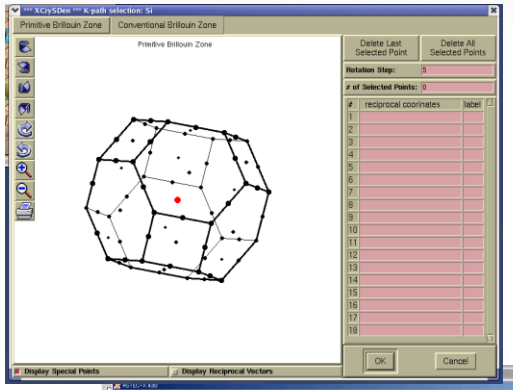
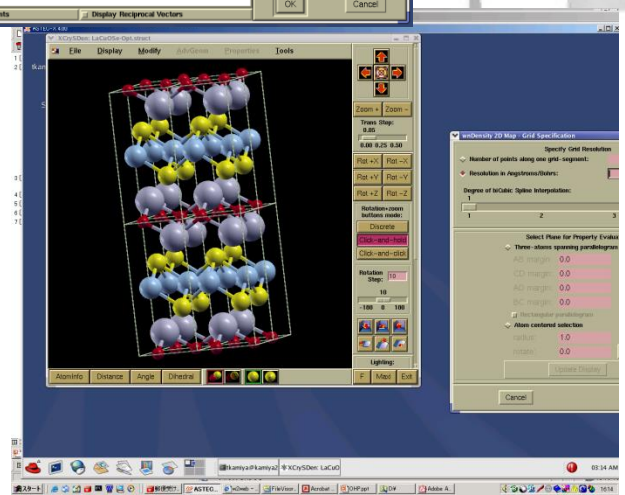
講義web: http://d2mate.mdxes.iir.isct.ac.jp/Lecture/FundamentalsCMS/

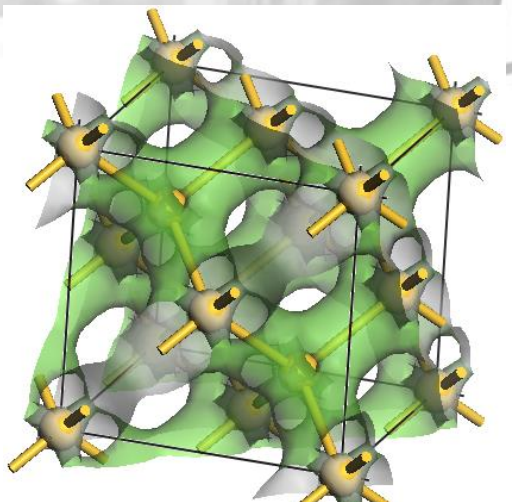
材料計算科学基礎

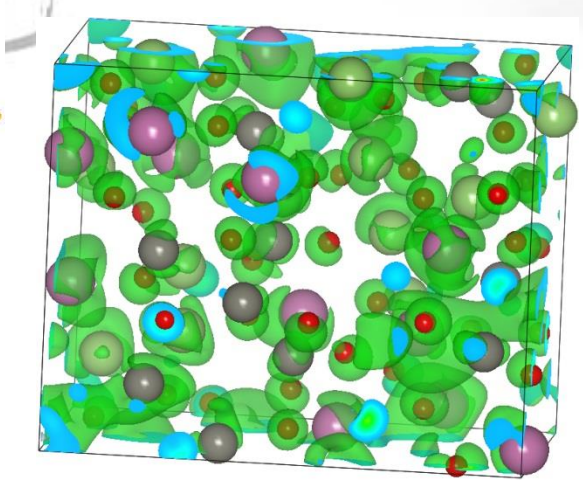
物質理工学院材料系
総合研究院 元素戦略MDX研究センター
神谷利夫

TA: 樋口龍生
南島悠人









授業日程

#09　2025/1/10 (金)　神谷担当

- プログラミングの基礎 (python)
- 例題: 回帰
  - 線形最小二乗法 (多項式)
  - 生成AI (ChatGPT、Microsoft 365 copilot, github copilot) を使ったデータ作成、プログラミング

#10　2025/1/14(火)

- 生成AIの使い方 復習
- python文法
- 例題: 回帰
  - 線形最小二乗法の理論
  - 線形最小二乗法 (一般関数)

#11　2025/1/21(火)

- pythonらしいプログラム (lsq_general_copilot.py)
- 行列計算
- 非線形最小二乗法 (numpy.curve_fit())
- 機械学習回帰 (Ridge回帰、過学習の確認)

#12　1/24〜#14　1/31　講義

評価

講義毎の課題で評価

注: 原則として、提出物の内容による評価は行わないが、

- 講義で配布された以外の ＋α があれば若干加点
- 一言感想だけのように「さすがにこれはないだろう」くらいに脱力感があったら若干減点

提出方法: T2SCHOLARで提出

提出期限: 1/23(木) 23:59

課題: 次のいずれかを選択して、成果物等を提出

- 成果物の例:
  - 授業で作成したプログラム (.py、.ipynb)
  - 授業に関連して作成したプログラム
  - 授業に関連して生成AIに質問し、その結果理解したこと
    質問のプロンプトと回答、回答に関しての自分の感想・理解など
- 講義に関する質問、要望

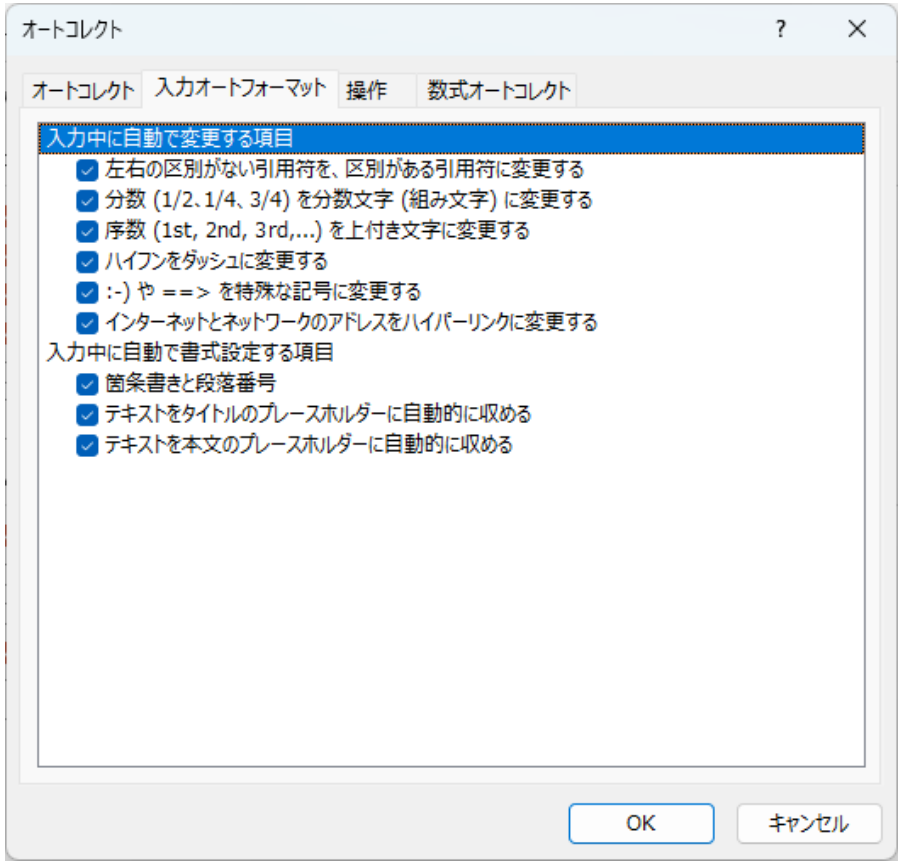
質問

問題点: パワーポイントの文字をターミナルに張り付けてもグラフが出てこなくて困った

原因: パワーポイントが以下の変換を勝手にしてしまうため、対応しきれていない

- -- => –
- " => ‘’
- "" => “”

“ファイル” => “オプション” => “文章校正”
=> “オートコレクトのオプション”
で無効化できるが・・・



質問: pythonモジュールのインストールエラー

質問: ローカル環境でpythonを使うとき、numpyがインストールされているのにターミナル上でnumpyがインストールされていないと表示されるときはどのように対応すればいいか
回答: .ipynbの実行アイコン、.pyの実行アイコン、ターミナルでの実行では、
VSCodeが使うpythonが違うので注意
.ipynbが使うpython kernel: .ipynb編集領域のツールバー右端で選択
.pyが使うpython: ステータスバーの “python”の右にあるバージョンで選択
あるいは、**Ctrl+Shift+Pでコマンドパレットを開き、python:select interpreter**で設定
ターミナルが使うpython: .pyのpythonと同じはずだが、問題が起こったら確認すること

確認方法:
> python --version
> pip --version
で、pipの対応pythonバージョンが同じかどうかを確認
> where python
> where pip
で、pythonとpipの実行ファイルのディレクトリが同じかどうかを確認

質問: pythonモジュールのインストールエラー

確認方法:

> python --version

> pip --version

で、pipの対応pythonバージョンが同じかどうかを確認

> where python

> where pip

で、pythonとpipの実行ファイルのディレクトリが同じかどうかを確認

バージョンの問題はなさそうで、かつ、モジュールのインストールに成功している場合は、-m オプションでpipを実行してみる

> **python –m pip install numpy**　(pipはpython -m pip のwrapperプログラム)

VSCode環境確認支援プログラム

- **which.py: python which.py python, python which.py pip など。**
  python, pip 実行ファイルについて、実行優先順位順に表示
- **check_vsc.ipynb, check_vsc.py: which.pyを含む、実行環境確認プログラム**

質問: if __name__ == “__main__” トリック

Q: main()関数のところの理解が追いつかなかった。また、初めて見る表記に戸惑った。
　lsq_poly1.pyの__name__変数の部分の書き方がよくわからなかった。

プロンプト: pythonにおいて次のようなmain()関数を作成した際，main()関数をライブラリとして使う時の例をつくり、わかりやすく説明してください

```
def main():
    --cut–
if __name__ == "__main__":
    main()
```

回答: 次ページ

質問: if __name__ == “__main__” トリック

回答: main() 関数を持つスクリプトをライブラリとして使用する際の例を紹介します。例えば、greeting.py というファイルに次のコードがあるとします

 # greeting.py

```
def main():
    name = input("名前を入力してください: ")
    print(f"こんにちは、{name}さん！")

if __name__ == "__main__":
    main()
```

python greeting.pyでの実行結果:
__name__には “__main__”が入っているので、main()関数が実行されます

ライブラリとしての利用方法

この greeting.py を他のスクリプトからライブラリとして使用することができます。
以下に import_greeting.py という別のスクリプトを作成してみましょう:

 # import_greeting.py

```
import greeting

def another_function():
    print("これは別の関数です。")

another_function()
```

python import_greeting.pyでの実行結果:
importされた場合は、greeting.py内での__name__にはモジュール名（ファイル名）“greeting”が入っているため、greeting.pyのif文はFalseになり、main()は実行されません

if __name__ == “__main__” トリックの例

#which.py

```
def search_path(command):
    --cut--

def main():
    print()
    print("実行可能ファイルを検索")
 --cut--
    command = sys.argv[1]
    search_path(command)

if __name__ == "__main__":
    main()
```

python which.py pythonでは、main()を通してsearch_path()が実行される

#check_vsc.ipynb, check_vsc.py

```
import which

which.search_path('python')
which.search_path('pip‘)
```

で search_path()を呼び出しているが、
which.main()は呼び出されないので、
print(“実行可能ファイルを検索”) は実行されない

わざわざmain()を定義して呼び出す理由

- pythonの標準ではない
  プログラム例では、if __name__ ブロック内にmain()の内容を書くことも多い
- **C言語では、スタートアップ関数がmain()に固定されている**
- main()があったら、ライブラリとしては使われない関数であると判断できる

Pythonらしく書き直す

Program: lsq_general.py

```
# 項毎に関数形が違うので、項が区別できる文字列リスト変数 flabels を作っておくとわかりやすい
flabels = ['1', 'cos(2x)', 'sin(2x)', 'cos(x)', 'sin(x)', 'cos(3x)', 'sin(3x)', 'x', 'x^2', 'exp(x)']

# i 番目の関数の値を返す関数。if文で条件分岐する
def lsqfunc(i, x):
    if i == 0:
        return 1.0
    elif i == 1:
        return cos(2.0*x)
    elif i == 2:
        return sin(2.0*x)
    elif i == 3:
        return cos(x)
    elif i == 4:
        return sin(x)
    elif i == 5:
        return cos(3.0*x)
    elif i == 6:
        return sin(3.0*x)
    elif i == 7:
        return x
    elif i == 8:
        return x**2
    elif i == 9:
        return exp(x)
```

lsq_general.py を pythonらしく書き直してもらう

プロンプト:次のプログラムを、リスト内包表記、sum,map関数などを使って、pythonらしいプログラムに書き直してください

lsq_general.py:

```
def lsqfunc(i, x):
    if i == 0:
        return 1.0
    elif i == 1:
        return cos(2.0*x)
    elif i == 2:
        return sin(2.0*x)
    elif i == 3:
        return cos(x)
    elif i == 4:
        return sin(x)
    elif i == 5:
        return cos(3.0*x)
    elif i == 6:
        return sin(3.0*x)
    elif i == 7:
        return x
    elif i == 8:
        return x**2
    elif i == 9:
        return exp(x)
```

lsq_general_copilot.py:

```
def lsqfunc(i, x):
    functions = [lambda x: 1.0,
                 lambda x: cos(2.0 * x),
                 lambda x: sin(2.0 * x),
                 lambda x: cos(x),
                 lambda x: sin(x),
                 lambda x: cos(3.0 * x),
                 lambda x: sin(3.0 * x),
                 lambda x: x,
                 lambda x: x ** 2,
                 lambda x: exp(x)
                ]
    return functions[i](x)
```

lambdaは無名関数。
関数名が無いので、
　　変数に代入したり、
　　関数の引数で渡したりして使う

ポイント:
- **pythonでは、関数も他の変数と同じように扱える**
　　(変数、関数などはすべてオブジェクト)
- 関数の変数から関数を呼び出すには、**()演算子**を使う: **functions[i](x)**

生成AI: pythonらしく書き直してもらう

プロンプト:次のプログラムを、リスト内包表記、sum,map関数などを使って、pythonらしいプログラムに書き直してください

lsq_general.py:

```
def mlsq(xs, ys, m):
    n = len(xs)
    Si  = np.empty([m, 1])
    Sij = np.empty([m, m])
    for l in range(0, m):
        v = 0.0
        for i in range(0, n):
            v += ys[i] * lsqfunc(l, xs[i])
        Si[l, 0] = v

    for j in range(0, m):
        for l in range(j, m):
            v = 0.0
            for i in range(0, n):
                v += lsqfunc(j, xs[i]) * lsqfunc(l, xs[i])
            Sij[j, l] = Sij[l, j] = v

    print("Vector and Matrix:")
    print("Si=")
    print(Si)
    print("Sij=")
    print(Sij)
    print("")

    ci = np.linalg.inv(Sij) @ Si
    ci = ci.transpose().tolist()
    return ci[0]
```

lsq_general_copilot.py:

```
def mlsq(xs, ys, m):
    Si = np.array([[sum(ys[i] * lsqfunc(l, xs[i]) for i in range(len(xs)))] for l in range(m)])
    Sij = np.array([[sum(lsqfunc(j, xs[i]) * lsqfunc(l, xs[i]) for i in range(len(xs))) for l in range(m)] for j in range(m)])

    print("Vector and Matrix:")
    print("Si=")
    print(Si)
    print("Sij=")
    print(Sij)
    print("")

    ci = np.linalg.inv(Sij) @ Si
    return ci.flatten().tolist()
```

ポイント:
・リスト内包表記を使うと、
forループブロックを１行で書ける

生成AI: pythonらしく書き直してもらう

プロンプト:次のプログラムを、リスト内包表記、sum,map関数などを使って、pythonらしいプログラムに書き直してください

lsq_general.py:

```
def main():
--cut—
df = pd.read_excel(infile, engine = 'openpyxl')
    labels = df.columns.to_list()
    x = df[labels[0]]
    y = df[labels[1]]
    ndata = len(x)
    minx = min(x)
    maxx = max(x)
    if minx < 0.0:
        xcal0 = minx * 1.5
    else:
        xcal0 = minx * 0.5
    if maxx < 0.0:
        xcal1 = maxx * 0.5
    else:
        xcal1 = maxx * 1.5
--cut--

    xcal = [xcal0 + i * xcalstep for i in range(ncal)]
    ycal = []
    for i in range(ncal):
        _x = xcal[i]
        yl = 0.0
        for k in range(nfunc):
            yl += ci[k] * lsqfunc(k, _x)

        ycal.append(yl)
```

lsq_general_copilot.py:

```
def main():
--cut--
    df = pd.read_excel(infile, engine='openpyxl')
    labels = df.columns.to_list()
    x, y = df[labels[0]], df[labels[1]]
    ndata = len(x)
    minx, maxx = min(x), max(x)
    xcal0 = minx * 1.5 if minx < 0.0 else minx * 0.5
    xcal1 = maxx * 0.5 if maxx < 0.0 else maxx * 1.5
```

ポイント: if~else文を代入分に使える

```
--cut--

    xcal = [xcal0 + i * xcalstep for i in range(ncal)]
    ycal = [sum(ci[k] * lsqfunc(k, xcal[i]) for k in range(nfunc)) for i in range(ncal)]
```

生成AI: pythonらしく書き直してもらう

プロンプト:次のプログラムを、リスト内包表記、sum,map関数などを使って、pythonらしいプログラムに書き直してください。また、書き直した部分にはコメントを入れてください。

lsq_general.py:

```
def main():
--cut--
    print("LSQ function")
    print("f(x) = {:6.3g}".format(ci[0]), end = '')
    for i in range(1, nfunc):
        print(" + {:6.3g} * {}".format(ci[i], flabel[i]), end = '')
    print("")

    xcal = [xcal0 + i * xcalstep for i in range(ncal)]
    ycal = []
    for i in range(ncal):
        _x = xcal[i]
        yl = 0.0
        for k in range(nfunc):
            yl += ci[k] * lsqfunc(k, _x)

        ycal.append(yl)

# Plot
--cut--
```

lsq_general_copilot.py:

```
def main():
--cut--
    print("LSQ function")
    print(f"f(x) = {ci[0]:6.3g}", end='')
    for i in range(1, nfunc):
        print(f" + {ci[i]:6.3g} * {flabel[i]}", end='')
    print("")

    xcal = [xcal0 + i * xcalstep for i in range(ncal)]
    ycal = [sum(ci[k] * lsqfunc(k, xcal[i]) for k in range(nfunc)) for i in
range(ncal)]

    # Plot
--cut--
```

ポイント:
・リスト内包表記を使うと、
forループブロックを１行で書ける

if __name__ == “__main__” トリック: testプログラム

・lsq_general_copilot.py: MS365 copilotに、lsq_general.py をpythonらしく書き直してもらった

注: 両方のプログラムは完全に同一な結果を出力しないといけない
しかし、以下のプログラムが想定した動作をしているかどうかは一目ではわからない

```
def lsqfunc(i, x):
    functions = [lambda x: 1.0,
                 lambda x: cos(2.0 * x),
                 lambda x: sin(2.0 * x),
                 lambda x: cos(x),
                 lambda x: sin(x),
                 lambda x: cos(3.0 * x),
                 lambda x: sin(3.0 * x),
                 lambda x: x,
                 lambda x: x ** 2,
                 lambda x: exp(x)]
    return functions[i](x)
```

if __name__ == “__main__” トリック: testプログラム

テストプログラムの作成:

まず、正しい値を知る必要がある: lsq_general.pyの出力を確認

```
Si=
[[33.53072826]
 [-7.13701921]
 [-1.01208874]
 [-0.04994265]]
Sij=
[[101.          5.25354138   3.40619043  -5.35521261]
 [  5.25354138  51.50220514   2.2420942   -4.02339385]
 [  3.40619043   2.2420942   49.49779486  10.20367355]
 [ -5.35521261  -4.02339385  10.20367355  53.12677069]]

LSQ function
f(x) =  0.344 +  -0.17 * cos(2x) + -0.0424 * sin(2x) +  0.029 * cos(x)
```

if __name__ == “__main__” トリック: testプログラム

lsq_general_copilot.pyの出力が想定通りか、
テストプログラム lsq_general_test.py を作成して確認

```
--cut--
from lsq_general_copilot import lsqfunc, mlsq

def test_lsqfunc(m, x):
    assert lsqfunc(0, x) == 1.0
    assert np.isclose(lsqfunc(1, x), np.cos(np.pi))
    assert np.isclose(lsqfunc(2, x), np.sin(np.pi))
--cut--

def test_mlsq(m, xs, ys):
    ci = mlsq(xs, ys, m)
    assert len(ci) == m
    assert np.isclose(ci[0], 0.344, atol = 0.01)
    assert np.isclose(ci[1], -0.17, atol = 0.01 )
```

```
if __name__ == '__main__’:
--cut--
    test_lsqfunc(nfunc, x)
    test_mlsq(nfunc, xs, ys)
```

テストプログラム lsq_general_test.pyを
実行しても、
・lsq_general_test.pyの main()は呼び出されない
・lsq_genera.pyの main()は実行される

行列計算

感想

感想: 行列を含む煩雑な計算を手軽に実行できることが便利

注意:

- リスト型ではベクトル、行列計算はできない

- 行列計算には癖があるので、matrix.ipynb, matrix.pyのように事前に確認しておきましょう

基本文法を使った行列計算

和: C = A+B:

```
  for ix in range(nx):
      for iy in range(ny):
          c[ix][iy] = a[ix][iy] + b[ix][iy];
```

積: 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];
```

numpy.ndarrayでの行列計算

- numpy.ndarray()型では直接の行列演算が可能。
  しかし、演算子、関数に注意

- 同じ次元の行列・ベクトルの和・差
  +, - 演算子を使う
- 行列、ベクトルとの積
  @演算子を使う

1次元配列 (ベクトル) が入る場合は注意

va1= $\begin{bmatrix}1\\2\end{bmatrix}$ (縦ベクトル)　　va1 = [1, 2] (横ベクトル)

- ベクトル同士の積では、縦ベクトルと 横ベクトル[1　2]を区別しない:
  va1 @ va2 で内積計算

行列計算に関する学生の回答: Copilotに質問

プロンプト: numpy.ndarray でベクトルも行列として扱う理由は？
回答: NumPy の `ndarray` がベクトル（1次元配列）と行列（2次元配列）を統一的に扱う理由は、以下のような利点があります：
1. **一貫性とシンプルさ**：`ndarray` が1次元から多次元の配列を同じように扱えるため、異なる次元のデータを処理する際に特別な処理を追加する必要がありません。
これにより、コードがシンプルで読みやすくなります。
2. **柔軟性**：`ndarray` が多次元配列をサポートすることで、ベクトルや行列だけでなく、高次元テンソル（3次元以上の配列）なども自然に扱えます。
これにより、データの表現や操作が柔軟になります。
3. **効率性**：NumPy はC言語で実装されており、配列操作が非常に高速です。
ベクトルや行列演算を同じ`ndarray` クラスで行うことで、内部的な最適化が行いやすくなり、計算効率が向上します。
4. **統一的なインターフェース**：`ndarray` を使うことで、ベクトルや行列、さらには多次元配列に対しても同じ操作（例えば、加算、内積、スライシングなど）が可能です。
これにより、数値計算やデータ解析のコードが汎用性を持ち、再利用性が高まります。

回答は、ndarrayの便利さについて説明している。
質問の主旨と、回答がずれている

numpy.ndarrayでの行列計算

- 以前のnumpyでは、2次元配列（行列）と1次元配列 (ベクトル)の積は計算できなかった

縦ベクトルと 横ベクトルを区別しないといけないので、前者は2x1、後者は1x2の配列

va1= [[1], (縦ベクトル)　　ma = [[1, 2],
　　[2]]　　[-1, 2]]

mv = ma @ va1 = $\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ = [[1, 2],　@ [[1],　= [[5],
　　[-1, 2]]　　[2]]　　[3]]　は計算できるが、

va1 @ ma = $\begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix}$ は計算できない

mv = ma @ va1 を一次元ベクトルになおす方法:

- mv.transpose() = [[5, 3]]
　　=>　mv.transpose()[0] = [5, 3] # 転置を取って最初の1次元配列を抜き出す
- mv.flatten() = [5, 3]　　# 多次元配列を1次元化

numpy.ndarrayでの行列計算: 現在のnumpyでの変更

・現在のnumpyでは、1次元ベクトルと2次元配列（行列）の積もできる

va1= [1　2] (横ベクトル)　　ma= $\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix}$

va1 @ ma = $[1 \quad 2]\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix} = [-1 \quad 6]$

1次元配列（横ベクトル） [-1 6]で返ってくる

ma @ va1 = $\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$

1次元配列（横ベクトル） [5 3]で返ってくる

縦ベクトルと 横ベクトル[1　2]を区別しなくなっているので、要注意

その他の行列計算

内積　va1 @ va1

　　np.inner(va1, va2)

　　np.dot(va1, va2)　　# これらをお勧め

外積

　　np.cross(va1, va2)

Outer product　np.outer(va1, va2)　# 外積ではなく、要素ごとの積

逆行列　np.linalg.inv(ma)

行列式　np.linalg.det(ma)

固有値　np.linalg.eigvals(ma)

　　# 解が複素数の場合は複素数型配列で返ってくる

一次連立方程式 ma @ X = va1 の解　X = solve(ma, va1)

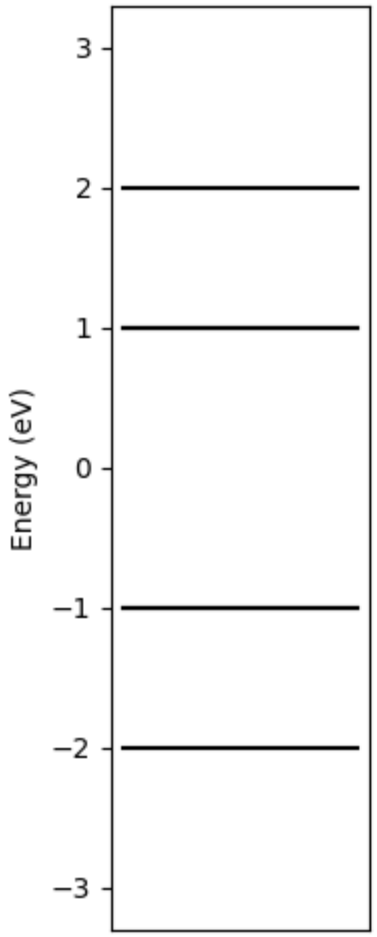
行列計算の応用: ベンゼンの電子準位

Hückel近似を使う: benzene.py

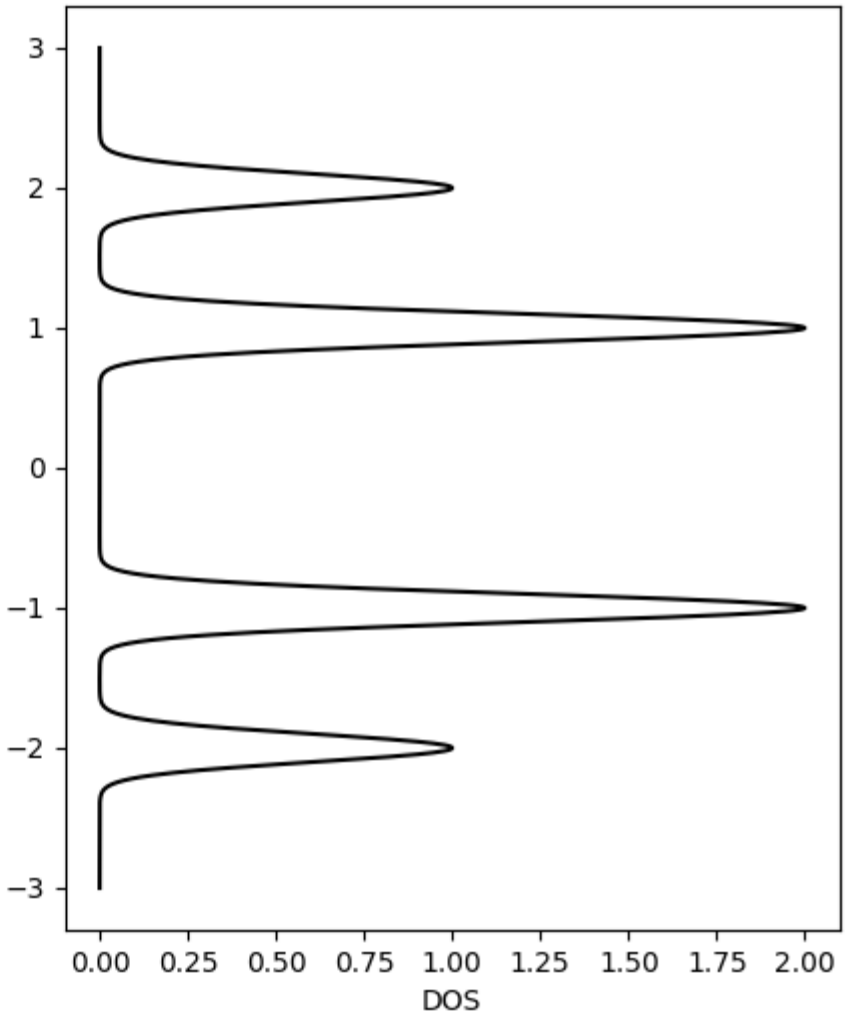
C 2pのエネルギー準位を $\varepsilon_{2p} = 0$、
共鳴積分を $\beta_{2p\pi\text{-}\pi} = 1$
としたときの
ハミルトニアンを対角化して
エネルギー準位を計算できる

```
H = np.array([[0, 1, 0, 0, 0, 1],
              [1, 0, 1, 0, 0, 0],
              [0, 1, 0, 1, 0, 0],
              [0, 0, 1, 0, 1, 0],
              [0, 0, 0, 1, 0, 1],
              [1, 0, 0, 0, 1, 0]])
```

エネルギー準位図



状態密度 (Density of states)



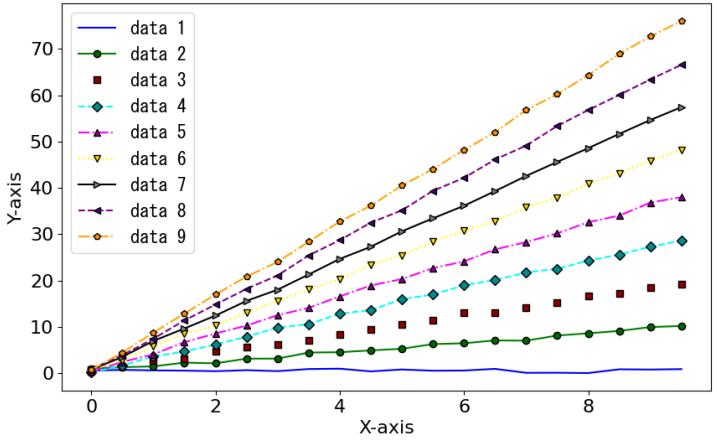
要望

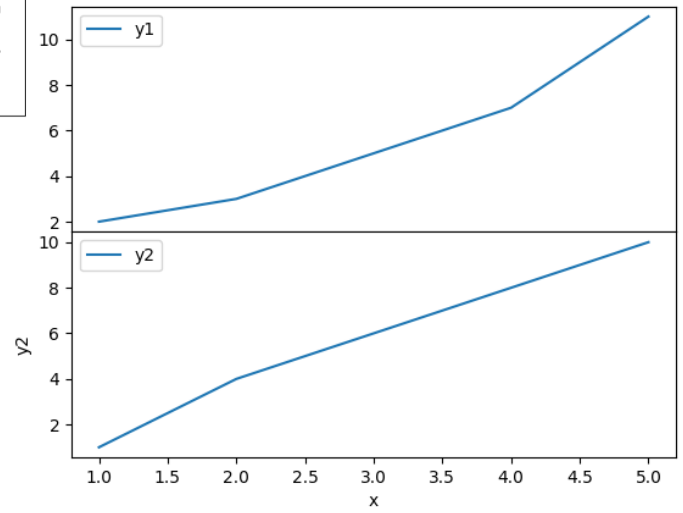
要望: 一つのグラフに何本かのグラフを描く書き方を知りたい

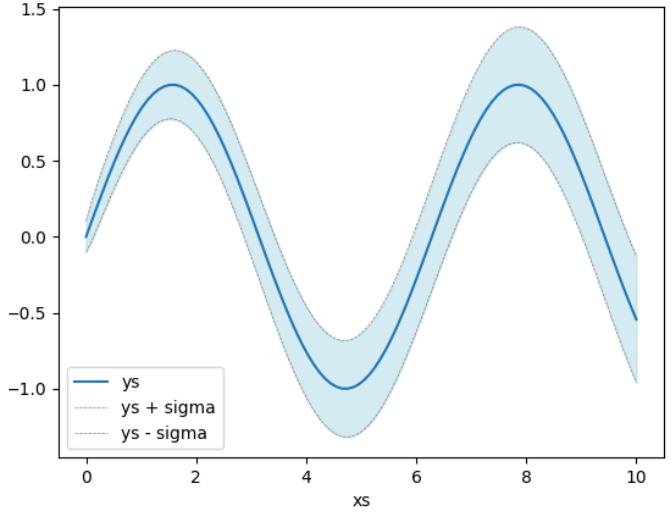
回答: プログラム例を参考

plot_manydata.py

http://conf.mdxes.iir.isct.ac.jp/D2MatE/python/python_syntax.html







質問

Q: Si, Sij(行列)の、「要素は初期化しない」とはどういうことか。

A: test_ndarray.ipynb
Sij_list = [[1, 2], [3, 4]]

Sij = np.**array**(Sij_list) = [[1.0 2.0], [3.0 4.0]]　# リスト型変数からndarayを作成
Sij = np.**zeros**([2, 2], dtype = float) = [[0.0 0.0], [0.0 0.0]]
　Siの要素には**初期値として0.0が入っている**ので、
　　Sij[0, 0] += 3
　のような計算ができる
Sij = np.**empty**([2, 2] , dtype = float) = **[[??? ???], [??? ???]]**
　Siの要素には**いくつの値が入っているかわからない (不定値が入っている)**
　　Sij[0, 0] += 3
　のような計算をしても、結果がいくつになるかわからない
　　かならず、Sij[0,0] ＝ 0.0 のように**自分で初期化した後で**
　　Sij[0, 0] += 3　などを実行しないといけない

質問

Q: x_sumなどで出てきた、+=とはどういう意味か。

A: 多くの言語で使われている演算子 (複合代入演算子)

- x += a　は　x = x + a　と同じ
- x -= a　は　x = x - a　と同じ
- x *= a　は　x = x * a　と同じ
- x /= a　は　x = x / a　と同じ

希望

希望: データベースを用いた材料探索の演習

回答: 講義では難しいので、まず、材料DBへアクセスしてみましょう

Materials Project: https://legacy.materialsproject.org/

- “Sign In or Register” でアカウントを作ってsign inする

- MP API Keyを取得すると、pymatgenライブラリを使ってpythonで検索などが可能
  ただし、pymatgenは2024年に後方互換性のないmajor version upをしているため、ChatGPT, Copilotが古いコード (動かない) を出力する可能性が高い

参考プログラム:

http://conf.mdxes.iir.isct.ac.jp/D2MatE/D2MatE_programs.html?page=tiny
の get_cif.py

非線形最小二乗法のプログラムを作る
(scipy.optimize.curve_fit())

学生のCopilotへの質問: logN = a + b * Tinvの回帰

プロンプト: pythonで logN = a + b * Tinv にフィッティングするプログラム
回答: Pythonで logN = a + b * Tinv の形のデータにフィッティングするためには、
scipy ライブラリの curve_fit 関数を使うのが便利です。以下に具体的なコード例を示します。
--cut--

```
T = np.array([1, 2, 3, 4, 5]) # 温度（例）
logN = np.array([2.3, 2.1, 1.8, 1.6, 1.5]) # ログ値のデータ（例）
Tinv = 1 / T

def model_func(Tinv, a, b):
    return a + b * Tinv

popt, pcov = curve_fit(model_func, Tinv, logN)
```

--cut--

- 生成AIの回答は、モデルのバージョンと、それまでの学習履歴によって変化する
- この場合は、変数名 logN と Tinv から、それぞれが対数値であることと温度の逆数であると推定して回答している
- **curve_fit()は線形回帰の関数ではなく、非線形回帰なので、good answerであってもbest answerではない**

線形最小二乗法と非線形最小二乗法

線形最小二乗法:

・**フィッティングパラメータが線形**の場合のみに使える

例 : f(x) = a0 + a1 * exp(-x) + a2 / x

・**パラメータの初期値は不要**

・一回の行列計算で答えが出る

非線形最小二乗法:

・**非線形のフィッティングパラメータがある場合**は非線形最小二乗法になる

例 : f(x) = a0 + a1 * exp(-a2 * x)

例 : f(x) = a0 + a1 / (x + a2)

・**パラメータの初期値を推定して与える必要**がある

・Tayler展開で線形近似をして繰り返し計算をする

・いつ収束するかわからない (プログラム中で収束判定を行う)

・パラメータの初期値が悪いと、変な値に収束 (局所最小解) したり、
収束しなかったり、発散したりする

初期値の推定が重要

Program: nlsq_curvefit.py

```
from scipy.optimize import curve_fit

# パラメータの初期値
p0=[2.0, 2.0, 0.5]              # それぞれ、f(x) = a * exp(-(x – x0)^2 / w^2) の a, x0, w

# フィッティングする関数の定義
def func(x, a, x0, w):
    return a * np.exp(-(x - x0)**2 / w**2)

# フィッティング
popt, pcov = curve_fit(func, x_data, y_data, p0 = p0)          # p0変数に初期値を与えている

# フィッティング結果の表示
print(f"Fitted parameters: a={popt[0]}, x0={popt[1]}, w={popt[2]}")
```

Program: nlsq_curvefit.py

```
# 初期値 p0 を使って計算した y_ini
y_ini = func(x_cal, *p0)

# フィッティングした係数を使って計算した x と y のデータ
x_cal = np.linspace(-10, 10, 100)
y_cal = func(x_cal, *popt)

# グラフのプロット
plt.plot(x_data, y_data, ‘b-’, label=‘data’)          # 入力データを、青線 “b-”でプロット
plt.plot(x_cal, y_ini, ‘g--’, label=‘initial fit’)    # 初期値を、緑色鎖線 “g--” でプロット
plt.plot(x_cal, y_cal, ‘r-’, label=‘fit’)             # 計算値を、赤線 “r-” でプロット
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

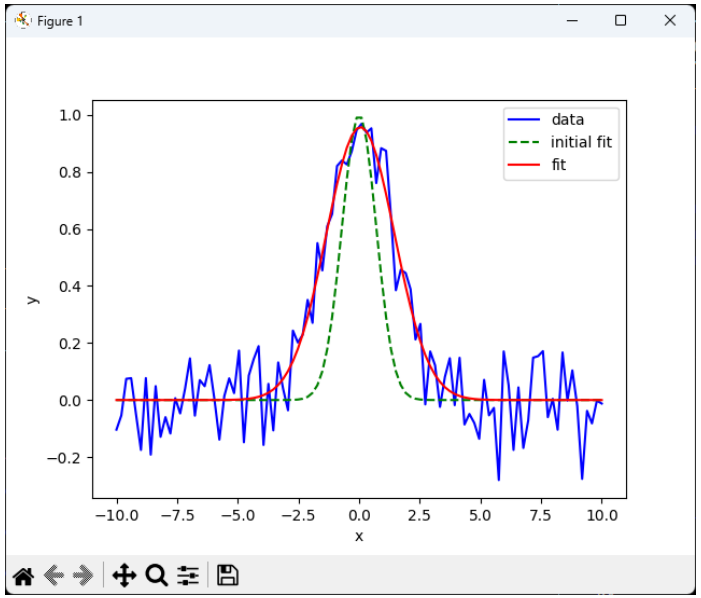
初期値を変えられるように: nlsq_curvefit2.py

```
import sys

# パラメータの初期値
p0=[2.0, 2.0, 0.5]

argv = sys.argv
narg = len(argv)
if narg == 1:
    print(f"¥nUsage: {argv[0]} [a] [x0] [w]¥n")
    sys.exit(1)
if narg >= 2:
    p0[0] = float(argv[1])
if narg >= 3:
    p0[1] = float(argv[2])
if narg >= 4:
    p0[2] = float(argv[3])

print()
print(f“Initial parameters: a={p0[0]}, x0={p0[1]}, w={p0[2]}”)   #必ずパラメータのrepeatを出力する
```

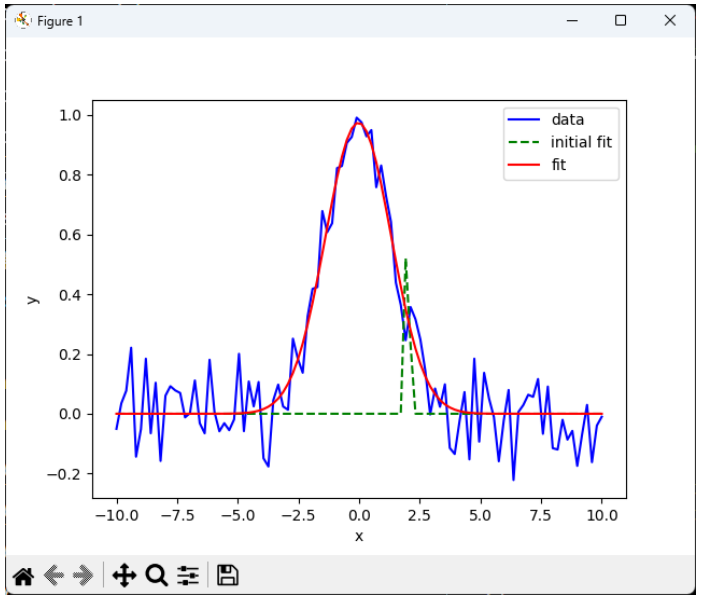
初期値の影響を調べる

Usage: python nlsq_curvefit2.py a x0 w

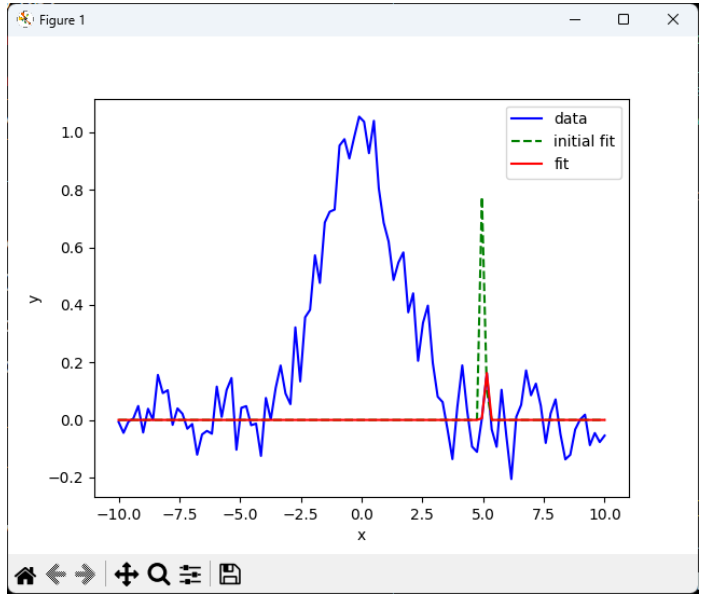
python nlsq_curvefit2.py 1.0 0.0 1.0



python nlsq_curvefit2.py 1.0 2.0 0.1



python nlsq_curvefit2.py 1.0 5.0 0.1



上級編: フィッティング過程を表示する

・フィッティング過程の情報を得るには、callback 関数を使う必要がある
　普通の関数: プログラムの実行順序に従って呼び出す
　callback関数: プログラムから呼び出される

curve_fit()ではcallbackを使えないので、**scipy.optimize.minimize()**を使う

nlsq_curvefit2_animation.py

・最小化関数 (損失関数、目的関数) を自分で定義する必要がある

```
# 残差関数
def residuals(params, x, y):
    return y - func(x, *params)

# 最小化する関数の定義
def minimize_func(params):
    return np.sum(residuals(params, xdata, y_data)**2)
```

上級編：フィッティング過程を表示する

nlsq_curvefit2_animation.py

フィッティングの実行

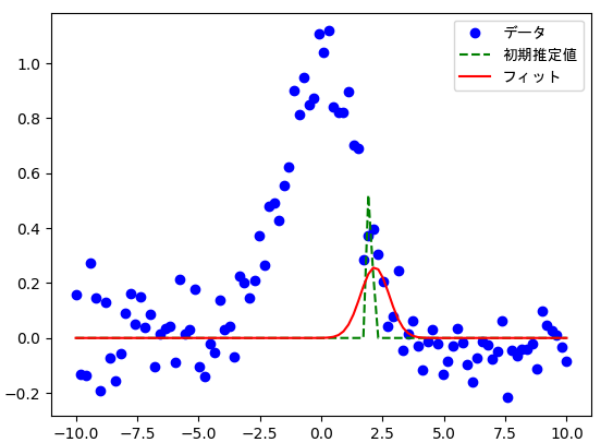
```
result = minimize(minimize_func,          #最小化関数 (損失関数、目的関数)
                  p0,                     #初期値 (解の推定値)
                  callback=callback,      #callback
                  options={'disp': True})


# callback関数
iter = 0                  # 繰り返し計算の回数 (カウンタ) をglobal変数で定義
def callback(params): # callbackには、フィッティング変数の現在値のリスト params が渡される
    global iter           # 関数内でglobal変数を変更するには、global宣言が必要
    iter += 1             # カウンタを一つ増やす
    print(f"Iteration #{iter}: params = {params}")
    line_fit.set_data(xdata, func(xdata, *params))      # グラフのデータを更新
    plt.pause(0.1)                                      # グラフの描画を更新
```

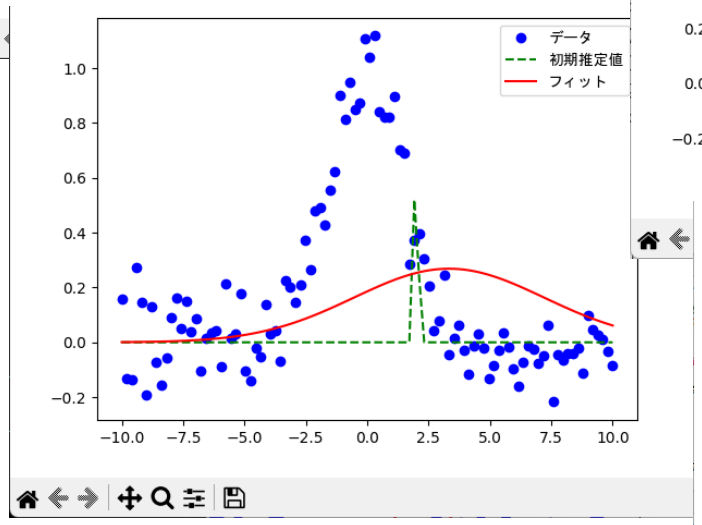
上級編：フィッティング過程を表示する

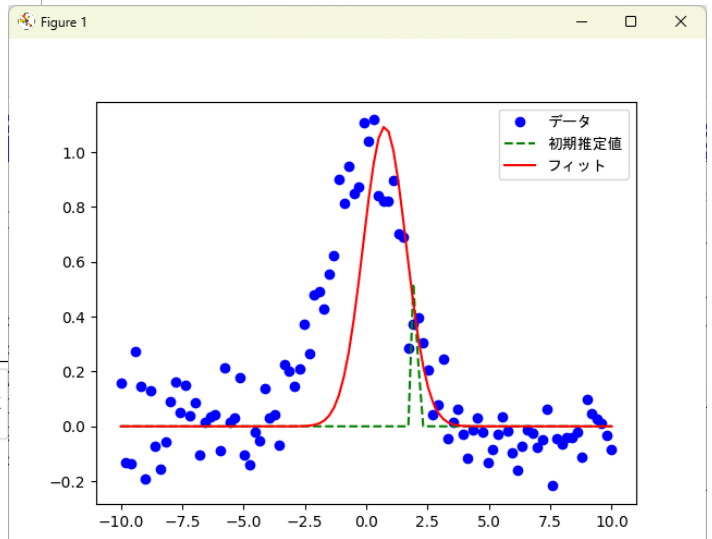
> python nlsq_curvefit2_animation.py 1 2 0.1

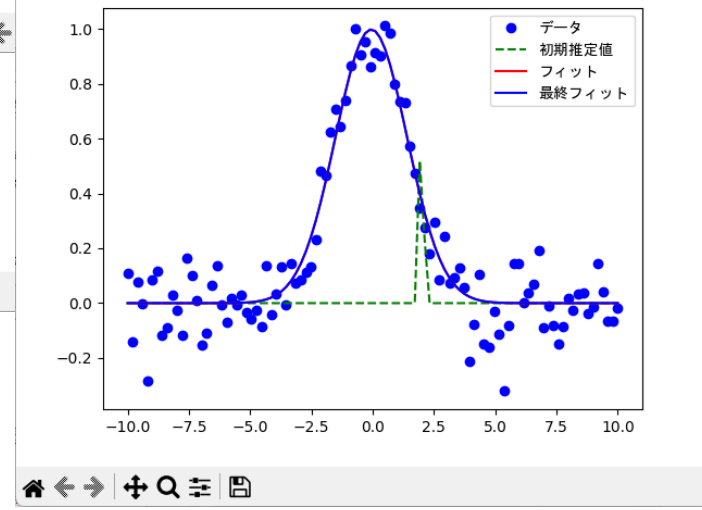
コンソール出力

```
Initial parameters: a=1.0, x0=2.0, w=0.1  初期値
Iteration #1: params = [0.99841246 2.01633665 0.07573477]
Iteration #2: params = [0.99521941 2.01224687 0.07371978]
Iteration #3: params = [0.98002372 2.00992215 0.07614509]
Iteration #4: params = [0.96144507 2.00604817 0.08234025]
Iteration #55: params = [1.22291982 0.15205843 1.80083726]
Iteration #56: params = [0.97508486 0.00253561 2.00751465]
Iteration #57: params = [ 1.02874737 -0.06627895  1.92979427
Iteration #58: params = [ 1.01240754 -0.04619863  1.97777818]
Iteration #59: params = [ 1.0102898  -0.05016889  1.98738612]
Iteration #60: params = [ 1.01006573 -0.05234199  1.99017261]
Iteration #61: params = [ 1.01008513 -0.05242515  1.9901454 ]
Iteration #62: params = [ 1.01008735 -0.05242841  1.99013922]
Optimization terminated successfully.  収束に成功
        Current function value: 1.044441  損失関数の値
        Iterations: 62
        Function evaluations: 344
        Gradient evaluations: 86
```









最小二乗法から
機械学習 (線形回帰) へ

モデル: $f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$

$x_j{}^{(k)}$: j番目のデータのk番目の記述子

$$j = 1, 2, \cdots, n$$

$$k = 1, 2, \cdots, p$$

$$y_j = f(x_j) = a_1 x_j{}^{(1)} + a_2 x_j{}^{(2)} + \cdots + a_p x_j{}^{(p)} = \sum_{k=0}^{p} a_k x_j{}^{(p)}$$

重回帰 (多変量解析): 複数変数の線形回帰

多成分解析 (複数の変数(記述子))

記述子　　$\{x^{(k)}\}=(x^{(1)}, x^{(2)}, \cdots, x^{(p)})$

データ　　$(\{x^{(k)}\}_1, \{x^{(i)}\}_2, \cdots, \{x^{(i)}\}_n)$

$\{x^{(k)}\}_j$の成分を $x^{(k)}{}_j$ と書く

モデル: $f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2$ を最小化

$$\frac{dS}{\mathrm{d}a_{k'}} = \sum_{j=1}^{n} f_{k'}(x_j)(\sum_{k=1}^{p} a_k x^{(k)}{}_j - y_j) = 0 \qquad k'=0, 1, \cdots, p$$

$$\sum_{j=1}^{n} (\sum_{k=1}^{p} a_k x^{(k')}{}_j x^{(k)}{}_j - y_j x^{(k')}{}_j) = 0$$

※ $\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j = \sum_{j=1}^{n} y_j x^{(k)}{}_j$

重回帰 (多変量解析): 複数変数の線形回帰

$$f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$$

$x^{(k)} = f_k(x)$ とすると、一般関数の整形最小二乗法になる

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2$ を最小化

解法: $\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j = \sum_{j=1}^{n} y_j x^{(k)}{}_j$

行列 (ベクトル) で表示: XA=Y $((\boldsymbol{x_{k'} \cdot x_k})(a_k) = (\boldsymbol{y \cdot x_k}))$ を解く

$$\boldsymbol{x_k} = \left(x^{(k)}{}_1, x^{(k)}{}_2, \cdots, x^{(k)}{}_n\right)$$

$$\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$$

$$\mathrm{X} = (X_{k'k}) = \left(\sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j\right) = \left(\boldsymbol{x^{(k')^T} x^{(k)}}\right)$$

$$\mathrm{A} = (A_k) = (a_k)$$

$$\mathrm{Y} = (Y_k) = \left(\sum_{j=1}^{n} y_j x^{(k)}{}_j\right) = \boldsymbol{y^T x^{(k)}}$$

機械学習の入力データ: 重回帰 (線形)

一般的な最小二乗法の入力データ random-poly.xlsx

x	y
0	4.810154
0.05	32.30261
0.1	18.78473
0.15	1.707199
0.2	15.75602

フィッティング関数 $f_i(x) = \{1, x, x^2, x^3, \cdots\}$ はプログラム中で計算する

機械学習の回帰の入力データ random-poly-ML.xlsx

x	y	x^1	x^2	x^3
0	4.810154	0	0	0
0.05	32.30261	0.05	0.0025	0.000125
0.1	18.78473	0.1	0.01	0.001
0.15	1.707199	0.15	0.0225	0.003375
0.2	15.75602	0.2	0.04	0.008

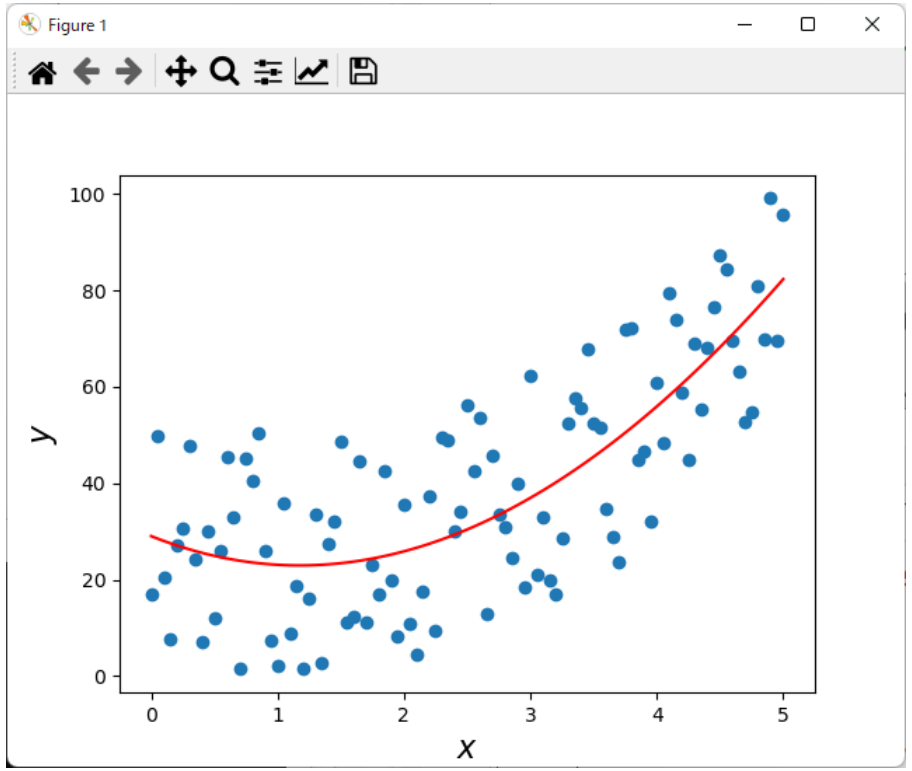
フィッティング関数 $f_i(x)$ の値は記述子として
$x^{(k)} = f_k(x)$ で与える

LinearRegression() では定数部分は勝手に
計算して intercept として返すので、
定数以外の項を記述子として与える
（x^1, x^2, x^3)

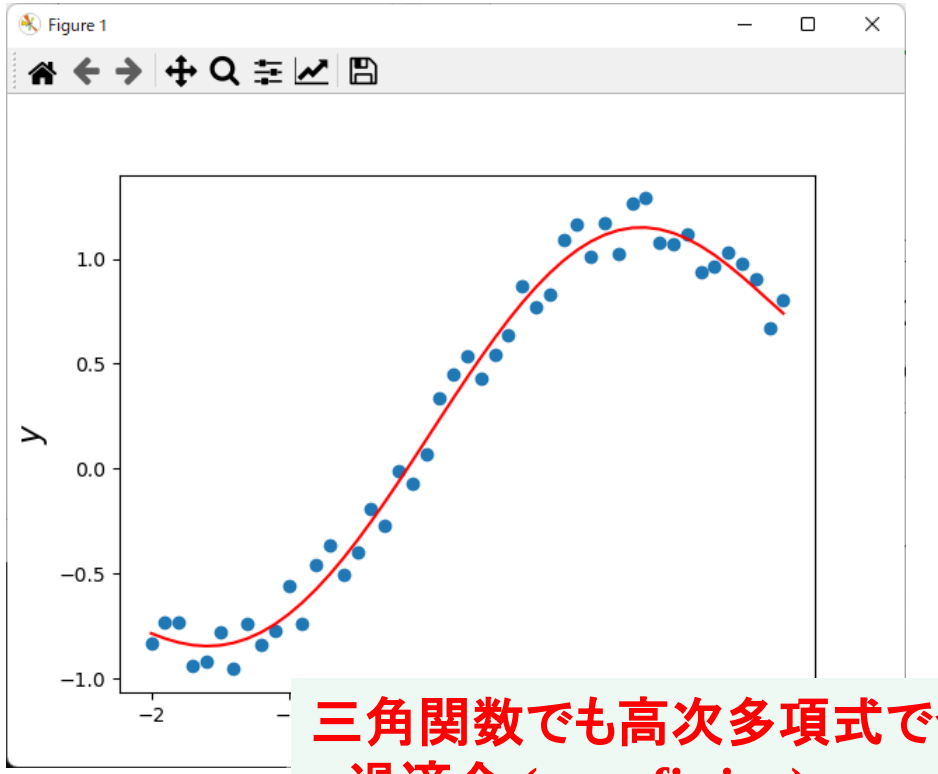
Program: lsq_polynomial_ML.py

Usage: python lsq_polynomial_ML.py input file

python lsq-polynomial-ML.py
random_poly_ML.xlsx
3次多項式で回帰



python lsq-polynomial-ML.py
random_sin_ML.xlsx
5次多項式で回帰



三角関数でも高次多項式で合わせられる
過適合 (over fitting)
過学習 (over learning)

質問: 過学習の判断方法

質問: 過学習の判断方法
A: 実験などの解析と、一般的な機械学習では状況が異なる

実験などの解析:
・データ数は多くない
・複雑なモデル (高次多項式など) を使えばフィッティングは良くなるので、
フィッティング結果からover fittingかどうかを判断することはできない
・回帰モデルの妥当性は、物理・化学などの根拠を考慮して判断する

機械学習:
・データ数は多い => over fittingの危険性が高い。
学習データ (training) と検証データ (test) に分離する
・学習データで回帰し、検証データを再現できるかどうかで性能を判断する (**汎化性能**)
・データの再現性の指標としてスコアで判断する
・over fittingしている場合、検証データのスコアが学習データのスコアより非常に大きくなる
・入力目的関数 (input) と、回帰モデルで計算した予測値 (predict) の相関図も使われる

回帰と機械学習

過剰適合していないか、予測性能はあるかを検証 (Validation)

1. 既知データを 学習データ (training) と 検証データ (test) に分ける
2. 学習データでモデルを作る
3. 学習データと検証データの予測値 (prediction) と入力値から評価を行う
4. 評価は MAE (Mean Absolute Error), MSE (Mean Squared Error), R^2 (決定係数) などで行う。

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - f(\{x_i^{(k)}\})\right|$$ 小さいほどいい

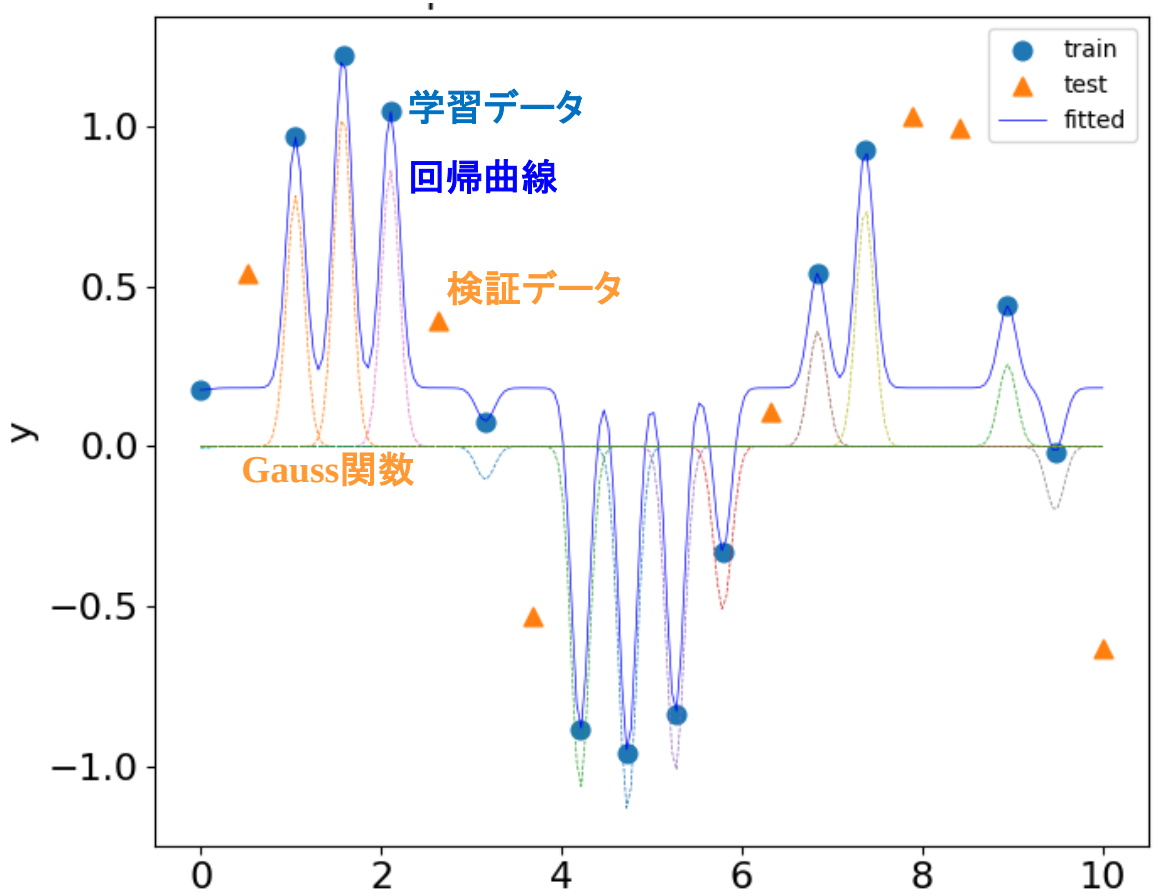
$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\{x_i^{(k)}\})\right)^2$$ 小さいほどいい

$$RMSE = \mathrm{sqrt}(MSE)$$ 小さいほどいい

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(y_i - f(\{x_i^{(k)}\})\right)^2}{\sum_{i=1}^{n}(y_i - \langle y_i \rangle))^2}$$ 1 に近いほどいい

Kernel ridge回帰

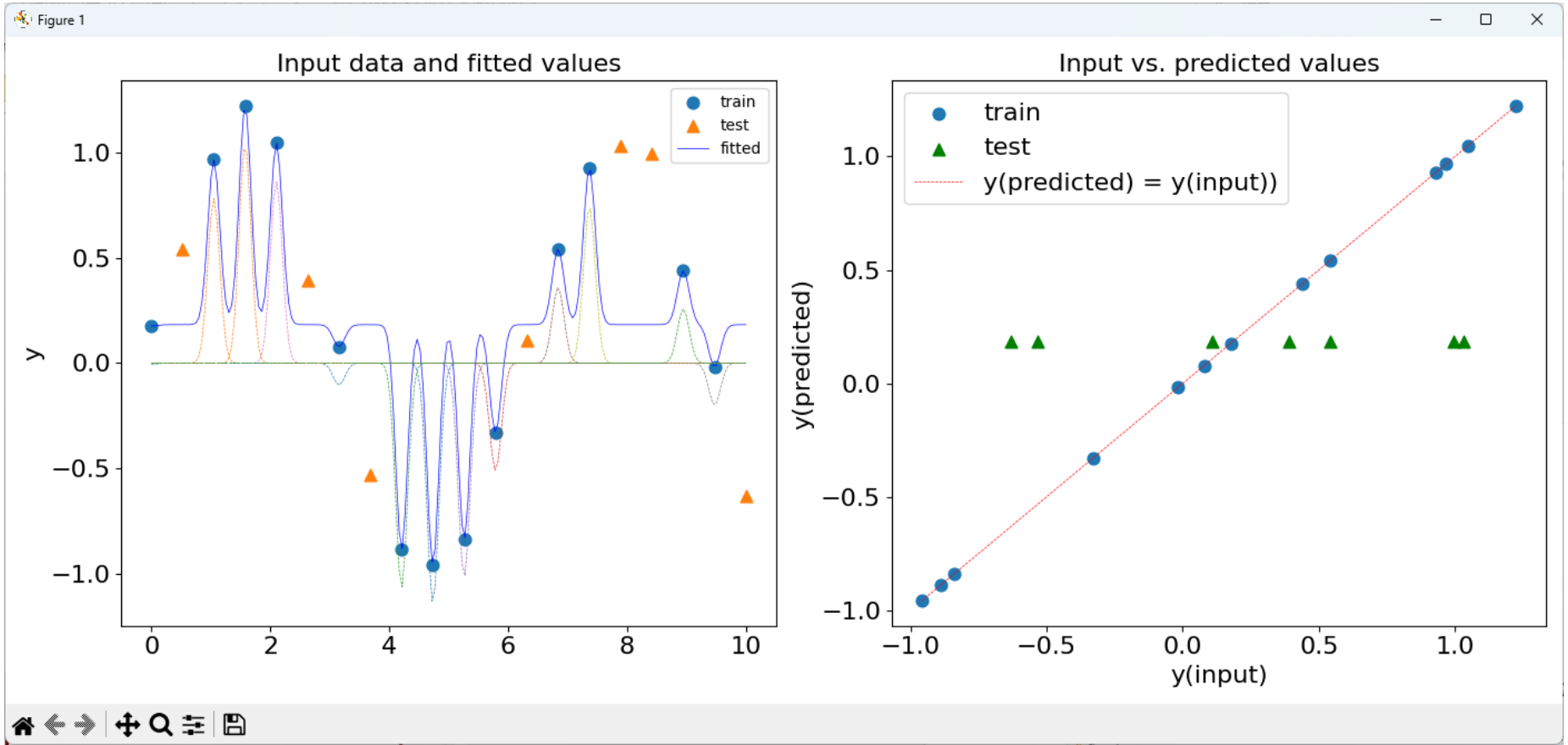
- **ノンパラメトリック回帰: 多くのパラメータをもつ柔軟なモデル関数で回帰。**
  具体的な関数形を知らなくても回帰モデルを作れる。
- **Kernel関数:** ここでは、一般関数の最小二乗法でできた基底関数 $f_i(x)$と同じと考えていい
  学習データ点 x_i 毎に、x_i を中心とし、幅 w のGauss関数 $G(x; x_i, w)$ を置く
- **Ridge回帰:**
  損失関数として残差二乗和 $\sum_{j=1}^{n}(\sum_{k=1}^{p} a_k x^{(k)}{}_j - y_j)^2$ の他に正則化項 $\alpha \sum_{k=0}^{p} a_k{}^2$ を加えて過学習を抑制する



過学習している例

> **python gaussian_ridge.py 0.7 0.1 0.001** (Gauss基底関数の幅が0.1)

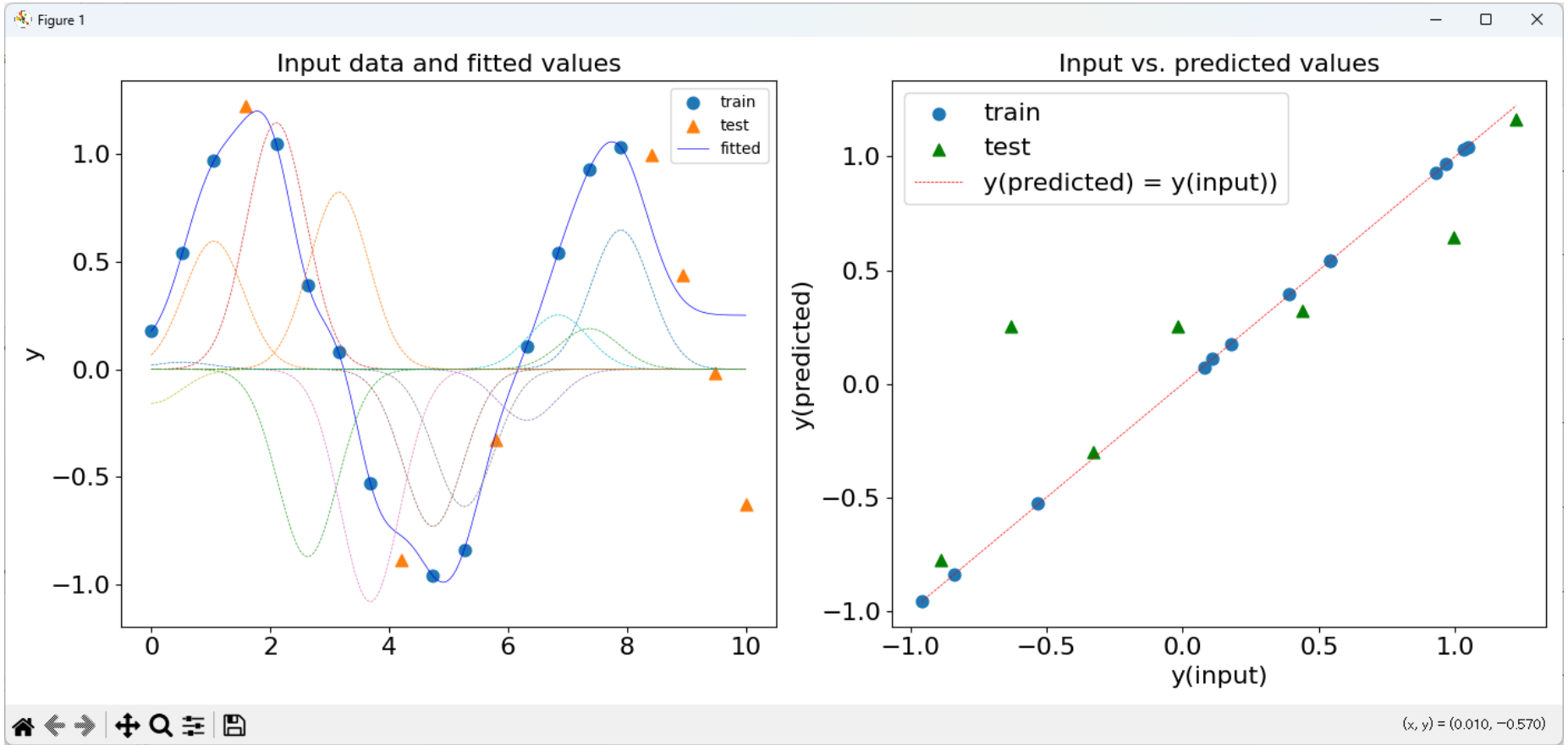
ratio_train = 0.7
w = 0.1
alpha = 0.001
Train MAE: 0.000622842, RMSE: 0.000732994, R2: 0.999999
Test MAE : 0.546841, RMSE: 0.624088, R2: -0.021084



基底関数 (幅 w) の選択で過学習を抑える

> **python gaussian_ridge.py 0.7 0.5 0.001** (Gauss基底関数の幅が0.5)

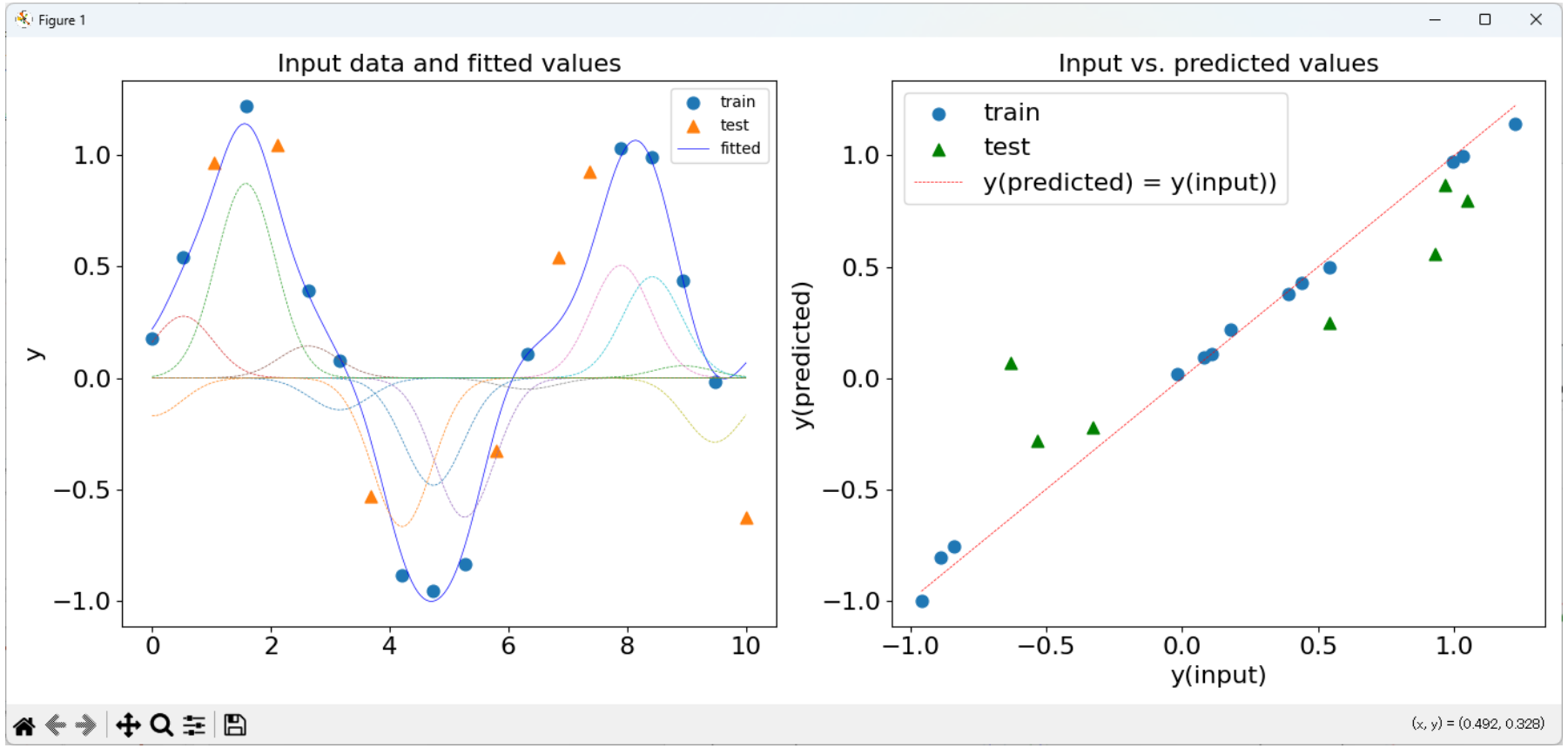
ratio_train = 0.7
w = 0.5
alpha = 0.001
Train MAE: 0.00203944, RMSE: 0.0031156, R2: 0.999978
Test MAE : 0.259566, RMSE: 0.378193, R2: 0.741828



正則化で過学習を抑える (この例では効果が無い)

> **python gaussian_ridge.py 0.7 0.5 0.1** (Ridge回帰の正則化係数が0.1)

ratio_train = 0.7
w = 0.5
alpha = 0.1
Train MAE: 0.039483, RMSE: 0.0475135, R2: 0.995301
Test MAE : 0.294835, RMSE: 0.349229, R2: 0.749321



機械学習回帰の手順

1. データを学習データと検証データにわける
2. 記述子 (と目的関数) の正規化 (標準化) を行う
3. 回帰モデル (アルゴリズム) の選択
4. 学習データで回帰を実行し、回帰関数を作成
5. 回帰関数の妥当性を検証:
   回帰関数をつかって学習データの予測値を計算し、スコアを計算
6. 汎化性能を検証:
   - 回帰関数をつかって検証データの予測値を計算し、スコアを計算
     学習データと検証データのスコアは同程度であるか？
   - 目的関数の入力値と予測値を比較する

正規化の必要性

多項式最小二乗法: $f(x) = \sum_{k=0}^{n} a_k x^k$

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

X行列の要素は 最小次数は 定数 n、最高次数は ${x_i}^{2p}$

・データが $|x_i| >> 1$ の場合、オーバーフロー、$|x_i| << 1$ ではアンダーフロー
丸め誤差など、計算誤差が大きくなる

=> 入力データを 1 のオーダーの数値に変換する必要がある

正規化の種類

よく使う最小二乗法: 正規化はしないことが多い

64bit CPUで誤差が問題になるケースは少ない

・高次の関数が必要な物理モデルは少ない (せいぜい6次)

・それほど高次の多項式を使うと、過適合の問題がでる

機械学習: 正規化あるいは標準化はほぼ必須

比較しようのない記述子 (猫の体重、身長、年齢など) から、
単位依存性が無いように目的関数を作る必要がある

正規化 (normalization): $x_i' = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$: データを 0 ~ 1の間に変換

ほかの範囲に変換してもいいが、計算誤差を減らすには
1のオーダーの範囲が望ましい

正規化: $x_i' = 2\frac{x_i - x_{\mathrm{mid}}}{x_{\max} - x_{\min}}$: データを –1 ~ 1の間に変換

標準化 (standardization): $x_i{}' = 2\frac{x_i - \langle x \rangle}{\sigma_x}$: 平均$\langle x \rangle$と標準偏差$\sigma_x$で変換

アルゴリズムによっては、
平均 0、標準偏差 1 を前提としている場合がある
標準化は必須

Ridge回帰

係数 a_i に制約をつける。Ridge回帰では a_i の絶対値が大きくならないように
目的関数にL2ノルムの二乗のペナルティ(正則化項)を加える

$$f(x) = \sum_{k=0}^{p} a_k x^{(k)}$$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k x^{(k)}{}_j - y_j)^2 + \boldsymbol{\alpha} \sum_{k=0}^{p} \boldsymbol{a_k}^2 \qquad \boldsymbol{\alpha \geq 0}$

$$\frac{dS}{da_{k\prime}} = 2 \sum_{j=1}^{n} x^{(k\prime)}{}_j (\sum_{k=1}^{p} a_k x^{(k)}{}_j - y_j) + \boldsymbol{2\alpha a_{k\prime}} = \boldsymbol{0}$$

$$\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k\prime)}{}_j x^{(k)}{}_j + \boldsymbol{\alpha a_{k\prime}} = \sum_{k=1}^{p} \sum_{j=1}^{n} x^{(k\prime)}{}_j y_j$$

$$\begin{pmatrix} \sum x^{(1)}x^{(1)} + \boldsymbol{\alpha} & \sum x^{(1)}x^{(2)} & \sum x^{(1)}x^{(3)} & \cdots & \sum x^{(1)}x^{(p)} \\ \sum x^{(2)}x^{(1)} & \sum x^{(2)}x^{(2)} + \boldsymbol{\alpha} & \sum x^{(2)}x^{(3)} & & \sum x^{(2)}x^{(p)} \\ \sum x^{(3)}x^{(1)} & \sum x^{(3)}x^{(2)} & \sum x^{(3)}x^{(3)} + \boldsymbol{\alpha} & & \sum x^{(3)}x^{(p)} \\ \vdots & & & \ddots & \\ \sum x^{(p)}x^{(1)} & \sum x^{(p)}x^{(2)} & \sum x^{(p)}x^{(3)} & & \sum x^{(p)}x^{(p)} + \boldsymbol{\alpha\lambda} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} \sum x^{(1)}y_i \\ \sum x^{(2)}y_i \\ \sum x^{(3)}y_i \\ \vdots \\ \sum x^{(p)}y_i \end{pmatrix}$$

$$X_{k,k\prime} = \sum_{j=1}^{n} x^{(k)}{}_j x^{(k\prime)}{}_j + \alpha\delta_{kk\prime} = \boldsymbol{x}^{(\boldsymbol{k})} \cdot \boldsymbol{x}^{(\boldsymbol{k'})} + \alpha I_p$$

XA=Y　を解く

scikit-learn: 回帰の手順

lsq-polynomial-ML.py :

```
#Import
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression                    # 方法 (アルゴリズム) をimport
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score        # 評価関数をimport
#データ読み込み
  df = pd.read_excel(infile, engine = 'openpyxl‘)
  x = df[labels[2:]]
  y = df[labels[1]]
#記述子の標準化
   scaler = StandardScaler()
   scaler.fit(x)
   x_scaled = scaler.transform(x)
#フィッティング
   model = LinearRegression()                                        # 方法 (アルゴリズム) を選択
   model.fit(x_scaled, y)
#フィッティング結果から計算
   y_cal = model.predict(x_scaled)
#評価
   mae  = mean_absolute_error(y, y_cal)
   mse  = mean_squared_error(y, y_cal)
   rmse = sqrt(mse)
#パラメータ
   print(f“   intercept: {model.intercept_}”)                    #  定数項。決定木、NNなどには無い
   for iv in range(len(x_labels)):
      print(f“   {x_labels[iv]:>10}: {model.coef_[iv]:12.4g}”)  # 多項式回帰などの場合、記述子の係数。
                                                                  # 決定木、NNなどには無い
```

scikit-learn: 回帰アルゴリズム

線形回帰系:

多重線形回帰

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

Ridge回帰

```
from sklearn.linear_model import Ridge
model = Ridge(alpha = 0.05)
```

LASSO回帰

```
from sklearn.linear_model import Lasso
model = Lasso(alpha = 0.05)
```

Elastic Net回帰

```
from sklearn.linear_model import ElasticNet
model = ElasticNet(alpha = 0.05)
```

カーネル回帰系:

Kernal Ridge回帰

```
from sklearn.kernel_ridge import KernelRidge
model = KernelRidge(alpha = 1.0, kernel = 'rbf')
```

ニューラルネットワーク系:

多層パーセプトロン (多層ニューラルネットワーク)

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(max_iter = 1000, hidden_layer_sizes = (10,), activation = 'logistic', solver = 'sgd',
                      learning_rate_init = 0.01)
```

scikit-learn: 分類回帰アルゴリズム

分類法系:

Random Forest回帰

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
```

勾配ブースティング木 回帰

```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor()
```

サポートベクターマシーン 回帰

```
from sklearn.svm import SVR
model = SVR(kernel='linear', C=1, epsilon=0.1, gamma='auto’)
```

数値微分・数値積分

数値微分: 微分を中心差分で近似する

h を十分小さい値に選ぶ

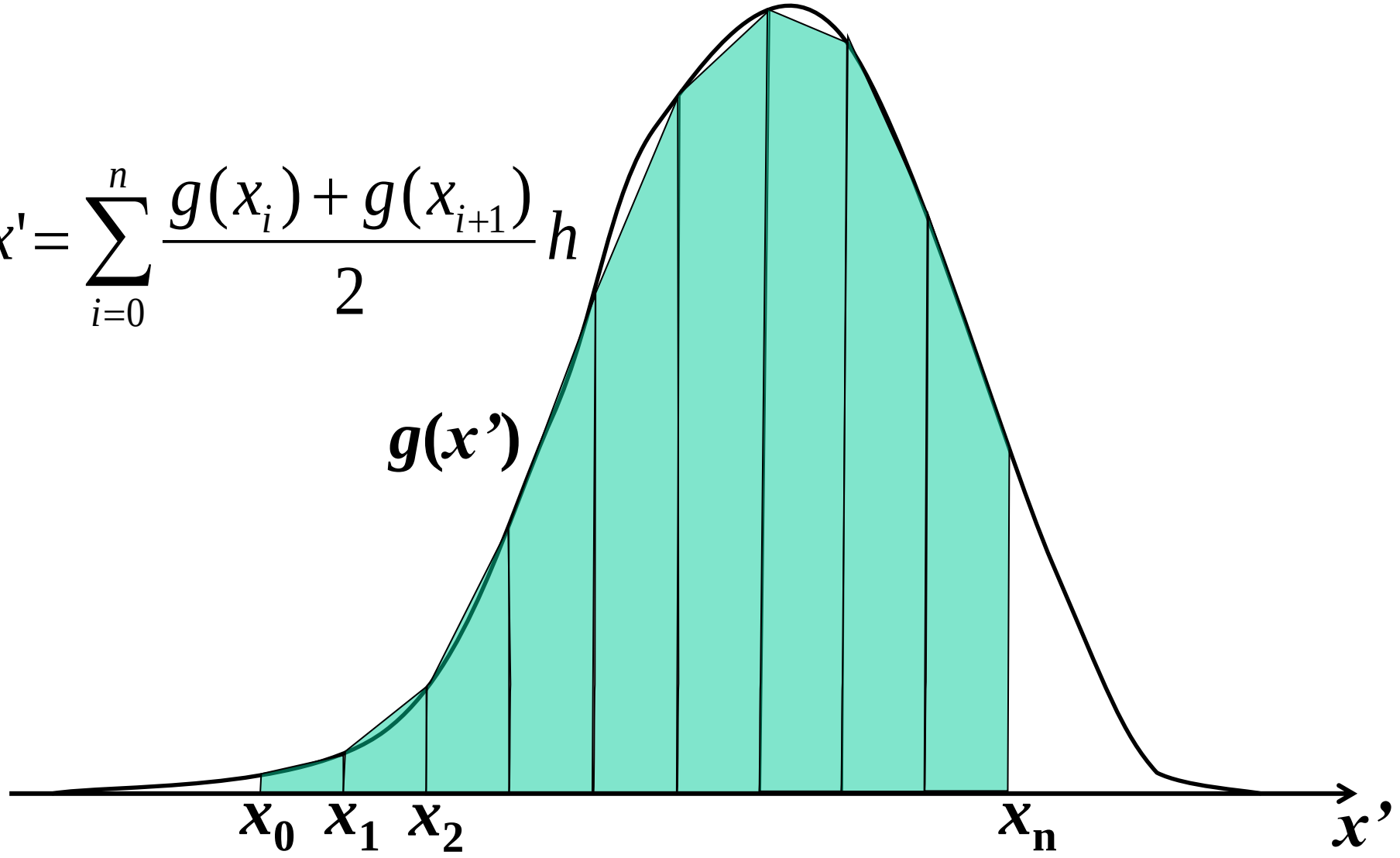
3点公式

一次微分: $\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x-h)}{2h}$

二次微分: $\frac{d^2f(x)}{dx^2} \sim \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$

数値積分: 台形で近似して面積を求める (台形公式)

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



高次の公式: Newton-Cotes公式

- **Trapezoid formula** (台形則)

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

- **Simpson formula** (Simpson則)

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

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

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

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

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

トラブルシューティングを含む生成AI利用

エラーが発生するプログラムを生成するプロンプト

Pythonは、デフォルトでは UTF8の日本語ファイルしか読めない
=> Shift-JISのCSVファイルを作り、
これを読みこんでフィッティングするプログラムを生成する

CSVファイル作成のプロンプト例:
ノイズを含み、pseudo-Voigt関数のピーク2つをもつデータを生成してください。x軸は0~90で、0.1刻み、コピペしてCSVファイルにできる形で出力してください。
=> **make_peaks_csv.py** => 実行して **peaks.csv** を作成し、ラベル行に日本語を入れ、Shift-JISで保存

プログラム作成のプロンプト例:
CSVファイル peaks.csv を読み込み、複数のGaussian関数でフィッティングするpythonプログラムを作ってください。argvが存在すれば、読み込むファイルの変数 infile、ピーク数 npeakをargvで設定してください
=> **peaks_fit.py**

エラーメッセージ

> **python peaks_csv.py**

エラーメッセージ:

Traceback (most recent call last):

--cut--

  File "D:¥doc1.current¥doc.講義¥01材料計算科学基礎¥20250120第11回¥03¥peaks_fit.py", line 28, in main

    data = pd.read_csv(infile)

           ^^^^^^^^^^^^^^^^^^^

--cut--

File "parsers.pyx", line 2053, in pandas._libs.parsers.raise_parser_error

  File "<frozen codecs>", line 322, in decode

UnicodeDecodeError: 'utf-8' codec can't decode byte 0x82 in position 1: invalid start byte

学生が実験データに対応して修正したプログラム

実験.py: 実験値.xlsxに $\cos^2(x)$でフィッティングするように修正

課題プログラムとしては完璧。

実際に使うプログラムとしては、**汎用性**を考慮するとbetter。

- 可変変数のハードコーディング (プログラム中での決め打ち) はしない。
  起動時引数、設定ファイルなどで変更できるようにする
- 入力データで設定できる変数 (グラフのxlabel, ylabelなど) も
  ハードコーディングしない

学生が実験データに対応して修正したプログラム

MS365 Copilotで修正:

プロンプト:
以下のプログラムに次の修正をしてください
#グラフで日本語が表示できるように
#入力ファイル名をargvでうけとれるように
#初期値p0はプログラムの最初の方で定義し、argvでも受け取れるようにする
#Excelのラベル文字列決め打ちではなく、1列目をx、2列目をyに入れる
#グラフのxlabelは、Excelの1列目のラベル、ylabelは2列目のラベル
#titleは入力ファイル名
ここに **実験.py** をコピペ

結果（Copilotの出力をさらに修正）:**実験_copilot.py**

より柔軟に

- x,yデータの列番号を起動時引数で変更できるようにする
  ただし、起動時引数の数が増えるとわけわからなくなる
  - Usageを表示するようにする
  - batchファイル、shell scriptを使う
  - argparseを使い、keywordで起動時引数を
    指定できるようにする

- Graphical User Interface で選択できるようにする
  (tkinter、PySimpleGUI)

その他、プログラムを改善するプロンプト

プロンプト: 以下のプログラムに例外処理を追加してください

例外処理 : ファイルが無いのにopen()しようとしたなどの
エラーを処理するプログラムコード

プロンプト: 以下のプログラムにコメント、docstringを追加してわかりやすくしてください

プロンプト: 以下のプログラムをなるべく関数でまとめたうえ、リファクタリングしてください

リファクタリング: プログラムを読みやすくしたり例外処理を加えて、
保守性・堅牢性の高いプログラムになおすこと

プロンプト: 以下のプログラムのコードレビューをお願いします

プロンプト: 以下のプログラムのtestプログラムを作ってください