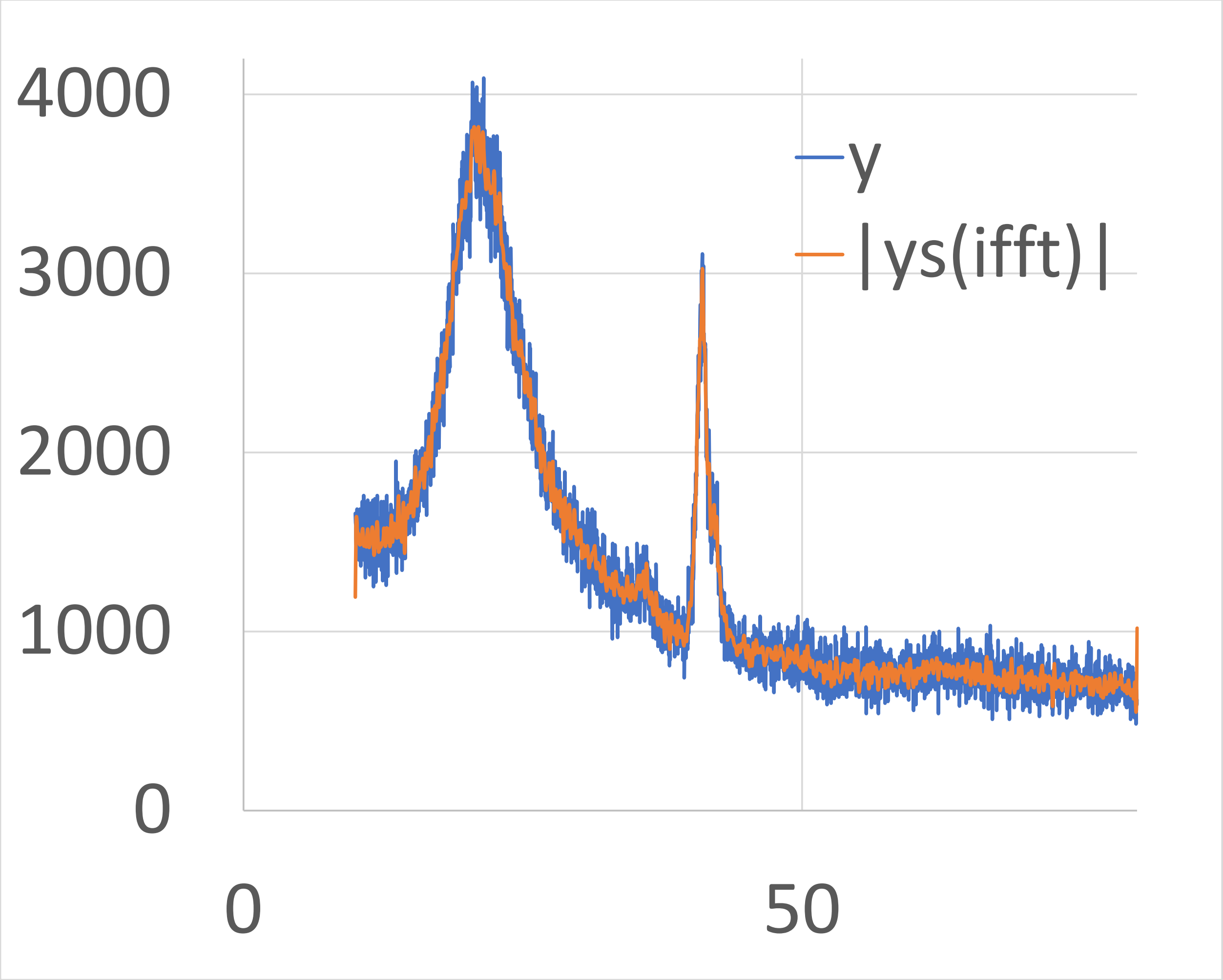


Program: smoothing-fft.py

Usage: python smoothing-fft.py xrd.csv 0 5

(note: the x range is different from the previous slide)

=> plot smoothing-fft.csv

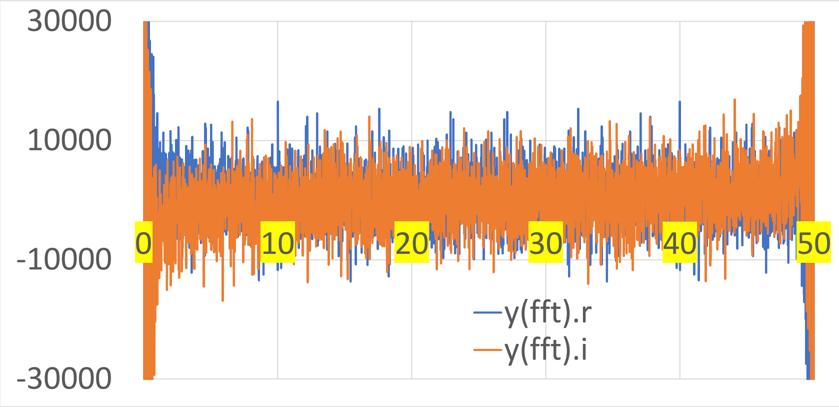


Note for FFT

smoothing-fft.py

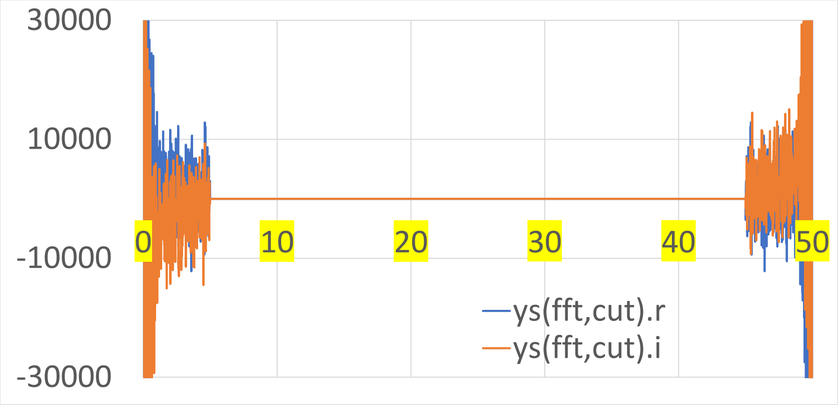
Numpy fft module: F = np.fft.fft(y) FFT

FFTed result is symmetric at the center of the inveterted x axis at $i_x = n_x/2$.



For smoothing, cut the data in i_x = [0, i_{xLF}], [i_{xHF}, $n_x/2$]

and [n_x–1, n_x–1–i $_{xLF}$], n_x/2+1, n_x–1–i $_{xHF}$ i_{xHF}].



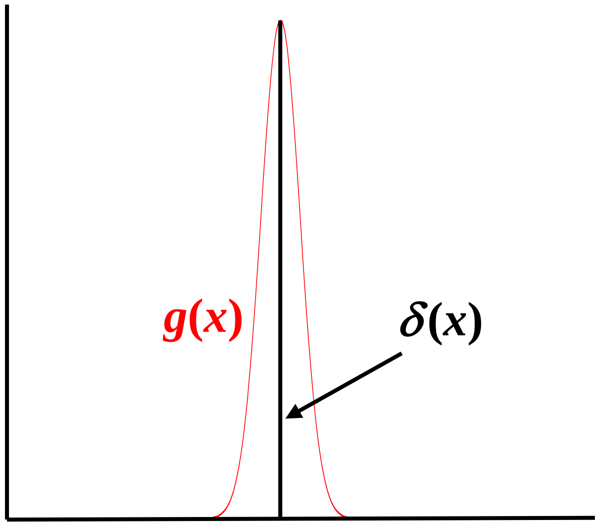
, then perform IFFT by fs = np.fft.ifft(Fs)

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 δ(*x*))

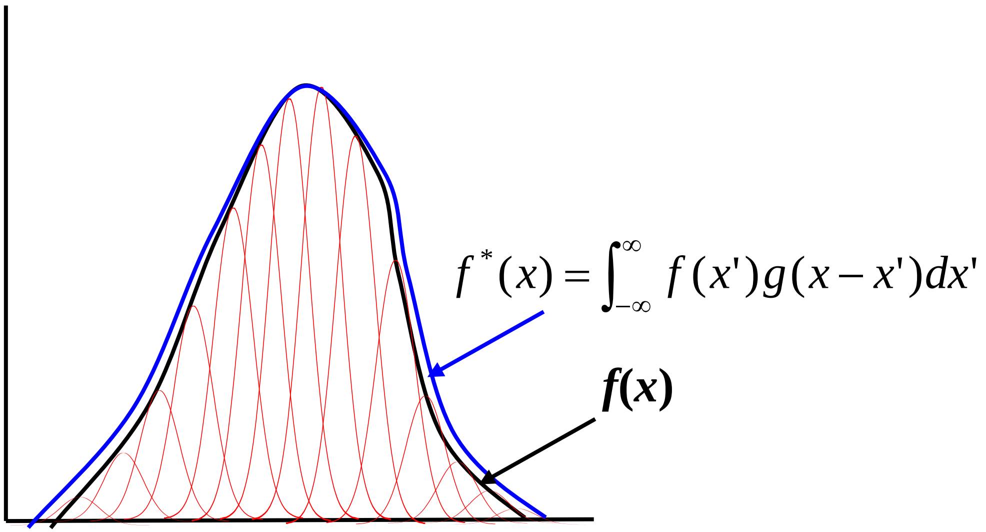
試料本来のデータは線幅ゼロ (δ関数) でも、
測定値は装置関数 *g*(*x*) の広がりを持つ



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

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*).**

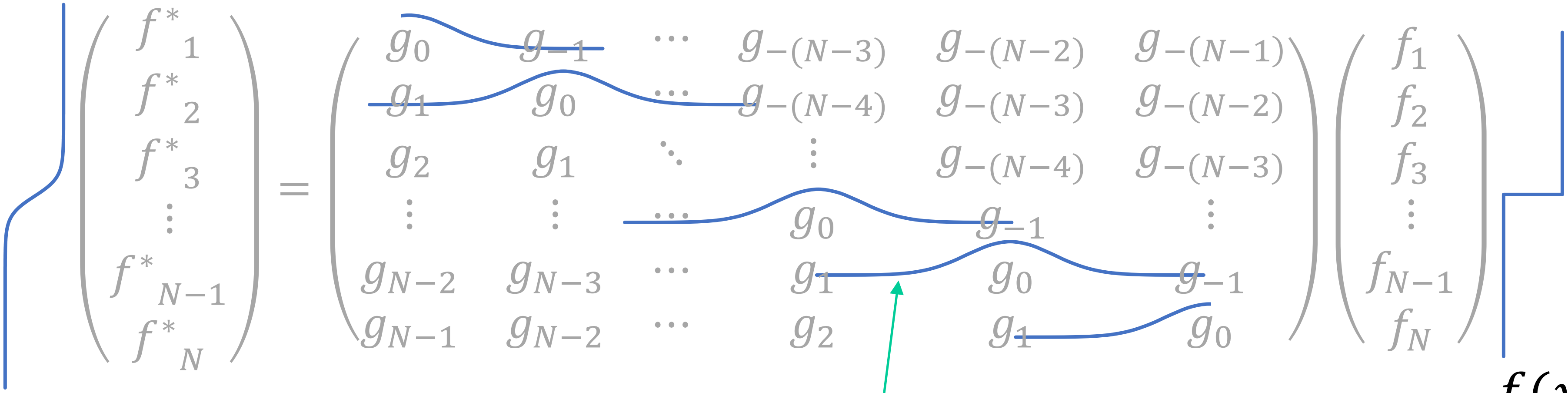
試料本来のデータは *f*(*x*) でも、測定されるのは
装置関数 *g*(*x*) の畳み込みをした $f^*(x)$



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 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)

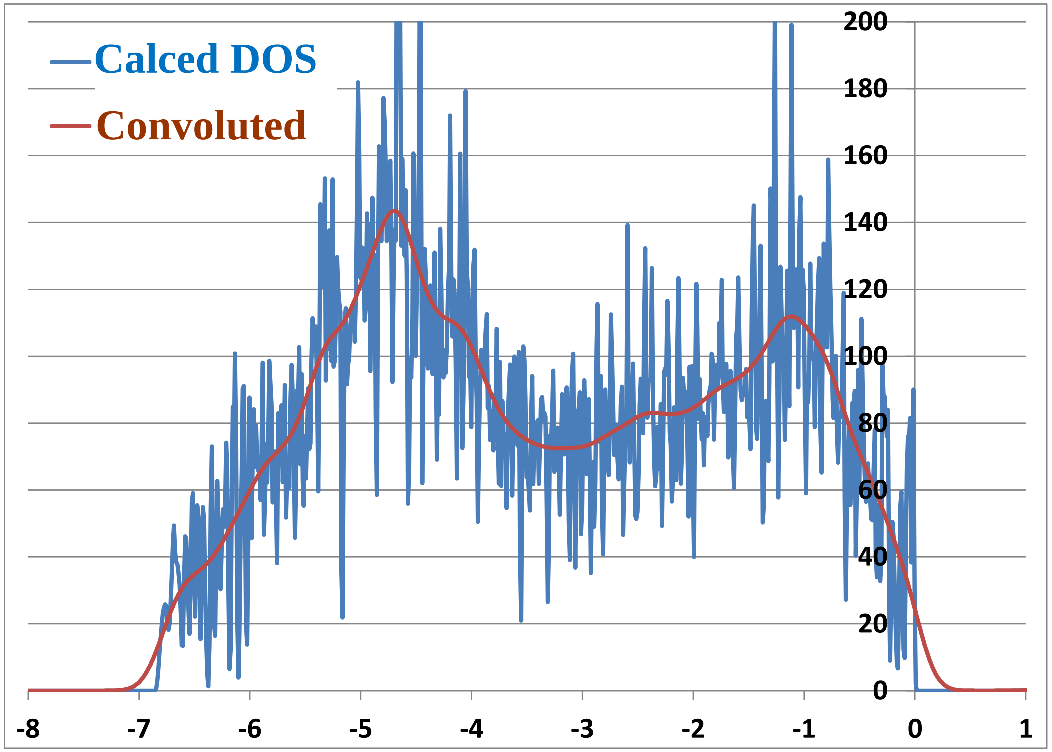
Density of state (DOS) function calculated by density functional theory

密度汎関数計算で得たa-InGaZnO$_4$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)

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



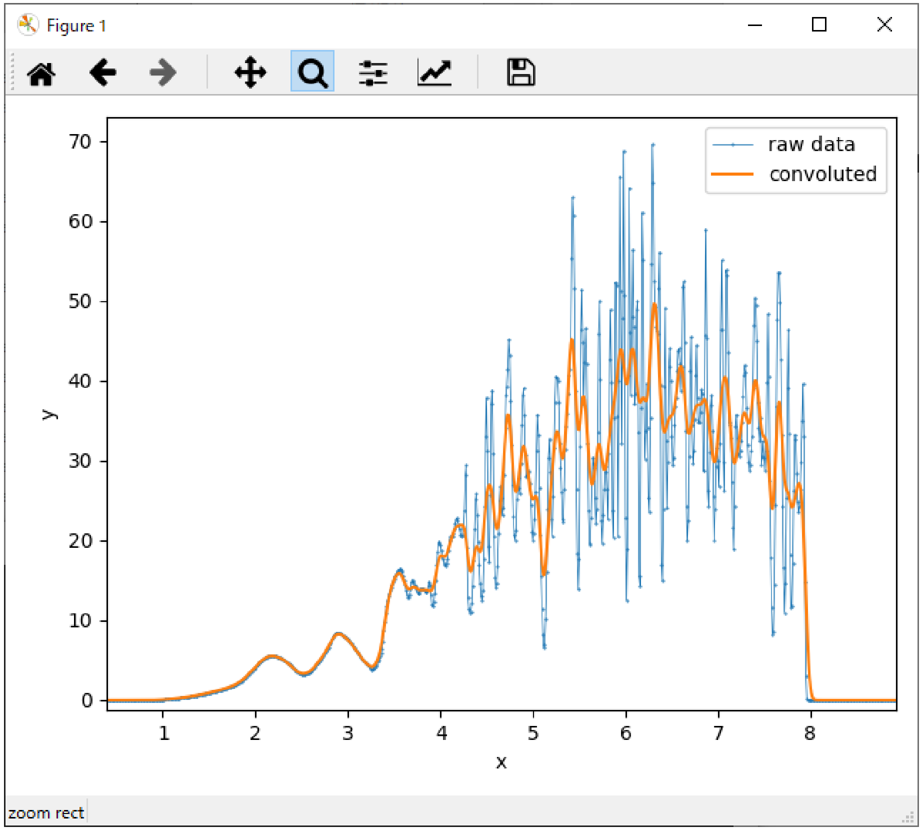
Note: Estimation of band, edge energies will have the errors originating from the smearing width *w*

Program: convolution.py

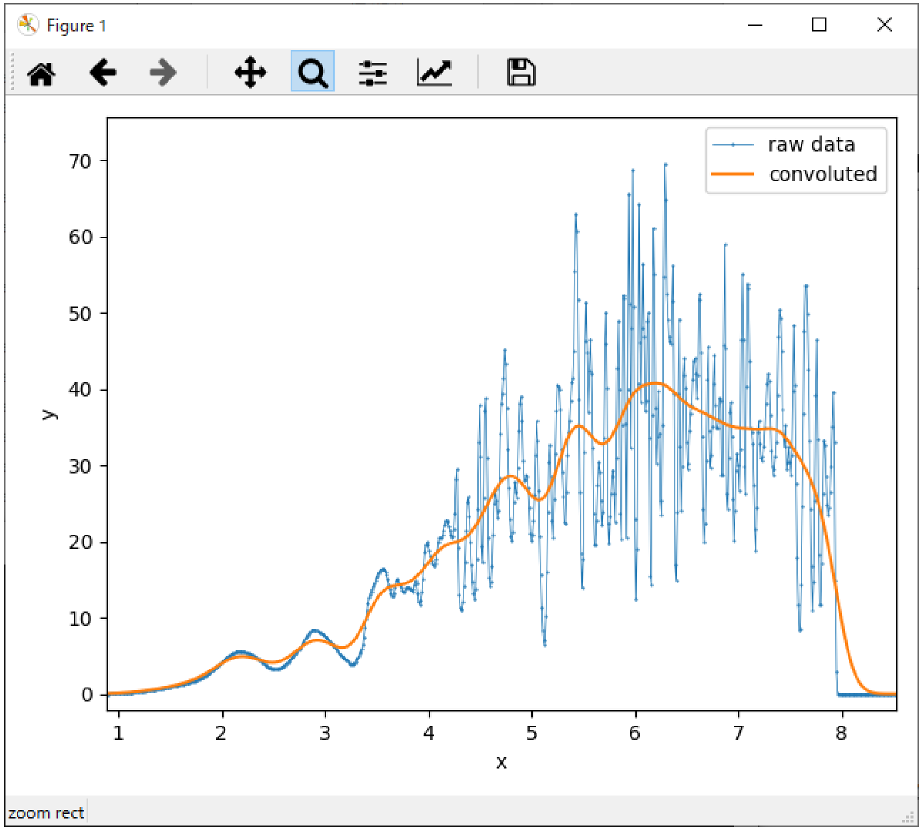
Usage: python convolution.py width

width: width of Gaussian function to convolute

python convolution.py 0.05



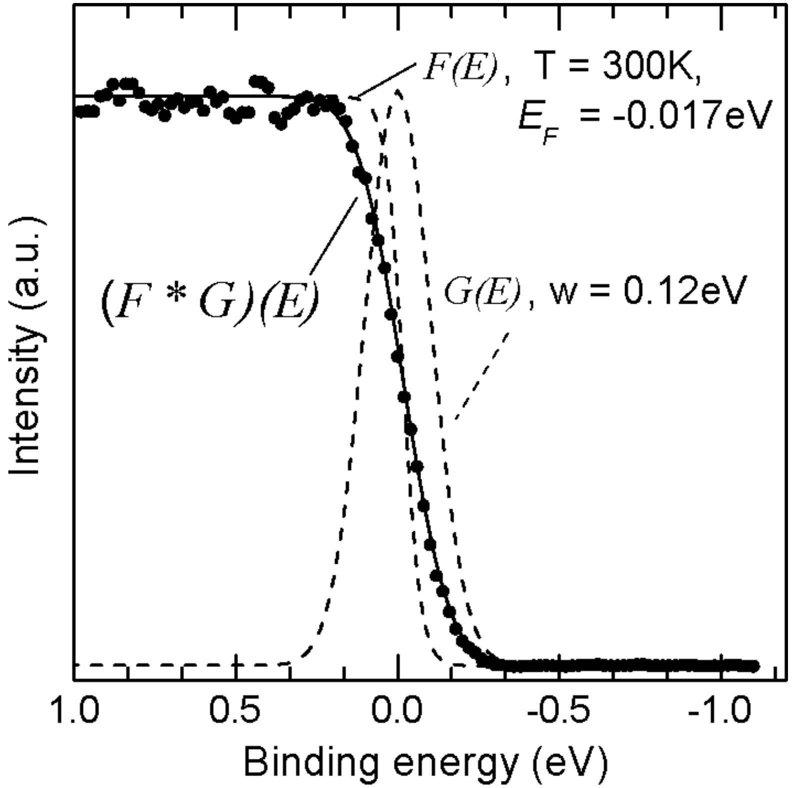
python convolution.py 0.2



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

$f(E) = 1/(1+\exp[(E-E_F)/k_BT])$ 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 eV

$S(E)$ is obtained by deconvolution from $G(E)$

$G(E)$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

Convolution:
Matrix product of
data vector and filter

First differential

$$\frac{dy}{dx_2} = \boxed{\underset{\text{Filter}}{\frac{1}{2h}\begin{pmatrix}-1 & 0 & 1\end{pmatrix}}}\begin{pmatrix}y_1\\ y_2\\ y_3\end{pmatrix}$$

Second differential

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

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 (2*m*+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

pythonライブラリィによる平滑化

Smoothing. py

単純移動平均

# 1/N の重みのフィルターを作る

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

# コンボリューション

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

多項式適合化平滑化: **Savizky-Golayフィルター**

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

nsmooth: 平滑化点数

norder: 多項式の次数

deriv: 微分次数

注意: savgol_filter() では deriv = 1とすると1次微分を取ってくれるが、x軸のデータを与えていないので、絶対値は異なる。

絶対値が必要な場合、平滑化した後、h^{deriv} で割ること

注: savgol_diff_test.py で限定的に確認した範囲です

多次元フィルター、コンボリューションと画像解析

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

データ$\begin{pmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{pmatrix}$と

フィルタ$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$の

各要素の積の和を

y_{22}のコンボリューションにとる

数学のデコンボリューションとは異なるので注意

X軸微分フィルター (エッジ検出) $\frac{1}{2h}\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$

Y軸微分フィルター (エッジ検出) $\frac{1}{2h}\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$

斜め微分フィルター (エッジ検出) $\frac{1}{2h}\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}$

平滑化フィルター (ぼかし) ⇔ デコンボリューションにより sharpeningができる

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

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

Neural network の中段にフィルターによる畳み込み層を入れて、
フィルターの要素の値を学習させる

学習したフィルターを見ることで、どのような処理が必要か理解できることもある

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)

$$f(x) = \int_{-\infty}^{\infty} F(k)\exp(-ikx)dk$$

$$g(x) = \int_{-\infty}^{\infty} G(k')\exp(-ik'x)dk'$$

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

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

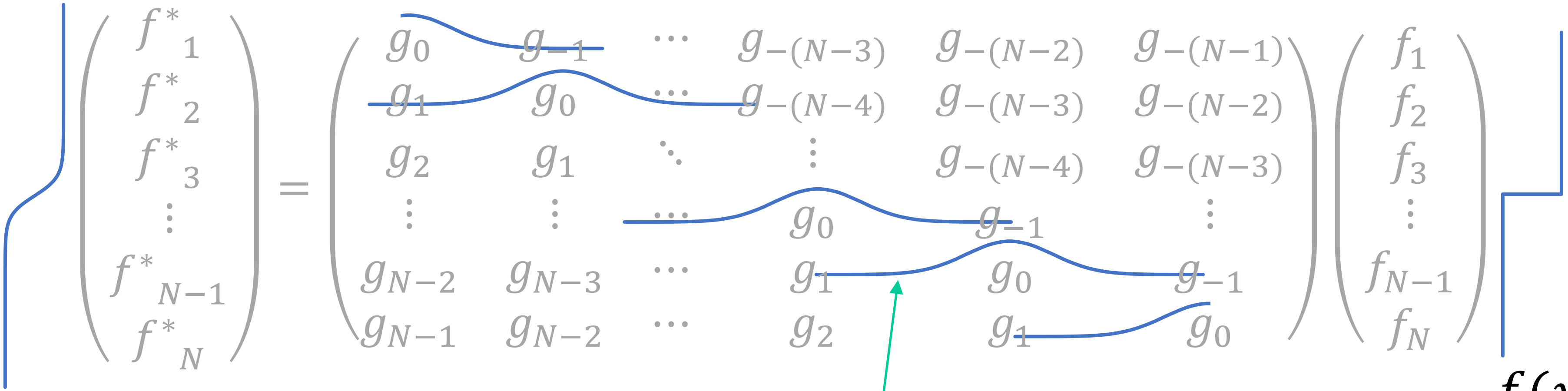
F(*k*) = *F**(*k*) / *G*(*k*)を計算して逆フーリエ変換で *f*(*x*) が得られる

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

$$x_1^{(k+1)} = \left(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}\right)/a_{11}$$
$$x_2^{(k+1)} = \left(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}\right)/a_{22}$$

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

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

$$x_1^{(k+1)} = \left(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}\right)/a_{11}$$
$$x_2^{(k+1)} = \left(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}\right)/a_{22}$$

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

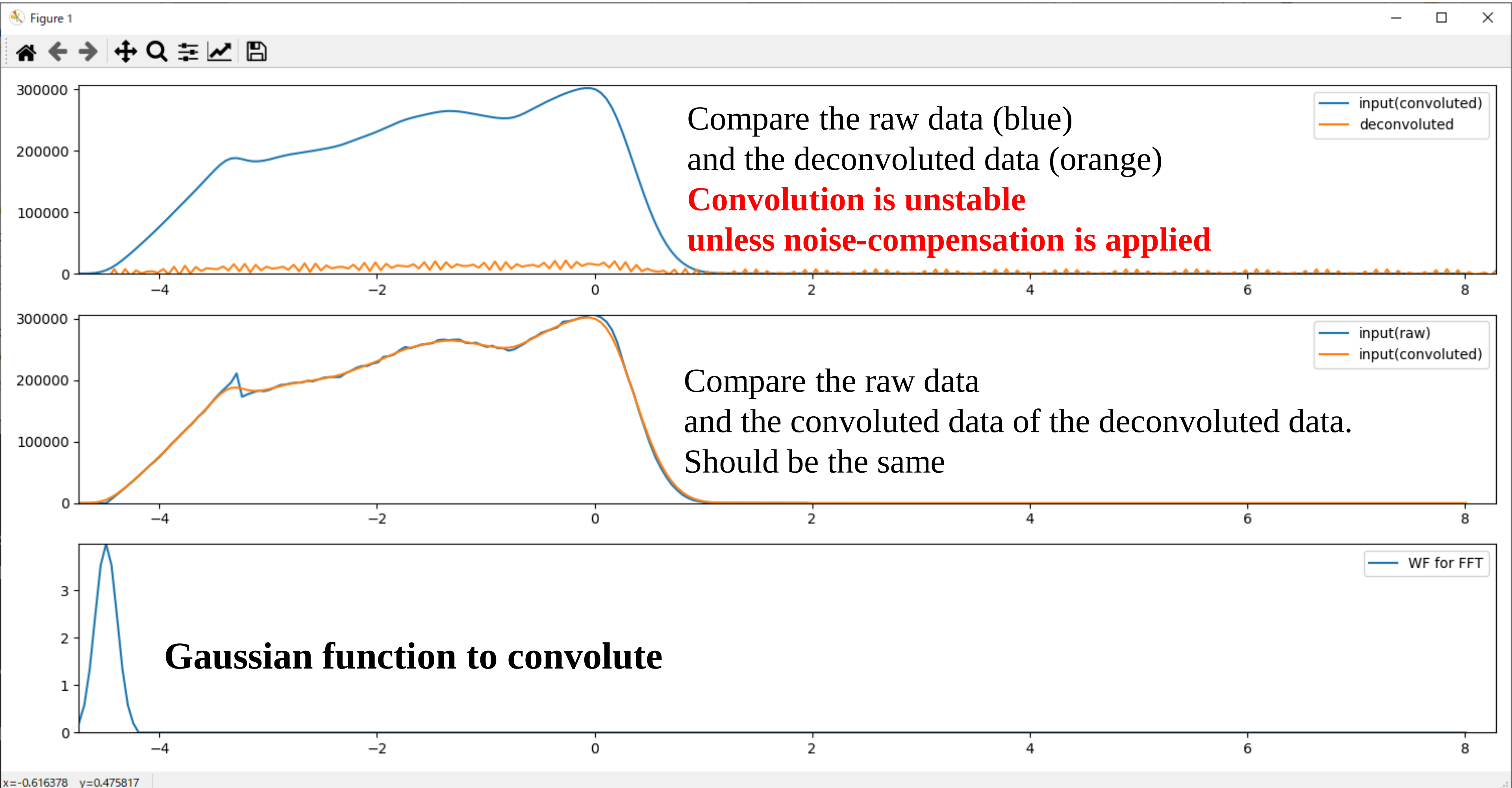
Program: deconvolution.py

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**



Compare the raw data (blue)
and the deconvoluted data (orange)
Convolution is unstable
unless noise-compensation is applied

Compare the raw data
and the convoluted data of the deconvoluted data.
Should be the same

Gaussian function to convolute

Deconvolution: Gauss-Seidel method w/o smooting

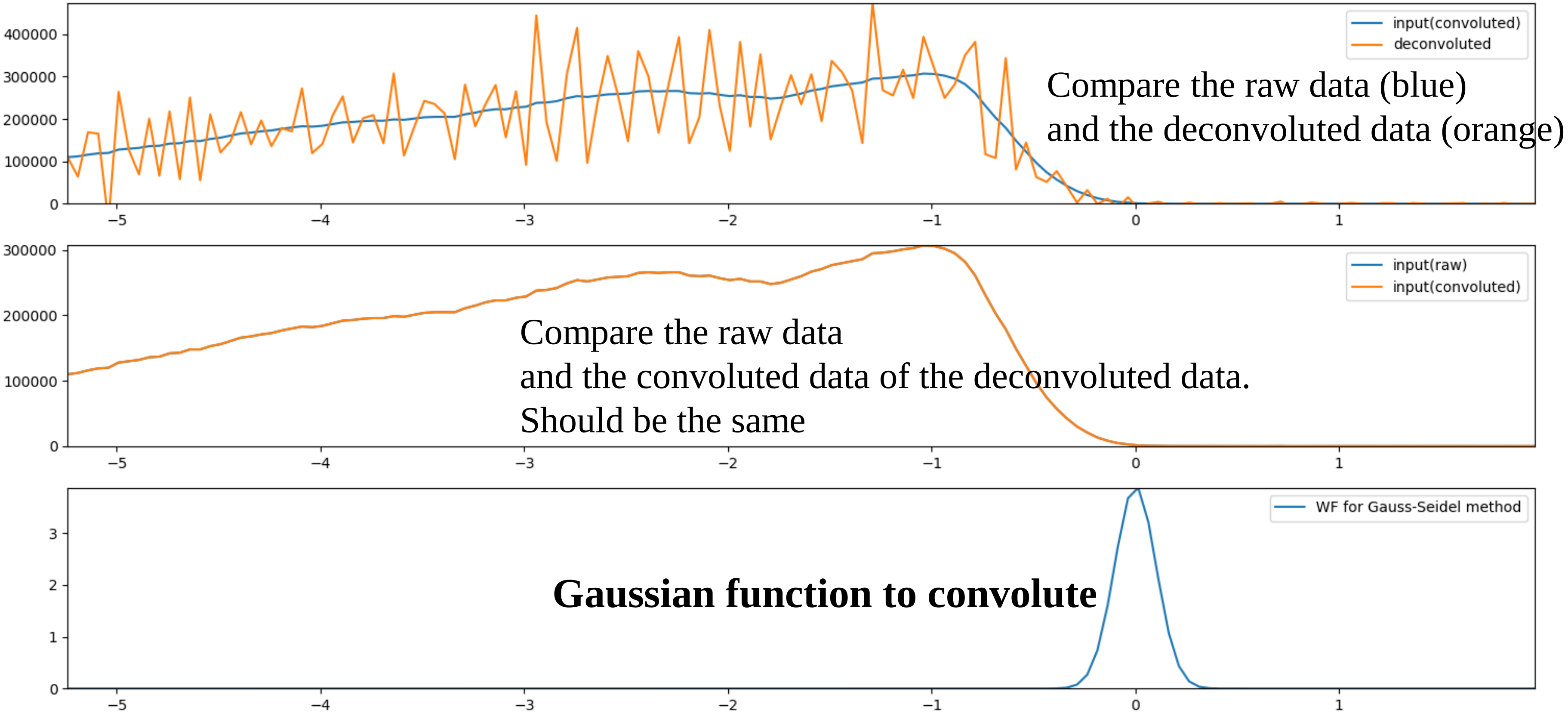
Usage: python deconvolution.py file mode xmin xmax Wa dump nmaxiter eps nsmooth zeroc

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.



Compare the raw data (blue)
and the deconvoluted data (orange)

Compare the raw data
and the convoluted data of the deconvoluted data.
Should be the same

Gaussian function to convolute

Program: Gauss-Seidel method with smoothing

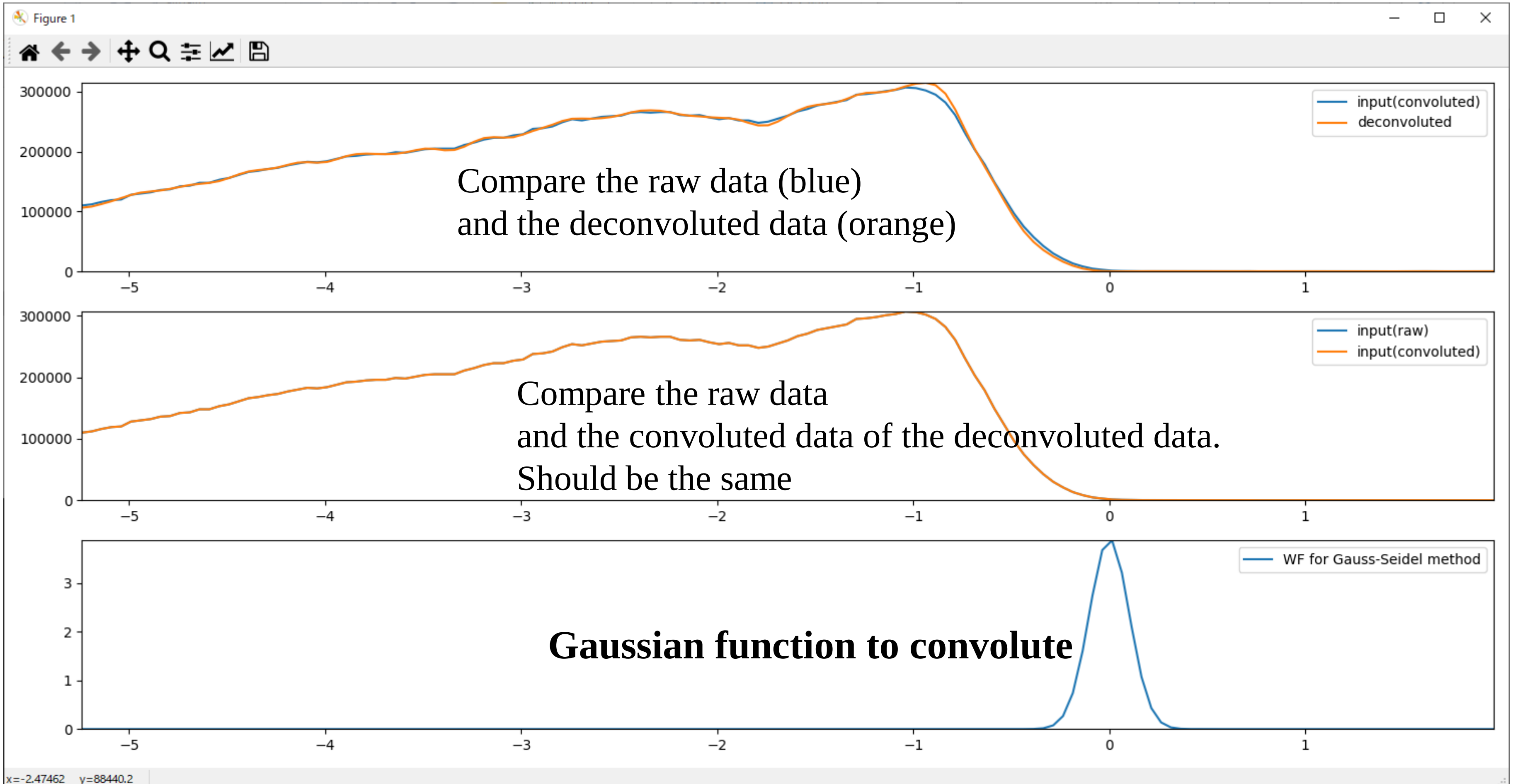
Usage: python deconvolution.py file mode xmin xmax Wa dump nmaxiter eps nsmooth zeroc

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.



Compare the raw data (blue)
and the deconvoluted data (orange)

Compare the raw data
and the convoluted data of the deconvoluted data.
Should be the same

Gaussian function to convolute

Linear least squares method (*l*LSQ)

線形最小自乗法

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), ･･･ (x_n, y_n) follow *f(x)* = *a* + *bx*,

※ Error ε_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 $\max_{a \le x \le b} |g(x) - f(x)|$

Minimize L1 norm **:** $S = \Sigma|f(x_i) - y_i|$

Least-squares (LSQ) method (最小自乗法) (L2 norm) **:** $S = \Sigma(f(x_i) - y_i)^2$

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

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

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

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

Even for *f(x)* = *a* + *bx* + *cx*² +･･･, only one matrix operation can give a final solution

ミニマックス近似:多項式

入門 数値計算

$$\max_{a\le x\le b}\left|g(x)-f(x)\right|$$ を最小にする

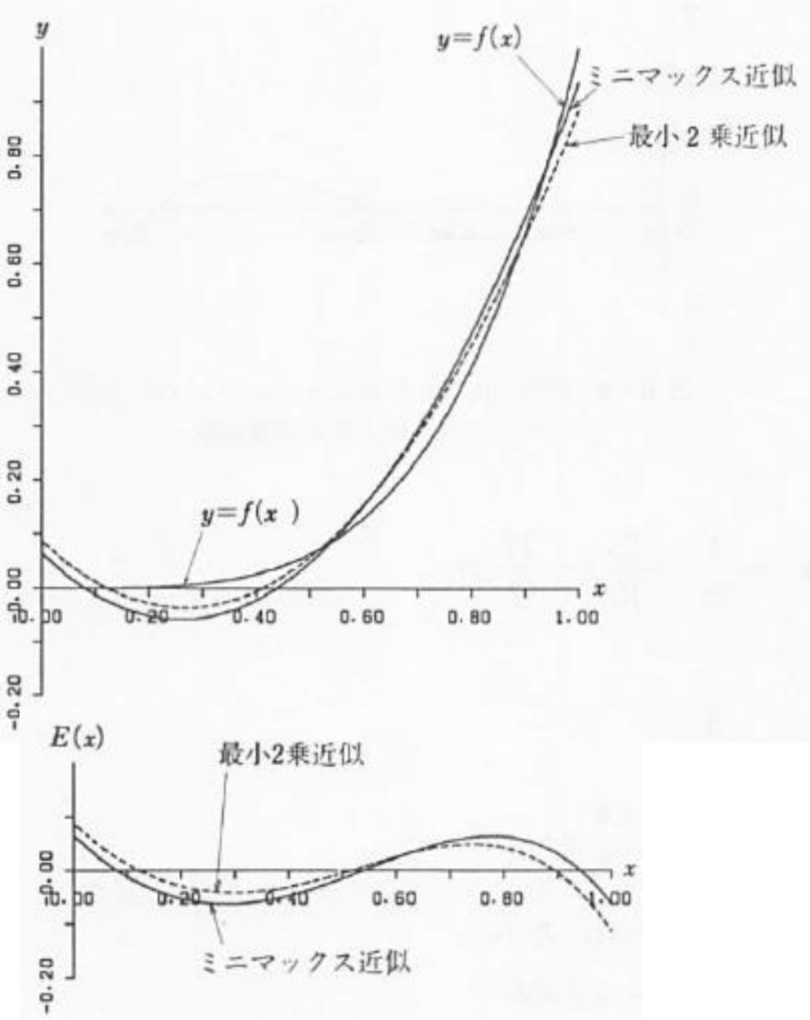


図6・3 ミニマックス近似と最小2乗近似

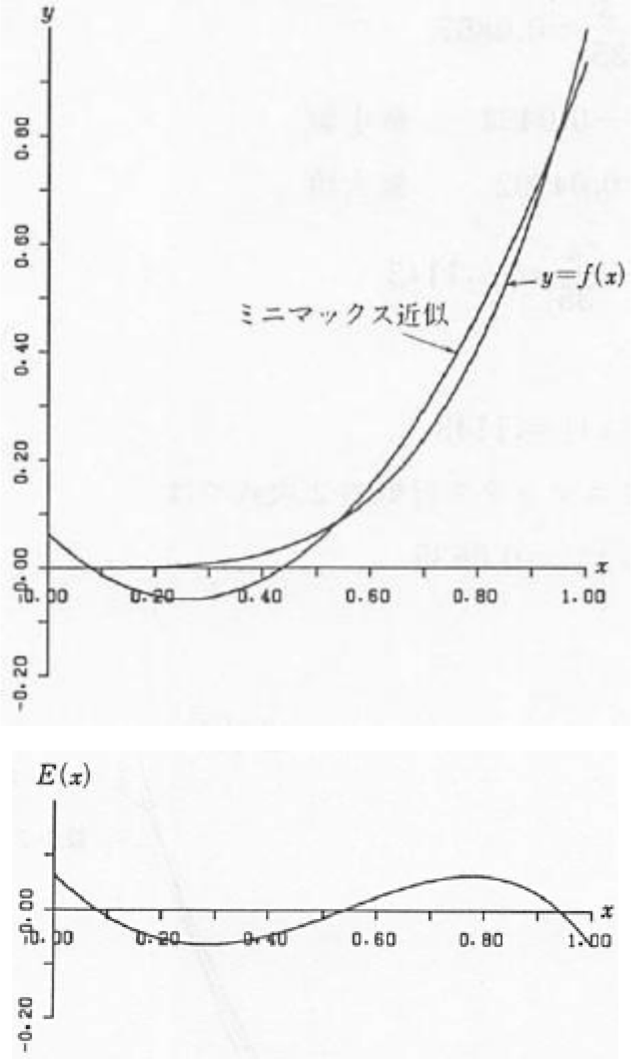


図6・2 区間 [0, 1] における $f(x)=x^4$ の2次の ミニマックス近似とその誤差曲線

*l*LSQ: Polynomial

線形最小二乗法: 多項式

$$f(x) = \sum_{k=0}^{n} a_k x^k \qquad S = \sum_{i=1}^{N} \left(y_i - \sum_{k=0}^{n} a_k {x_i}^k \right)^2$$

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

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

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

$|x_i| > 1$ might cause overflow,

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

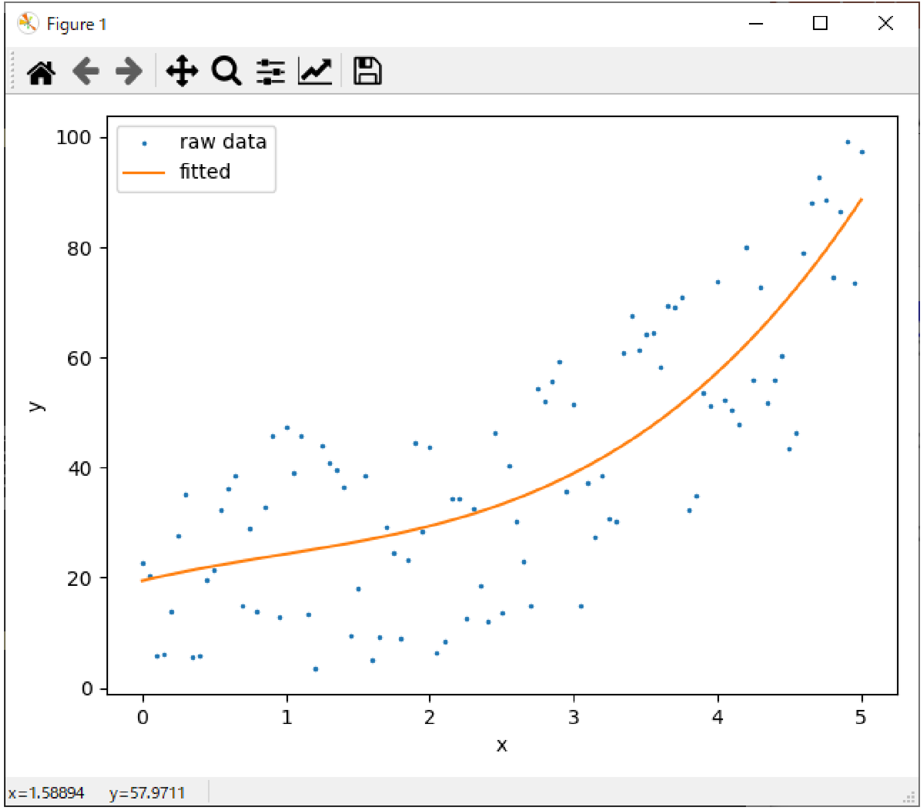
=> Normalize the *x* range e.g. to [-1, 1] : $x_i' = 2\frac{x_i - x_{\mathrm{mid}}}{x_{\mathrm{max}} - x_{\mathrm{min}}}$

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

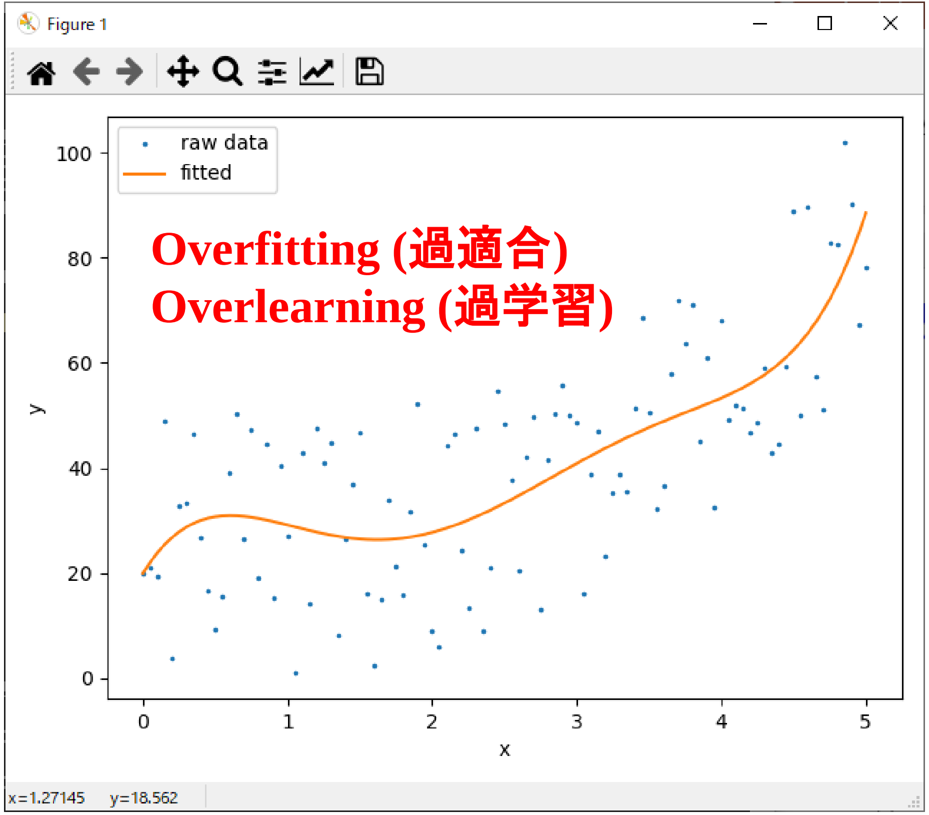
Program: lsq-polynomial.py

Usage: python lsq-polynomial.py n_{order}

python lsq-polynomial.py **3**



python lsq-polynomial.py **6**



Overfitting (過適合)
Overlearning (過学習)

*l*LSQ: General functions

線形最小二乗法: 一般関数の場合

$$f(x) = \sum_{k=1}^{n} a_k f_k(x) \qquad S = \sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{n} a_k f_k(x_i) \right)^2$$

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

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

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

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

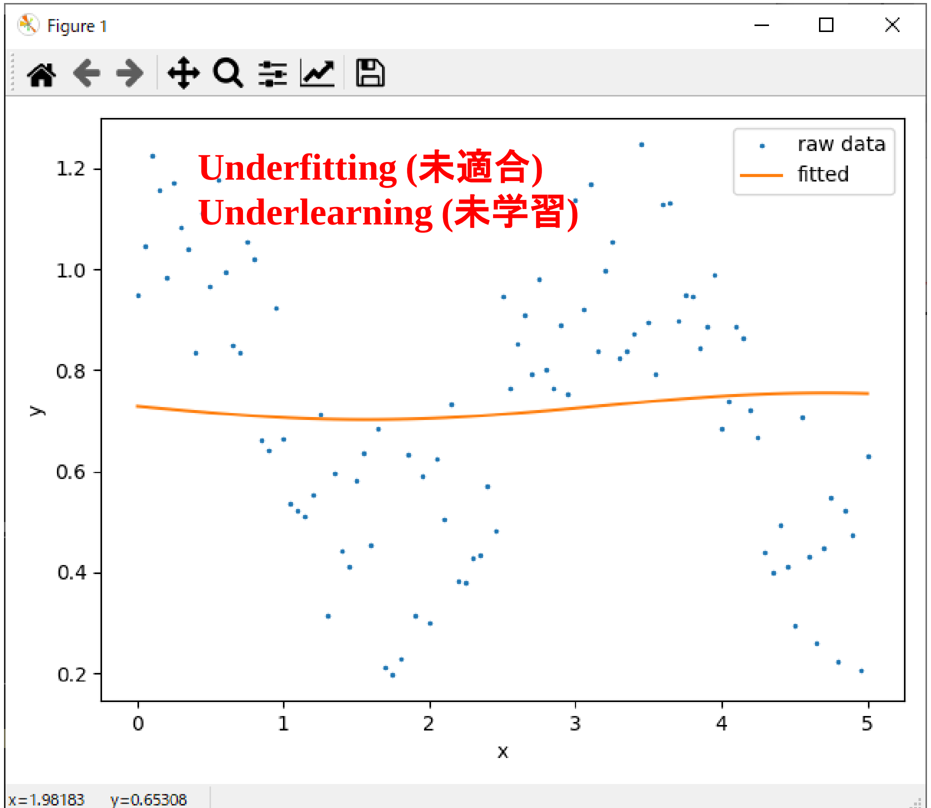
Program: lsq-general.py

Usage: python lsq-general.py n_{func}

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**

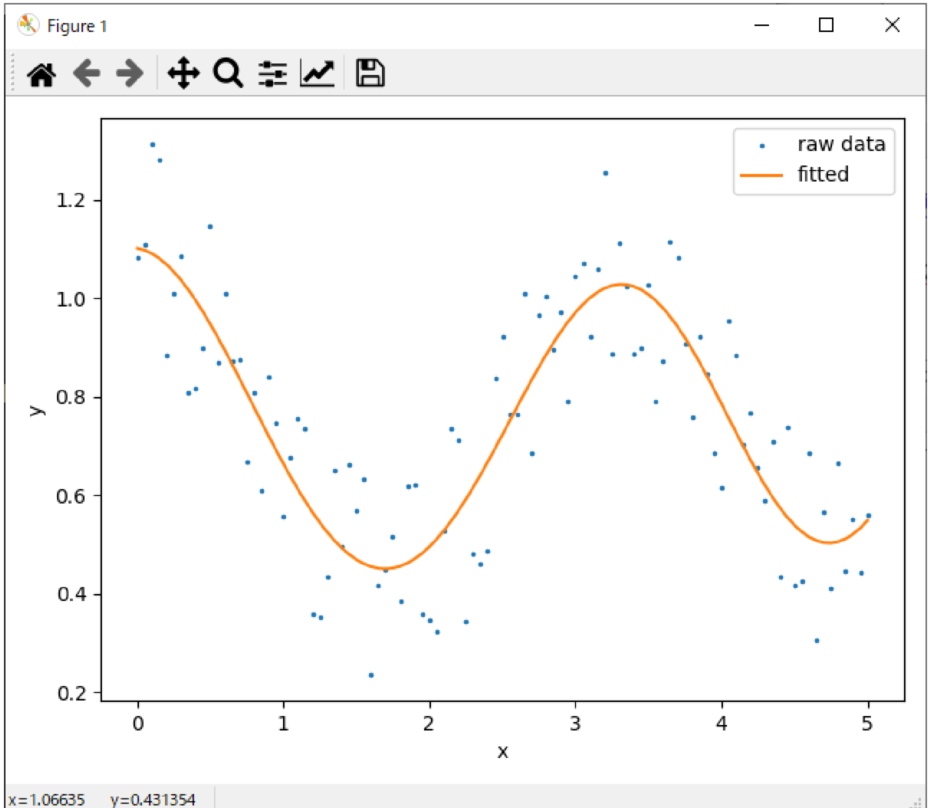
$y = 0.740 + 0.000432 \sin(x)$



Underfitting (未適合)
Underlearning (未学習)

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 *l*LSQ: Lattice spacing of triclinic lattice

(三斜晶結晶の面間隔)

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

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

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

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

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

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

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

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

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

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

1. S_{ij} is obtained by *l*LSQ
2. S_{ij} => Reciprocal lattice parameters (a^*, b^*, c^*, α^*, β^*, γ^*)
3. => Lattice parameters (a, b, c, α, β, γ)

How to solve equations?

Self-consistent method

自己無撞着法

Linear least-squares method

3rd order polynomical eq.

・Final solution is obtained just by one step calculation

・Unique solution

Four 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: $x_2 = f(x_1)$ ….

Difficult to converge: Diverge, Oscillation

(収束しにくい：発散、振動)

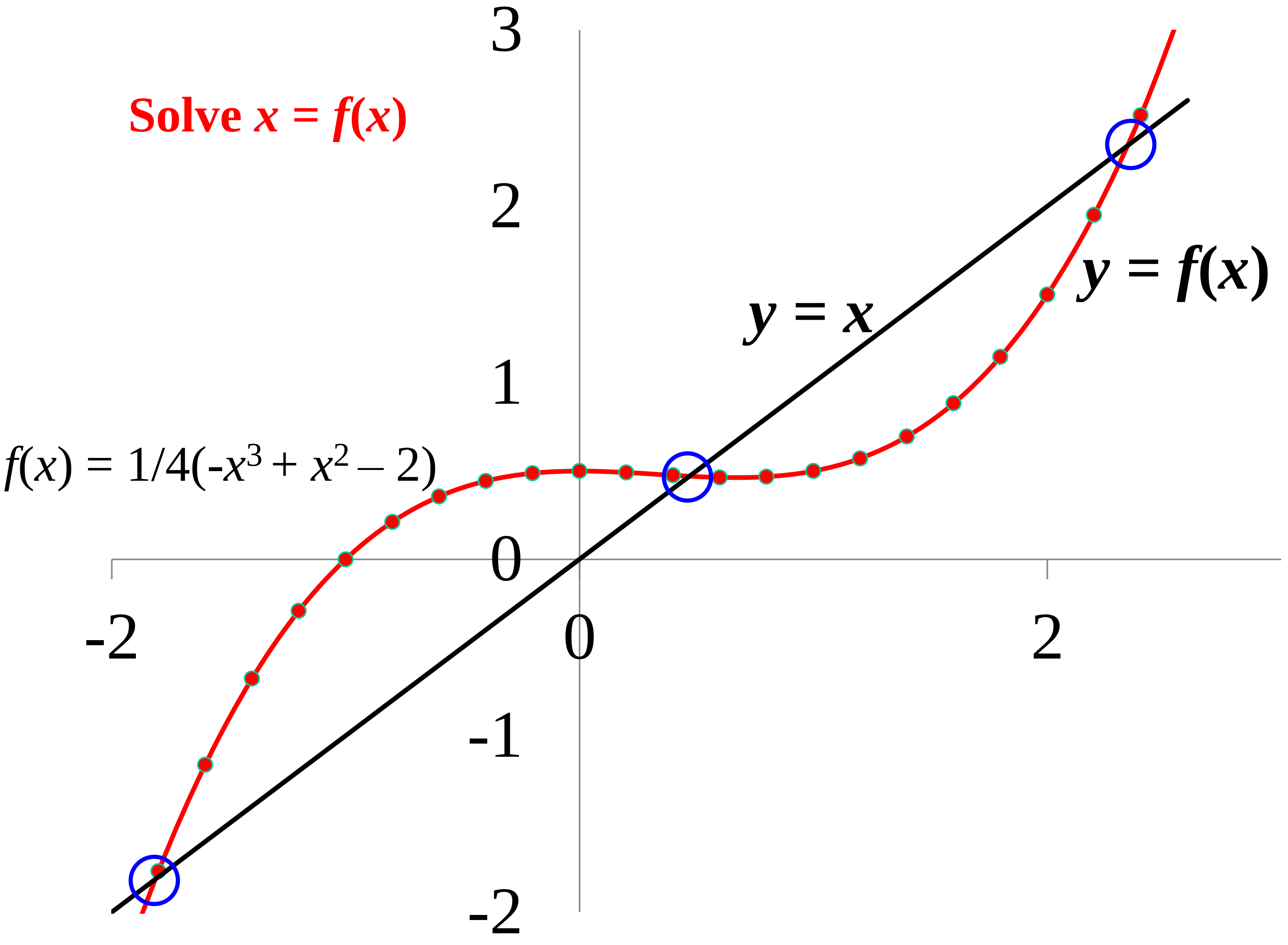
Mixing factor (混合係数) **k_{mix}: Stabilize convergence**

Initial value x_0

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

2nd iteration: $x_2 = f(x_1')$ ….

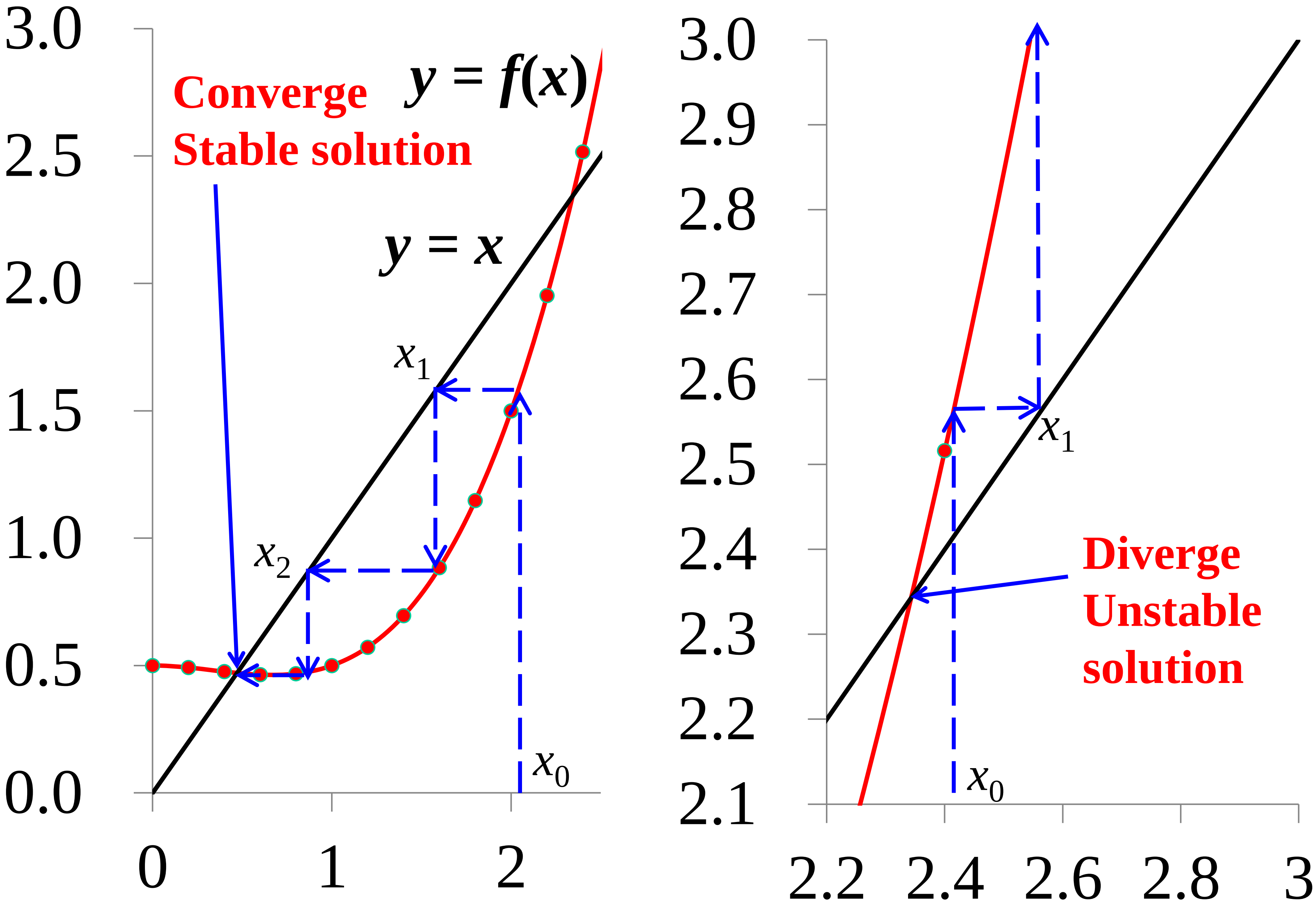
Illustrative explanation of SC



Solve $x = f(x)$

$f(x) = 1/4(-x^3 + x^2 - 2)$

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

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

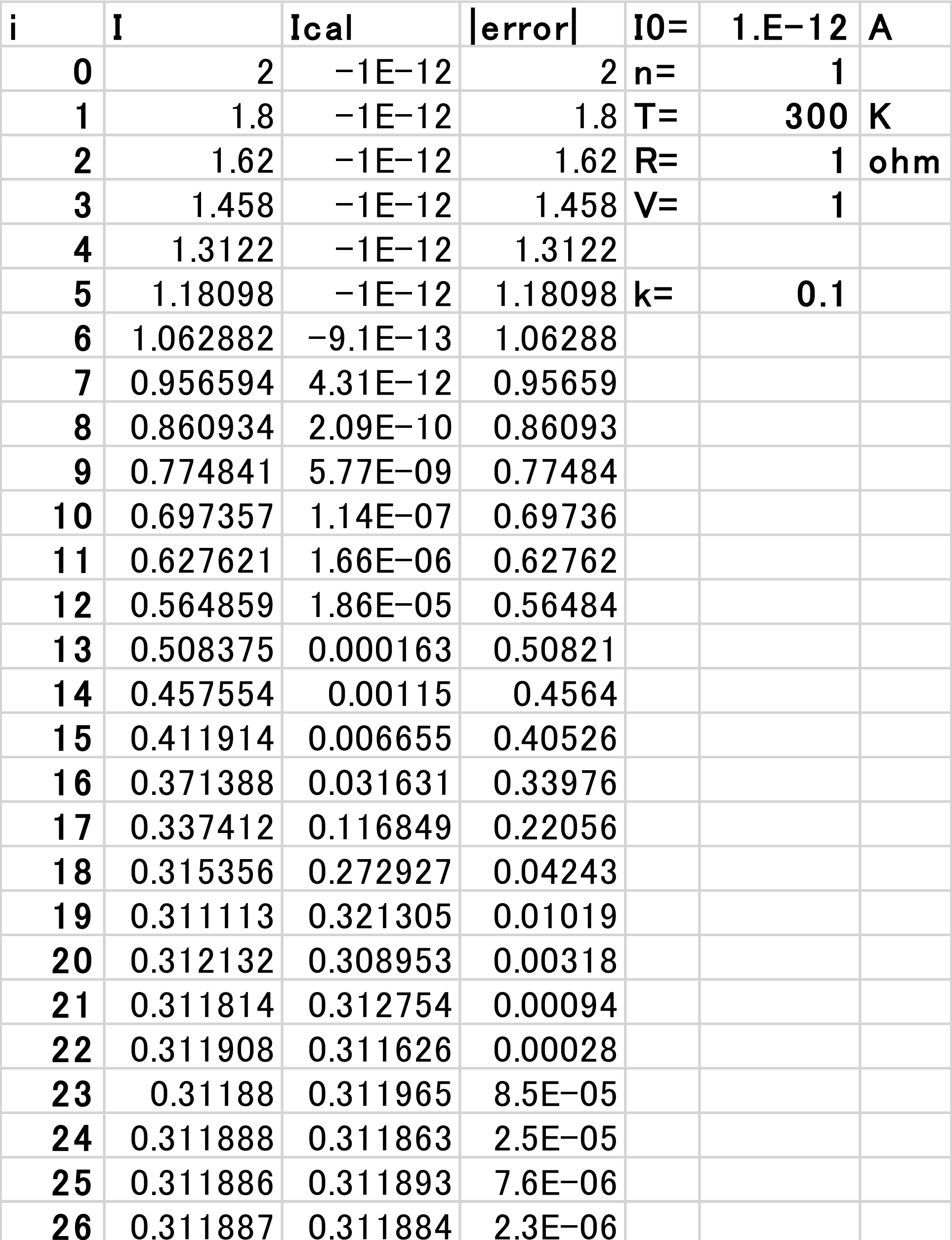
until abs(I_i – I_{i-1}) < EPS is achieved

- 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

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

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

- Electron density ρ_{fin} is calculated the solved wave functions, but ρ_{fin} would be different from ρ_{ini}

ρ_{ini} must be equal to ρ_{fin}, otherwise these loss physical meaning

- More appropriate ρ_{new} is guessed from ρ_{fin} and ρ_{ini}, and repete the above calculations

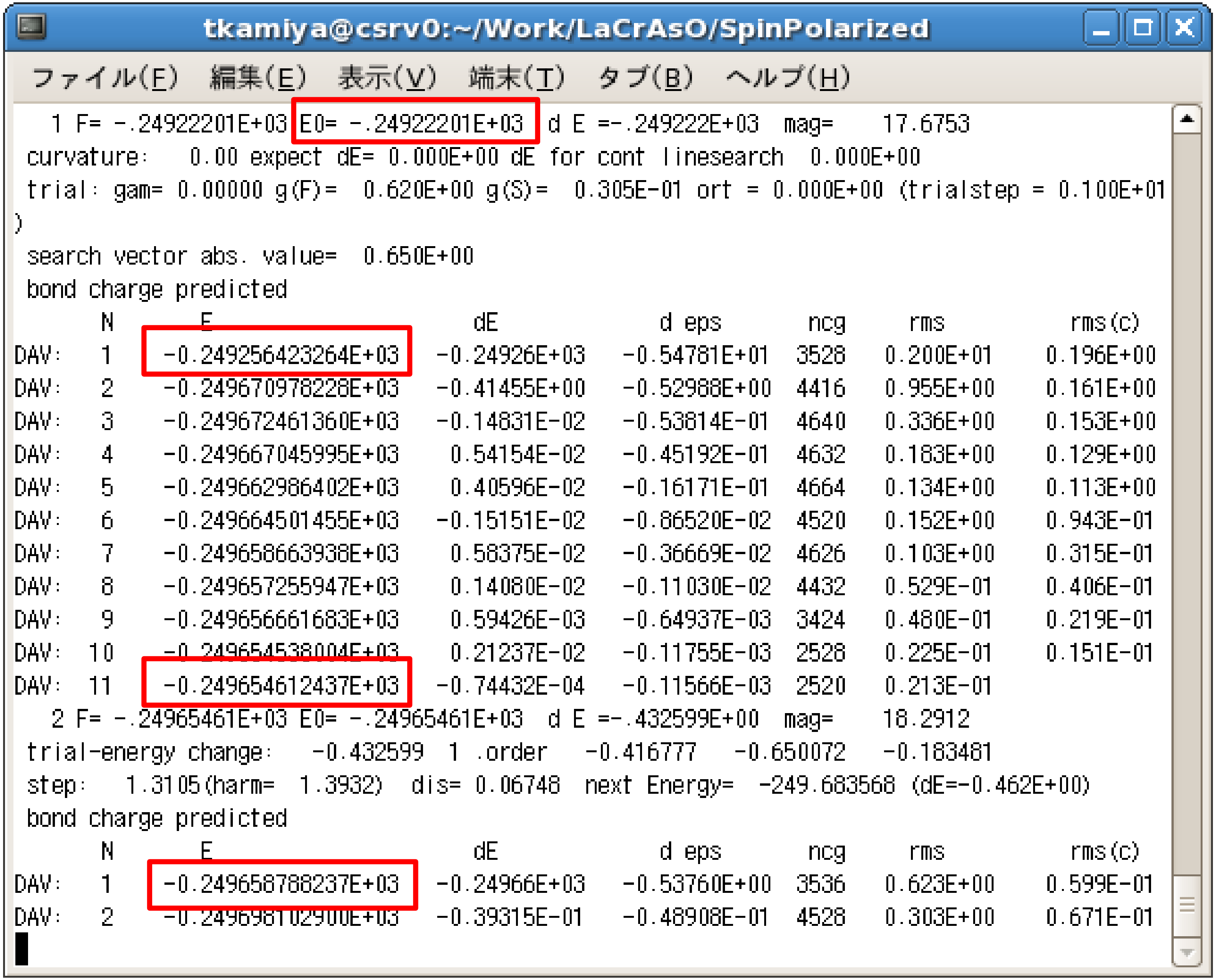
  ex.： $\rho_{\mathrm{new}} = \rho_{\mathrm{ini}} + k_{\mathrm{mix}}(\rho_{\mathrm{fin}} + \rho_{\mathrm{ini}})$

  k_{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サイクル
Repeat until $\rho_{\mathrm{fin}} = \rho_{\mathrm{ini}}$

Example: SCF/structure relaxation by VASP



```
   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_{mix}

Initial x_0

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

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)

Transcendental equation

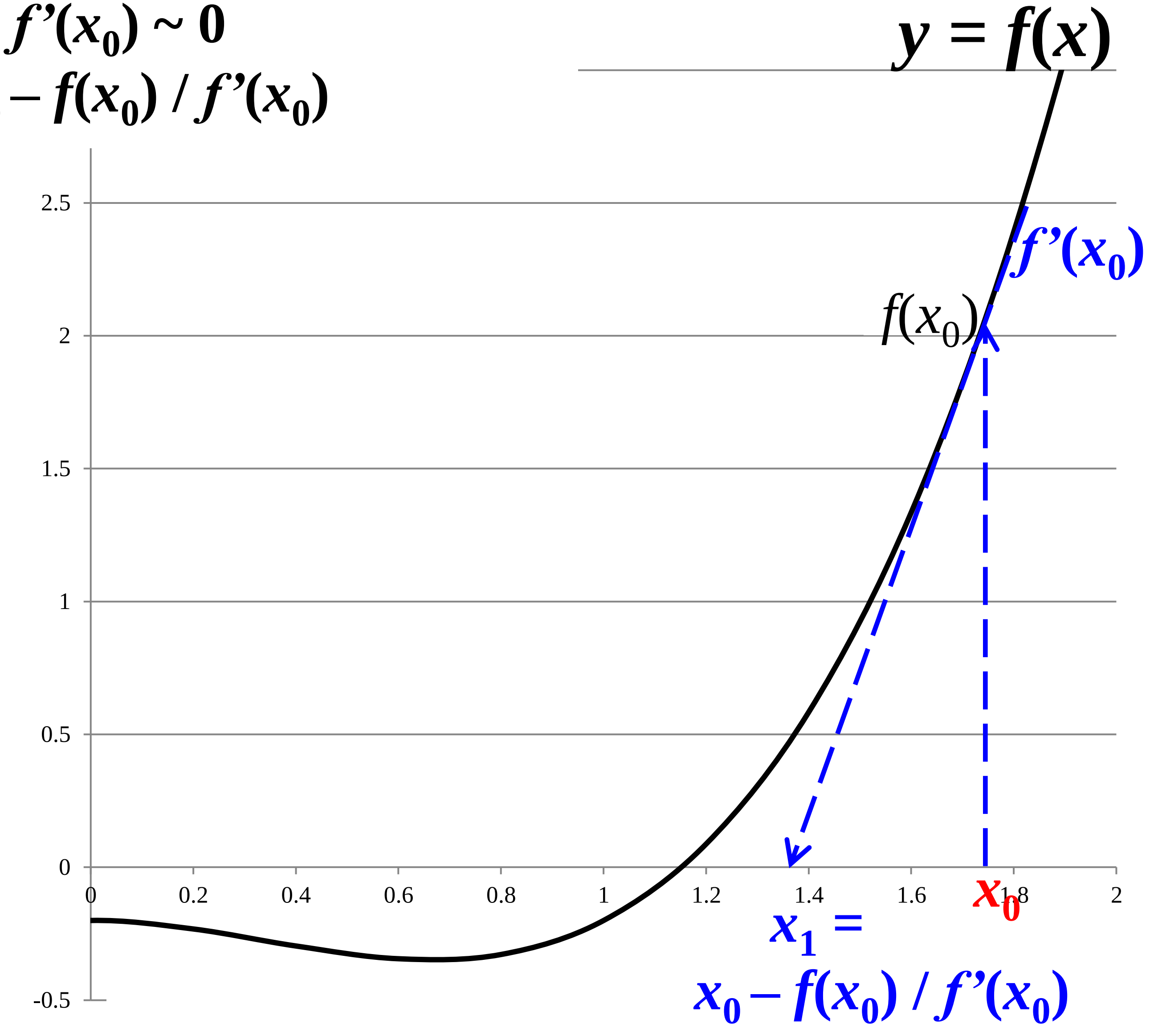
超越方程式の解法

Newton-Raphson method

Solve $f(x) = 0$

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

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



Newton-Raphson method

Solve $f(x) = 0$

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

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

$f'(x_0)$ can be substituted with finite difference

$$f'(x_0) = (f(x_0+\mathrm{h}) - f(x_0)) / h$$

Secant method (割線法, はさみうち法)**:**

$$f'(x_n) = (f(x_n) - f(x_{n-1})) / (x_n - x_{n-1})$$

Save the number of

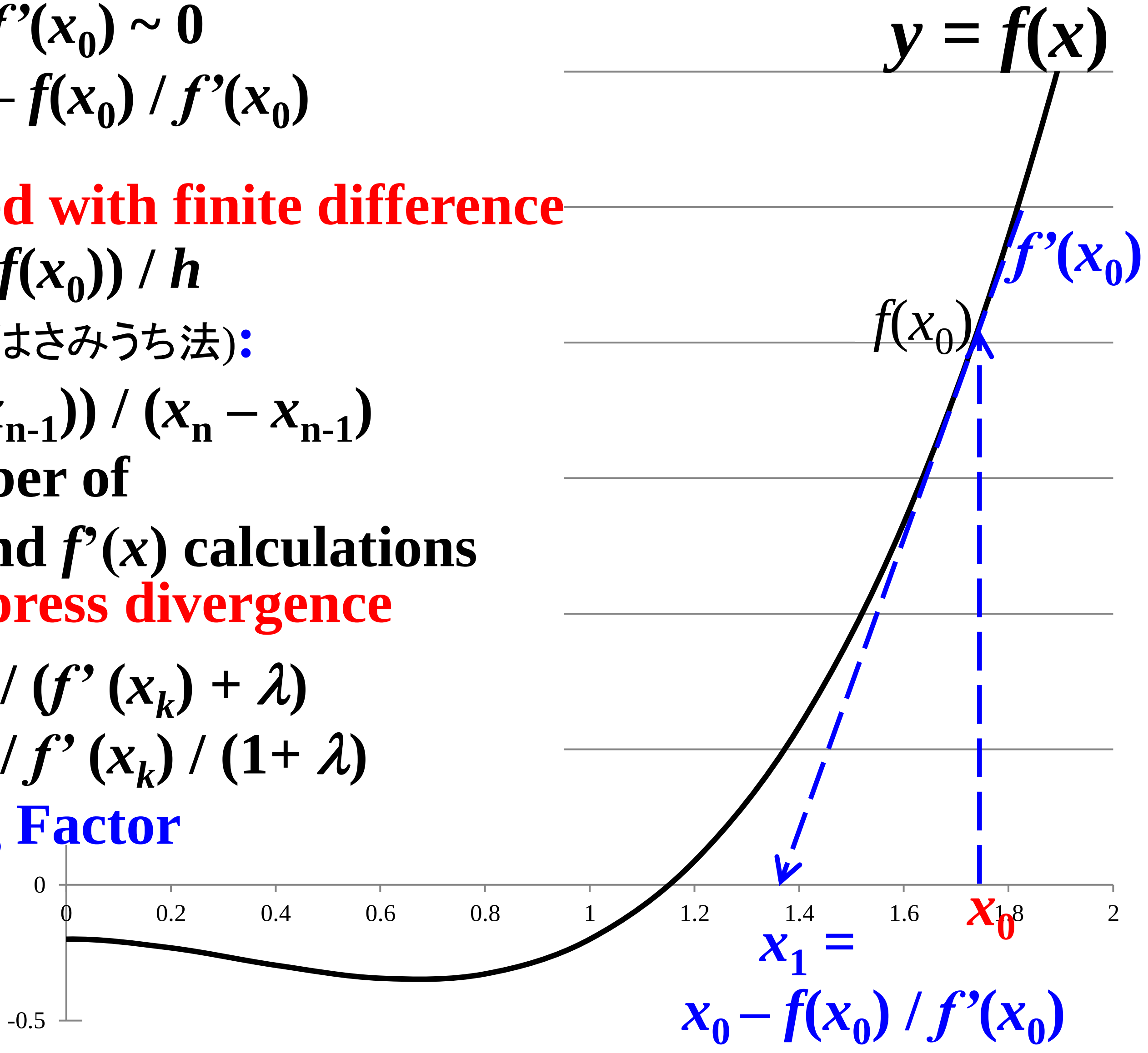
$f(x)$ and $f'(x)$ calculations

Variation to suppress divergence

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

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

λ: Dumping Factor



Program: equation-newton-Raphson.py

Usage: python equation-newton-raphson.py x0 dump t_{sleep}

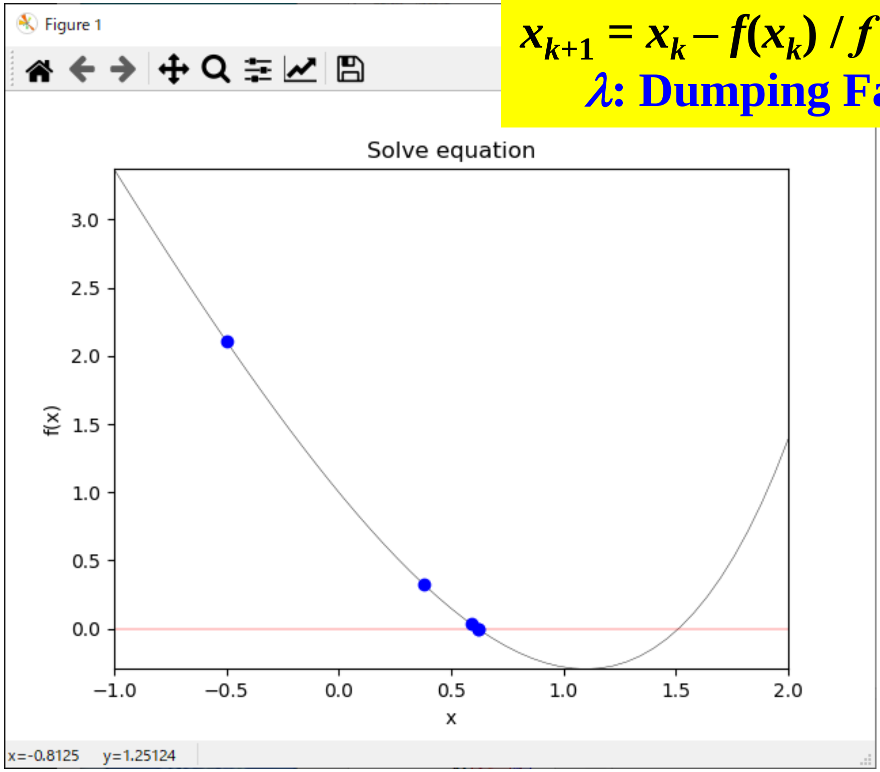
$f(x) = \exp(x) - 3.0x$

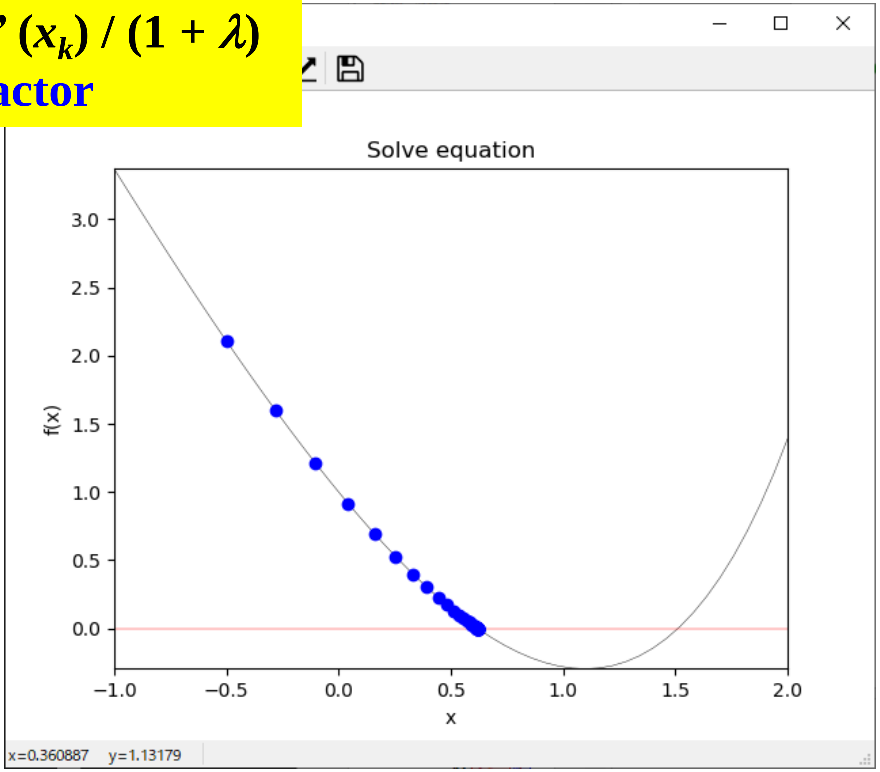
python equation-newton-raphson.py -0.5 0

python equation-newton-raphson.py -0.5 3

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

λ: Dumping Factor





Effect of dumping factor (収束過程の比較)

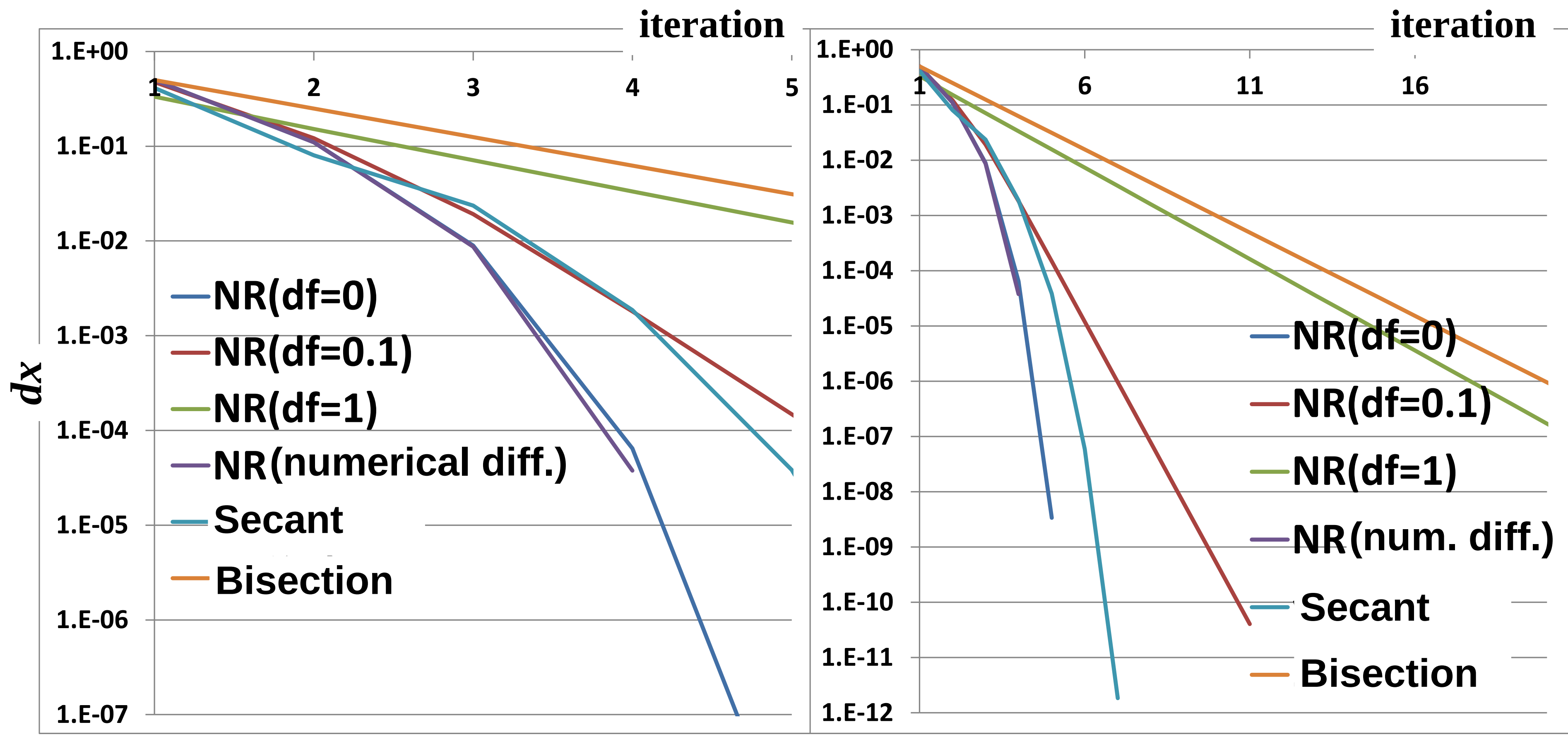
$f(x) = \exp(x) – 3x = 0$ (initial $x = 0$)   Exact 0.619061

Newton-Raphson (Dumping factor = 0)

Iter.	x	$\lvert x_i – x_{i-1} \rvert$
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

Effect of dumping factor: Convergence process

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



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

λ: Dumping Factor

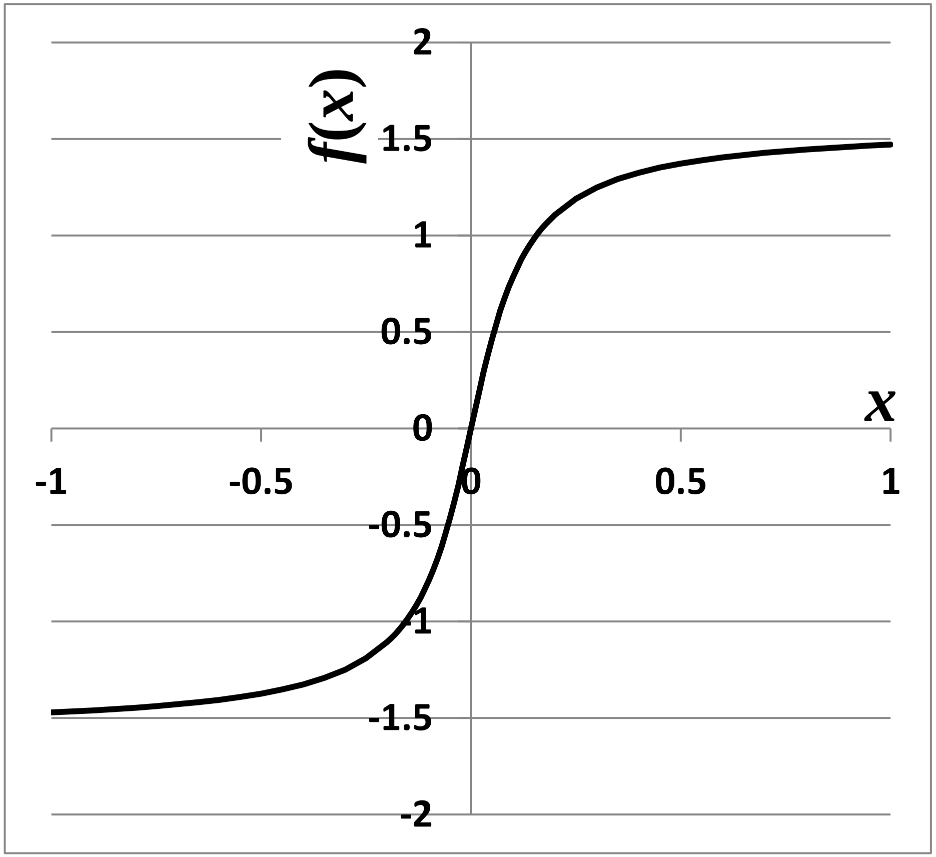
NR: Newton-Raphson method

df: Dumping Factor

Case Newton method fails

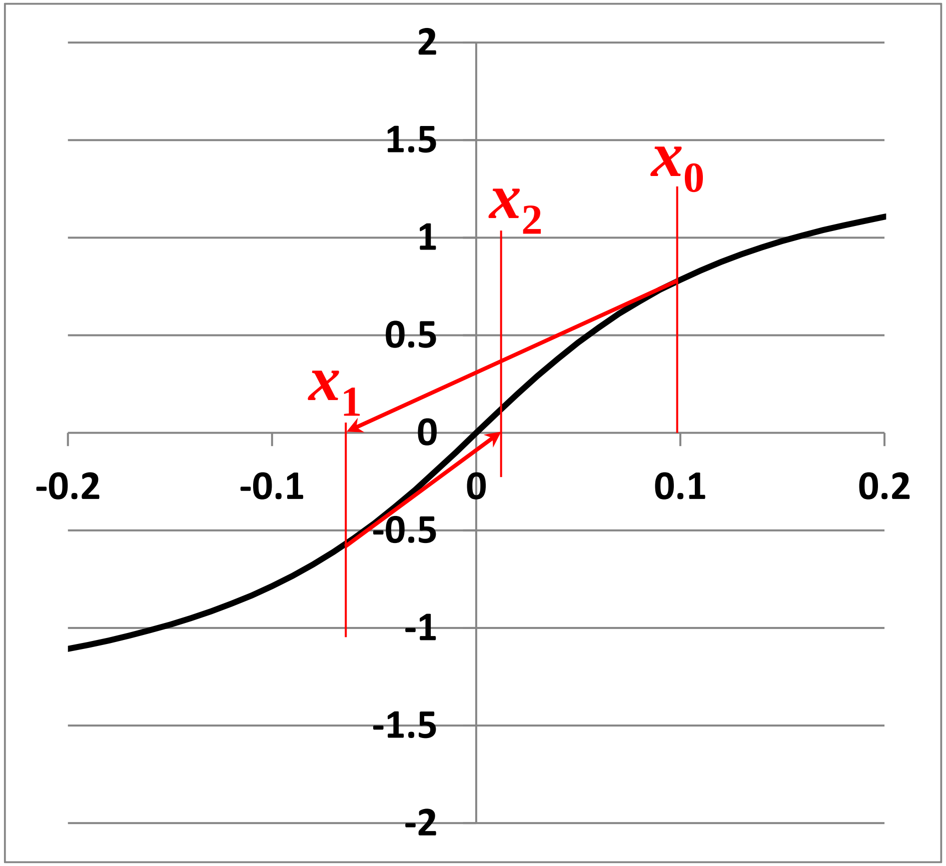
$f(x) = \tan^{-1}(10x)$

initial $x = 0.1$



A case to reach convergence

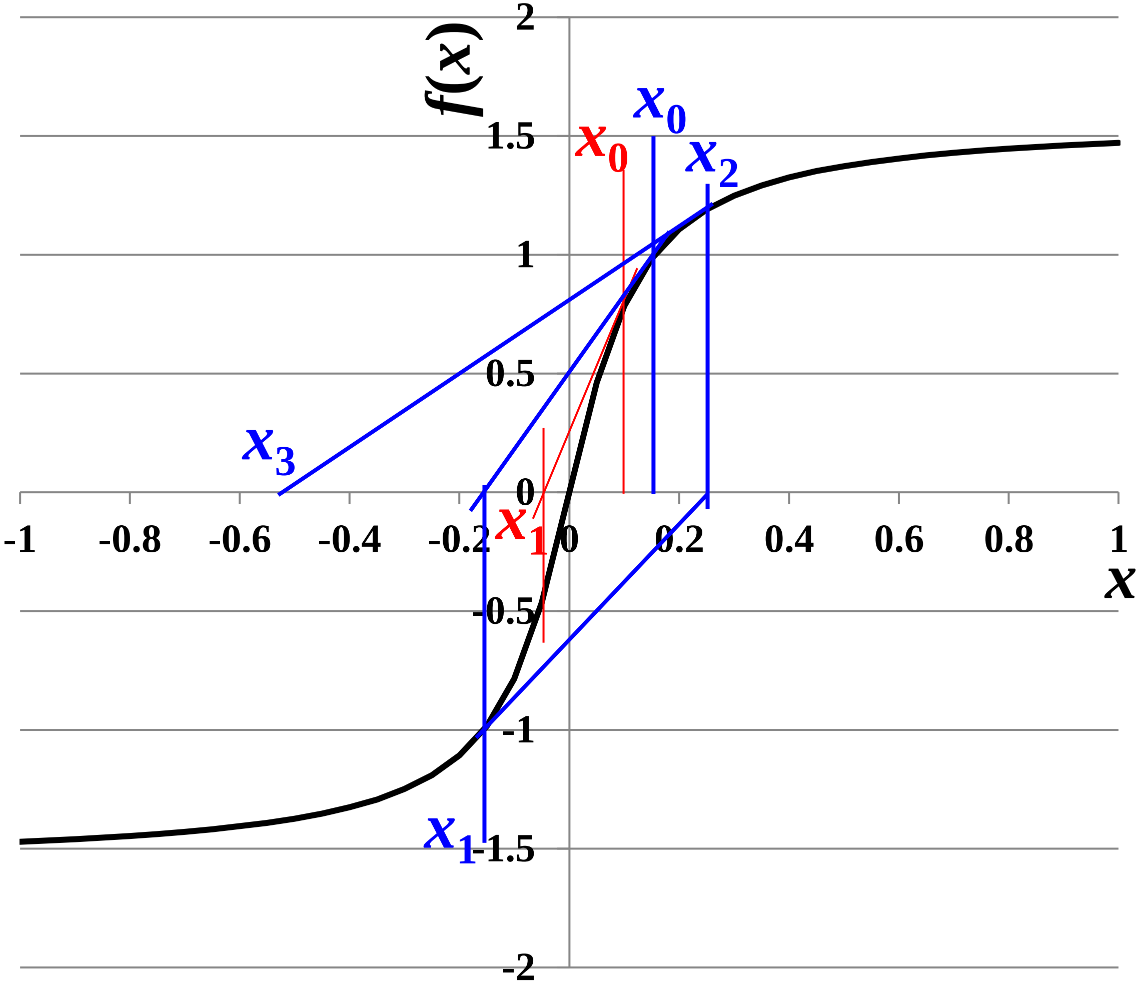
i	x	f(x)	df/dx	dx
0	0.1	0.7854	5	−0.1571
1	−0.05708	−0.5187	7.54257	0.06877
2	0.011686	0.11633	9.86527	−0.0118
3	−0.00011	−0.0011	9.99999	0.00011
4	1.15E−10	1.2E−09	10	−1E−10



Case Newton method fails

$f(x) = \tan^{-1}(10x)$

initial $x = 0.15$



Diverged ($\lambda = 0$)

i	x	f(x)	df/dx	dx
0	0.15	0.98279	3.07692	−0.3194
1	−0.16941	−1.0375	2.58404	0.40152
2	0.232112	1.164	1.56553	−0.7435
3	−0.51141	−1.3777	0.36827	3.74095
4	3.229546	1.53984	0.00958	−160.76
5	−157.529	−1.5702	4E−06	389644
6	389486.7	1.5708	1.1E−12	−1E+12

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

λ: Dumping Factor

Stabilize convergence by choosing λ($\lambda = 1$)

i	x	f(x)	df/dx	dx
0	0.15	0.98279	3.07692	−0.2411
1	−0.09106	−0.7387	5.46675	0.11422
2	0.023161	0.2276	9.49088	−0.0217
3	0.001466	0.01466	9.99785	−0.0013
4	0.000133	0.00133	9.99998	−0.0001
5	1.21E−05	0.00012	10	−1E−05
6	1.1E−06	1.1E−05	10	−1E−06
7	1E−07	1E−06	10	−9E−08
8	9.09E−09	9.1E−08	10	−8E−09
9	8.27E−10	8.3E−09	10	−8E−10

Program: Electron density in metal

Issues for integrating *N*(*e*)*f*(*e*)

- Wide integration range $E = 0 \sim E_F + \alpha k_B T$ – several eV (accuracy at the order of exp(-α))
- Important range for accuracy is the range of $\alpha k_B T \sim 0.1$ eV around E_F
- For numerical integration, E mesh ΔE should be very small around E_F (if $0.01\alpha k_B T$, $\Delta E \sim 1$ meV)

  => Not good to use the same ΔE for the whole integration range $E = 0 \sim E_F + \alpha k_B T$

=> ▪ **Divide integration range** (Analytical integration may be employed for $0 \sim E_F - \alpha k_B T$)

▪ **Better to employ accuracy-guaranteed library for integration**

python integrate.quad() can accept accuracy as epsrel variable

Program: N-integration-metal.py

Ex.: python N-integration-metal.py 300 5.0

At 300 K, $E_F = 5.0$ eV

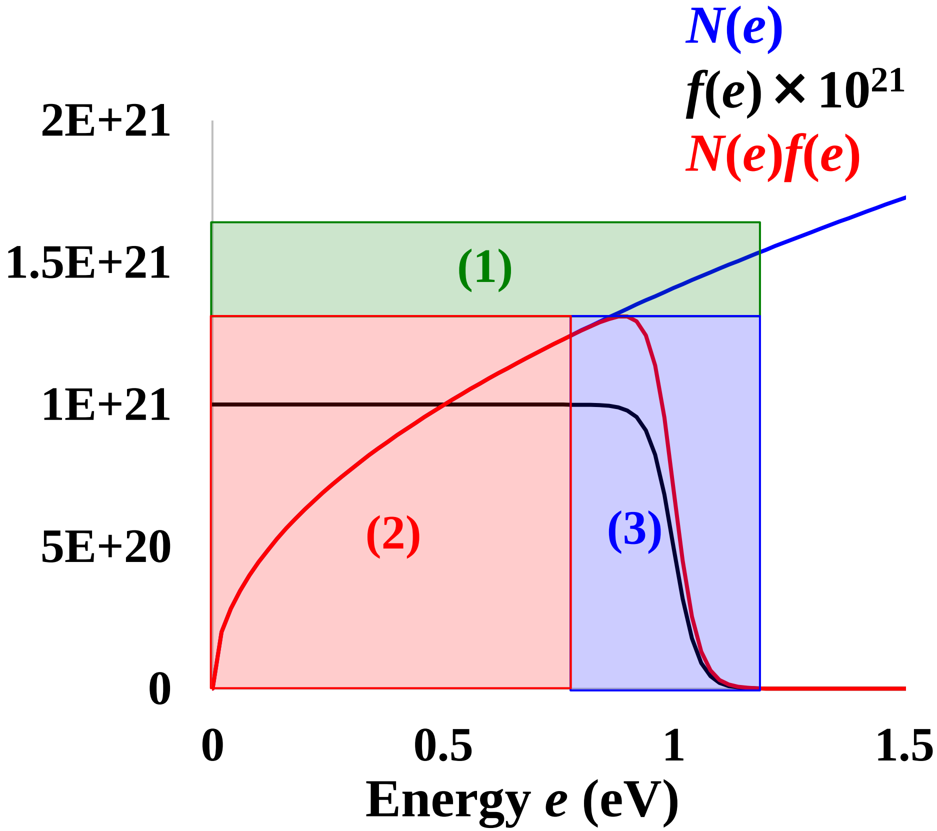
Time is measured for 300 cycles calculation

8 digit accuracy (epsrel = 1e-8), α = 6:

range	time (300 cycles)
(1) $0 \sim E_F + \alpha k_B T$	**0.109 s**
(2) $0 \sim E_F - \alpha k_B T$	**0.063 s**
(3) $E_F - \alpha k_B T \sim E_F + \alpha k_B T$	**0.016 s**

(2) + (3) is faster by ~30 % than (1).

Employing analytical integration for (2) is faster by a factor of 10



Program: T dependence of E_F for metal

$E_F(T)$ is determined by $N_e = \int N(e)f(e, E_F)de$ for the given electron number N_e

$N(e)f(e, E_F)$ is integrated in the range $E = 0 – \infty$ (acutualy up to $E_F + \alpha k_B T$)

The initial value of $E_F(T)$ can be taken as the analytical form of $E_F(0)$ at 0 K.

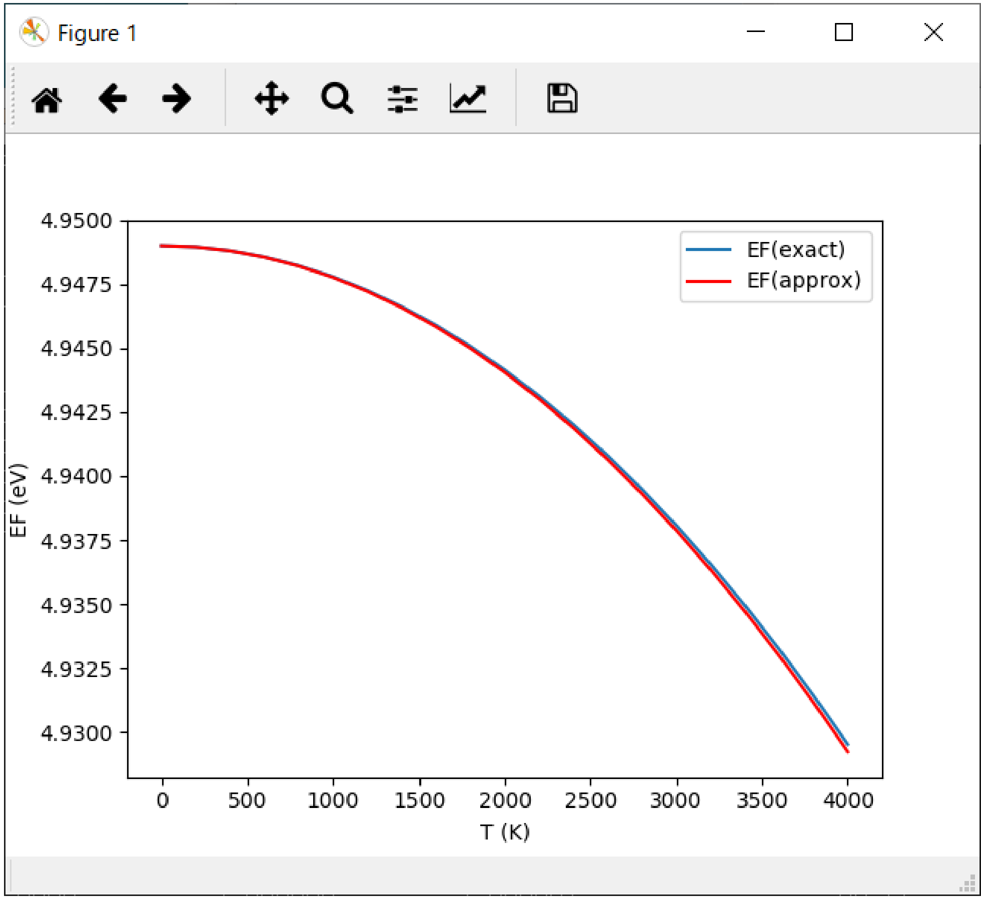
Since the variation of $E_F(T)$ is small, the Newton method stability converges.

Compare with the approx. form $E_F(T) = E_F(0) - \frac{\pi^2}{6}(k_B T)^2 N'(E_F(0))/N(E_F(0))$

Program: EF-T-metal.py

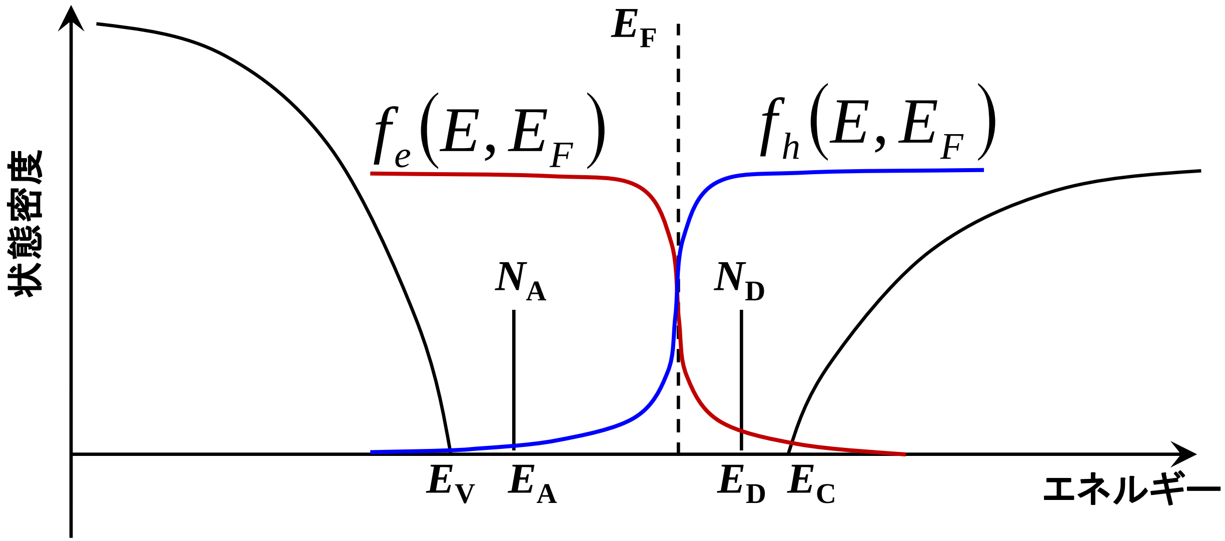
Ex.: python EF-T-metal.py

T (K)	E_F (Newton, eV)	E_F (approx., eV)
0	4.948988	4.948988
600	4.948554	4.948544
1200	4.947248	4.947211
1800	4.945069	4.944990
2400	4.942013	4.941880
3000	4.938075	4.937882
3600	4.933247	4.932994
4000	4.929529	4.929243



Density of states, n_e, and n_h in semiconductor

Total density of states: $D(E) = D_e(E) + D_h(E) + D_D(E) + D_A(E)$



Valence band

$$D_h(E) = D_{V0}\sqrt{E_V - E}$$

$$D_A(E) = N_A\delta(E - E_A)$$

$$f_h(E, E_F) = \frac{1}{\exp(\beta(E_F - E)) + 1}$$

Free hole density

$$n_h = \int_{-\infty}^{E_V} f_h(E, E_F) D_h(E) dE$$

Ionized acceptor density

$$N_A{}^{-} = N_A(1 - f_h(E_A, E_F))$$

Conduction band

$$D_e(E) = D_{C0}\sqrt{E - E_C}$$

$$D_D(E) = N_D\delta(E - E_D)$$

$$f_e(E, E_F) = \frac{1}{\exp(\beta(E - E_F)) + 1}$$

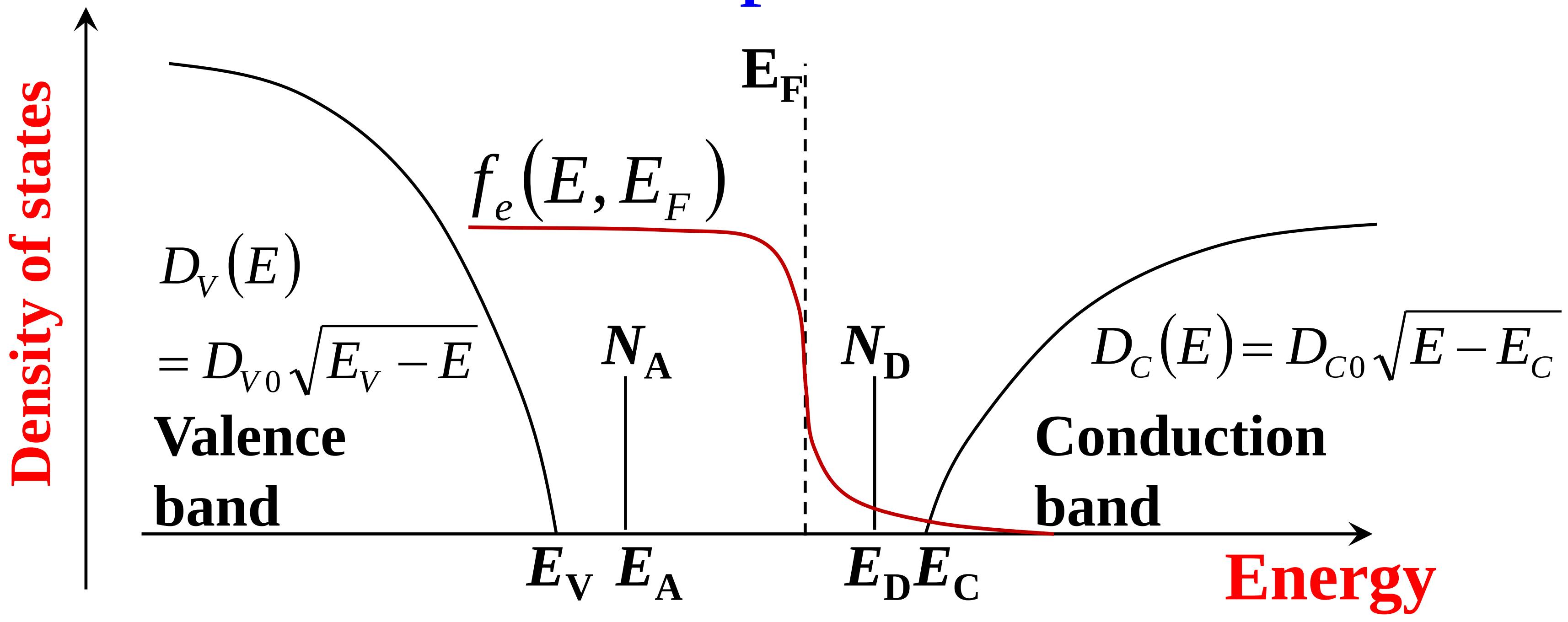
Free electron density

$$n_e = \int_{E_C}^{\infty} f_e(E) D_e(E) dE$$

Ionized donor density

$$N_D{}^{+} = N_D(1 - f_e(E_D, E_F))$$

How to determine E_F for semiconductors



$$E_g = E_C - E_V$$

Charge neutrality condition

$$N_A^- + N_e = N_D^+ + N_h \Longrightarrow E_F$$

$$N_e = \int_{E_C}^{\infty} D_C(E) f_e(E, E_F)\, dE$$

$$N_D^{\;+} = N_D\left[1 - f_e(E_D, E_F)\right]$$

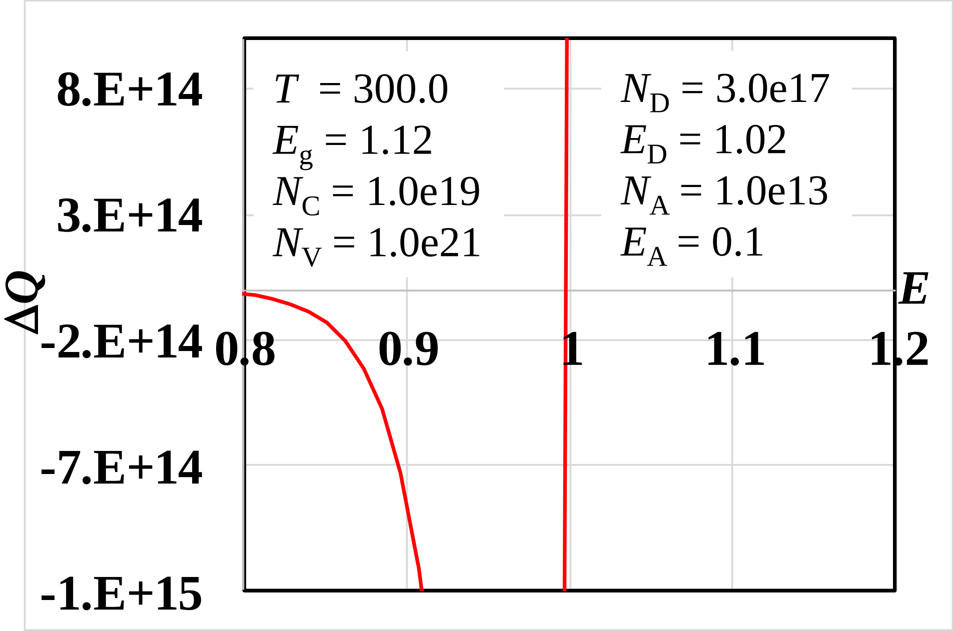
How to calculate E_F: Illustrative solution

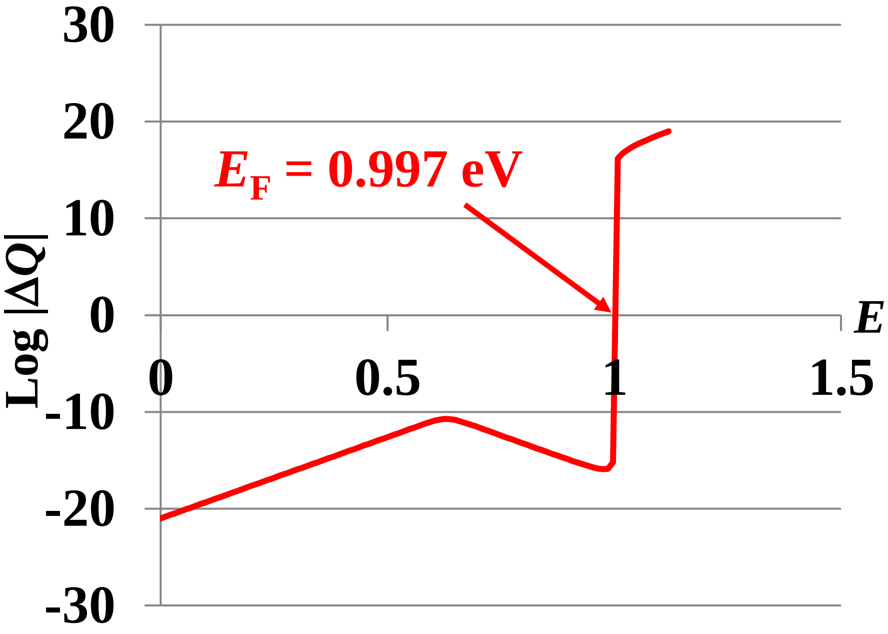
$$N_e = \int_{E_C}^{\infty} D_C(E) f_e(E,E_F)\,dE \qquad N_h = \int_{E_C}^{\infty} D_V(E) f_h(E,E_F)\,dE$$

$$N_D^{\,+} = N_D\left[1 - f_e(E_D,E_F)\right] \qquad N_A^{\,-} = N_A\left[1 - f_h(E_A,E_F)\right]$$

$$f_h(E,E_F) = 1 - f_e(E,E_F)$$

Plot $\Delta Q = (N_A^- + N_e) - (N_D^+ + N_h)$ w.r.t. E_F and find $\Delta Q = 0$





E_F = 0.997 eV

Bisection method (二分法): Continuous func（連続関数)

Solution of $f(x) = 0$ for (monotonic) continuous function $f(x)$

1. **Start from a range $[x_0, x_1]$ where $f(x_0) < 0$ & $f(x_1) > 0$**

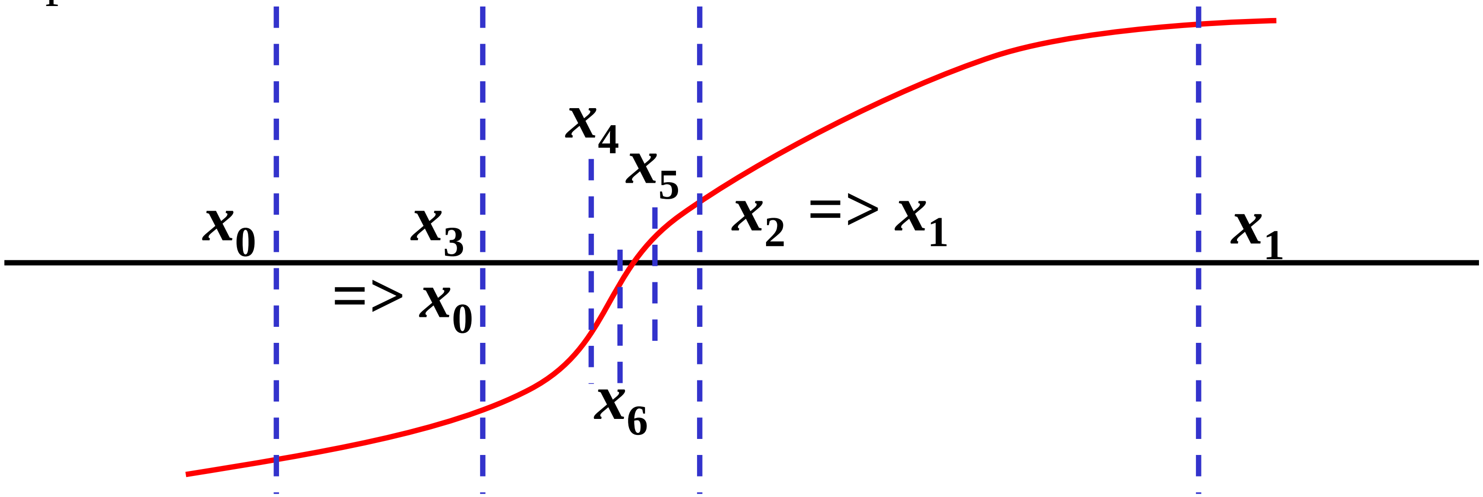
   (or $f(x_0) > 0$ & $f(x_1) < 0$)

   *** Solution exist in this range for a monotonic function**

2. **Solve the equation by the following iterative procedure**

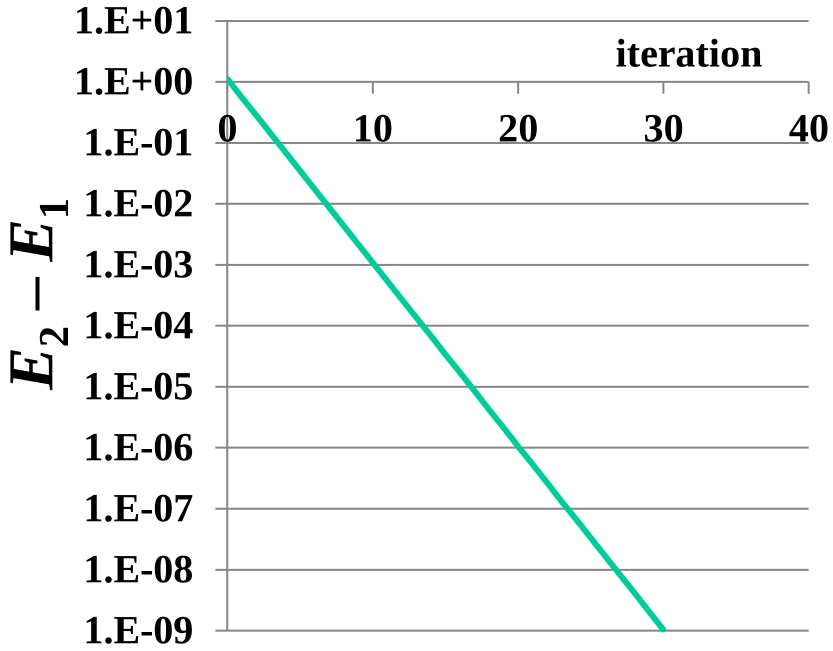
Case $f(x_0) < 0$ and $f(x_1) > 0$: Judge by $f(x_0) \cdot f(x_1) < 0$

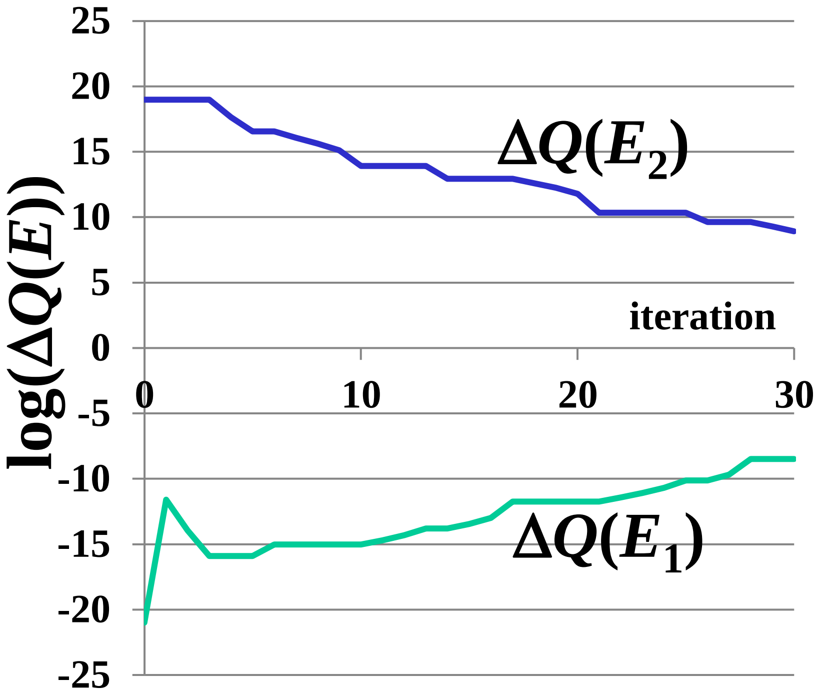
1. $x_2 = (x_0 + x_1) / 2.0$
2. If $f(x_2) > 0$ ($f(x_0) \cdot f(x_2) < 0$), x_1 is replaced with x_2
   If $f(x_2) < 0$ ($f(x_1) \cdot f(x_2) < 0$), x_0 is replaced with x_2
3. Solution x_2 is obtained when $|x_1 - x_0|$, $|f(x_1) - f(x_0)|$ becomes less than EPS.
4. Repet 1 – 3



E_F by bisection method: Convergence procedure

Initial range: $[E_1, E_2] = [E_V = 0, E_C = E_g]$
Find $\Delta Q = (N_A^- + N_e) - (N_D^+ + N_h) = 0$





After 30 times iterations

$E_F = [0.9985173589, 0.9985173599]$

$dQ = [-3\times10^8, 8\times10^8]$

Program: EF-T-semiconductor.py

Program: EF-T-semiconductor.py

Usage: python EF-T-semiconductor.py EA NA ED ND Ec Nv Nc

Ex.: python EF-T-semiconductor.py 0.05 1.0e15 0.95 1.0e16 1.0 1.2e19 2.1e18

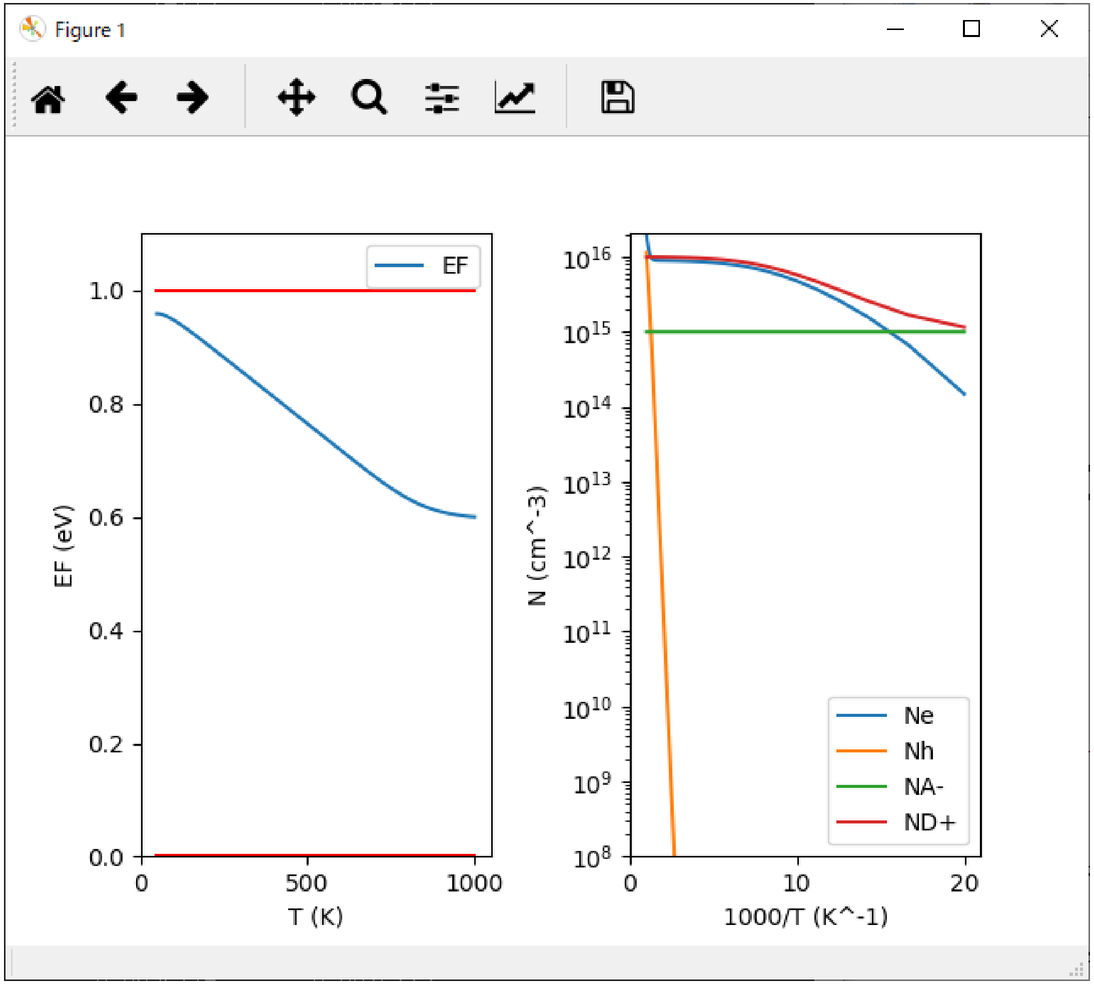
$E_c = 0$, $E_c = 1.0$ eV (= band gap)

$E_A = 0.05$ eV, $N_A = 10^{15}$ cm^{-3},

$E_D = 0.95$ eV, $N_D = 10^{16}$ cm^{-3}

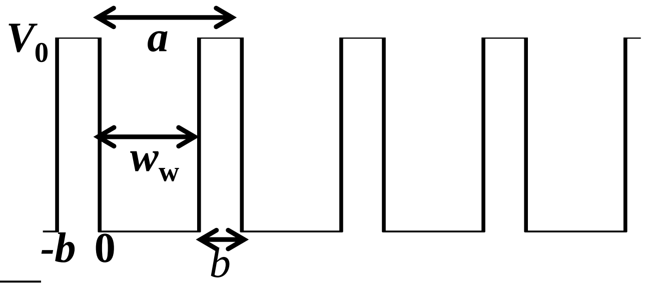
$N_c = 1.2\mathrm{x}10^{19}$ cm^{-3}

$N_v = 2.1\mathrm{x}10^{18}$ cm^{-3}



Multi-values equation: Kronig-Penney model

Solution of $\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}+V(x)\right)\phi = E\phi$



$\boldsymbol{\phi_k(x)}$ **= exp(*ikx*)*u*(*x*), *u*(*x*+*a*) = *u*(*x*)**

In well: $\phi(x) = A\exp(i\alpha x) + B\exp(-i\alpha x)$ $\quad\alpha = \sqrt{2mE}/\hbar$

In barrier: $\phi(x) = C\exp(\beta x) + D\exp(-\beta x)$ $\quad\beta = \sqrt{2m(V_0 - E)}/\hbar$

Boundary condition: $\boldsymbol{\phi_k(x)}$ **and** $\boldsymbol{\phi_k'(x)}$ **are continuous at *x* = 0 and -*b***

Bloch's theorem : $\boldsymbol{\phi_k(x+a) = \lambda\phi_k(x),\ \lambda = \exp(ika)}$

$$\begin{pmatrix} 1 & 1 & -1 & -1 \\ i\alpha & -i\alpha & -\beta & \beta \\ \exp(i\alpha w_w) & \exp(-i\alpha w_w) & -\lambda\exp(-\beta b) & -\lambda\exp(\beta b) \\ i\alpha\exp(i\alpha w_w) & -i\alpha\exp(-i\alpha w_w) & -\beta\lambda\exp(-\beta b) & \beta\lambda\exp(\beta b) \end{pmatrix}\begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The determinant of the left matrix must be 0:

$$\cos ka = \left(\frac{\beta(E)^2 - \alpha(E)^2}{2\alpha(E)\beta(E)}\sin\alpha(E)w_w \sinh\beta(E)b + \cos\alpha(E)w_w \cosh\beta(E)b\right)$$

Scan *E* in possible range to find all the solutions,
then use them for initial values
to obtain accurate values by Newton-Raphson method

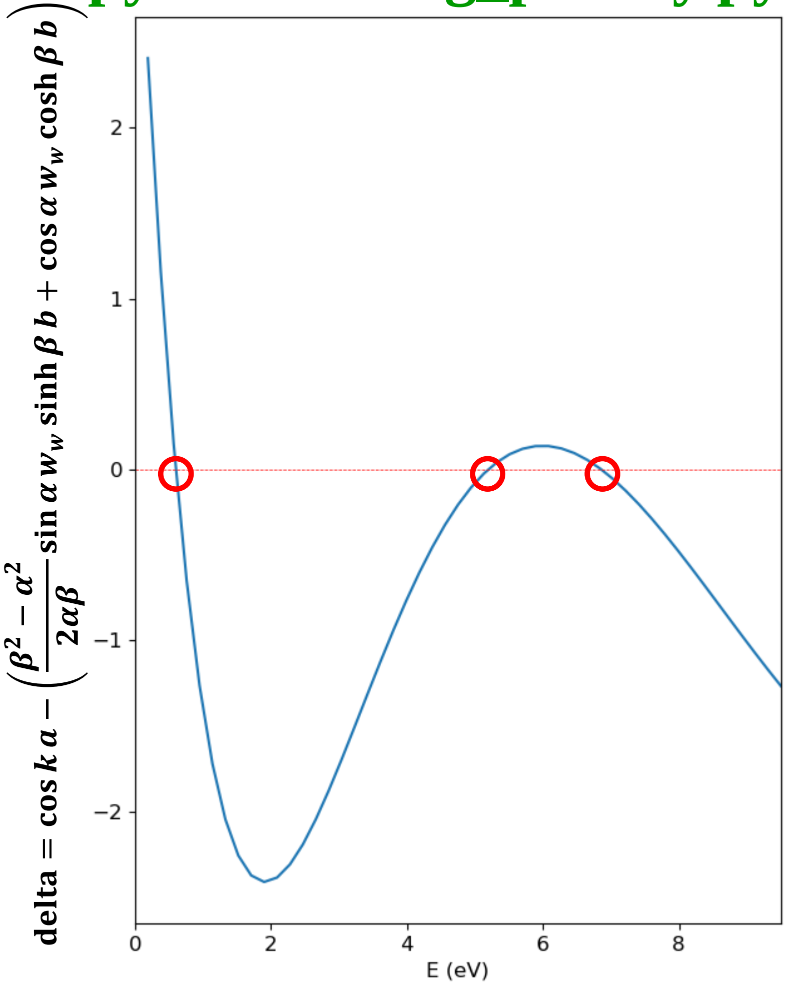
Program: Kronig-Penney model

Program: kronig_penney.py

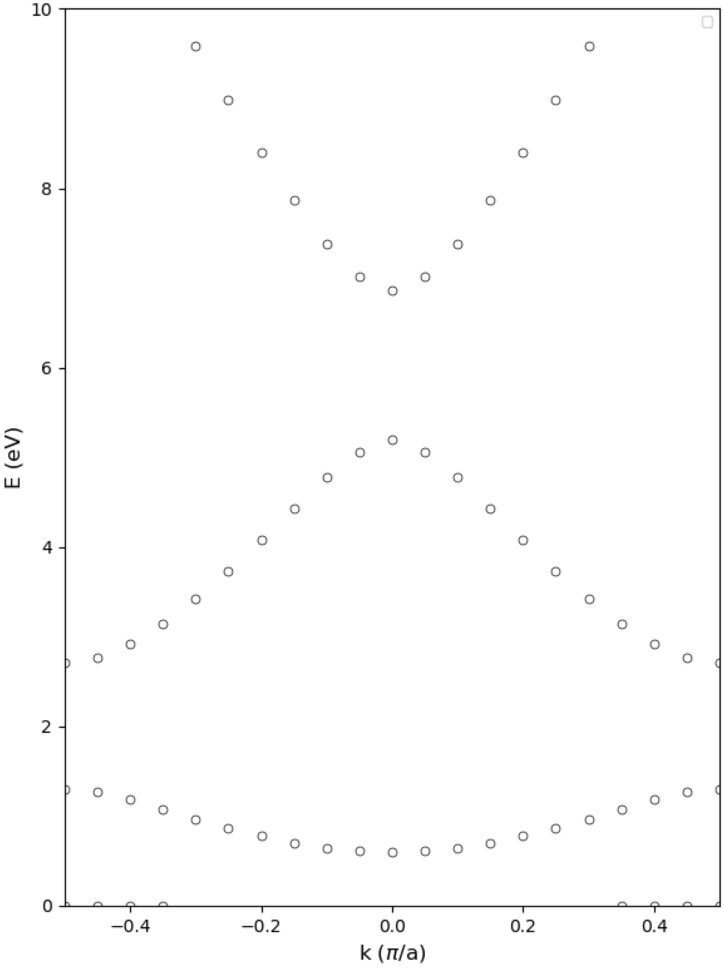
Lattice parameter (Si) a = 5.4064 Å Effective mass m* = 1.0m_e

Barrier width 0.5 Å Barrier height 10.0 eV

python kronig_penney.py



python kronig_penney.py band



Non-linear (NL) optimization

非線形最適化

Optimization

Objective: Find parameters x_i to minimize or maximize a objective function $F(x_i)$

Maximization problem for $F(x_i)$
is equivalent to minimization problem for $–F(x_i)$

Examples:

- **Linear least-squares method:** A linear minimization problem for L2 norm of errors
- **Curve fitting:** A non-linear minimization problem for L2 norm of errors
- **Structure relaxation:** A non-linear minimization for total energy

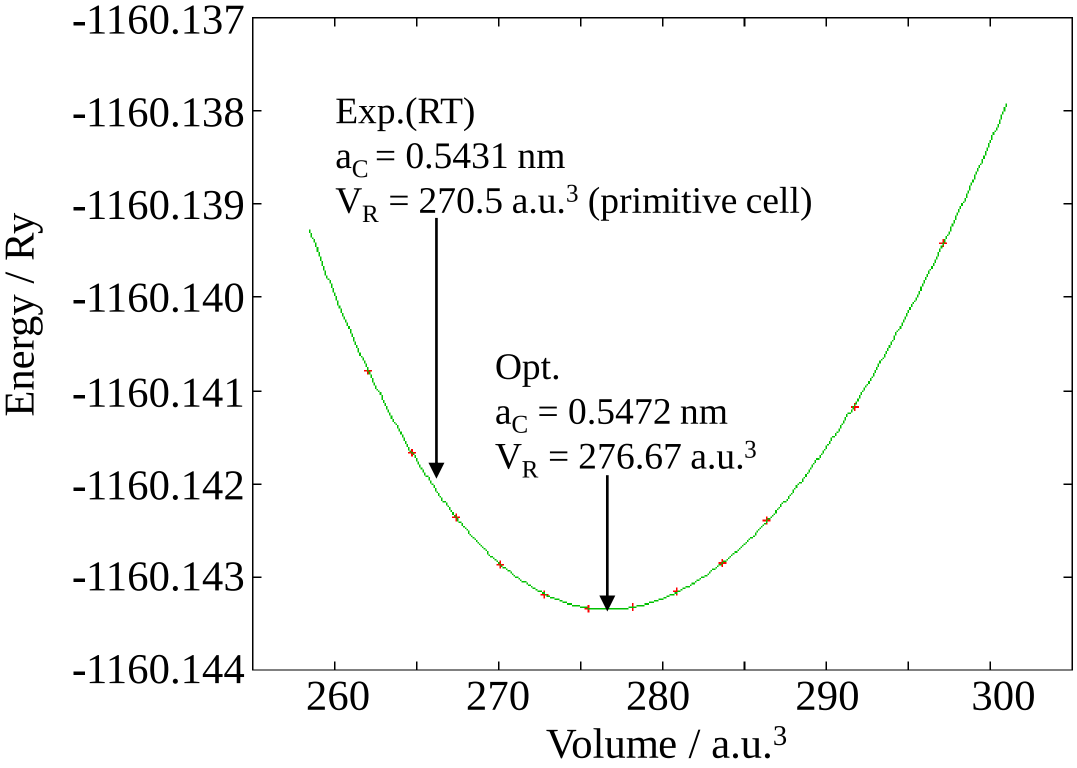
Focus on minimization problem

NL optimization of crystal structure: Illustrative approach

安定構造: 図解による解法

Calculate total energy by quantum calculations by varying a lattice parameter

***ex*. Si**



$$E = E_{\min} + 1/2 B_0 (V/V_0)^2$$

B_0 (GPa) = 87.57 GPa (exp: 97.88 GPa)

Profile models used for spectroscopy

Lorentz function

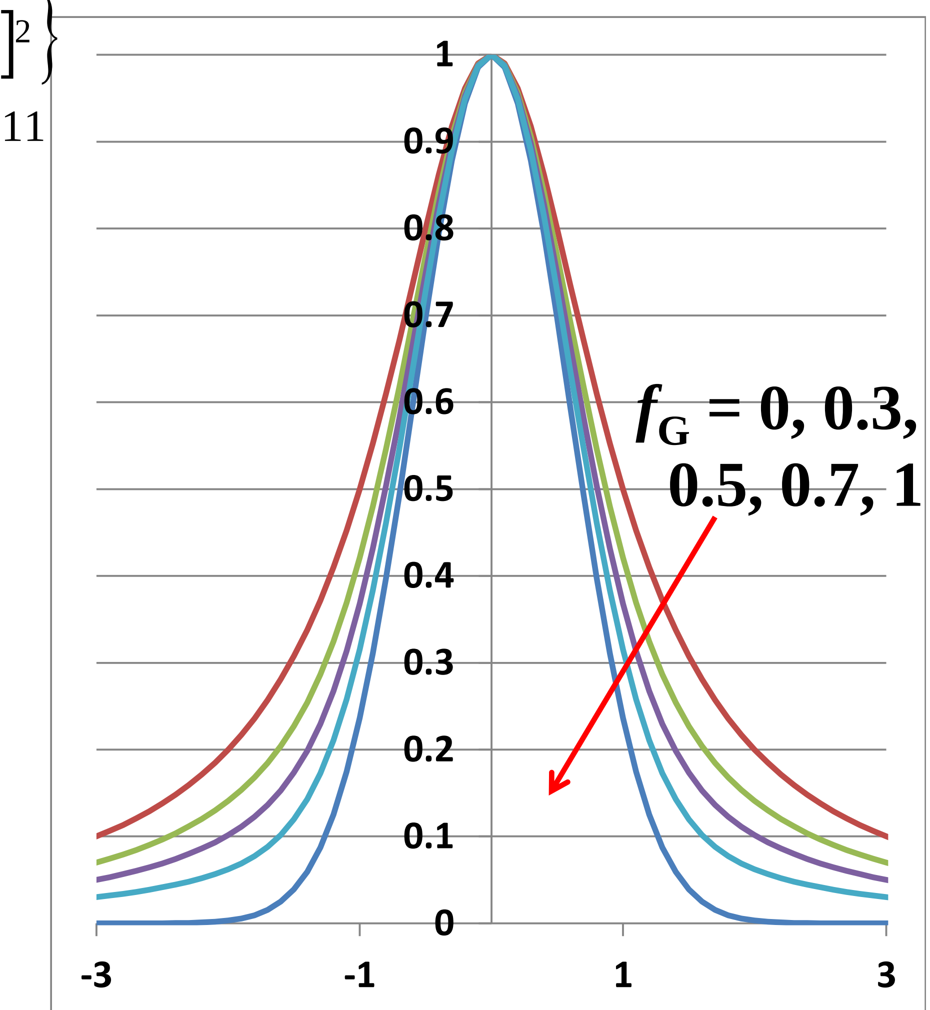
$$I_L(x) = \frac{1}{1+\left[\left(x-x_0\right)/w\right]^2}$$

***w*: half width at half maximum**

Gauss function

$$I_G(x) = \frac{1}{a_w w \pi^{1/2}} \exp\left\{-\left[\left(x-x_0\right)/(a_w w)\right]^2\right\}$$

$$a_w = (\ln 2)^{-1/2} = 0.832554611$$



f_G = 0, 0.3, 0.5, 0.7, 1

Voigt function:

E.g., observed is convolution of sample spectrum $I_L(x)$ and apparatus function $I_G(x)$

$$I_V(x) = \int_{-\infty}^{\infty} I_G(x') I_L(x-x') dx'$$

$$= \frac{a_V}{\pi} \int_{-\infty}^{\infty} \frac{\exp(-x'^2)}{a_V{}^2 + (x-x')^2} dx'$$

Pseudo-Voigt function:

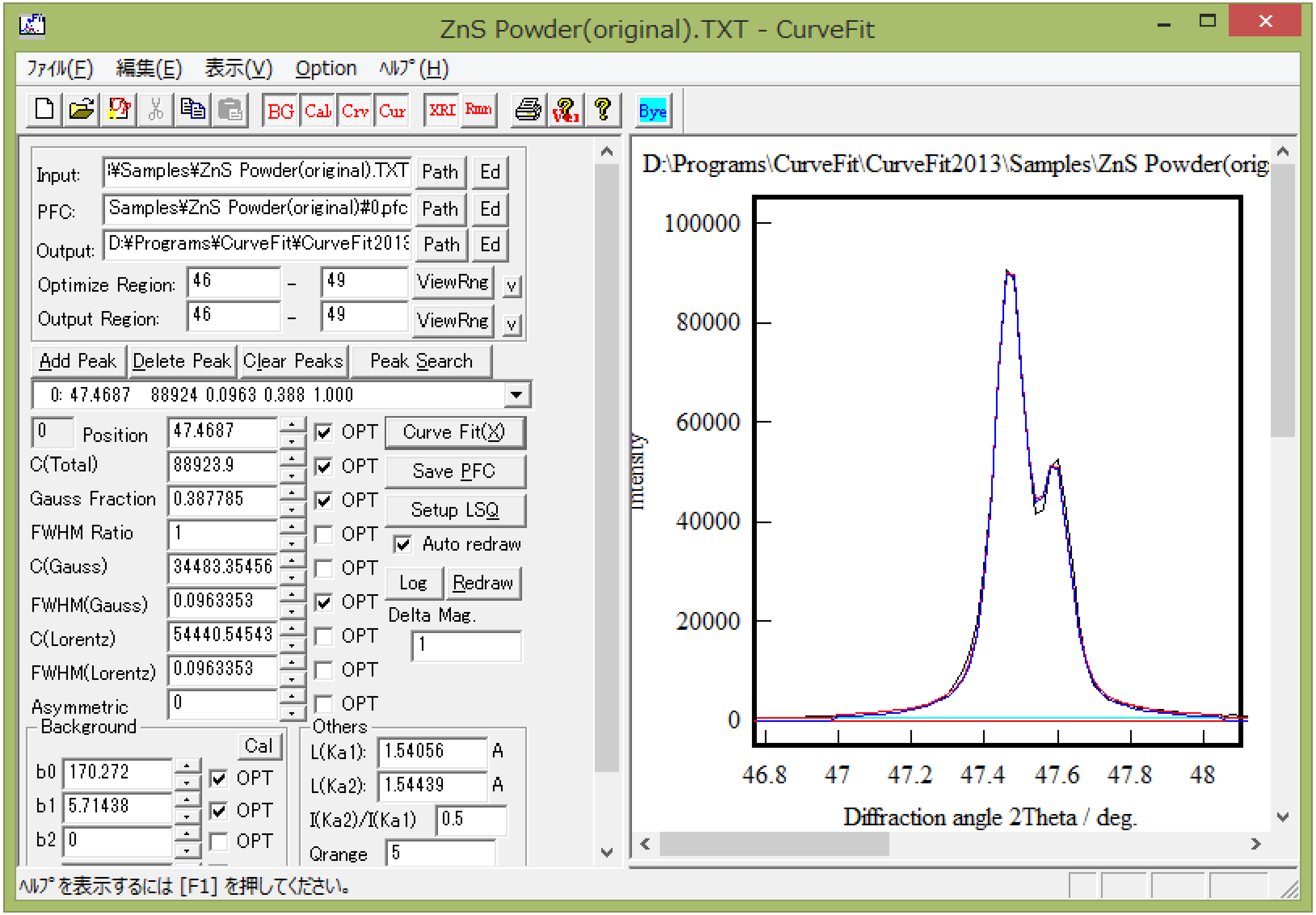
Simplified Voigt function

$$I_{PV}(x) = f_G I_G(x) + (1-f_G) I_L(x)$$

f_G: Gauss fraction

Ex.: Deconvolution of powder XRD peak

Incorporate the intensity ratio from Kα_1 and Kα_2 at 2:1



ZnS Powder(original).TXT - CurveFit

Input: ¥Samples¥ZnS Powder(original).TXT

PFC: Samples¥ZnS Powder(original)#0.pfc

Output: D:¥Programs¥CurveFit¥CurveFit201

Optimize Region: 46 – 49

Output Region: 46 – 49

0: 47.4687 88924 0.0963 0.388 1.000

0 Position 47.4687 ☑ OPT

C(Total) 88923.9 ☑ OPT

Gauss Fraction 0.387785 ☑ OPT

FWHM Ratio 1 □ OPT

C(Gauss) 34483.35456 □ OPT

FWHM(Gauss) 0.0963353 ☑ OPT

C(Lorentz) 54440.54543 □ OPT

FWHM(Lorentz) 0.0963353 □ OPT

Asymmetric 0 □ OPT

☑ Auto redraw

Delta Mag. 1

Background

b0 170.272 ☑ OPT

b1 5.71438 ☑ OPT

b2 0 □ OPT

Others

L(Ka1): 1.54056 A

L(Ka2): 1.54439 A

I(Ka2)/I(Ka1) 0.5

Qrange 5

D:\Programs\CurveFit\CurveFit2013\Samples\ZnS Powder(orig

ﾍﾙﾌﾟ を表示するには [F1] を押してください。

Methods of non-linear (NL) optimization

To find a minimum (maximum) of target function *F*(*x*):

Direct search method (直接探索法)

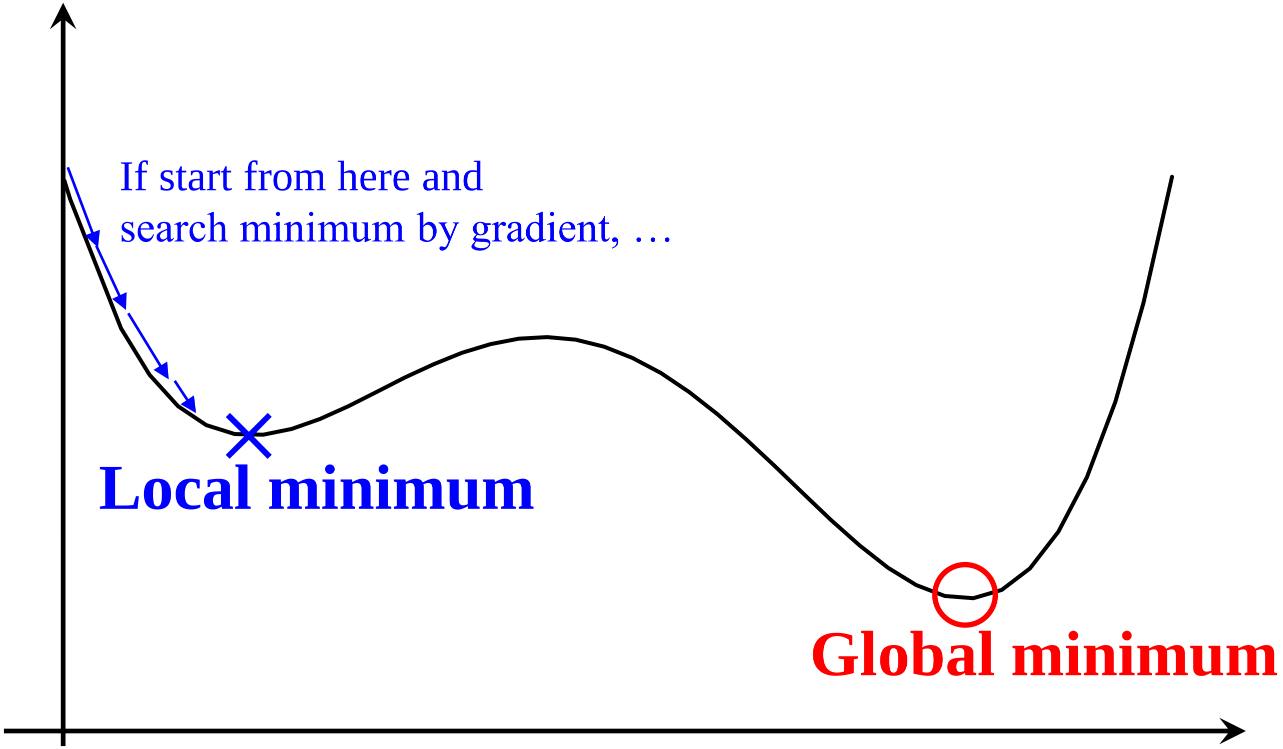
Trial and errors to find a minimum,

but following a certain defined procedures

Gradient method (勾配法)**:**

Use first differential to find the direction of minimum

Global minimum (大域的最小值) vs local minimum (極小值)



How to avoid to be trapped by local minimum:

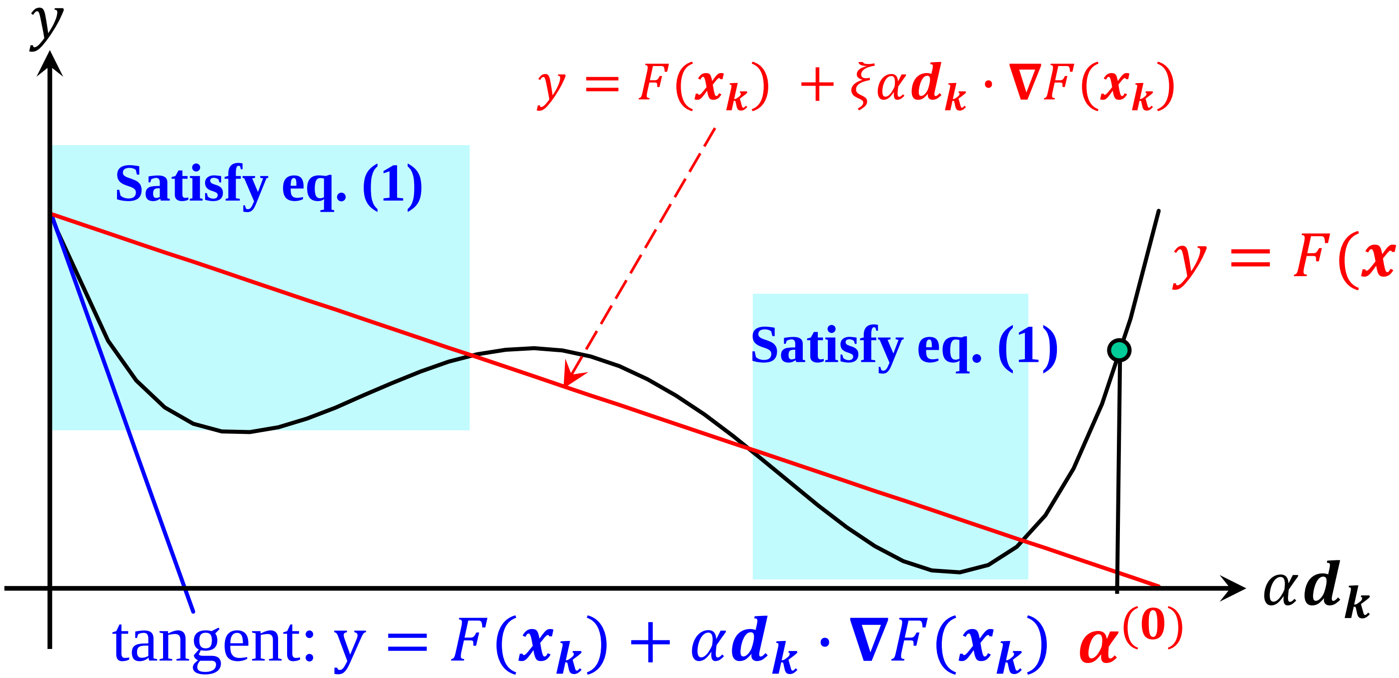
1. **Employ a large initial search range**
2. **Not use a direct value of gradient**

Line search (直線探索法): Armijo condition

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Armijo (アルミホ) condition (eq. (1)) and algorism:

1. **Provide initial $\boldsymbol{x}_k$, choose constant ξ and τ ($0 < \xi < 1$, $0 < \tau < 1$)**
2. **Find search direction $\boldsymbol{d}_k$** (e.g., by steepest descent method)
3. **Find $\alpha > 0$ so as to satisfy** $F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \le F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$ **(1)**
   (i) $\beta_{k,0} = 1, i = 0$
   (ii) if $F(\boldsymbol{x}_k + \beta_{k,i} \boldsymbol{d}_k) \le F(\boldsymbol{x}_k) + \xi \beta_{k,i} \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$ go to **step 4**, or go to (iii)
   (iii) $\beta_{k,i+1} = \tau \beta_{k,i}$ and go to (ii)
4. $\alpha = \beta_{k,i}$



y
$y = F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$
Satisfy eq. (1)
Satisfy eq. (1)
$y = F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$
$\alpha \boldsymbol{d}_k$
tangent: $y = F(\boldsymbol{x}_k) + \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$
$\alpha^{(0)}$

Line search (直線探索法): Armijo condition

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

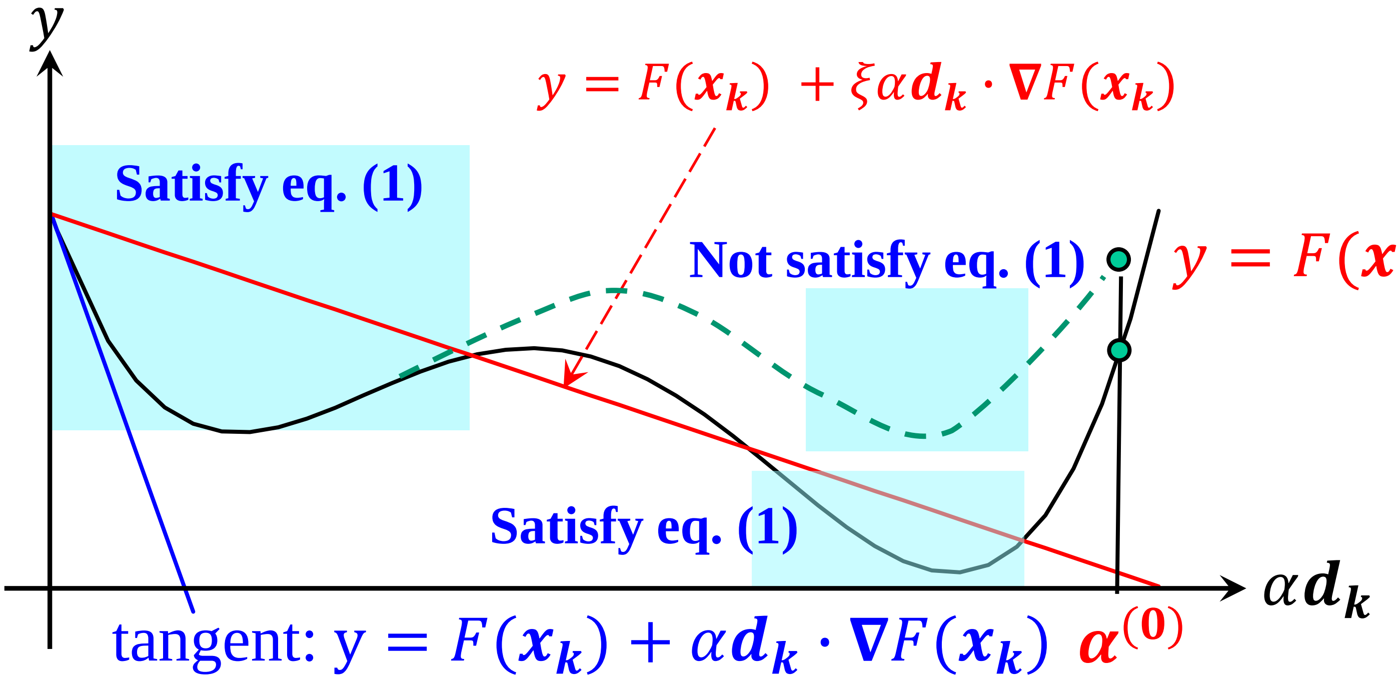
Armijo (アルミホ) condition (eq. (1)) and algorism:

1. **Provide initial $\boldsymbol{x}_k$, choose constant ξ and τ ($0 < \xi < 1$, $0 < \tau < 1$)**
2. **Find search direction $\boldsymbol{d}_k$** (e.g., by steepest descent method)
3. **Find $\alpha > 0$ so as to satisfy** $F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \leq F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$ **(1)**

   (i) $\beta_{k,0} = 1, i = 0$

   (ii) if $F(\boldsymbol{x}_k + \beta_{k,i} \boldsymbol{d}_k) \leq F(\boldsymbol{x}_k) + \xi \beta_{k,i} \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$ go to **step 4**, or go to (iii)

   (iii) $\beta_{k,i+1} = \tau \beta_{k,i}$ and go to (ii)
4. $\alpha = \beta_{k,i}$



y
$y = F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$
Satisfy eq. (1)
Not satisfy eq. (1)
$y = F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$
Satisfy eq. (1)
$\alpha \boldsymbol{d}_k$
tangent: y = $F(\boldsymbol{x}_k) + \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$
$\alpha^{(0)}$

Line search (直線探索法): Wolfe condition

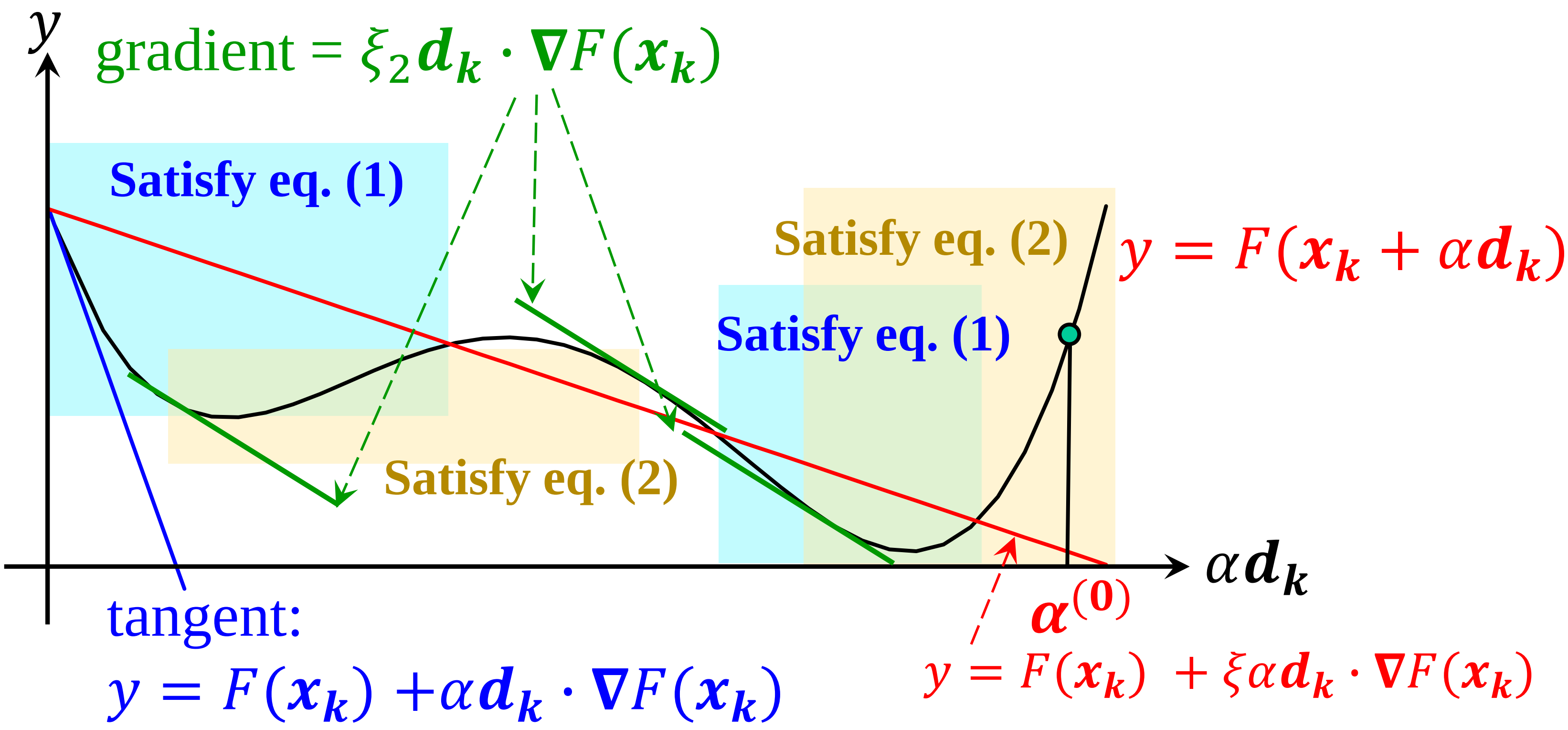
矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Wolfe (ウルフ) condition:

1. **Find search direction** $\boldsymbol{d}_k$ (e.g., by steepest descent method)
2. **Choose constants** ξ_1 **and** ξ_2 **that satisfy** $\mathbf{0} < \xi_1 < \xi_2 < \mathbf{1}$
3. **Find** $\alpha > 0$ **so as to satisfy:**

$$F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \le F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k) \qquad \textbf{(1)}$$

$$\xi_2 \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k) \le \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k) \qquad \textbf{(2)}$$



gradient $= \xi_2 \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$

Satisfy eq. (1)

Satisfy eq. (2)

Satisfy eq. (2)

Satisfy eq. (1)

$y = F(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$

tangent:
$y = F(\boldsymbol{x}_k) + \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$

$\boldsymbol{\alpha}^{(0)}$

$y = F(\boldsymbol{x}_k) + \xi \alpha \boldsymbol{d}_k \cdot \nabla F(\boldsymbol{x}_k)$

Bisection method (二分法) vs Golden-section search (黄金分割探索)

Bisection method：Find solution of $f(x) = 0$ for monotonous continuous function

Unique solution exists in the range $[x_0^{(0)}, x_2^{(0)}]$ if $\boldsymbol{f(x_0^{(0)})f(x_2^{(0)}) < 0}$

Add $x_1^{(0)}$ in $[x_0^{(0)}, x_2^{(0)}]$ $(x_0^{(0)} < x_1^{(0)} < x_2^{(0)})$

Case 1: If $f(x_0^{(0)})f(x_1^{(0)}) < 0$, solution is in $[x_0^{(0)}, x_1^{(0)}]$

Next search range is reduced to $x_0^{(1)} := x_0^{(0)}$, $x_1^{(1)} := x_3^{(0)} = \frac{x_0^{(0)}+x_1^{(0)}}{2}$, $x_2^{(1)} := x_1^{(0)}$

Case 2: If $f(x_1^{(0)})f(x_2^{(0)}) < 0$, solution is in $[x_1^{(0)}, x_2^{(0)}]$

Next search range is reduced to: $x_0^{(1)} := x_1^{(0)}$, $x_1^{(1)} := x_3^{(0)} = \frac{x_1^{(0)}+x_2^{(0)}}{2}$, $x_2^{(1)} := x_2^{(0)}$

Golden-section search: Find minimum for single downward convex continuous func $f(x)$

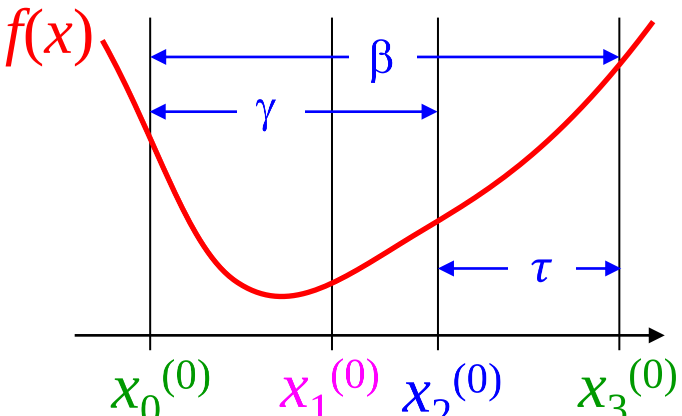
Unique solution exists in the range $[x_0^{(0)}, x_3^{(0)}]$ if $\boldsymbol{f(x_1^{(0)}) < f(x_0^{(0)}), f(x_3^{(0)})}$ for $x_0^{(0)} < x_1^{(0)} < x_3^{(0)}$

Add $x_2^{(0)}$ in $[x_0^{(0)}, x_3^{(0)}]$ $(x_0^{(0)} < x_1^{(0)} < x_2^{(0)} < x_3^{(0)})$

Case 1: if $f(x_1^{(0)}) < f(x_2^{(0)})$, solution is in $[x_0^{(0)}, x_2^{(0)}]$

Replace $x_2^{(0)}$ with $x_4^{(0)}$ in $[x_0^{(0)}, x_1^{(0)}]$

Next search range is reduced to $x_0^{(1)} := x_0^{(0)} < x_1^{(1)} := x_4^{(0)} < x_2^{(1)} := x_1^{(0)} < x_3^{(1)} := x_2^{(0)}$

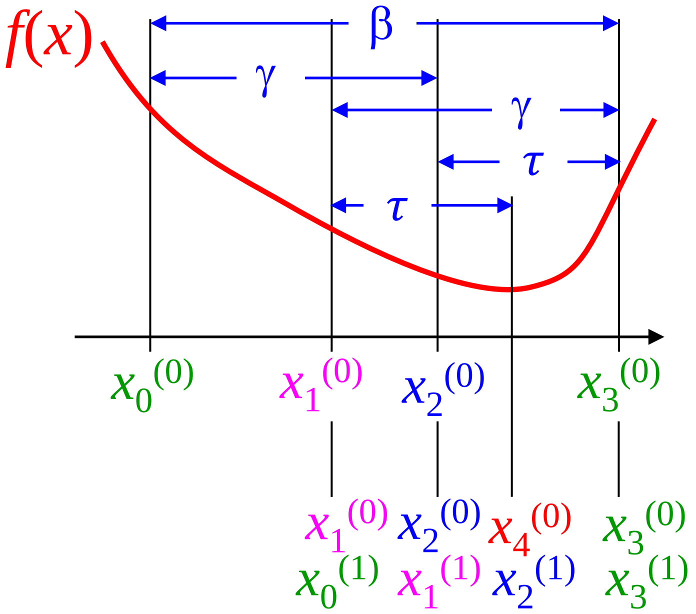


Golden-section search (黄金分割探索)

For downward convex continuous function, unique solution exists
in the range $[x_0^{(0)}, x_3^{(0)}]$ if **$f(x_1^{(0)}), f(x_2^{(0)}) < f(x_0^{(0)}), f(x_3^{(0)})$** for $x_0^{(0)} < x_1^{(0)} < x_2^{(0)} < x_3^{(0)}$

Case 1: if $f(x_1^{(0)}) < f(x_2^{(0)})$, solution is in $[x_0^{(0)}, x_2^{(0)}]$
Replace $x_2^{(0)}$ with $x_4^{(0)}$ in $[x_0^{(0)}, x_1^{(0)}]$
Next search range is reduced to $x_0^{(1)} := x_0^{(0)} < x_1^{(1)} := x_4^{(0)} < x_2^{(1)} := x_1^{(0)} < x_3^{(1)} := x_2^{(0)}$

Case 2: if $f(x_1^{(0)}) > f(x_2^{(0)})$, solution is in $[x_1^{(0)}, x_3^{(0)}]$
Replace $x_0^{(0)}$ with $x_4^{(0)}$ in $[x_2^{(0)}, x_3^{(0)}]$
Next search range is reduced to $x_0^{(1)} := x_1^{(0)} < x_1^{(1)} := x_2^{(0)} < x_2^{(1)} := x_4^{(0)} < x_3^{(1)} := x_3^{(0)}$


$f(x)$
β
γ
γ
τ
τ
$x_0^{(0)}$
$x_1^{(0)}$
$x_2^{(0)}$
$x_3^{(0)}$
$x_1^{(0)}$ $x_2^{(0)}$ $x_4^{(0)}$ $x_3^{(0)}$
$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$ $x_3^{(1)}$

Strategy: keep the ratio of $x_0^{(k)}, x_1^{(k)}, x_2^{(k)}, x_3^{(k)}$ constant for iteration steps

$$\beta = x_3^{(k)} - x_0^{(k)}$$
$$\gamma = x_2^{(k)} - x_0^{(k)} = x_3^{(k)} - x_1^{(k)}$$
$$\tau = \beta - \gamma$$
$$\mathbf{x_4^{(k)} = x_1^{(k)} + \tau}$$

To keep the ratio for next step (k+1)

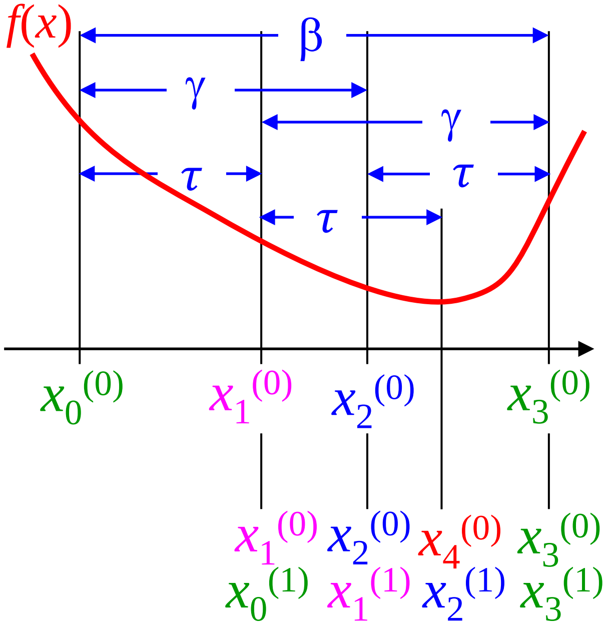
$$\beta : \gamma = x_3^{(k+1)} - x_0^{(k+1)} : x_2^{(k+1)} - x_0^{(k+1)}$$
$$= x_3^{(k)} - x_1^{(k)} : x_4^{(k)} - x_1^{(k)} = \gamma : \tau$$

$\tau = \beta - \gamma$から

$$\frac{\beta}{\gamma} = \frac{1+\sqrt{5}}{2}$$ Golden number

Golden-section search (黄金分割探索)

Minimum solution of downward convex continuous function $f(x)$


$f(x)$
β
γ
γ
τ
τ
τ
$x_0^{(0)}$
$x_1^{(0)}$
$x_2^{(0)}$
$x_3^{(0)}$
$x_1^{(0)}$ $x_2^{(0)}$ $x_4^{(0)}$ $x_3^{(0)}$
$x_0^{(1)}$ $x_1^{(1)}$ $x_2^{(1)}$ $x_3^{(1)}$

$$\frac{\beta}{\gamma} = \frac{1+\sqrt{5}}{2}$$

$$\boldsymbol{\eta = \frac{\tau}{\beta} = \frac{\beta-\gamma}{\beta} = 1 - \frac{2}{\sqrt{5}+1} = \frac{\sqrt{5}-1}{\sqrt{5}+1}}$$

1. For $x_0^{(0)} < x_1^{(0)} < x_2^{(0)} < x_3^{(0)}$, assign initial parameters as:

$$\beta^{(0)} = x_3^{(0)} - x_0^{(0)}$$
$$\tau^{(0)} = \eta\beta^{(0)}$$
$$x_1^{(0)} = x_0^{(0)} + \tau^{(0)}$$
$$x_2^{(0)} = x_3^{(0)} - \tau^{(0)}$$

2. Terminate if $\left|\beta^{(k)}\right| < EPS$

3. $\beta^{(k+1)} = \tau^{(k)}$
   $\tau^{(k+1)} = \eta\beta^{(k+1)}$

4. If $f(x_1^{(k)}) < f(x_2^{(k)})$, substitute $x_3^{(k)}$ for $x_4^{(k)} = x_2^{(k)} - \tau^{(k)}$ as:
   $x_0^{(k+1)} = x_0^{(k)}$, $x_1^{(k+1)} = x_2^{(k)} - \tau^{(k)}$, $x_2^{(k+1)} = x_1^{(k)}$, $x_3^{(k+1)} = x_2^{(k)}$
   If $f(x_1^{(k)}) > f(x_2^{(k)})$, substitute $x_0^{(k)}$ for $x_4^{(k)} = x_2^{(k)} + \tau^{(k)}$ as:
   $x_0^{(k+1)} = x_1^{(k)}$, $x_1^{(k+1)} = x_2^{(k)}$, $x_2^{(k+1)} = x_1^{(k)} + \tau^{(k)}$, $x_3^{(k+1)} = x_3^{(k)}$
   Go to step 1

Methods of non-linear (NL) optimization

To find a minimum (maximum) of target function *F*(*x*):

Direct search method (直接探索法)

Trial and errors to find a minimum,
but following a certain defined procedures

Gradient method (勾配法)**:**

Use first differential to find the direction of minimum

Steepest descent method (SD, 最急降下法)

Search minimum/maximum only by first derivatives

Concept: Minimum/Maximum may be found

in the direction $(\partial F(x_i)/\partial x_i)$

$$x_i^{(k+1)} = x_i^{(k)} - \alpha \partial F(x_i^{(k)})/\partial x_i$$

Need to choose/find an appropriate α

so as to take the minimum $F(x_i^{(k)})$

Variations to choose α:

(i) Simple: Choose small α

(ii) Direct search (直接探索)

Armijo / Wolfe condition

Steepest Descend (SD) method (最急降下法)

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Search minimum only by first derivatives. Simplest one among gradient methods

- **SD: S^2 would decrease in the vector $-(df / dx_i)dx_i$**

  $x_i^{(k+1)} = x_i^{(k)} - \alpha(df / dx_i)$

  α_k may be a small constant step
  or determined by a line search method

_ex._ in right figure:

$S^2 = f(x_i) = 5x_1^2 + x_2^2$, initial $x_1 = 0.7$, $x_2 = 1.5$

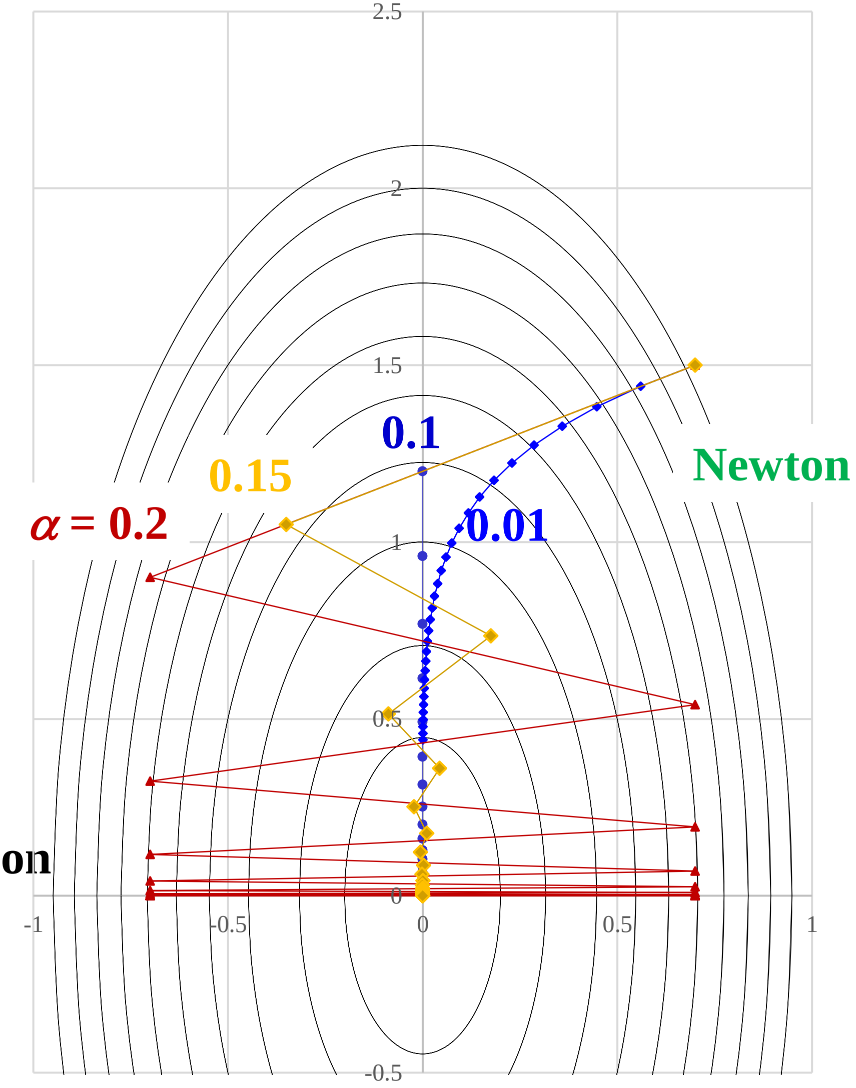
- **Newton method**

  One cycle calculation provides the final solution for quadratic problems

  楕円問題の場合は一度目の計算で最適値に到達

- **SD method**

  α = 0.3: Diverged (not shown in the graph)
  0.2, 0.15: Converged, but oscillated
  0.1: Reach final solution by one cycle calculation
  0.01: Not oscillated, but slowly converged

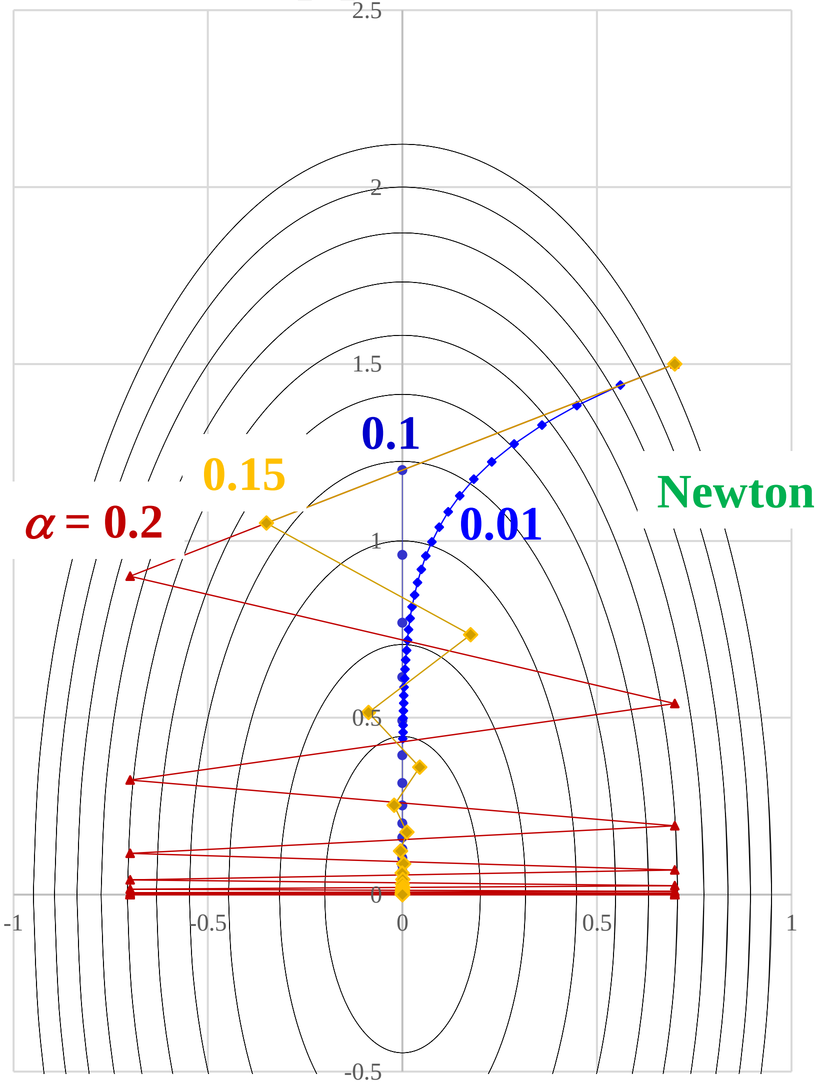


Problem: If S^2 is highly anisotropic, the SD direction would be different largely from the minimization direction S^2 が大きく非対称な場合、最急勾配方向は最小値方向とは
大きく異なることがある

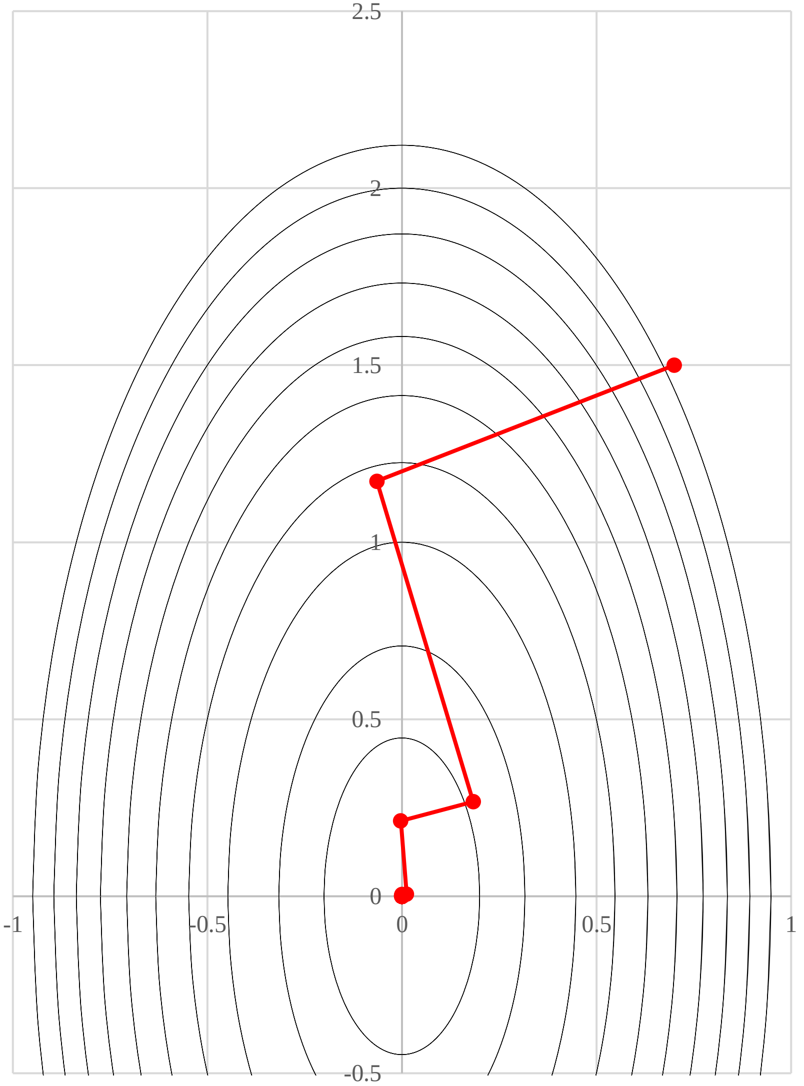
=> **Conjugate Gradient (CG)** method (共役勾配法)

Steepest descend method

Without line search:
use fixed step parameter



With line search



SD method in Deep Learning

- **All batch data are divided to mini batches, and apply SD to each mini batch**

Example:

$S^2 = f(a, b; x_1, x_2) = ax_1^2 + bx_2^2$, $a = 5$, $b = 1$

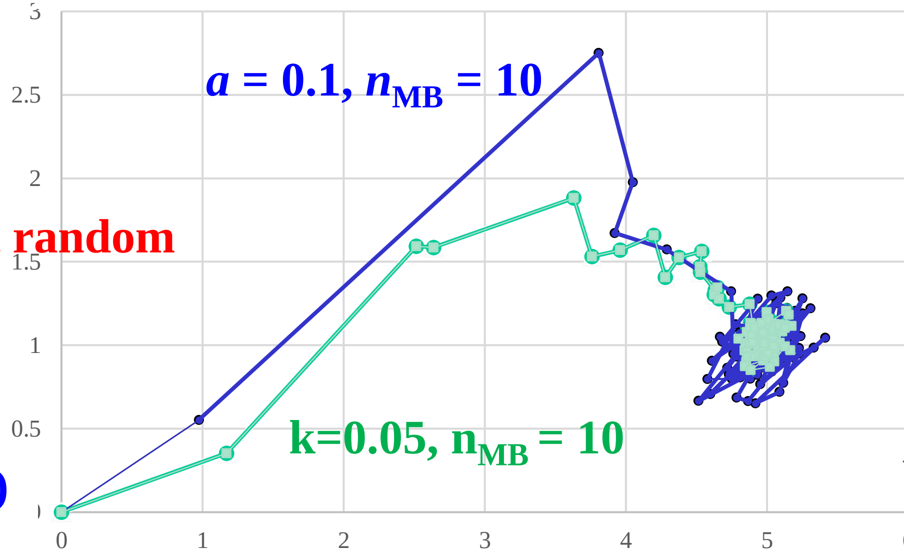
1000 batch data (x_{1i}, x_{2i}, $f(x_i)$) are generated at random

(note: the data were re-generated for different runs)

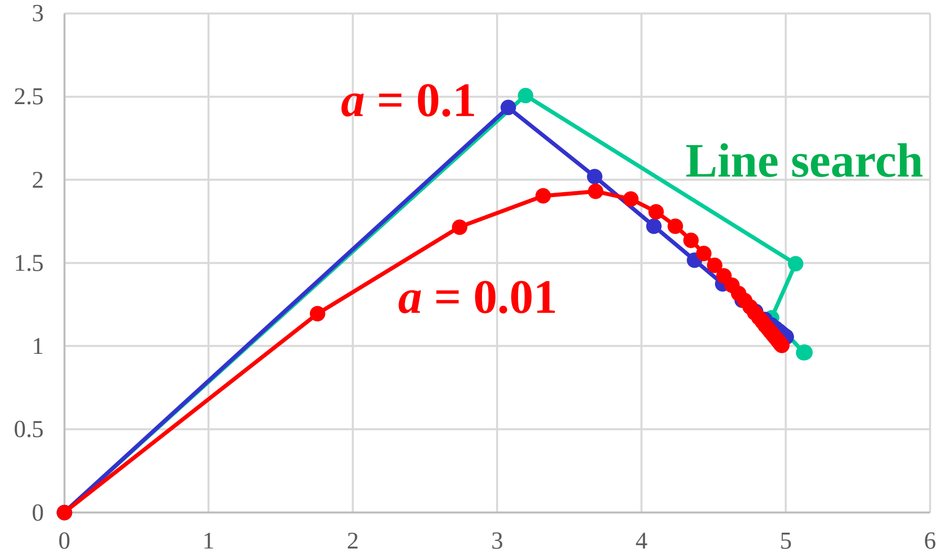
Speculate *a* and *b*

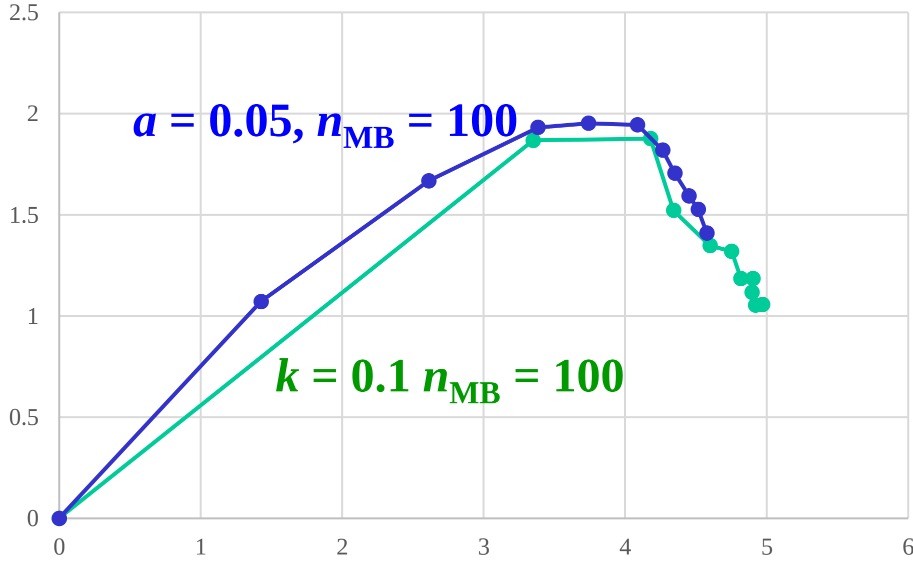
Initial *a* = 0, *b* = 0

DL: SD method



SD (Convergence iterations with all batch data)





Multiple parameter Newton-Raphson method

Extend to multiple parameter optimization: Minimize $F(x_l)$

$$f_k(x_l) = \partial F(x_l)/\partial x_k = 0$$

To solve $f_k(x_l) = 0\ (k, l = 1, 2, \cdots N)$

$$f_k(x_l + \delta x_l) \sim f_k(x_l) + \sum_{k'} \delta x_{k'} \partial f_k(x_l)/\partial x_{k'} = 0$$

$$\Rightarrow x_{l,1} = x_{l,0} - (\partial f_k(x_l)/\partial x_{k'})^{-1}(f_k) = x_{l,0} - (F''_{kk'})^{-1}(F'_k)$$

$$F''_{kk'} = \frac{\partial^2 F(x)}{\partial x_k \partial x_{k'}}$$ **Hessian matrix** (ヘッセ行列)

(ヘッセ行列の固有値をヘッシアンと呼ぶ)

Hessian matrix is not always positive definite (正定値であるとは限らない)

(Maximum, Saddle point 極大値、鞍点**)**

=> *F*” dose not always give decreasing direction

Convert *F*” to positive definite and suppress divergence

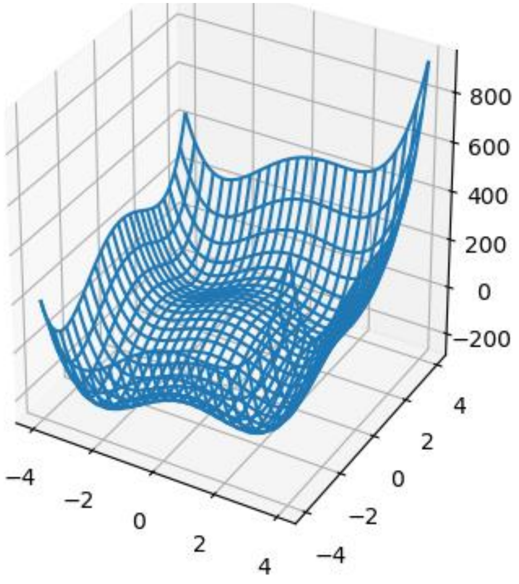
$$x_{l,1} = x_{l,0} - (F''_{kk'} + \lambda I)^{-1}(F'_k)$$

λ: Dumping Factor

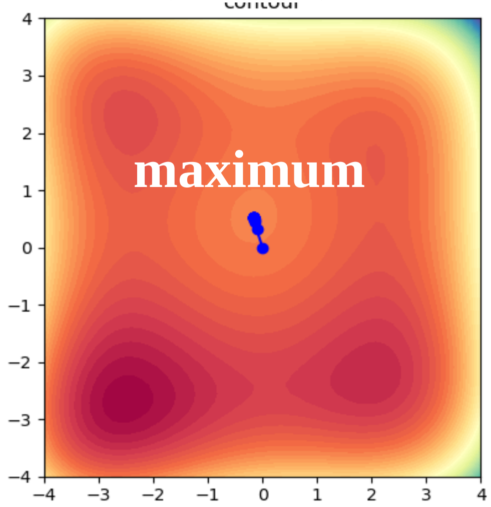
Program: optimize-newton-raphson2d.py

$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30y - 30y^2 + 3y^4 + 3xy^2$

Usage: python optimize-newton-raphson2d.py x_0 y_0

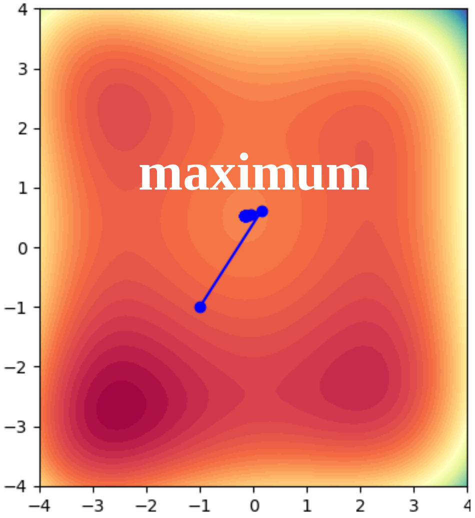


From (0.0 0.0) Newton



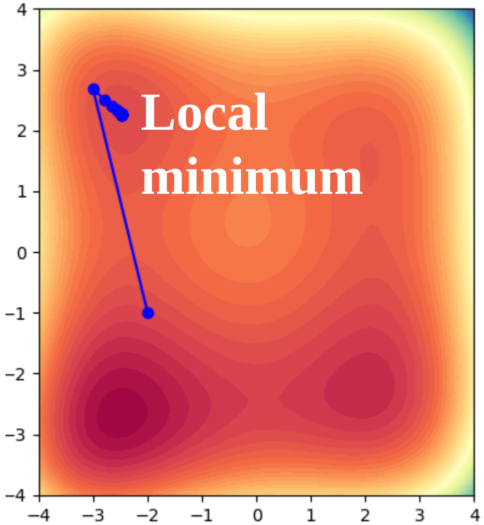
maximum

From (-1.0 -1.0)



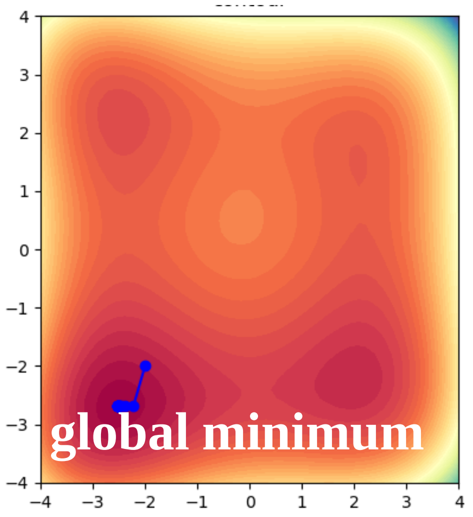
maximum

From (-2.0 -1.0)



Local minimum

From (-2.0 -2.0)



global minimum

Quasi-Newton method (準Newton法)

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Target function to minimize: $\boldsymbol{F(x_l)}$

Iteration: $\boldsymbol{{x_l}^{(i+1)} = {x_l}^{(i)} - \left(\partial^2 F/\partial x_k \partial x_{k'}\right)^{-1} (\partial F/\partial x_k)}$

$\boldsymbol{F''_{kk'} = \partial^2 F/\partial x_k \partial x_{k'}}$**: Hessian (ヘッセ) matrix**

Issues of Newton method:

(1) Calculation of Hessian matrix is very high cost as it is a 2D matrix

(2) Eigen value of Hessian matrix can be negative => lead to maximum

(3) Easy to diverge

Quasi-Newton method:

(1,2) Hessian matrix is approximated from 1st differentials

(3) Line search algorism is applied along the search direction

$$-(\partial^2 F/\partial x_k \partial x_{k'})^{-1}(\partial F/\partial x_k)$$

Davidon-Fletcher-Powell (DFP) method

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

$$F\left({x_l}^{(k)}+\alpha d\right)=F\left({x_l}^{(k)}\right)+\alpha\nabla F({x_l}^{(k)})^T d+\frac{1}{2}\alpha^2 d^T B^{(k)} d\sim 0$$

Search direction *d* is determined from $B^{(k)}d=-\nabla F({x_l}^{(k)})$

DFP method: The first formulation of quasi-Newton method

$$s^{(k)}=x^{(k+1)}-x^{(k)},\quad y^{(k)}=\nabla F\left({x_l}^{(k+1)}\right)-\nabla F({x_l}^{(k)})$$

$$B^{(k+1)}=B^{(k)}+\frac{\left(y^{(k)}-B^{(k)}s^{(k)}\right)\cdot y^{(k)T}+y^{(k)}\cdot\left(y^{(k)}-B^{(k)}s^{(k)}\right)^T}{s^{(k)T}\cdot y^{(k)}}$$

$$-\frac{s^{(k)T}\cdot\left(y^{(k)}-B^{(k)}s^{(k)}\right)}{\left(s^{(k)T}\cdot y^{(k)}\right)^2}y^{(k)}\cdot y^{(k)T}$$

$$=B^{(k)}-\frac{B^{(k)}s^{(k)}\cdot y^{(k)T}+y^{(k)}\cdot\left(B^{(k)}s^{(k)}\right)^T}{s^{(k)T}\cdot y^{(k)}}+\left(1+\frac{s^{(k)T}B^{(k)}s^{(k)}}{s^{(k)T}\cdot y^{(k)}}\right)$$

Broyden-Fletcher-Goldfarb-Shanno (BFGS) method

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

BFGS method： **Regarded as most efficient among quasi-Newton methods**

$$\boldsymbol{s}^{(k)} = \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}, \quad \boldsymbol{y}^{(k)} = \nabla F\left(\boldsymbol{x}_l{}^{(k+1)}\right) - \nabla F\left(\boldsymbol{x}_l{}^{(k)}\right)$$

$$\boldsymbol{B}^{(k+1)} = \boldsymbol{B}^{(k)} - \frac{\boldsymbol{B}^{(k)}\boldsymbol{s}^{(k)}\left(\boldsymbol{B}^{(k)}\boldsymbol{s}^{(k)}\right)^T}{\boldsymbol{s}^{(k)T}\boldsymbol{B}^{(k)}\boldsymbol{s}^{(k)}} + \frac{\boldsymbol{y}^{(k)}\boldsymbol{y}^{(k)T}}{\boldsymbol{s}^{(k)T}\cdot\boldsymbol{y}^{(k)}}$$

Algorism：

STEP 0: Provide initial values $x^{(0)}$ and initial matrix $\mathrm{B}^{(0)}$ (can be unit matrix)

STEP 1: Search direction $d^{(\mathrm{k})}$ is determined from $\boldsymbol{B}^{(k)}\boldsymbol{d} = -\nabla F\left(\boldsymbol{x}_l{}^{(k)}\right)$

STEP 2: Step width $\alpha^{(\mathrm{k})}$ is determined by line search algorism

STEP 3: Calculate $x^{(\mathrm{k+1})} = x^{(\mathrm{k})} + \alpha^{(\mathrm{k})}d^{(\mathrm{k})}$

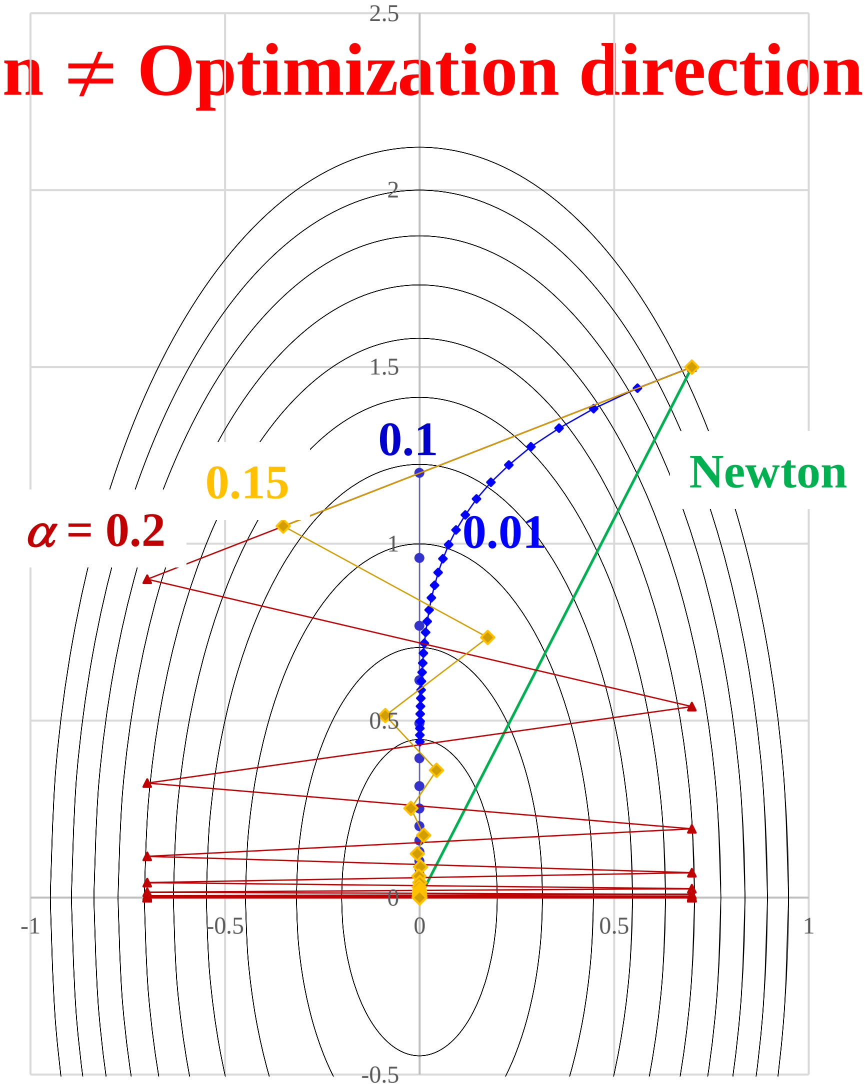
STEP 4: End if self-consistency is achieve.

If not, go to STEP 5

STEP 5: Calculated $s^{(\mathrm{k})}$ and $y^{(\mathrm{k})}$, and then $B^{(\mathrm{k+1})}$, and go to STEP 1

SD vs. Newton-Raphson methods

Steepest direction ≠ Optimization direction



Improve SD method to follow optimization directions

Conjugate Gradient method (共役勾配法)

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

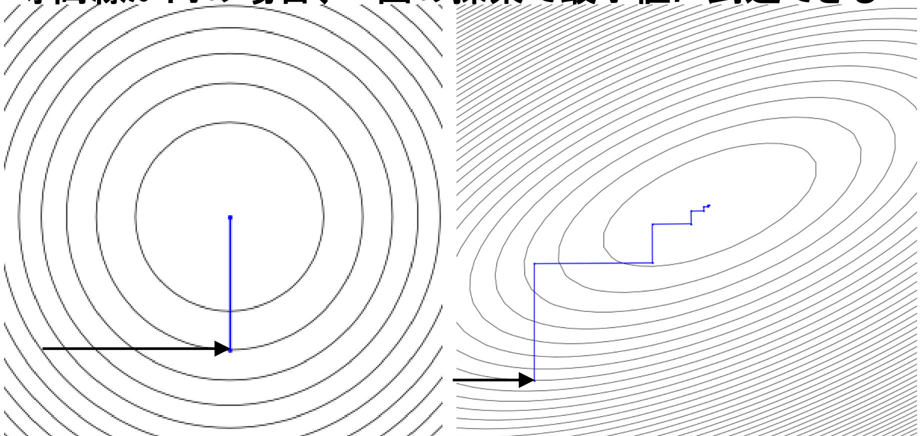
Vectors *u* and *v* satisfy $u^tAv = 0$ *for a matrix A:* *u* and *v and conjugate with each other*

- **For quadratic function, repetition of the conjugate direction will find the minimum in finite cycles if exact line search is employed**

共役な探索方向に沿って正確な直線探索を実行 => 有限回の反復で2次関数の最小解に到達

Case contour is a circle, one cycle calculation reaches the minimum

等高線が円の場合、一回の探索で最小値に到達できる



Conjugate vectors and ellipsoido – circle conversion $\quad u^TP^TPv = u^TAv = 0$

1. **Give initial value x_0**
2. **Initial direction *d* is determined by SD**

$$\mathbf{d} = -\nabla f$$

3. **Find x_{k+1} using appropriately chosen α_k**

$$x_{k+1} = x_k + \alpha_k d_k$$

α_k may be a small constant step or determined by a line search method

4. **Search direction is updated by**

$$\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$$

$$\mathbf{d}_{k+1} = -\nabla f(\mathbf{x}_{k+1}) + \frac{\nabla f(\mathbf{x}_{k+1})^T \mathbf{y}_k}{\mathbf{d}_k^T \mathbf{y}_k} \mathbf{d}_k$$

5. Repeat 3 – 4 to reach convergence

As the freedom of cg directions is the number of parameters (n_{param}), need to go back to **2** to reset $\boldsymbol{d}_{\mathbf{k}}$ at some interval (typically n_{param}, necessary for $n_{\text{param}} = 2$).

Ex.: Curve fit to powder XRD peak

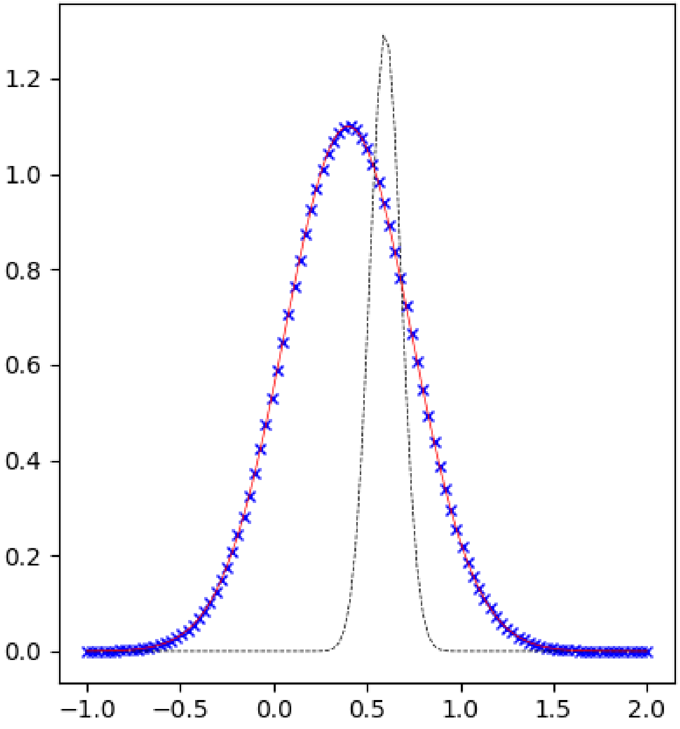
Usage: python peakfit-scipy-minimize.py I0 x0 w

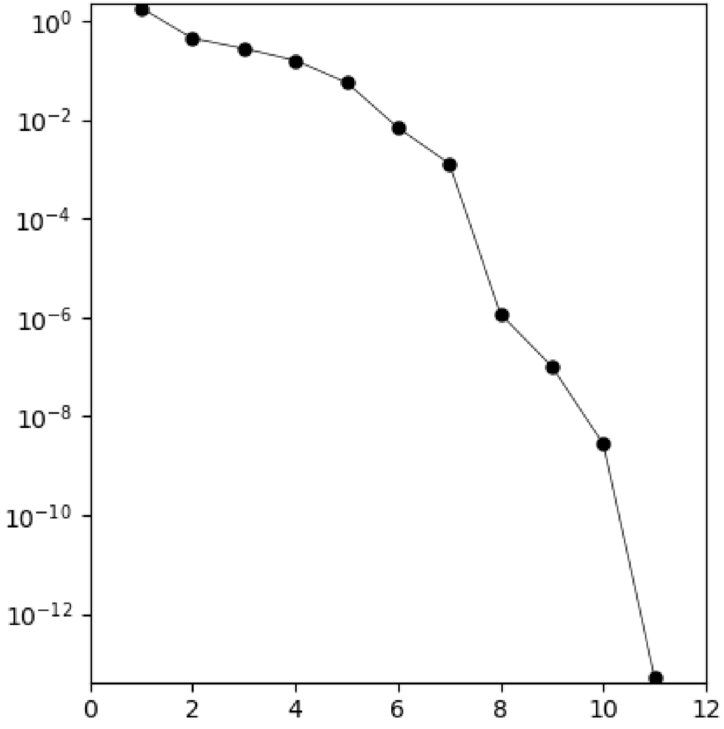
uses **scipy.minimize()** function, employs conjugate gradient method

python peakfit-scipy-minimize.py

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.6, w = 0.1



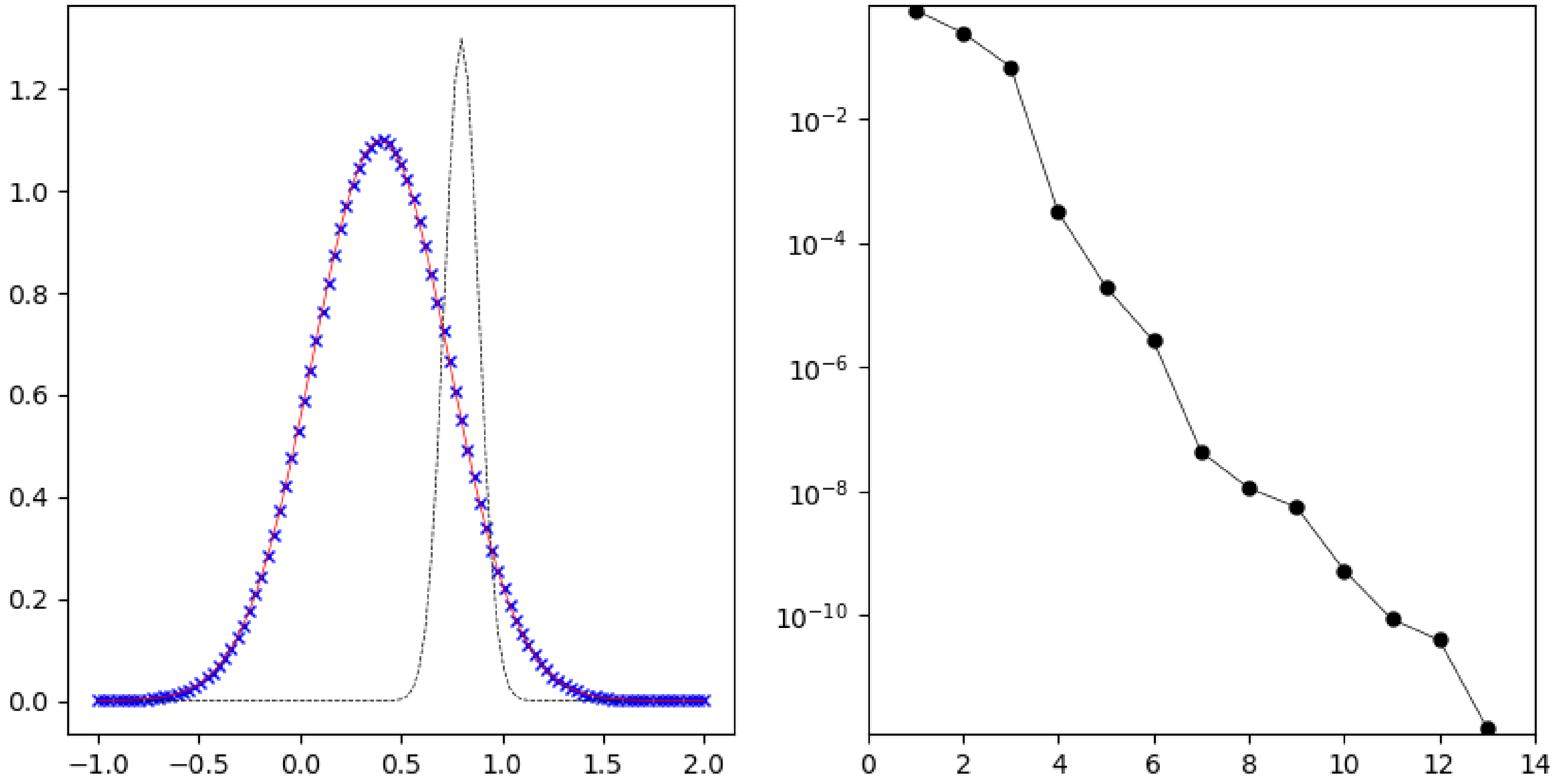


Converging range

```
python peakfit-scipy-minimize.py 1.3 0.8 0.1
```

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.8, w = 0.1

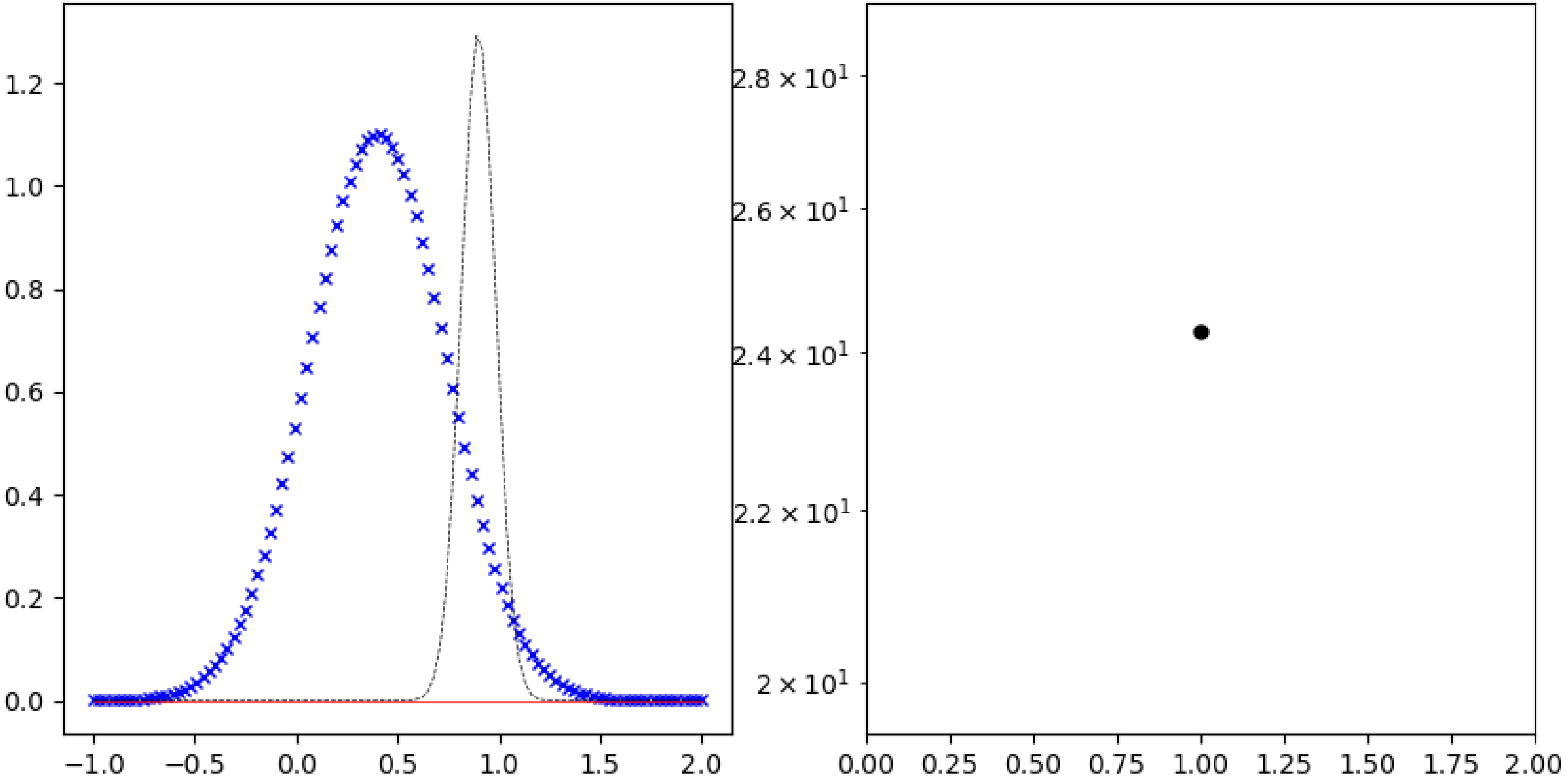


Diverged

```
python peakfit-scipy-minimize.py 1.3 0.9 0.1
```

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.9, w = 0.1



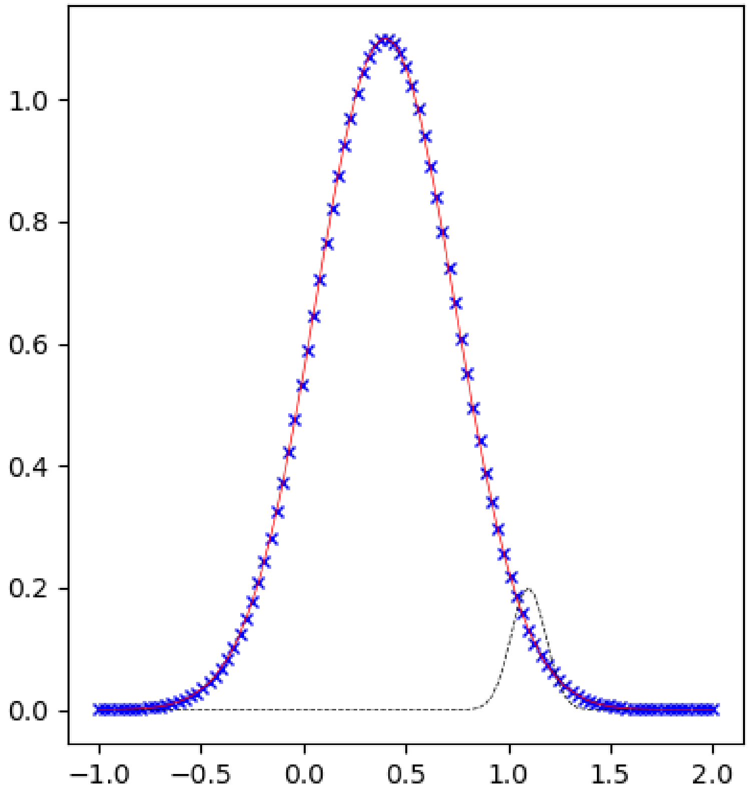
Initial peak must be in the FWHM of the target peak

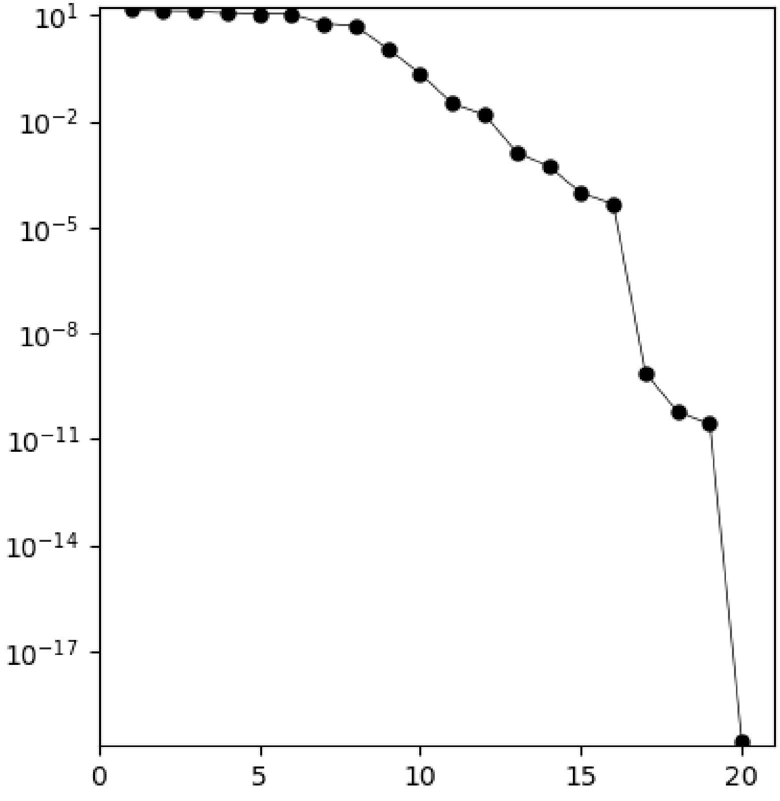
Converged

python peakfit-scipy-minimize.py 0.2 1.1 0.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 0.2, x0 = 1.1, w = 0.1





If I0 is close to the target curve, it can be converged

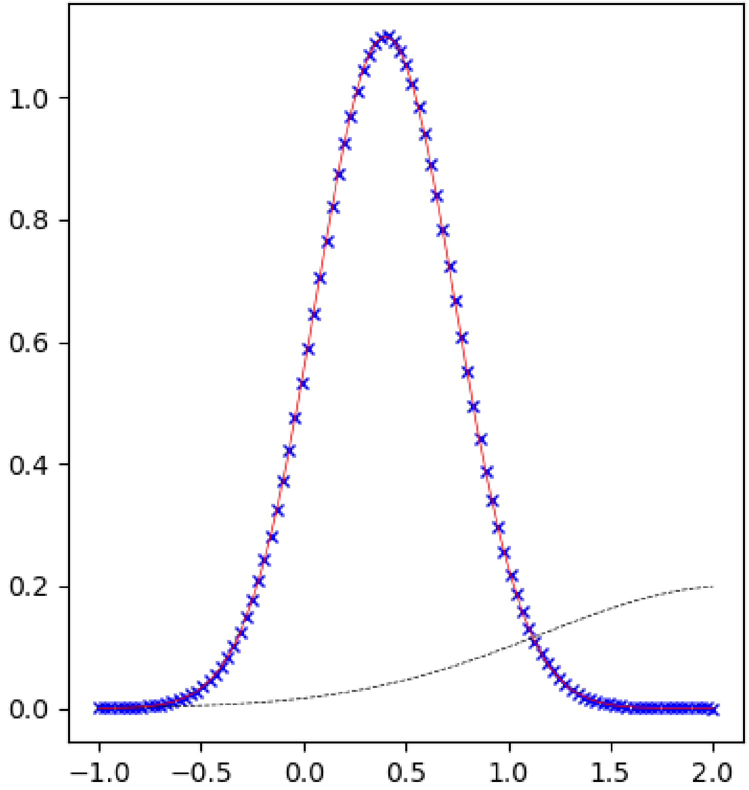
even if the initial peak position is out of the FWHM of the target peak

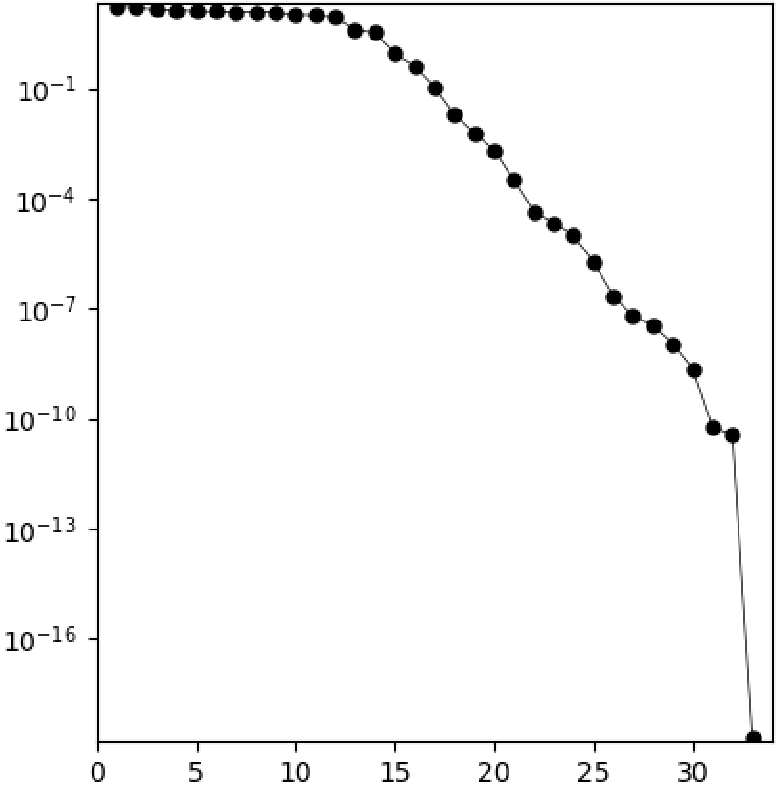
Converged

python peakfit-scipy-minimize.py 0.2 2.1 1.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 0.2, x0 = 2.1, w = 1.1





If peaks are overlapped satisfactory, can be converged

Marquart method (マーカート法)

Minimize a square sum of *m* functions $f_j(x_i)$ with *N* parameters

$$F\left(x_i\right)=\sum_{j=1}^{m} f_j(x_i)^2$$

Approximate by

$$f_j(x_i+\delta x_i)\sim f_j(x_i)+\left(\frac{\partial f_j}{\partial x_k}\right)\left(\delta x_i\right)= f_j(x_i)+\mathbf{A}\boldsymbol{\delta}x_{\mathbf{i}} \qquad A_{jk}=\frac{\partial f_j}{\partial x_k}$$

$$F(x_i+\delta x_i)\sim F(x_i) \quad +2\sum_{j,k} f_j A_{jk}\delta x_k+\sum_{j,k,k\prime} A_{jk}A_{ik\prime}\delta x_k \delta x_{k\prime}$$

$$\frac{\partial F\left(x_i\right)}{\partial \delta x_k}\sim 2\sum_j\left(A_{jk}f_j+\sum_k A_{ik}A_{jk}\delta x_j\right)=0$$

$$\boldsymbol{\delta x}=-(\mathbf{A}^{\mathbf{t}}\mathbf{A})^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$ **Gauss-Newton method**

Levenberg-Marquart method

$$\boldsymbol{\delta x}=-(\mathbf{A}^{\mathbf{t}}\mathbf{A}+\lambda I)^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$

λ: dumping factor

$$\boldsymbol{\delta x}=-(\mathbf{A}^{\mathbf{t}}\mathbf{A}+\lambda\mathrm{diag}(\mathbf{A}^{\mathbf{t}}\mathbf{A}))^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$

**e.g. chosen proportional
to diagonal sum of A^{t}A**

Simplex method (単体法, Amoeba法)

(Nelder-Mead algorithm)

服部力、名取亮、小国力 監修、Fortranによる数値計算ソフトウェア、丸善株式会社（１９８９年）

Simplex: Polyhedron formed by (n+1) vertexes in n-dimension space

(単体: n次元空間で (n+1) 個の頂点が作る多面体)

Minimize $F(x_i)$

1. (n+1) initial values $\boldsymbol{x}_i$ (i = 1, 2,･･･, n+1) => Sort $F(\boldsymbol{x}_i)$ so that $F(\boldsymbol{x}_i) > F(\boldsymbol{x}_{i'})$ ($i < i'$)

   $\boldsymbol{x}_h = \boldsymbol{x}_1, \boldsymbol{x}_l = \boldsymbol{x}_{n+1}$

2. Average except the maximum vertex $\boldsymbol{x}_i$ $x_G = \sum_{i=2} x_i / n$

3. New x will be examined along the line $\boldsymbol{x}_1 - \boldsymbol{x}_G$ by the following selections

(i) Reflection (鏡映) : $\boldsymbol{x}_R = (1+\alpha)\boldsymbol{x}_G - \alpha\boldsymbol{x}_1$ ($\alpha > 0$, *ex.* 1.0)

(ii) Expansion (拡大) : $\boldsymbol{x}_E = \gamma\boldsymbol{x}_r + (1-\gamma)\boldsymbol{x}_G$ ($\gamma > 0$, *ex.* 2.0)

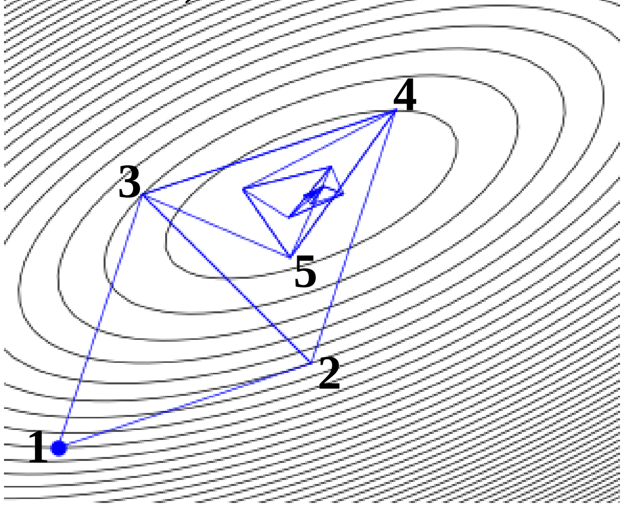
(iii) Contraction (収縮) : $\boldsymbol{x}_C = \beta\boldsymbol{x}_1 + (1-\beta)\boldsymbol{x}_G$ ($0 < \beta < 1$, *ex.* 0.5)

(iv) Reduction (縮小) : $\boldsymbol{x}_{RD} = (\boldsymbol{x}_1 + \boldsymbol{x}_l) / 2$

4. Replace $\boldsymbol{x}_1$ with the x in (i) – (iv) that firstly satisfies

   $F(\boldsymbol{x}) < F(\boldsymbol{x}_1)$

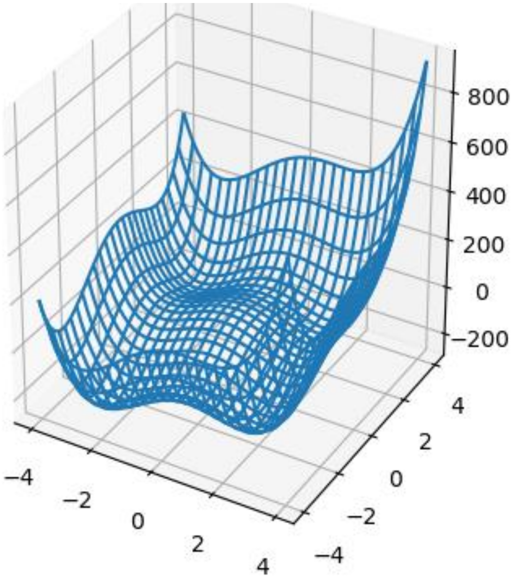
5. Repeat 2 - 4



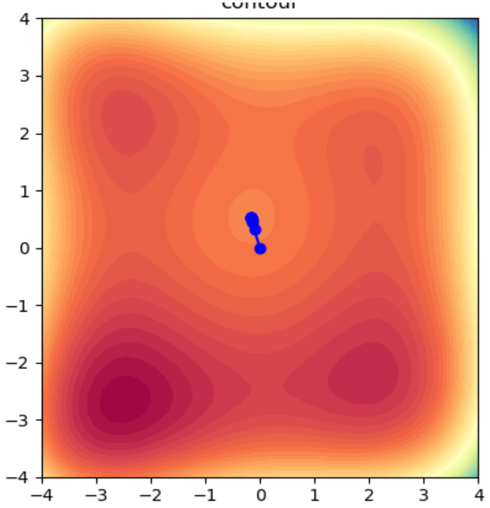
Comparison

$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30y - 30y^2 + 3y^4 + 3xy^2$

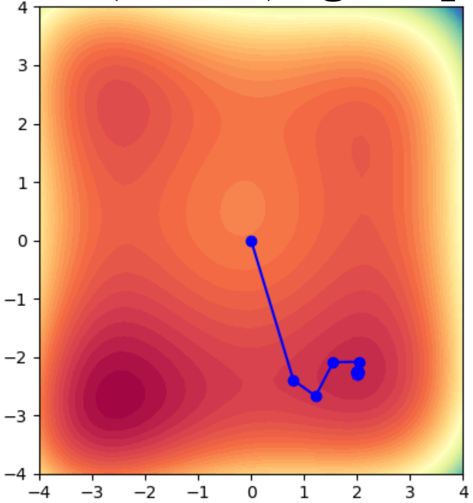
Programs: optimize-sd-cg2d-linesearch.py, optimize-newton-raphson2d.py



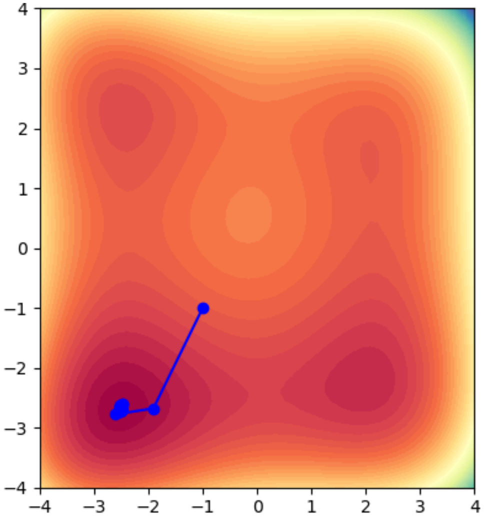
From (0.0 0.0) Newton



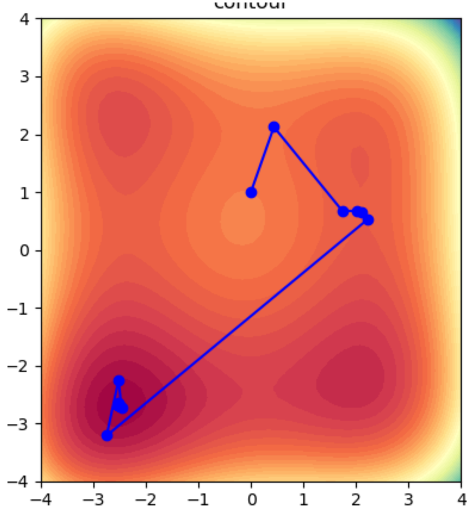
From (0.0 0.0) cg simple



From (-1.0 -1.0) cg simple



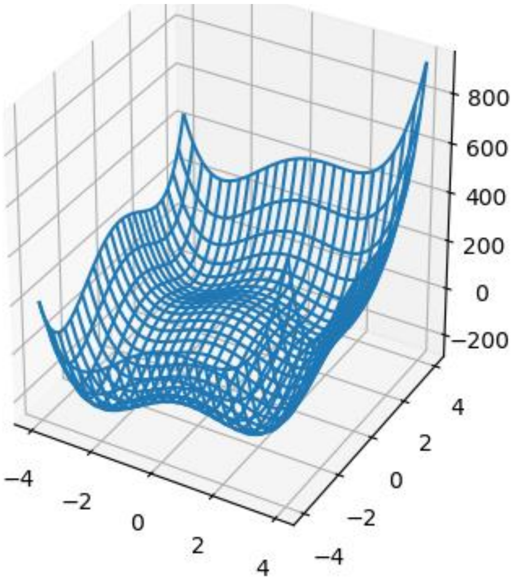
From (0.0 1.0) SD armijo



Comparison

$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30\mathrm{y} - 30y^2 + 3y^4 + 3xy^2$

Program not distributed



Main algorism:

Newton, DFP, BFGS

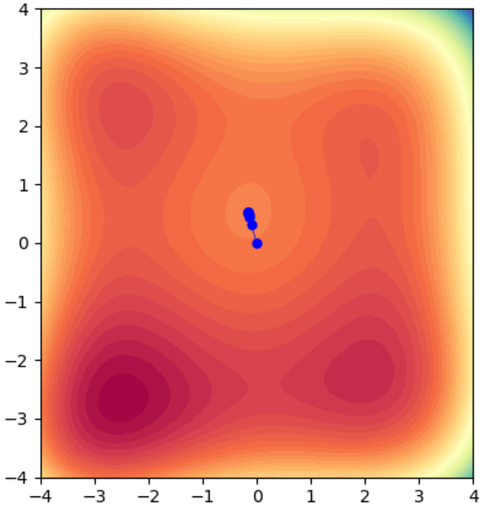
SD, CG

Simplex

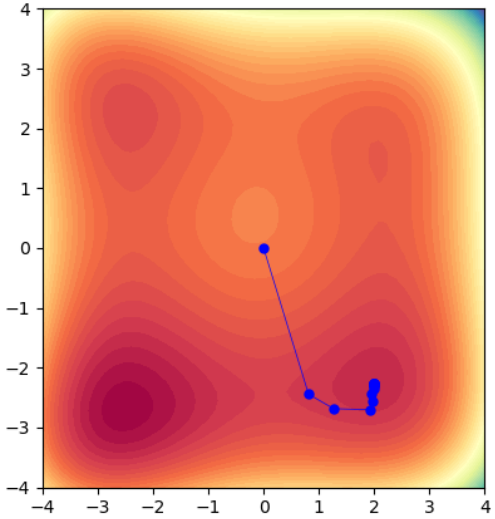
Line search:

Golden, Armijo

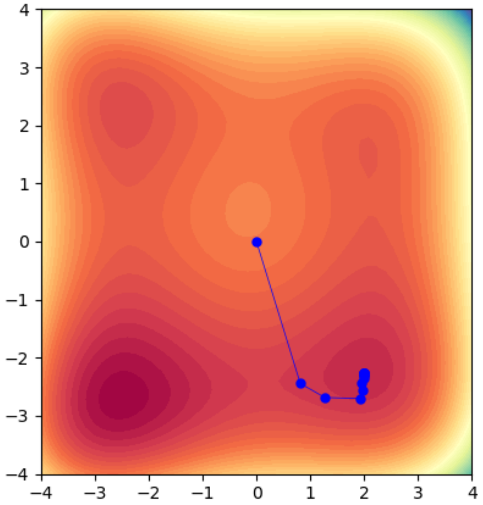
From (0.0 0.0) Newton



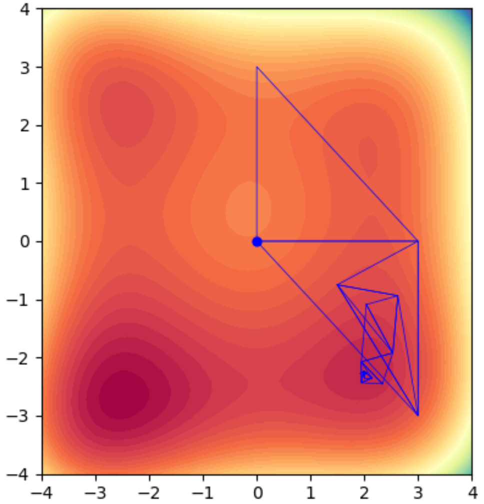
From (-1.0 -1.0) DFP golden



From (0.0 0.0) BFGS golden



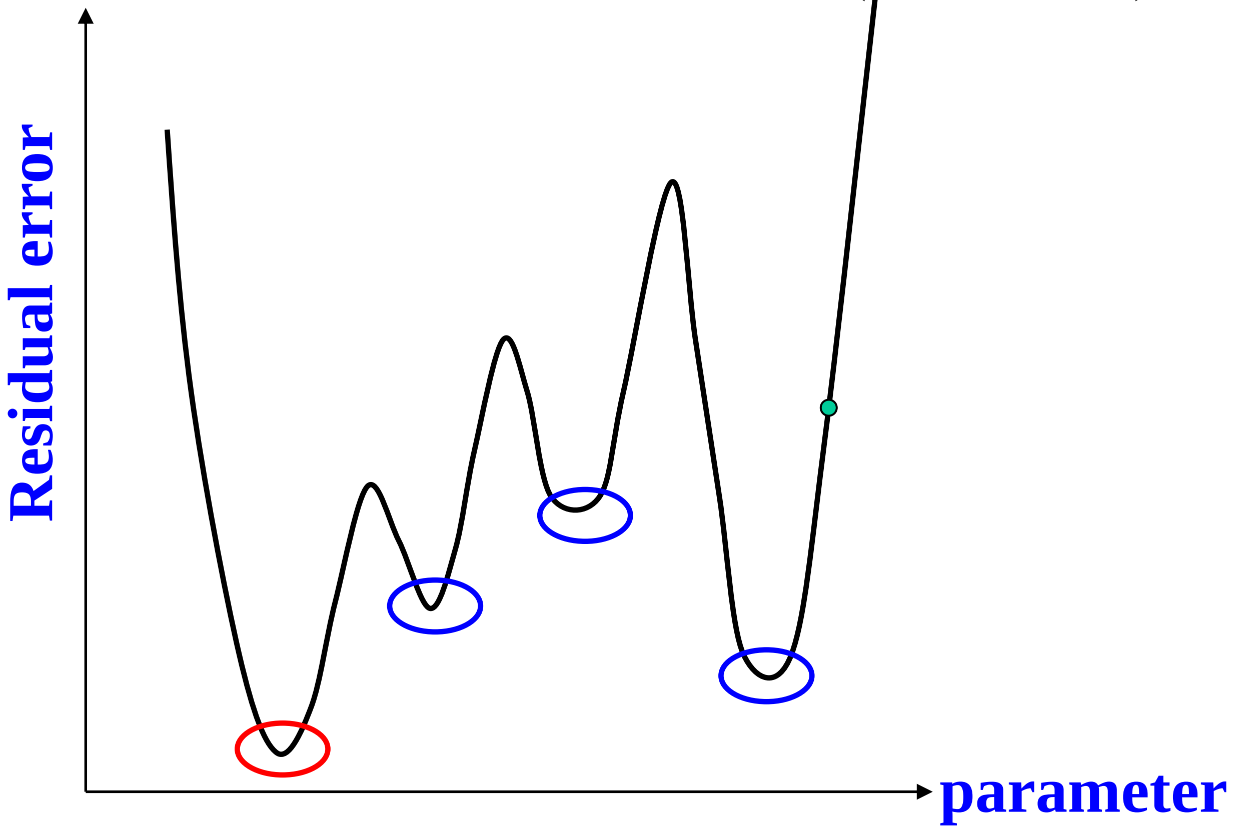
From (0.0 1.0) Simplex



Notes for NL optimization

- Solutions may be more than one
- Final solution is not obtained by one step calculation
- **Convergence must be confirmed**
- **Confirm the solution is the global minimum** (大域最小値)

  ⇔ **Often fall in a local minimum** (局所極小値)



Features of NL optimization algorisms

Convergence	A	B
Speed	×	○
Stability	○	×
Global convergence	○	×
For:	Initial cycles	Later cycles for fast convergence

A　: Simplex (単体法)

A,B: with line search algorism:

Conjugate Gradient (CG, 共役勾配法)

Steepest Descent (SD, 最急降下法)

Quasi Newton methods

- **Davidson-Fletcher-Powell (DFP)**
- **Broyden-Fletcher-Goldfarb-Shanno (BFGS)**

B　: Newton-Raphson method

Methods of non-linear (NL) optimization

To find a minimum (maximum) of target function $F(x)$:

Gradient method (勾配法)**: Use first differential to find the direction of minimum**

- **Newton-Raphson method:**
  Use 1st and 2nd differentials to efficiently find minimum
- **Quasi-Newton method** (準Newton法)**:**
  2nd differential matrix is approximated from 1st differentials.
  Line search method is combined to improve global convergence.
- **Steepest Descent method** (最急降下法)**:**
  Only 1st differentials are used to search minimum
- **Conjugate Gradient method** (共役勾配法)**:**
  Search direction is corrected by conjugate gradient of 1st differentials
- **Marquart method**
  For least-squares fitting of $f_j(x_i)$, 2nd differential matrix is build from 1st differentials of $f_j(x_i)$

Direct search method (直接探索法)

- **Simplex method** (単体法)
  Search minimum by trial-and-error with a defined procedure

Features of NL optimization

- **Newton-Raphson method: Gradient method**
  Use second derivatives (Hessian matrix)
  Fast convergence, easily diverged, complex program
  May reach to a maximum if Hessian matrix is not positive definite.
- **Quasi Newton method: DFP, BFGS, Broyden etc**
  Hessian matrix is iteratively approximated from 1st differentials.
  Better convergence by combining with linear search algorisms.
- **Steepest Descent:**
  Use first derivatives only
  Simple program, Slower convergence than NR and CG
- **Conjugate Gradient:**
  Use conjugate direction for efficient search
  Better convergence than NR, faster than SD, complex program
- **Marquart:**
  Use first derivatives of $f_j(x_i)$
  Simple program, Slower convergence than NR
- **Simplex: Direct search**
  Trial and error with a pre-determined selections of next candidate parameters
  Very slow but good convergence

Fourier transformation

フーリエ変換

Fourier series expansion (Fourier級数展開)

Period: *T*

$$x(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos \frac{2\pi n}{T} t + b_n \sin \frac{2\pi n}{T} t \right)$$

$$a_n = \frac{2}{T} \int_0^T x(t) \cos \frac{2\pi n}{T} t dt$$

$$b_n = \frac{2}{T} \int_0^T x(t) \sin \frac{2\pi n}{T} t dt$$

$$x(t) = \sum_{n=-\infty}^{\infty} c_n \exp\left(i \frac{2\pi n}{T} t \right)$$

$$c_n = \frac{1}{T} \int_0^T x(t) \exp\left(-i \frac{2\pi n}{T} t \right) dt$$

Riemann–Lebesgue lemma
(リーマン・ルベーグの補題): $\lim_{n \to \infty} c_n = 0$

Fourier transformation

Take limit to *T* => ∞ for Fourier series expansion

$$\begin{cases} \textbf{FT} \quad F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(i\omega t)dt \\ \textbf{IFT} \quad f(t) = \dfrac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\exp(-i\omega t)d\omega \end{cases}$$

$$\begin{cases} \textbf{FT} \quad F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(i2\pi ft)dt \\ \textbf{IFT} \quad f(t) = \int_{-\infty}^{\infty} F(\omega)\exp(-i2\pi ft)d\omega \end{cases}$$

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**

  Width *W* Gauss func is converted to width W^{-1} Gauss func
- **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

*l*LSQ for general function

$$f\left(x\right)=\sum_{k=1}^{n} a_k f_k(x) \qquad S=\sum_{i=1}^{N}\left(y_i-\sum_{k=1}^{n} a_k f_k(x_i)\right)^2$$

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

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

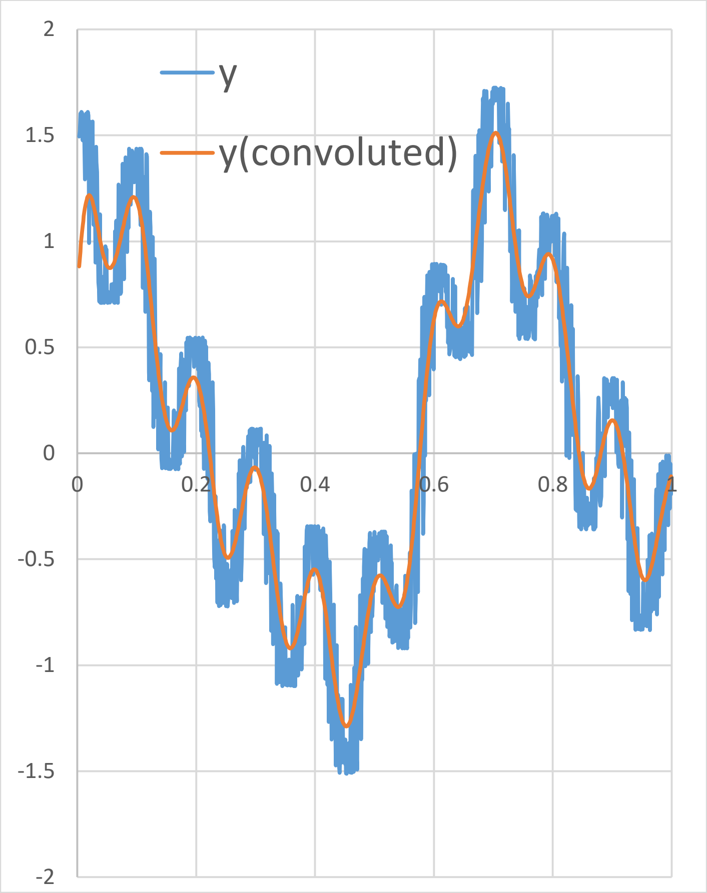
Application to sin / cos expansion

$f_i(x)=\cos 2\pi f_i x$ **(*i* = odd numbers (奇数))**

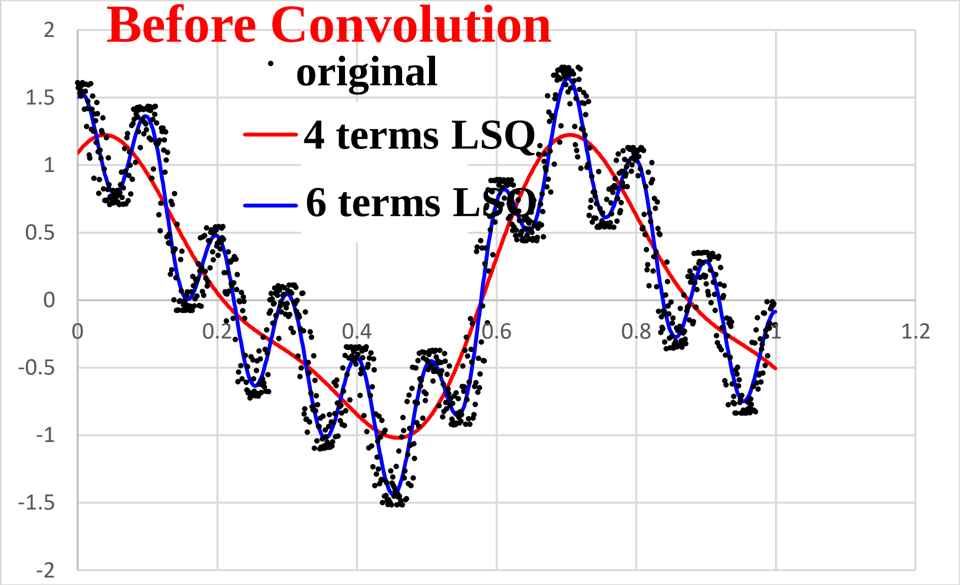
$f_i(x)=\sin 2\pi f_i x$ **(*i* = even numbers (偶数))**

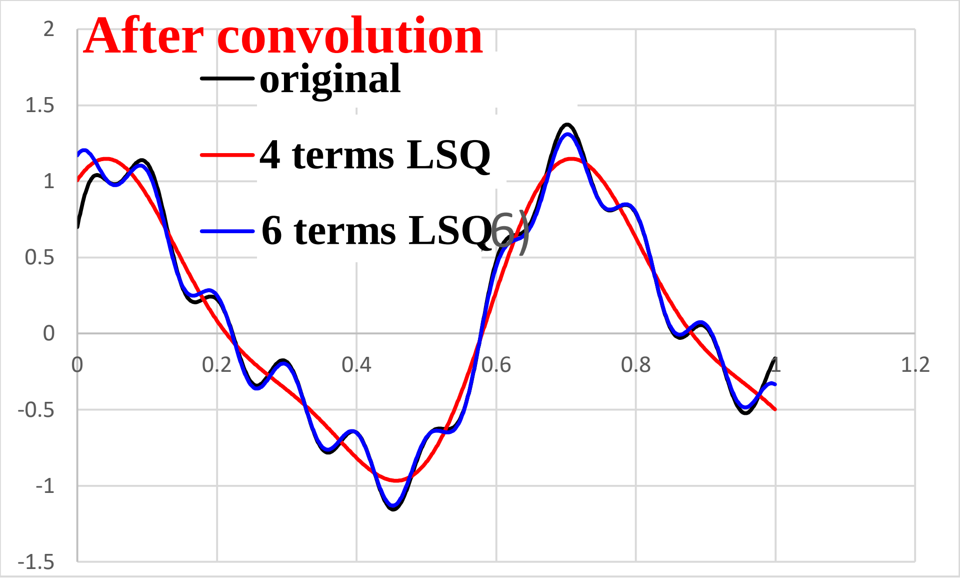
*l*LSQ for Fourier series expansion

```
f1, p1, A1 =  1.5, pi/4.0, 1.0
f2, p2, A2 =  3.0, pi/3.0, 0.3
f3, p3, A3 = 10.0, pi/6.0, 0.5
x += random(0.03) # noise is simulated by random()
y =  A1 * sin(2.0*pi * f1 * x + p1)
   + A2 * sin(2.0*pi * f2 * x + p2)
   + A3 * sin(2.0*pi * f3 * x + p3)
Convolution: Gauss function with w = 0.03
```



LSQ results


Before Convolution


After convolution

Discrete FT (DFT, 離散フーリエ変換)

Assume *x*(*t*) is periodic in the range [0, *T*^w^] and *x*(0) = *x*(*T*^w^)

$$X(f_k) = T_s^w \sum_{j=0}^{N-1} x(t_j) \exp(-i2\pi f_k \cdot jT^w/N) \qquad T_s^w = T^w/N$$

Usually the coefficient T_s^w is not included for DFT formulations

$$y(f_k) = \sum_{j=0}^{N-1} x(t_j) \, \mathbf{exp(-\boldsymbol{i2\pi kj/N})} \qquad f_k = k/T^w$$

DFT can be carried out without many trigonometric function (三角関数) calculations

$$y_k = \sum_{j=0}^{N-1} x_j \boldsymbol{w_N}^{\boldsymbol{kj}}$$

$$\boldsymbol{w_N} = \mathbf{exp}(-\boldsymbol{i2\pi/N})$$ **: Rotation factor (回転因子)**

$$\begin{aligned} {w_N}^{k+1} &= (\cos(-2\pi k/N) + i\sin(-2\pi k/N))(\cos(-2\pi/N) + i\sin(-2\pi/N)) \\ &= (\cos(-2\pi k/N)\, w_{N,r} - \sin(-2\pi k/N)\, w_{N,i}) \\ &\quad + i(\cos(-2\pi k/N)\, w_{N,i} + \sin(-2\pi k/N)\, w_{N,r}) \\ &= (w_{N,r}^k w_{N,r} - w_{N,i}^k w_{N,i}) + i(w_{N,r}^k w_{N,i} + w_{N,i}^k w_{N,r}) \end{aligned}$$

DFT: Matrix expression (行列表現)

$$y(f_k) = \sum_{j=0}^{N-1} x(t_j) \exp(-i2\pi \cdot k \cdot j/N)$$

$$\boldsymbol{y_k = \sum_{j=0}^{N-1} x_j w_N{}^{kj}} \qquad w_N = \exp(-i2\pi/N)$$

DFT

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & w_N{}^1 & w_N{}^2 & w_N{}^{N-1} \\ \vdots & w_N{}^2 & \ddots & \vdots \\ 1 & w_N{}^{N-1} & \cdots & w_N{}^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

Inverse DFT

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & w_N{}^{-1} & w_N{}^{-2} & w_N{}^{-(N-1)} \\ \vdots & w_N{}^{-2} & \ddots & \vdots \\ 1 & w_N{}^{-(N-1)} & \cdots & w_N{}^{-(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

Using $w_N{}^k = w_N{}^{k \bmod N}$ and $w_N{}^{k+N/2} = -w_N{}^k$, only k = 1 – $N/2$ terms should be calculated

Fast FT (FFT, 高速フーリエ変換)

金谷健一, これならわかる応用数学教室, 共立出版社 (2003)

1. The data number must be $N = 2^m$ (m: integer)
2. The identical calculation to DFT, but the calculation cost is proportional only to $N\log N$ (proportional to N^2 for DFT)
3. Simple circuits can implement FFT, easy for parallelization (GPU)

The DFT formulation is written as polynomial by converting $w_N{}^k = z$

$$y_k = \sum_{j=0}^{N-1} x_j w_N{}^{kj} = \sum_{j=0}^{N-1} x_j z^j$$

$$\begin{aligned} y_k &= x_0 z^0 + x_1 z^1 + x_2 z^2 + \cdots + x_{N-1} z^{N-1} \\ &= x_0 z^0 + x_2 z^2 + \cdots + x_{N-2} z^{N-2} \\ &\quad + z\left(x_1 z^0 + x_3 z^2 + \cdots + x_{N-1} z^{N-2}\right) \end{aligned}$$

Note that the last line equation becomes a polynomial with respect to $z_2 = z^2$ with a half number of the terms

$$y_k = \sum_{j=0}^{N/2-1} x_{2j} z_2{}^j + z \sum_{j=0}^{N/2-1} x_{2j+1} z_2{}^j$$

FFT

金谷健一, これならわかる応用数学教室, 共立出版社 (2003)

$$y_{k,N} = x_0(z^2)^0 + x_2(z^2)^1 + \cdots + x_{N-2}(z^2)^{\frac{N}{2}-1} + z\left(x_1(z^2)^0 + x_2(z^2)^1 + \cdots + x_{N-1}(z^2)^{\frac{N}{2}-1}\right)$$
$$= y_{k,N/2,1} + z y_{k,N/2,2}$$

$$y_{k,N/2,1} = x_0(z^4)^0 + x_4(z^4)^1 + \cdots + x_{N-2}(z^4)^{\frac{N}{4}-1} + (z^2)\left(x_2(z^4)^0 + x_6(z^4)^1 + \cdots + x_{N-3}(z^4)^{\frac{N}{4}-1}\right)$$
$$= y_{k,N/4,1} + (z^2) y_{k,N/4,3}$$

$$y_{k,N/2,2} = x_1(z^4)^0 + x_5(z^4)^1 + \cdots + x_{N-1}(z^4)^{\frac{N}{4}-1} + (z^2)\left(x_3(z^4)^0 + x_7(z^4)^1 + \cdots + x_{N-2}(z^4)^{\frac{N}{4}-1}\right)$$
$$= y_{k,N/4,2} + (z^2) y_{k,N/4,4}$$

$$y_{k,N} = y_{k,N/2,1} + z y_{k,N/2,2}$$
$$y_{k,N/2,1} = y_{k,N/4,1} + z^2 y_{k,N/4,3}$$
$$y_{k,N/2,2} = y_{k,N/4,2} + z^2 y_{k,N/4,4}$$
$$y_{k,N/4,1} = y_{k,N/8,1} + z^4 y_{k,N/8,5}$$
$$y_{k,N/4,2} = y_{k,N/8,2} + z^4 y_{k,N/8,6}$$
$$y_{k,N/4,3} = y_{k,N/8,5} + z^4 y_{k,N/8,7}$$
$$y_{k,N/4,4} = y_{k,N/8,6} + z^4 y_{k,N/8,8}$$

The above is a recursion formula and can be solved from the last two-terms FT to upper equations in the series of the number of terms $2^2, 2^3, \ldots, 2^N$

漸化式の形になっているので、最後の項数2のFTから順次 項数2^2, 2^3, …, 2^NのFTの計算をすることでFT計算ができる

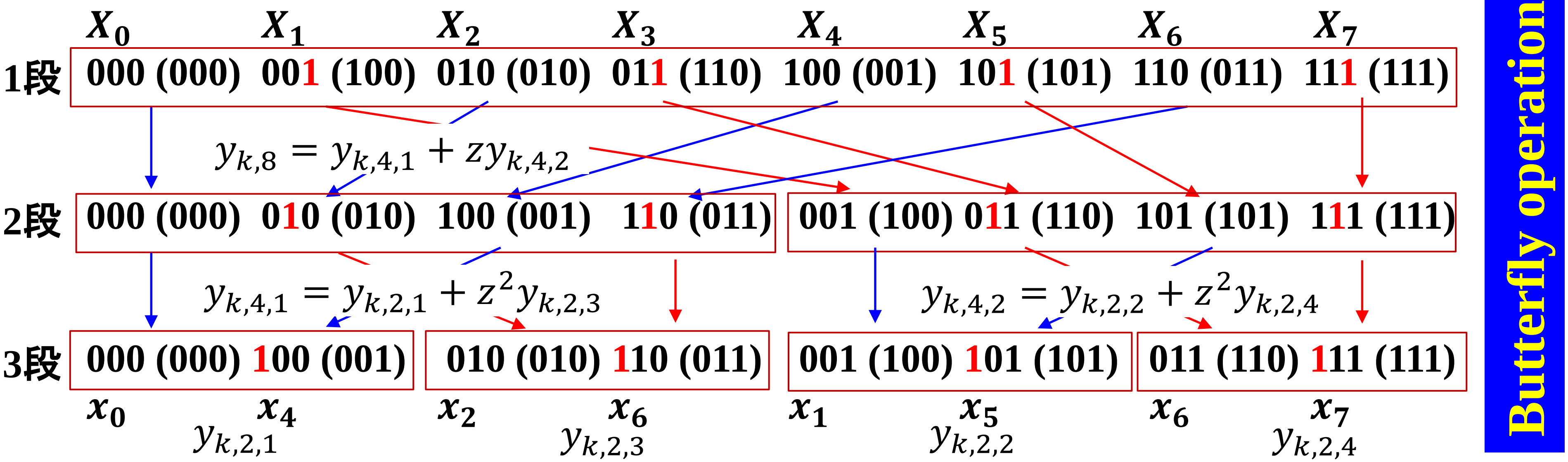
Data swap in the FFT procedure

N data series $x_0 x_1 x_2 \cdots x_{N-1}$ => FT: $X_0 X_1 X_2 \cdots X_{N-1}$
Represent the index number by binary (順序数を二進数であらわす)

At each stage k, the data are split to two, and the data of odd order are moved to the second half (*note the order is counted from 0*)

=> Data whose k-th bit is 1 are move to the second half
=> The change of the order numbers corresponds to bit reversal

Initial data order (values in parentheses are bitwise reversed)



X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7

1段 000 (000) 001 (100) 010 (010) 011 (110) 100 (001) 101 (101) 110 (011) 111 (111)

$y_{k,8} = y_{k,4,1} + z y_{k,4,2}$

2段 000 (000) 010 (010) 100 (001) 110 (011) | 001 (100) 011 (110) 101 (101) 111 (111)

$y_{k,4,1} = y_{k,2,1} + z^2 y_{k,2,3}$ $y_{k,4,2} = y_{k,2,2} + z^2 y_{k,2,4}$

3段 000 (000) 100 (001) | 010 (010) 110 (011) | 001 (100) 101 (101) | 011 (110) 111 (111)

x_0 x_4 $y_{k,2,1}$ x_2 x_6 $y_{k,2,3}$ x_1 x_5 $y_{k,2,2}$ x_6 x_7 $y_{k,2,4}$

Butterfly operation

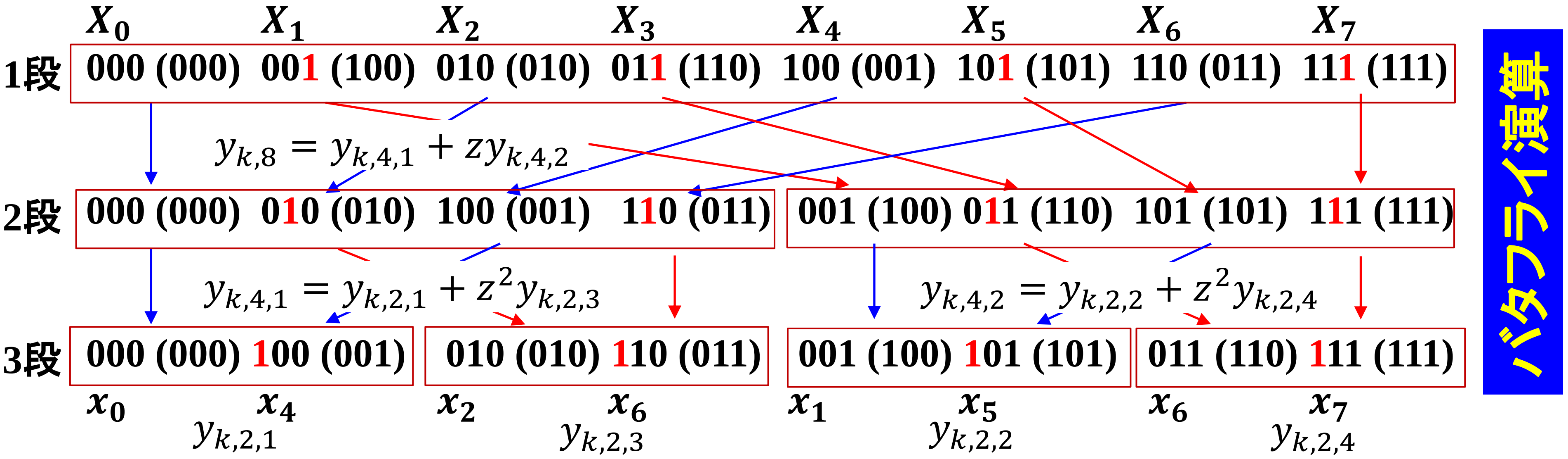
The order to sum up for FFT is different from the order of x_i.
FFT summation is performed in the order of the bit reversal of the index

FFT演算の項順序の変換: ビット反転

*N*個のデータ列 $x_0x_1x_2\cdots x_{N-1}$ => FT: $X_0X_1X_2\cdots X_{N-1}$
順序数を二進数であらわす

FFTのそれぞれの段階で「奇数番目のデータを後半にずらす」操作をする
=> 順序数の右から「段階数に対応するビットが 1 のデータを後半にずらす」
=> 順序数の変換がビット反転に対応する

最初のデータの並び順 (カッコ内は順序数のビット反転)



X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7

1段 000 (000) 001 (100) 010 (010) 011 (110) 100 (001) 101 (101) 110 (011) 111 (111)

$y_{k,8} = y_{k,4,1} + zy_{k,4,2}$

2段 000 (000) 010 (010) 100 (001) 110 (011) | 001 (100) 011 (110) 101 (101) 111 (111)

$y_{k,4,1} = y_{k,2,1} + z^2y_{k,2,3}$ $y_{k,4,2} = y_{k,2,2} + z^2y_{k,2,4}$

3段 000 (000) 100 (001) | 010 (010) 110 (011) | 001 (100) 101 (101) | 011 (110) 111 (111)

x_0 x_4 x_2 x_6 x_1 x_5 x_6 x_7

$y_{k,2,1}$ $y_{k,2,3}$ $y_{k,2,2}$ $y_{k,2,4}$

バタフライ演算

FFTの和を取る順番は x_i の並び順と変わる。
最初の順序数の二進数表現 (カッコ内の数字) をビット反転 (カッコ外の数字) してソートすると、その順序でFFTの和をとれる

Logical operations (bitwise operations)
(論理演算, ビット演算)

Logical NOT (Bitwise inversion) (論理否定, ビット反転)

NOT 0 = 1; NOT 1 = 0

python: ~x, not x *~1 == 0, ~0 == 1*

Logical AND (論理積)

0 AND 0 = 0; 1 AND 0 = 0
0 AND 1 = 0; 1 AND 1 = 1

python: x & y, x and y *1 & 1 == 1*

Logical OR (論理和)

0 OR 0 = 0; 1 OR 0 = 1
0 OR 1 = 1; 1 OR 1 = 1

python: x | y, x or y

Logical Exclusive OR (排他的論理和)

0 XOR 0 = 0; 1 XOR 0 = 1
0 XOR 1 = 1; 1 XOR 1 = 0

python: x ^ y, x xor y

Bit shift (*n* bit shift)

python: a << n, a >> n *0b0001 << 2 == 0b0100*
0b0110 >> 1 == 0b0011

Bit reversal (ビット列反転)

Note: **bit reversal (ビット列反転) != bitwise inversion (ビット反転) (~x, not x)**

bit_reverse.py

```
def bit_reverse(val):
    ret = 0
    while 1:
        v0 = val & 0b001
        ret = ret | v0
        val = val >> 1

        if val == 0:
            break
        else:
            ret = ret << 1

    return ret
```

val = 11001_2 を例に

ret = 0 # ビット反転値を0で初期化

1. **v0** = val & 1_2 => 11001_2 & 001_2 **= 1** # 第1桁のビット値を v0 に保存

2. **ret** = ret | v0 => 0 | 1 **= 1_2** # retの第一桁に v0 を設定

3. **val** = val >>1 => 11001_2 >> 1 **= 1100_2**

# 一桁右にビットシフトし、valの2桁目を第1桁に移動

4. val が 0 の場合、処理するbitが残っていないので

ループを終了

5. val が 0でない場合、ret を1ビットシフトし、2.で ret の第1位に設定した v0 を左にずらす。

ret = ret << 1 => 1_2 << 1 **= 10_2**

1.に戻って繰り返し

6. **v0** = val & 1_2 => 1100_2 & 001_2 = 0 # 第1桁のビット値を v0 に保存

7. **ret** = ret | v0 => 10_2 | 1 **=** 10_2 # retの第一桁に v0 を設定

8. **val** = val >>1 => 1100_2 >> 1 **= 110_2**

9. **ret** = ret << 1 => 10_2 << 1 **= 100_2**

1.に戻って繰り返し

10. **v0** = val & 1_2 => 110_2 & 001_2 = **0**

11. **ret** = ret | v0 => 100_2 | 1 = 100_2

12. **val** = val >>1 => 110_2 >> 1 **= 11_2**

13. **ret** = ret << 1 => 100_2 << 1 **= 1000_2**

1.に戻って繰り返し

14. **v0** = val & 1_2 => 11_2 & 1_2 = **1**

15. **ret** = ret | v0 => 1000_2 | 1 **=** 1001_2

16. **val** = val >>1 => 11_2 >> 1 **= 1_2**

17. **ret** = ret << 1 => 1001_2 << 1 **= 10010_2**

1.に戻って繰り返し

18. **v0** = val & 1_2 => 1_2 & 1_2 = **1**

19. **ret** = ret | v0 => 10010_2 | 1 **=** 10011_2

20. **val** = val >>1 => 1_2 >> 1 **= 0_2** => **ループ終了 解: ret = 10011_2**

Bitwise operation can be replaced with other op

Usually bitwise operations are faster, but it is not the case for python …

```
python bit_reverse_compare.py 1001100011110101101111011 1000000
```

measure time to reverse $1001100011110101101111011_2$ for 1000000 times

```
by bitwise operation       : 6.265091180801392 s
without bitwise operation: 4.462110280990601 s
```

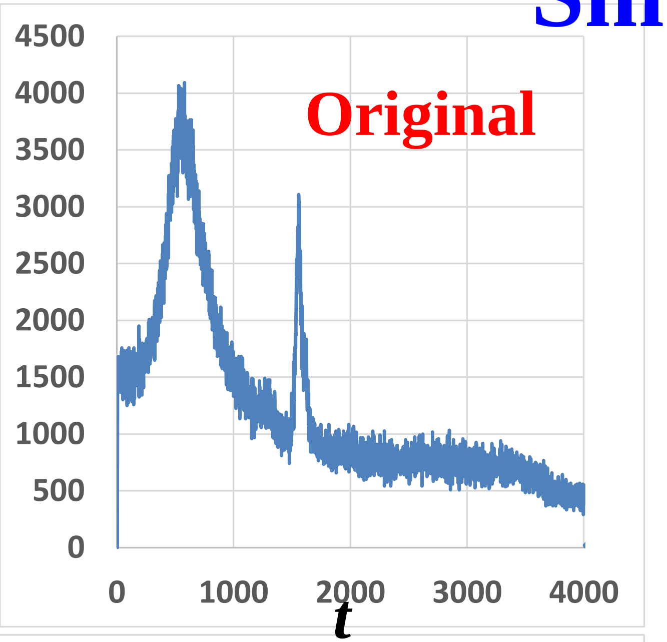
bit_reverse_compare.py

```
def bit_reverse(val):
    ret = 0
    while 1:
        v0 = val & 0b001
        ret = ret | v0
        val = val >> 1

        if val == 0:
            break
        else:
            ret = ret << 1

    return ret
```

bit_reverse_compare.py

```
def bit_reverse_nobitop(val):
    ret = 0
    while 1:
        v0 = val % 2         # save the final bit to v0
        ret = ret + v0       # put v0 to the final bit of ret
        val = val // 2       # bit shift for next iteration

        if val == 0:
            break
        else:
            ret = ret * 2    # bit shift ret to left

    return ret
```

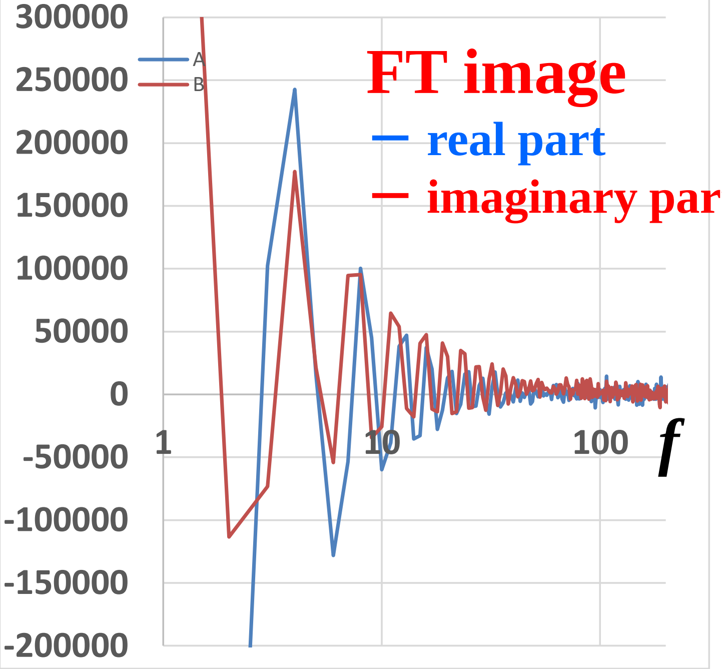
Smoothing: FT

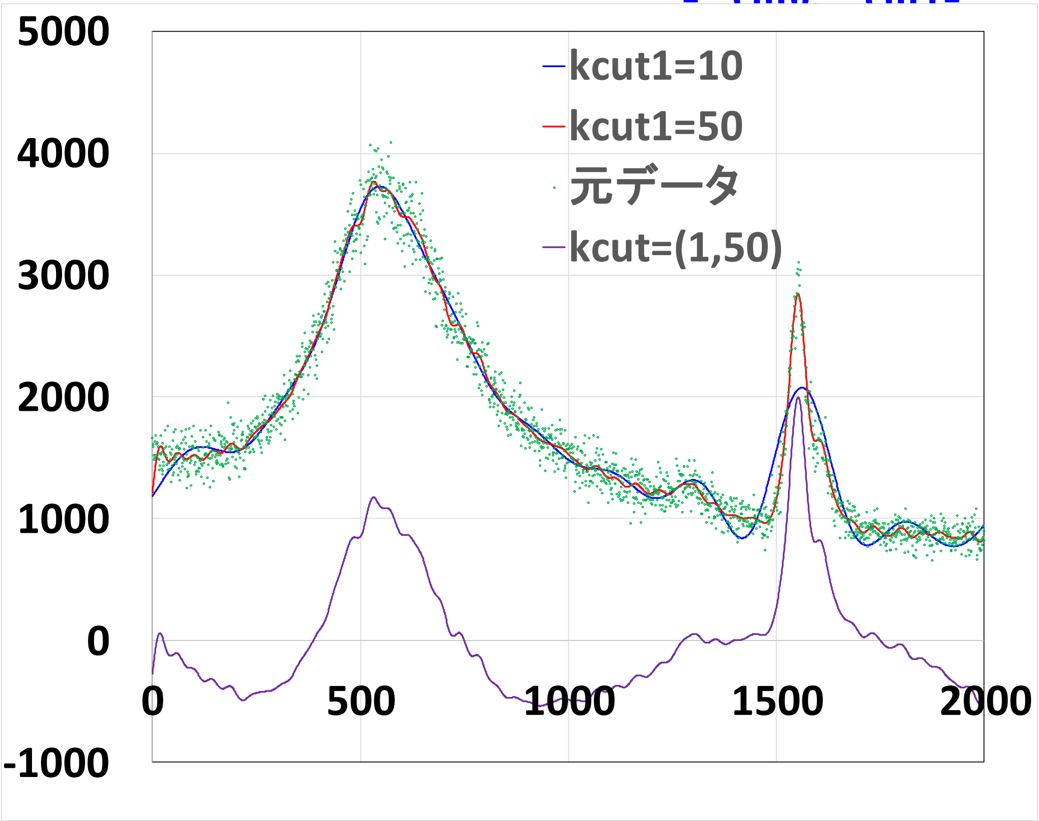


Remove high-frequency FT data: Smoothing
Low-pass filter

Remove low-frequency FT data: Cut drift
High-pass filter

***Ex.* Cut FT data outside $[k_{cut0}\ k_{cut1}]$**



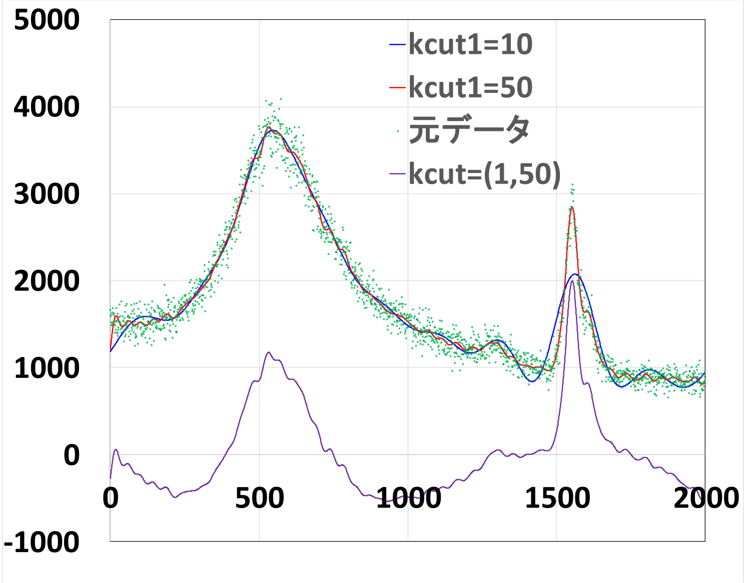


Note:

Be careful: FFT high-pass filter can remove a baseline, but that baseline includes some signals

Usual ways:

1. Baseline function is optimized simultaneously with peaks.
3. Baseline function is determined from selected data where peaks do not affect.

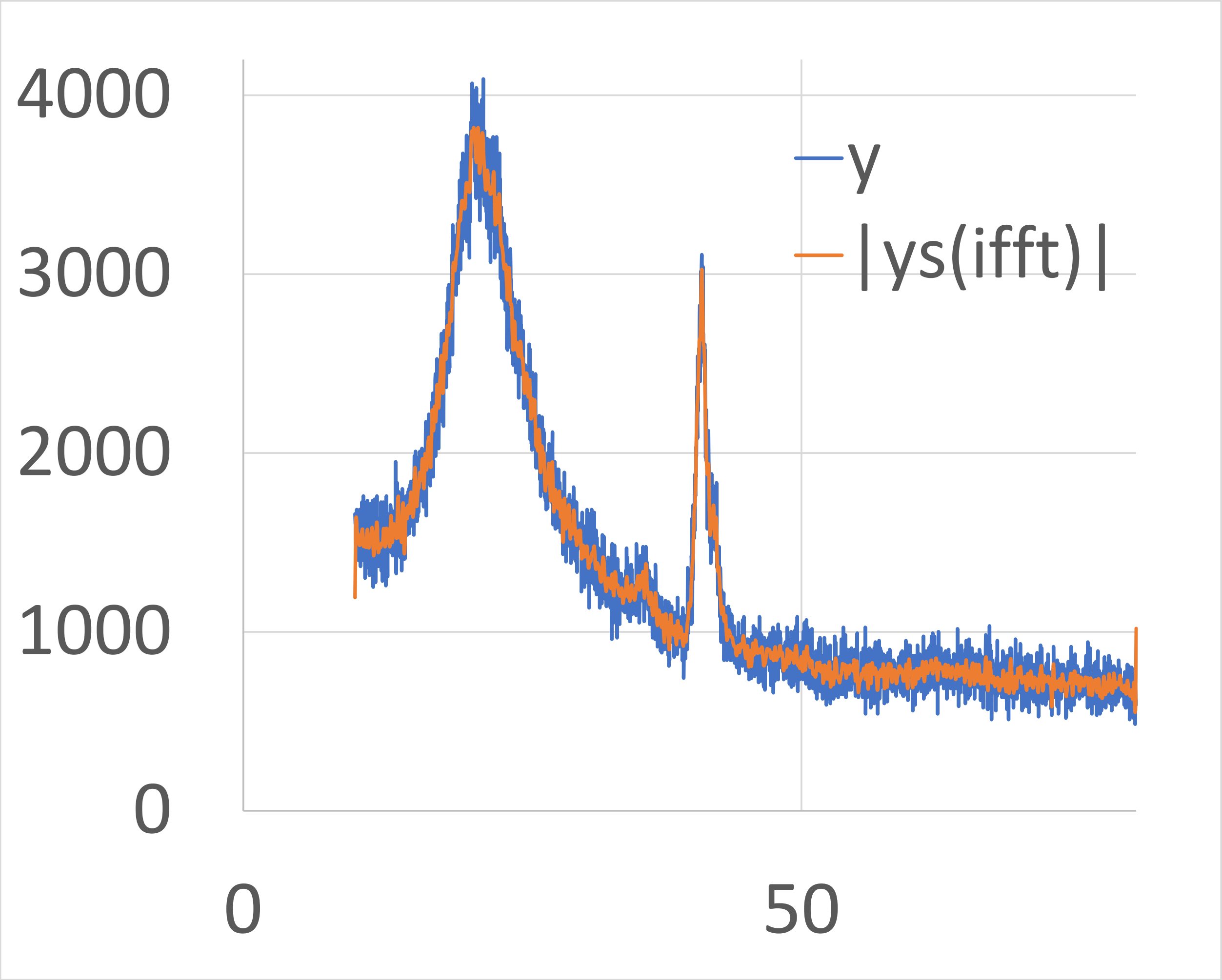


Program: smoothing-fft.py

Usage: python smoothing-fft.py xrd.csv 0 5

(note: the x range is different from the previous slide)

=> plot smoothing-fft.csv

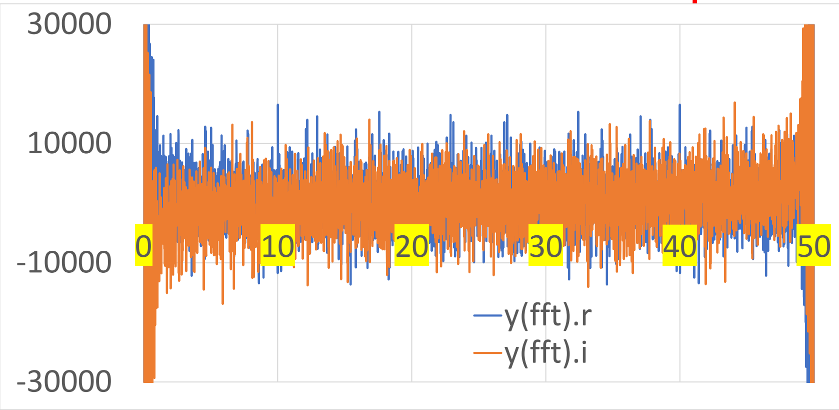


Note for FFT

smoothing-fft.py

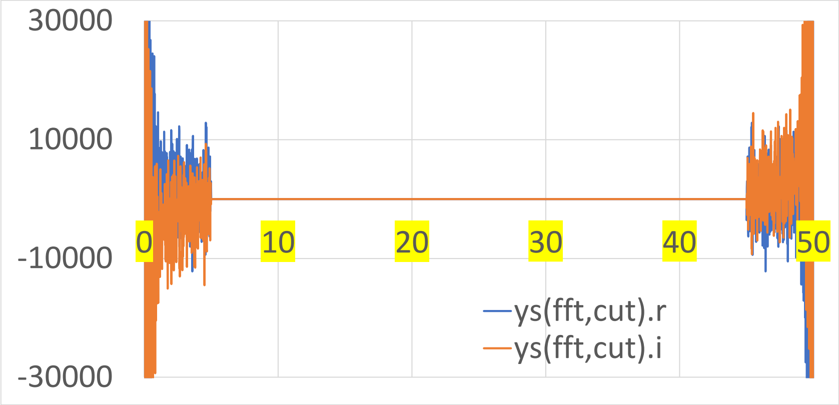
Numpy fft module: F = np.fft.fft(y)   FFT

FFTed result is symmetric at the center of the reciprocal x axis at $i_x = n_x/2$.



For smoothing, cut the data in $i_x = [0, i_{xHF}], [i_{xHF}, n_x/2]$

and $[n_x-1, n_x-1-i_{xHF}], n_x/2+1, n_x-1-i_{xHF}\ i_{xHF}]$.



, then perform IFFT by fs = np.fft.ifft(Fs)

Comparison: Calculation time by python

```
Usage: python dft.py ndata
ex: python dft.py 1024
    python dft.py 2048
```

DFT1: DFT using rotation factor
DFT2: DFT not using rotation factor (calculate sin/cos every time)
FFT : numpy.fft.fft()

Time for DFT/FFT (sec)

N	DFT1	DFT2	FFT	N log N
1024	1.32	2.41	1.87e-5	3080
2048	5.59	10.3	3.54e-5	6780
4096	23.6	47.7	6.62e-5	14800
8192	97.3	165	16.1e-5	32100

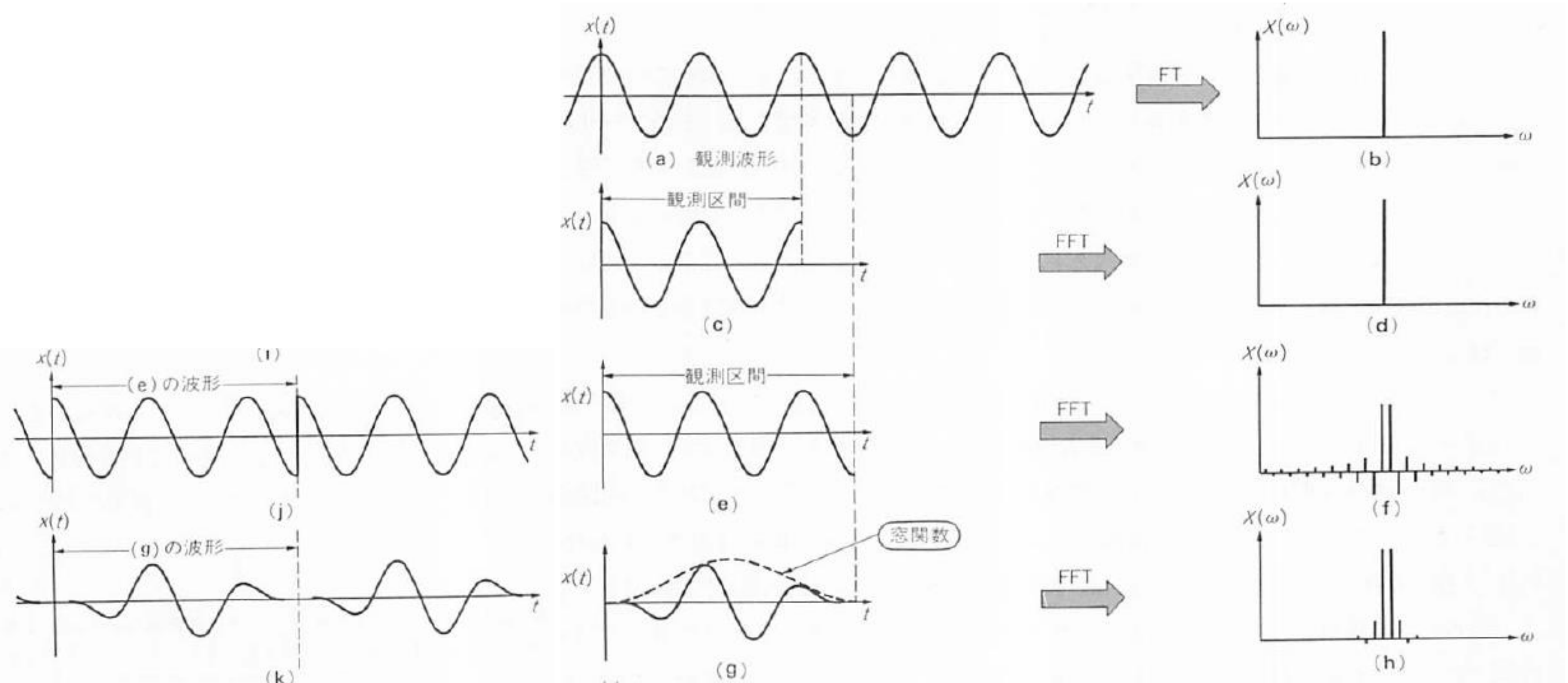
Problems of DFT/FFT

南茂夫, 科学計測のための波形データ処理, CQ出版社 (1986)

- Usually FT needs integration from -∞ to ∞, but DFT/FFT reduces data to finite range => Loss of data

  ex.: Fourier charge analysis by XRD gives ghost peaks and fringes
- Original data include noise/errors, giving rise to extra frequency peak
- Artificial periodicity required for DFT/FFT gives rise to artefact frequency peaks => can be suppressed by Hanning Window (窓関数),

  but it may also give extra peaks



Maximum entropy method (MEM, 最大エントロピー法)

南茂夫, 科学計測のための波形データ処理, CQ出版社 (1986)

Concept of MEM

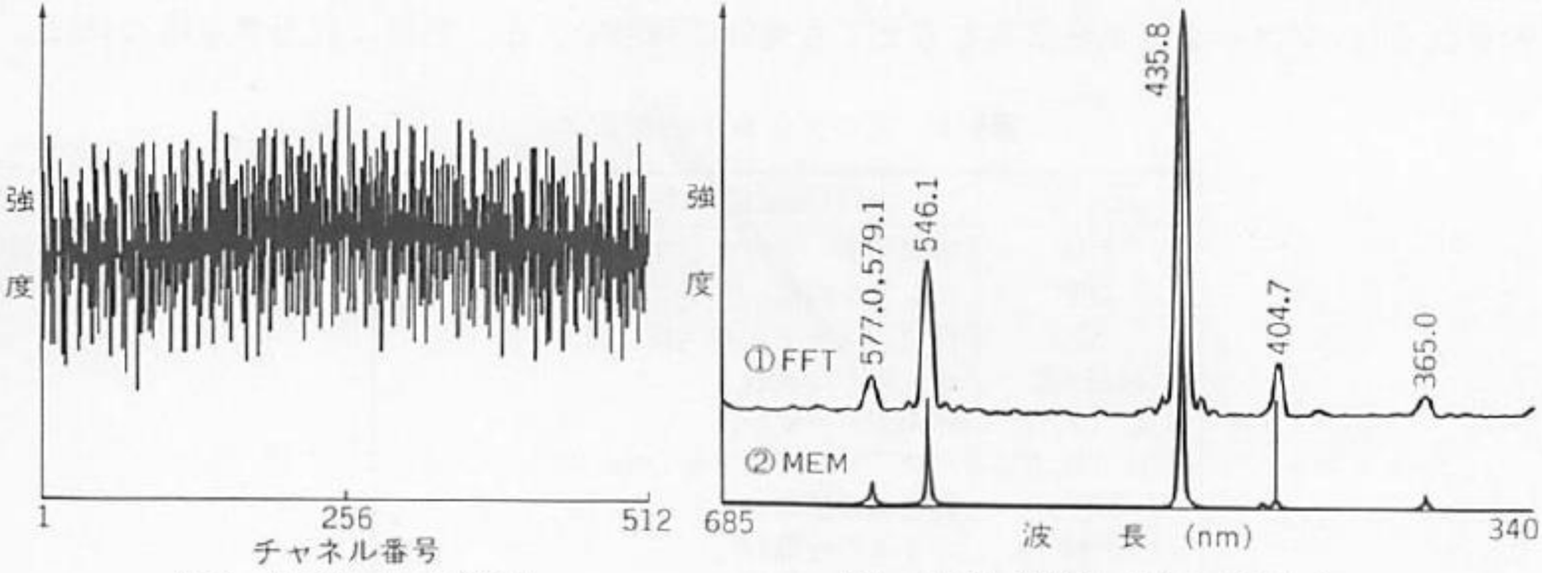
- Assume the lost data would have some constraints
- Use the concept of ‘information entropy’ and maximize it to estimate the spectrum
- Akaike’s autoregressive model (赤池による自己回帰モデル) => identical to MEM

  The order of the autoregressive model *m* must be determined

  - So as to minimize Final Prediction Error (最終予測誤差)
- Algorisms: Burg method, etc

Features

- Sharper spectrum than FFT
- Less ghost peaks and fringes



(a) インタフェログラム　(b) FFTとMEMによるスペクトル

MEM-Rietveld analysis

坂田誠 日本結晶学会誌 **30**, 135 (1988)

Charge density calculated from structure factors $\tau'_i = \tau_i / \sum \tau_i$

Charge density calculated from structure model $\rho'_i = \rho_i / \sum \rho_i$

Constrained entropy: $S = -\sum \rho'_i \ln \frac{\rho'_i}{\tau'_i}$

=> smoothing ρ' and suppress fringes and ghost peaks

Minimize the structure factor residual $C = \sum \frac{\left|F_{\mathrm{cal}}{}^{hkl} - F_{\mathrm{obs}}{}^{hkl}\right|^2}{\sigma_{hkl}{}^2}$

Maximize constrained entropy $Q(\lambda) = -\sum \rho'_i \ln \frac{\rho'_i}{\tau'_i} - \frac{\lambda}{2} \sum \frac{\left|F_{\mathrm{cal}}{}^{hkl} - F_{\mathrm{obs}}{}^{hkl}\right|^2}{\sigma_{hkl}{}^2}$

=> ρ ＝ exp(ln τ + difference Fourier (差フーリエ) **term)**
When converged to $F_{\mathrm{cal}} = F_{\mathrm{obs}}$, ρ = τ will be achieved

Kramers-Kronig Transformation

KK transformation: Numerical approach

$$\theta(\nu_g) = -\frac{2\nu_g}{\pi}\int_0^\infty \frac{\ln\sqrt{R(\nu)/R(\nu_g)}}{\nu^2-\nu_g{}^2}d\nu$$

$$\theta_{med}(\nu_g) = \frac{4\nu_g}{\pi}\Delta\nu \sum_{i=odd\ or\ even}^{i<=nData} \frac{\ln\sqrt{R_i}}{\nu_i{}^2-\nu_g{}^2}$$

Numerical integration for given data

$$\theta_{high}(\nu_g) = -\frac{\ln\sqrt{R_{high}}}{\pi}\ln\frac{\nu_{\max}+\nu_g}{\nu_{\max}-\nu_g}$$

Extrapolation to high *f*

$$\theta_{low}(\nu_g) = -\frac{\ln\sqrt{R_{low}}}{\pi}\ln\frac{\nu_g-\nu_{\min}}{\nu_{\min}+\nu_g}$$

Extrapolation to low *f*

$$n(\nu) = \frac{1-R(\nu)}{1+R(\nu)-2\sqrt{R(\nu)}\cos\theta}$$

$$k(\nu) = \frac{2\sqrt{R(\nu)}\sin\theta}{1+R(\nu)-2\sqrt{R(\nu)}\cos\theta}$$

光学スペクトルのKK変換: 外挿問題

Fourier変換と同様に、測定周波数外の情報がないことが問題
測定周波数外のデータは結果に大きく影響する

0.1eVにおける分散を正確に求める場合、
少なくとも4~5eVまでの測定が必要

- 高エネルギー領域: ν^{-4}で外挿するのが一般的
- 知りたい領域が0.1eV以下:
  - 金属の低エネルギー領域: Drude反射率で外挿
  - 半導体・誘電体の低エネルギー領域:
    静的誘電率ε_0から求めた反射率 で外挿 $\left(\sqrt{\varepsilon_0}-1\right)^2/\left(\sqrt{\varepsilon_0}+1\right)^2$

- 知りたい領域が1eV程度まで:
  フォノン分散の始まる0.1eV以下は考慮する必要はない

- 半導体・誘電体の低エネルギー領域:
  高周波誘電率ε_∞から求めた反射率 で外挿 $\left(\sqrt{\varepsilon_\infty}-1\right)^2/\left(\sqrt{\varepsilon_\infty}+1\right)^2$

クラマース－クローニッヒ (KK) の関係式

因果律が成立していれば
(現在の状態が、過去の履歴の蓄積で決定していれば)、
線形応答の範囲で、周波数応答関数 ε(ω) の実部と虚部には
以下のKramers-Kronighの関係が成立する

$$\varepsilon_r = 1 + \frac{1}{\pi} P \int_{-\infty}^{\infty} \frac{\omega' \varepsilon_i(\omega')}{\omega'^2 - \omega^2} d\omega' \left(= 1 + \frac{2}{\pi} P \int_{0}^{\infty} \frac{\omega' \varepsilon_i(\omega')}{\omega'^2 - \omega^2} d\omega' \right)$$

$$\varepsilon_i = -\frac{1}{\pi} P \int_{-\infty}^{\infty} \frac{\varepsilon_r(\omega') - 1}{\omega'^2 - \omega^2} d\omega' \left(= -\frac{2}{\pi} P \int_{0}^{\infty} \frac{\varepsilon_r(\omega') - 1}{\omega'^2 - \omega^2} d\omega' \right)$$

インパルス応答が実数の
場合、カッコ内の式が使える

P: 積分の主値 $P \int_{0}^{\infty} d\omega' \equiv \lim_{\delta \to 0} \left(\int_{0}^{\omega - \delta} d\omega' + \int_{0}^{\omega + \delta} d\omega' \right)$

注意: 主値積分は極限の取り方によって値が変わる
=> 必ず ωの両側から同じように極限を取る

クラマースークローニッヒの関係式

Kramers-Kronig relation

$$\varepsilon_r = 1 + \frac{2}{\pi} P \int_0^\infty \frac{\omega' \varepsilon_i(\omega')}{\omega'^2 - \omega^2} d\omega'$$

$$\varepsilon_i = -\frac{2}{\pi} P \int_0^\infty \frac{\varepsilon_r(\omega') - 1}{\omega'^2 - \omega^2} d\omega'$$

P: Principal value of the integral

$$P \int_0^\infty d\omega' \equiv \lim_{\delta \to 0} \left(\int_0^{\omega - \delta} d\omega' + \int_0^{\omega + \delta} d\omega' \right)$$

The above equation is derived from Cauchy integral $\alpha(\omega) = \frac{1}{\pi i} P \int_0^\infty \frac{\alpha(s)}{s - \omega} ds$ that is valid for complex functions $\alpha(\omega)$ satisfying $\lim_{|\omega| \to \infty} \alpha(\omega) = 0$

KK relation: Refrectivity spectrum and phase

反射率と位相

$$r^*(\nu)=\sqrt{R(\nu)}e^{i\theta(\nu)} \qquad \ln r^*(\nu)=\ln R^{1/2}+i\theta(\nu)$$

にKK変換を当てはめる

$$\theta(\nu)=-\frac{1}{2\pi}P\int_0^\infty\frac{\ln R(\nu')}{\nu'^2-\nu^2}d\nu'=-\frac{1}{2\pi}\int_0^\infty\ln\frac{|\nu'+\nu|}{|\nu'-\nu|}\frac{dR(\nu')}{d\nu'}d\nu'$$

Optical spectrum (誘電関数ε*, 吸収係数α)

$$\mathcal{H} = \mathcal{H}_{\mathbf{0}} - \boldsymbol{e}\boldsymbol{r} \cdot \boldsymbol{E}$$

$$\varepsilon_1(\omega) = 1 + 4\pi \sum_j \frac{e^2 \left|T_{0j}\right|^2}{\hbar} \frac{2\omega_j}{\omega_j{}^2 - \omega^2}$$

$$T_{ij} = \langle \Psi_i | \mathbf{r} | \Psi_j \rangle = \int \Psi_i{}^* \mathbf{r} \Psi_j d\mathbf{r}$$

Kramers-Kronig transformation

$$\varepsilon_2(\omega) = \frac{4\pi N e^2}{m} \sum_j f_j \pi \delta\left(\omega^2 - \omega_j{}^2\right)$$

$$= \frac{4\pi N e^2}{m} \sum_j f_j \frac{\pi}{2\omega} \left[\delta\left(\omega - \omega_j\right) + \delta\left(\omega + \omega_j\right)\right]$$

$$n(\omega) - i\kappa(\omega) = \sqrt{\varepsilon_1(\omega) - i\varepsilon_1(\omega)}$$

$$\alpha(\omega) = \frac{4\pi}{\lambda} \kappa(\omega)$$

Monte Carlo method
for numerical integration

Q: Monte Carlo simulation for materials

Monte Carlo simulations:

- **Based on random number**
  How to generate random numbers in computer?
- **Application to multi-dimensional integration**
  Hit-and-miss Monte Carlo method
  Crude Monte Carlo method
- **Application to materials simulation**
  Metropolis Monte Carlo simulation

A: Monte Carlo simulation for integration

Q: 多変数の積分で、各積分範囲がお互いに別な変数を含んでしまう場合、どのように解けるのか知りたい。
Q: python scipyに付属する関数で複雑な三重積分を計算しましたが、様々なエラーによりうまくいきませんでした

A: Better to use python numpy/scipy libralies in particular for multiple integral because those are generally time-consuming calculations.
However, scipy.integrate.tplquad() is slow because it guarantees the obtained accuracy given by the argument epsrel (and .quad() for 1-D integration as well).
Therefore, the following Rieman sum may be faster that tplquad()

$$\int\int f(x,y)dxdy \sim h_x h_y \sum_i \sum_j f(x_i, y_j)$$

by Rieman sum for $x_i = x_0 + ih_x$ and $y_i = y_0 + ih_y$

If you don’t need a high accuracy, Monte Calro method can be easily applied.
For infinite integrals, apply double exponential conversion algorisms

一様乱数と疑似乱数

▪コンピュータで “ランダム” な事象は発生させにくい

＝＞ アルゴリズムによって疑似乱数を発生する

▪**乗積合同法**: a, b, L を正数とし、

$N_1 = a$

$N_2 = bN_1 \bmod L$　　　　$N \bmod L$ は N を L で除した余り

$N_3 = bN_2 \bmod L$

▪▪▪

とすると、Nは $0 \leq N \leq \mathrm{L}-1$ の疑似乱数になる。

▪**混合合同法**: a, b, L を正数とし、

$N_1 = a$

$N_2 = bN_1 + c \bmod L$

$N_3 = bN_2 + c \bmod L$

▪▪▪

＊$N_k = N_m$ となると、乱数に周期性が発生する

疑似乱数の検定

良い疑似乱数 (一様乱数に近い) の条件

- 分布が均一
- 周期性がない
- 標準偏差が $N^{1/2}$ に比例して増大

- **疑似乱数の発生の多くには “種 (seed)” が必要**

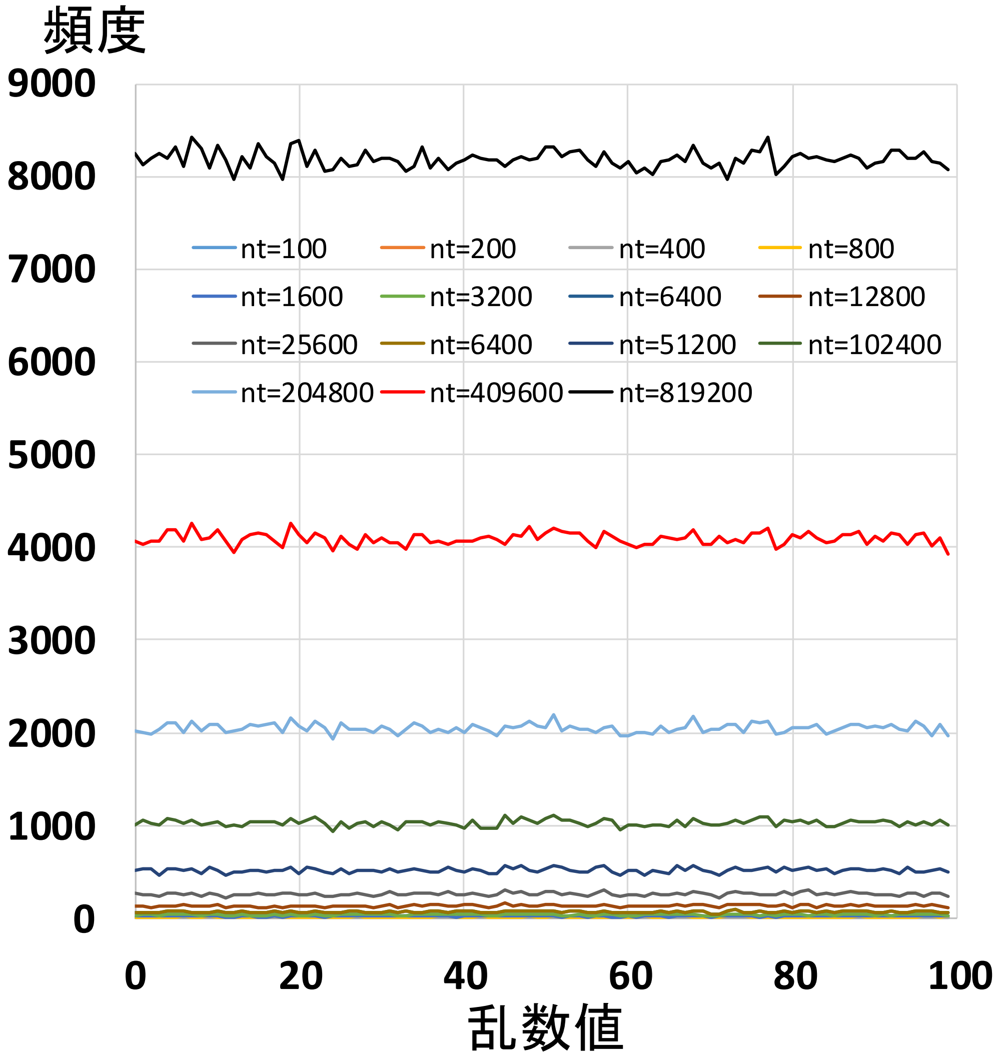
  seedが同じなら乱数も同じになる。
  毎回seedを変える必要がある
  (時計、乱数発生器など)

- **計算ごとに乱数が変わると困ることもある**

  (デバッグ、計算結果の比較など)
  seedを同じにして計算

Perlの例 =>

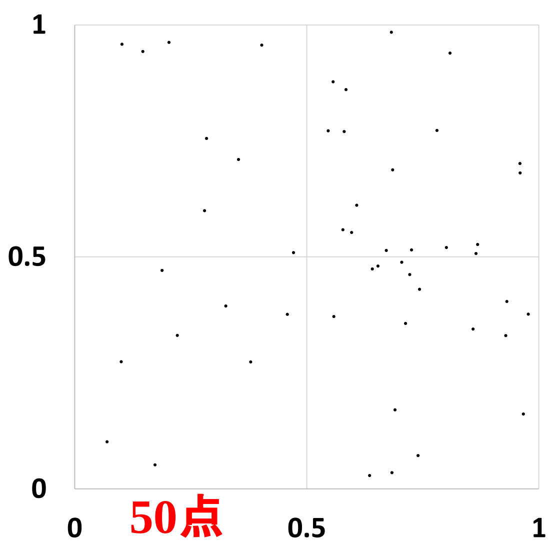
```
srand(0);
my @r;
for(my $i = 1 ; $i <= $n ; $i-
{
    $r[int(rand(100))]++;
}
```



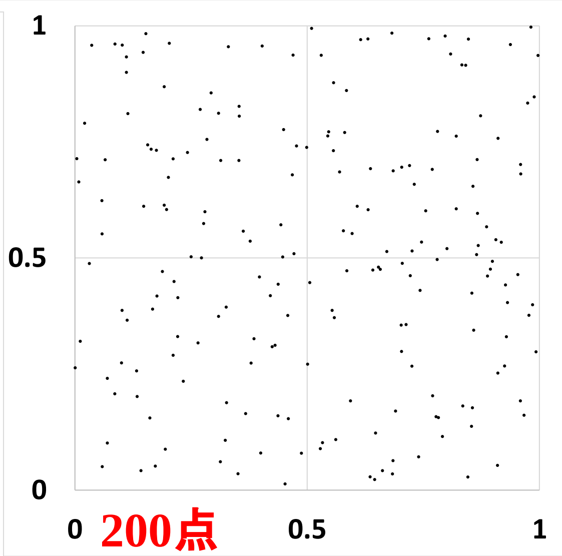
Perlのrand関数

seedの指定 (srand()) は不要

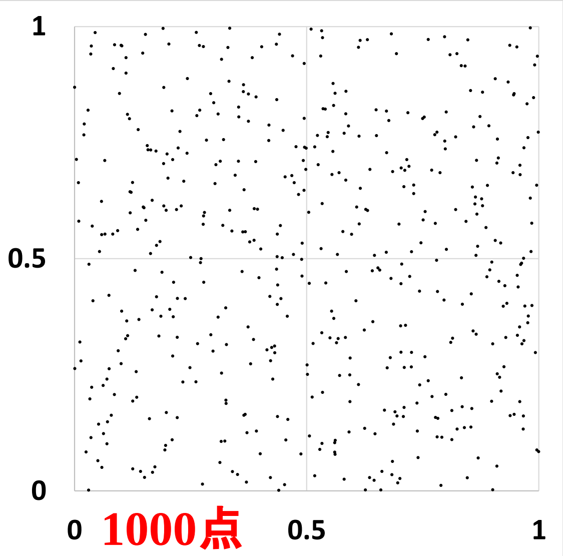
```
for(my $i = 0 ; $i < $nMax ; $i++) {
            my $x = rand(1.0);
            my $y = rand(1.0);
}
```



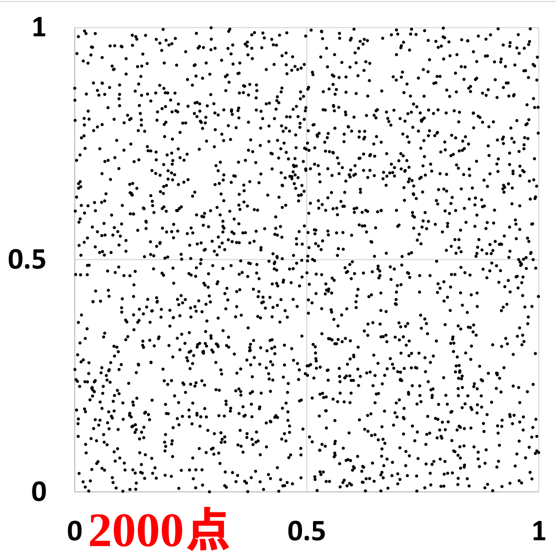
50点



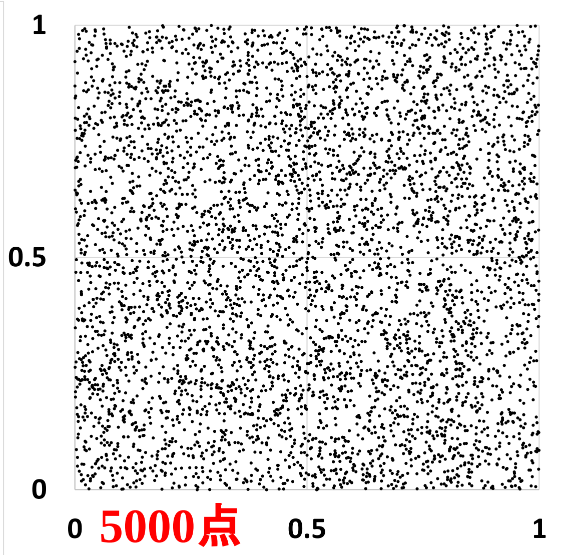
200点



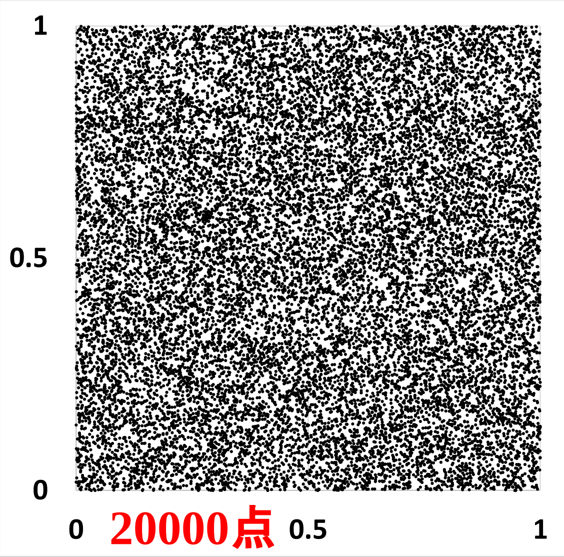
1000点



2000点



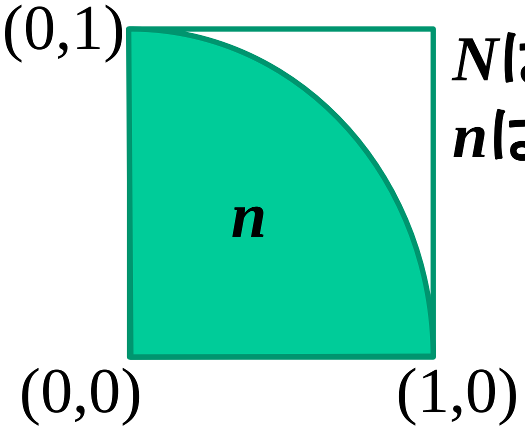
5000点



20000点

試行錯誤的 (hit-or-miss) Monte Carlo法

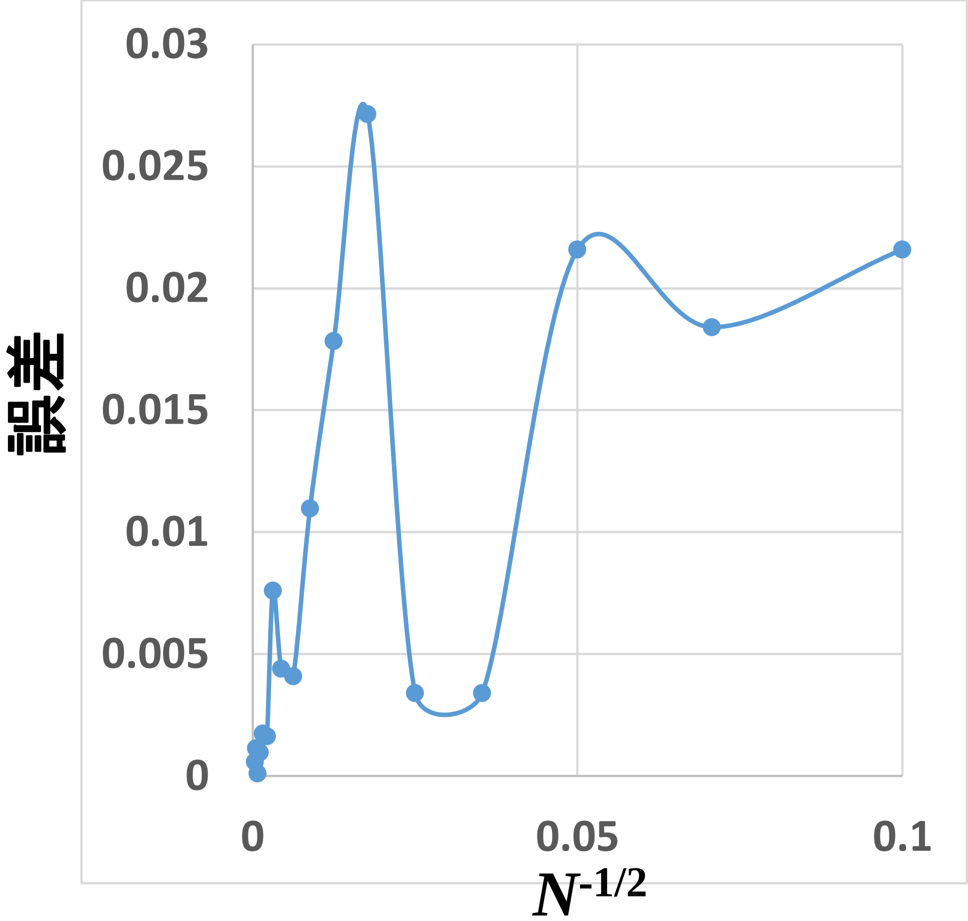
$0 \le r < 1$の疑似乱数 (x, y) を N回発生し、$(x^2 + y^2)^{1/2} < 1.0$ となる回数 n を求めると、n/N は 四分の一円の面積の近似値となる



***N*は正方形全体に分布**

***n*は1/4円の中に落ちた数**

N	$N^{-1/2}$	4S	|error|
100	0.1	3.12	0.021593
200	0.070711	3.16	0.018407
400	0.05	3.12	0.021593
800	0.035355	3.145	0.003407
1600	0.025	3.145	0.003407
3200	0.017678	3.16875	0.027157
6400	0.0125	3.12375	0.017843
12800	0.008839	3.130625	0.010968
25600	0.00625	3.1375	0.004093
51200	0.004419	3.137188	0.004405
102400	0.003125	3.133984	0.007608
204800	0.00221	3.139961	0.001632
409600	0.001563	3.139854	0.001739
819200	0.001105	3.14063	0.000963
1638400	0.000781	3.141702	0.000109
3276800	0.000552	3.14045	0.001142
6553600	0.000391	3.141	0.000593



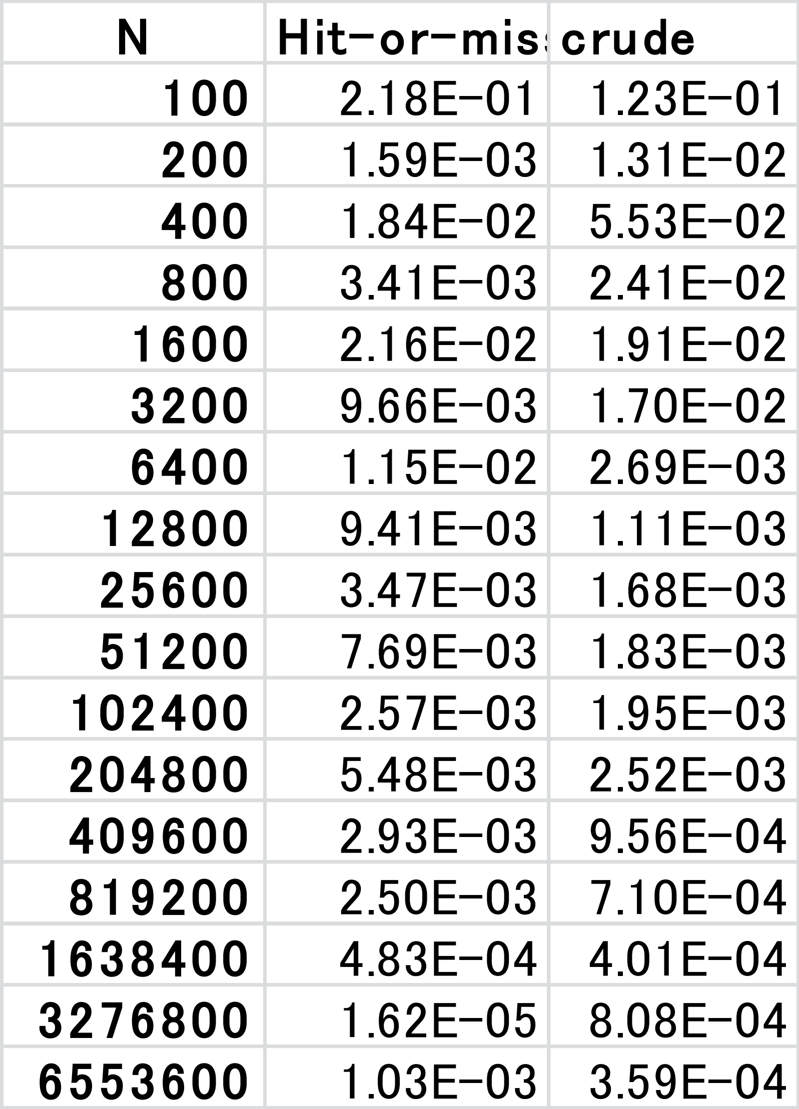
基礎的 (crude) Monte Carlo法

$0 \leq r < 1$の疑似乱数 (x) を N回発生し、

$$S = \int_0^1 f(x)dx \sim \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

で近似できる。

$$f(x) = 4\sqrt{1-x^2}$$



N	Hit−or−mis	crude
100	2.18E−01	1.23E−01
200	1.59E−03	1.31E−02
400	1.84E−02	5.53E−02
800	3.41E−03	2.41E−02
1600	2.16E−02	1.91E−02
3200	9.66E−03	1.70E−02
6400	1.15E−02	2.69E−03
12800	9.41E−03	1.11E−03
25600	3.47E−03	1.68E−03
51200	7.69E−03	1.83E−03
102400	2.57E−03	1.95E−03
204800	5.48E−03	2.52E−03
409600	2.93E−03	9.56E−04
819200	2.50E−03	7.10E−04
1638400	4.83E−04	4.01E−04
3276800	1.62E−05	8.08E−04
6553600	1.03E−03	3.59E−04

乱数を用いた数値積分

多次元積分で用いられる

例: Discrete Variational Xα法

3次元積分点を疑似乱数で発生

integ_montecarlo3d.py

Calculate the volume of radius 1.0 sphere

Python integ_montecarlo3d.py

Output:

```
Hit-or-miss Monte-Carlo method
i               V                               |error|
100        4.3200000000        0.13120979521360976
200        4.2000000000        0.011209795213609652
400        4.0200000000        0.16879020478639095
800        4.2000000000        0.011209795213609652
1600        4.2450000000        0.05620979521360958
3200        4.2025000000        0.013709795213609155
6400        4.1537500000        0.035040204786390916
12800        4.1868750000        0.001915204786390845
25600        4.1618750000        0.026915204786390312
51200        4.1620312500        0.026758954786390454
102400        4.1902343750        0.0014441702136096524
204800        4.1915625000        0.0027722952136093326
409600        4.1894921875        0.0007019827136094392
819200        4.1852148437        0.0035753610363906674
1638400     4.1913476562        0.002557451463609084
3276800     4.1906274414        0.0018372366198597945
6553600     4.1887829590        7.245802015276581e-06
```

Error ∝ 1/N

指数分布に従う乱数

http://www.sat.t.u-tokyo.ac.jp/~omi/random_variables_generation.html#Gauss

$$p(x;\lambda)=\lambda\exp\left(-\lambda x\right)$$ **(平均 1/λ, 分散 1/λ²)**

変換 *y* = exp(-*x*) を考えると、変換後の確率分布関数は

$$P(y)=P(x)\,|\,dx/dy\,|$$

となる。一様乱数 *y* から逆変換

$$x=-\log(y)$$

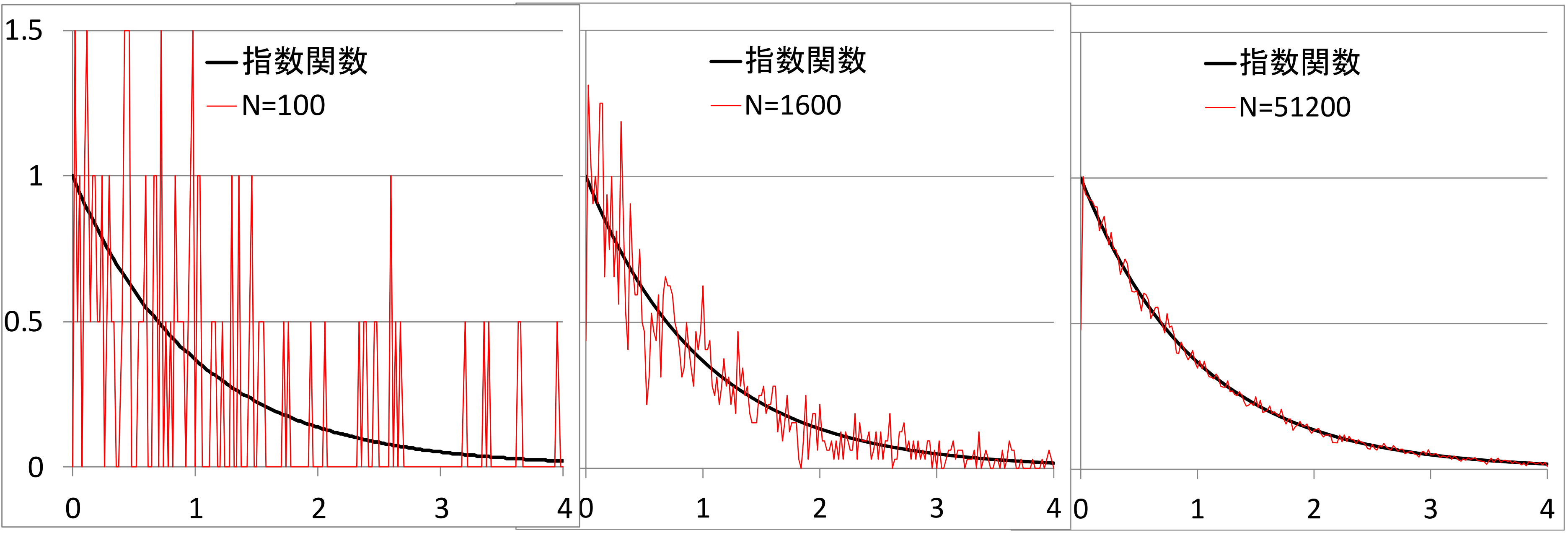
により、λ = 1の指数分布に従う乱数が得られる。

任意の λ に対しては

$$x' = x/\lambda$$

にすればよい

指数分布に従う乱数



正規分布に従う乱数 (Box-Muller法)

http://www.sat.t.u-tokyo.ac.jp/~omi/random_variables_generation.html#Gauss

$$p(x)=\left(\frac{1}{2\pi\sigma^2}\right)^{1/2}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$ **(平均 μ, 分散 σ の正規分布)**

一様乱数 *x*, *y* を作り、極座標へ変換

$$P(x,y)=P(x)P(y)=P(r,\theta)=\left(\frac{1}{2\pi}\right)r\exp\left(-\frac{r^2}{2}\right)$$

変数を *r* から *r*² に変える $P(r^2)=P(r)\,|\,dx/dy\,|=P(r)/(2r)$

$$P(r^2,\theta)=\left(\frac{1}{4\pi}\right)\exp\left(-\frac{r^2}{2}\right)$$

一様乱数 *r*, θ から

$x = r\cos(\theta),\ y = r\sin(\theta)$

が正規分布に従う乱数となるので、

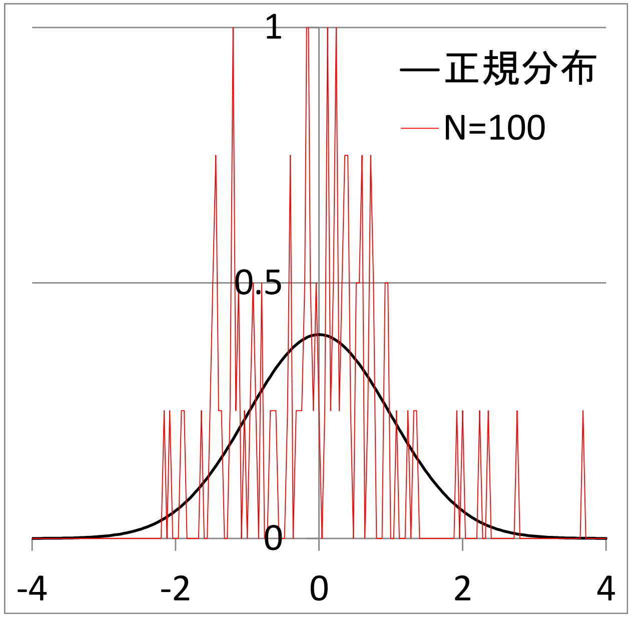
$$z=\left(-2.0*\log(x)\right)^{1/2}*\sin(2\pi y)$$

で計算できる。平均 μ, 分散 σ にするには

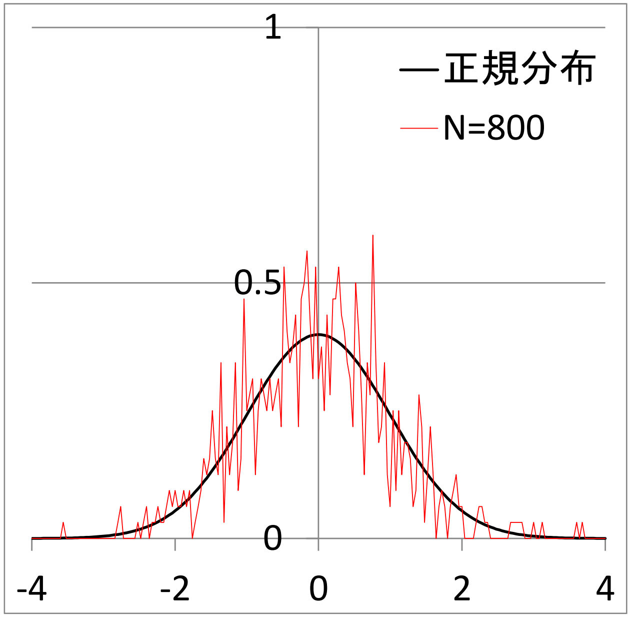
$z' = \mu + \sigma z$

にすればよい

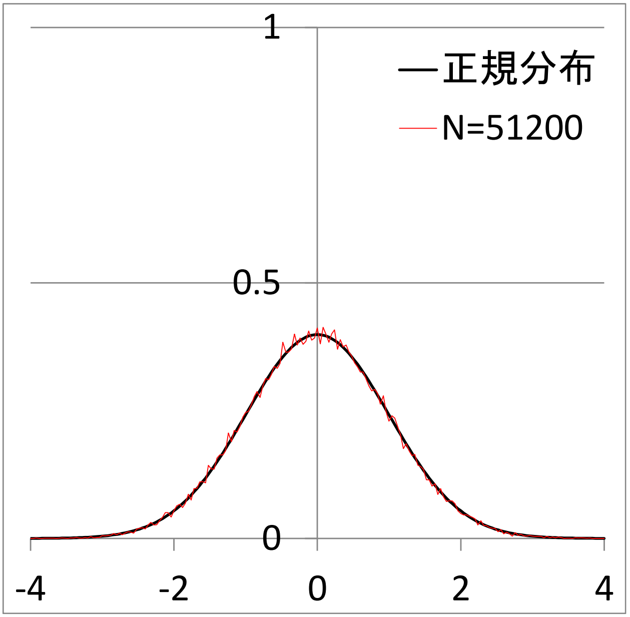
正規分布に従う乱数



—正規分布
N=100
1
0.5
0
-4
-2
0
2
4



—正規分布
N=800
1
0.5
0
-4
-2
0
2
4



—正規分布
N=51200
1
0.5
0
-4
-2
0
2
4

Q: Monte Carlo simulation for materials

Question of Monte Carlo simulations for statistical physics

How to collect **an ensemble that follows**

canonical statistics $P_i \propto \exp(-E_i/k_B T)$

Metropolis Monte Carlo法

ある物理状態を考え、このポテンシャルエネルギーを計算し U_1 とする。
乱数を使って別の物理状態を作り、このポテンシャルエネルギーを U_2 とする。

１．$\Delta U = U_2 - U_1 \leq 0$ であれば、無条件にその状態を採択する
２．$\Delta U > 0$ であれば、$\exp(-\Delta U/k_B T)$ の確率で採択する
　２．において、乱数 $0 \leq r \leq 1$ が $r \leq \exp(-\Delta U/k_B T)$ であれば採択、
　そうでなければ棄却し、状態1 をとりもどす

という手続により作られた集団は、統計力学の母集団に一致する

この母集団について物理量の平均をとれば統計平均としての
物理量が得られる。

Monte Carlo法の例: トンネリング

谷口研二 H6科研費(一般B) 研究成果報告書

エネルギー E_1 の状態から電子がトンネリングにより E_2 の状態へ遷移する
(トンネルでなくても、確率過程による遷移であれば同じ)

1=>2のトンネルレートをΓ^+, 2=>1をΓ^-,
正味のトンネルレートを $\Gamma = \Gamma^+ - \Gamma^-$ とする

$$\frac{\Gamma^+}{\Gamma^-} = \exp\left(-\frac{E_1 - E_2}{k_B T}\right) \qquad I = -e\Gamma = \frac{\Delta E}{eR_T}$$

トンネル抵抗 R_T の定義

$$\Gamma^+ = (E_1 - E_2) \Big/ \left[e^2 R_T \left(1 - \exp\left(-\frac{E_1 - E_2}{k_B T} \right) \right) \right]$$

トンネル時間

時間 $0 \sim t$ の間にトンネルが起こらない確率を $P(t)$ とする

$$P(t + dt) = P(t)(1 - \Gamma dt) \quad \Rightarrow \quad t = -\frac{1}{\Gamma} \ln P(t)$$

ある時刻から、実際にトンネルが生じるまでの時間 u は
$0 < r < 1$ の一様乱数 r を用いて右のように与えられる

$$u = -\frac{1}{\Gamma} \ln r$$

Matrix problems

行列問題の解法

Fundamental matrix operations

C = A+B:

```
for ix in range(nx):
    for iy in range(ny):
        c[ix][iy] = a[ix][iy] + b[ix][iy];
```

C = A*B:

```
for ix in range(nx):
    for iy in range(ny):
        c[ix][iy] = 0.0;
        for k in range(nk):
            c[ix][iy] = c[ix][iy] + a[ix][k]*b[k][iy];
```

To solve BC = A

(i) B^{-1} is obtained and calculate $B^{-1}A$

(ii) Directly solve BC = A

=> Better to use open libraries

Gauss elimination method (Gaussの消去法)

Upon a square matrix (正方行列) *A* and a vector *B* are given,
solution of ***AX* = *B*** is obtained by $X = A^{-1}B$.

- Efficient for case more than one solutions for the same *A* and different *B*.
- Can produce roundoff errors and not efficient

=> Solve the linear simultaneous equations directly.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & & a_{2n} \\ a_{31} & a_{32} & a_{33} & & a_{3n} \\ \vdots & & & \ddots & \\ a_{n1} & a_{n2} & a_{n3} & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Multiply a_{i1}/a_{11} ($i = 2, 3, \ldots, n$) to the first line and subtract it from *i*-th line

=> make all a_{i1} ($i \geq 2$) zero.

Repeat this procedure for all the lines, *A* will be converted to upper-right triangle matrix (右上三角行列)

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}' & a_{23}' & \cdots & a_{2n}' \\ 0 & 0 & a_{33}' & & a_{3n}' \\ \vdots & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}' \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1' \\ b_2' \\ \vdots \\ b_n' \end{pmatrix}$$

Solve from the last line to upper lines, giving all x_i

Note: Converting *A* to a band or triangle matrix enables solve the equation very easy

Row reduction method (掃き出し法)

Similar to the Gauss elimination method, but eliminates all non-diagonal terms

$$\begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22}' & 0 & \cdots & 0 \\ 0 & 0 & a_{33}' & & 0 \\ \vdots & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}' \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1' \\ b_2' \\ \vdots \\ b_n' \end{pmatrix}$$

Obtain the solution by $x_i = b_i' / \mathrm{a}_{ii}'$

Important: Regular matrix can be converted to triangle / band matrixes

(正則行列は、適当な行列による変換で三角行列や帯行列に分解できる)

=> *ex.* *LU* decomposition (LU分解)**: *A* = *LU***

***L*: Left-lower triangle, *U*: Right-upper triangle matrix**

Solution of linear simul. eqs. : LU decomposition

1. Convert *AX* = *B* to *LUX* = *B* by *A* = *LU*

2. Solve *LY* = *B* to obtain *Y*

3. Solve *UX* ＝ *Y* to obtain *X*

Diagonalization of real symmetric matrix: Jacobi method (ヤコビ法)

Diagonalization of $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix}$

=> can be done by conversion *U*

=> can be done by conversion $U^{\mathrm{T}}AU$ with an orthogonal matrix (直交行列) ***U***

$$U = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

$$U^{T}AU = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix}\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}\cos^2\theta + 2a_{12}\cos\theta\sin\theta + a_{22}\sin^2\theta & (-a_{11}+a_{22})\cos\theta\sin\theta + a_{12}(\cos^2\theta - \sin^2\theta) \\ (-a_{11}+a_{22})\cos\theta\sin\theta + a_{12}(\cos^2\theta - \sin^2\theta) & a_{11}\sin^2\theta - 2a_{12}\cos\theta\sin\theta + a_{22}\cos^2\theta \end{pmatrix}$$

$$(-a_{11}+a_{22})\cos\theta\sin\theta + a_{12}(\cos^2\theta - \sin^2\theta) = 1/2\left[(-a_{11}+a_{22})\sin 2\theta + a_{12}\cos 2\theta\right] = 0$$

⇒

$$\begin{array}{ll} \theta = \pi/4 & a_{11} = a_{22} \\ \theta = (1/2)\tan^{-1}(2a_{12}/(a_{11}-a_{22})) & a_{11} \neq a_{22} \end{array}$$

Jacobi method

1. Choose the largest absolute value non-diagonal element a_{ij} in

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{12} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{13} & a_{23} & a_{33} & & 0 \\ \vdots & & & \ddots & \vdots \\ a_{1n} & a_{2n} & & \cdots & a_{nn} \end{pmatrix}$$

2. Converting by $A' = U^T A U$ with

$$U = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & & & & & \vdots \\ \vdots & & \cos\theta & & -\sin\theta & & \vdots \\ \vdots & & & 1 & & & \vdots \\ \vdots & & \sin\theta & & \cos\theta & & \vdots \\ \vdots & & & & & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix}$$

will give $a_{ij}' = 0$

3. Choose the largest absolute value element a_{ij}' and repeat 2

=> The square sum of non-diagonal elements is reduce by a factor of $2a_{ij}^2$

=> finite iterations will complete the diagonalization

But it is hard to estimate the number of iterations required, and Jacobi method is not efficient for a large-size materix

Diagonalization of large-size matrix

Householder method

1. **Convert a symmetric matrix *A* to a triple diagonal matrix** (三重対角行列) ***D*** **using an orthogonal matrix** (直交行列) ***U***

   Note: eigen values of U^TAU are equal to those of A

2. **Solve eigen values of *D* by bisection method**

QR method

1. **Regular $n \times n$ matrix *A* is decomposed to $A = QR$** (QR分解) **using a regular orthogonal matrix *Q* and a right-upper matrix with positive diagonal elements *R*.**
2. **QR-decompose A_k: $A_k = Q_k R_k$**
3. **Convert A_k to $A_{k+1} = Q_k^T A_k Q_k = R_k Q_k$ (similar transformation,** 相似変換**)**
4. **Repeating 2 and 3 will converge A_k to a right-upper triangle matrix A_R => Solve eigen values of A_R**

   If A is a symmetric matrix, A_R will be a diagonal matrix.

Applications

応用

Linear algebra libraries

(線形幾何学・行列計算ライブラリ)

Fortran, C, C++, etc

LAPACK (Linear Algebra PACKage)

ScaLAPACK (Scalable LAPACK)

Intel Math Kernel Library (MKL)

One API: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

Python: numpy.linalg, scipy.linalg

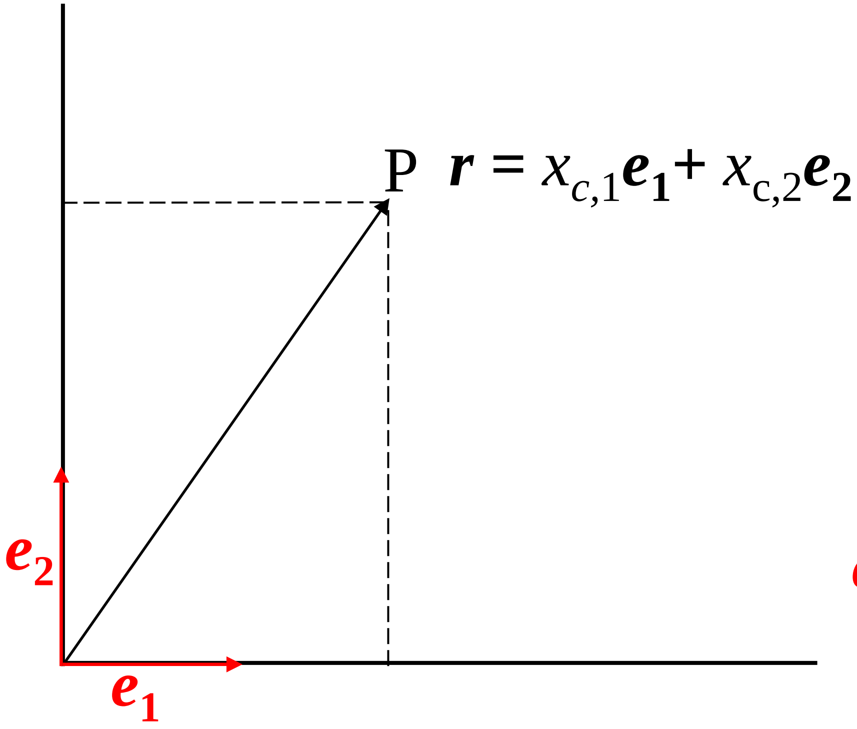
matrix.py

Product of matrixes	**AB**	**:** C	= A @ B
Inner product	**V1･V2**	**:** inner	= numpy.dot(V1, V2)
		inner	= numpy.inner(V1, V2)
Outer product	**V1 × V2**	**:** V3	= numpy.cross(V1, V2)
Inverse matrix		**:** Ai	= numpy.linalg.inv(A)
Determinant		**:** det	= numpy.linalg.det(A)
Eigen values/vectors		**:** lA, vA	= numpy.linalg.eig(A)
Solve simul. linear eqs.	**AX = B**	**:** X	= numpy.linalg.solve(A, B)
LU decomposition		**:** P, L, U	= numpy.linalg.lu(A)
Cholesky decomposition	**$A=LL^T$**	**:** L	= numpy.linalg.cholesky(A)
QR decomposition	**A=QR**	**:** Q, R	= scypy.linalg.qr(A)

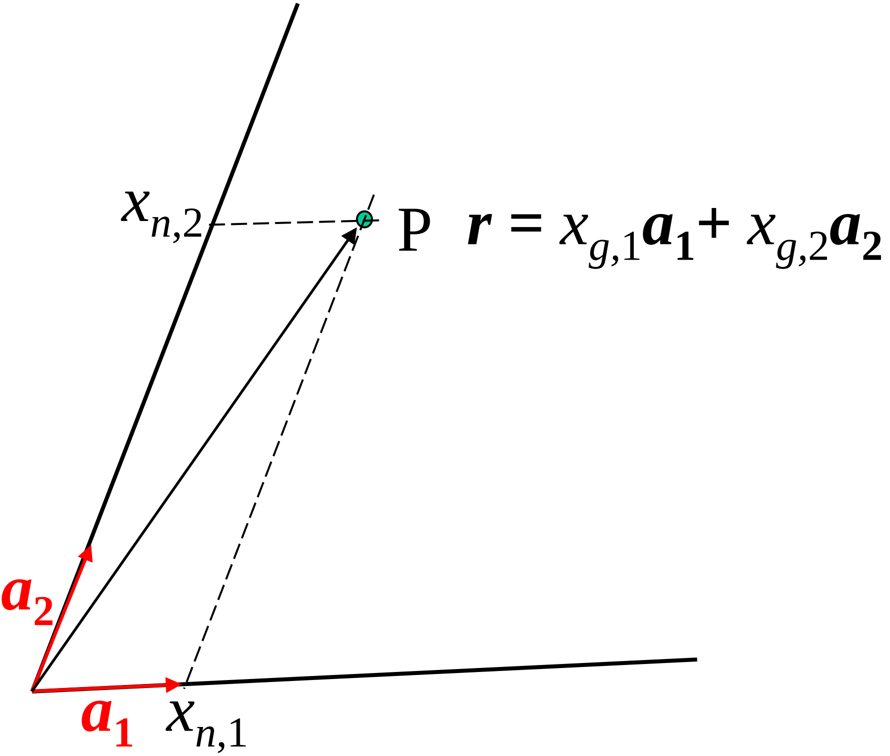
一般座標系 (general coordinate system)

直交座標系 (Orthogonal)
デカルト座標系 (Cartesian)

一般座標/非直交系 (Non-Cartesian)



$\boldsymbol{r} = x_{c,1}\boldsymbol{e}_1 + x_{c,2}\boldsymbol{e}_2$



$\boldsymbol{r} = x_{g,1}\boldsymbol{a}_1 + x_{g,2}\boldsymbol{a}_2$

正規直交系 (orthonormal system)

$$\boldsymbol{e_i} \cdot \boldsymbol{e_j} = \delta_{ij}$$

$$|\boldsymbol{e_i}| = 1$$

一般座標系 (general coordinate system)

$$\boldsymbol{a_i} \cdot \boldsymbol{a_j} \neq \delta_{ij}$$

$\boldsymbol{e_i}$, $\boldsymbol{a_i}$: **基底ベクトル (base vecor)**

Cartesian – general coord. Conversion

(直交系 − 一般座標系変換)

$$\boldsymbol{r} = x_{c,1}\boldsymbol{e}_1 + x_{c,2}\boldsymbol{e}_2 = x_{g,1}\boldsymbol{a}_1 + x_{g,2}\boldsymbol{a}_2$$

$$x_{c,1} = x_{g,1}\,\boldsymbol{a}_1 \cdot \boldsymbol{e}_1 + x_{g,2}\,\boldsymbol{a}_2 \cdot \boldsymbol{e}_1$$
$$x_{c,2} = x_{g,1}\,\boldsymbol{a}_1 \cdot \boldsymbol{e}_2 + x_{g,2}\,\boldsymbol{a}_2 \cdot \boldsymbol{e}_2$$

If $\boldsymbol{a}_1 = a_{11}\boldsymbol{e}_1 + a_{12}\boldsymbol{e}_2$

$\boldsymbol{a}_2 = a_{21}\boldsymbol{e}_1 + a_{22}\boldsymbol{e}_2$

are given,

$$\begin{pmatrix}\boldsymbol{a}_1\\ \boldsymbol{a}_2\end{pmatrix} = \begin{pmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{pmatrix}\begin{pmatrix}\boldsymbol{e}_1\\ \boldsymbol{e}_2\end{pmatrix}$$

$$x_{c,1} = x_{g,1}a_{11} + x_{g,2}a_{21}$$
$$x_{c,2} = x_{g,1}a_{12} + x_{g,2}a_{22}$$

$$\begin{pmatrix}x_{c,1}\\ x_{c,2}\end{pmatrix} = \begin{pmatrix}a_{11} & a_{21}\\ a_{12} & a_{22}\end{pmatrix}\begin{pmatrix}x_{g,1}\\ x_{g,2}\end{pmatrix}$$

Fractional coordinates in crystal
(結晶の内部座標)

Lattice parameters: $a, b, c\ (= a_1, a_2, a_3), \alpha, \beta, \gamma (= \alpha_{23}, \alpha_{13}, \alpha_{12})$
Lattice vectors: $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3 = \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$

$$\boldsymbol{r} = x_{f,1}\boldsymbol{a}_1 + x_{f,2}\boldsymbol{a}_2 + x_{f,3}\boldsymbol{a}_3 = x_{c,1}\boldsymbol{e}_1 + x_{c,2}\boldsymbol{e}_2 + x_{c,3}\boldsymbol{e}_3$$

$(x_{f,1}, x_{f,1}, x_{f,3})$: Fractional coordinate (部分座標)
Internal coordinate (内部座標)

$$|\boldsymbol{a}_i| = a_i$$

$$\boldsymbol{a}_i \cdot \boldsymbol{a}_i = a_i a_i \cos \alpha_{ij} \quad (i \neq j)$$

$$\begin{pmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \boldsymbol{a}_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{e}_1 \\ \boldsymbol{e}_2 \\ \boldsymbol{e}_3 \end{pmatrix}$$

Fractional coordinate to Cartesian coordinate

$$\begin{pmatrix} x_{c,1} \\ x_{c,2} \\ x_{c,3} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{pmatrix} \begin{pmatrix} x_{f,1} \\ x_{f,2} \\ x_{f,3} \end{pmatrix}$$

Conversion matrix

$$\begin{pmatrix}\boldsymbol{a_1}\\ \boldsymbol{a_2}\\ \boldsymbol{a_3}\end{pmatrix}=\begin{pmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\\ a_{31} & a_{32} & a_{33}\end{pmatrix}\begin{pmatrix}\boldsymbol{e_1}\\ \boldsymbol{e_2}\\ \boldsymbol{e_3}\end{pmatrix}$$

$|\boldsymbol{a_i}| = a_i$

$\boldsymbol{a_i}\cdot\boldsymbol{a_j} = \cos\alpha_{ij}\ \ (i \neq j)$

$a, b, c\ \ (= a_1, a_2, a_3)$

$\alpha, \beta, \gamma\ (= \alpha_{23}, \alpha_{13}, \alpha_{12})$

tkcrystalbase.cal_lattice_vectors()

$$\begin{pmatrix}\boldsymbol{a_1}\\ \boldsymbol{a_2}\\ \boldsymbol{a_3}\end{pmatrix}=\begin{pmatrix}a & 0 & 0\\ b\cos\gamma & b\sin\gamma & 0\\ c\cos\beta & c\cos\beta - c\cos\beta\cos\gamma & a_{33}\end{pmatrix}\begin{pmatrix}\boldsymbol{e_1}\\ \boldsymbol{e_2}\\ \boldsymbol{e_3}\end{pmatrix}$$

$$a_{33} = \sqrt{c^2 - {a_{31}}^2 - {a_{32}}^2}$$

Lattice properties

Unit cell volume

$$V = \boldsymbol{a}_1 \cdot (\boldsymbol{a}_2 \times \boldsymbol{a}_3)$$

tkcrystalbase.cal_volume ()

Distance $\boldsymbol{r}_{kl} = \boldsymbol{r}_k - \boldsymbol{r}_l$

tkcrystalbase.distance2() / .distance()

$$r_{kl}{}^2 = |\boldsymbol{r}_{kl}|^2 = \sum_{\boldsymbol{i}=0}^{2} \sum_{\boldsymbol{j}=0}^{2} \boldsymbol{a}_{\boldsymbol{i}} \cdot \boldsymbol{a}_{\boldsymbol{j}} x_{kl,i} x_{kl,j} = \sum_{i,j} g_{ij} x_{kl,i} x_{kl,j}$$

$g_{ij} = \boldsymbol{a}_i \cdot \boldsymbol{a}_j$: **Metric tensor (計量テンソル)**

tkcrystalbase.cal_metrics()

Reciprocal lattice vectors

tkcrystalbase.cal_reciprocal_lattice_vectors()

$$\boldsymbol{a}^*{}_{\boldsymbol{1}} = \boldsymbol{a}_2 \times \boldsymbol{a}_3 / \mathrm{V}$$

$$\boldsymbol{a}^*{}_{\boldsymbol{2}} = \boldsymbol{a}_3 \times \boldsymbol{a}_1 / \mathrm{V}$$

$$\boldsymbol{a}^*{}_{\boldsymbol{3}} = \boldsymbol{a}_1 \times \boldsymbol{a}_2 / \mathrm{V}$$

Reciprocal vector at $(h\ k\ l)$

$$\boldsymbol{G}_{\boldsymbol{hkl}} = h\boldsymbol{a}^*{}_{\boldsymbol{1}} + k\boldsymbol{a}^*{}_{\boldsymbol{2}} + l\boldsymbol{a}^*{}_{\boldsymbol{3}}$$

Lattice space

$$d_{hkl}{}^{-2} = |\boldsymbol{G}_{\boldsymbol{hkl}}|^2 = \sum_{\boldsymbol{i}=0}^{3} \sum_{\boldsymbol{j}=0}^{3} \boldsymbol{a}^*{}_{\boldsymbol{i}} \cdot \boldsymbol{a}^*{}_{\boldsymbol{j}} h_i h_j = \sum_{\boldsymbol{i},\boldsymbol{j}} Rg_{ij} h_i h_j$$

Bragg angle

$$2d_{hkl} \sin\theta = \lambda$$

$h, k, l \ \ (= h_1, h_2, h_3)$

$Rg_{ij} = \boldsymbol{a}^*{}_{\boldsymbol{i}} \cdot \boldsymbol{a}^*{}_{\boldsymbol{j}}$

Inter-atomic distances

python crystal_distance.py

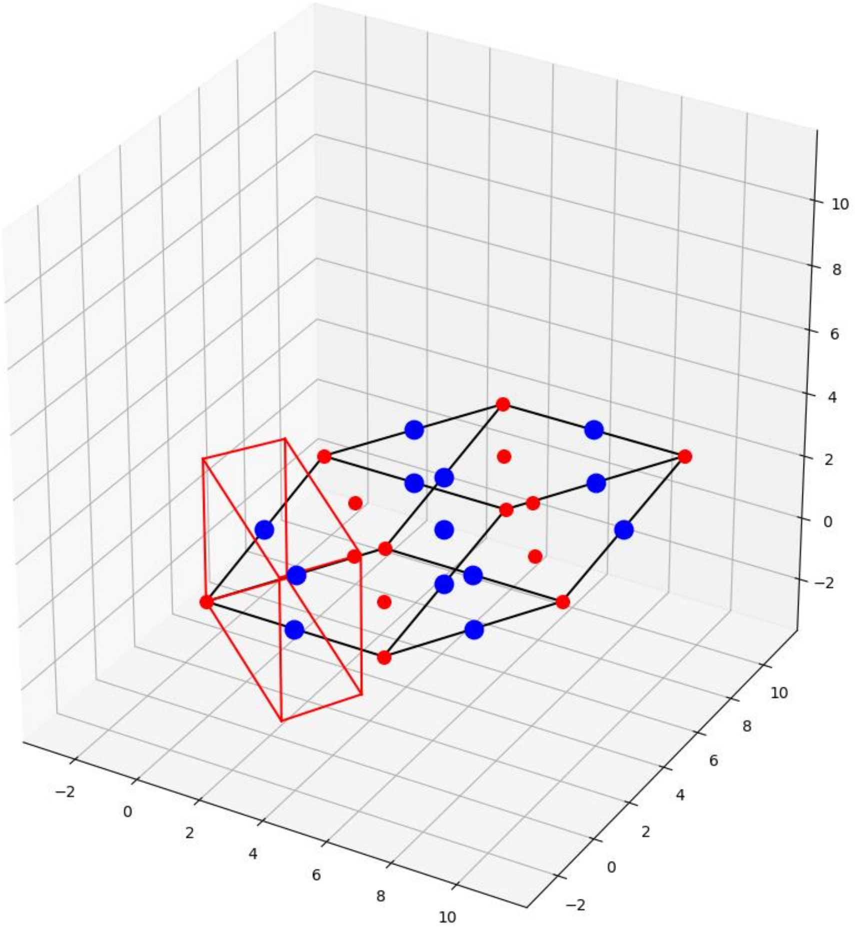
NaCl

```
Lattice parameters: [5.62, 5.62, 5.62, 90.0, 90.0, 90.0]
Lattice vectors:
  ax: (      5.62,          0,          0) A
  ay: ( 2.546e-10,       5.62,          0) A
  az: ( 2.546e-10,          0,       5.62) A
Metric tensor:
  gij: (     31.58,  1.431e-09,  1.431e-09) A
       ( 1.431e-09,      31.58,   6.48e-20) A
       ( 1.431e-09,   6.48e-20,      31.58) A
Volume:        177.5 A^3
Unit cell volume:        177.5 A^3
Reciprocal lattice parameters: [0.17793594306049823, 0.17793594306049823, 0.17793594306049823, 90.00000000257246,
90.00000000516778, 90.00000000516778]
Reciprocal lattice vectors:
  Rax: (    0.1779,  -8.06e-12,  -8.06e-12) A^-1
  Ray: (         0,     0.1779,          0) A^-1
  Raz: (         0,          0,     0.1779) A^-1
Reciprocal lattice metric tensor:
  Rgij: (   0.03166, -1.422e-12, -1.422e-12) A^-1
        (-1.422e-12,    0.03166,  6.382e-23) A^-1
        (-1.422e-12,  6.382e-23,    0.03166) A^-1
Reciprocal unit cell volume:     0.005634 A^-3

nmax: 1 1 1

Interatomic distances:
  Cl1  (     0.5,        0,        0) - Na4  (     0.5,      0.5,        0) + ( 0, -1,  0): dis =       2.81 A
(cut)
  Na4  (     0.5,      0.5,        0) - Na1  (       0,        0,        0) + ( 0,  1,  0): dis =      3.974 A
  Na4  (     0.5,      0.5,        0) - Na2  (       0,      0.5,      0.5) + ( 1,  0, -1): dis =      3.974 A
  Na4  (     0.5,      0.5,        0) - Na1  (       0,        0,        0) + ( 1,  0,  0): dis =      3.974 A
```

Fractional – Cartesian conversion

python crystal_draw_cell.py

Rhombohedral cell
and reciprocal unit cell



Bragg angles

NaCl

```
python crystal_xrd.py
```

```
Lattice parameters: [5.62, 5.62, 5.62, 90.0, 90.0, 90.0]
Lattice vectors:
  ax: (       5.62,          0,          0) A
  ay: ( 2.546e-10,       5.62,          0) A
  az: ( 2.546e-10,          0,       5.62) A
Metric tensor:
  gij: (      31.58,  1.431e-09,  1.431e-09) A
       ( 1.431e-09,      31.58,   6.48e-20) A
       ( 1.431e-09,   6.48e-20,      31.58) A
Volume:        177.5 A^3
Unit cell volume:        177.5 A^3
Reciprocal lattice parameters: [0.17793594306049823, 0.17793594306049823, 0.17793594306049823, 90.00000000257246,
90.00000000516778, 90.00000000516778]
Reciprocal lattice vectors:
  Rax: (    0.1779,  -8.06e-12,  -8.06e-12) A^-1
  Ray: (         0,     0.1779,          0) A^-1
  Raz: (         0,          0,     0.1779) A^-1
Reciprocal lattice metric tensor:
  Rgij: (   0.03166, -1.422e-12, -1.422e-12) A^-1
        (-1.422e-12,    0.03166,  6.382e-23) A^-1
        (-1.422e-12,  6.382e-23,    0.03166) A^-1
Reciprocal unit cell volume:     0.005634 A^-3
hkl range: 7 7 7

Diffraction angle, d, h, k, l:
  2Q=       15.75  d=        5.62  ( -1   0   0)
  2Q=       15.75  d=        5.62  (  0  -1   0)
(cut)
  2Q=       22.35  d=     3.97394  ( -1  -1   0)
  2Q=       22.35  d=     3.97394  ( -1   0  -1)
  2Q=       22.35  d=     3.97394  (  1   0   1)
¥
```

Madelung potential

Sum of Coulomb potential in 3D is very slowly converging

Potential is proportional to r^{-1}

Polarization potential due to +/- ions is to r^{-2}

Number of ions on the sphere surface at radius r is to r^2

=> Contribution of ions from a surface region at r
to Coulomb sum is almost constant, independent of r

$$U_{ij}\left(r_{ij}\right)=\frac{Z_i Z_j e^2}{4\pi\varepsilon_0}\frac{1}{r_{ij}}+U_{Rij}\left(r_{ij}\right)$$

$$U=\frac{1}{2}\sum_{i\neq j}U_{ij}=-A_M N_A\frac{Z^2e^2}{4\pi\varepsilon_0 R}+U_R$$

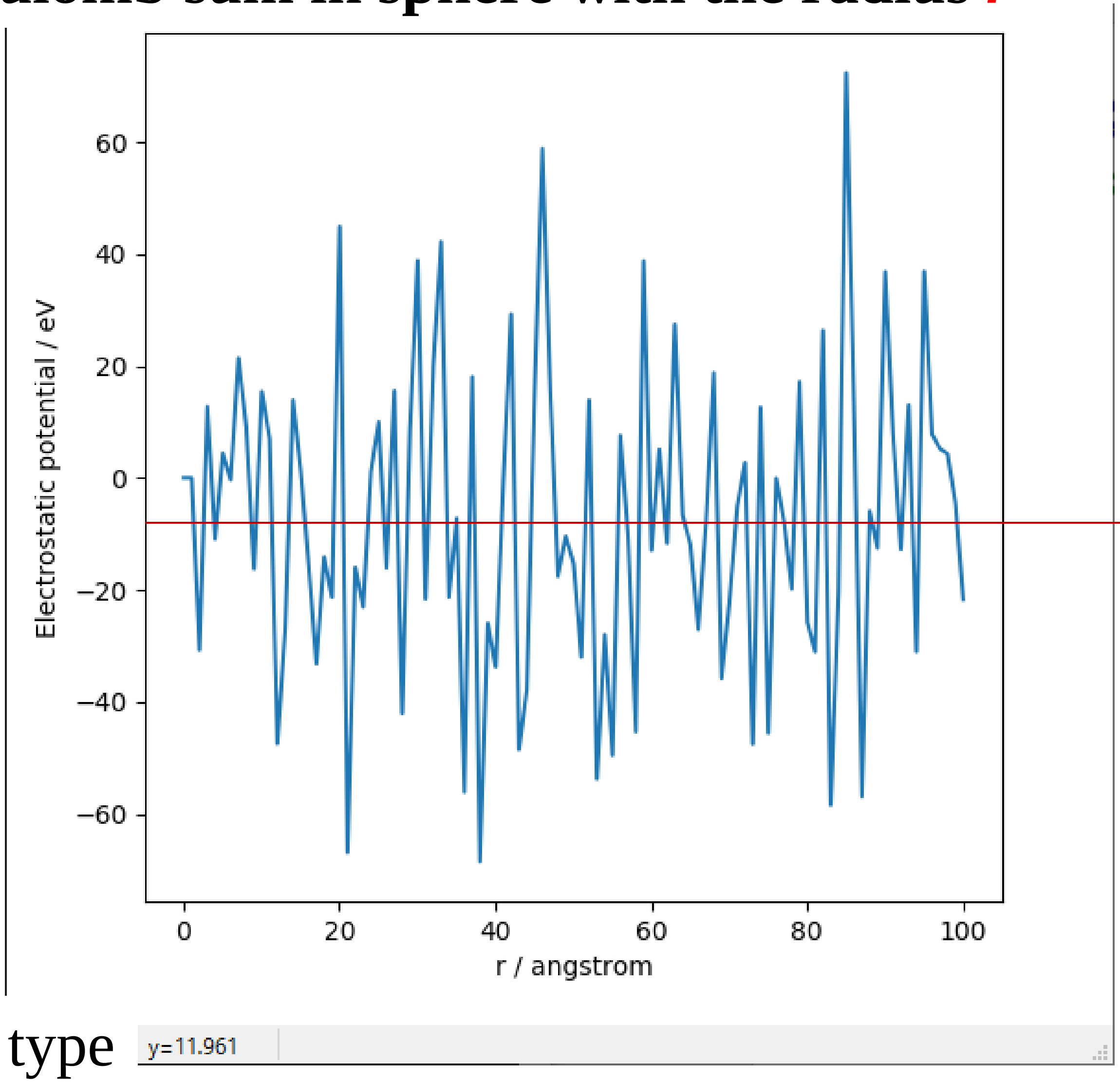
Crystal structure	A_r
Rock salt type(NaCl)	1.7476
CsCl type(CsCl)	1.7627
Zinc blend (CuCl)	1.6380
Wurzite(ZnO)	1.6413
Cu_2O type	4.116
Fluorite type (CaF_2)	2.520

$$A_M=\frac{1}{2}\sum_{i\neq j}\frac{1}{r_{ij}/R}$$ **Madelung constant**

Madelung potential: Simple sum

```
python crystal_MP_simple.py
```

Coulomb sum in sphere with the radius *r*



Exact: -8.9 eV

Rock salt type

Efficient Coulomb sum: Evjen method

Sum up Coulomb potential in units with zero net charge

Ion charges: Z_i
On boundary plane : $1/2Z_\mathrm{i}$
On boundary edge : $1/4Z_\mathrm{i}$
On boundary corner : $1/8Z_\mathrm{i}$

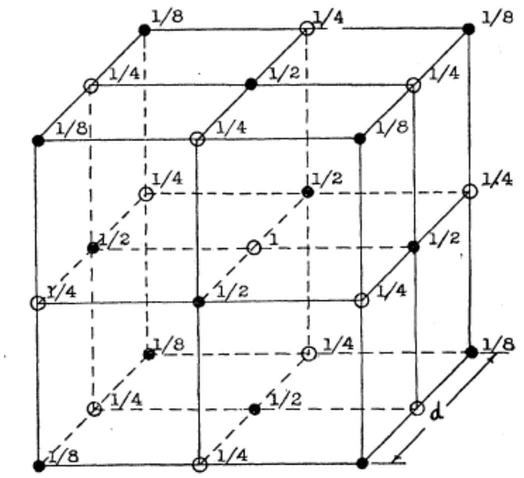


Fig. 1. Elementary cell of the NaCl-type.

Madelung constant of Rock salt type structure

$$A_M = -\frac{1}{2}\sum_{n_x,n_y,n_z=-\infty,\neq(0,0,0)}^{\infty}(-1)^{n_x+n_y+n_z}\frac{1}{\sqrt{n_x{}^2+n_y{}^2+n_z{}^2}}$$

$$A_M = 6\times\frac{1}{2}\times\frac{1}{\sqrt{1}} - 12\times\frac{1}{4}\times\frac{1}{\sqrt{1+1}} + 8\times\frac{1}{8}\times\frac{1}{\sqrt{1+1+1}} = 1.456$$

Madelung potential: Evjen method

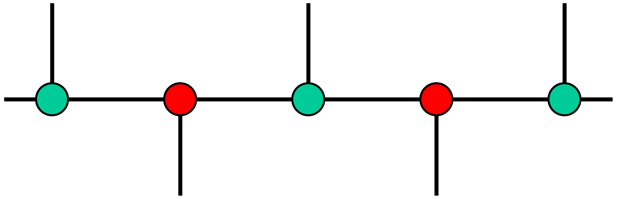
Usage: python crystal_MP_Evjen.py n_{cell}

n_{cell}	MP	Madelung constant
1	-8.9766	1.7517691
2	-8.95586	1.7477211
3	-8.95521	1.7475955
4	-8.9511	1.7475744
5	-8.95508	1.7475686
6	-8.95507	1.7475665
8	-8.95506	1.7475652
10	-8.95506	1.7475648
Exact (精確値)		1.74756

Rock salt type

3D sum of Coulomb potential: Ewald method

Periodic calculation can be enhanced by FT?



Periodic positions of charge

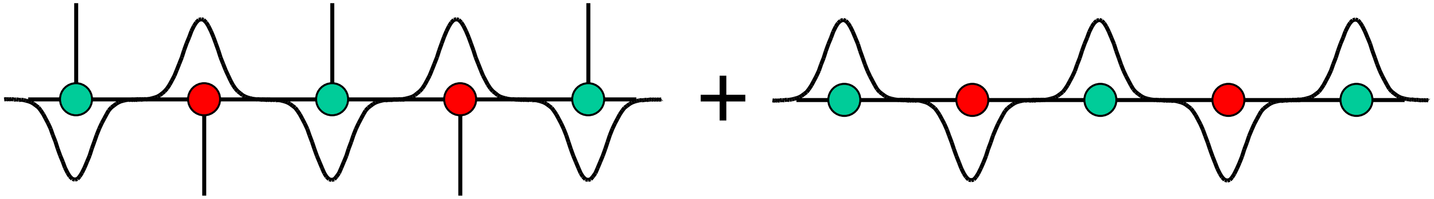
=> converted to the origin of FT data

But the charges are point charges

=> converted to infinite in FT space

=> Calculate for charges with finite width

(拡がりのある電荷の周期配列として計算する)



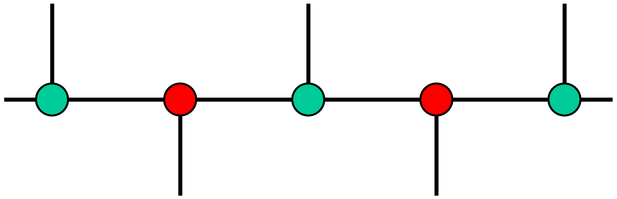
3D sum of Coulomb potential: Ewald method

The finite width charge distributions are converted by FT

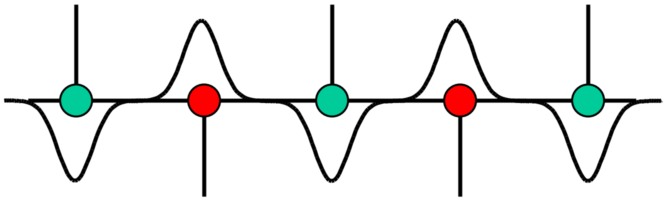
=> Take faster calculation parts in the real space and the reciprocal space

拡がった電荷のフーリエ変換を利用し、実空間和と逆空間和の計算の速い部分をとる

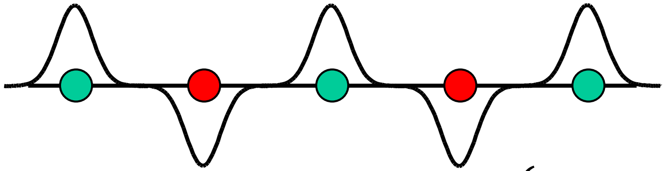
$$\Phi_i = K_C Z_i \sum_j \frac{Z_j}{r_{ij}} \quad \left(K_C = \frac{e^2}{4\pi\varepsilon_0}\right)$$



$$\Phi_i^I = K_C Z_i \sum_j Z_j \frac{\operatorname{erfc}(\alpha|r_{ij}|)}{|r_{ij}|}$$



$$\Phi_i^{II} = K_C \frac{Z_i}{\pi V} \sum_{h,k,l} \frac{1}{|\mathbf{G}_{hkl}|^2} \exp\left(-\frac{\pi^2 |\mathbf{G}_{hkl}|^2}{\alpha^2}\right)$$



$$\times \left\{\cos(2\pi \mathbf{G}_{hkl} \cdot \mathbf{r}_i) \textstyle\sum_j Z_j \cos(2\pi \mathbf{G}_{hkl} \cdot \mathbf{r}_j) + \sin(2\pi \mathbf{G}_{hkl} \cdot \mathbf{r}_i) \sum_j Z_j \sin(2\pi \mathbf{G}_{hkl} \cdot \mathbf{r}_j)\right\}$$

$$\boldsymbol{G}_{hkl} \cdot \boldsymbol{r_i} = hx_i + ky_i + lz_i$$

$$\Phi_i^{III} = K_C Z_i \frac{2\alpha Z_i}{\sqrt{\pi}}$$

$$\boxed{\Phi_i = \Phi_i^I + \Phi_i^{II} - \Phi_i^{III}}$$

Madelung potential: Ewald method

Usage: python crystal_MP_Ewald.py alpha prec

Alpha	**Precision**	**MP**	**Madelung constant**	**Range**	**Time (s)**
0.3	10^{-3}	-8.95558	1.7476663	10.1/222 0.063 /222	0.016/0 /0.016
0.3	$\mathbf{10^{-5}}$	**-8.95506**	**1.7475646**	11.9/333 0.105 /222	0.031/0 /0.031
0.3	$\mathbf{10^{-7}}$	**-8.95506**	**1.7475646**	13.6/333 0.147 /333	0.047/0 /0.047
0.2	10^{-3}	-8.95506	1.7475646	15.2/333 0.028 /111	0.042/0 /0.042
0.6	10^{-3}	-8.95607	1.7477629	5.1/111 0.25 /333	0 /0.016 /0.016
0.8	10^{-3}	-8.95584	1.747718	3.8/111 0.45 /444	0 /0.016 /0.016
0.2	$\mathbf{10^{-10}}$	**-8.95506**	**1.7475646**	24.3/555 0.093/222	0.16/0 /0.16
0.4	$\mathbf{10^{-10}}$	**-8.95506**	**1.7475646**	12.1/333 0.373/444	0.036/0.016/0.052
0.5	$\mathbf{10^{-10}}$	**-8.95506**	**1.7475646**	9.7/222 0.58 /555	0.016/0.016/0.031
0.6	$\mathbf{10^{-10}}$	**-8.95506**	**1.7475646**	8.1/222 0.84 /666	0.016/0.031/0.047
Exact (精確値)			**1.74756**		

Range: R_{max} [Å]/$n_{xmax}n_{ymax}n_{zmax}$ G_{max} [Å^{-1}]/$h_{max}k_{max}l_{max}$
Time: Real space sum / Reciprocal space sum / Total [s]

Rock salt type

Comparison: Evjen method

Rock salt type

$$A_M = -\frac{1}{2}\sum_{n_x,n_y,n_z=-\infty,\neq(0,0,0)}^{\infty}(-1)^{n_x+n_y+n_z}\frac{1}{\sqrt{{n_x}^2+{n_y}^2+{n_z}^2}}$$

nx	ny	nz	r	m	Z	S(mZ/r)	f	S(mZf/r)
0	0	1	1	6	−1	−6	0.5	−3
0	1	1	1.4142	12	1	8.48528	0.25	2.12132034
1	1	1	1.7321	8	−1	−4.6188	0.13	−0.5773503
						−2.13		−1.456

nx	ny	nz	r	m	Z	S(mZ/r)	f	S(mZf/r)
0	0	1	1	6	−1	−6	1	−6
0	1	1	1.4142	12	1	8.48528	1	8.48528137
1	1	1	1.7321	8	−1	−4.6188	1	−4.6188022
0	0	2	2	6	1	3	0.5	1.5
0	1	2	2.2361	24	−1	−10.733	0.5	−5.3665631
0	2	2	2.8284	12	1	4.24264	0.25	1.06066017
1	1	2	2.4495	24	1	9.79796	0.5	4.89897949
1	2	2	3	24	−1	−8	0.25	−2
2	2	2	3.4641	8	1	2.3094	0.13	0.28867513
						−1.52		−1.7518

nx	ny	nz	r	m	Z	S(mZ/r)	f	S(mZf/r)
0	0	1	1	6	−1	−6	1	−6
0	1	1	1.4142	12	1	8.48528	1	8.485281374
1	1	1	1.7321	8	−1	−4.6188	1	−4.61880215
0	0	2	2	6	1	3	1	3
0	1	2	2.2361	24	−1	−10.733	1	−10.7331263
0	2	2	2.8284	12	1	4.24264	1	4.242640687
1	1	2	2.4495	24	1	9.79796	1	9.797958971
1	2	2	3	24	−1	−8	1	−8
2	2	2	3.4641	8	1	2.3094	1	2.309401077
0	0	3	3	6	−1	−2	0.5	−1
0	1	3	3.1623	24	1	7.58947	0.5	3.794733192
0	2	3	3.6056	24	−1	−6.6564	0.5	−3.32820118
0	3	3	4.2426	12	1	2.82843	0.25	0.707106781
1	1	3	3.3166	24	−1	−7.2363	0.5	−3.61813613
1	2	3	3.7417	48	1	12.8285	0.5	6.414269806
1	3	3	4.3589	24	−1	−5.506	0.25	−1.3764944
2	2	3	4.1231	24	−1	−5.8209	0.5	−2.9104275
2	3	3	4.6904	24	1	5.11682	0.25	1.279204298
3	3	3	5.1962	8	−1	−1.5396	0.13	−0.19245009
						−1.91		−1.7470

Exact value = 1.7476

Basis function in quantum calculation

How to plot band structure

Schrödinger eq.: **Plane wave method** (平面波法)

Plane waves are employed as basis set of linear combination

$$\phi_{\mathbf{k}}(\mathbf{r}) = \exp\left(i\mathbf{k}\cdot\mathbf{r}\right)\sum C_{hkl}u_{hkl}(\mathbf{r}) \qquad u_{hkl}(\mathbf{r}) = \exp\left[i\mathbf{G}_{\mathbf{hkl}}\cdot\mathbf{r}\right]$$

Plane waves with wave numbers $\mathbf{G}_{\mathbf{hkl}}$ forms a perfect basis of periodic system

Any function is represented if use all G_{hkl} for all

=> **In actual calculation, approximate by $|G_{\mathbf{hkl}}| < G_{\mathbf{max}}$**

$$\begin{vmatrix} H_{11}-ES_{11} & H_{12}-ES_{12} & \cdots & H_{1n}-ES_{1n} \\ H_{21}-ES_{21} & H_{22}-ES_{ss} & & H_{2n}-ES_{2n} \\ \vdots & & \ddots & \vdots \\ H_{n1}-ES_{n1} & H_{n2}-ES_{n2} & \cdots & H_{nn}-ES_{nn} \end{vmatrix} = 0$$

$$\left\langle u_{h'k'l'}\middle|H\middle|u_{hkl}\right\rangle = \int e^{-i\left(\mathbf{k}+\mathbf{G}_{\mathbf{h'k'l'}}\right)\cdot\mathbf{r}}\left[-\frac{\hbar^2}{2m}\nabla^2+V(\mathbf{r})\right]e^{i\left(\mathbf{k}+\mathbf{G}_{\mathbf{hkl}}\right)\cdot\mathbf{r}}d\mathbf{r}$$

$$= \delta_{hkl,h'k',l'}\frac{\hbar^2}{2m}\left(\mathbf{k}+\mathbf{G}_{\mathbf{hkl}}\right)^2 + \underline{V^*(\mathbf{G}_{\mathbf{hkl}}-\mathbf{G}_{\mathbf{h'k'l'}})}$$

Most of PW calculations are done by Fourier transformation

=> Possibly speed up with GPU

プログラム: 一次元平面波法

http://conf.msl.titech.ac.jp/jsap-crystal/

平面波基底による一次元バンド計算　pw1d.py

Usage: python pw1d.py

python pw1d.py (ft a na pottype bwidth bpot)

python pw1d.py (band a na pottype bwidth bpot nG kmin kmax nk)

python pw1d.py (wf a na pottype bwidth bpot nG kw iLevel xwmin xwmax nxw)

pottype: rect|gauss

実行例: python pw1d.py ft 5.4064 64 rect 0.5 10.0

ポテンシャルのフーリエ変換を表示。

格子定数5.4064Å、単位格子を $2^6 = 64$分割 (FFTのためnaは2^n)

矩形ポテンシャル 0.5 Å幅、10.0 eV高さ

実行例: python pw1d.py band 5.4064 64 rect 0.5 10.0 3 -0.5 0.5 21

バンド構造を計算。構造、分割数、ポテンシャルは上と同じ

バンド構造を 逆空間内部座標 [-½ ½] (第一ブリルアンゾーン) で21分割して表示

実行例: python pw1d.py wf 5.4064 64 rect 0.5 10.0 3 0.0 0 0.0 16.2192 101

結晶波動関数を表示。構造、分割数、ポテンシャルは上と同じ

波数ベクトルはΓ点に近い3点を用いる。

k = 0.0 (Γ点), 固有解の 0 番目の準位の波動関数を、

0.0 ~ 16.2192 オングストロームの範囲で101分割して表示

(注意: 固有解はエネルギー順にソートしていないので、
コンソール出力のEnergy levels:で準位の番号を確認)

```
Energy levels:
  0     0.624459 eV
  1      6.39666 eV
  2      6.08362 eV
```

Program: 1-D PW method

pw1d.py

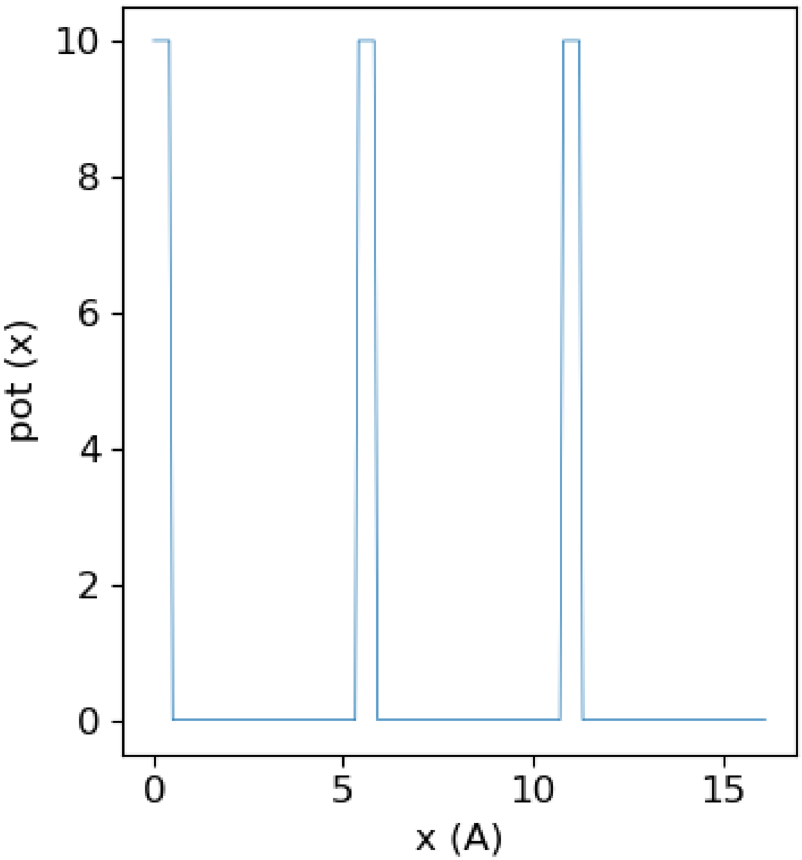
Lattice parameter (Si) a = 5.4064 Å m* = 1.0m_e

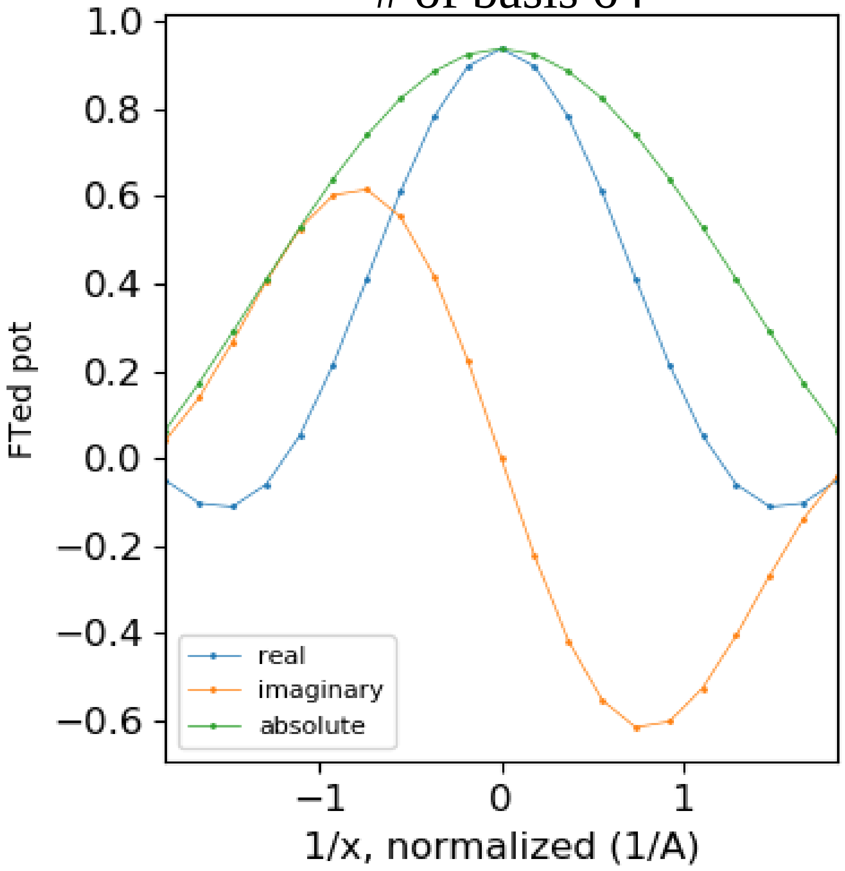
Potential *V*(*x*): barrier width 0.5 Å barrier height 10.0 eV

```
python pw1d.py ft 5.4064 64 rect 0.5 10.0 9 -0.5 0.5 21
```

FT coefficinets of potential

# of basis 64





Program: 1-D PW method

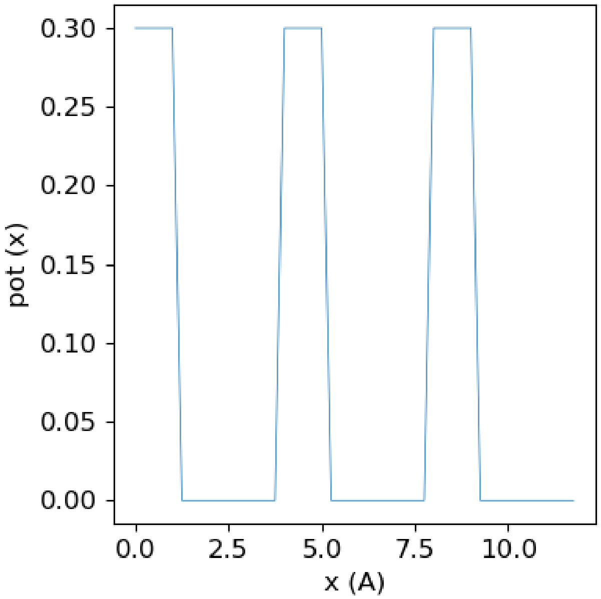
pw1.py

```
python pw1d.py ft 5.4064 64 rect 0.5 10.0 9 -0.5 0.5 21
python pw1d.py band 5.4064 64 rect 0.5 10.0 3 -0.5 0.5 21
python pw1d.py wf 5.4064 64 rect 0.5 10.0 3 0.0 0 0.0 16.2192 101
```

a = 4.0 Å

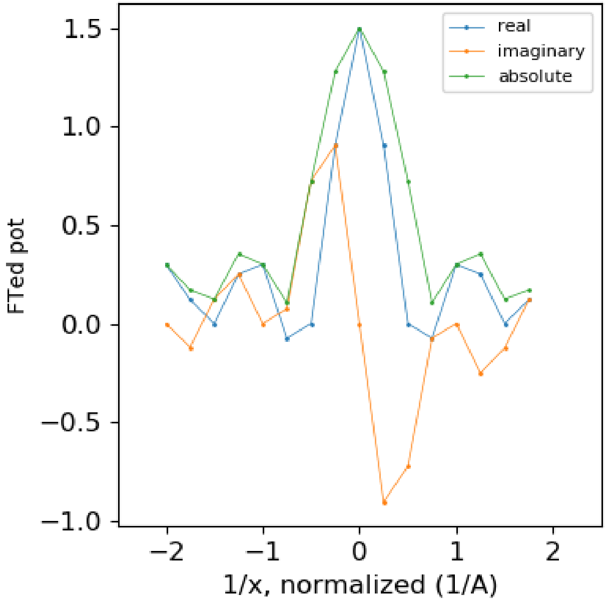
potential $V(x)$:

w = 1.0 Å, h = 0.3 eV



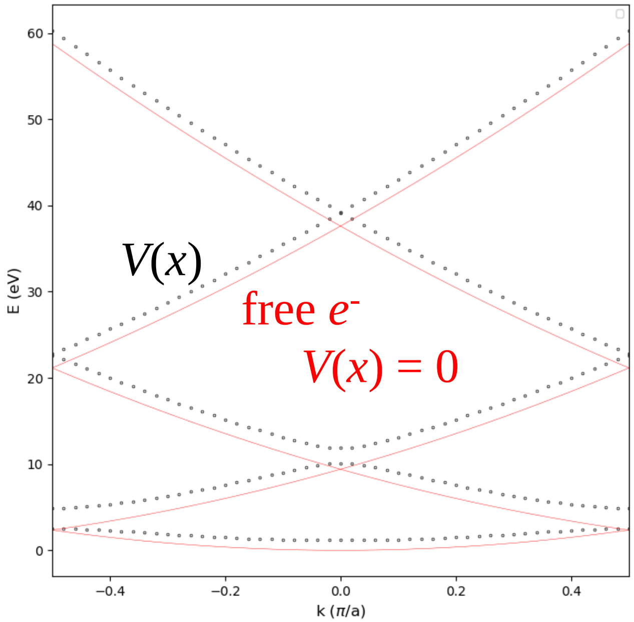
FT coefficients of potential

# of basis 16



Band structure

# of basis 5



$V(x)$

free e^-

$V(x) = 0$

Analytical DFT XC calculation
Transfer matrix method

図10-2 H原子の波動関数

Hartree-Fock (HF) 方程式

$$\left\{-\frac{1}{2}\nabla^2 - \frac{Z}{r} + \underline{\int \frac{\rho(\mathbf{r}_m)}{|\boldsymbol{r}_m - \mathbf{r}|} d\mathbf{r}_m - \int \frac{\rho(\mathbf{r}_m)}{|\boldsymbol{r}_m - \mathbf{r}|} d\mathbf{r}_m}\right\}\varphi(\mathbf{r}) = \varepsilon\varphi(\mathbf{r})$$

自己相互作用 (Self-interaction: SI) は HF 法では相殺される

Slater’s Xα (DFT)

$$\left\{-\frac{1}{2}\nabla^2 - \frac{Z}{r} + \underline{\int \frac{\rho(\mathbf{r}_m)}{|\boldsymbol{r}_m - \mathbf{r}|} d\mathbf{r}_m - 3\alpha\left\{\frac{3}{4\pi}\rho(\mathbf{r})\right\}^{1/3}}\right\}\varphi(\mathbf{r}) = \varepsilon\varphi(\mathbf{r})$$

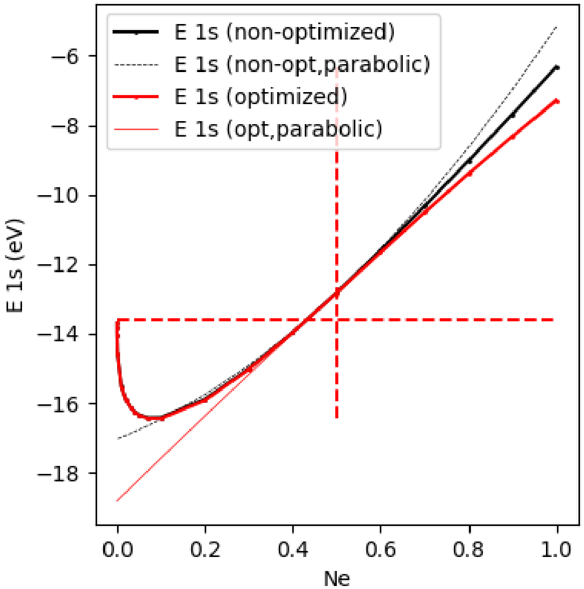
DFTでは SI は相殺されず、誤差として残る

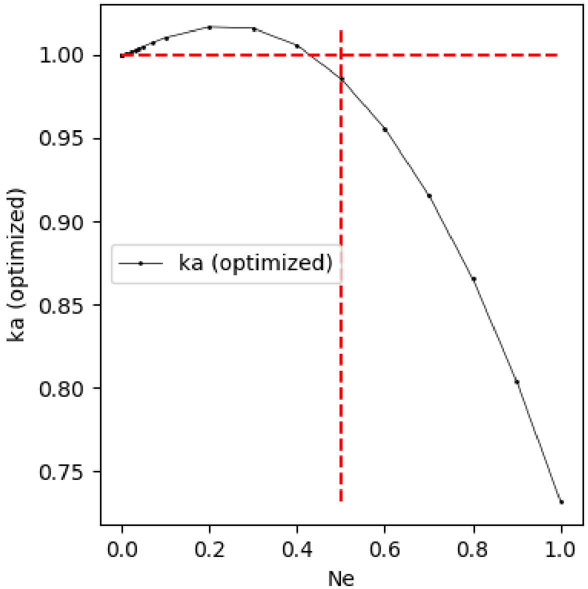
H1s-HF-LDA.py

1s軌道内の電子数 N_e を変化

α = 2/3

厳密解: E(1s) = -13.6 eV





p.16

図10-2 H原子の波動関数

http://conf.msl.titech.ac.jp/jsap-crystal/

DFTの自己相互作用誤差: HF近似とLDAによる水素原子1s 軌道

Usage: python H1s-HF-LDA.py mode Z ka Ne

実行例1: python H1s-HF-LDA.py ng 1.0 1.0 1.0

ka = 1.0 (HFの H 1s 軌道の指数関数の係数) での
1s 軌道準位の電子数 Ne を 0 ~ 1 と変化させてプロット

実行例2: python H1s-HF-LDA.py nvg 1.0 1.0 1.0

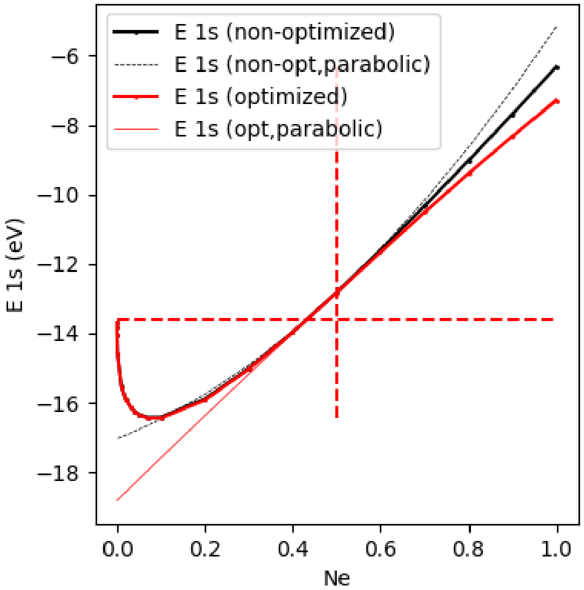
実行例1に、kaを変分原理で最適化させた結果を追加

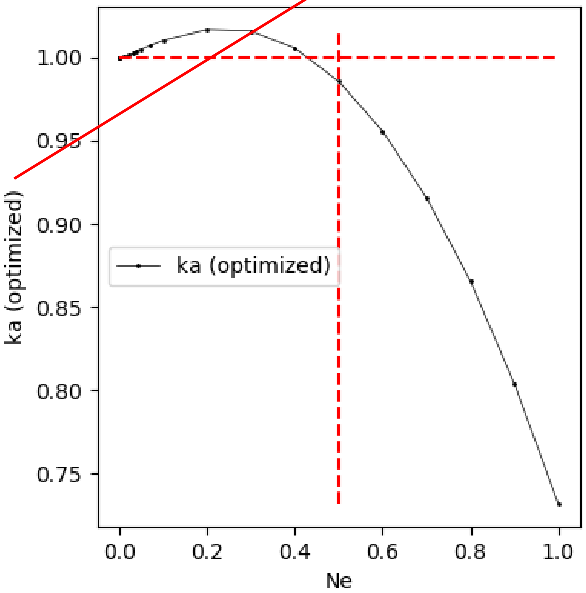
$$R_{1s}(r) = 2a_0^{-3/2}\exp\left(-k_a\frac{1}{2}\frac{2}{a_0}r\right)$$

python H1s-HF-LDA.py nvg 1.0 1.0 1.0

1s軌道内の電子数 N_e を
変化

厳密解: E(1s) = -13.6 eV



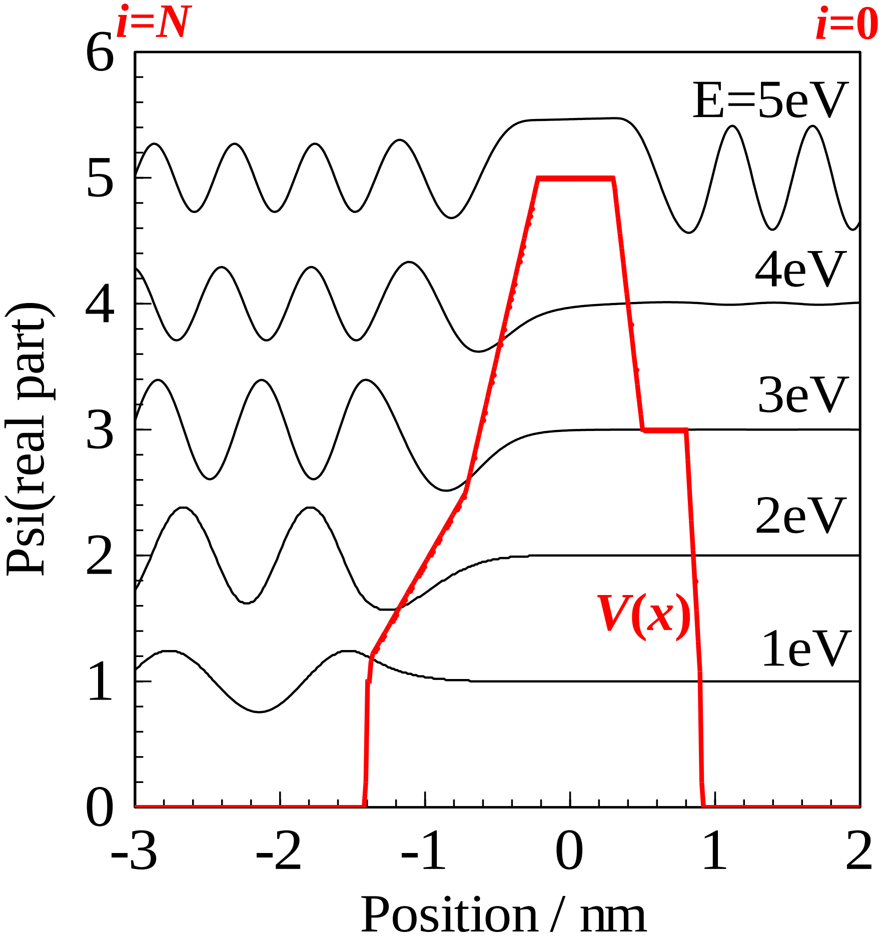


平面波近似: 転送行列法

H. Mizuta, T. Tanoue, “The Physics and Applications of Resonant Tunnelling Diodes,” Cambridge Univ Press (1995)
Y. Ando and A. Itoh, J. Appl. Phys. 61 (1987) 1497

$$\Psi_i(x) = A_i \exp(ik_i x) + B_i \exp(-ik_i x)$$

$$k_i = \sqrt{\frac{2m_i}{\hbar^2}(E - V_i)}$$



境界条件

$$\Psi_i(x_{i+1}) = \Psi_{i+1}(x_{i+1})$$

$$m_i^{-1}\Psi'_i(x_{i+1}) = m_{i+1}^{-1}\Psi'_{i+1}(x_{i+1})$$

$$\begin{pmatrix} A_{i+1} \\ B_{i+1} \end{pmatrix} = \begin{pmatrix} \alpha^{+}{}_i P_i & \alpha^{-}{}_i / Q_i \\ \alpha^{-}{}_i Q_i & \alpha^{+}{}_i / P_i \end{pmatrix} \begin{pmatrix} A_i \\ B_i \end{pmatrix}$$

$$\alpha^{\pm}{}_i = \frac{1}{2}\left[1 \pm (m_{i+1}/m_i)(k_i/k_{i+1})\right]$$

$$P_i = \exp[i(k_i - k_{i+1})x_{i+1}]$$

$$Q_i = \exp[i(k_i + k_{i+1})x_{i+1}]$$

平面波近似: 転送行列法

H. Mizuta, T. Tanoue, “The Physics and Applications of
Resonant Tunnelling Diodes,” Cambridge Univ Press (1995)

$$\begin{pmatrix} A_N \\ B_N \end{pmatrix} = \begin{pmatrix} \alpha^{+}{}_{N-1} P_{N-1} & \alpha^{-}{}_{N-1} / Q_{N-1} \\ \alpha^{-}{}_{N-1} Q_{N-1} & \alpha^{+}{}_{N-1} / P_{N-1} \end{pmatrix} \begin{pmatrix} A_{N-1} \\ B_{N-1} \end{pmatrix} = T_{N-1} \begin{pmatrix} A_{N-1} \\ B_{N-1} \end{pmatrix} = T_{N-1} T_{N-2} \begin{pmatrix} A_{N-2} \\ B_{N-2} \end{pmatrix} = T \begin{pmatrix} A_0 \\ B_0 \end{pmatrix}$$

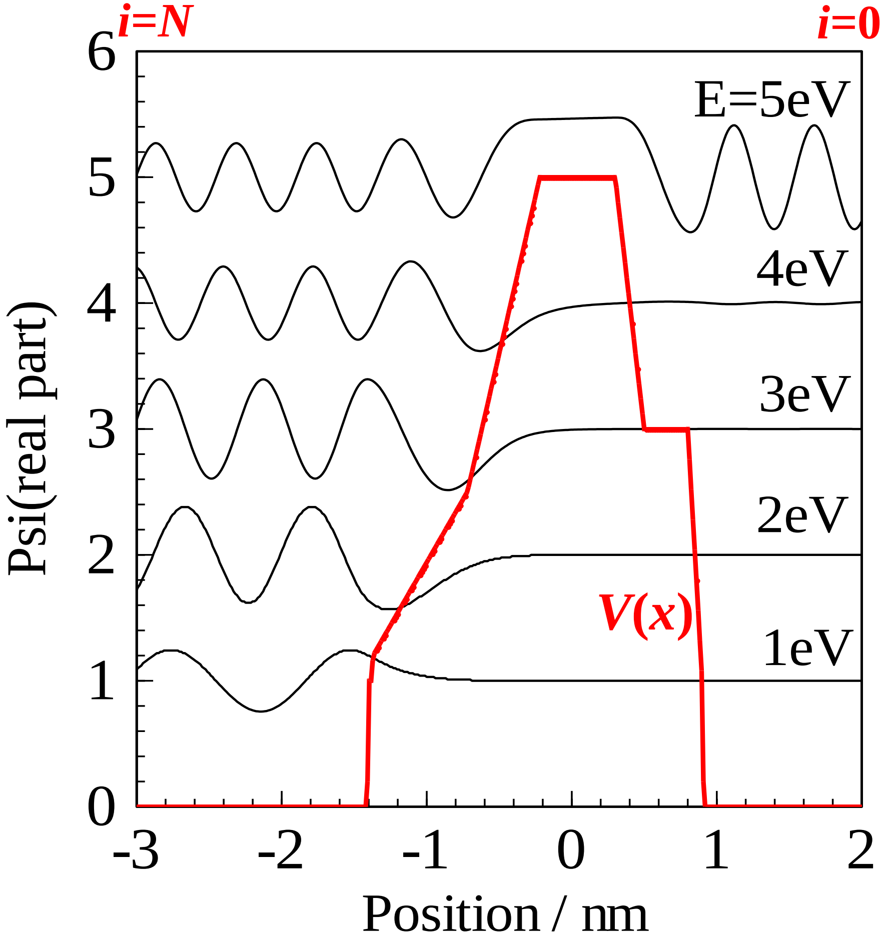
$$T = T_{N-1} T_{N-2} \cdots T_0$$

境界条件例:

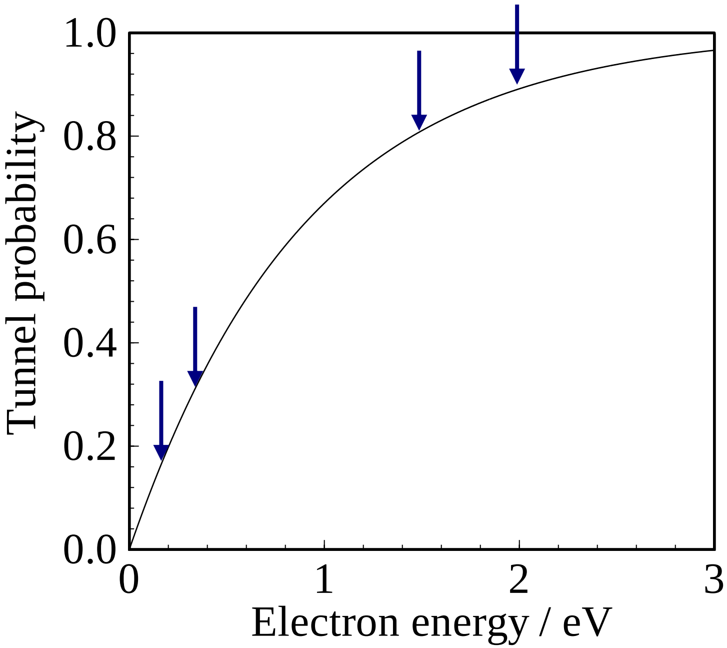
放出側 (*i* = 0) では

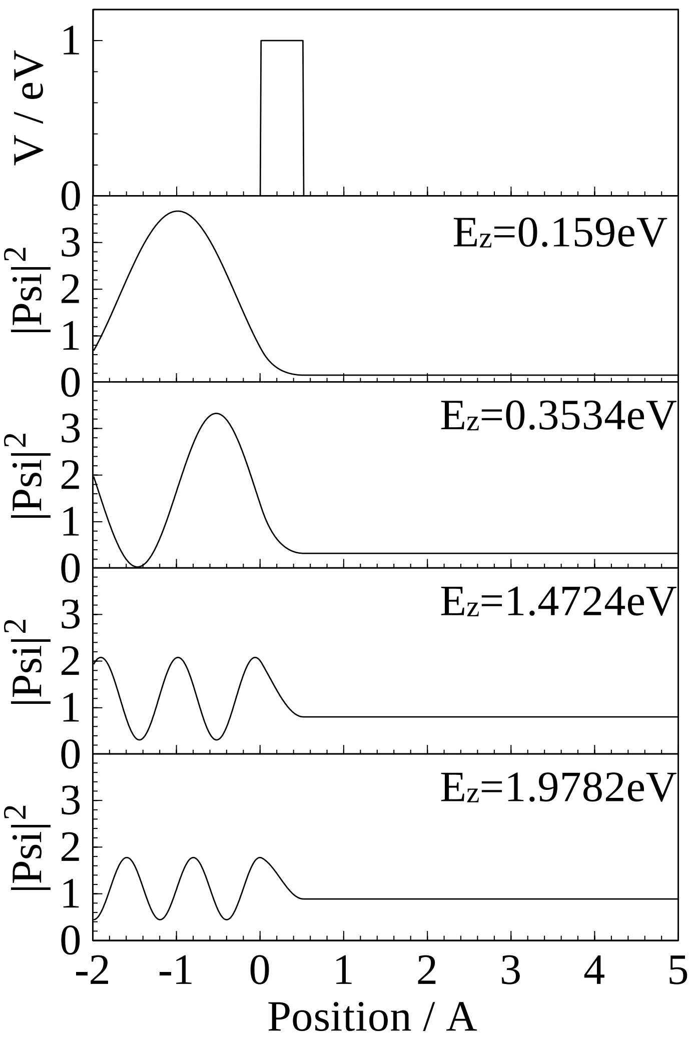
進行波のみが残る

$A_0 = 1$, $B_0 = 0$



1枚の障壁のトンネル

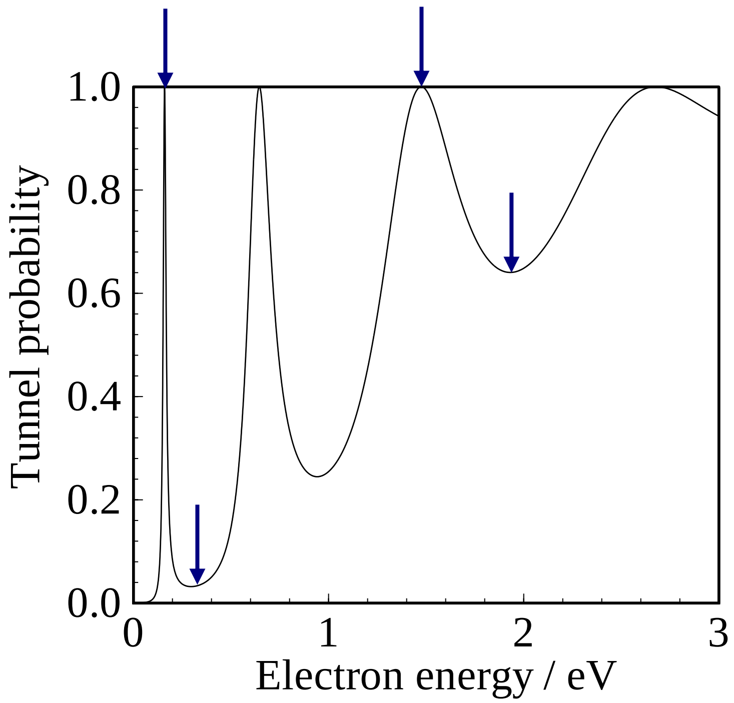


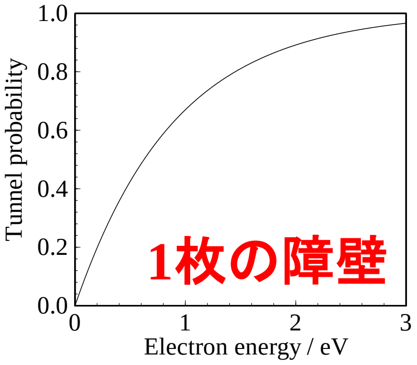


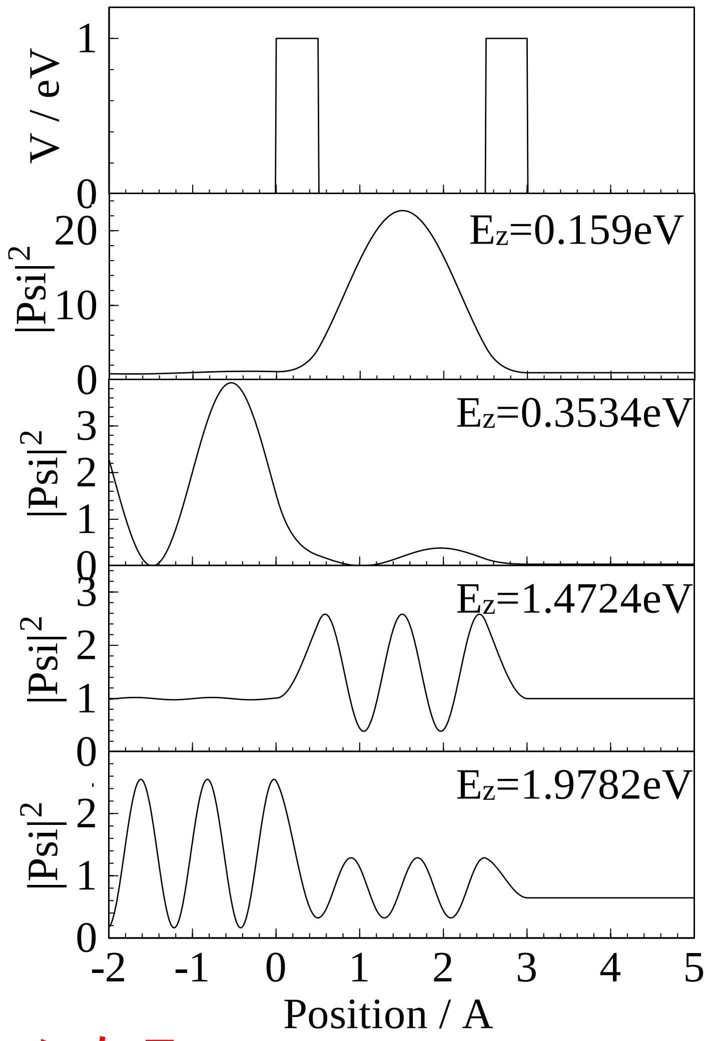
原子 (障壁) による散乱で、透過率は必ず 1 より小さい

=> 原子がたくさんあったら、透過率は 0 になる？

2枚の障壁のトンネル(QW, RTD)





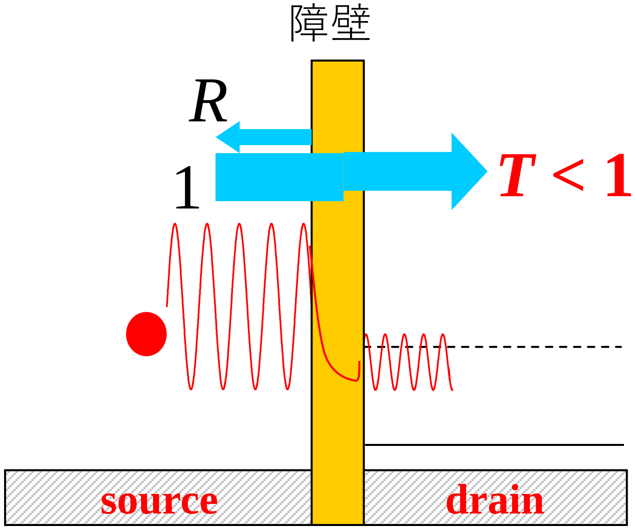


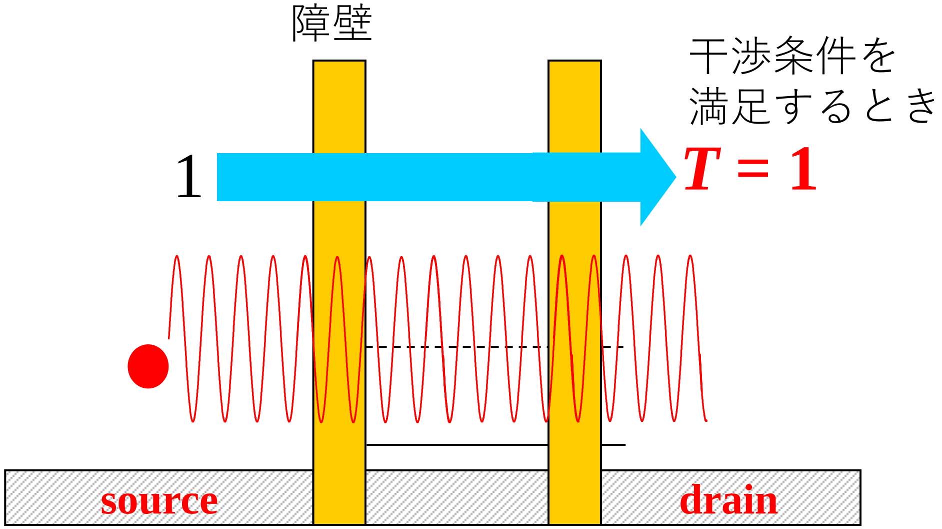
原子がたくさんあったら、透過率は 0 になる？

=> 原子 (障壁) が 2つ以上あれば、特定のエネルギーで 100% 透過する

電子と光の散乱

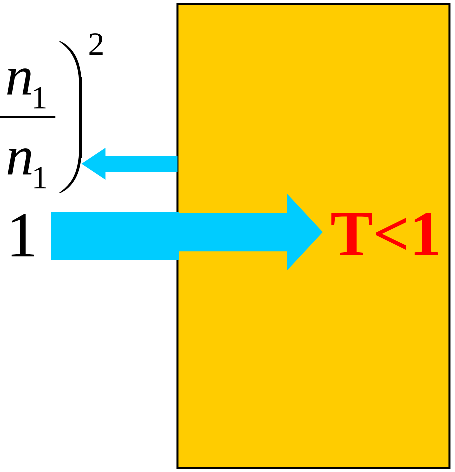
電子の透過と反射





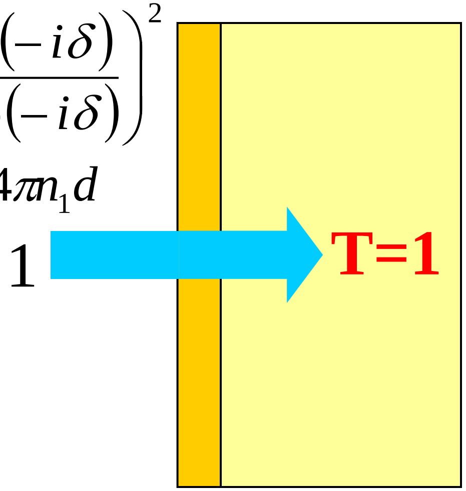
光の透過と反射

$$R = \left(\frac{n_2 - n_1}{n_2 + n_1}\right)^2$$

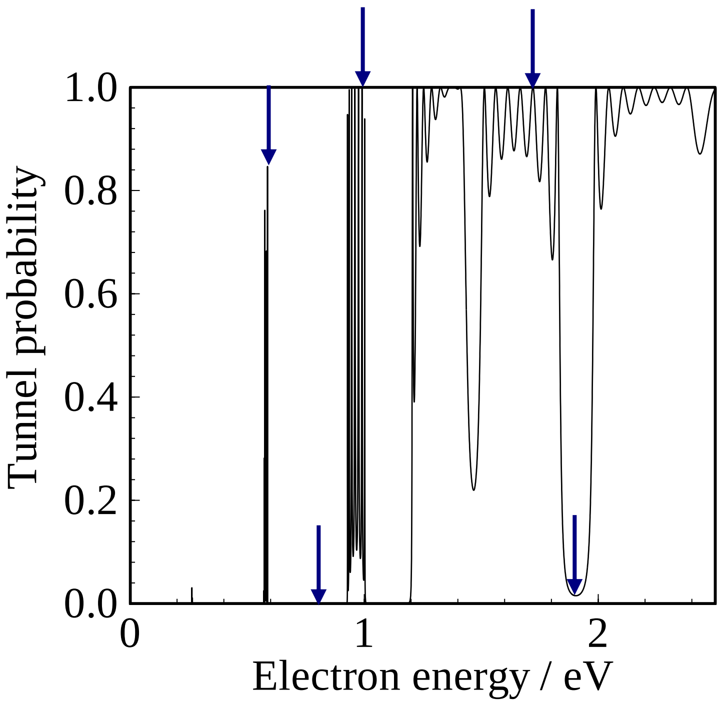


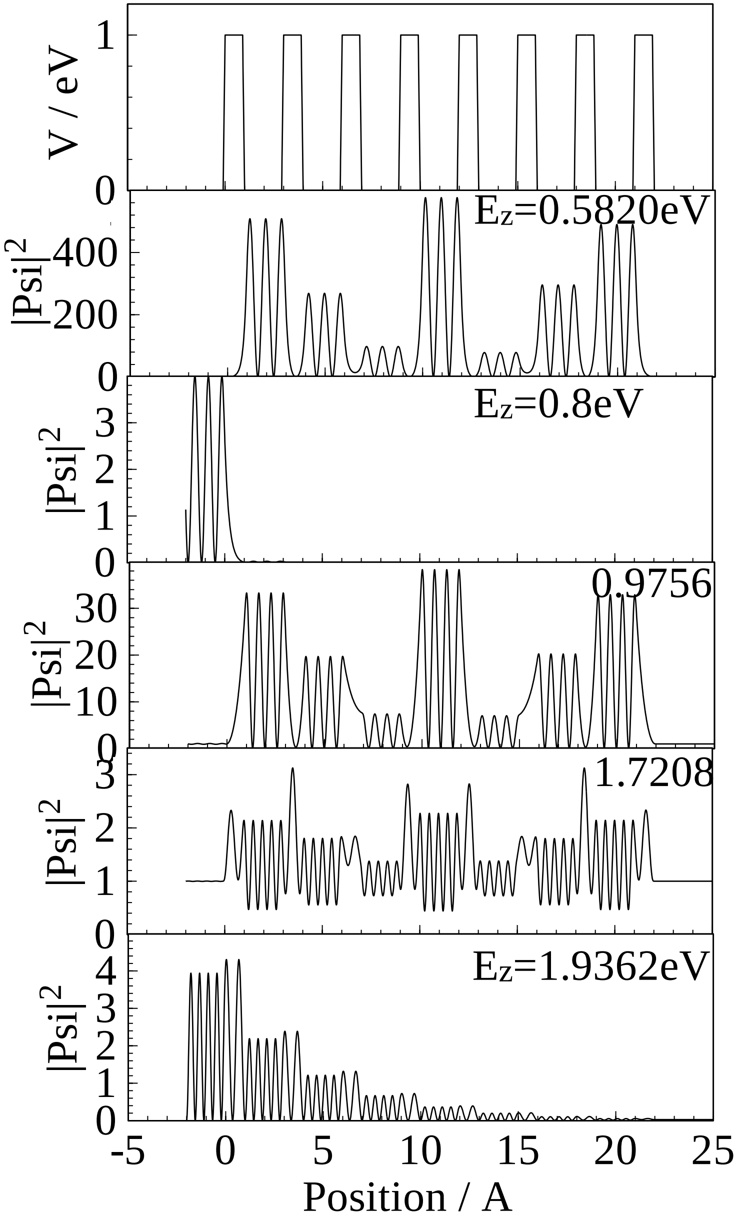
$$R = \left(\frac{r_1 + r_2 \exp(-i\delta)}{1 + r_1 r_2 \exp(-i\delta)}\right)^2$$

$$\delta = 4\pi n_1 d$$



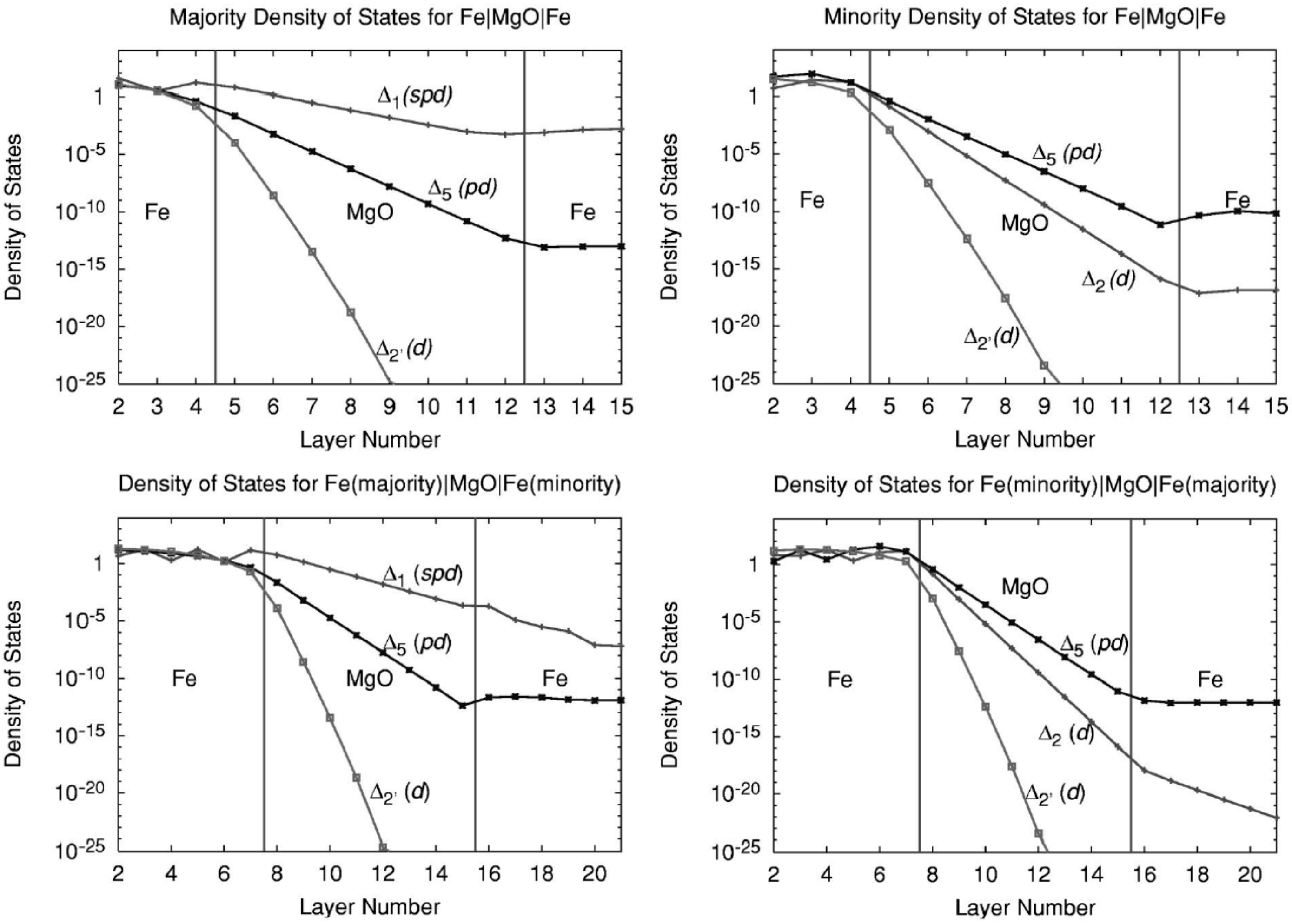
多重量子井戸 (MQW) の透過: バンド





Fe/MgO/Fe TMR素子のスピン依存透過率

W.H. Butler, X.-G. Zhang and T.C. Schulthess, Spin-dependent tunneling conductance of Fe|MgO|Fe sandwiches



Majority Density of States for Fe|MgO|Fe

Minority Density of States for Fe|MgO|Fe

Density of States for Fe(majority)|MgO|Fe(minority)

Density of States for Fe(minority)|MgO|Fe(majority)

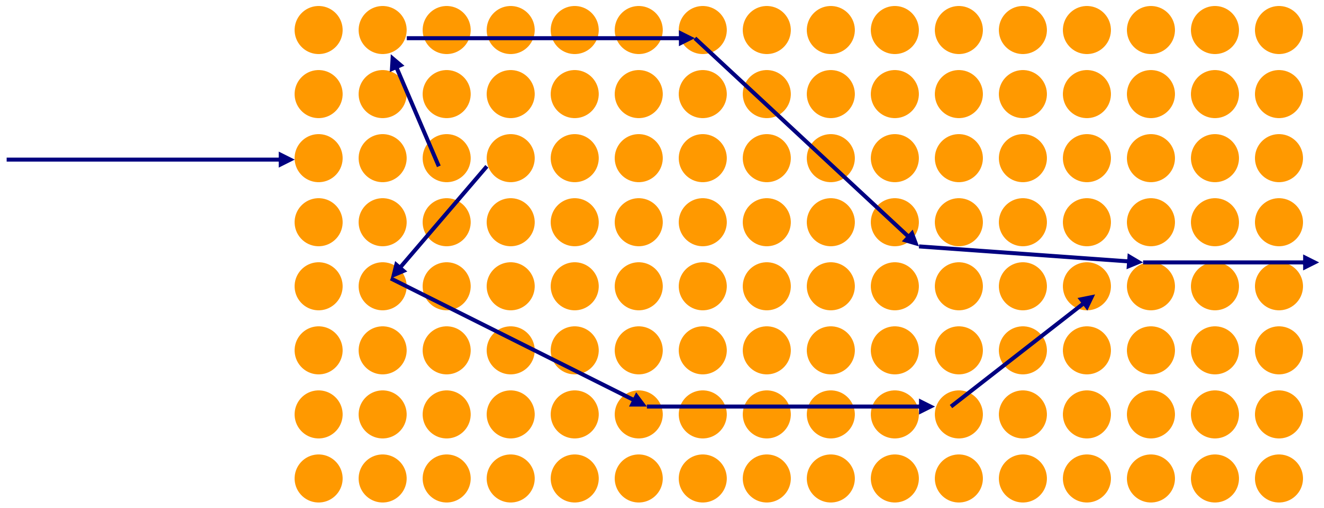
FIG. 7. Tunneling DOS for $k_{\parallel}=0$ for Fe(100)|8MgO|Fe(100). The four panels show the tunneling DOS for majority (upper left) minority (upper right), and antiparallel alignment of the moments in the two electrodes (lower panels). Additional Fe layers are included in the lower panels to show the TDOS variation in the Fe. Each TDOS curve is labeled by the symmetry of the incident Bloch state in the left Fe electrode.

結晶における電子の透過

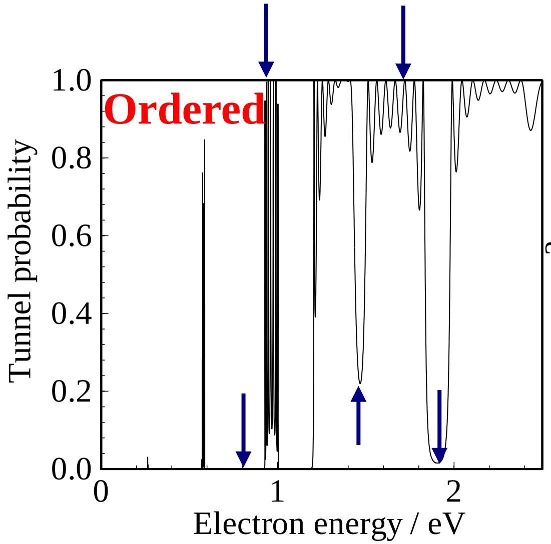
- 電子が結晶を透過できる (T = 1) のは、
  三次元に配列した原子からの散乱波が干渉する結果

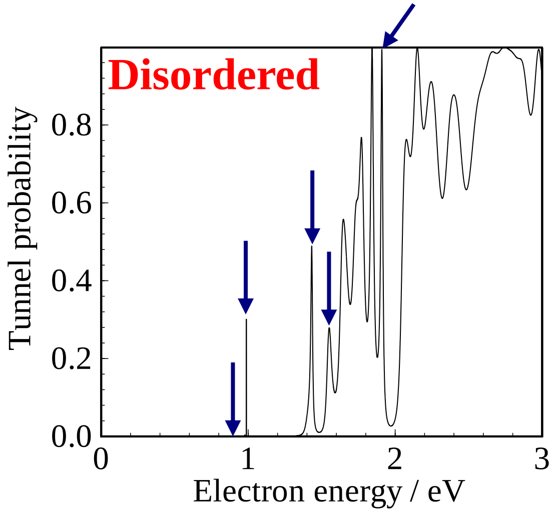
- バンド構造は、透過できる状態のみを表示

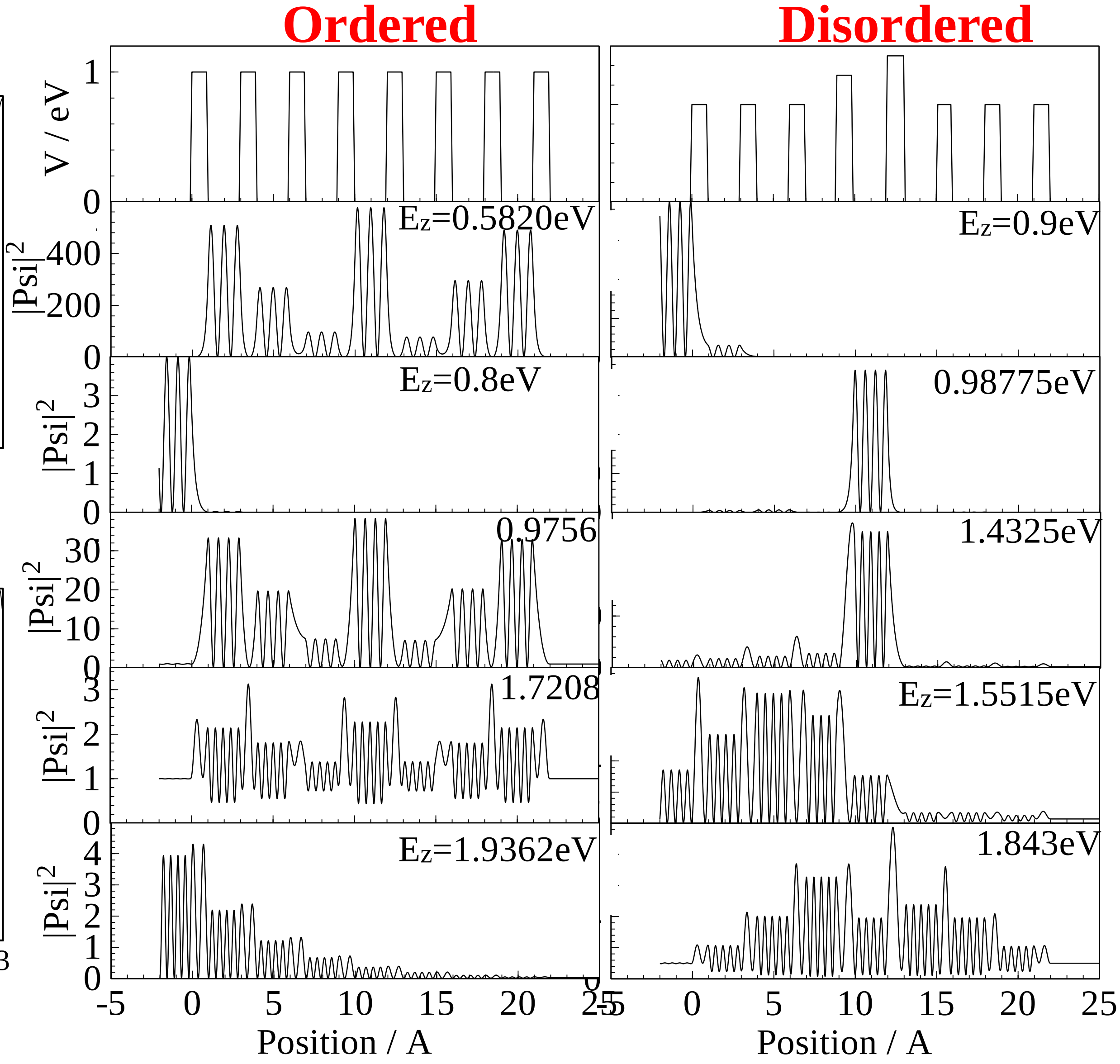
- 任意の運動エネルギーにおいて状態は存在する
  ただし、そのほとんどは減衰（散乱）を伴う



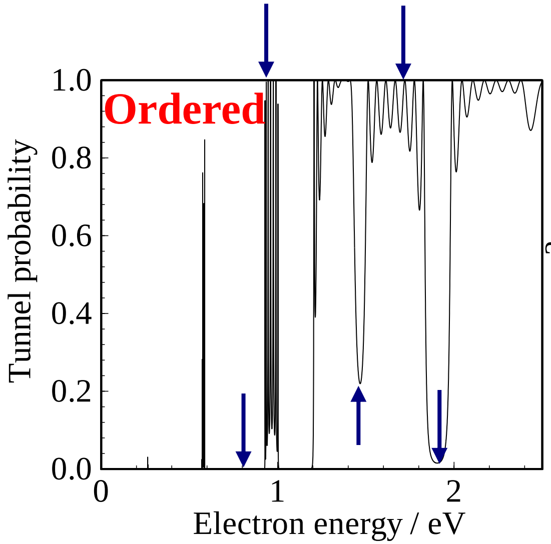
欠陥のある多重量子井戸における電子の透過

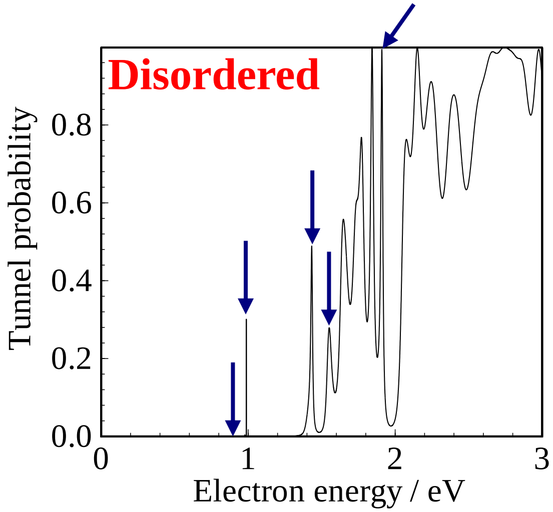


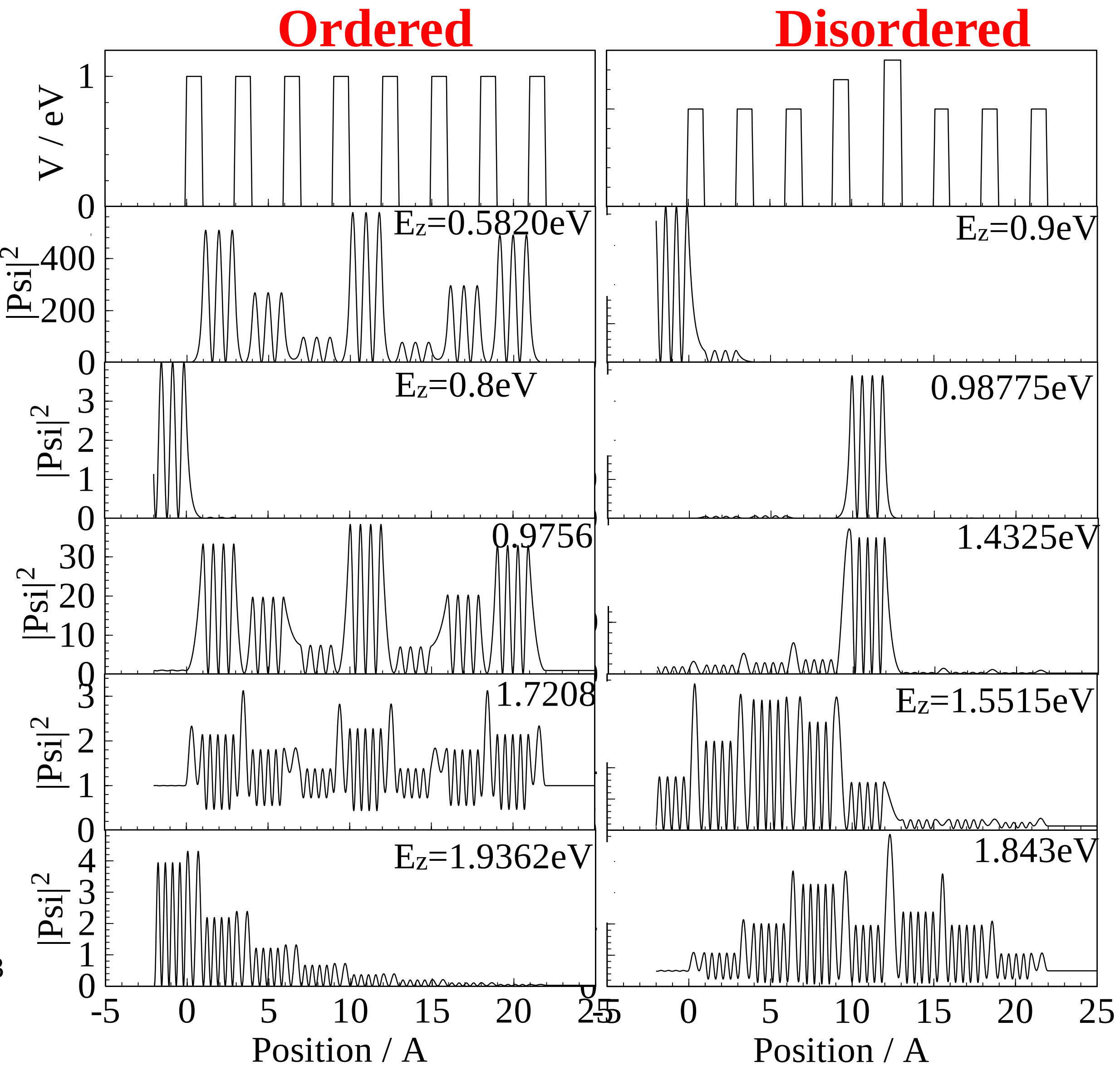




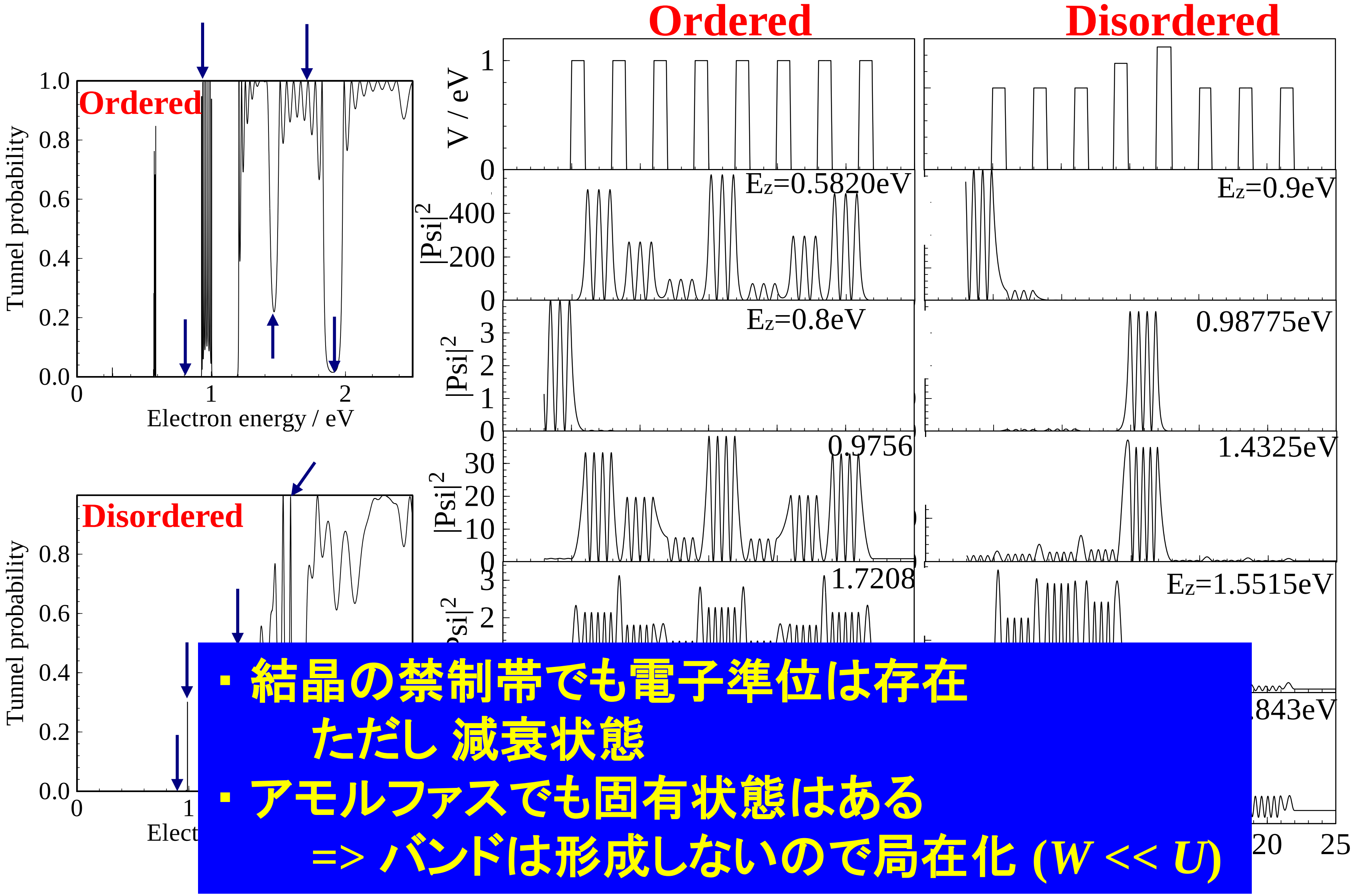
欠陥のある多重量子井戸における電子の透過







欠陥のある多重量子井戸における電子の透過



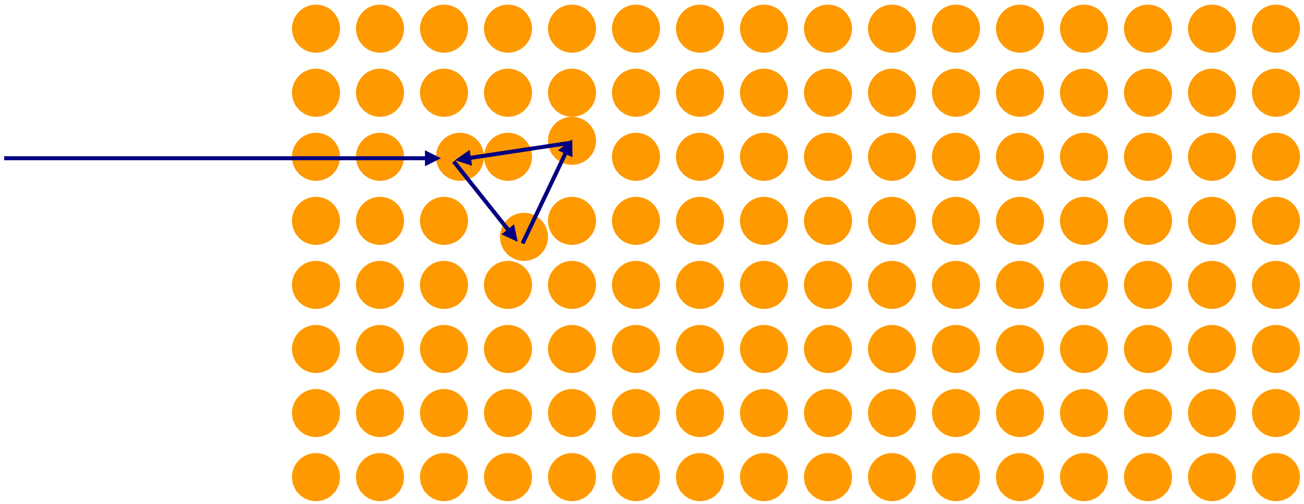
・結晶の禁制帯でも電子準位は存在

ただし 減衰状態

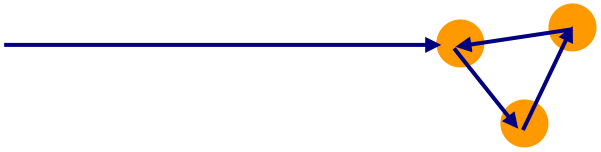
・アモルファスでも固有状態はある

=> バンドは形成しないので局在化 (*W* << *U*)

乱れのある結晶における電子の透過



- 背景の結晶部分は電子の透過だけに寄与するので
  差分だけ考える



- 乱れた構造による散乱と干渉の結果、定在波をつくる
  アンダーソン局在

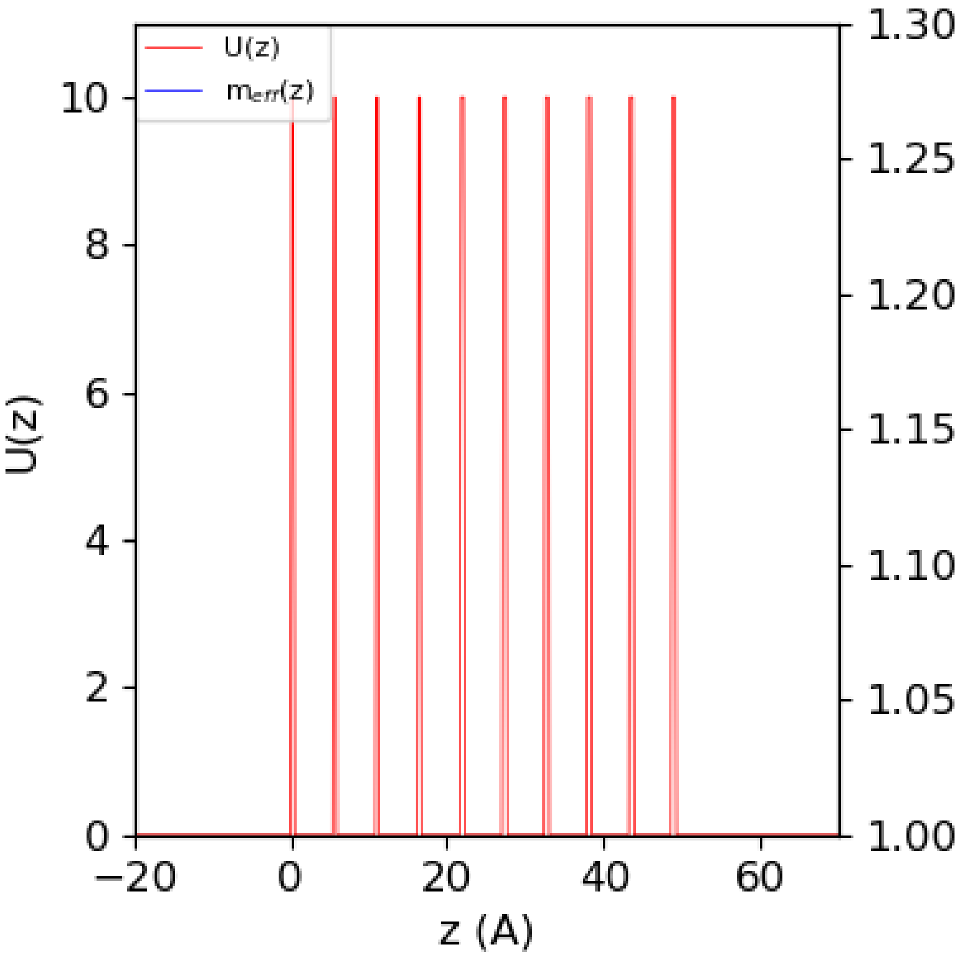
プログラム: 転送行列法

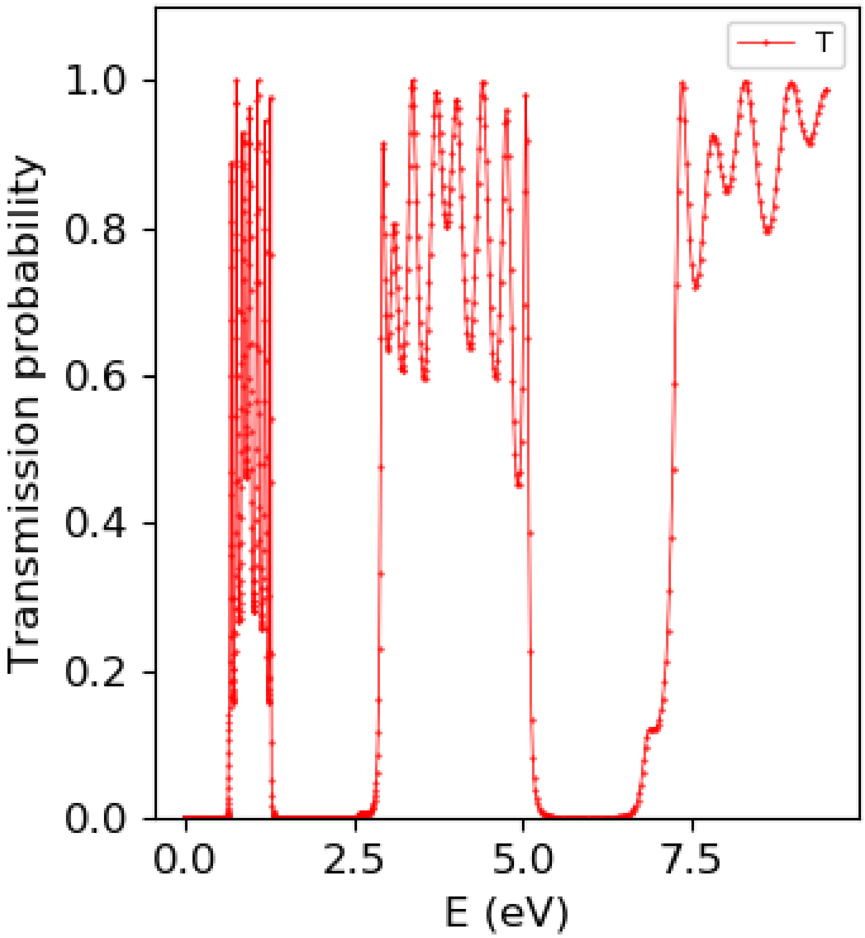
Transfer_matrix.py

Si の格子定数　a　= 5.4064 Å　　$m^* = 1.0m_e$

障壁幅　　0.5 Å　　障壁高さ 10.0 eV　10周期

```
python transfer_matrix.py tr 501 0.1 0.01 9.5 2001
```





プログラム: 転送行列法

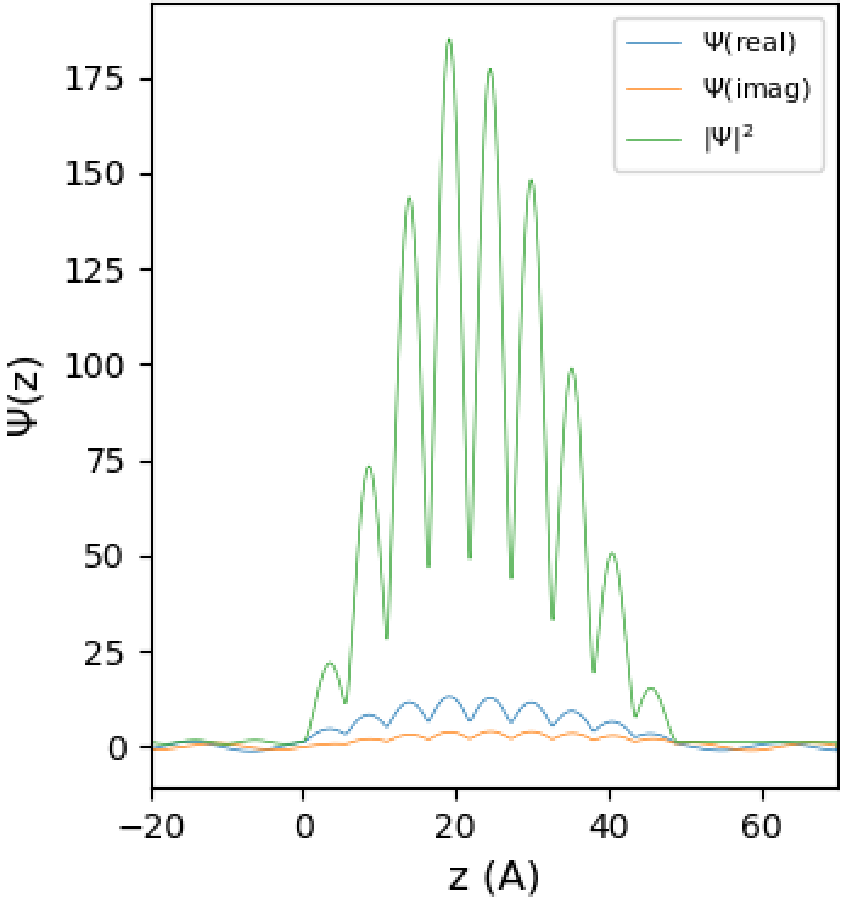
Transfer_matrix.py

Si の格子定数 a = 5.4064 Å m* = 1.0m_e

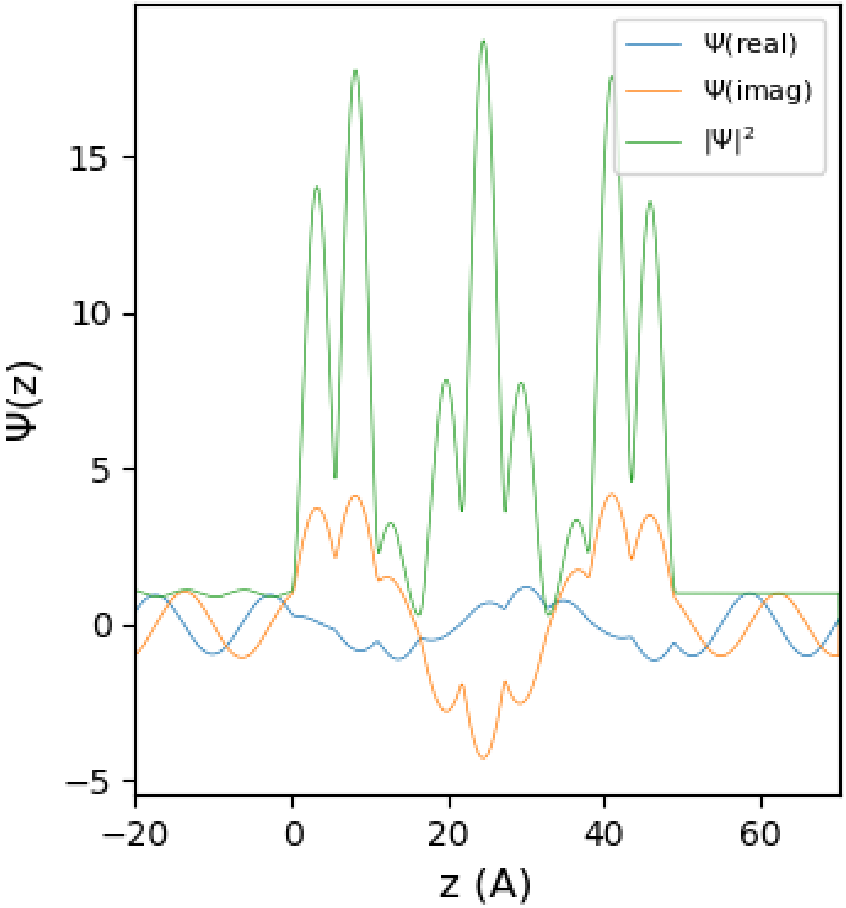
障壁幅 0.5 Å 障壁高さ 10.0 eV 10周期

```
python transfer_matrix.py wf 5001 Ez
```

Ez = 0.61371 eV



Ez = 0.701645 eV



常微分方程式の境界値問題: Thomas-Fermiモデル

後藤憲一 他, 詳解現代物理学演習、共立出版 (1972)

φ(r): 遮蔽された原子核ポテンシャル

$$\begin{aligned} \phi(r) \to & \frac{1}{4\pi\varepsilon_0}\frac{Ze}{r} \quad (r \to 0) \\ & 0 \qquad\qquad\quad (r \to \infty) \end{aligned}$$

規格化

$$\chi(r) = \frac{4\pi\varepsilon_0}{Ze} r\varphi(r) = \frac{4\pi\varepsilon_0}{Ze} r\left(E_F / e + \phi(r)\right)$$

$$b = Z^{-1/3}\left(\frac{3\pi}{4}\right)^{2/3}\frac{a_0}{2}$$

$$r = by = 0.8853 Z^{-1/3} a_0 y$$

$$a_0 = \frac{4\pi\varepsilon_0\hbar^2}{me^2} = 0.52921 \text{ Å}$$

電子密度による近似 (Thomas-Fermiモデル)

$$y^{1/2}\frac{d^2\chi}{dy^2} = \chi^{3/2}$$

$$\begin{aligned} \chi(r) \to & 1 \;\; (r \to 0) \\ & 0 \;\; (E_F = 0, r \to \infty) \end{aligned}$$

常微分方程式の境界値問題: Thomas-Fermiモデル

後藤憲一 他, 詳解現代物理学演習、共立出版 (1972)

$$\frac{d^2\chi}{dy^2} = y^{-1/2}\chi^{3/2} \qquad \chi_{n+1} - 2\chi_n + \chi_{n-1} = h^2\chi''_n + O(h^4) = h^2 y_n^{-1/2}\chi_n^{3/2} + O(h^4)$$

$$\chi_{n+1} = 2\chi_n - \chi_{n-1} + h^2 y_n^{-1/2}\chi_n^{3/2}$$

初期条件: $\chi_0 = 1 \qquad \chi_1 = 1-\alpha$

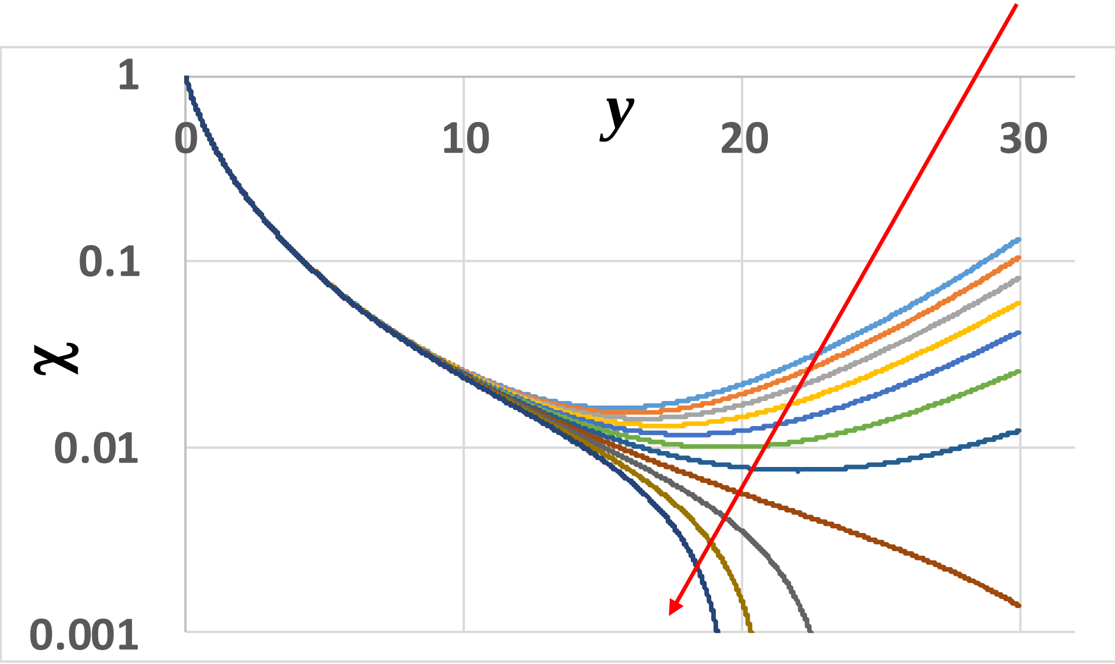
境界条件:

$\chi_n > 0, \ |\chi_n| < \mathrm{EPS}$

$\chi_n' < 0, |\chi_n'| < \mathrm{EPS'}$

$\alpha = 0.01442860$

~ 0.01442869



常微分方程式の境界値問題: ノイメロフ積分

菅野暁 監修, 足立裕彦、塚田 著, スレーター分子軌道計算,第3章
東京大学出版会 (1982)

原子のSchrödinger方程式の動径関数 (Rydberg単位)

$$-\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{dR(r)}{dr}\right)+\left[-\varepsilon+V(r)+\frac{l(l+1)}{r^2}\right]R(r)=0 \qquad \lim_{r\to 0,\infty} R(r)=0$$

$$P(r)=rR(r)$$

$$\frac{d^2}{dr^2}P(r)=g(r)P(r)$$

$$g(r)=-\varepsilon+V(r)+\frac{l(l+1)}{r^2}$$

$$=-\varepsilon-2\frac{Z}{r}+\frac{l(l+1)}{r^2}$$

$$P_{n+1}-2P_n+P_{n-1}=h^2P''_n+O(h^4)=h^2g_nP_n+O(h^4)$$

$$P_{n+1}=(2+h^2g_n)P_n-P_{n-1}$$

中央の式までは
Verlet法

ノイメロフ (Noumerov) 積分:

$$y_n=P_n-\frac{h^2P''_n}{12}=P_n\left(1-\frac{h^2g_n}{12}\right)$$

として次の式を使うと、さらに精度が上がる

$$y_{n+1}=\left(2+\frac{h^2g_n}{1-h^2g_n/12}\right)y_n-y_{n-1}+O(h^6)$$

常微分方程式の境界値問題: 波動関数

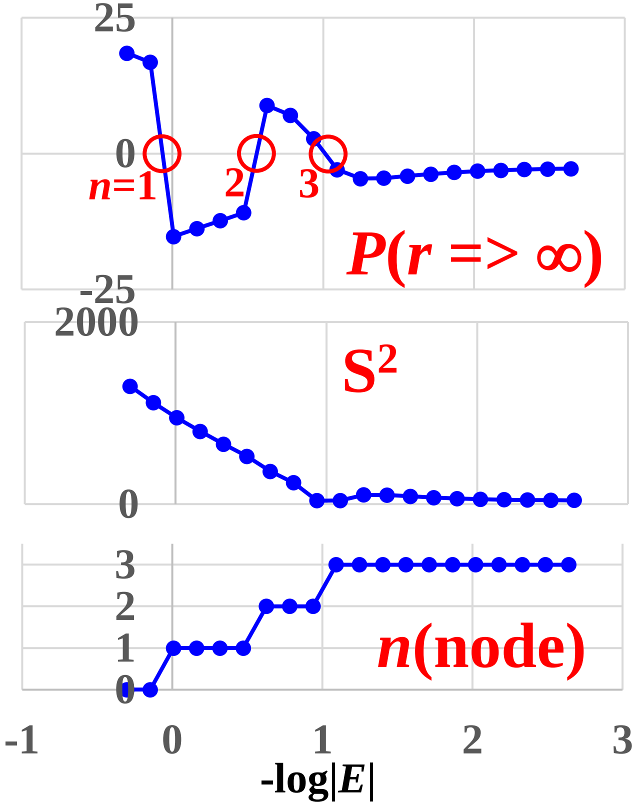
菅野暁 監修, 足立裕彦、塚田 著, スレーター分子軌道計算,第3章
東京大学出版会 (1982)

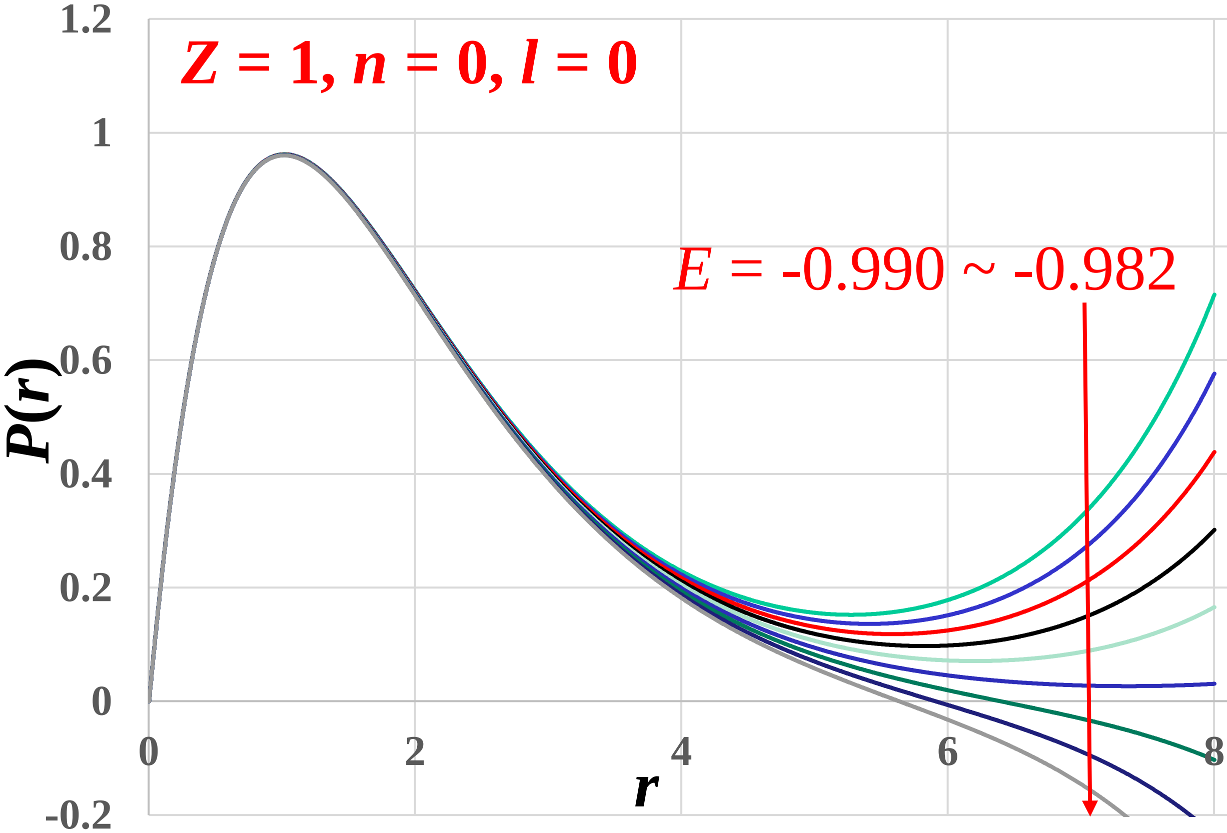
$$P(r) = rR(r)$$

$$P_{n+1} = (2 + h^2 g_n) P_n - P_{n-1} \qquad g_n = -E - 2\frac{Z}{r_n} + \frac{l(l+1)}{r_n^{\ 2}}$$

初期条件: $P_0 = 0,\ P_1 = \alpha$

境界条件: $\lim\limits_{r \to 0,\infty} P(r) = 0$





常微分方程式の境界値問題: 波動関数

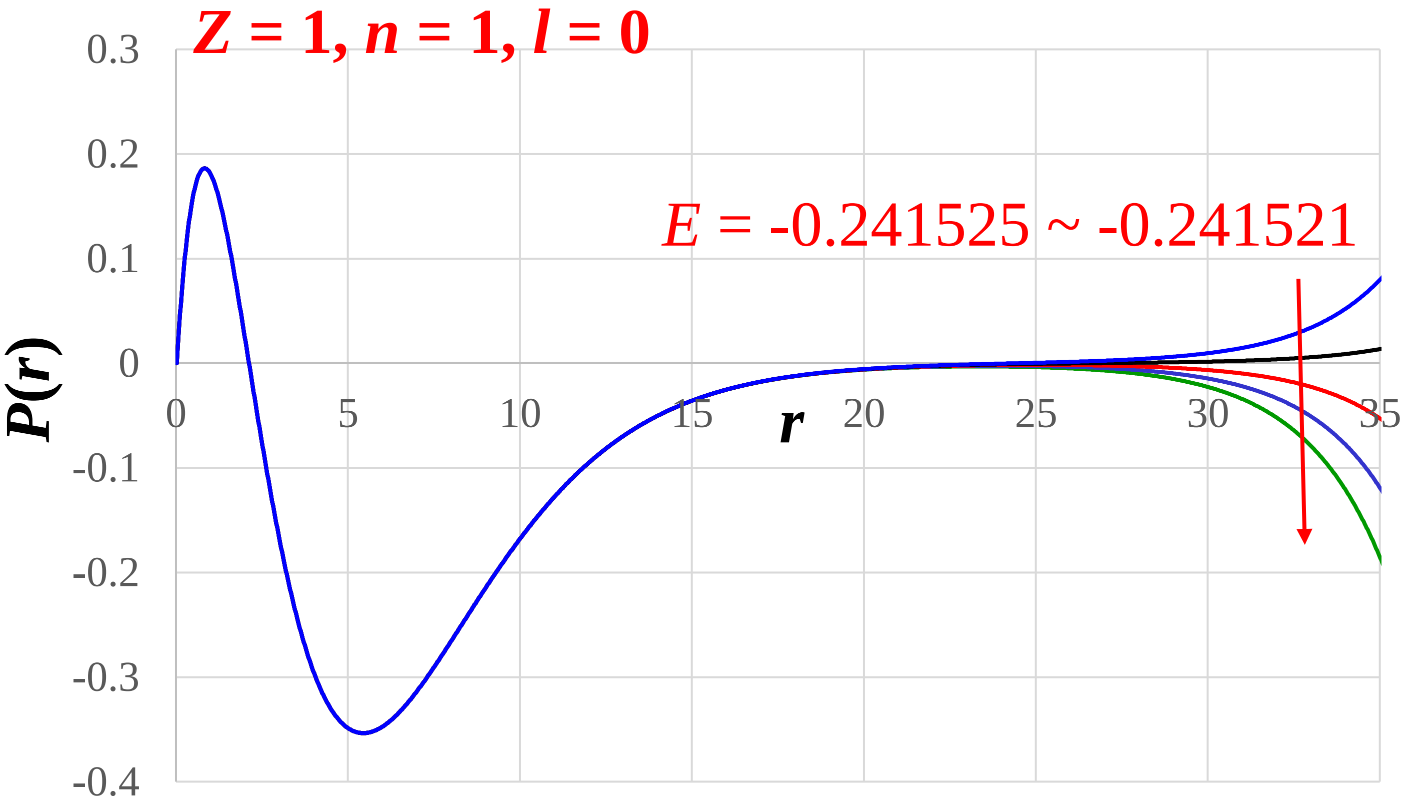
菅野暁 監修, 足立裕彦、塚田 著, スレーター分子軌道計算,第3章
東京大学出版会 (1982)

$$P(r) = rR(r)$$

$$P_{n+1} = (2 + h^2 g_n)P_n - P_{n-1} \qquad g_n = -E - 2\frac{Z}{r_n} + \frac{l(l+1)}{r_n^{\ 2}}$$

初期条件: $P_0 = 0,\ P_1 = \alpha$

境界条件: $\lim\limits_{r\to 0,\infty} P(r) = 0$


Z = 1, n = 1, l = 0
E = -0.241525 ~ -0.241521
P(r)
r
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
5
10
15
20
25
30
35