DxMT 東京科学大学拠点 D²MatE 2026年度チュートリアルシリーズ
東京科学大学 総合研究院 フロンティア材料研究所学術講演会

生成AIとToyプログラムで理解するバンド理論・データ解析

第２回第２部　プログラムの構造化・リファクタリング

神谷利夫

東京科学大学
総合研究院
元素戦略MDX研究センター

Institute of SCIENCE TOKYO

本チュートリアルには、
Microsoft365 Copilot / Chat GPT / Gemini / Claude等
を使用した結果が含まれています

プログラムに要求される性能

1. 正確性
2. 速度 (pythonでは優先順位は低い)
   - 最適化：無駄な処理を減らし、パフォーマンスを向上。
   - リソース管理：CPUやメモリの使用量を抑える。
3. 保守性（メンテナンス性）
   - コードの可読性：他の開発者が理解しやすいコードを書くこと。
   - 適切なコメント：意図が明確になるように記述。
   - モジュール化：変更が容易になるように、機能を分割。
4. 汎用性
   - 設定ファイル、起動時引数の活用：環境に依存せず動作できるようにする。
   - 柔軟な設計：さまざまなユースケースに対応できる構造を持たせる。
5. 例外耐性 (堅牢性)
   - 適切なエラーハンドリング：エラーを捕捉し、適切に処理する。
   - フェイルセーフ設計：予期しない障害に強い構成にする。
6. 再利用性
   - ライブラリ化：共通機能をモジュールとして分け、再利用を促進。
   - DRY（Don't Repeat Yourself）原則：同じコードを何度も書かない。

プログラムに要求される性能

1. 正確性
2. 速度 (pythonでは優先順位は低い)
   - 最適化：無駄な処理を減らし、パフォーマンスを向上。
   - リソース管理：CPUやメモリの使用量を抑える。
3. 保守性（メンテナンス性）
   - コードの可読性：他の開発者が理解しやすいコードを書くこと。
   - 適切なコメント：意図が明確になるように記述。
   - モジュール化：変更が容易になるように、機能を分割。
4. 汎用性
   - 設定ファイル、起動時引数の活用：環境に依存せず動作できるようにする。
   - 柔軟な設計：さまざまなユースケースに対応できる構造を持たせる。
5. 例外耐性 (堅牢性)
   - 適切なエラーハンドリング：エラーを捕捉し、適切に処理する。
   - フェイルセーフ設計：予期しない障害に強い構成にする。
6. 再利用性
   - ライブラリ化：共通機能をモジュールとして分け、再利用を促進。
   - DRY（Don't Repeat Yourself）原則：同じコードを何度も書かない。

Pythonの特長: コードの再利用性とライブラリ化

- 多くのプログラミング言語では、実行プログラムとライブラリは別
- **pythonでは1つのファイルを**
  実行プログラムとしても、ライブラリとしても使える

```
def execute():              # importしてexecute()を呼び出せる
    ….

def main():                 # 実行プログラムとして呼ばれた場合に実行される
    initialize()
    execute()
    ….

# mainガード
if __name__ == "__main__":    # 実行プログラムとして呼ばれた場合に実行する
    main()
```

再利用性: ライブラリ

ライブラリ化の利点

- 同じコードを何度も書かなくても済む (DRY: Don't Repeat Yourself)
- 使用頻度が高くなり、問題点が発掘されやすくなる
- ライブラリ化することで、同じコードを、テストやプロファイリングにも使え
- 計算エンジンとGUIを分離できる
- 同じ構造・interfaceをもつライブラリを置き換えて、動作を変えられる。

  import mymodel1 as model を import mymodel2 as model

  と書き換えると、プログラム本体を修正せず、ライブラリを切り替えられる

ライブラリを差し替える例: matplotlibのGUI backend

matplotlibでは、グラフを描画する際にGUIライブラリを使っている

TkAgg	: tkinterを使ったバックエンド
GTK3Agg	: GTKを使ったバックエンド
MacOSX	: macOS専用のバックエンド
WebAgg	: ブラウザを利用するバックエンド

interfaceが異なる場合でも、間にwrapper関数をいれて切り替えられる

再利用性を高めるための注意

ライブラリでも、関数でも、**以下の原則を守る**

- 入力 (引数) が同じであれば、戻り値も同じになる
  引数の変数名は具体的に。default引数で与えるのがbetter
  型ヒントを使うのが推奨
- 入力は変更されない
  引数にリスト、辞書やオブジェクトがあり、
  要素を変更する場合、
  関数内でコピーし、修正した変数を戻り値として返す

つまり、

- global変数を使わない
- global変数を使う場合でも、変更しない (定数のみ)

生成AIによるプログラミングの注意

コーディングスタイルが毎回異なる

- 全コードベた書きか、関数化しているか　: 関数化してくれる方が理解が容易
- __name__を使うか、使わないか　: 再利用を考えるなら、使うのがよい
- 変数をどこで初期化するか: プログラムの先頭、あるいはinitialize()関数に初期化をまとめてほしい
- 類似のプログラムでも、関数名、変数名が変わる: 共通部分をライブラリ化するのが困難

最低限の機能しか実装してくれない

- 起動時引数や設定ファイルの処理もしてほしい

生成AIで行う コード評価・修正の流れ

作った後を整える6ステップ

1 コードを読ませる



- 目的の推定
- 入出力の整理
- 処理の流れの説明
- 問題点の抽出

2 リファクタリング



- 関数分割
- main()化
- 型ヒント
- docstring
- 例外処理
- 処理は変えない

3 テスト・動作確認



- pytestの基本
- 正常系・異常系
- エッジケース
- 複数パターン生成
- 乱数seedの固定
- モデルデータ検証

4 実行レポートCLI



- 実行前後の差分記録
- ログ保存
- エラー・警告抽出
- Markdown/HTML出力
- AIによるレポート解釈

5 マニュアル作成



- 目的・概要
- インストール・環境
- 入力・出力仕様
- コマンドライン引数
- 実行例
- トラブルシューティング
- 最終整理・共有しやすく

神谷の推奨コードスタイル

謝罪: 配布しているプログラムには、コードスタイルに則っていないことや、十分な検証をしていないものがあります。

http://d2mate.mdxes.iir.isct.ac.jp/sphinx/code_style/index.html#



生成AIで行う コード評価・修正の流れ

作った後を整える6ステップ

1 コードを読ませる



- 目的の推定
- 入出力の整理
- 処理の流れの説明
- 問題点の抽出

2 リファクタリング



- 関数分割
- main()化
- 型ヒント
- docstring
- 例外処理
- 処理は変えない

3 テスト・動作確認



- pytestの基本
- 正常系・異常系
- エッジケース
- 複数パターン生成
- 乱数seedの固定
- モデルデータ検証

4 実行レポートCLI



- 実行前後の差分記録
- ログ保存
- エラー・警告抽出
- Markdown/HTML出力
- AIによるレポート解釈

5 マニュアル作成



- 目的・概要
- インストール・環境
- 入力・出力仕様
- コマンドライン引数
- 実行例
- トラブルシューティング
- 最終整理・共有しやすく

生成AIによるプログラミングの例: vdPauw.py

ChatGPT-6.1 Solで作成

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq

def vdp_factor(r):
    func = lambda x: np.exp(-x*r) + np.exp(-x) - 1
    x = brentq(func, 1e-10, 100)
    f = 2 * np.log(2) / (x * (r + 1))
    return f

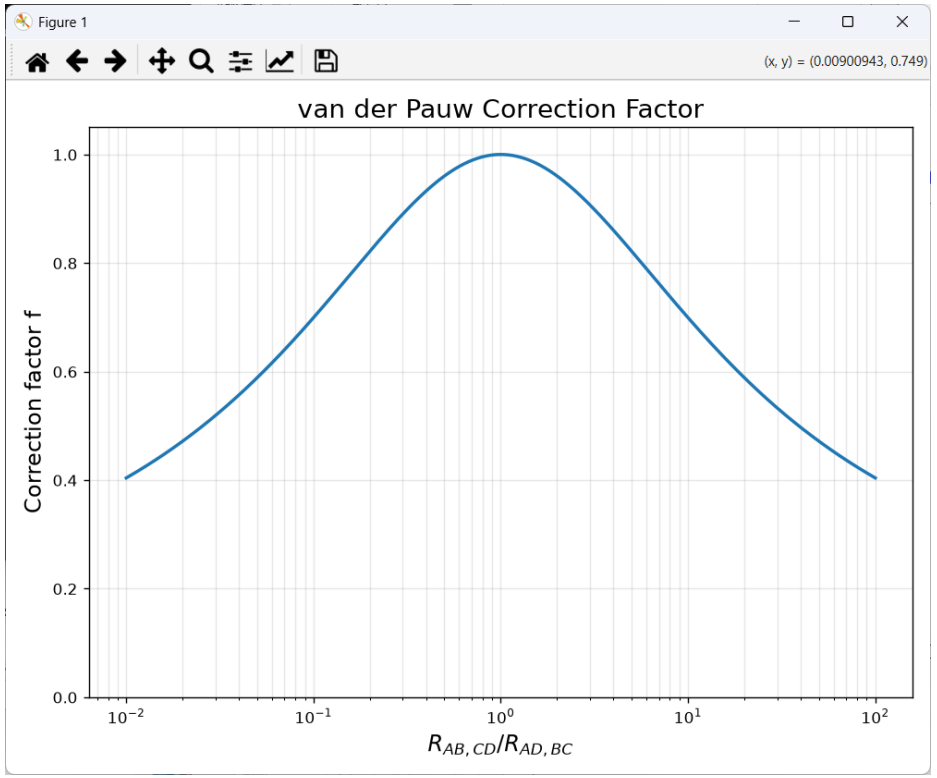
r_values = np.logspace(-2, 2, 500)
f_values = np.array([vdp_factor(r) for r in r_values])

plt.figure(figsize=(8,6))
plt.semilogx(r_values, f_values, lw=2)

plt.xlabel(r"$R_{AB,CD}/R_{AD,BC}$", fontsize=14)
plt.ylabel("Correction factor f", fontsize=14)
plt.title("van der Pauw Correction Factor", fontsize=16)

plt.grid(True, which="both", alpha=0.3)
plt.ylim(0, 1.05)

plt.tight_layout()
plt.show()
```



不満点

- べた書き
- ライブラリとしてimportしても、機能単位で呼び出せない

関数化、mainガード、リファクタリング：vdPauw_refactored.py

ChatGPT-6.1 Solで作成

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import brentq

def vdp_factor(r):
    func = lambda x: np.exp(-x*r) + np.exp(-x) - 1
    x = brentq(func, 1e-10, 100)
    f = 2 * np.log(2) / (x * (r + 1))
    return f

r_values = np.logspace(-2, 2, 500)
f_values = np.array([vdp_factor(r) for r in r_values])

plt.figure(figsize=(8,6))
plt.semilogx(r_values, f_values, lw=2)

plt.xlabel(r"$R_{AB,CD}/R_{AD,BC}$", fontsize=14)
plt.ylabel("Correction factor f", fontsize=14)
plt.title("van der Pauw Correction Factor", fontsize=16)

plt.grid(True, which="both", alpha=0.3)
plt.ylim(0, 1.05)

plt.tight_layout()
plt.show()
```

Q:添付プログラムを__name__化、関数化、リファクタリングして

```
(略)
def vdp_factor(r: float) -> float:                  # 型ヒント
    if not np.isfinite(r) or r <= 0:
        raise ValueError("抵抗比rは有限の正の値にしてください")

    def equation(x):                                # 関数内関数
        return np.exp(-x * r) + np.exp(-x) - 1

    x = brentq(equation, 1e-10, 100)
    return 2 * np.log(2) / (x * (r + 1))

def calculate_factors(r_values: np.ndarray) -> np.ndarray:   # 責務別に関数として分離
    return np.array([vdp_factor(r) for r in r_values])

def plot_factors(r_values: np.ndarray, f_values: np.ndarray):   # 描画機能を関数に分離
    fig, ax = plt.subplots(figsize=(8, 6))
(略)
    fig.tight_layout()
    return fig, ax

def main():                                         # main()関数化
    r_values = np.logspace(-2, 2, 500)
    f_values = calculate_factors(r_values)          # 主機能 (計算) 呼び出し
    plot_factors(r_values, f_values)                # 可視化準備呼び出し
    plt.show()                                      # グラフ表示

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

初期値や計算結果の詳細な出力: vdPauw_verbose.py

Q: 初期値のリピート出力、計算過程、計算結果をコンソールなどに出力して

```
(略)
def print_results(r_values: np.ndarray, f_values: np.ndarray):
    print("¥n代表値（Rは抵抗比 R_AB,CD / R_AD,BC）")
    print(f"{'R':>12} {'f':>14} {'左辺 exp(-xR)+exp(-x)':>27} {'残差 (左辺-1)':>20}")
    representatives = sorted({float(r_values[0]), float(r_values[-1])} |
                             {r for r in (0.01, 0.1, 1, 2, 10, 100)
                              if r_values[0] <= r <= r_values[-1]})
    for r in representatives:
        f = vdp_factor(r)
        x = 2 * np.log(2) / (f * (r + 1))
        lhs = np.exp(-x * r) + np.exp(-x)
        print(f"{r:12.6g} {f:14.10f} {lhs:27.15f} {lhs - 1:20.3e}")
    x_values = 2 * np.log(2) / (f_values * (r_values + 1))
    residuals = np.exp(-x_values * r_values) + np.exp(-x_values) - 1
    print(f"全{len(r_values)}点の最大絶対残差: {np.max(np.abs(residuals)):.3e}")

def plot_factors(r_values: np.ndarray, f_values: np.ndarray):
(略)
```

```
def initialize(argv=None):
    """コマンドライン引数を読み、設定値を検査する。"""
    parser = argparse.ArgumentParser(description="van der Pauw補正因子の計算")
    parser.add_argument("--max_R", type=float, default=100.0,
                        help="抵抗比Rの上限（下限0.01、既定値100）")
    args = parser.parse_args(argv)
    if not np.isfinite(args.max_R) or args.max_R < 0.01:
        parser.error("max_Rは0.01以上の有限値にしてください")
    return args

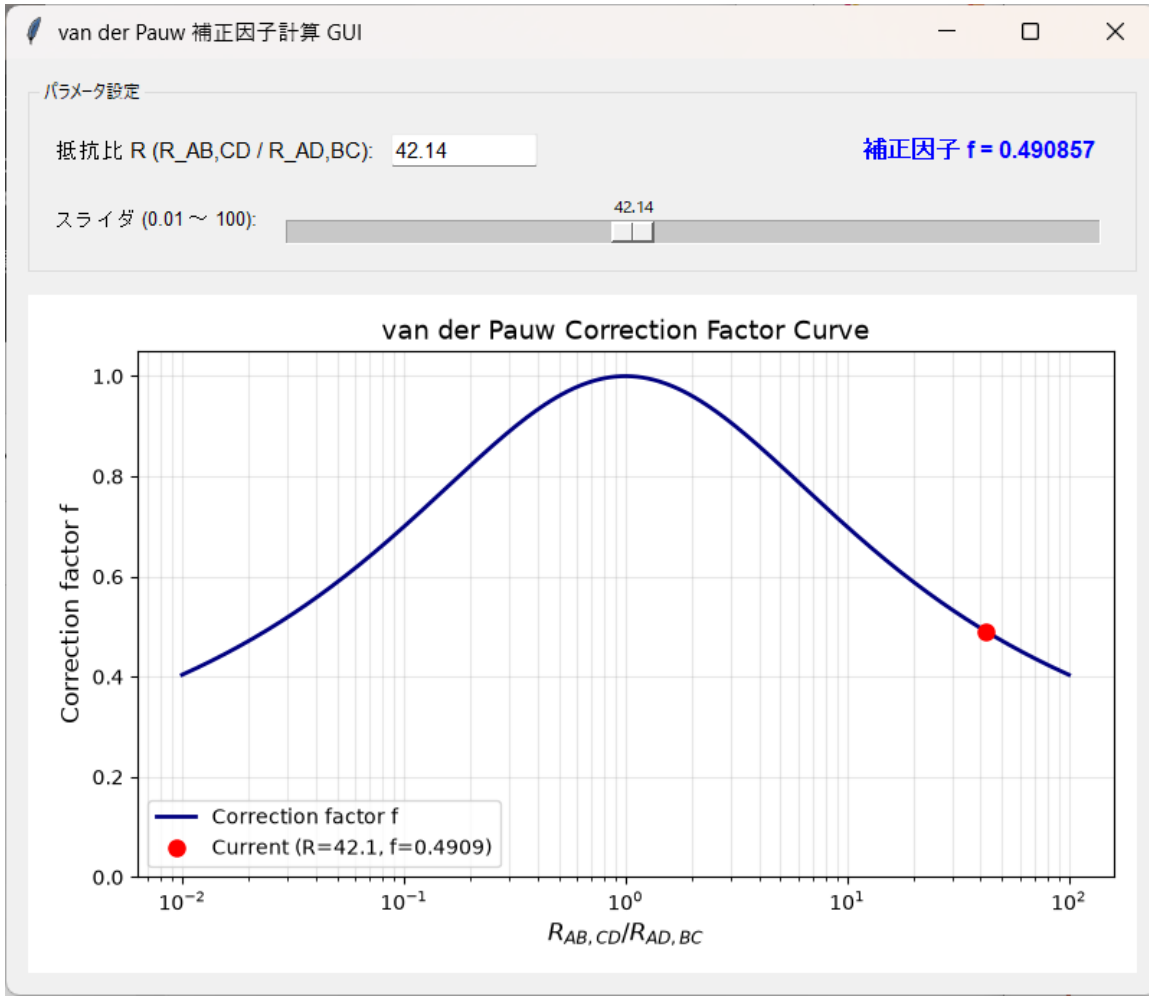

def main(argv=None):
    """設定を表示し、計算・結果表示・描画を順に行う。"""
    args = initialize(argv)
    min_R, n_points = 0.01, 500
    print(f"設定: max_R={args.max_R:g}, min_R={min_R:g}, n_points={n_points}",
          flush=True)
    r_values = np.geomspace(min_R, args.max_R, n_points)
    f_values = calculate_factors(r_values, show_progress=True)
    print_results(r_values, f_values)
    print("計算完了。グラフを表示します。", flush=True)
    plot_factors(r_values, f_values)
    plt.show()


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

vdPauw.pyをライブラリにしてGUIを作る

vdPauw_verbose.pyを添付して
プロンプト: 添付のvdPauw_verbose.pyを計算エンジンライブラリとして使い、Rを入力してfを計算するGUIを作って。
**　Rは値を指定できるとともに、スライダでも変更できるようにして、Rが変更されたら自動的にfを計算して出力して**

A: vdPauw_gui.py
Pythonの標準ライブラリである Tkinter と、グラフ描画のための Matplotlib を使用しています
（追加で matplotlib のキャンバスをTkinterに埋め込む matplotlib.backends.backend_tkagg を使用します）。



プロンプト例: プログラムを汎用化・ユーザフレンドリにする

Q: “#修正するコード” 以下のコードに、次の機能を追加してください。
ただし、以下の点に注意してください。

argparse を使って起動時引数を受け取る
入力ファイル名は input 変数で受け取り、デフォルト値を “input.xlsx” にする
input が与えられない場合は usage() を呼び出し、プログラム名・スクリプトパス・バージョン情報を表示して終了する
入力ファイルの拡張子が .xlsx でない場合はエラーを出して終了する
出力ファイル名は output 変数で受け取る
output が与えられていない場合、input の拡張子を “-out.xlsx” に変えて使う
引数の受け取り・変換は関数として独立化し、決定した変数だけを戻り値として返す
ファイルパス操作には pathlib.Path を使用する
引数を読み込んだ後、プログラム名・バージョン・設定変数を整形して repeat 出力する
処理の途中経過がわかるコンソール出力を行う
途中経過の出力は、見やすいフォーマット（区切り線やインデント）で表示する
開始時刻と終了時刻を表示する
プログラム終了時には必ず usage() を実行する（正常終了・異常終了どちらでも）

#修正するコード
(ここにコードをコピペ)

プロンプト例: pythonらしいコード (可読性・保守性優先)

Q: “#修正するコード” 以下のコードを、Python らしいスタイルでリファクタリングしてください。
ただし、以下の点に注意してください。

```
# 可読性・保守性を最優先する
# __name__ == "__main__" ブロックを使い、ライブラリとしても利用できる構造にする
# __main__ ブロックからは main() 関数のみを呼び出す
# 標準的なライブラリ (os, sys, pathlib, math, numpy, scipy, matplotlib, csv, pandas, openpyxl) を優先的に使用する
# 例外処理は __main__ ブロックでのみ行い、main() は正常系の処理だけを書く
# __main__ ブロックでは try～except～finally を用い、
#   try 内では main() の呼び出しだけを行い、
#   except では traceback を表示し、
#   finally では終了時刻と usage() を必ず実行する
# 下位関数では例外を捕捉せず、main() に伝播させる
# ファイル存在確認など簡単に回避でき、実行を継続できる例外処理は if 文を優先する
# 必要に応じて try 文を「条件判断のためだけに」使用してよい
# 下位関数で例外を握りつぶすことは禁止する
# グローバル変数は、共有変数が必要な場合のみ、namespace 変数 1つだけを許容する（例: cfg = SimpleNamespace()）
# 多重リスト内包表記など、可読性を損なう書き方は禁止する
# 関数は「1関数1責務」を徹底し、同じ入力なら同じ出力を返す純粋性をできるだけ保つ
# すべての関数に docstring と型ヒントを付ける
# ネストはなるべく浅くする（早期 return を許可）
# 科学計算コードとして読みやすい構造を優先し、PEP8 準拠は必須としない

#修正するコード
(コードをコピペ)
```

リファクタリング: 処理を変えずに整える

main化・引数化・docstring・例外処理を加え、扱いやすい形にします。

リファクタリングでおこなうことの例

1. ハードコーディングの削除: **argparse**を使う
2. global変数に依存しない: global変数は定数のみ
   プログラムの設定は **argparseの戻り値にまとめる**
   計算用の設定は**@dataclass** (以前: simplaNameSpace) で、項目・型・既定値を定義する。
   @dataclassを使うと、定数の実装も可能 (frozen=True)
3. **関数化**: 計算本体を、設定・入出力・描画から分離する
   通常実行・テスト・ベンチマークで、同じ計算関数を使う
4. 理解しやすい関数 (副作用が無い: 同じ引数なら同じ結果を返す)
   必要なデータと条件を、引数で受け取る
   計算結果を戻り値で返す
   入力配列の書換えや乱数を使う場合は、扱いを明確にする
5. Mainガード: ライブラリとして再利用可能にする

変えてはいけないこと: 機能、動作

- 基準プログラムの代表的な入力データ、設定値、主要な出力と、可能なら実行時間を残す
- リファクタリング過程では、物理式や解析手法を変える修正・変更はしない
- 整理した後に、**同じ条件で結果が一致するかテスト**する

定数の定義方法のおすすめ: @dataclass(frozen=True)

```
from dataclasses import dataclass

@dataclass(frozen=True, kw_only=True)
class Params:
    Eg: float = 1.12       # eV
    me: float = 1.08       # m0単位
    ND: float = 1e16       # cm^-3

params = Params(Eg=1.2, ND=1e17)
```

指定	効果	使いどころ
frozen=True	作成後の属性の再代入を禁止	計算中に設定を固定したい
kw_only=True	生成時にキーワード引数を必須化	引数の順番による取り違え防止
slots=True	任意の属性追加を制限し、省メモリ化	必要に応じて採用

基本構造：main()は処理の流れだけ

役割	担当する処理
main(argv=None)	引数・設定・入出力・主要関数の呼出し
initialize / read_input	入力の読込み、設定の確認、必要な準備
calculate(data, params)	計算を実行し、結果を戻り値として返す
write_outputs / plot	保存・表示・グラフ作成

```
def main(argv=None):
    args = parse_args(argv)
    data, params = initialize(args)
    result = calculate(data, params)
    write_outputs(result, args)

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

補足: 知っていると便利なプロンプト

プロンプト例Web

http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/ai/prompt/prompt_examples.html

生成AIのプロンプト例

履歴のリセット

- ▶ 過去の履歴をリセット.md (download/show)

対話型学習

- ▶ 対話学習プロンプト_目的を達成するため.md (download/show)
- ▶ 対話学習プロンプト_方程式を目的として与える.md (download/show)

講義の評価

- ▶ 生成AIの記憶をリセット.md (download/show)
- ▶ 講義資料の評価.md (download/show)
- ▶ 講義資料の評価(厳しい意見).md (download/show)

データ作成

- ▶ CSVデータの作成.md (download/show)

program作成

- ▶ 誤コードを出力する例.md (download/show)
- ▶ 線形最小二乗法プログラムの作成(minimum).md (download/show)
- ▶ フィッティングプログラムを作る.md (download/show)
- ▶ 1次元Schrodinger方程式を解くプログラム.md (download/show)
- ▶ 仕様を与えてプログラムを作る (一般).md (download/show)
- ▶ 仕様を与えてプログラムを作る (フィッティングプログラム).md (download/show)
- ▶ 仕様を与えてプログラムを作る (peakfit).md (download/show)
- ▶ 仕様を与えてプログラムを作る (キャリア濃度の温度依存性).md (download/show)

program修正

- ▶ プログラムを説明(コードレビュー).md (download/show)
- ▶ 関数の説明.md (download/show)
- ▶ 極限整合性の確認とエッジケースの提案とプログラム作成の手順.md (download/show)
- ▶ プログラムの改善(リファクタリング).md (download/show)
- ▶ プログラムを汎用化・ユーザフレンドリにする.md (download/show)
- ▶ pythonらしいコード (可読性・保守性優先).md (download/show)
- ▶ 既存のプログラムをもとに修正する.md (download/show)
- ▶ 2つのコードのスタイル・定義を合わせる.md (download/show)

ドキュメント作成

- ▶ マニュアルの作成.md (download/show)
- ▶ 詳細な教科書を作成するプロンプト・手順.md (download/show)

template.pyを使ってプロンプト: peakfit_template.py

ChatGPT o4-mini

Q: 次の機能を持つpythonプログラムを、下記のtemplateを参考に作ってください
非線形最小二乗法で、擬Voigt関数を使ってピークフィットする
x,yデータをExcelファイルから読み込む
ピークパラメータの初期値から初期プロファイルを計算
非線形最小二乗法を実行
最適化パラメータで最適化プロファイルを計算
入力データ、初期プロファイル、最適化プロファイルをグラフ化

#template pythonプログラム
(ここにtemplate.pyをペースト)

初期値を設定してくれなかったので、追加プロンプト
Q: add_argumentでは、適当なdefault値を設定してください

対話学習プロンプト: 目的を達成するため

あなたは優秀なプログラミング家庭教師です。
私は初心者（または初学者）で、次のプログラムを作りたいです：

【作りたいもの】
（ここに目的を書く）

【学習方針】
・一度に完成コードは出さない
・小さな問題に分解して進める
・毎ステップで私に考えさせる
・必要ならヒントだけ出す
・私の回答を見て次に進む
・理解が曖昧な場合は戻る

【進め方】
1. まず問題を設計レベルで分解してください
2. 最初の小問題を提示してください
3. 私の回答を待ってから次に進んでください
4. 完成コードは最後にまとめて出してください

【使用言語】
（Pythonなど）

--追加オプション--
・各ステップの意図を必ず説明してください
・今のステップが全体のどこに位置するか示してください
・計算量や設計の観点も随時説明してください

対話学習プロンプト: 方程式を目的として与える

あなたは「数値解析と物理の専門家」です。
私は学生で、今から
「解くべき方程式 → 解法の候補整理 → 解法選択」
を対話的に進めたいです。

・いきなり完成コードは出さないでください
・まずアルゴリズムの候補を列挙してください
・それぞれの長所・短所・前提条件を明示してください
・私が選ぶまで結論を出さないでください