

Computational Materials Science (計算材料学特論)

Lecture materials updated (this morning, 8:24)

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

2024年度Q2 計算材料学特論 (資料: 英語 + 日本語版)

Computational Materials Science 2024 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 for numerical analyses (by Kamiya)

講義で使うプレゼン資料は

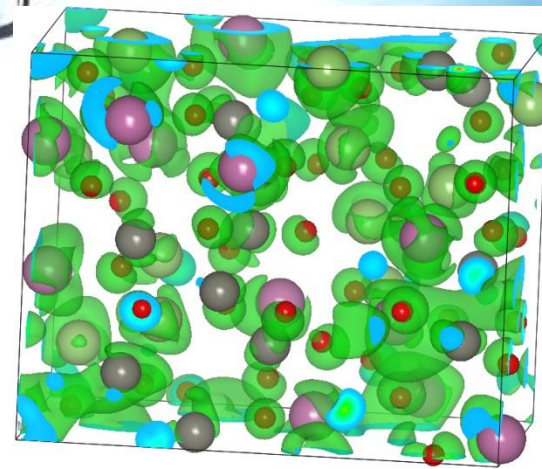
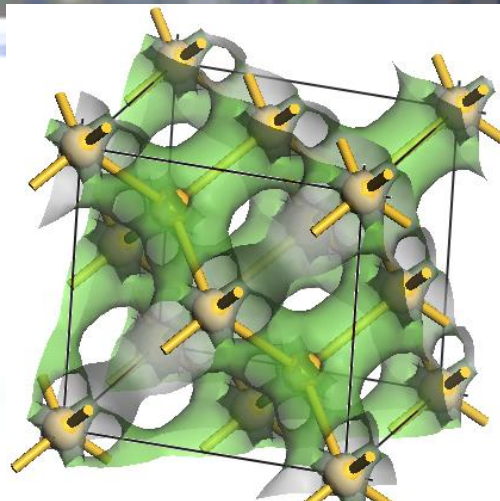
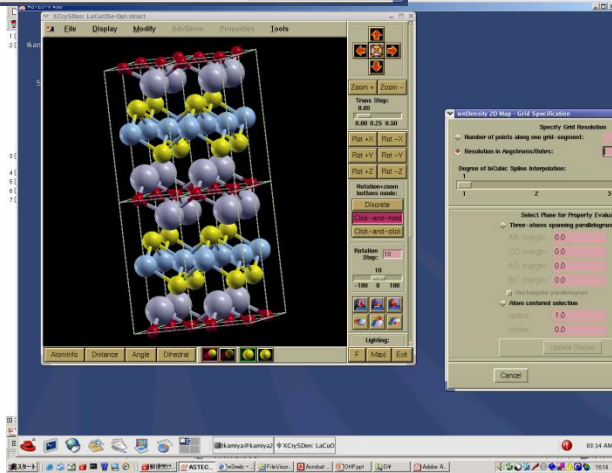
共通して使うpythonプログラム」の下にあります

Lecture presentation slides are found after the "Common python programs" section below.

Update News:

- June 28, 8:24 Lecture materials on June 28 have been updated ([20240628FT_Matrix2.zip](#))
- June 27, 10:09 Lecture materials on June 28 have been uploaded ([20240628FT_Matrix.zip](#))
- June 27, 9:25 Lecture materials on June 25 have been updated ([20240627LSQEquationOptimize.zip](#))
- June 25, 12:51 Lecture materials on June 25 have been updated
- June 24, 10:40 Lecture materials on June 25 have been updated
- June 23, 10:12 Lecture materials on June 25 have been uploaded
- June 23, 9:32 Lecture materials on June 21 have been updated ([20240621InterporlateSmoothing.zip](#))

神谷利夫



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 11 (Tue) Kamiya (Fundamental of computer, Sources of errors (コンピュータの基礎、誤差))
- #02 June 14 (Fri) Kamiya (Numerical differentiation/integration (数値微分/積分))
- #03 June 18 (Tue) Kamiya (Numerical integration (数値積分),
Differential equation (微分方程式), Molecular dynamics (分子動力学法))
- #04 June 21 (Fri) Kamiya (Interpolation (補間), Smoothing (平滑化), Linear least-squares method (線形最小二乗法))
- #05 June 25 (Tue) Kamiya (Numerical solutions of equations (方程式の数値解法),
Nonlinear optimization (非線形最適化))
- #06 June 28 (Fri) Kamiya (Nonlinear optimization (非線形最適化)), Fourier transformation (フーリエ変換), Matrix (行列))
- July 2 (Tue) No lecture (休講)**
- #07 July 5 (Fri) Kamiya, Review (復習)
- #08 July 9 (Tue) Sasagawa (Review of quantum theory 1: 量子論おさらい1)
- #09 July 12 (Fri) Sasagawa (Review of quantum theory 2: 量子論おさらい2)
- #10 July 16 (Tue) Sasagawa (First principles calculations: basics 1 第一原理計算: 基礎1)
- #11 July 19 (Fri) Sasagawa (First principles calculations: basics 2 第一原理計算: 基礎2)
- #12 July 23 (Tue) Sasagawa (First principles calc.: applications 1 第一原理計算: 応用1)
- #13 July 26 (Tue) Sasagawa (First principles calc.: applications 2 第一原理計算: 応用2)
- #14 Sasagawa (Classical and Quantum Computers 古典および量子コンピュータ)

Evaluation (Kamiya)

- **Small quiz**
Not evaluate correctness of the answers
but consider how you answered them
- **Term-end assignment**
Problems will be given at the end of Q2
from T2SCHOLAR

Explanation of the answers, June 25

課題解答の解説

PROBLEM, June 25

- Submit electronic file(s) via T2SCHOLAR in 2 days
(If T2SCHOLAR doesn't work, send the files to kamiya.t.aa@m.titech.ac.jp.
In this case, file name must include your STUDENT ID and FULL NAME)

PROBLEM:

Solve $5\cos(x) - x = 0$.

- Plot the functions $y = 5\cos(x)$ and $y = x$ in the range $x = 0 - 3$, find an initial x for Newton-Raphson method.
- Solve $5\cos(x) - x = 0$ by Newton-Raphson method at least with four significant digits.
- Optional: Propose if you have any other numerical analysis you want to learn in Computational Materials Science
- Optional: Propose if you have any python program (should be simple) you want to learn

PROBLEM, June 25

- Submit electronic file(s) via T2SCHOLAR in 2 days
(If T2SCHOLAR doesn't work, send the files to kamiya.t.aa@m.titech.ac.jp.
In this case, file name must include your STUDENT ID and FULL NAME)

PROBLEM:

Solve $5\cos(x) - x = 0$.

- Plot the functions $y = 5\cos(x)$ and $y = x$ in the range $x = 0 - 3$, find an initial x for Newton-Raphson method.
- Solve $5\cos(x) - x = 0$ by Newton-Raphson method at least with four significant digits.

Newton-Raphson method:

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

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

See equation_answer.xlsx

PROBLEM, June 25

- (i) **Optional: Propose if you have any other numerical analysis you want to learn in Computational Materials Science**
- (ii) **Optional: Propose if you have any python program (should be simple) you want to learn**

- 最適化手法全般について前回の授業以上に学びたい。おすすめの書籍を知りたい
- 数値解析を勉強する上でのおすすめの参考書

Textbooks

- 自分で何かしらのデータをフィッティングするプログラムを書く上での注意点とおすすめの方法 (scipy optimize?)
データ例: XRDピーク、光の干渉によるフリンジ、エネルギーバンド

How to make fitting program

- 測定データを解析するためにベイズ最適化を用いたプログラムをpythonのライブラリのoptunaを用いて作成していますが、

Bayesian optimization

- tetrahedron法について(基礎理論と使うメリット、どのような場面で利用されるか)

3D numerical integration

- モンテカルロ法に関連する(特にising modelなど) 事

Monte Carlo method: for integration, stochastic simulations, optimization

- 回帰で用いられる正則化について(基礎的な内容)

Machine learning regression / sparse modelling / parameter reduction

Will be explained on July 5th

PROBLEM, June 25: Textbooks

- 最適化手法全般について前回の授業以上に学びたい。おすすめの書籍を知りたい
- 数値解析を勉強する上でのおすすめの参考書

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++

**Numerical analysis, Data analysis, Machine learning cover huge different areas.
No textbook satisfies you.**

**Go to big bookstores, and check the contents
Or ask me by specifying your interesting topics**

A: Text books

Machine learning, data processing

下山輝昌

株式会社Iroribi 代表取締役。日本電気株式会社(NEC)の中央研究所にてハードウェアの研究開発に従事した後、独立。機械学習を活用したデータ分析やダッシュボードデザイン等に裾野を広げ、データ分析コンサルタントとして幅広く案件に携わる。それと同時に、最先端にはテクノロジーとビジネスの接点、IoT、AI、データサイエンス、クラウド、セキュリティ、ブロックチェーン、量子コンピューティング、宇宙技術、環境技術、社会課題解決、未来社会の構築に貢献したい。

Graduated Tokyo Tech, the former materials science course, now operating a data analysis company

機械学習を活用したデータ分析やダッシュボードデザイン等に裾野を広げ、データ分析コンサルタントとして幅広く案件に携わる。それと同時に、最先端にはテクノロジーとビジネスの接点、IoT、AI、データサイエンス、クラウド、セキュリティ、ブロックチェーン、量子コンピューティング、宇宙技術、環境技術、社会課題解決、未来社会の構築に貢献したい。

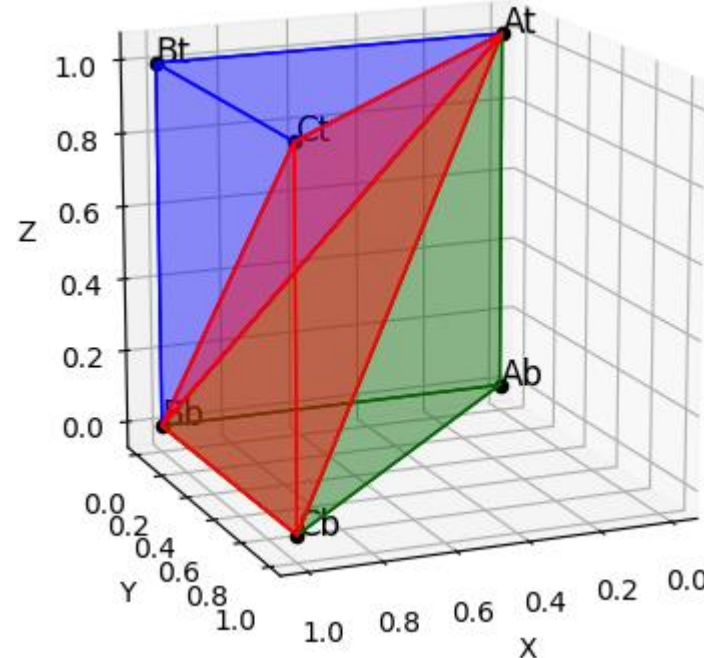
PROBLEM, June 25: Tetrahedron method

Used to perform the first Brillouin Zone integration for band calculations

- A method for 3D numerical integration for function $E(x, y, z)$
- Divide 3D space to parallelepipeds (平行六面体)
- Divide a parallelepiped to two triangular prisms (三角柱)
- Divide a triangular prisms to three tetrahedrons
- Normalize the vertexes to x, y and z to be in $[0, 1]$
- **Liner interpolation** by $E(x, y, z) = E_{000} + (E_{100} - E_{000})x + (E_{010} - E_{000})y + (E_{001} - E_{000})z$
A general method for multi-dimensional numerical integration (Finite Element Method etc)
- Integrate $E(x, y, z)$ in the tetrahedron

tetrahedron.py:

**How to divide a triangular prism to
three tetrahedrons**



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

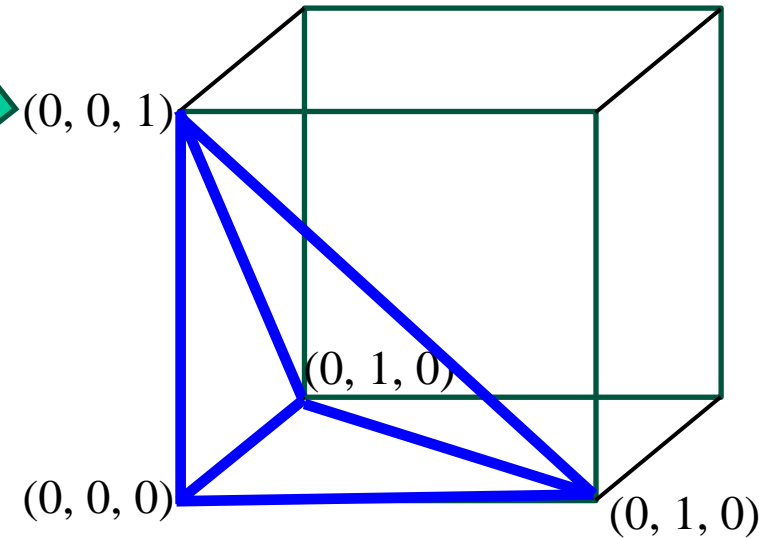
3. Interpolate by

$$\begin{aligned} E(\mathbf{k}) = & E_{000} \\ & + (E_{100} - E_{000})x \\ & + (E_{010} - E_{000})y \\ & + (E_{001} - E_{000})z \end{aligned}$$

, where E_{ijk} is $E(x, y, z)$ at a vertex (i, j, k)

4. Integrate $E(x, y, z)$ in the tetrahedron

$$0 \leq x, y, z \leq 1 \text{ and } 0 \leq x + y + z \leq 1$$



PROBLEM, June 25: Bayesian optimization

See (Japanese) tutorial course:

<http://conf.msl.titech.ac.jp/D2MatE/2022Tutorial/tutorial2022.html>

材料計算科学・データ解析チュートリアルコース 2022年度

2022年度チュートリアルコースは終了いたしました。

チュートリアル資料の更新は、下記の4/2 14:04更新と録画公開で終了しました。

配布プログラムの更新は、[Topページ](#)にて行います。

-
- 2023/4/2 14:04 第5回チュートリアル (3月22日) 講義資料 最終更新版: [20230322Tutorial.zip](#)
2023/4/1 注: Arrhenius plot機能について、活性化エネルギーの計算に間違いがあったものを修正しました。
 - 2023/3/22 第1～5回チュートリアル録画 2023年3月いっぱいので予定で公開: [2022年度チュートリアル録画](#)

チュートリアル資料 再構成版: チュートリアルの資料をテーマごとにまとめなおしました。

ファイル名が日本語になっています。ダウンロードができない場合、tkamiya@msl.titech.ac.jp までご連絡

- [01-注意.pdf](#)
- [02-Launcher.pdf](#)
- [02-python-tips.pdf](#)
- [03-強化学習プログラムbayes_gp_gui.pyの使い方.pdf](#)
- [04-線形最適化・最小二乗法\(回帰\).pdf](#)
- [05-機械学習\(回帰\).pdf](#)
- [06-非線形最適化・最小二乗法.pdf](#)
- [07-強化学習\(ガウス過程回帰ベイズ推定\)の基礎.pdf](#)
おまけ: [07'-ベイズ統計.pdf](#)
- [08-スペクトル解析.pdf](#)

材料計算科学・データ解析チュートリアルコース 2022年度

文部科学省「データ創出・活用型マテリアル研究開発プロジェクト」[半導体拠点 D2MatE](#) では、
材料計算科学・データ解析に関するチュートリアルコースを開催いたします。

PROBLEM, June 25: Monte Carlo simulation

Monte Carlo methods are used for various purposes: for integration, stochastic simulations, optimization

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

Uniform random and Pseudorandom numbers

- It is not easy to generate “random” phenomenon

=> Generate pseudorandom numbers by algorithm

- Product congruence method (乗積合同法): a, b, L are positive integers

$$N_1 = a$$

$$N_2 = bN_1 \bmod L$$

$N \bmod L$ is the remainder of N divided by L

$$N_3 = bN_2 \bmod L$$

...

produces pseudorandom numbers N in $0 \leq N \leq L - 1$

- Mixed congruence method (混合合同法): a, b, L are positive integers

$$N_1 = a$$

$$N_2 = bN_1 + c \bmod L$$

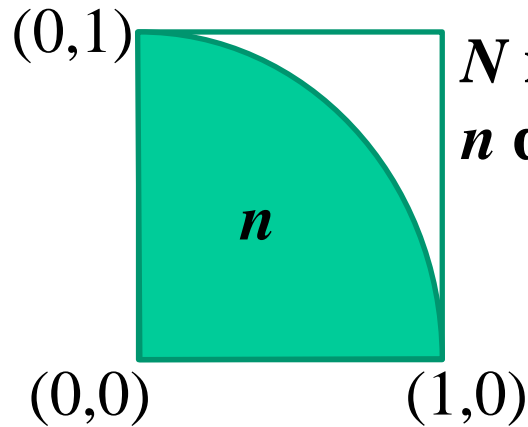
$$N_3 = bN_2 + c \bmod L$$

...

* NOTE: $N_k = N_m$ results in periodicity in the generated numbers

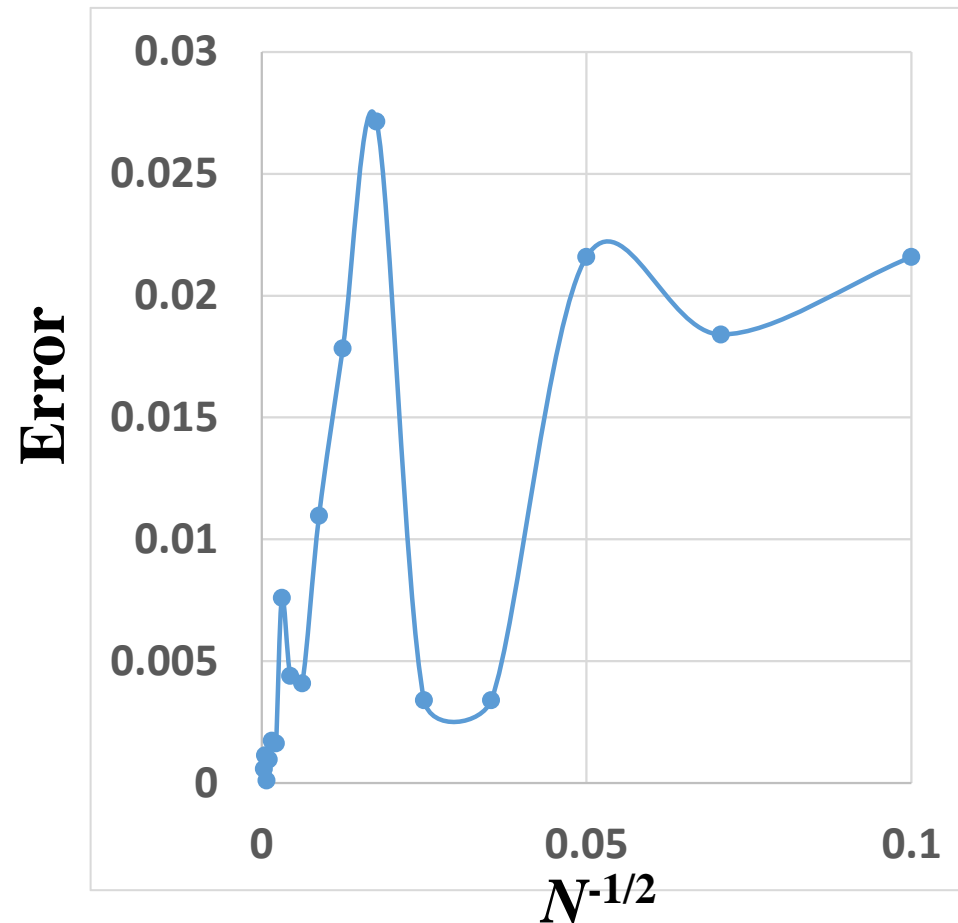
Hit-or-miss (試行錯誤的) Monte Carlo method

1. Generate random numbers (x, y) N times
2. Count the number that satisfies $(x^2 + y^2)^{1/2} < 1.0 \Rightarrow n/N$ approaches the area of a quarter circle



N numbers distributes over the square
 n drops in the quarter circle

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



Crude (基礎的) Monte Carlo method

Generate random numbers $0 \leq r < 1$ N times

$S = \int_0^1 f(x)dx \sim \frac{1}{N} \sum_{i=1}^N f(x_i)$ is approximated

$$f(x) = 4\sqrt{1-x^2}$$

Numerical integration by random numbers

Good for multi-dimensional integration

Ex: Discrete Variational X α method

N	Hit-or-miss	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

integ_montecarlo3d.py

Calculate the volume of radius 1.0 sphere: Exact value 4.188790205

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 $\propto 1/N$

Random numbers that follows exponential distribution

http://www.sat.t.u-tokyo.ac.jp/~omi/random_variables_generation.html#Gauss

$$p(x; \lambda) = \lambda \exp(-\lambda x) \quad (\text{平均 } 1/\lambda, \text{ 分散 } 1/\lambda^2)$$

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

$$x = -\log(y)$$

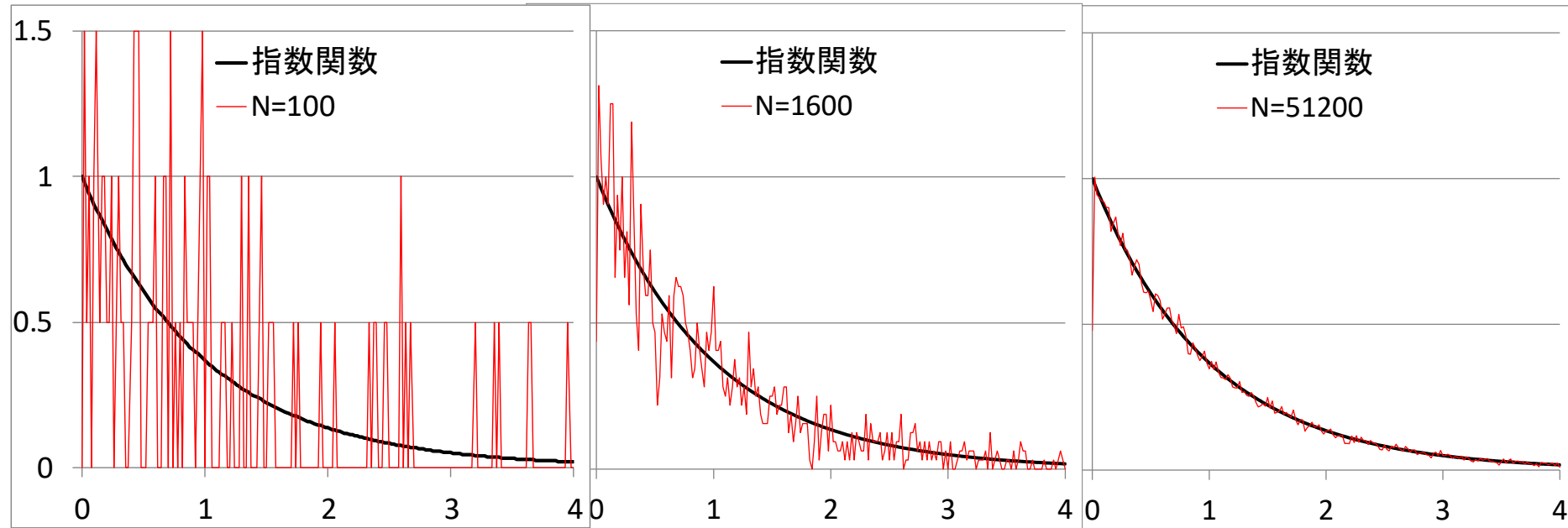
により、 $\lambda = 1$ の指数分布に従う乱数が得られる。

任意の λ に対しては

$$x' = x / \lambda$$

にすればよい

Random numbers that follows exponential distribution



Random numbers that follows normal distribution (Box-Muller method)

http://www.sat.t.u-tokyo.ac.jp/~omi/random_variables_generation.html#Gauss

$$p(x) = \left(\frac{1}{2\pi\sigma^2} \right)^{1/2} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right) \quad (\text{平均 } \mu, \text{ 分散 } \sigma \text{ の正規分布})$$

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、

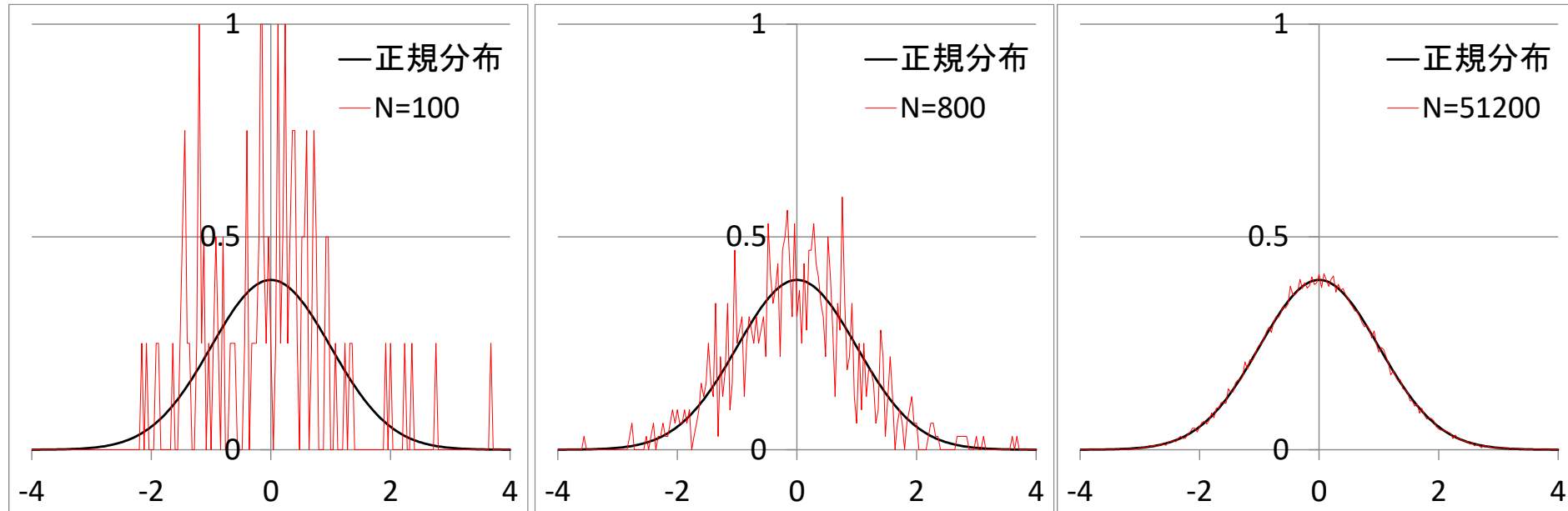
$$z = \left(-2.0 * \log(x) \right)^{1/2} * \sin(2\pi y)$$

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

Random numbers that follows normal distribution



Monte Carlo simulation for statistical physics

How to collect **an ensemble that follows**
canonical statistics $P_i \propto \exp(-E_i/k_B T)$

Metropolis Monte Carlo method

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

という手順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。

PROBLEM, June 28

- Submit electronic file(s) via T2SCHOLAR in 2 days
(If T2SCHOLAR doesn't work, send the files to kamiya.t.aa@m.titech.ac.jp.
In this case, file name must include your STUDENT ID and FULL NAME)

PROBLEM: Answer can be in Japanese or English

- (i) Find (x, y) to minimize $\exp(-x^2) * \sin(x+y)$ by your-chosen algorithms
hint: use SD method with fixed alpha
if df/dx and df/dy are approximated, use central differences
- (i) Optional: Propose if you have any other numerical analysis you want to learn in Computational Materials Science
- (ii) Optional: Propose if you have any python program (should be simple) you want to learn

Non-linear (NL) optimization

非線形最適化

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)

Objective function to minimize: $F(x_j)$

Iteration: $x_j^{(i+1)} = x_j^{(i)} - (\partial^2 F / \partial x_k \partial x_{k'})^{-1} (\partial F / \partial x_k)$

$F''_{kk'} = \partial^2 F / \partial x_k \partial x_{k'}$: 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 algorithm 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)

$$F(x_j^{(k)} + \alpha d) = F(x_j^{(k)}) + \alpha \nabla F(x_j^{(k)})^T d + \frac{1}{2} \alpha^2 d^T B^{(k)} d \sim 0$$

Search direction d is determined from $B^{(k)} d = -\nabla F(x_l^{(k)})$

DFP method: The first formulation of quasi-Newton method

$$s^{(k)} = x^{(k+1)} - x^{(k)}, \quad y^{(k)} = \nabla F(x_l^{(k+1)}) - \nabla F(x_l^{(k)})$$

$$\begin{aligned} B^{(k+1)} &= B^{(k)} + \frac{(y^{(k)} - B^{(k)} s^{(k)}) \cdot y^{(k)T} + y^{(k)} \cdot (y^{(k)} - B^{(k)} s^{(k)})^T}{s^{(k)T} \cdot y^{(k)}} \\ &\quad - \frac{s^{(k)T} \cdot (y^{(k)} - B^{(k)} s^{(k)})}{(s^{(k)T} \cdot y^{(k)})^2} y^{(k)} \cdot y^{(k)T} \\ &= B^{(k)} - \frac{B^{(k)} s^{(k)} \cdot y^{(k)T} + y^{(k)} \cdot (B^{(k)} s^{(k)})^T}{s^{(k)T} \cdot y^{(k)}} + \left(1 + \frac{s^{(k)T} B^{(k)} s^{(k)}}{s^{(k)T} \cdot y^{(k)}} \right) \end{aligned}$$

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

$$s^{(k)} = x^{(k+1)} - x^{(k)}, \quad y^{(k)} = \nabla F(x_j^{(k+1)}) - \nabla F(x_j^{(k)})$$

$$B^{(k+1)} = B^{(k)} - \frac{B^{(k)} s^{(k)} (B^{(k)} s^{(k)})^T}{s^{(k)T} B^{(k)} s^{(k)}} + \frac{y^{(k)} y^{(k)T}}{s^{(k)T} \cdot y^{(k)}}$$

Algorithm:

STEP 0: Provide initial values $x^{(0)}$ and initial matrix $B^{(0)}$ (can be unit matrix)

STEP 1: Search direction $d^{(k)}$ is determined from $B^{(k)} d = -\nabla F(x_j^{(k)})$

STEP 2: Step width $\alpha^{(k)}$ is determined by **direct search algorism**

STEP 3: Calculate $x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$

STEP 4: End if self-consistency is achieve.

If not, go to STEP 5

STEP 5: Calculated $s^{(k)}$ and $y^{(k)}$, and then $B^{(k+1)}$, and go to STEP 1

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)

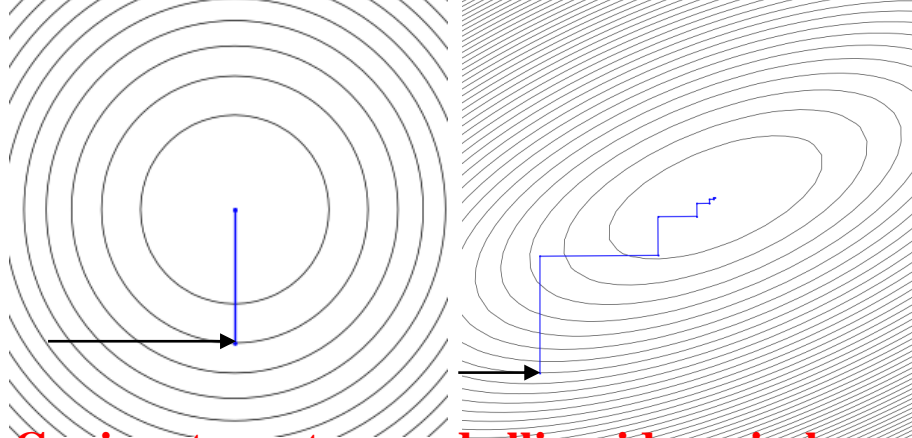
Vectors u and v satisfy $u^T A v = 0$ for a matrix A : u and v are conjugate with each other

- 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 \Rightarrow 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 ellipsoids – circle conversion

$$u^T P^T P v = u^T A v = 0$$

1. Give initial value x_0
2. Initial direction d is determined by SD
$$d = -\nabla f$$

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

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

$$d_{k+1} = -\nabla f(x_{k+1}) + \frac{\nabla f(x_{k+1})^T y_k}{d_k^T y_k} d_k$$

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 d_k at some interval (typically n_{param} , necessary for $n_{\text{param}} = 2$).

PROBLEM, June 25

- 自分で何かしらのデータをフィッティングするプログラムを書く上での注意点とおすすめの方法 (scipy optimize ?)
データ例: XRDピーク、光の干渉によるフリンジ、エネルギーバンド

Non-linear fitting is sensitive to initial parameters.

Fitting program is better to have following functions

1. Simulation

sim()

read_file(): read input file and return xin and yin

cal_ylist(): calculate ysim values for xin

Plot xin vs. yin and ysim

2. Fitting

fit()

addition to sim(), perform scipy.optimize.minimize() for fitting

better to plot and show fitting processes (using callback)

3. Specify input file, optimization algorithm, and initial parameters

See [peakfit.py](#)

PROBLEM, June 25

Optimization algorithms available in `scipy.optimize.minimize()`

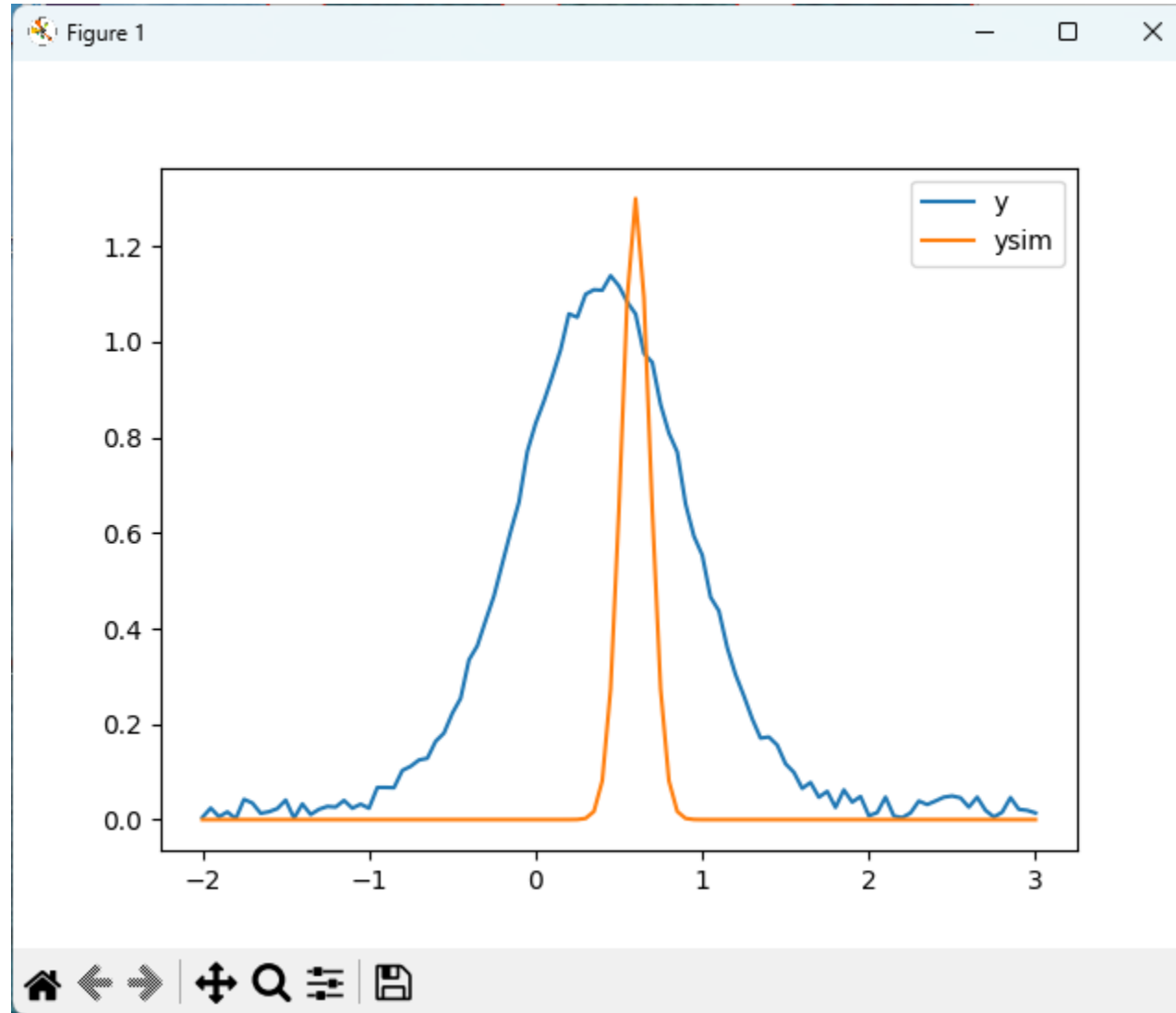
<https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.htm>

#nelder-mead	Downhill simplex <= Simples, Easiest, but Slow
#powell	Modified Powell
#cg	conjugate gradient (Polak-Ribiere method) <= Major gradient method
#bfgs	BFGS法 <= Major gradient method
#newton-cg	Newton-CG
#trust-ncg	信賴領域 Newton-CG 法
#dogleg	信賴領域 dog-leg 法
#L-BFGS-B' (see here)	
#TNC' (see here)	
#COBYLA' (see here)	
#SLSQP' (see here)	
#trust-constr' (see here)	
#dogleg' (see here)	
#trust-exact' (see here)	
#trust-krylov' (see here)	

Ex.: Curve fit

Usage: `python peakfit.py mode input_file method I0 x0 w`

`python peakfit.py sim peak.xlsx` (initial values: $I0 = 1.3$, $x0 = 0.6$, $w = 0.1$)

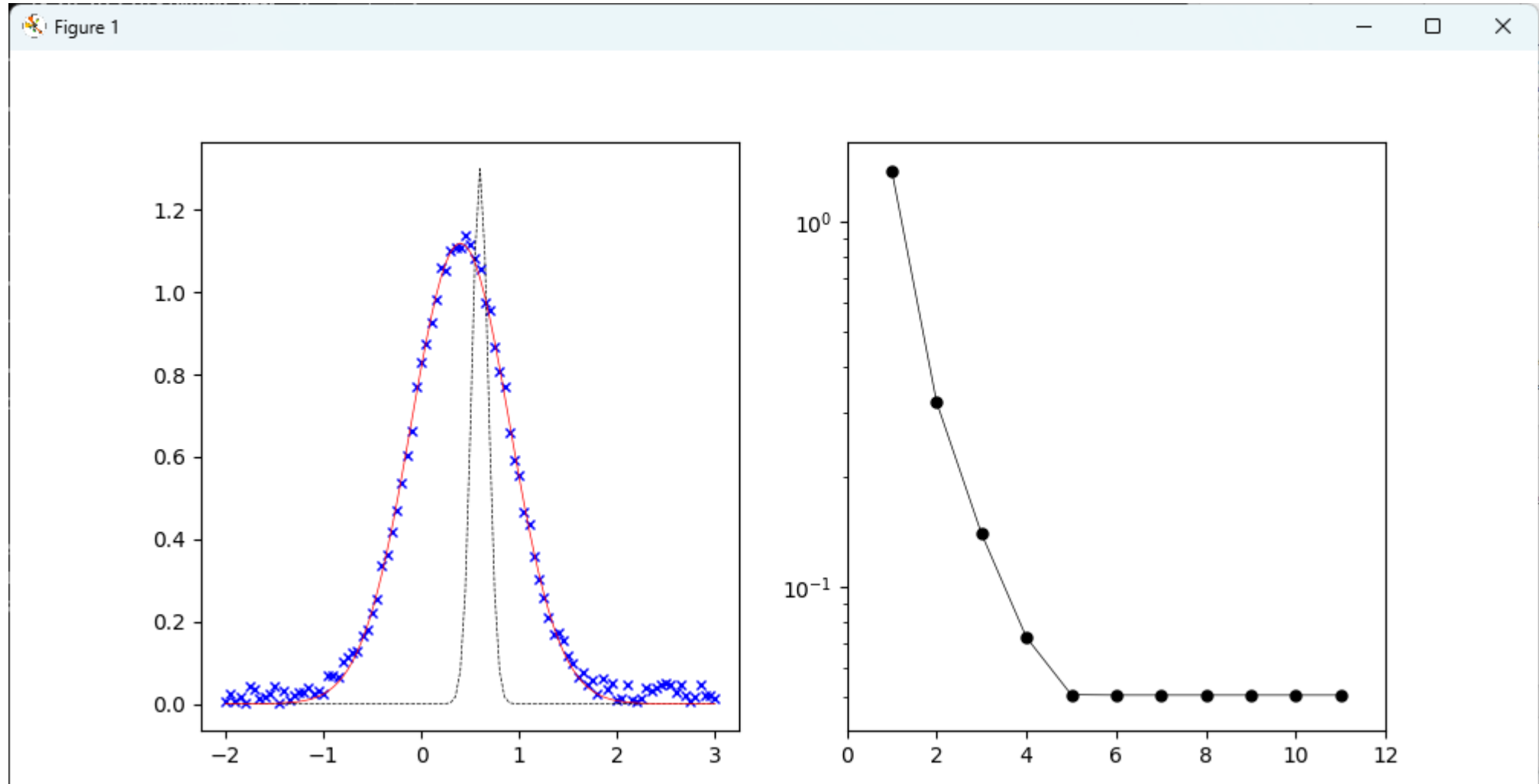


Ex.: Curve fit

python peakfit.py fit peak.xlsx

method: cg

initial values: $I_0 = 1.3$, $x_0 = 0.6$, $w = 0.1$

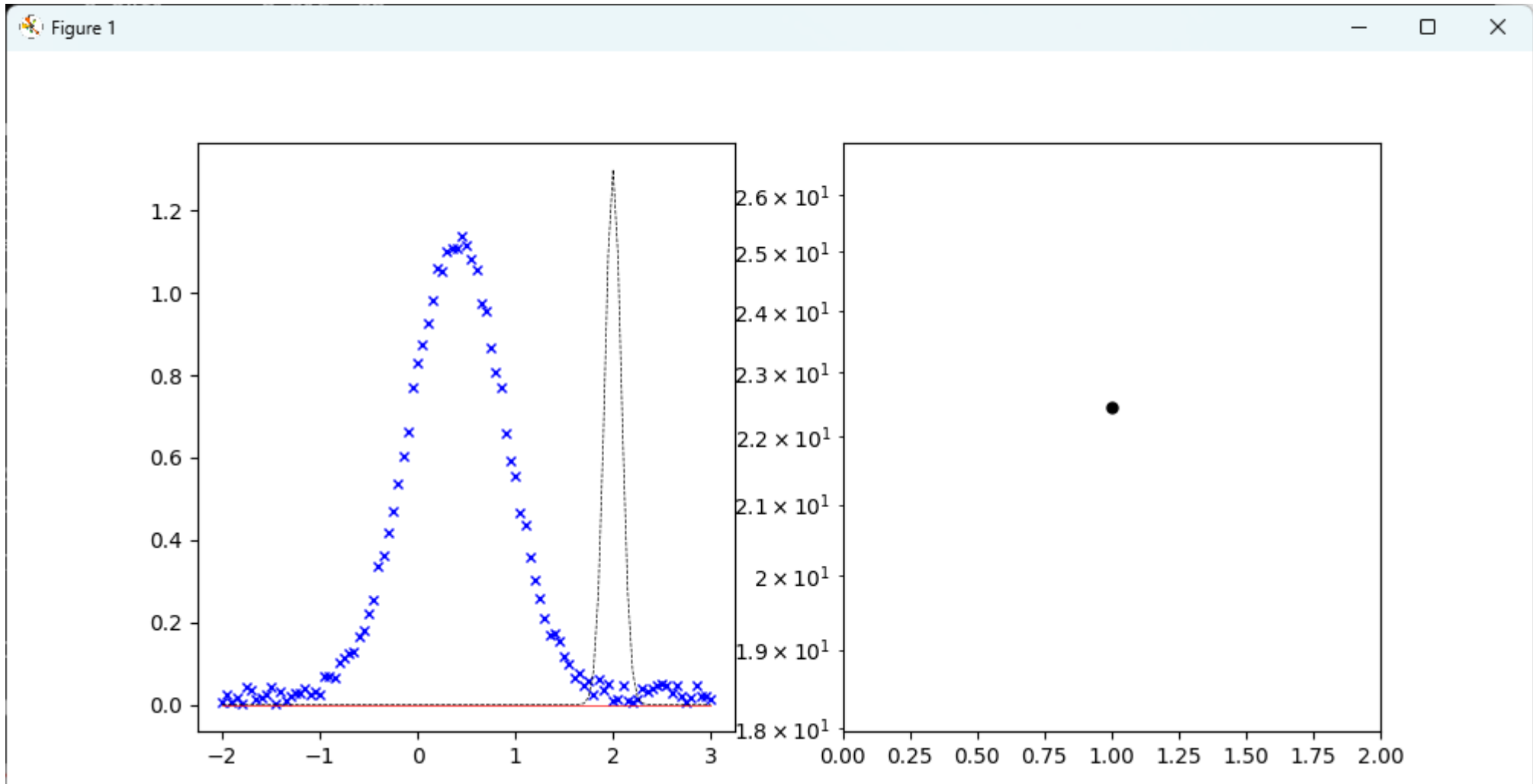


Converging range

python peakfit.py fit peak.xlsx cg 1.3 2.0 0.1

method: cg

initial values: $I_0 = 1.3$, $x_0 = 2.0$, $w = 0.1$

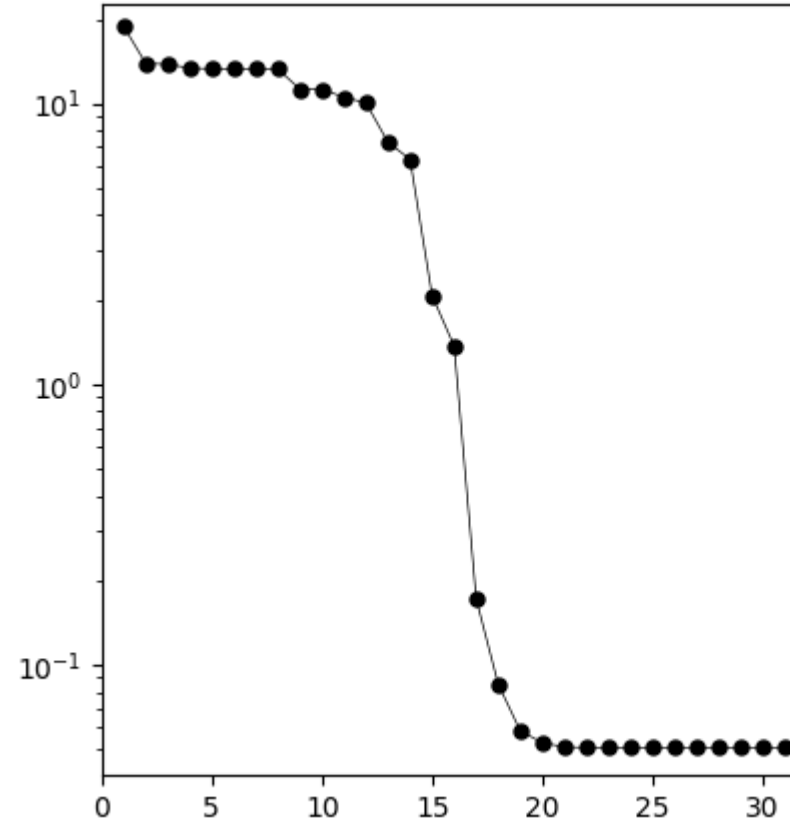
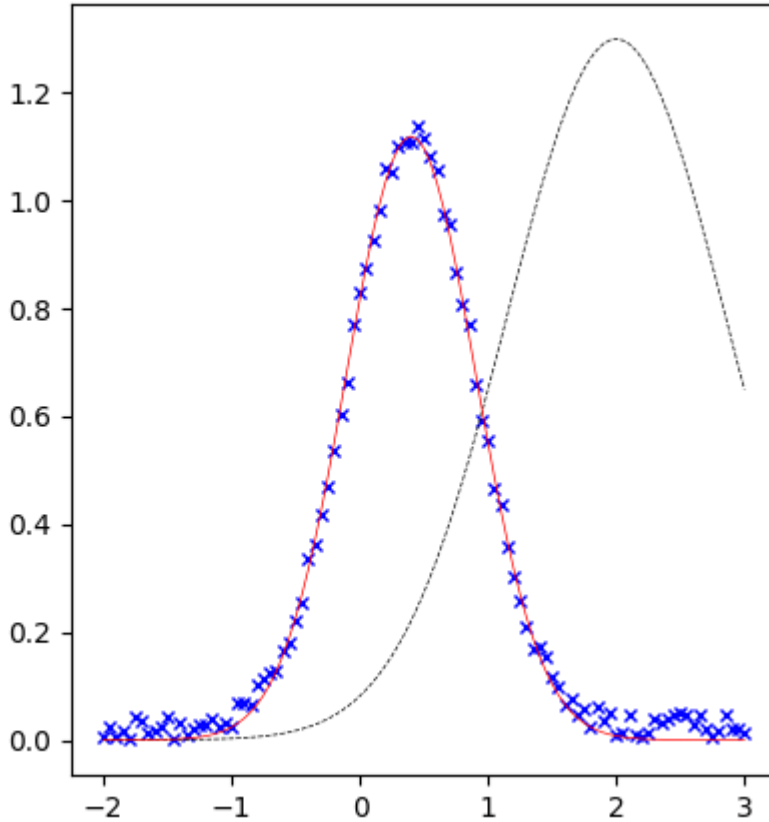


Converged

python peakfit.py fit peak.xlsx cg 1.3 2.0 1.0

method: cg

initial values: $I_0 = 1.3$, $x_0 = 2.0$, $w = 1.0$



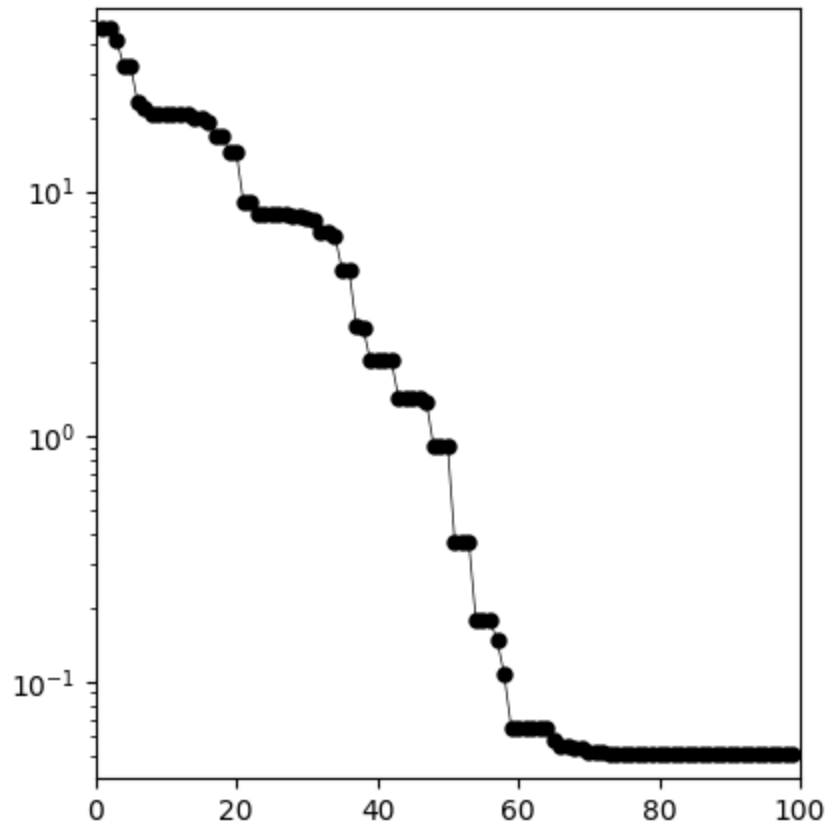
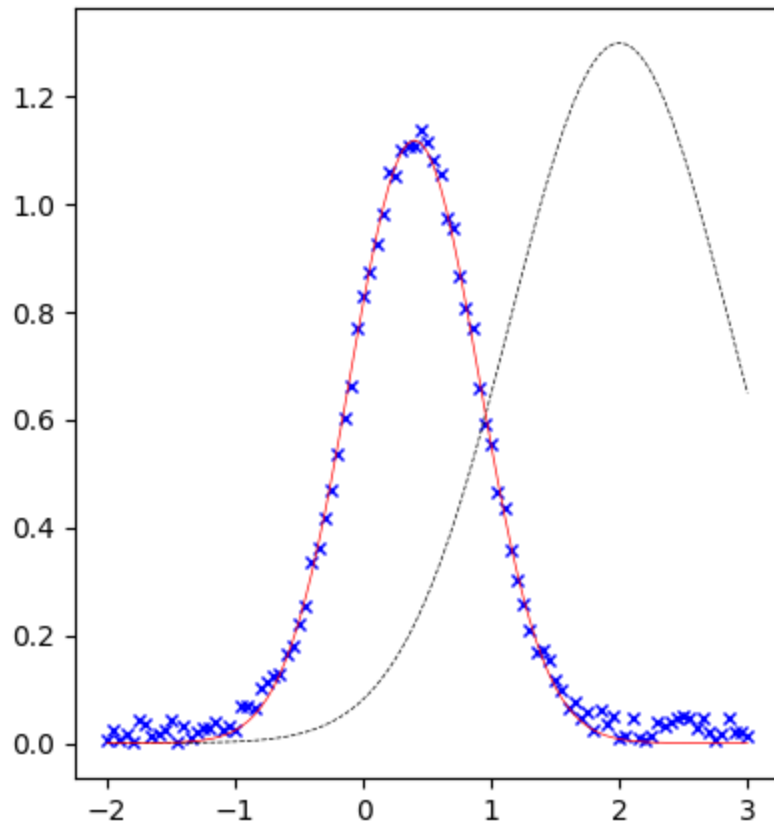
If w is wide to cover the curve region, it can be converged even if the initial peak position is out of the FWHM of the target peak

Converged

python peakfit.py fit peak.xlsx nelder-mead 1.3 2.0 1.0

method: nelder-mead (SIMPLEX)

initial values: $I_0 = 1.3$, $x_0 = 2.0$, $w = 1.0$

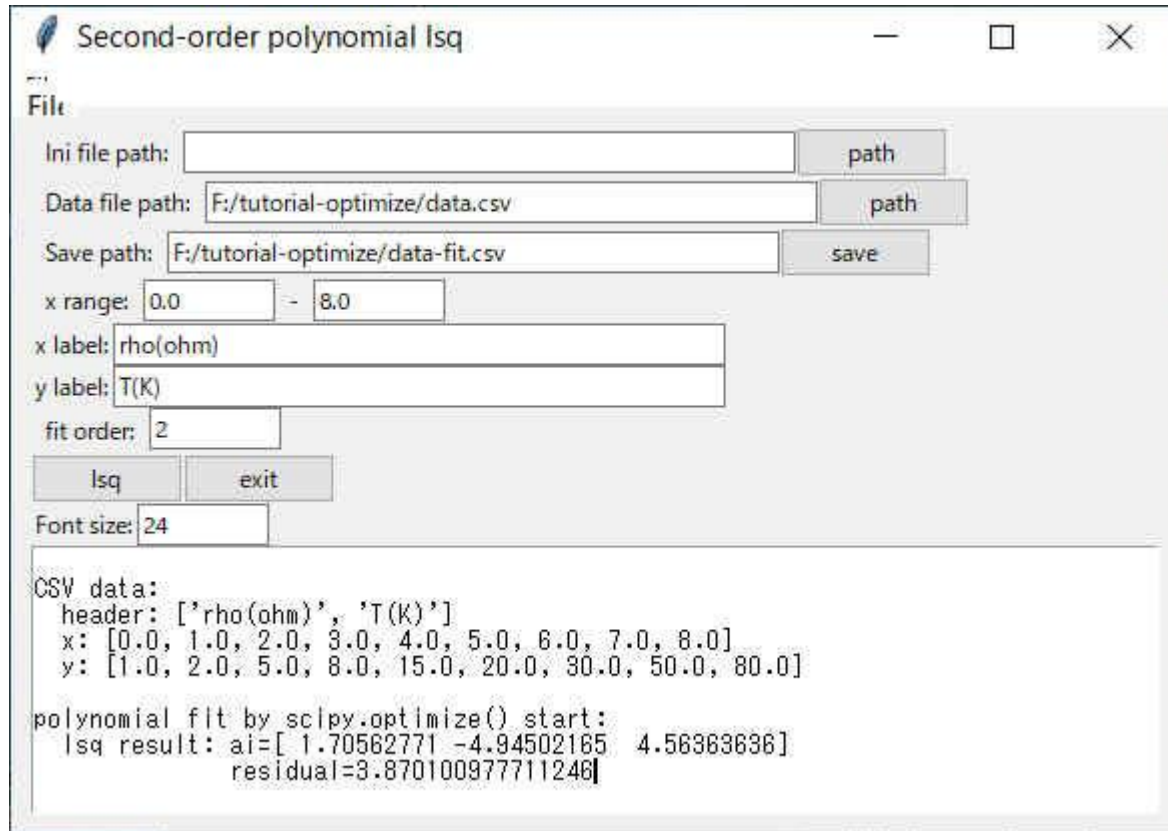


PROBLEM, June 25

- 自分で何かしらのデータをフィッティングするプログラムを書く上での注意点とおすすめの方法 (scipy optimize?)
データ例: XRDピーク、光の干渉によるフリンジ、エネルギーバンド

LSQ+GUI python programming:

<http://conf.msl.titech.ac.jp/Lecture/python/tutorial-optimize/index-python-optimize-gui.html>



pythonによる最小二乗法・最適化問題
GUIプログラミング

関連Web

1. 計算材料科学特論 2022

全ファイルのZIPアーカイブ: [tutorial-optimize.zip](#)

pythonの基本

1. トップページ
2. pythonのインストール方法
[Install python](#)
3. Pythonの基本変数型
4. pythonの起動と対話モード


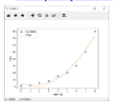
pythonによる最適化GUIプログラミング

a) 基本

1. CSVファイルの読み込みとグラフ
プロット
[01-readcsv.py](#)

[02-plotcsv.py](#)

b) numpy.polyfit()

1. 多項式最小二乗法
[03-polynomial-lsq.py](#)

y
2. グラフにプロット
[04-polynomial-lsq-plot.py](#)


c) scipy.optimize.leastsq()

1. 多項式最小二乗法を行い、グラフ
にプロット
[05-leastsq-plot.py](#)
2. フィッティング範囲を指定

pythonによる最小二乗法・最適化問題 GUIプログラミング

ファイル

- 全ファイルのZIPアーカイブ: [tutorial-optimize.zip](#)

1. 最小二乗法講義スライド: [講義-プログラム-数値計算2019E.pptx](#)

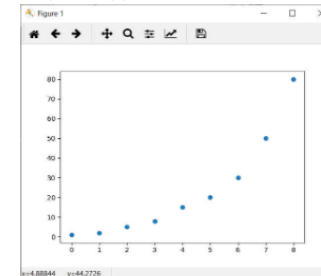
pythonプログラム

a) 基本

1. 入力データ (CSVファイル)
[data.csv](#)
2. CSVの読み込み
[01-readcsv.py](#)



3. CSVから読み込んだデータをグラフ(散布図)にプロット
[02-plotcsv.py](#)



b) numpy.polyfit() を使った多項式最小二乗

1. 多項式最小二乗法を行う
[03-polynomial-lsq.py](#)
2. 多項式最小二乗法を行い、グラフにプロット
[04-polynomial-lsq-plot.py](#)

c) scipy.optimize.leastsq() を使って多項式最小二乗法

1. 多項式最小二乗法を行い、グラフにプロット
[05-leastsq-plot.py](#)

PROBLEM, June 25

Desirable functions for non-linear fitting:

1. **Scaling / standardization** of fitting parameters

e.g., if a parameter x_1 ranges from 10^{-10} to 10^{10} , better to take logarithm
new fitting parameter $x'_1 = \log(x_1)$

2. **Option to choose fixed parameters**

For many parameters fitting, convergence becomes difficult.

Better to fix some parameters, and perform step-by-step optimization
with small number of fitting parameters,
and then finally optimize all parameters.

3. **Constraints**

Some parameters may have limited range.

Constraints can be introduced by the method of Lagrange multiplier)
or by introducing Barrier function / Penalty function.

E.g., additional penalty can be

*if $x_1 < 1.0$: $penalty += k_{penalty} * (x_1 - 1.0)**2$*

*elif $x_1 > 3.0$: $penalty += k_{penalty} * (x_1 - 3.0)**3$*

to limit the X_1 range to $1.0 \leq x_1 \leq 3.0$.

Marquart method (マーカート法)

Minimize a square sum of m functions $f_j(x_i)$ with N parameters

$$F(x_i) = \sum_{j=1}^m f_j(x_i)^2$$

Approximate by

$$f_j(x_i + \delta x_i) \sim f_j(x_i) + \left(\frac{\partial f_j}{\partial x_k} \right) (\delta x_i) = f_j(x_i) + \mathbf{A} \delta x_i \quad A_{jk} = \frac{\partial f_j}{\partial x_k}$$

$$F(x_i + \delta x_i) \sim F(x_i) + 2 \sum_{j,k} f_j A_{jk} \delta x_k + \sum_{j,k,k'} A_{jk} A_{ik'} \delta x_k \delta x_{k'}$$

$$\frac{\partial F(x_i)}{\partial \delta x_k} \sim 2 \sum_j \left(A_{jk} f_j + \sum_k A_{ik} A_{jk} \delta x_j \right) = 0$$

$$\delta x = -(\mathbf{A}^t \mathbf{A})^{-1} \mathbf{A}^t(f_j) \quad \text{Gauss-Newton method}$$

Levenberg-Marquart method

$$\delta x = -(\mathbf{A}^t \mathbf{A} + \lambda I)^{-1} \mathbf{A}^t(f_j) \quad \lambda: \text{dumping factor}$$

$$\delta x = -(\mathbf{A}^t \mathbf{A} + \lambda \text{diag}(\mathbf{A}^t \mathbf{A}))^{-1} \mathbf{A}^t(f_j) \quad \text{e.g. chosen proportional to diagonal sum of } \mathbf{A}^t \mathbf{A}$$

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 $(n+1)$ 個の頂点で作る多面体)

Minimize $F(\mathbf{x}_i)$

1. $(n+1)$ initial values \mathbf{x}_i ($i = 1, 2, \dots, n+1$) \Rightarrow Sort $F(\mathbf{x}_i)$ so that $F(\mathbf{x}_i) > F(\mathbf{x}_{i'})$ ($i < i'$)

$$\mathbf{x}_h = \mathbf{x}_1, \mathbf{x}_1 = \mathbf{x}_{n+1}$$

2. Average except the maximum vertex \mathbf{x}_i $\mathbf{x}_G = \sum_{i=2}^{n+1} \mathbf{x}_i / n$

3. New x will be examined along the line $\mathbf{x}_1 - \mathbf{x}_G$ by the following selections

(i) Reflection (鏡映) : $\mathbf{x}_R = (1 + \alpha)\mathbf{x}_G - \alpha\mathbf{x}_1$ ($\alpha > 0$, ex. 1.0)

(ii) Expansion (拡大) : $\mathbf{x}_E = \gamma\mathbf{x}_R + (1 - \gamma)\mathbf{x}_G$ ($\gamma > 0$, ex. 2.0)

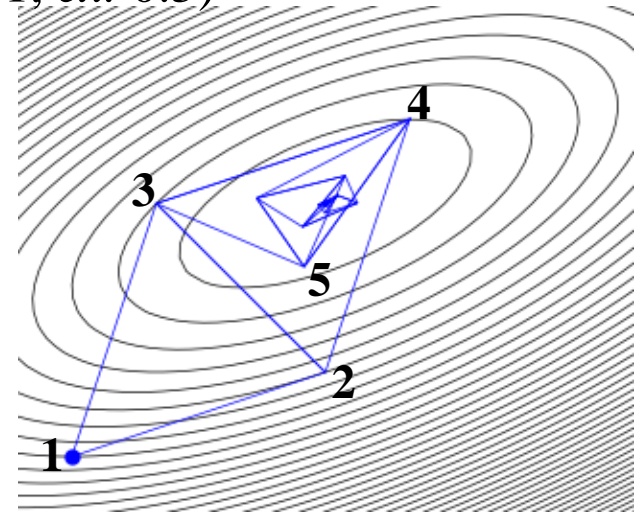
(iii) Contraction (収縮) : $\mathbf{x}_C = \beta\mathbf{x}_1 + (1 - \beta)\mathbf{x}_G$ ($0 < \beta < 1$, ex. 0.5)

(iv) Reduction (縮小) : $\mathbf{x}_{RD} = (\mathbf{x}_1 + \mathbf{x}_i) / 2$

4. Replace \mathbf{x}_1 with the x in (i) – (iv) that firstly satisfies

$$F(\mathbf{x}) < F(\mathbf{x}_1)$$

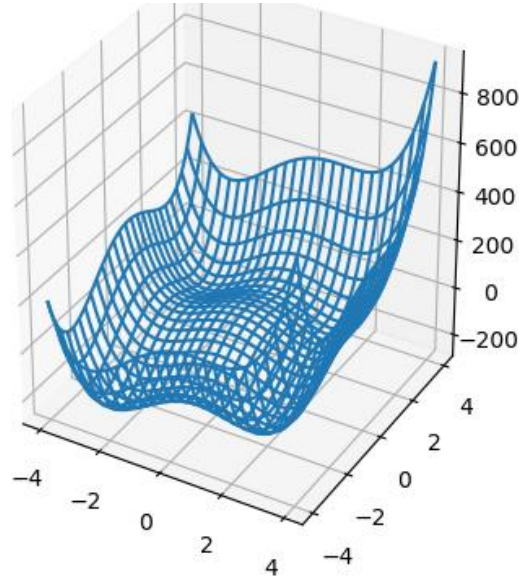
5. Repeat 2 - 4



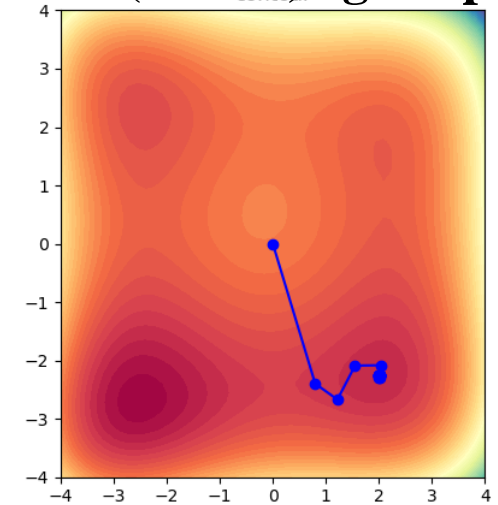
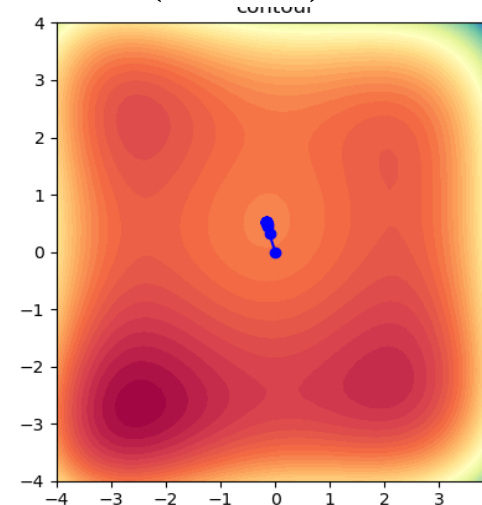
Comparison

$$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30y - 30y^2 + 3y^4 + 3xy^2$$

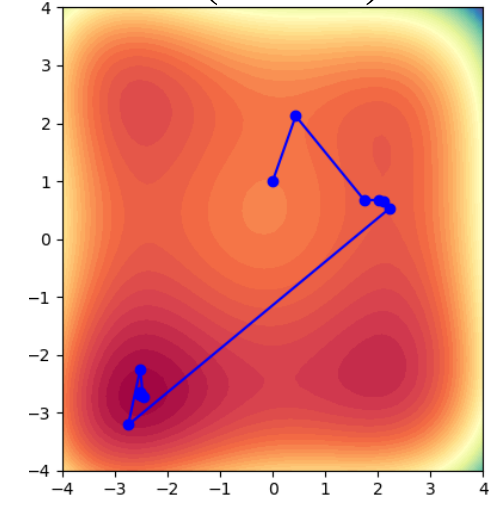
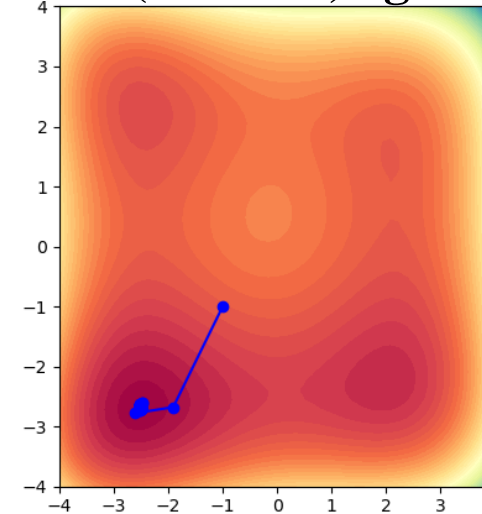
Programs: `optimize-sd-cg2d-linesearch.py`, `optimize-newton-raphson2d.py`



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

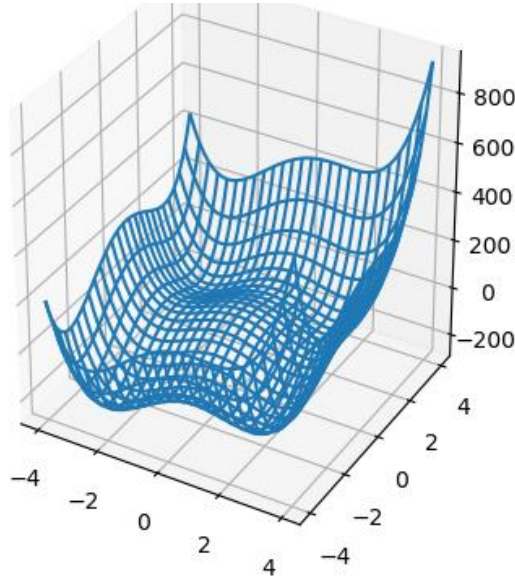


e.g.,
`optimize-sd-cg2d-linesearch.py -1.0 -1.0 sd armijo`

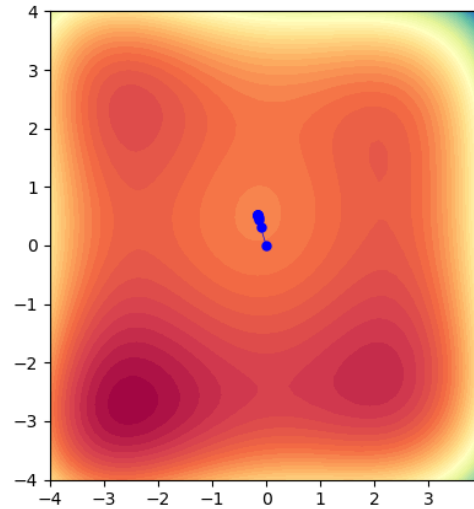
Comparison

$$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30y - 30y^2 + 3y^4 + 3xy^2$$

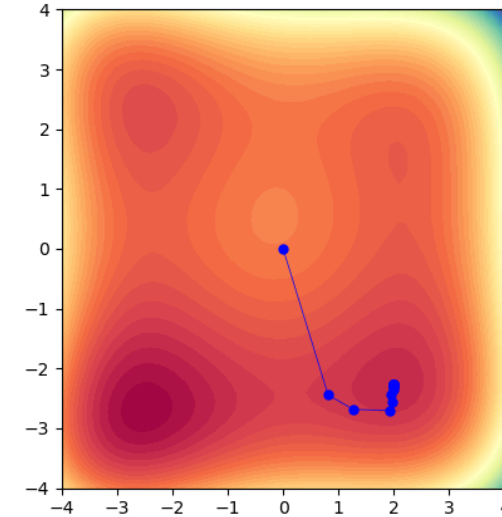
Program not distributed



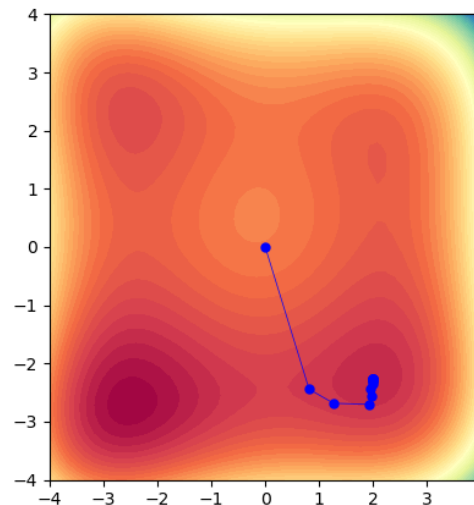
From (0.0 0.0) Newton



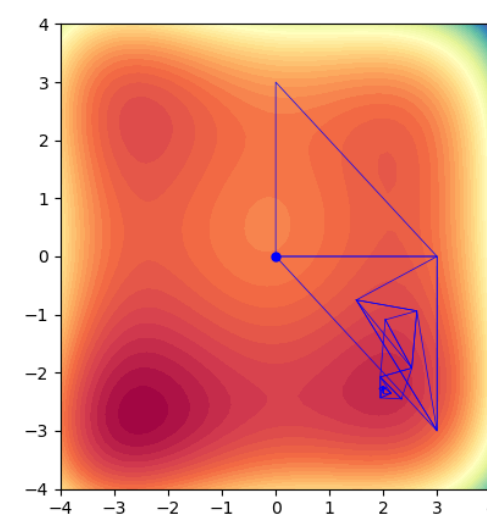
From (-1.0 -1.0) DFP golden



From (0.0 0.0) BFGS golden



From (0.0 1.0) Simplex



Main algorism:

Newton, DFP, BFGS

SD, CG

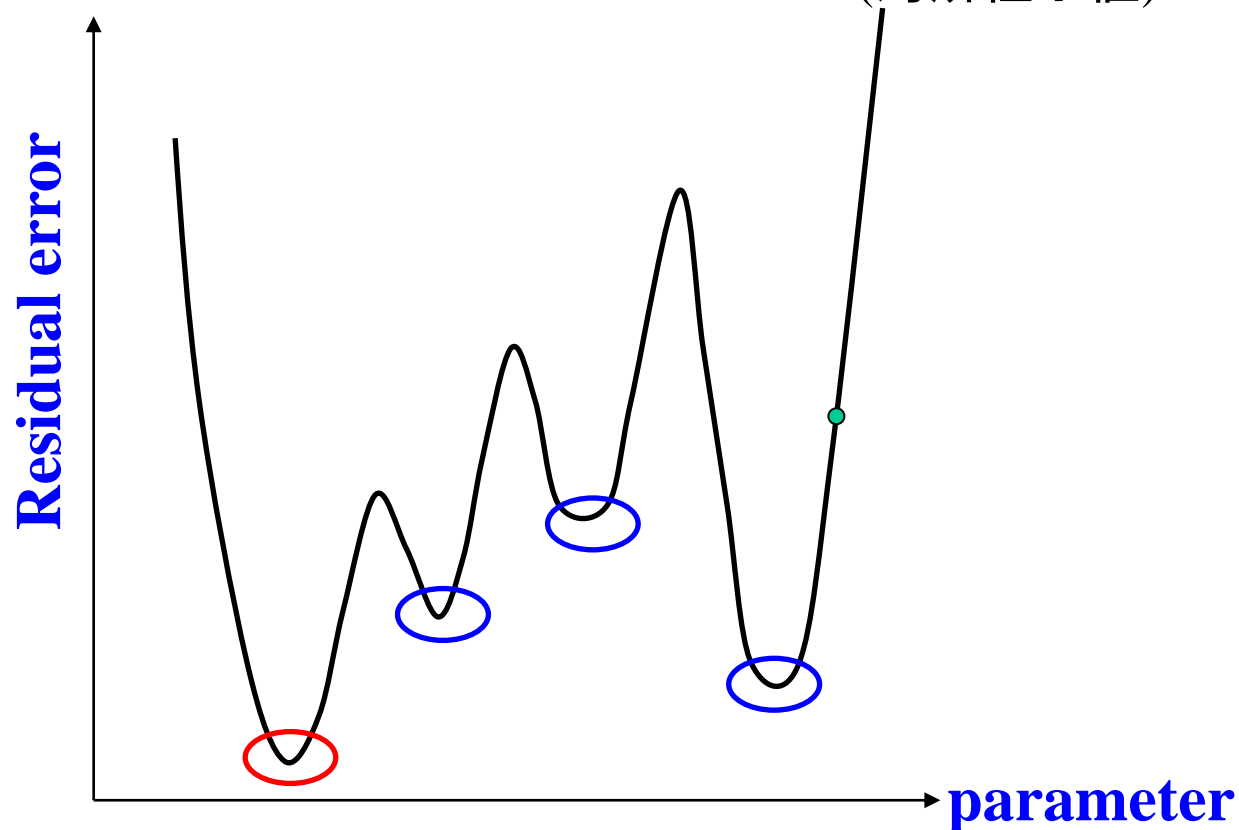
Simplex

Direct search:

Golden, Armijo

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 algorithms

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 algorithm:

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 algorithms.
- **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・ルベグの補題): $\lim_{n \rightarrow \infty} c_n = 0$

Fourier transformation

Take limit to $T \Rightarrow \infty$ for Fourier series expansion

$$\left\{ \begin{array}{l} \text{FT} \quad F(\omega) = \int_{-\infty}^{\infty} f(t) \exp(i\omega t) dt \\ \text{IFT} \quad f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \exp(-i\omega t) d\omega \end{array} \right.$$

$$\left\{ \begin{array}{l} \text{FT} \quad F(\omega) = \int_{-\infty}^{\infty} f(t) \exp(i2\pi ft) dt \\ \text{IFT} \quad f(t) = \int_{-\infty}^{\infty} F(\omega) \exp(-i2\pi ft) d\omega \end{array} \right.$$

Features of Fourier transformation

- Convert time-dependent data to frequency data
- Convert position-dependent data to wavenumber data
- Origin of original data is converted to whole range of FT data
- Whole range of original data is converted to origin of FT data

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

LSQ for general function

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

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

Application to sin / cos expansion

$$f_i(x) = \cos 2\pi f_i x \quad (i = \text{odd numbers (奇数)})$$

$$f_i(x) = \sin 2\pi f_i x \quad (i = \text{even numbers (偶数)})$$

LSQ for Fourier series expansion

$f1, p1, A1 = 1.5, \pi/4.0, 1.0$

$f2, p2, A2 = 3.0, \pi/3.0, 0.3$

$f3, p3, A3 = 10.0, \pi/6.0, 0.5$

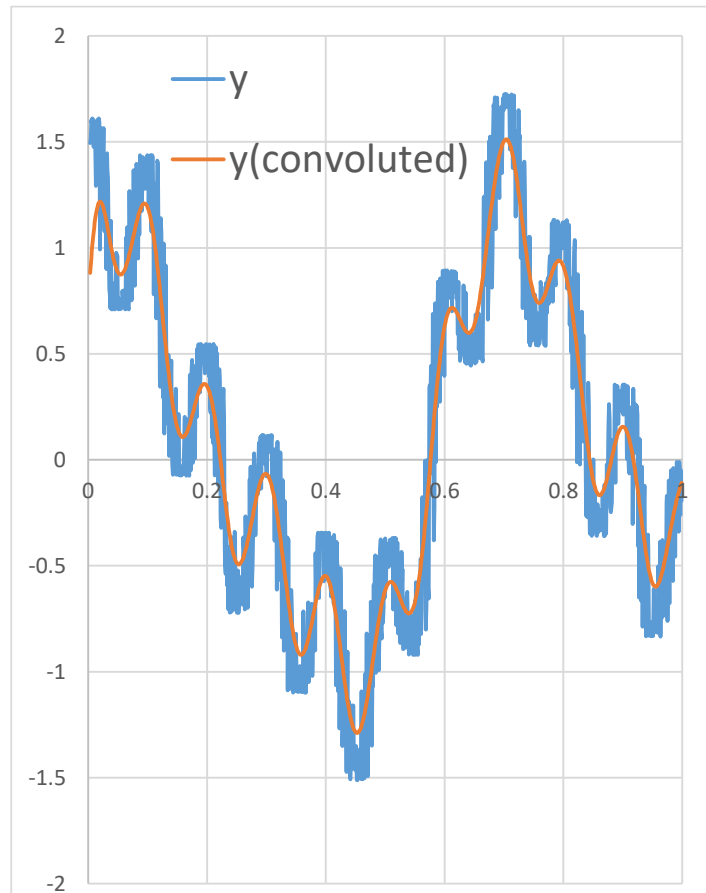
$x += \text{random}(0.03)$ # noise is simulated by random()

$y = A1 * \sin(2.0 * \pi * f1 * x + p1)$

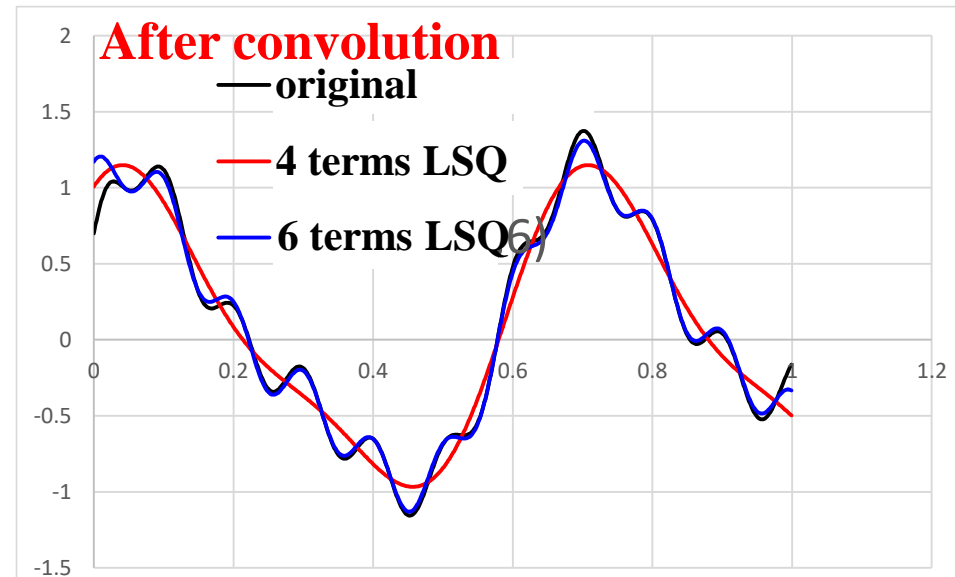
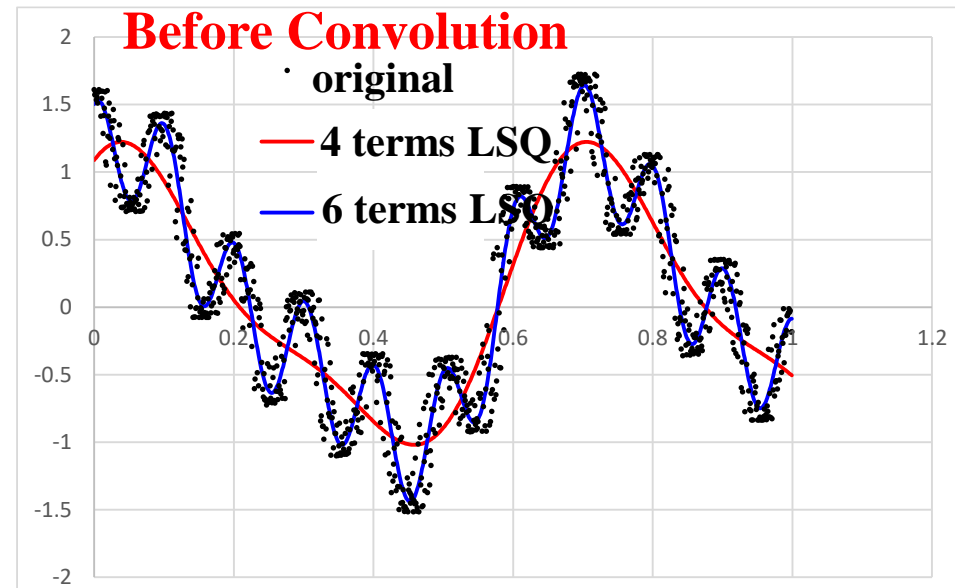
$+ A2 * \sin(2.0 * \pi * f2 * x + p2)$

$+ A3 * \sin(2.0 * \pi * f3 * x + p3)$

Convolution: Gauss function with $w = 0.03$



LSQ results



Discrete FT (DFT, 離散フーリエ変換)

Assume $x(t)$ is periodic in the range $[0, T^w]$ and $x(0) = x(T^w)$

$$X(f_k) = T_s^w \sum_{j=0}^{N-1} x(t_j) \exp(-i2\pi f_k \cdot jT^w/N) \quad T_s^w = T^w/N$$

Usually the coefficient T_s^w is not included for DFT formulations

$$y(f_k) = \sum_{j=0}^{N-1} x(t_j) \exp(-i2\pi kj/N) \quad f_k = k/T^w$$

DFT can be carried out without many trigonometric function (三角関数) calculations

$$y_k = \sum_{j=0}^{N-1} x_j w_N^{kj}$$

$w_N = \exp(-i2\pi/N)$: 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)

$$y(f_k) = \sum_{j=0}^{N-1} x(t_j) \exp(-i2\pi \cdot k \cdot j/N)$$

$$\mathbf{y}_k = \sum_{j=0}^{N-1} \mathbf{x}_j \mathbf{w}_N^{kj}$$

$$\mathbf{w}_N = \exp(-i2\pi/N)$$

DFT

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \mathbf{w}_N^1 & \mathbf{w}_N^2 & \mathbf{w}_N^{N-1} \\ \vdots & \mathbf{w}_N^2 & \ddots & \vdots \\ 1 & \mathbf{w}_N^{N-1} & \cdots & \mathbf{w}_N^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

Inverse DFT

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \mathbf{w}_N^{-1} & \mathbf{w}_N^{-2} & \mathbf{w}_N^{-(N-1)} \\ \vdots & \mathbf{w}_N^{-2} & \ddots & \vdots \\ 1 & \mathbf{w}_N^{-(N-1)} & \cdots & \mathbf{w}_N^{-(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

Using $\mathbf{w}_N^k = \mathbf{w}_N^{k \bmod N}$ and $\mathbf{w}_N^{k+N/2} = -\mathbf{w}_N^k$, 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. FFT is 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 $w_N^k = z$

$$y_k = \sum_{j=0}^{N-1} x_j w_N^{kj} = \sum_{j=0}^{N-1} x_j z^j \quad \mathbf{w_N = \exp(-i2\pi/N): \text{Rotation factor}}$$

$$\begin{aligned} y_k &= x_0 z^0 + x_1 z^1 + x_2 z^2 + \cdots + x_{N-1} z^{N-1} \\ &= x_0 z^0 + x_2 z^2 + \cdots + x_{N-2} z^{N-2} \\ &\quad + z(x_1 z^0 + x_3 z^2 + \cdots + x_{N-1} z^{N-2}) \end{aligned}$$

**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)

$$\begin{aligned}y_{k,N} &= x_0(z^2)^0 + x_2(z^2)^1 + \cdots + x_{N-2}(z^2)^{\frac{N}{2}-1} + z \left(x_1(z^2)^0 + x_3(z^2)^1 + \cdots + x_{N-1}(z^2)^{\frac{N}{2}-1} \right) \\&= y_{k,N/2,1} + z y_{k,N/2,2} \\y_{k,N/2,1} &= x_0(z^4)^0 + x_4(z^4)^1 + \cdots + x_{N-2}(z^4)^{\frac{N}{4}-1} + (z^2) \left(x_2(z^4)^0 + x_6(z^4)^1 + \cdots + x_{N-3}(z^4)^{\frac{N}{4}-1} \right) \\&= y_{k,N/4,1} + (z^2) y_{k,N/4,3} \\y_{k,N/2,2} &= x_1(z^4)^0 + x_5(z^4)^1 + \cdots + x_{N-1}(z^4)^{\frac{N}{4}-1} + (z^2) \left(x_3(z^4)^0 + x_7(z^4)^1 + \cdots + x_{N-2}(z^4)^{\frac{N}{4}-1} \right) \\&= y_{k,N/4,2} + (z^2) y_{k,N/4,4}\end{aligned}$$

$$y_{k,N} = y_{k,N/2,1} + z y_{k,N/2,2}$$

$$y_{k,N/2,1} = y_{k,N/4,1} + z^2 y_{k,N/4,3}$$

$$y_{k,N/2,2} = y_{k,N/4,2} + z^2 y_{k,N/4,4}$$

$$y_{k,N/4,1} = y_{k,N/8,1} + z^4 y_{k,N/8,5}$$

$$y_{k,N/4,2} = y_{k,N/8,2} + z^4 y_{k,N/8,6}$$

$$y_{k,N/4,3} = y_{k,N/8,5} + z^4 y_{k,N/8,7}$$

$$y_{k,N/4,4} = y_{k,N/8,6} + z^4 y_{k,N/8,8}$$

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 $2^2, 2^3, \dots, 2^N$

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 $2^2, 2^3, \dots, 2^N$ のFTの計算をすることでFT計算ができる

Data swap in the FFT procedure

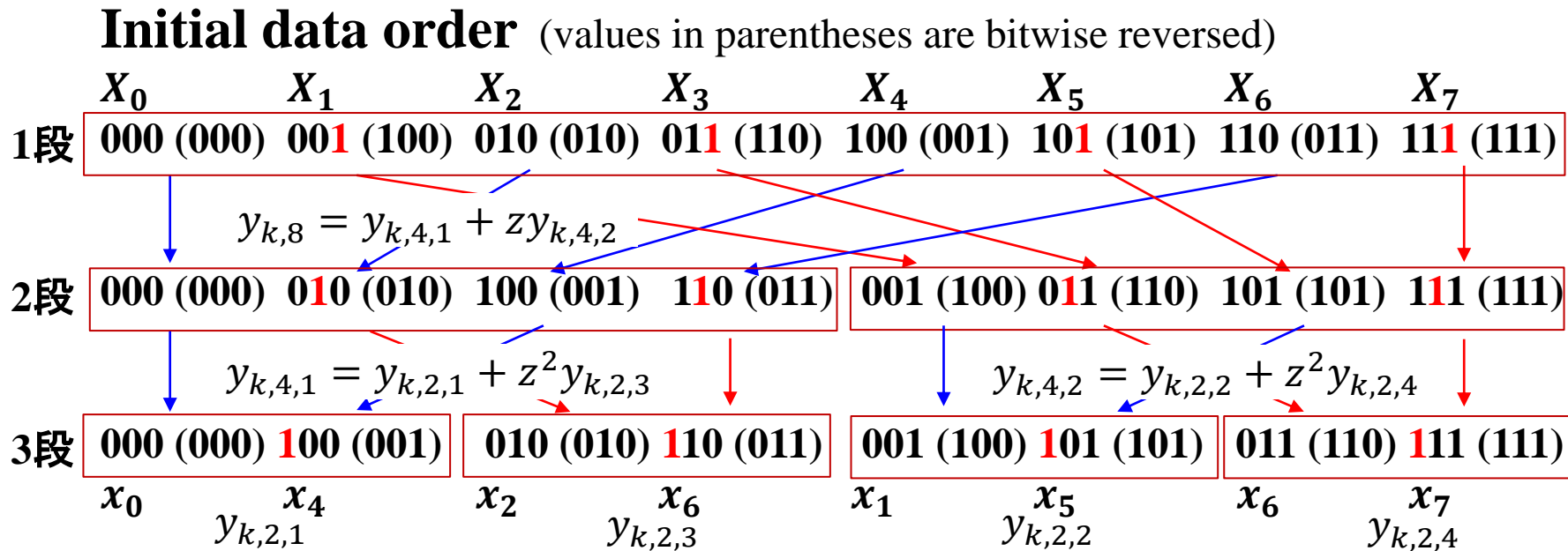
N data series $x_0 x_1 x_2 \cdots x_{N-1} \Rightarrow \text{FT: } X_0 X_1 X_2 \cdots X_{N-1}$

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)

\Rightarrow **Data whose k -th bit is 1 are move to the second half**

\Rightarrow The change of the order numbers corresponds to bit reversal



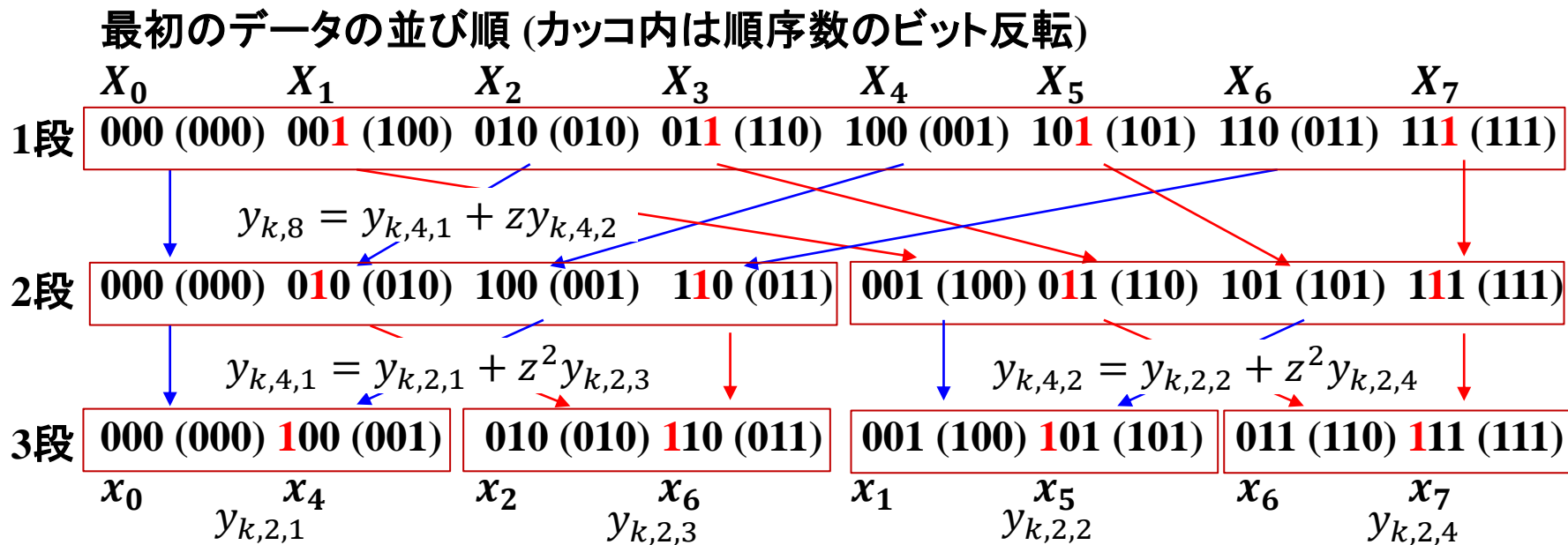
The order to sum up for FFT is different from the order of x_i .

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 $x_0 x_1 x_2 \cdots x_{N-1} \Rightarrow \text{FT: } X_0 X_1 X_2 \cdots X_{N-1}$
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
 \Rightarrow 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」
 \Rightarrow 順序数の変換がビット反転に対応する



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) ($\sim x$, 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 = 11001₂ を例に

ret = 0

ビット反転値を0で初期化

1. $v0 = val \& 1_2 \Rightarrow 11001_2 \& 001_2 = 1$

第1桁のビット値を v0 に保存

2. $ret = ret | v0 \Rightarrow 0 | 1 = 1_2$

retの第1桁に v0 を設定

3. $val = val >> 1 \Rightarrow 11001_2 >> 1 = 1100_2$

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。

$ret = ret << 1 \Rightarrow 1_2 << 1 = 10_2$

1.に戻って繰り返し

6. $v0 = val \& 1_2 \Rightarrow 1100_2 \& 001_2 = 0$

第1桁のビット値を v0 に保存

7. $ret = ret | v0 \Rightarrow 10_2 | 0 = 10_2$

retの第1桁に v0 を設定

8. $val = val >> 1 \Rightarrow 1100_2 >> 1 = 110_2$

9. $ret = ret << 1 \Rightarrow 10_2 << 1 = 100_2$

1.に戻って繰り返し

10. $v0 = val \& 1_2 \Rightarrow 110_2 \& 001_2 = 0$

11. $ret = ret | v0 \Rightarrow 100_2 | 0 = 100_2$

12. $val = val >> 1 \Rightarrow 110_2 >> 1 = 11_2$

13. $ret = ret << 1 \Rightarrow 100_2 << 1 = 1000_2$

1.に戻って繰り返し

14. $v0 = val \& 1_2 \Rightarrow 11_2 \& 1_2 = 1$

15. $ret = ret | v0 \Rightarrow 1000_2 | 1 = 1001_2$

16. $val = val >> 1 \Rightarrow 11_2 >> 1 = 1_2$

17. $ret = ret << 1 \Rightarrow 1001_2 << 1 = 10010_2$

1.に戻って繰り返し

18. $v0 = val \& 1_2 \Rightarrow 1_2 \& 1_2 = 1$

19. $ret = ret | v0 \Rightarrow 10010_2 | 1 = 10011_2$

20. $val = val >> 1 \Rightarrow 1_2 >> 1 = 0_2 \Rightarrow$ 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 1001100011110101101111011₂ 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:
```

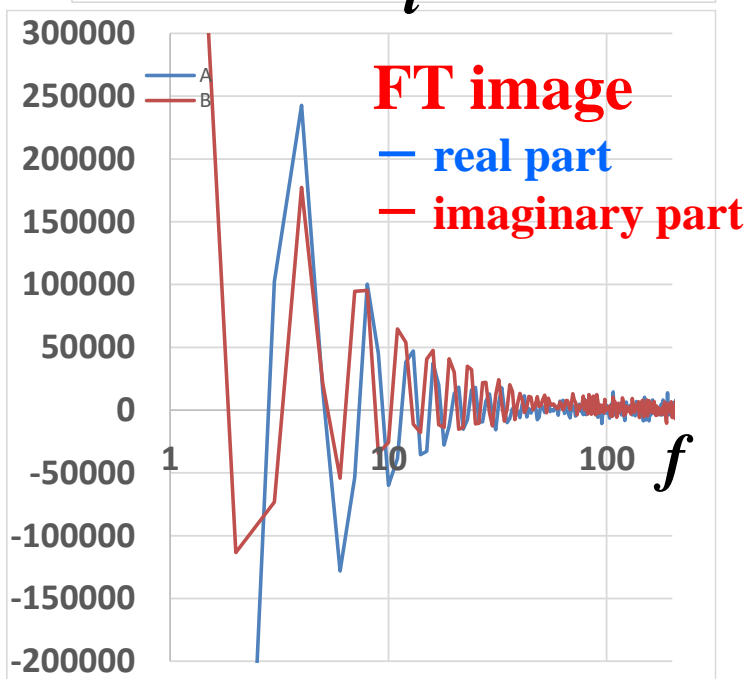
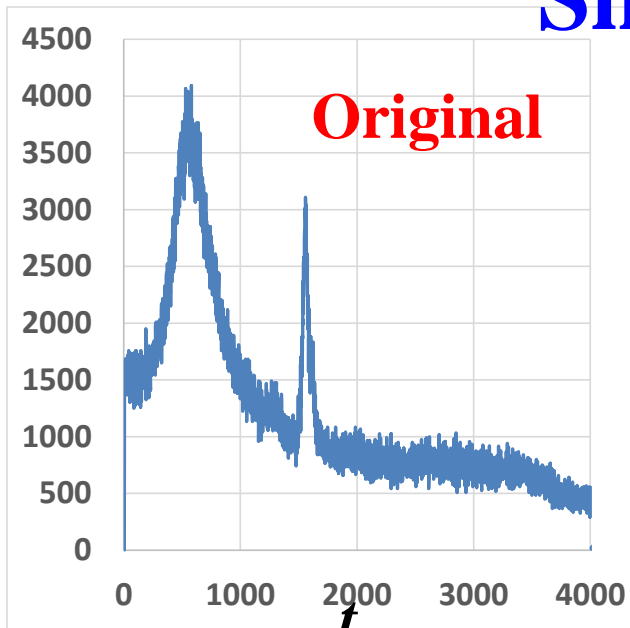
```
        break
```

```
    else:
```

```
        ret = ret * 2          # bit shift ret to left
```

```
    return ret
```

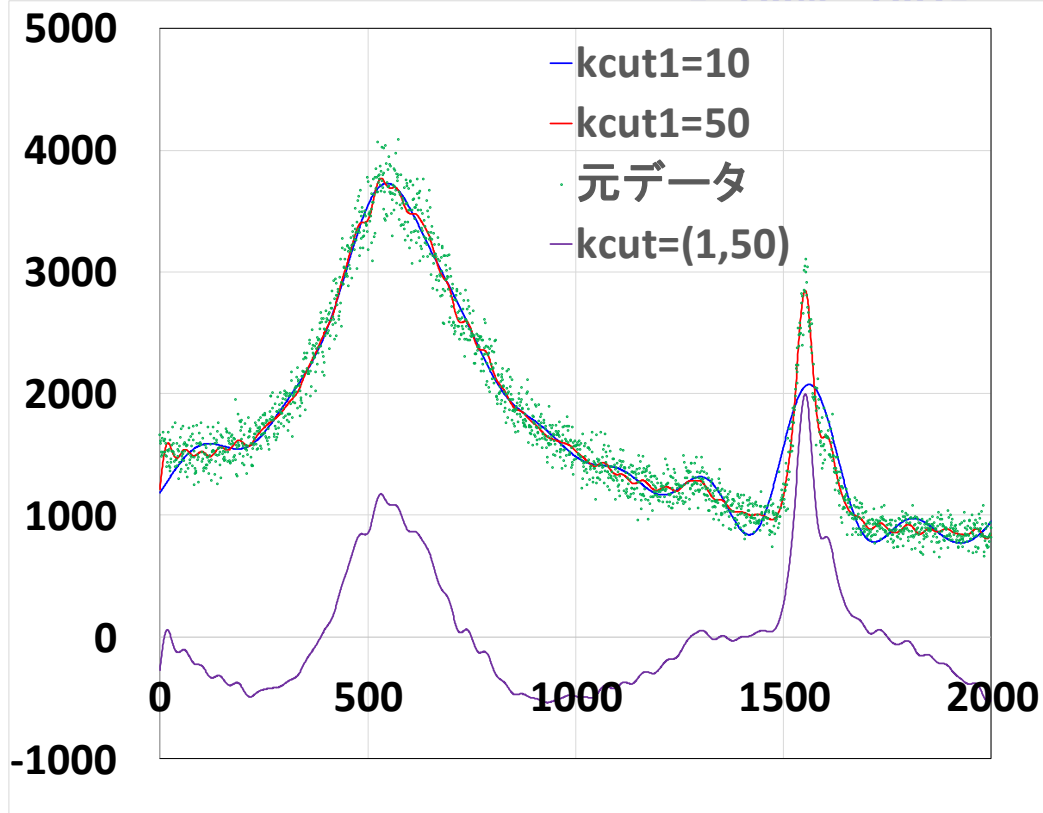
Smoothing: FT



Remove high-frequency FT data: Smoothing
Low-pass filter

Remove low-frequency FT data: Cut drift
High-pass filter

Ex. Cut FT data outside $[k_{\text{cut}0}, k_{\text{cut}1}]$

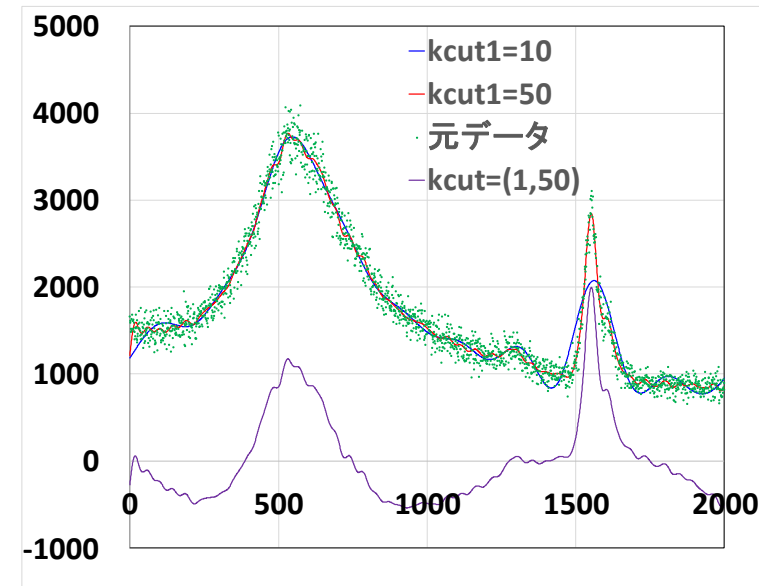


Note:

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.

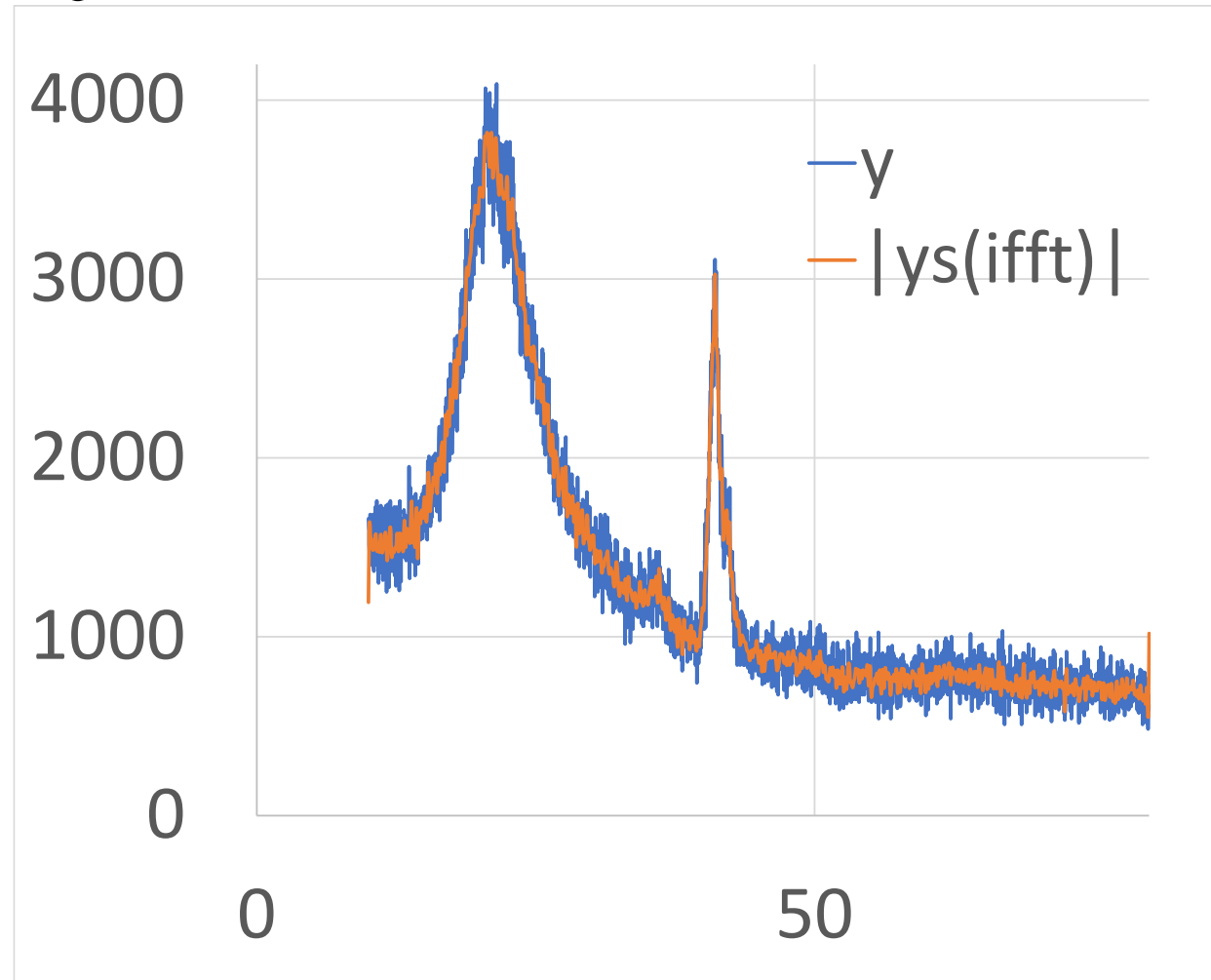


Program: smoothing-fft.py

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

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

=> `plot smoothing-fft.csv`



Interpolation by FT

Periodic function $f(t)$: Period of T

N t points are given in $T = N\Delta t$ at uniform step Δt : $t_j = j\Delta t$ ($j = 0, \dots, N - 1$)

can be expanded by $\exp(i2\pi f_n t_j) = \exp\left(i2\pi \frac{n}{T} t_j\right) = \exp\left(i2\pi \frac{n}{N\Delta t} j\Delta t\right) = \mathbf{\exp\left(i2\pi \frac{nj}{N}\right)}$, $f_n = n/T$

$$\times f(t_j) = \sum_{n=0}^N a_n \exp\left(i2\pi \frac{n}{N} j\right)$$

For a case for $N = 4$: a_j are determined so as to reproduce $f(t_j)$ for integer j

$$f(t_j) = a_0 + a_{\pi/4} \mathbf{\exp\left(i \frac{\pi}{4} j\right)} + a_{2\pi/4} \mathbf{\exp\left(i \frac{2\pi}{4} j\right)} + a_{3\pi/4} \mathbf{\exp\left(i \frac{3\pi}{4} j\right)}$$

a_j are obtained by Fourier transformation

Interpolate: for arbitrary floating-point number $t_r = x\Delta t$:

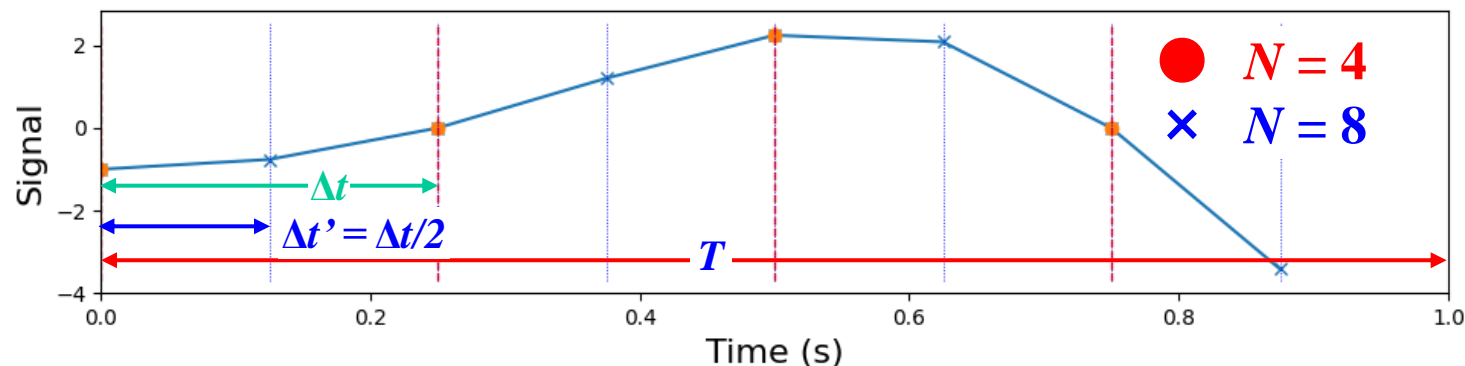
$$f(t_r = x\Delta t) = a_0 + a_{\pi/4} \mathbf{\exp\left(i \frac{\pi}{4} x\right)} + a_{2\pi/4} \mathbf{\exp\left(i \frac{2\pi}{4} x\right)} + a_{3\pi/4} \mathbf{\exp\left(i \frac{3\pi}{4} x\right)}$$

fft.py

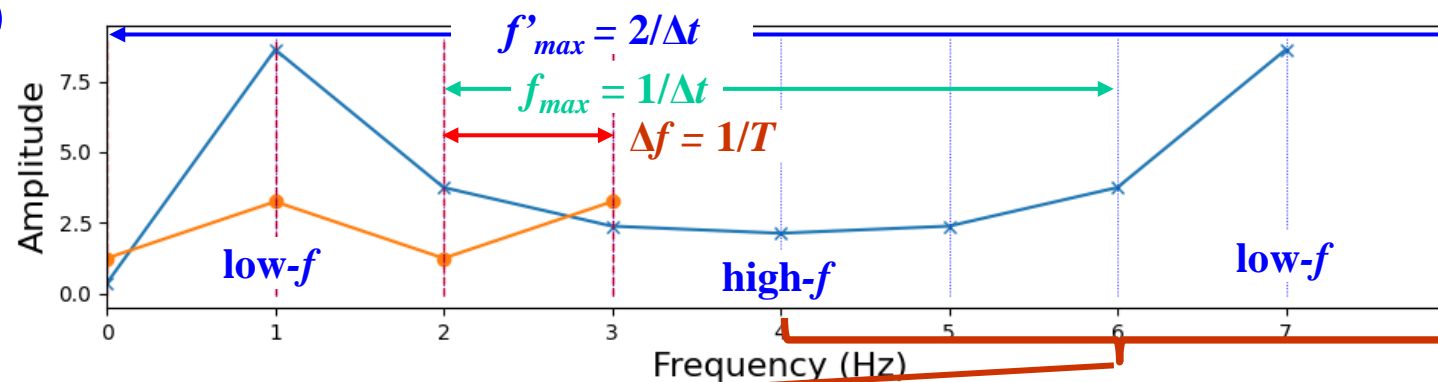
FFT: python numpy.fft.fft()

$f(t) = -\cos(2\pi t)(1+5t^2)$, periodic in $t = [0, 1)$

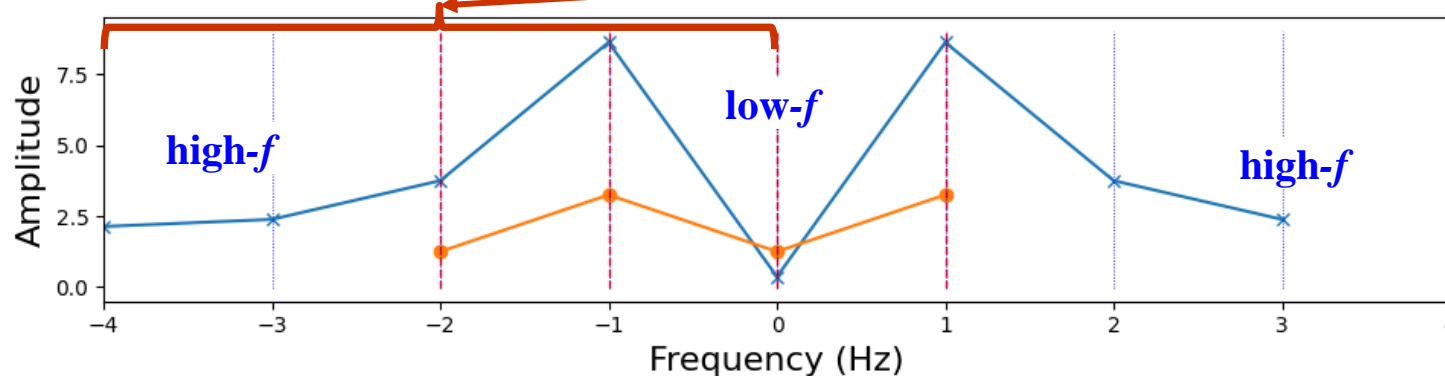
$f(t)$



$F(f) = \text{np.fft.fft}(f)$



$F(f) = \text{np.fft.fft}(f)$



NOTE:

Periodicity of Fourier func:

$$F(f + f_{\max}) = F(f)$$

python list: $a[-n] = a[N - n]$

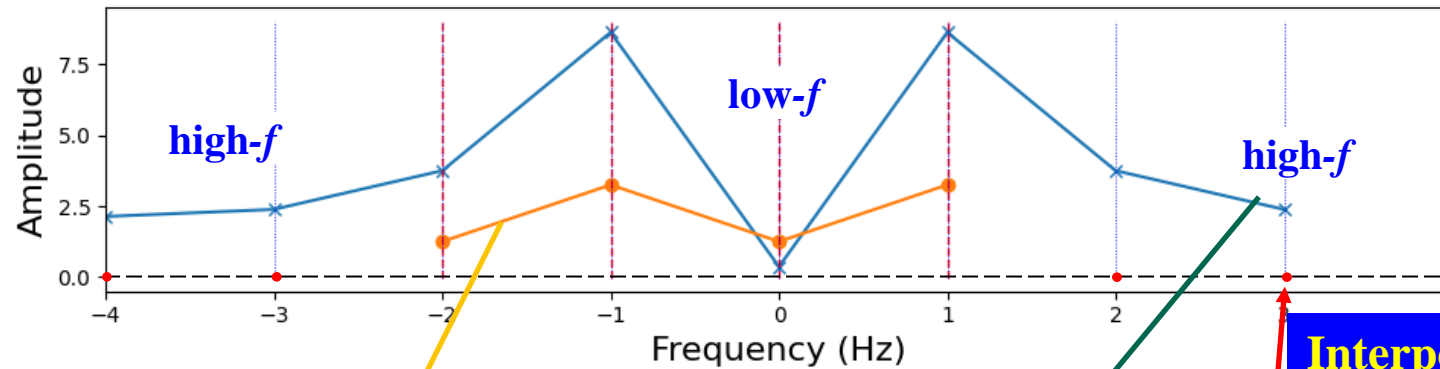
($N = \text{len}(a)$)

FFT: python numpy.fft.fft()

fft.py

$f(t) = -\cos(2\pi t)(1+5t^2)$, periodic in $t = [0, 1)$

$F(f) = \text{np.fft.fft}(f)$



For a case for $N = 4$:

$$f(t_j) = +a_{-2} \exp(-i\pi j) + a_{-1} \exp\left(i\frac{\pi}{2}j\right) + a_0 + a_1 \exp\left(i\frac{\pi}{2}j\right)$$

For a case for $N = 8$:

$$f(t'_j) = a'_{-4} \exp(i2\pi j) + a'_{-3} \exp\left(i\frac{3}{2}\pi j\right) + a'_{-2} \exp(i\pi j) + a'_{-1} \exp\left(i\frac{\pi}{2}j\right) \\ + a'_0 + a'_1 \exp\left(i\frac{1}{2}\pi j\right) + a'_2 \exp(i\pi j) + a'_3 \exp\left(i\frac{3}{2}\pi j\right)$$

Interpolation by FFT:

1. Increase number of FTed data
2. Add zeros to additional high-f data
3. Inverse FFT to get interpolated values
4. Correct scale

Interpolate by the FFT result for $N = 4$: e.g., to get $f(t_{1/2} = \Delta t/2)$:

$$f(t_{1/2}) = a_0 + a_{\pi/4} \exp\left(i\frac{\pi}{8}\right) + a_{2\pi/4} \exp\left(i\frac{2\pi}{8}\right) + a_{3\pi/4} \exp\left(i\frac{6\pi}{8}\right)$$

take $a'_j = a_j$ ($j = -2, -1, 0, 1$), $a'_j = 0$ (else)

Interpolation by FFT: Example for 1D band structure

python interpolate_fft.py interpolate_fft_test.xlsx 1

Step 2: Compute the FFT

```
y_fft = np.fft.fft(y)
```

Step 3: Pad zeros to FTed data ($n_{\text{interp}} = n_{\text{samples}} * \text{interp_factor}$)

```
y_fft_padded = np.zeros(n_interp, dtype = complex)
```

```
y_fft_padded[:n_samples//2] = y_fft[:n_samples//2]
```

```
y_fft_padded[-n_samples//2:] = y_fft[-n_samples//2:]
```

```
y_interp = np.fft.ifft(y_fft_padded) * interp_factor
```

make an increased array for interpolated FTed data

copy the first half of the original FTed data y_fft

copy the last half of the original FTed data y_fft

correct scale

