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３部　プログラムの検証と保証

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。

Why review

# なぜコードレビューが必要か

## 最低ライン: 想定される入力範囲に対して「静かなバグ」を取り除く

### 生成直後のコードで起きやすいこと

- 正確に動くかどうかわからない
- 前提条件がわからない
  ・単位など
- 例外処理が薄い
- エッジケース (異常系・境界条件) の確認が甘い

**たまたま動いただけの可能性**

### レビューで確認すること

- 目的と責務が明確か
- データの妥当性を検査しているか
- 関数単位で説明できるか
- 異常系・境界条件を想定しているか

### 最終的に得たい状態

- **正確性・妥当性が担保**されて他人が実行できる
- 結果の正当性をユーザが確認できる
- 再現性を確認できる
- エラー時に原因を追える
- マニュアルと実行記録が残る

# 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

# 使用する生成AIモデル

モデル選択
- 最終段階では信頼性の高い推論モデルを使う
- 一度で完璧を狙わず、評価→修正→検証に分ける
- 「処理内容を変えない」など制約を明記する

この資料では 以下を使用
- TFT解析プログラムなど、以前作成したプログラム・教材
  ChatGPT 5.5 Thinking (拡張モード)
- 今回作成したプログラム・教材
  ChatGPT 6 Luna (2026/10時点の最小モデルですが、問題ないことを確認)

# 便利なツール: pandoc

**Pandoc: https://pandoc.org/**

**Markdownファイルを MS-Word (.docx) や HTML (.html) などに変換できます。**

.docxに変換

> pandoc input.md -o output.docx

.htmlに変換

> pandoc input.md -o output.html -s -t html5 --mathjax (あるいは--katex)

*(逆の変換や他のフォーマットへの変換もできます)*

**注意: 数式を変換させるためには、数式を $$ LaTeXコード $$ で出力する必要がある。そのため、プロンプトでは以下のようにするのがよい**

プロンプト例:

以上の議論を、pandocでWordファイルなどに変換できるMarkdownテキストで出力して下さい。ブロック数式は $$ LaTeX $$, インライン数式は $ LaTeX$ 形式で出力してください

# 生成AIに評価させる

コードの目的、処理、問題点をAIに説明させ、人間が確認する土台を作ります。

# 前回チュートリアルで問題だったコードの検証を行う

# CopilotとChatGPTで作成した
# 半導体のキャリア密度の温度依存性を計算する
# プログラム

# Vibeコーディング (難易度★★★★): 半導体のキャリア密度の温度依存性の one-shotプロンプト

半導体のキャリア統計をシミュレートする、以下の仕様を満たすPythonプログラムを作成してください。

1.【物理モデル】
- 伝導帯、価電子帯、ドナー/アクセプター準位を考慮し、電荷中性条件 (p - n + ND+ - NA- = 0) を満たすフェルミ準位 Ef を決定する。
- キャリア密度 (n, p) は、ボルツマン近似ではなく「フェルミ・ディラック積分 (オーダー1/2)」を用いて計算すること。
- 不純物のイオン化 (ND+, NA-) は、占有関数と退化因子 (gD=2, gA=4) を用いて計算すること。

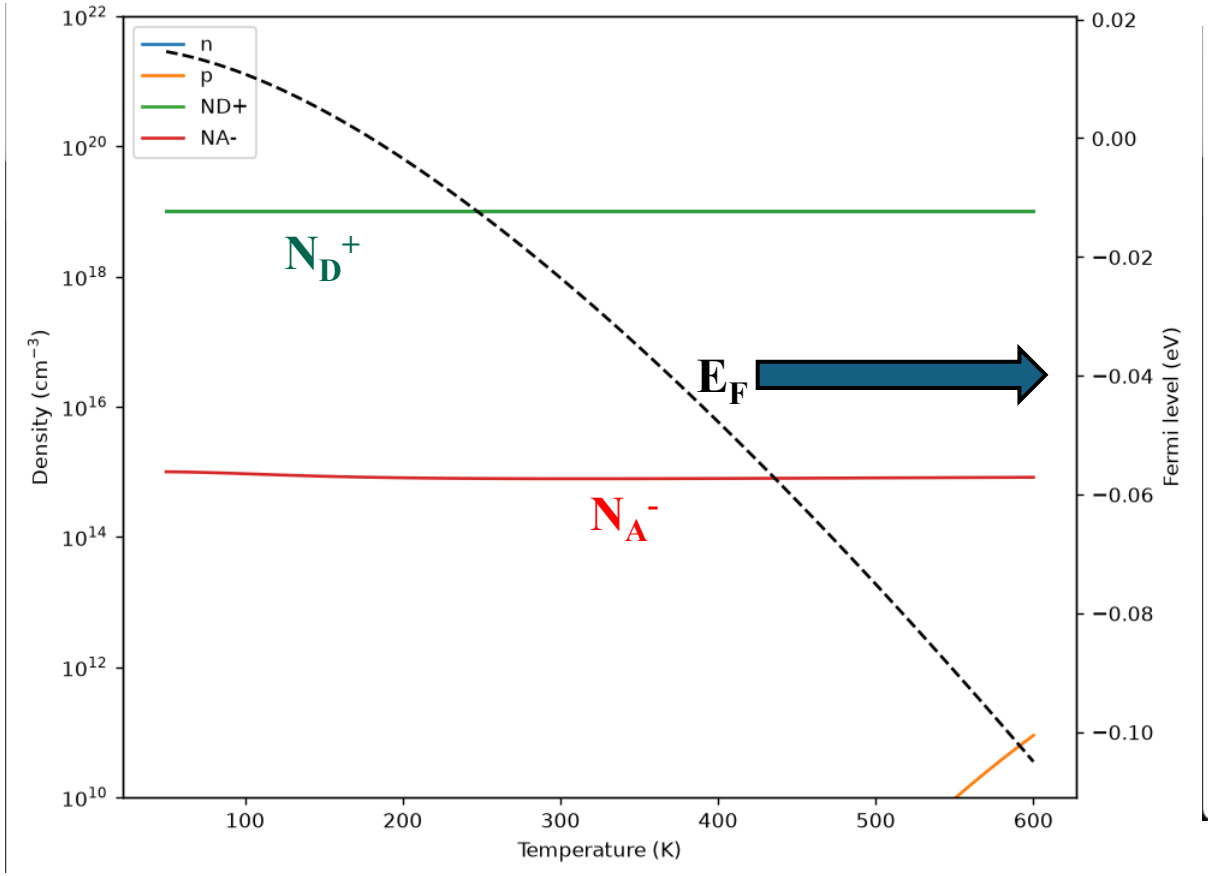
2.【数値計算の安定化】
- 強い縮退（Efがバンド内にある状態）でも計算できるよう、指数関数のオーバーフローを避けるために `scipy.special.expit` を使用した数値積分を行うこと。
- Efの決定には `scipy.optimize.brentq` (Brent法) を使用し、解を囲む範囲 (bracket) が不明な場合は自動で符号反転を確認するまで探索範囲を広げるロジックを実装すること。

3.【プログラム構造】
- `argparse` を使用して、バンドギャップ(Eg)、有効質量(me, mh)、不純物密度(ND, NA)などのパラメータを外部から与えられるようにする。
- `initialize()` 関数でパラメータの初期設定を行い、`main()` で計算とプロットを行う。
- mainガードを使用すること。
- 各関数名（m2Nc, m2Nv, fe, fh, Fj, Ne, Nh, NDp, NAm, deltaQ）は短く、物理的な意味を反映させること。

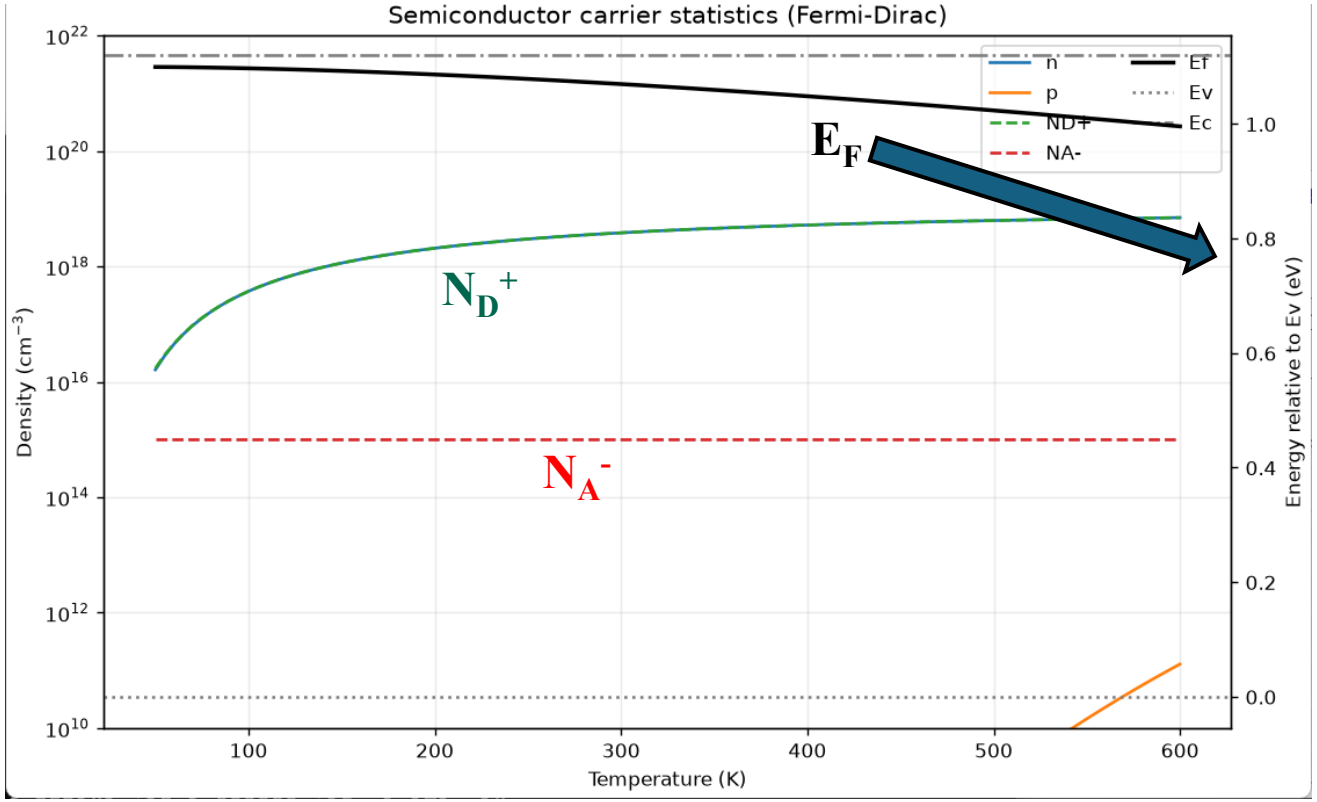
4.【出力とプロット】
- 指定された温度範囲（例: 50K～600K）でスイープを行い、結果をコンソールに表示する。
- グラフ表示の密度範囲は、Nmin, Nmaxパラメータで指定できるようにする (デフォルト Nmin=1e10, Nmax=1e22)
- グラフは2軸プロットとし、左軸に「キャリア密度・イオン化不純物密度 (ログスケール)」、右軸に「フェルミ準位 Ef (リニアスケール)」を描画すること。

# MS365 Copilot (無料版) とChatGPT-6.1 Solで作ってみた

> python Ne-T_copilot.py --ND 1e19 --NA 1e15 --Eg 1.12




> python Ne-T_sol.py --ND 1e19 --NA 1e15 --Eg 1.12




$N_D^+$をみると、Copilotでは、デフォルト値が違うか、
低温 (<200K)の計算精度が出ていないかもしれません。
原因を調べる必要があります

# 比較テスト

**本来であれば、以下の手順を踏むべきです**

- **各コードの内容を確認する**
- **正しいと確認できている計算結果 (基準値) と比較する**

**今回は、同じ動作をするはずのコードが複数あるので、まずは**
**生成AIによる比較・評価と、比較テスト (回帰テスト) を行ってみます**

- **元コードは Vibe coding (Gemini 3 Flash)で生成・検証済み**
- 元コードをもとに、Gemini 2.5で**再現用プロンプト**を作成
- そのプロンプトから
  **Copilot・ChatGPT-6.1 Sol・Sonnet5.5・Gemini 3.8 Flash版を生成**

**評価の基準: 利用目的に十分な精度が出ていればOKとする**

# 生成AIによる比較結果

**共通**

- 電荷中性条件：$p - n + N_D^+ - N_A^- = 0$ からEfを求める
- 電子・正孔密度：3次元放物線バンドのFD積分$F_{1/2}$を使う
- 不純物イオン化：ドナーgD=2、アクセプターgA=4
- 比較時はEg・有効質量・不純物密度・温度を統一する

**Copilot版で見つかった物理式の不整合**

- キャリア計算はEc=0、Ev=−Eg。不純物計算はEv=0、Ec=Egに相当
- アクセプターのイオン化率は、Efが上がると減少する実装
  正しくは、Efが上がると電子を受け取ったアクセプターが増える

**デフォルト値の違い: 束縛エネルギーは全版0.045 eV**

| 項目 | sol / Copilot | Gemini 3 / 3.8 / Sonnet |
|---|---|---|
| Eg / me | 1.12 eV / 1.08 | 同左 |
| mh | 0.56 | 0.55 |
| ND / NA（$cm^{-3}$） | $10^{16}$ / 0 | $10^{17}$ / $10^{15}$ |
| 温度点数 | sol 112 / Copilot 100 | 50 / 100 / Sonnet 200 |

# 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。

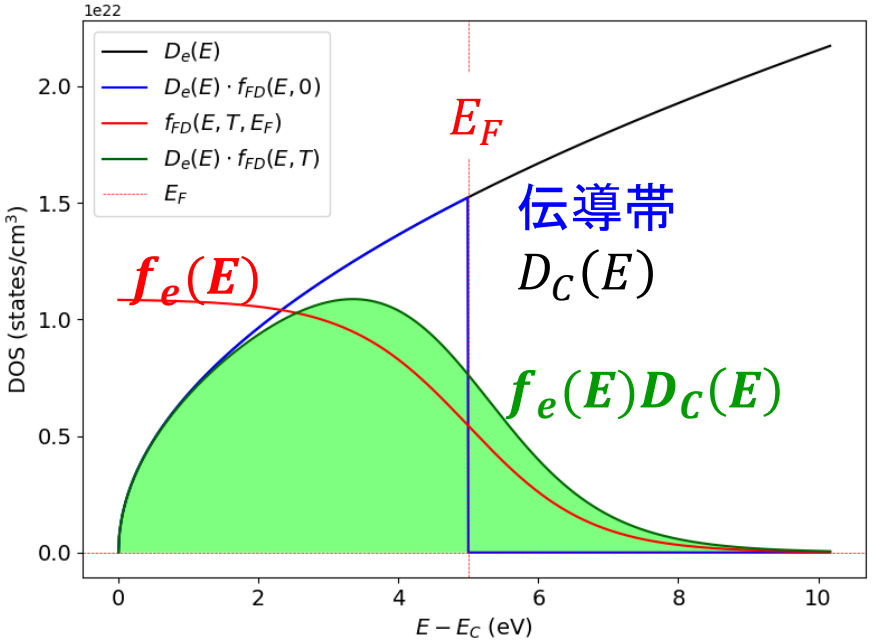
**前のチュートリアルを参照**

# テストを作って実行する

人工データで正常系・異常系・境界条件を検証し、レビューを実行可能な品質確認にします。

# 自由電子密度

自由電子: 伝導帯の状態を占有し、自由に動ける電子




**自由電子密度の計算方法**

状態密度: $D_C(E) = D_{C0}\sqrt{E - E_C}$

**Fermi-Dirac分布関数:**

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

**自由電子密度:** $\boldsymbol{n_e(T; E_F) = \int_{E_C}^{\infty} f_e(E; T, E_F) D_C(E) dE}$

左図の緑色部分の面積

**次のように書き直せる:** $\boldsymbol{n_e(T; E_F) = N_C \frac{2}{\sqrt{\pi}} F_{1/2}\left(\frac{E_F - E_C}{k_B T}\right)}$

**Fermi積分:** $\boldsymbol{F_r(\eta) = \int_0^{\infty} \frac{x^r}{1+e^{x-\eta}} dx}$ $\left(\eta = \frac{E_F - E_C}{k_B T}\right)$

**自由電子モデルのいたる所にでてくるので、**
**信頼できる関数・ライブラリにしてみる**

# プログラムを見てみましょう: 010-Ne_v1.py

(抜粋)

```
from scipy.integrate import quad                # 数値積分関数をimport
from scipy.constants import k, hbar, eV, m_e # 物理定数をimport

def calculate_carrier_density(T, Ef_ev, meff_ratio, Ec_ev=0):
--cut--
    # 状態密度
    def dos(E):
        if E < Ec:
            return 0
        return (1 / (2 * np.pi**2)) * (2 * m_eff / hbar**2)**1.5 * np.sqrt(E - Ec)

    # フェルミ・ディラック分布
    def fermi_dist(E):
        if T == 0:
            return 1.0 if E <= Ef else 0.0

        # 指数関数のオーバーフロー対策
        res = (E - Ef) / (k * T)
        if res > 40: return 0.0
        if res < -40: return 1.0
        return 1 / (np.exp(res) + 1)
```

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

$f_e(E)$

```
    # 被積分関数
    def integrand(E):
        return dos(E) * fermi_dist(E)

    # 積分範囲
    limit_upper = Ef + 20 * k * T if T > 0 else Ef
    # 積分実行。戻り値は 積分値、推定誤差が返る
    density_m3, error = quad(integrand, Ec, max(limit_upper, Ec
+ 1*eV))

    # m^-3 から cm^-3 に変換
    density_cm3 = density_m3 * 1e-6
    return density_cm3
```

$f_e(E)D_C(E)$

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

# 計算の正しさをWebアプリで確認 (Javascriptも生成AIで作成)

http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/D2MatE_programs.html?page=webapps

D²MatE Top
this page
source HTML | display HTML

source code

*Web Apps*

- Periodic table
- CIF View
- Electron properties
- FET simulator
- FET properties
- Schottky junction
- P/N junction
- Photon properties
- File viewer

*WASM/python*

- python runner
- 数式処理(1行数式)
- 数式処理(python)
- 数式処理(単位込)
- グラフ(複数/python/plotly)
- グラフ(数式/matplotlib)
- グラフ(pytnon/matplotlib)

*COVID19 simulator*

## Electronic Properties Calculator

Temperature (T): 300 K
Effective Mass ($m_{eff}$): 1 $m_e$
Band Multiplicity (M): 1 -
Carrier Density ($N_e$): 1.2848e+22 $cm^{-3}$

計算結果のNeを入力

Electric Field (E): 1 V/cm
Relaxation Time (τ): 5e-15 s

### Results

青字の項目は、プログラムのテストに使える

Mobility (μ): 8.794100 $cm^2/V \cdot s$
Conductivity (σ): 1.810245e+6 S/cm
} τを仮定すると計算できる

Effective density-of-staes ($N_c$): 2.509412e+19 $cm^{-3}/eV$ $m_{eff}$から計算できる
Plasma energy ($E_p$): 4.208908 eV
Density-of-staes: D(E) = 6.812176e+21 * sqrt(E) $cm^{-3}eV^{-1}$ $m_{eff}$から計算できる
Thermal Velocity ($v_{th}$): 116793.426933 m/s
Mean Free Path (from $v_{th}$): 0.583967 nm
Fermi level (BM shift) ($E_F$): 2.000295 eV

$E_F$の値があっていることを確認

# キャリア密度 (自由電子近似): 極限チェック

$$N_e(E_F) = \int_{E_C}^{\infty} D(E) f(E) dE = N_C \frac{2}{\sqrt{\pi}} F_{1/2}\left(\frac{E_C - E_F}{k_B T}\right)$$

**Fermi積分:** $$F_j(\eta) = \int_0^{\infty} \frac{x^j}{1+e^{x-\eta}} dx$$

生成AIを使ってプログラムを作ったら: 普通の計算結果だけでなく
- 極限近似の結果の精度を確認　　: 極限ケース
- 厳密解から極限近似の接続を確認: エッジケース

　　を確認する

近似: 金属 $E_C \ll E_F$

$$N_e(E_F) \sim \frac{2}{3} D_{C0} {E_F}^{3/2}, \; E_F = \left(\frac{3}{2}\frac{N_e}{D_{C0}}\right)^{2/3} = \frac{h^2}{{m_e}^*}\left(\frac{3N_e}{16\sqrt{2}\pi}\right)^{2/3}$$

近似: 半導体 $E_F \ll E_C$

$$N_e(E_F) \sim N_C \frac{2}{\sqrt{\pi}} \exp\left(-\frac{E_C - E_F}{k_B T}\right) \Gamma(3/2) = N_C \exp\left(-\frac{E_C - E_F}{k_B T}\right)$$

# Fermi-Dirac分布関数も簡単に実装してはいけない

**Fermi-Dirac分布関数:** $f_e(E:T,E_F) = \frac{1}{\exp(x-eta)+1}$, $x = \frac{E}{k_B T}$, $\eta = \frac{E_F}{k_B T}$

```
# もっとも簡単な実装
def fermi_dirac_distribution(x, eta, g=1.0):
    xe = x - eta
    return 1.0 / (1 + g * exp(xe))
```

問題点: xe の絶対値が大きいすぎると、overflow/underflow が起こる

```
# xeの範囲によって処理を分ける
xlim_exp = 40.0
def fermi_dirac_e_if(x, eta, g=1.0):
    xe = x - eta
    if xe < -xlim_exp: return 1.0
    if xe > xlim_exp: return 0.0
    return 1.0 / (1 + g * exp(xe))
```

**pythonにはもっと良い方法がある:**

```
    scipy.special: expit(x) = 1 / (exp(-x)+1)

def fermi_dirac_e(x, eta, g=1.0):
    return expit((eta - x) - np.log(g))
```

# Fermi-Dirac分布関数の精度と速度: fe_test.py

**tkfermi_integral.pyに実装している、以下の3つの関数を比較している**

**fermi_dirac_e_if : if文で条件分岐**
**fermi_dirac_e_npvector : numpy機能を使って高速化**
**fd_expit_scalar : expit()を呼び出す**

*注: ref値は tkfermi_integral. fermi_dirac_e_npvector() の値を使っているので、diffは精確値からの誤差ではない*

```
> python fe_test.py
=== Testing scalar_if ===
x=  -40.00  val=1.00000000e+00  ref=1.00000000e+00  diff=0.000e+00  PASS
x=   -1.00  val=7.31058579e-01  ref=7.31058579e-01  diff=0.000e+00  PASS
x=   -0.10  val=5.24979187e-01  ref=5.24979187e-01  diff=0.000e+00  PASS
x=   40.00  val=4.24835426e-18  ref=4.24835426e-18  diff=0.000e+00  PASS
x=   50.00  val=0.00000000e+00  ref=4.24835426e-18  diff=1.000e+00  FAIL
=== Testing expit ===
x=  -40.00  val=1.00000000e+00    ref=1.00000000e+00  diff=0.000e+00  PASS
x=  -20.00  val=9.99999998e-01    ref=9.99999998e-01  diff=0.000e+00  PASS
x=   -1.00  val=7.31058579e-01    ref=7.31058579e-01  diff=0.000e+00  PASS
x=   -0.10  val=5.24979187e-01    ref=5.24979187e-01  diff=0.000e+00  PASS
x=    1.00  val=2.68941421e-01    ref=2.68941421e-01  diff=0.000e+00  PASS
x=   40.00  val=4.24835426e-18    ref=4.24835426e-18  diff=1.813e-16  FAIL
x=  700.00  val=9.85967654e-305  ref=4.24835426e-18  diff=1.000e+00  FAIL
=== Benchmark (lower is faster) ===
npvector : 0.3031 sec
scalar_if: 0.1407 sec
expit    : 0.0570 sec
```

- **1e-16以下の誤差は、倍精度実数の誤差なので、問題ない**
- **3関数とも、全 x 範囲で十分な精度がある**
- **expit()は、x~700でも条件分岐を必要とせずに答えを返す**
- **expit()を使うのが最も高速・安定。コードも簡単**

# Fermi積分の実装方法

**Fermi積分:** $F_r(\eta) = \int_0^\infty \frac{x^r}{1+e^{x-\eta}} dx$

**# もっとも簡単な実装**

```
def integrand (x, eta, r):
    return x ** r * expit(eta - x)

def FermiIntegral (eta, r, limit):
    return  integrate.quad(lambda x: integrand(x, eta, r), 0, inf)[0]
```

問題点: 積分範囲、精度が不明。数値積分は計算時間がかかる

**# x – eta の範囲によって処理を分ける**

- x – eta が**正**の大きな値: Γ関数で近似 (Boltzmann近似)
- x – eta が**負**の大きな値: Sommerfeld展開で多項式近似
- 中間領域: 数値積分。eta - x = $t^2$ の変数変換をすると高速に計算できる

# Fermi積分の精度と速度 (benchmark): Fj_test.py

**以下の3つの関数を比較している**

**expit : Fj_expit(j, eta) : Sigmoid関数 expit()とquad()を使った場合**
**raw : Fj_raw(j, eta) : exp()とquad()を使った場合**
**fast : tkfermi_integral.FermiIntegral_fast(eta, j) : 最適化したFermi積分関数 (tkfermi_integral.py)**

```
> python Fj_test.py
Testing Fermi integrals Fj(j, eta)
diff tolerance: 1e-10
   j     eta      values     diff          result
-0.5  -10.00    8.04667e-05 -  8.04667e-05 -   8.04667e-05 =  3.28661e-12   PASS
-0.5   -2.50    1.37576e-01 -  1.37576e-01 -   1.37576e-01 =  5.44592e-13   PASS
-0.5    0.00    1.07215e+00 -  1.07215e+00 -   1.07215e+00 =  1.42553e-13   PASS
-0.5    2.50    2.95346e+00 -  2.95346e+00 -   2.95346e+00 =  3.30846e-13   PASS
-0.5    5.00    4.38326e+00 -  4.38326e+00 -   4.38326e+00 =  7.99361e-15   PASS
-0.5   10.00    6.29714e+00 -  6.29714e+00 -   6.29714e+00 =  2.62901e-13   PASS
----------
 2.0  -10.00    9.07
 2.0   -5.00    1.34
 2.0   -2.50    1.62
 2.0    0.00    1.80
 2.0    2.50    1.35
 2.0    5.0
 2.0   10.0
```

- **3関数とも、テスト範囲では10桁以上の精度はある**
- **fast版が最高速**
  **注意:** 整数jは解析式
  (cached)では同じ引数では再計算していない
- **小規模計算であればexpit, rawも実用的**

**これ以上の最適化は、生成AIで精確に実装するのは かなり難しい。やるなら人間の責任で！**

```
Benchmarking Fj implementations
j_test = 0.5
eta_test = -1.0
expit version          :    0.321 ms per call
raw  version           :    0.301 ms per call
fast version           :    7.060 µs per call
fast version(cached)   :    1.440 µs per call
-----------------------------------------------
eta_test = -0.1
expit version          :    0.309 ms per call
raw  version           :    0.296 ms per call
fast version           :  180.080 µs per call
fast version(cached)   :    9.630 µs per call
-----------------------------------------------
```

# 極限値・エッジケースも確認しましょう

**エッジケース**とは: **極端な値のパラメータ**でプログラムが正常動作するかどうか
**極限**: 理論やプログラムが正しい計算をしているかを確認する、**最低限の条件**

**電子密度・Fermi積分の場合:**
**極限**:

- 非縮退極限: Boltzmann近似に漸近するか。$N_C$の値を確認
- 縮退極限: Fermi-Dirac分布をステップ関数と近似した解析式に漸近するか

$$n_e(E_F) = \frac{2}{3} D_{C0}(E_F - E_C)^{3/2}$$

**エッジケース**:

- 0 K 極限:
  - プログラムは正常終了するか
  - Fermi-Dirac分布をステップ関数と近似した解析式に漸近するか
- 近似式を切り替える前後: 計算値は平滑に接続されているか

**プログラム例**: Fj_test_extreme.py には、極限・エッジケースのテストを追加してあります

# 全体の流れを追ってみよう：
# TFT解析プログラムの例

# 例題コードと実行の確認

## analyze_TFT.py

薄膜トランジスタ (TFT) の電気特性データを読み取り、
TFTの動作パラメータを計算・抽出する試作プログラム

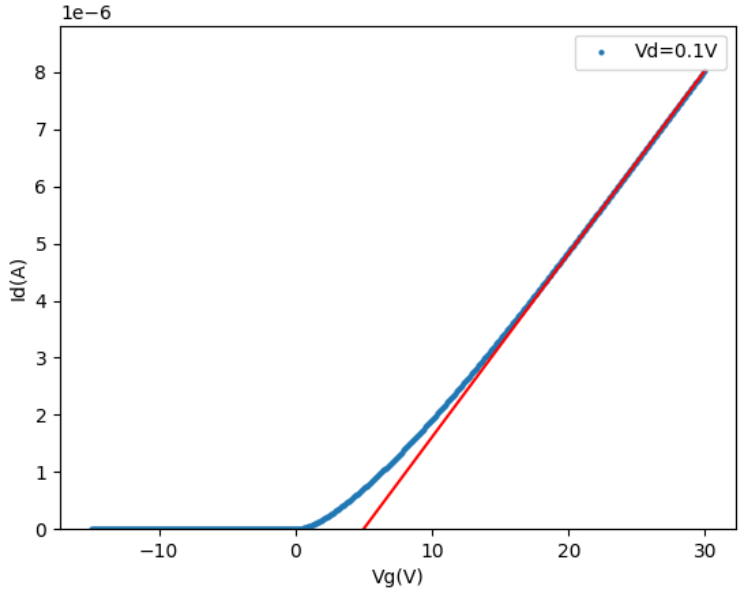
まず、実行して確認する

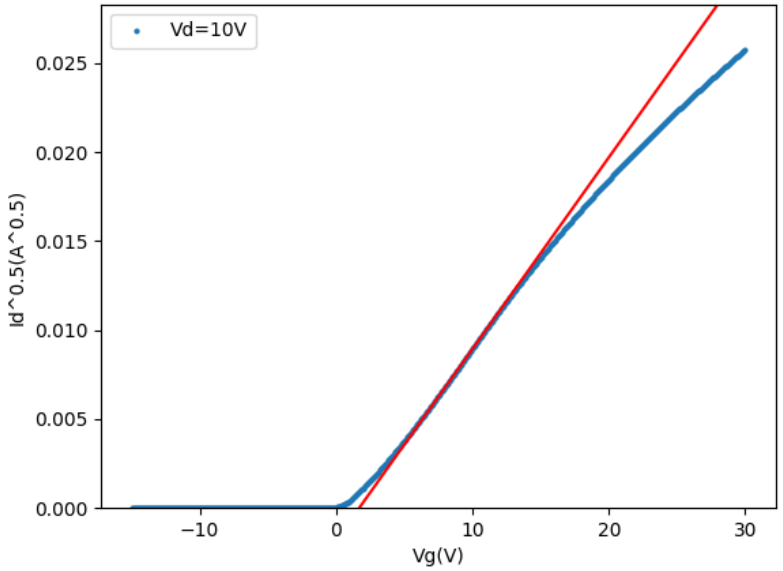
実行方法: python analyze_TFT.py

必要ファイル: IdVg-Vd0.1.xlsx、IdVg-Vd10.xlsx

出力ファイル:

- IdVg_LIN.png: 線形領域での移動度フィットグラフ
- rootIdVg.png: 飽和領域での移動度フィットグラフ
- IdVg-Vd10_analyze.csv
  IdVg-Vd0.1_analyze.csv: 解析I-Vデータ
- output.csv: 移動度等解析値




| Vth_SAT | muSAT | SS_SAT | Ion_SAT | Vth_LIN | muFE | SS_LIN | Ion_LIN |
|---|---|---|---|---|---|---|---|
| 1.69126 | 17.23837 | -0.55015 | -3.17872 | 4.98284 | 23.89296 | -0.57143 | -5.09597 |

# 生成AIによる評価

目的: 想定通りの設計、動作になっているか、問題はないかなどを確認する

**プロンプト:** 添付のコードを読み、目的、正確性、コードの読みやすさ、汎用性、
出力のわかりやすさ、例外処理、エッジケースへの対応、その他の問題点について解析し、
改善意見を出してください

添付ファイル: analyze_TFT.py

**回答: 01_code_review.md (.docxも配布)**

目的: TFTの Id–Vg 測定データから Vth、移動度、SS、Ion などを自動抽出する
試作解析スクリプト

評価:

研究用の一回限り解析スクリプトとしては、必要な計算も一通り入っている。

正確性: おおむね妥当

汎用プログラムとして使うには、入力・例外処理・エッジケース・出力説明がかなり弱い

# 問題点

問題点:

1. 入力ファイル名が固定　(ハードコーディングはNG)
2. 必須列の確認がない
3. broad except でエラー原因が消える　(必ずtracebackメッセージを出力)
4. データ点数不足や閾値未到達で落ちる可能性
5. Vth や SS の計算条件が出力に説明されない　(学会・論文発表に使えない)
6. root_folder が定義されているが使われていない
7. 未使用 import / 未使用関数が多い
8. docstringがない　(コメント・説明が不十分)
9. 関数名が短すぎる・意味があいまい
10. global変数が多い
11. 処理がトップレベルに書かれている
12. 組み込み関数名を上書き (int())
13. 最大傾き一点で Vth を決めている
14. SS の計算点が固定閾値
15. 例外処理が弱い

# 改善提案

青地: 推奨　　赤字: 絶対必要

- main() を作る
- argparse で入力ファイル名を指定する
- 出力ディレクトリを指定できるようにする
- Cox, L, W, Vd をコマンドライン引数または設定ファイルで指定する
- しきい値 1e-10, 1e-14 も引数化する
- read_input() / analyze_sat() / analyze_lin() / write_outputs() に分ける
- 関数ごとに docstring を付ける
- 物理定数や解析条件を dataclass や設定辞書にまとめる
- 計算中のpandas.DataFrameでは、log10_Ion_SAT のように列名を変えて追加する
- 単位を列名に入れる：muSAT_cm2Vs, SS_mVdec など
- 解析条件も summary.csv に出す
- 図には Vth, mobility, SS をテキスト表示する
- except Exception as e でログを出す
- 入力列が存在しない場合は明確なエラーメッセージを出す
- 閾値に到達しない場合は SS = NaN とし、warning を出す
- 有効点数が足りない場合は解析を中止する
- Vg で sort してから解析する
- 重複 Vg を検出する
- NaN の割合を出力する

# TFT解析プログラムのリファクタリングで行うこと

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し、printだけで終わらせない
   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: main()は初期設定と主要関数の呼出しで、処理の流れを示す

変えてはいけないこと: 機能、動作

- 代表的な入力データと設定値を一組保存する
- 変更前の主要な出力と、可能なら実行時間を残す
- 物理式や解析手法を変える修正は、構造の整理と分ける
- 整理した後に、同じ条件で結果が一致するか確認する

# リファクタリング・docstring追加

**プロンプト: 処理内容を変えずに、main化・引数化・docstring追加・例外処理追加を含めたリファクタリングをしてください**

**回答:**

・main() / run() / parse_args() を追加
・入力ファイルを --lin, --sat で指定可能
・出力先を --outdir で指定可能
・Cox, L, W, target current などを引数化
・関数ごとに docstring を追加
・必須列 Vg, Vd, Id のチェックを追加
・ファイル未検出、Excel読込失敗、無効な設定値に対する例外処理を追加
・except: によるエラー握りつぶしを避け、想定される失敗は NaN または明示エラーに変更
・元の analyze_IdVd_LIN 相当の関数も残置

**出力コード:** 02_analyze_TFT_refactored.py

# リファクタリング・docstring追加: 動作・再現性確認

修正プログラムを実行して確認
> python 02_analyze_TFT_refactored.py

**コンソール出力:**
Analysis finished.
Output directory: .
Vth_SAT = 1.6912599905125347
muSAT = 17.238372372360885
SS_SAT = 0.2817389488925539
Ion_SAT = 0.000662638
Vth_LIN = 4.982848054807924
muFE = 23.892963435374146
SS_LIN = 0.26826589644737664
Ion_LIN = 8.01731e-06

**元プログラムの出力と比較して正常動作を確認する**
計算結果は問題ないが、**output.csvの一部の出力は log10() で出力されている**

| Vth_SAT | muSAT | SS_SAT | Ion_SAT | Vth_LIN | muFE | SS_LIN | Ion_LIN |
|---|---|---|---|---|---|---|---|
| 1.69126 | 17.23837 | -0.55015<br>0.281740966 | -3.17872<br>0.000662644 | 4.982848 | 23.89296 | -0.57143<br>0.268268 | -5.09597<br>8.01733e-06 |

# リファクタリング：コードの修正

**02_analyze_TFT_refactored.py**

```
"""Analyze TFT transfer characteristics from Id-Vg Excel files.

This script reads two Excel files, one for the linear-region transfer curve
and one for the saturation-region transfer curve, and calculates typical TFT
parameters such as Vth, mobility, SS, and Ion.
…
"""
def parse_args(argv: Optional[list[str]] = None) -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Analyze TFT Id-Vg data and calculate Vth, mobility, SS, and Ion."
    )
    parser.add_argument(
        "--lin",
        default="IdVg-Vd0.1.xlsx",
        help="Excel file for linear-region Id-Vg data. Default: %(default)s",
    )
)

def main(argv: Optional[list[str]] = None) -> int:
    …

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

全体のdocstringを追加

argparseで引数を読み込み

処理本体をmain()に分離

スクリプトとして実行された場合だけmain()を実行

# 生成AIと相談: テストプログラムの方針

プロンプト: テストプロラムを作りたいのですが、まず方針を提案してください

回答: 03_test_proposal.md
テスト方針: 大きく分けて、次の4段階
1. 小さな関数の単体テスト
   対象例 : ・局所傾き計算　・Vth計算　・Ion/Ioff計算　・入力列チェック
2. 解析関数の数値テスト
   既知のモデルデータを人工的に作る。
3. CLIとしての実行テスト
   python analyze_TFT_refactored.py --lin ... --sat ... --outdir ...
   として正常実行できることを確認
4. エラー・エッジケースのテスト
   チェック項目　・入力ファイルが存在しない　・必要列 Vg, Vd, Id がない
   ・Id がすべて負またはゼロ　・データ点数が少なすぎる　・target current に到達しない
   ・Vd = 0 が含まれる　・NaN が含まれる　・Vg が昇順でない

# テストプログラムの方針#2

## 回答や方針が明確でない場合は、必ず確認する

プロンプト: 人工データはどのようなモデルで作成しますか

回答: 04_test_proposal2.md

飽和領域は、理想的には次の式で作ります: $I_d = \frac{WC_{ox}\mu_{sat}}{2L}(V_g - V_{th})^2$

ただし、これは (V_g > V_{th}) のオン領域だけです。 オフ領域もテストしたいので、実際には次のような piecewise モデルにします。

$$I_d = \begin{cases} I_{off} \cdot 10^{(V_g - V_{th})/SS} & (V_g < V_{th}) \\ I_{off} + A_{sat}(V_g - V_{th})^2 & (V_g \geq V_{th}) \end{cases}$$

線形領域は、低 (V_d) 近似なら次の式で作れます: $I_d = \frac{W}{L} C_{ox}\mu_{FE} V_d (V_g - V_{th})$

より少し正確には、$I_d = \frac{W}{L} C_{ox}\mu_{FE} \left[(V_g - V_{th})V_d - \frac{V_d^2}{2}\right]$

ですが、Vd = 0.1 V なら簡単のため前者で十分です。

最初のテストでは、**ノイズなし**にするべきです。
理由は、コードの基本動作確認と、理論値との比較が簡単だからです。
次に、乱数 seed を固定して、少しだけノイズを入れます。

# テストプログラム作成・実行

柔軟なテストができるように指示して、テストコードを作成

プロンプト: それでは、A~Cのケースの人工データを含めたテストプログラムを作ってください。
乱数のseedはデフォルトで固定しますが、引数でseedとノイズレベルも変えられるようにして

出力コード:05_test_analyze_TFT_refactored.py

問題: テストはすべてパスしたが、どのようなテストの結果か、詳細が出力されなかった。

# 詳細を出力するようにしたテストプログラム

**Q: どのようなテストをしたのか、詳細も出力して**

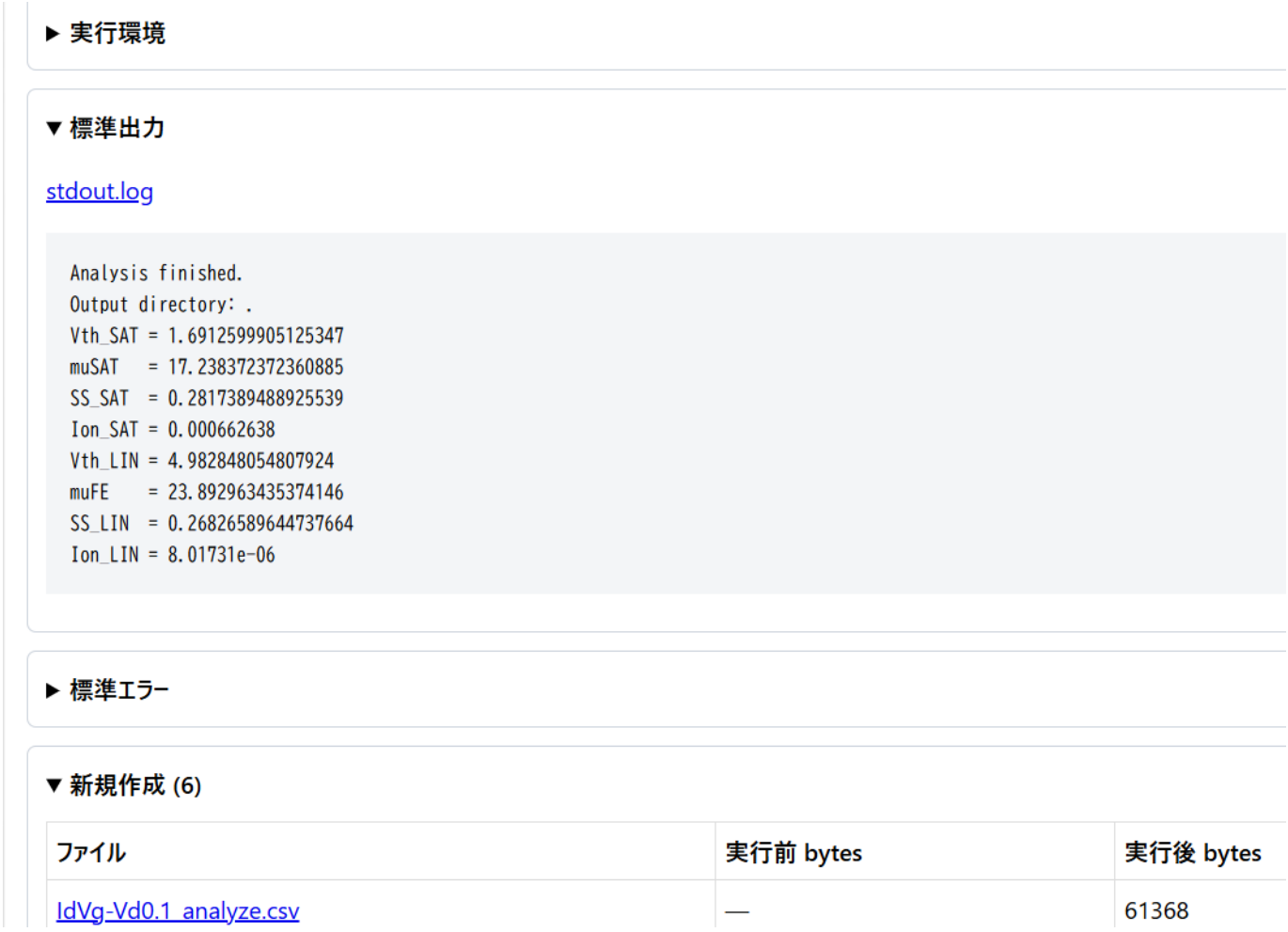
**> pytest -q -s test_analyze_TFT_refactored_verbose.py**

...Analysis finished.
--cut--
9 passed in 5.23s
--cut--
seed: 12345
noise_level: 0.005
テスト分類: Case A=理想データ, Case B=ノイズ/再現性, Case C=異常系・エッジケース
---
Case A | 理想データでしきい値電圧と移動度を再現できるか
---
目的: ノイズなしの人工TFTデータを使い、解析式に対応する既知パラメータを正しく回収できるか確認します。
データ条件:
  - Vth=2 V, muSAT=10 cm^2/Vs, muFE=8 cm^2/Vs
  - Vg=-5..15 V, 点数=201
確認項目:
  - SAT/LINのVthが真値に近い
  - muSAT/muFEが真値に近い
  - Ionが正の値になる
.結果: [PASS] test_case_a_ideal_data_recovers_vth_and_mobility

# 実行レポート作成: run_report.py

WebUIではプログラムを実行してレポートを作成できないので、実行プログラム run_report.py を作った
(コマンドライン／デスクトップツールのAIエージェントならAIだけで作れるかも)
> cd TFT¥06_analyze_TFT_refactored
> python ..¥..¥run_report.py --timeout 3600 --ignore ".git,.venv,__pycache__" --cmd python analyze_TFT_refactored.py

実行結果:
run_report_20261105_145512_882661¥
report.html



▶ 実行環境

▼ 標準出力

stdout.log

```
Analysis finished.
Output directory: .
Vth_SAT = 1.6912599905125347
muSAT   = 17.238372372360885
SS_SAT  = 0.2817389488925539
Ion_SAT = 0.000662638
Vth_LIN = 4.982848054807924
muFE    = 23.892963435374146
SS_LIN  = 0.26826589644737664
Ion_LIN = 8.01731e-06
```

▶ 標準エラー

▼ 新規作成 (6)

| ファイル | 実行前 bytes | 実行後 bytes |
| --- | --- | --- |
| IdVg-Vd0.1_analyze.csv | — | 61368 |

# マニュアル作成

プロンプト:
analyze_TFT_refactored.pyのマニュアルMarkdownを作ってください。
プログラム名は02_analyze_TFT.pyにしてください。
回答: 08_analyze_TFT_manual.md

上の.mdには数式が入っていないのでいいですが、数式がある場合は次のプロンプトを使います
プロンプト:
analyze_TFT_refactored.pyのマニュアルMarkdownを作ってください。
- プログラム名は02_analyze_TFT.pyにしてください。
- pandocでWordファイルなどに変換できるMarkdownテキストで出力して下さい。ブロック数式は $$ LaTeX $$, インライン数式は $ LaTeX $ 形式で出力してください
回答: 08_analyze_TFT_manual_math.md
(.doc、.htmlも配布)

# 生成AIがプログラムを作り、批評し、修正し、マニュアルを作る

http://d2mate.mdxes.iir.isct.ac.jp/D2MatE/?page=manual&id=tkprog


ト
xrd_fit ドキュメント
xrd_pymatgen ドキュメント
XRD_GUI ドキュメント
XRD_GUI.py 技術ドキュメント
プログラムの動作
原理
必要な非標準ライブラリとインストール方法
必要な入力ファイル
生成される出力ファイル
コマンドラインでの使用例 (Usage)
コマンドラインでの具体的な使用例
XRD_GUI 実行例
XRD_GUI プログラム仕様
crystal_XRD ドキュメント
ファイルコンバータ(再帰版)
OCR=>Markdownコンバータ
半導体関係プログラム


## XRD_GUI.py 技術ドキュメント

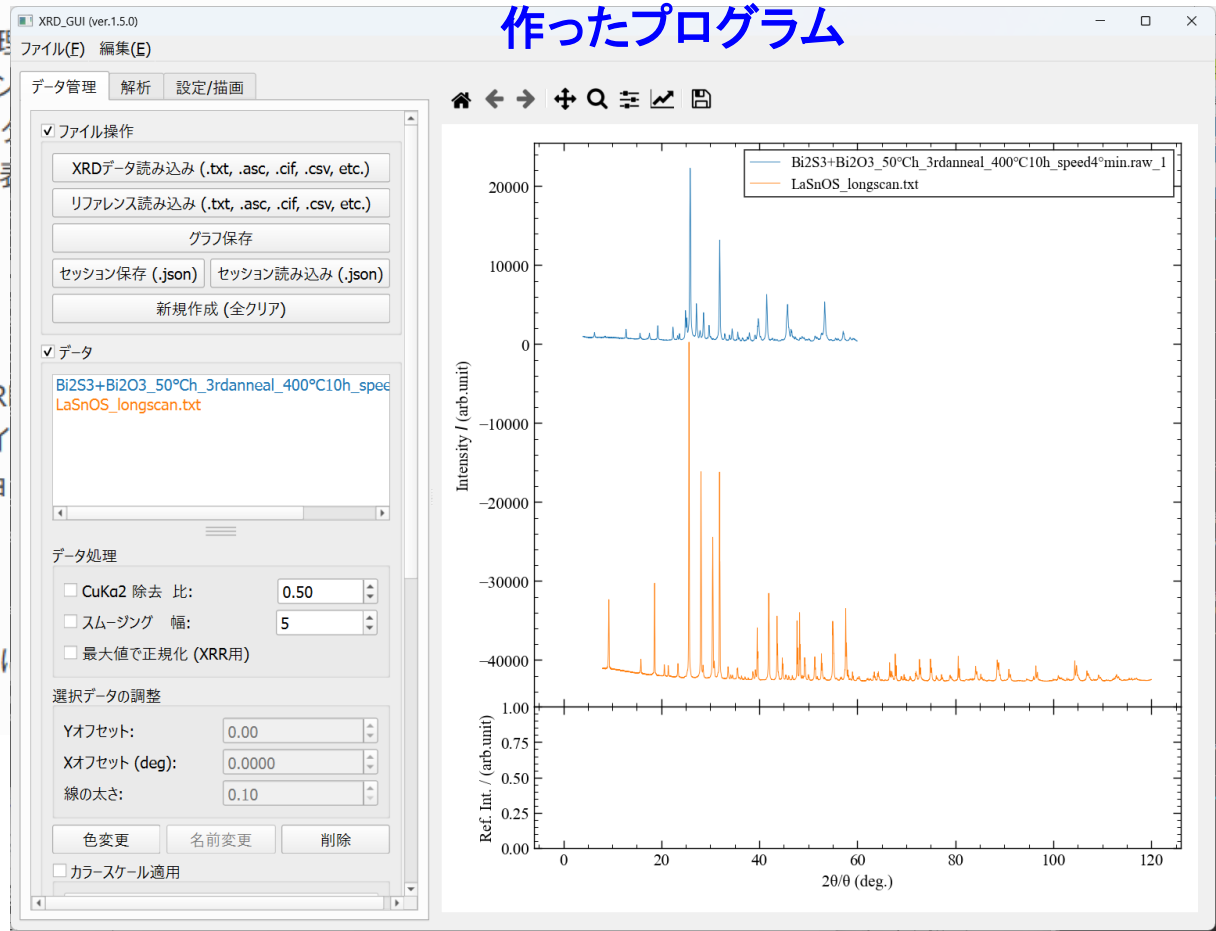
### プログラムの動作

`XRD_GUI.py` は、X線回折 (XRD) データの視覚化、処理
ィカルユーザーインターフェース (GUI) アプリケーシ
がXRDパターンを直感的に分析し、複数の測定データ
情報を素早く取得し、高品質なグラフを作成して発表
援します。

**主な機能:**

- **データとリファレンスの読み込み**: 測定されたXR
  など）および文献値のリファレンスデータファイ
  通じて簡単に読み込むことができます。セッショ
  応しています。
- **グラフ表示の柔軟性**:
  - 線形、対数、平方根のY軸スケール切り替え。
  - 複数のデータセットやリファレンスを視覚的に
    び手動オフセット調整。

小林君 (当時M1) がGeminiで作ったプログラム



# 長いプロンプト

- prompt_リファクタリング.md
- prompt_マニュアル.md
- prompt_品質確認.md

使い方：

- **Web UI**：指示ファイルと対象ファイルを添付し、
  「添付の指示に従って、対象コードをリファクタリングしてください」と依頼
- **Skills／プラグイン**：対応環境 (ChatGPT, Geminiなど)
  繰り返し使う作業手順として登録
- **AIエージェント**：Codex, Antigravity, Claude Codeなど
  - AGENTS.md/CLAUDE.md に記述
  - Skillとして登録して使う

# まとめ: 常用・共有するプログラムはコードレビュー

- エラーがなく出力結果が出たからと言って、使っていいとは限らない
  正しく動いているとは限らない
- 読めるコードでも、保守しやすいとは限らない

## 人間がやるべきこと

1. 可能な範囲でコードを追う
   (i) global変数、関数、argparseの変数名とデフォルト値を確認する
   (ii) main()関数から処理の流れを追う
   (iii) VSCodeの「参照」「定義へ移動」を使う
   (iv) pdbやVSCodeのデバッガを使う

## 生成AIに支援させること

1. まず評価させる
2. 処理を変えずに整理させる
3. 人工データでテストを作らせる
4. pytestの結果を確認・解釈させる
5. マニュアルと実行レポートを残す