非線形最適化

最小化問題と最小二乗法

最適化問題:　変数: 記述子の組　$\{x_i\}$　　目的関数: $f(x_i)$
　目的: $f(x_i)$ を最大化、あるいは最小化する

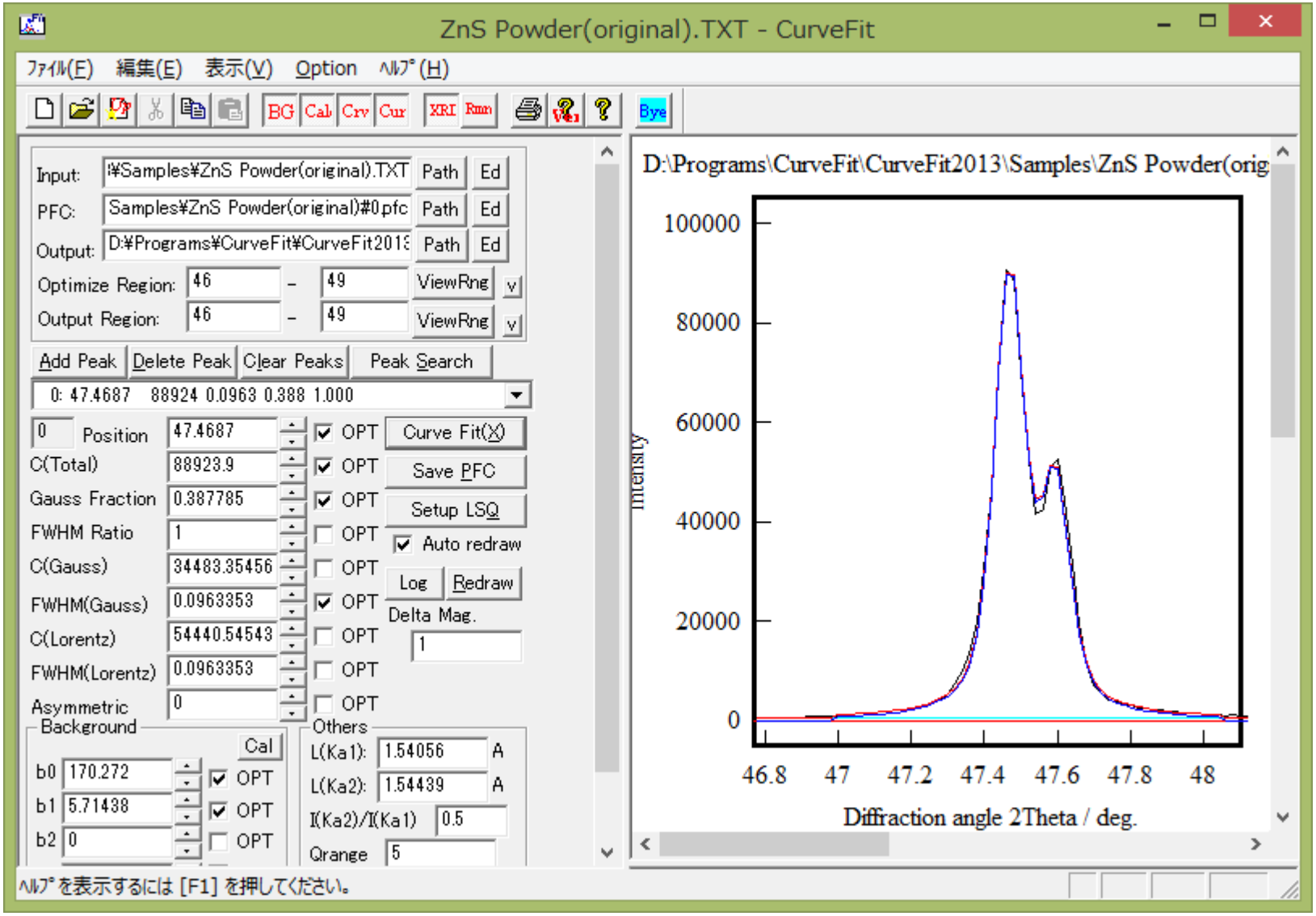
回帰問題: あるデータの組 (x_1, y_1), ・・・ (x_n, y_n) が理論式 $f(x) = f(x;a_k)$ に
　　従うことがわかっている。未知変数 $\{a_k\}$ を求めよ。
　　ただし、データ (y_i) には誤差 ε_i が含まれている: $y_i = f(x_i) + \varepsilon_i$
　最小二乗法: $S = \Sigma(f(x_i;a_k) - y_i)^2$ を最小化
　　(線形問題では) プログラムが非常に簡単
　・ε_i が正規分布 (標準偏差σ_i) している場合、
　　最尤推定法により最小二乗法が正当化される

$$S = \sum \left(\frac{f(x_i;a_k) - y_i}{\sigma_i} \right)^2$$ を最小化

線形最小二乗法: $f(x_i; a_k)$が $\{a_k\}$ に関する線形関数。
　　S は$\{a_k\}$ に関する2次多項式
　　ベクトル (a_k) に関する連立一次方程式を解けば (a_k) が求まる
非線形最小二乗法: 直接・精確に (a_k) を求めることはできない。
　　逐次繰り返し計算により、より正解に近い解を求める

多変数最適化問題例: 粉末XRDのピーク分離

$K\alpha_1$と$K\alpha_2$線が 強度 2:1 で出現することを考慮



非線形最適化の分類

目的関数$F(x)$ の最小値 (最大値) を求めるために

勾配法: 1次微分 (勾配) を使って減少方向 (最小化方向) を探す

- **Newton-Raphson法:**
  1次微分ベクトルおよび2次微分行列を使って効果的に最小値を探す
- **準Newton法:**
  2次微分行列を1次微分ベクトルから近似。
  直線探索法を使って大局的収束性を上げる。
- **再急降下 (Steepest Descent) 法:**
  1次微分ベクトルと直線探索法で最小値を探す
- **共役勾配 (Conjugate Gradient) 法:**
  探索方向を1次微分ベクトルの共役方向から探す。
- **Marquart法**
  $f_j(x_i)$への最小二乗法において、2次微分行列を$f_j(x_i)$の1次微分ベクトルから構成する

直接探索法:

- **単体 (Simplex) 法**
  決められた試行錯誤により変数空間を縮小していく。
  微分が必要ないのでプログラムが簡単になる

単変数Newton-Raphson法: 方程式の解

$F(x)$ を最小化するための条件

$$f(x) = dF(x)/dx = 0$$

$f(x) = 0$ を満たす x はわからないが、近似値 x_0 がわかっているとする

1. 真の解を $x_0 + \delta x$ とすると、$|\delta x| \ll 1$ が期待できるので、
   Taylor展開する

$$f(x_0 + \delta x) \sim f(x_0) + \delta x \frac{df(x_0)}{dx} = 0 \quad \Rightarrow \delta x \sim -(df(x_0)/dx)^{-1} f(x_0)$$

2. $x_1 = x_0 - (df(x_0)/dx)^{-1} f(x_0)$ は x_0 よりも真の解に近いと期待される
3. x_1 を x_0 に置き換えて同じ計算をし、より真に近い x_2 を得る
4. 1 ~ 3 を繰り返し **(逐次近似)**、
   反復誤差 $|x_i - x_{i-1}|$ が必要な精度以下になったら x_i を最終解とする

方程式の数値解法としてのNewton-Raphson法

関数の解: Newton-Raphson法 (Newton法)

$f(x) = 0$の解を求める

$$f(x_0+dx) = f(x_0) + dx\, f'(x_0) \sim 0$$
$$\Rightarrow \quad x_1 = x_0 + dx = x_0 - f(x_0) / f'(x_0)$$

計算では $f'(x_0)$ を差分計算で置き換えられる

割線法 (セカント法、はさみうち法):

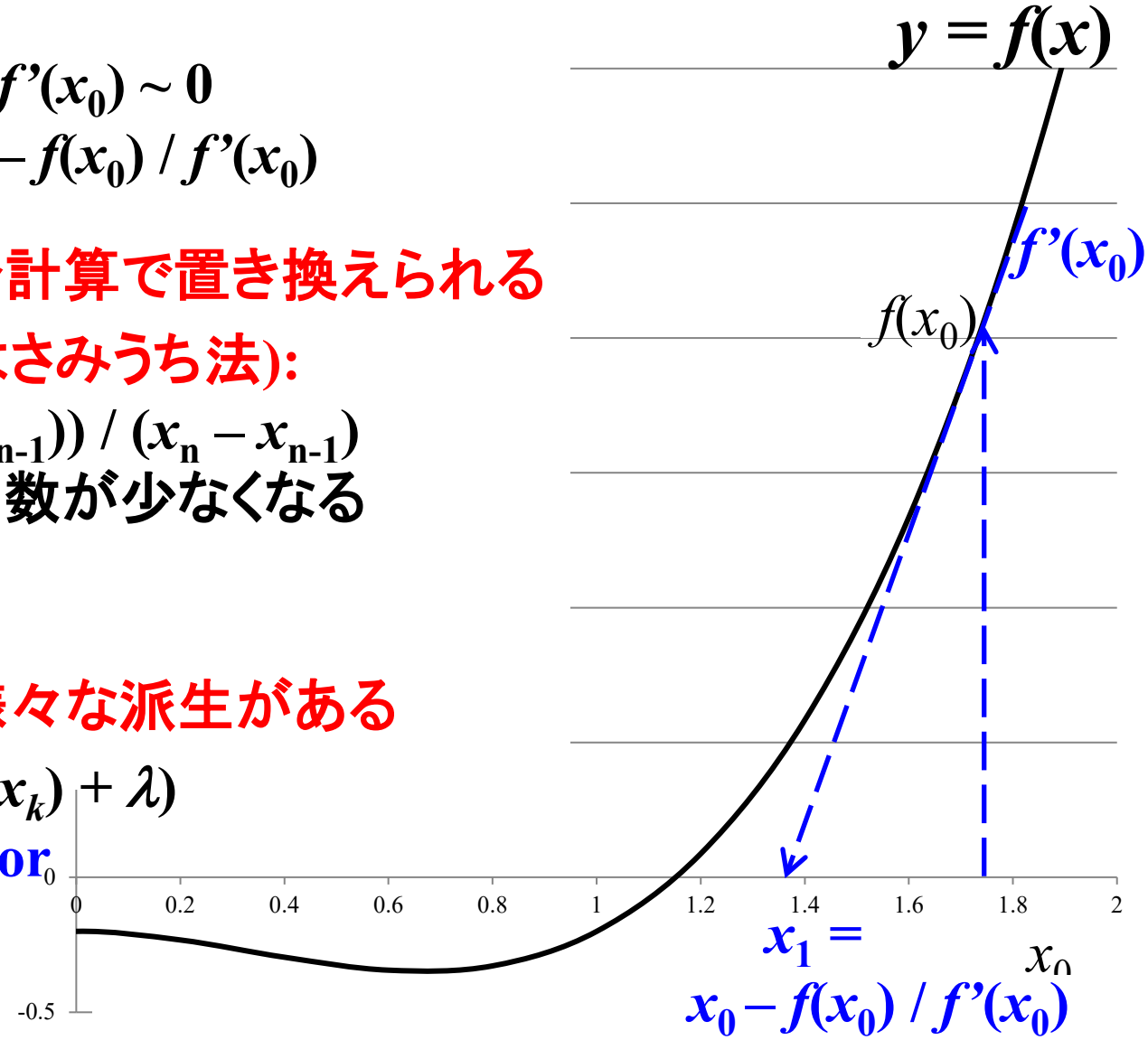
$$f'(x) = (f(x_n) - f(x_{n-1})) / (x_n - x_{n-1})$$

を使う。$f(x)$の計算回数が少なくなる

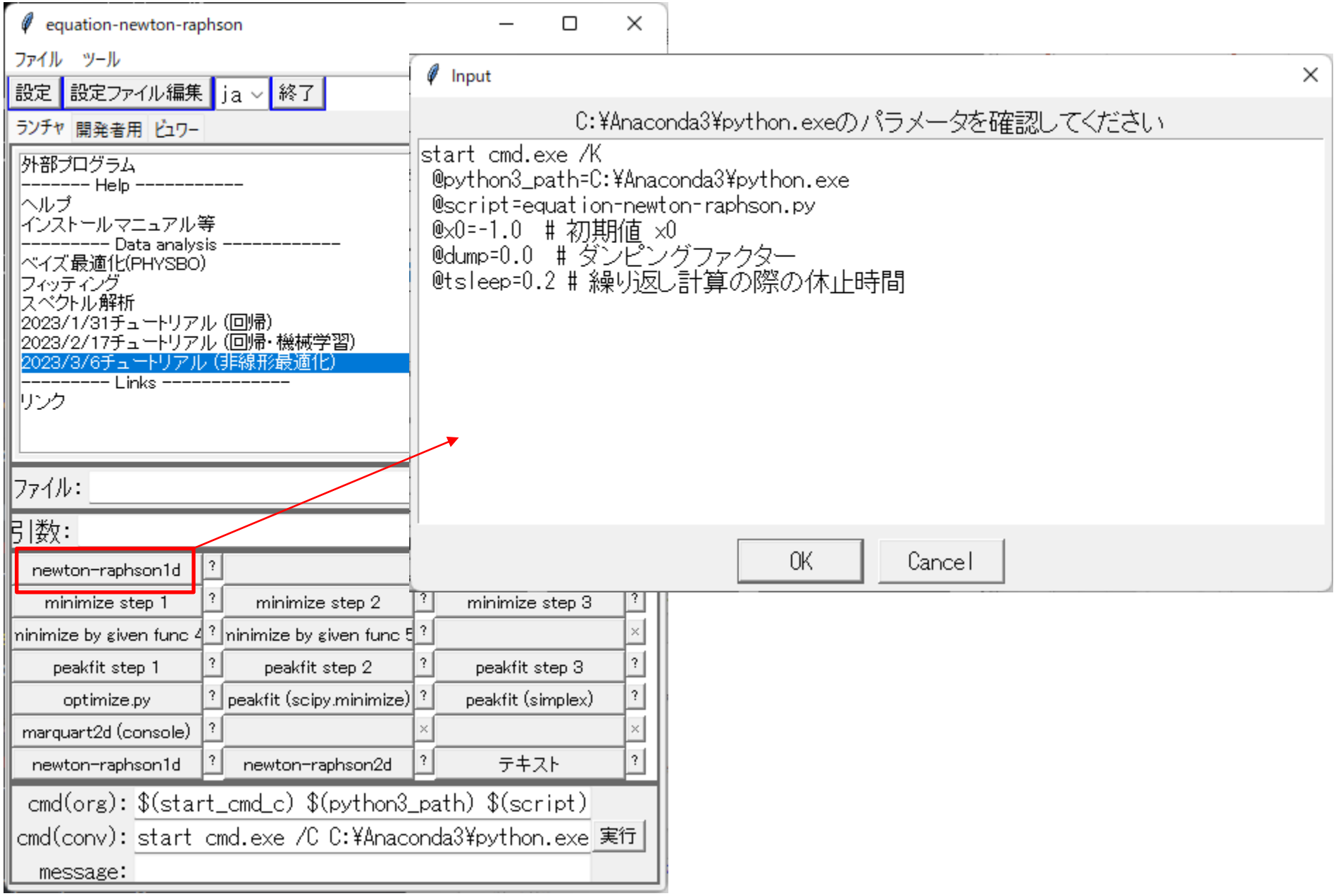
発散を抑える工夫で様々な派生がある

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

λ: Dumping Factor



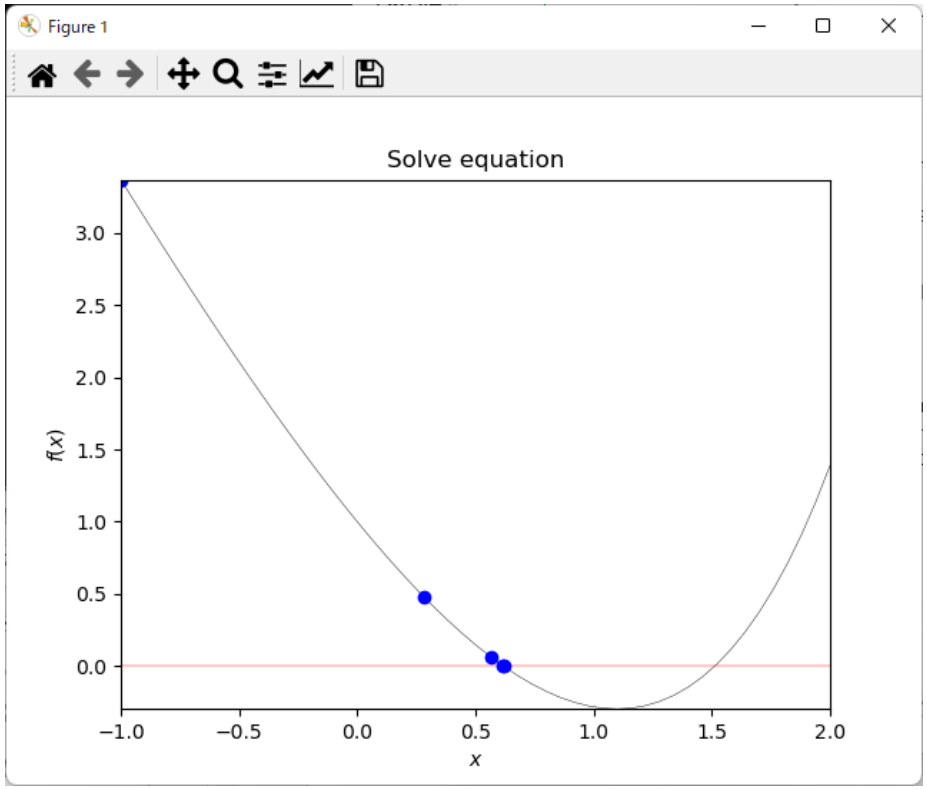
関数の解: Newton-Raphson法 (Newton法)



関数の解: Newton-Raphson法 (Newton法)

方程式:　exp(x) - 3.0 * x = 0
初期値: x0 = -1　ダンピングファクター 0

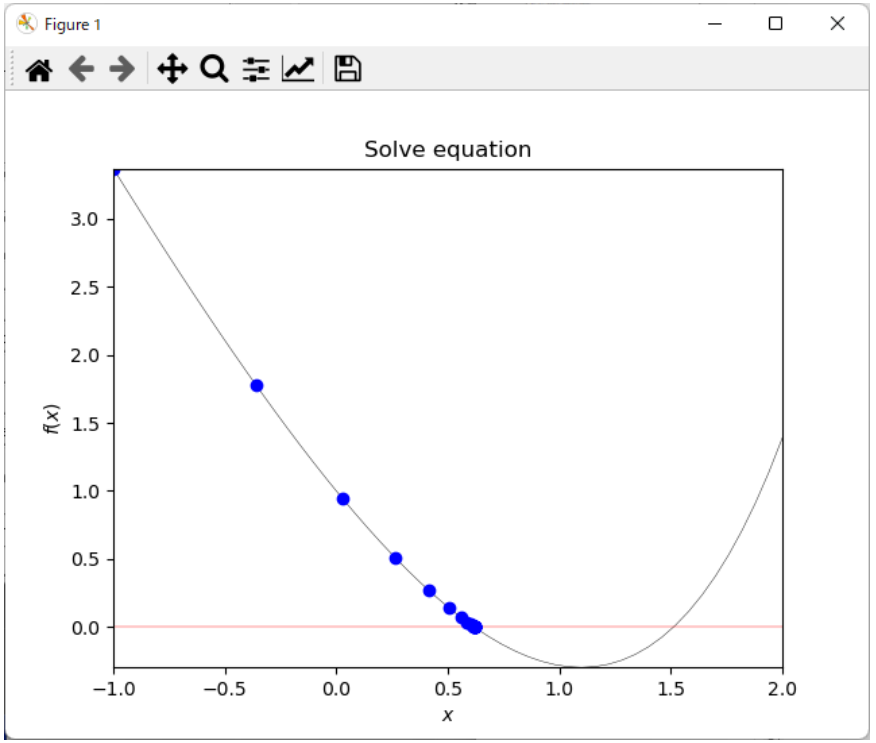
Solution of transcendental equation by Newton-Raphson method
x0 = -1.0
dumping factor = 0.0
Iter　0: x:　　-1 =>　0.2795, dx =　　1.28
Iter　1: x:　0.2795 =>　0.568, dx =　0.2885
Iter　2: x:　0.568 =>　0.6172, dx =　0.04916
Iter　3: x:　0.6172 =>　0.6191, dx =　0.001891
Iter　4: x:　0.6191 =>　0.6191, dx =　2.902e-06
Iter　5: x:　0.6191 =>　0.6191, dx =　6.844e-12
　Success: Convergence reached: dx =　6.844e-12 < eps =　　1e-10



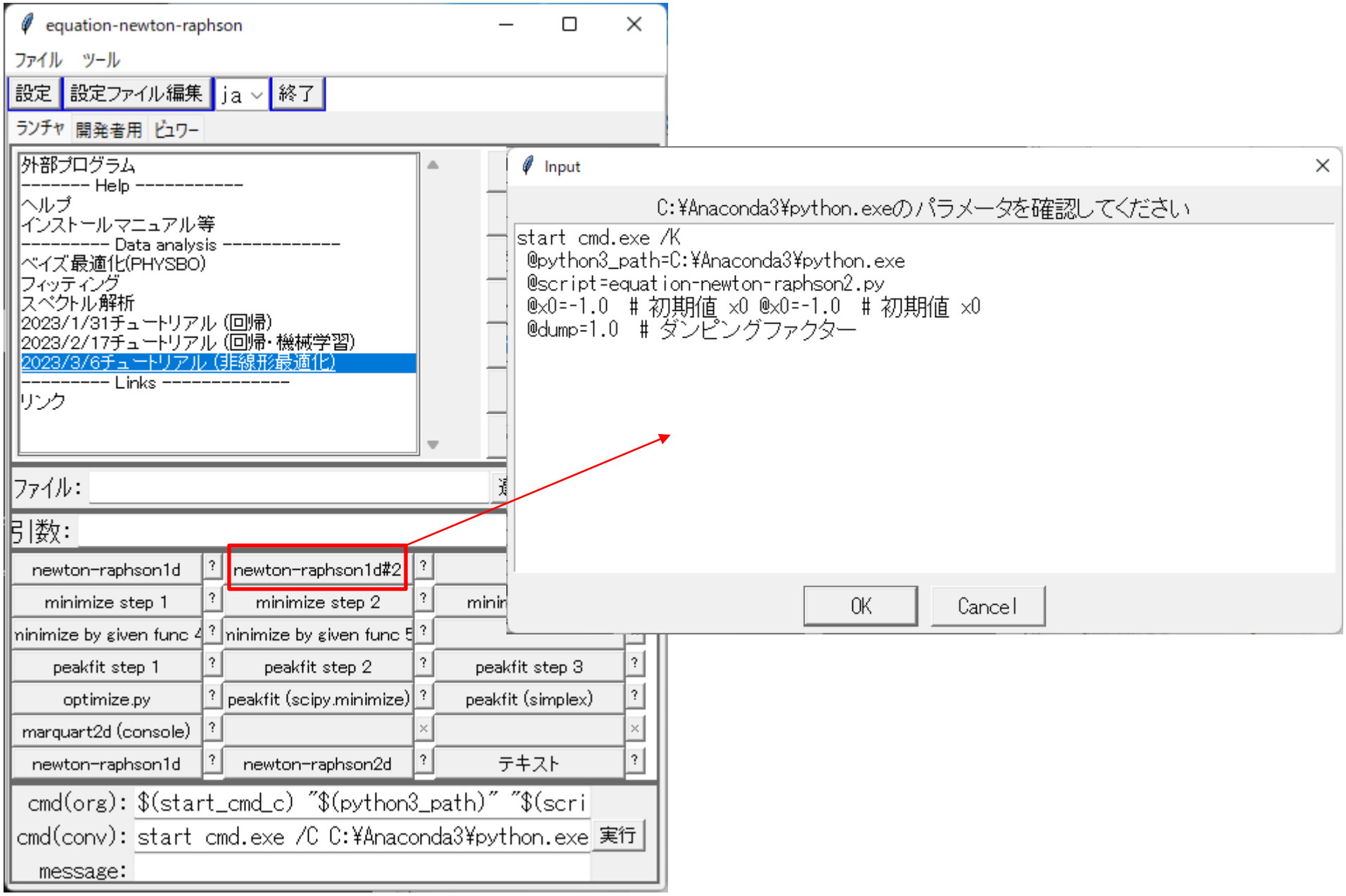
ダンピングファクターの効果

方程式:　exp(x) - 3.0 * x = 0
初期値: x0 = -1　ダンピングファクター 1.0

Solution of transcendental equation by Newton-Raphson method
x0 = -1.0
dumping factor = 1.0
Iter　0: x:　-1 =>　-0.3602, dx =　0.6398
Iter　1: x:　-0.3602 =>　0.02592, dx =　0.3862
Iter　2: x:　0.02592 =>　0.2662, dx =　0.2403
Iter　3: x:　0.2662 =>　0.4156, dx =　0.1494
Iter　4: x:　0.4156 =>　0.506, dx =　0.09042
…
Iter　31: x:　0.6191 =>　0.6191, dx =　9.904e-10
Iter　32: x:　0.6191 =>　0.6191, dx =　4.952e-10
Iter　33: x:　0.6191 =>　0.6191, dx =　2.476e-10
Iter　34: x:　0.6191 =>　0.6191, dx =　1.238e-10
Iter　35: x:　0.6191 =>　0.6191, dx =　6.19e-11
　Success: Convergence reached: dx =　6.19e-11 < eps =　1e-10



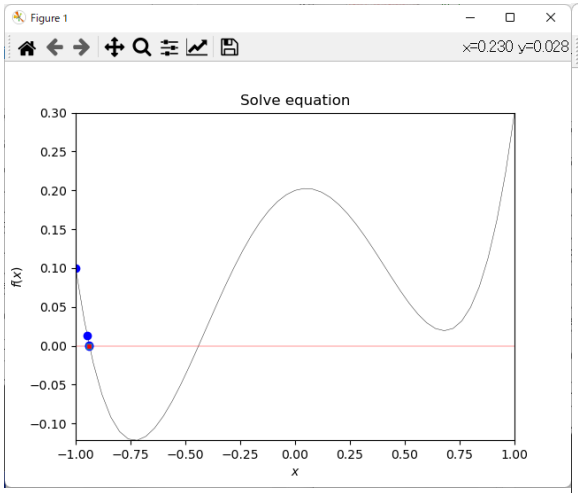
関数の解: Newton-Raphson法 (Newton法)



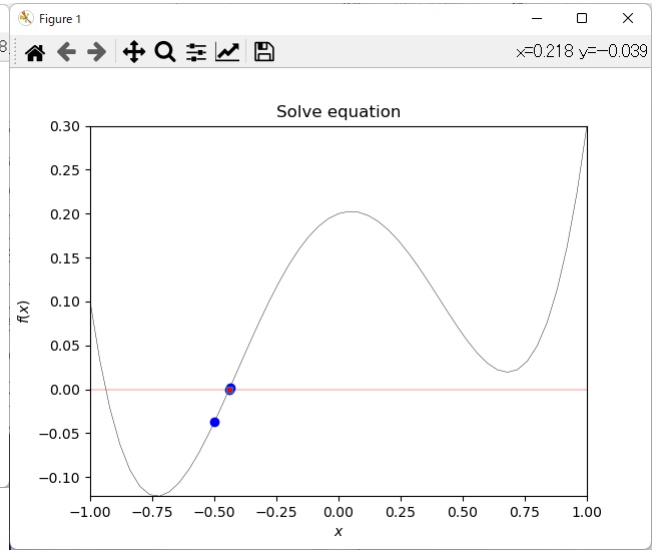
初期値とダンピングファクターの効果

方程式:　0.2 + 0.1 * x - 1 * x**2 + 0 * x**3 + 1 * x**4 ＝0
ダンピングファクター 0

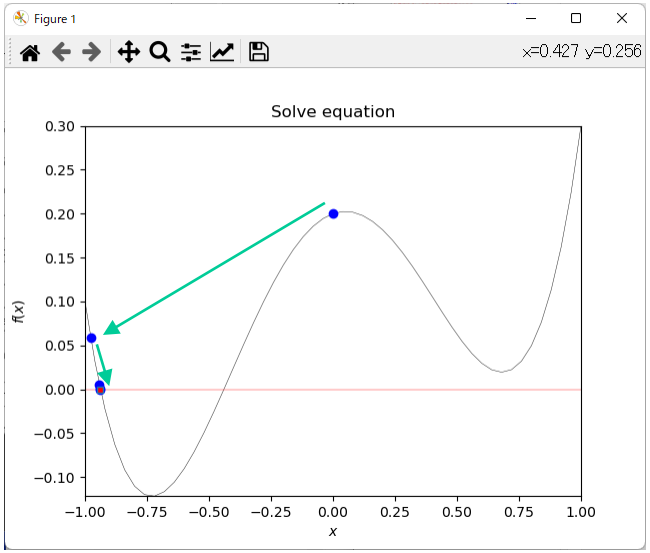
dump = 0.0　x0 = -1



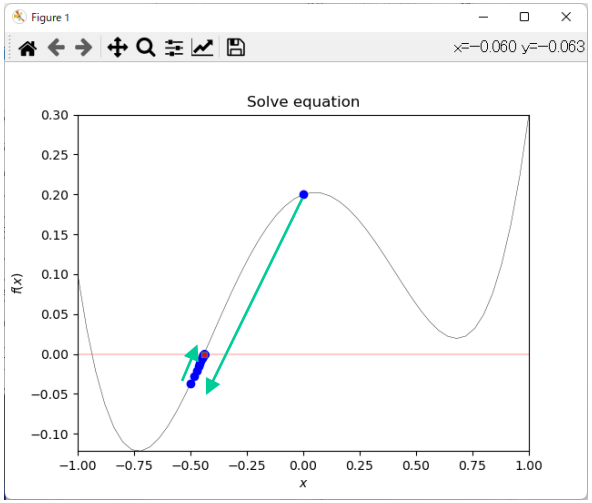
x0 = -0.5



x0 = 0.0



x0 = 0.0　dump = 3.0



pythonプログラム: 関数の解

scipy.optimize.fsolve() が使えるはずだが、よくわからいので自前のNewton-Raphson法のコードを書く

[tkProg]¥tkprog_tutorial¥optimize¥equation-newton-raphson.py

```
# 1階微分
def diff(x):
    return exp(x) - 3.0

# 関数 func(x) = 0 となる x を求める
def func(x):
    return exp(x) - 6.0 * x

def main():
    x = x0
    xt = []
    yt = []
    for i in range(nmaxiter):
# Dumped Newton-Raphson法: x(i+1) = x(i) – f(x(i)) / (f'(x(i)) + dump)
        f  = func(x)
        f1 = diff(x)
        f1 *= (1.0 + dump)
        xnext = x - f / f1
        dx = xnext - x
        if i % iprintiterval == 0:
            print("Iter {:5d}: x: {:>10.4g} => {:>10.4g}, dx = {:>10.4g}".format(i, x, xnext, dx))
#収束判定
        if abs(dx) < eps:
            print("  Success: Convergence reached: dx = {:10.4g} < eps = {:8.3g}".format(dx, eps))
            break

# xの値を更新
        x = xnext
```

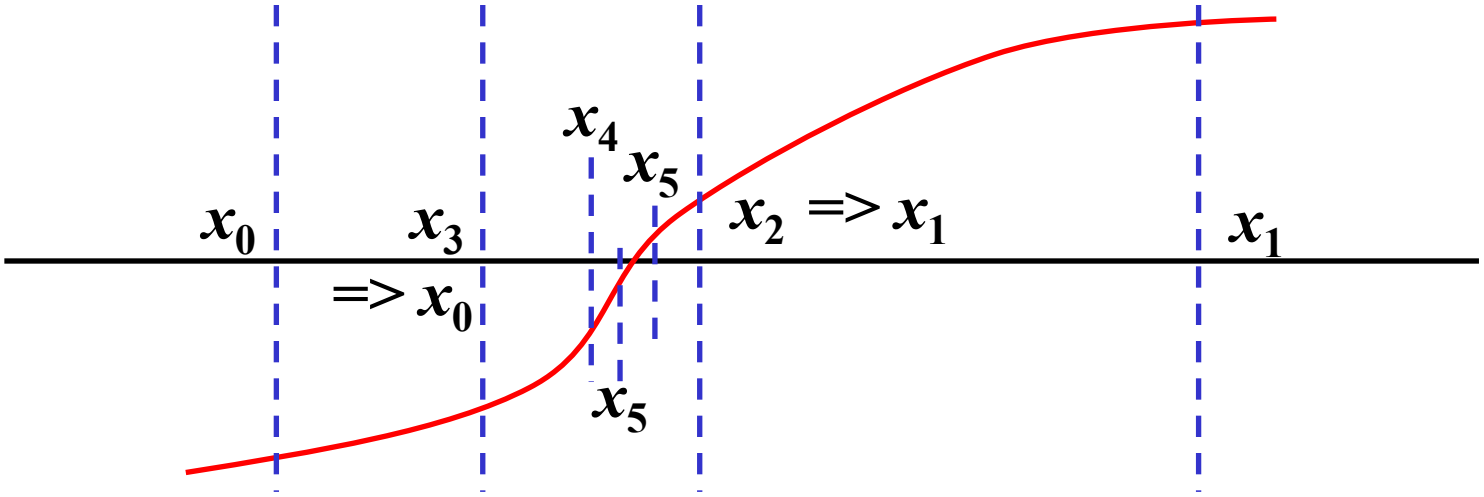
二分法 (bisection method): 単調関数

単調関数 $f(x) = 0$ の解の求めかた:

1. **$f(x_0) < 0$ & $f(x_1) > 0$ を満たす初期範囲 $[x_0, x_1]$ から始める**
   (あるいは $f(x_0) > 0$ & $f(x_1) < 0$)
   *** 解 x は必ず $[x_0, x_1]$ 内に存在する**
2. **以下の操作を繰り返す**

$f(x_0) < 0, f(x_1) > 0$ の場合 ($f(x_0) \cdot f(x_1) < 0$ を満たすかどうかで判断する)

1. $x_2 = (x_0 + x_1) / 2.0$
2. $f(x_2) > 0$ ($f(x_0) \cdot f(x_2) < 0$) である場合, x_1 を x_2 で置き換える
   $f(x_2) < 0$ ($f(x_1) \cdot f(x_2) < 0$) の場合, x_0 を x_2 で置き換える
3. $|x_1 - x_0|$ と $|f(x_1) - f(x_0)|$ が収束判定条件以内に収まったら、x_2 を解とする
4. 3.を満たさない場合、1 – 3 を繰り返す



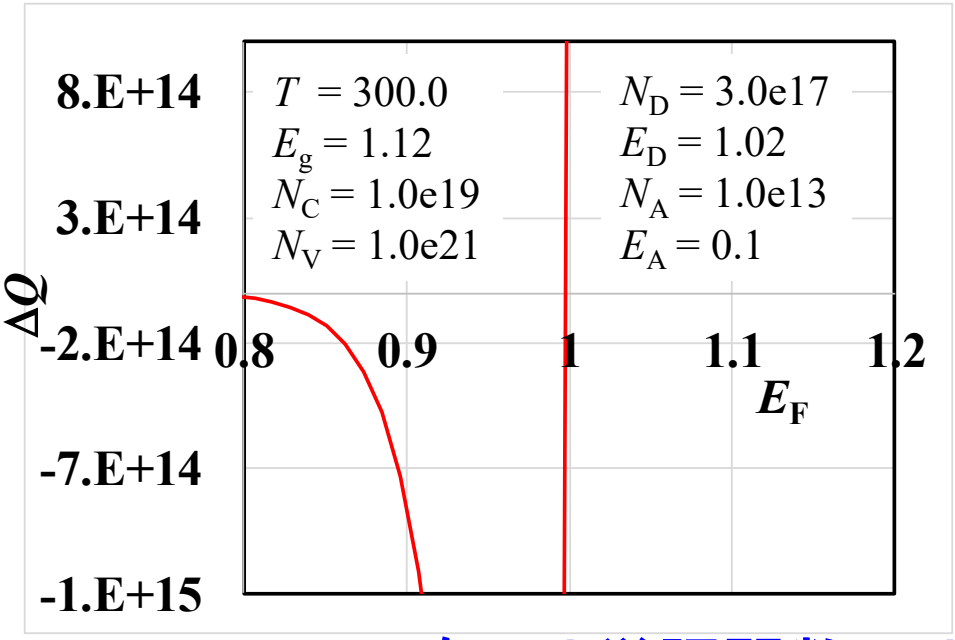
半導体の E_F の求め方

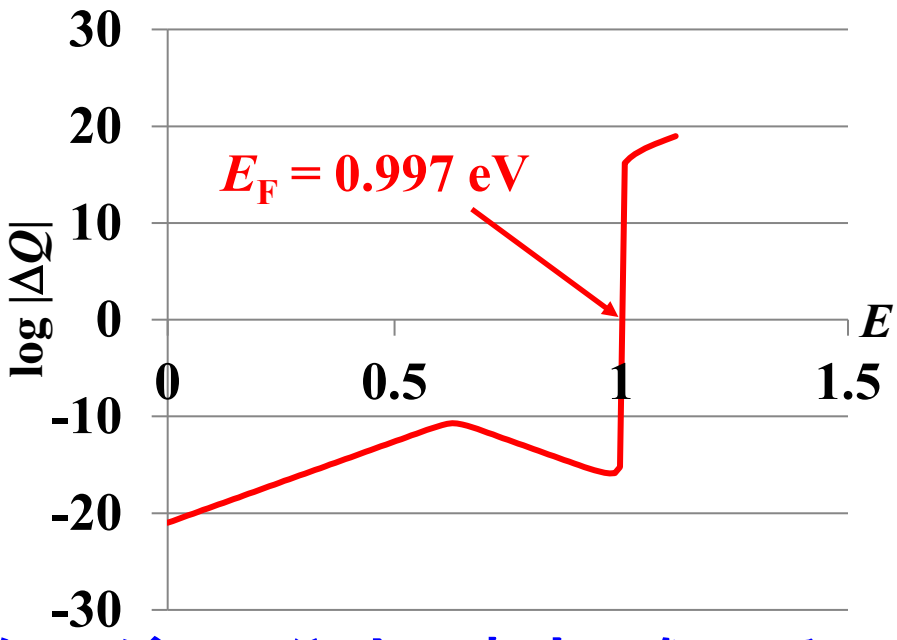
$$N_e = \int_{E_C}^{\infty} D_C(E) f_e(E, E_F) dE \qquad N_h = \int_{E_C}^{\infty} D_V(E) f_h(E, E_F) dE$$

$$N_D^+ = N_D\left[1 - f_e(E_D, E_F)\right] \qquad N_A^- = N_A\left[1 - f_h(E_A, E_F)\right]$$

$$f_h(E, E_F) = 1 - f_e(E, E_F)$$

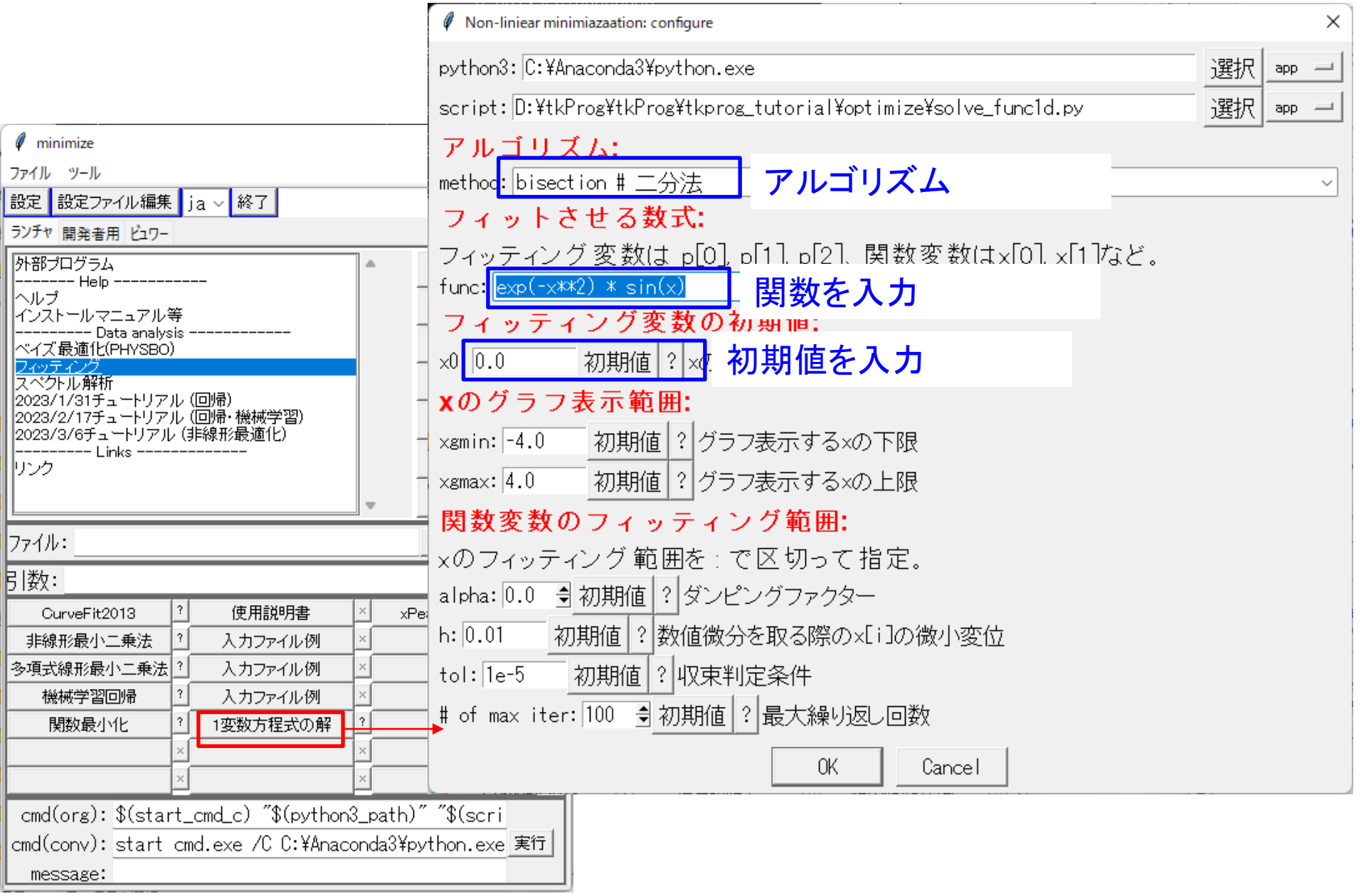
過剰電荷 $\Delta Q = (N_A^- + N_e) - (N_D^+ + N_h)$ を E_F に対してプロットし、
$\Delta Q = 0$ となる E_F を見つける



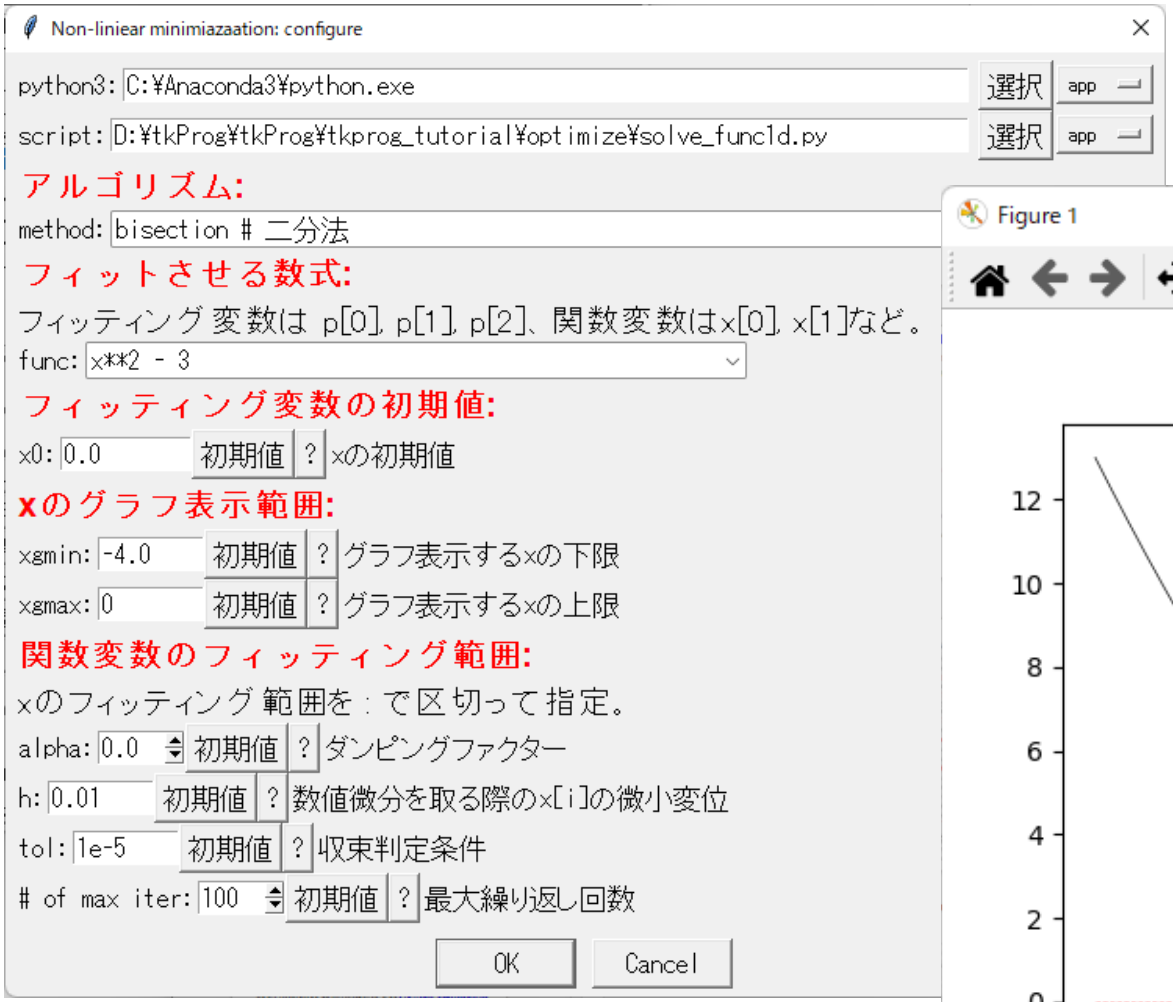


正確には単調関数ではないが、二分法で安定に解ける

1変数関数の解


Non-liniear minimiazaation: configure
python3: C:¥Anaconda3¥python.exe
選択
app
script: D:¥tkProg¥tkProg¥tkprog_tutorial¥optimize¥solve_func1d.py
選択
app
アルゴリズム:
method: bisection # 二分法
アルゴリズム
フィットさせる数式:
フィッティング変数は p[0], p[1], p[2]、関数変数はx[0], x[1]など。
func: exp(-x**2) * sin(x)
関数を入力
フィッティング変数の初期値:
x0 0.0 初期値 ?
初期値を入力
xのグラフ表示範囲:
xgmin: -4.0 初期値 ? グラフ表示するxの下限
xgmax: 4.0 初期値 ? グラフ表示するxの上限
関数変数のフィッティング範囲:
xのフィッティング範囲を：で区切って指定。
alpha: 0.0 初期値 ? ダンピングファクター
h: 0.01 初期値 ? 数値微分を取る際のx[i]の微小変位
tol: 1e-5 初期値 ? 収束判定条件
# of max iter: 100 初期値 ? 最大繰り返し回数
OK
Cancel
minimize
ファイル ツール
設定
設定ファイル編集
ja
終了
ランチャ
開発者用
ビュワー
外部プログラム
-------- Help ----------
ヘルプ
インストールマニュアル等
---------- Data analysis ----------
ベイズ最適化(PHYSBO)
フィッティング
スペクトル解析
2023/1/31チュートリアル (回帰)
2023/2/17チュートリアル (回帰・機械学習)
2023/3/6チュートリアル (非線形最適化)
---------- Links ----------
リンク
ファイル:
引数:
CurveFit2013
使用説明書
非線形最小二乗法
入力ファイル例
多項式線形最小二乗法
入力ファイル例
機械学習回帰
入力ファイル例
関数最小化
1変数方程式の解
cmd(org): $(start_cmd_c) "$(python3_path)" "$(scri
cmd(conv): start cmd.exe /C C:¥Anaconda3¥python.exe
実行
message:

1変数関数の解: 2分法


Non-liniear minimiazaation: configure
python3: C:¥Anaconda3¥python.exe
選択
app
script: D:¥tkProg¥tkProg¥tkprog_tutorial¥optimize¥solve_func1d.py
選択
app
アルゴリズム:
method: bisection # 二分法
フィットさせる数式:
フィッティング変数は p[0], p[1], p[2]、関数変数はx[0], x[1]など。
func: x**2 - 3
フィッティング変数の初期値:
x0: 0.0
初期値
?
xの初期値
xのグラフ表示範囲:
xgmin: -4.0
初期値
?
グラフ表示するxの下限
xgmax: 0
初期値
?
グラフ表示するxの上限
関数変数のフィッティング範囲:
xのフィッティング範囲を : で区切って指定。
alpha: 0.0
初期値
?
ダンピングファクター
h: 0.01
初期値
?
数値微分を取る際のx[i]の微小変位
tol: 1e-5
初期値
?
収束判定条件
# of max iter: 100
初期値
?
最大繰り返し回数
OK
Cancel

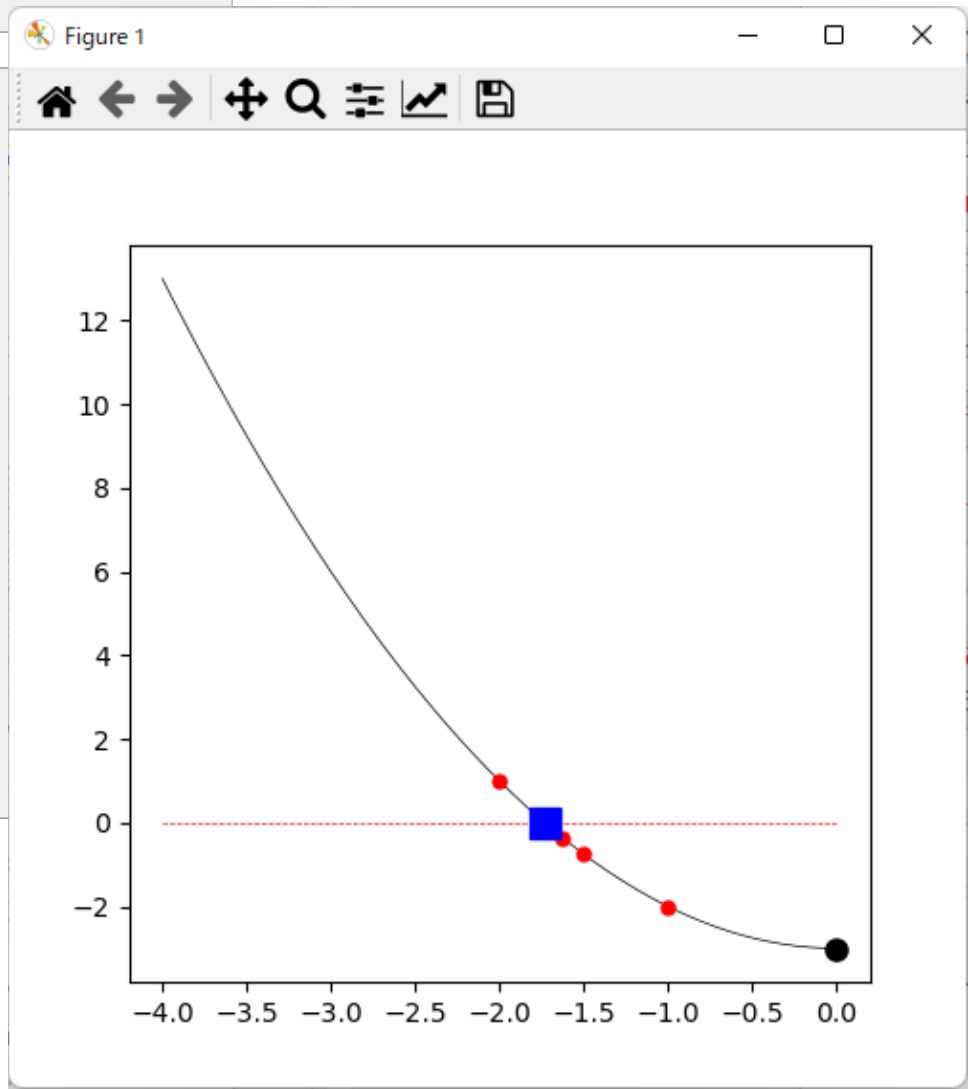


Figure 1
12
10
8
6
4
2
0
−2
−4.0
−3.5
−3.0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0

1変数関数の解: Newton法

Non-liniear minimiazaation: configure

python3: C:¥Anaconda3¥python.exe 選択 app

script: D:¥tkProg¥tkProg¥tkprog_tutorial¥optimize¥solve_func1d.py 選択 app

アルゴリズム:

method: newton # Newton-Raphson 法

フィットさせる数式:

フィッティング変数は p[0], p[1], p[2]、関数変数はx[0], x[1]など。

func: x**2 - 3

フィッティング変数の初期値:

x0: -0.5 初期値 ? xの初期値

xのグラフ表示範囲:

xgmin: -4.0 初期値 ? グラフ表示するxの下限

xgmax: -1 初期値 ? グラフ表示するxの上限

関数変数のフィッティング範囲:

xのフィッティング範囲を : で区切って指定。

alpha: 0.0 初期値 ? ダンピングファクター

h: 0.01 初期値 ? 数値微分を取る際のx[i]の微小変位

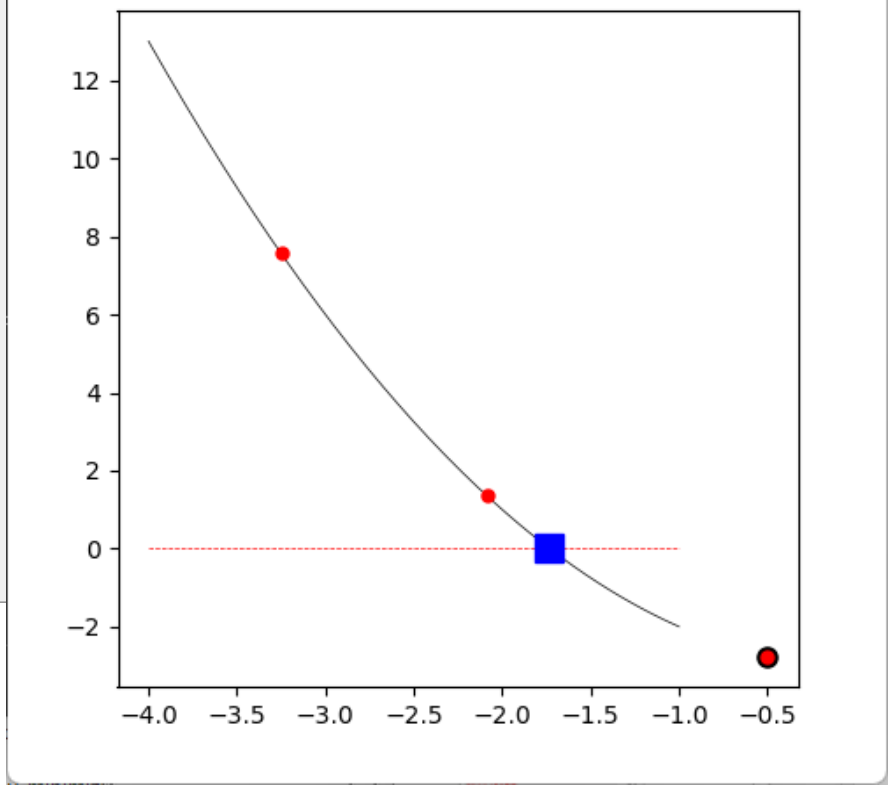
tol: 1e-5 初期値 ? 収束判定条件

# of max iter: 100 初期値 ? 最大繰り返し回数

OK Cancel

Figure 1

x=−2.710 y=−3.13



Newton-Raphson法: 単変数関数の最適化

$F(x)$ を最小化するための条件

$$f(x) = dF(x)/dx = 0$$

$f(x) = 0$ を満たす x をNewton-Raphson法で解く

$$f(x_i + \delta x) \sim f(x_i) + \delta x \frac{df(x_i)}{dx} = 0$$

$$\Rightarrow \delta x_{i+t} \sim -(df(x_i)/dx)^{-1} f(x_i)$$

を繰り返す

=> 目的関数 F(x) に書き直す

$$\delta x_{i+t} \sim -(d^2F(x_i)/dx^2)^{-1} (dF(x_i)/dx)$$

最適化問題の数値解法としてのNewton-Raphson法

目的関数の2階微分の逆数 $(d^2F(x_i)/dx^2)^{-1}$ が必要

Newton-Raphson法: 多変数関数の最適化

多変数への拡張: 最小化関数 $F(x_l)$ の最小値を求める

$$f_k(x_l) = \partial F(x_l)/\partial x_k = 0$$

繰り返し計算: $f_k(x_l + \delta x_l) \sim f_k(x_l) + \sum_{k'} \delta x_{k'} \partial f_k(x_l)/\partial x_{k'} = 0$

$$x_{l,1} = x_{l,0} - (\partial f_k(x_l)/\partial x_{k'})^{-1}(f_k) = x_{l,0} - (F''_{kk'})^{-1}(F'_k)$$

$$F''_{kk'} = \frac{\partial^2 F(\mathbf{x})}{\partial x_k \partial x_{k'}}$$

Hessian (ヘッセ) 行列

(ヘッセ行列の固有値をヘッシアンと呼ぶ)

Hessian行列は正定値であるとは限らない (極大値、鞍点)

＝＞ F''が降下方向を与えるとは限らない

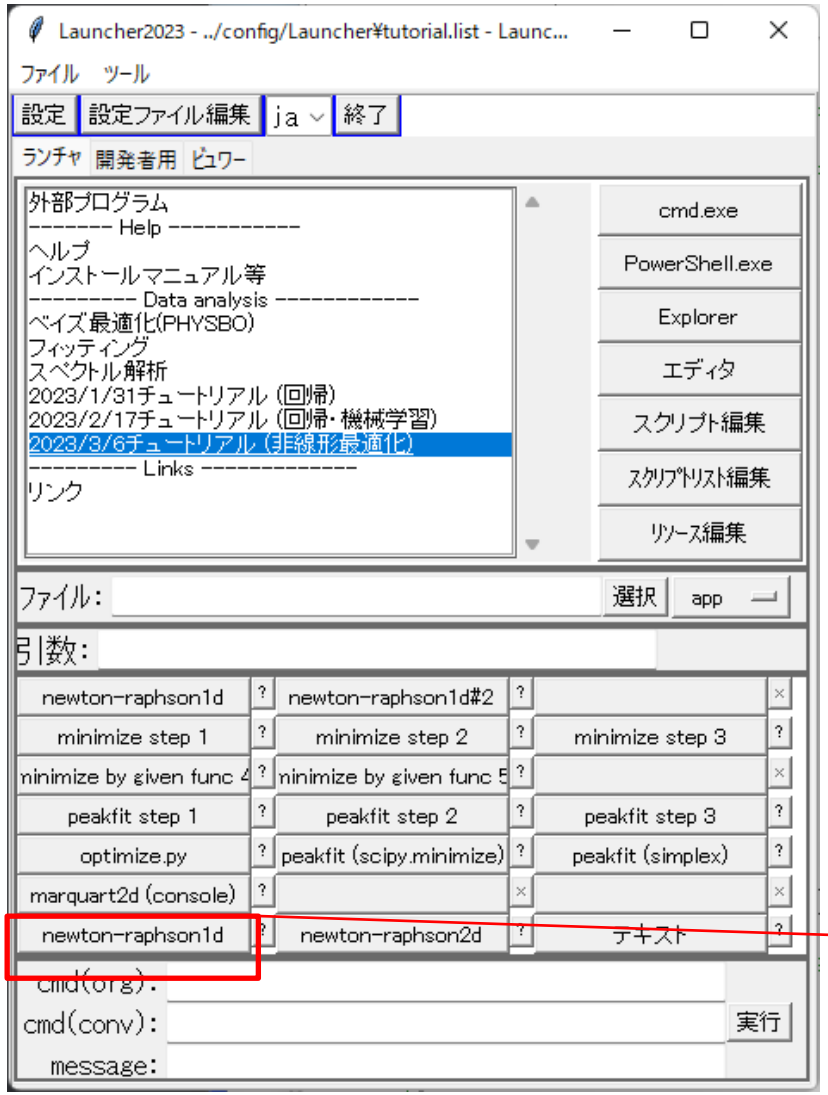
F''を正定値行列で置き換え、発散を抑える

$$x_{l,1} = x_{l,0} - (F''_{kk'} + \lambda I)^{-1}(F'_k)$$

λ: Dumping Factor

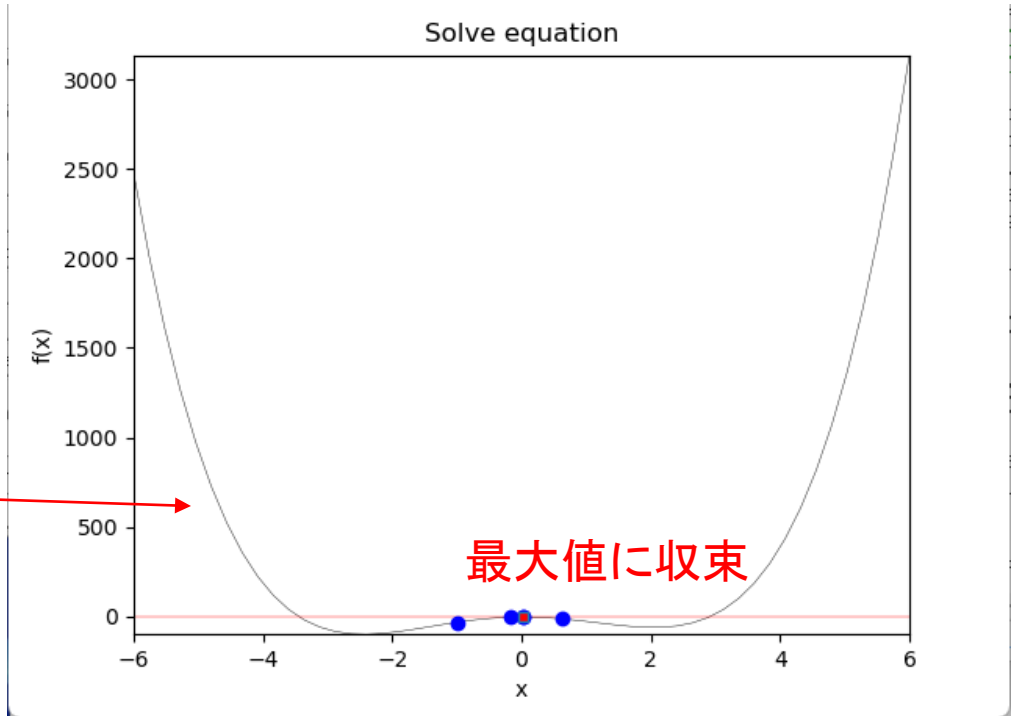
Newton-Raphson法による最小化？

最小化: -3.0 + x - 30.0 * x*x + 1.5 * x*x*x + 3.0 * x*x*x*x
初期値 -1.0



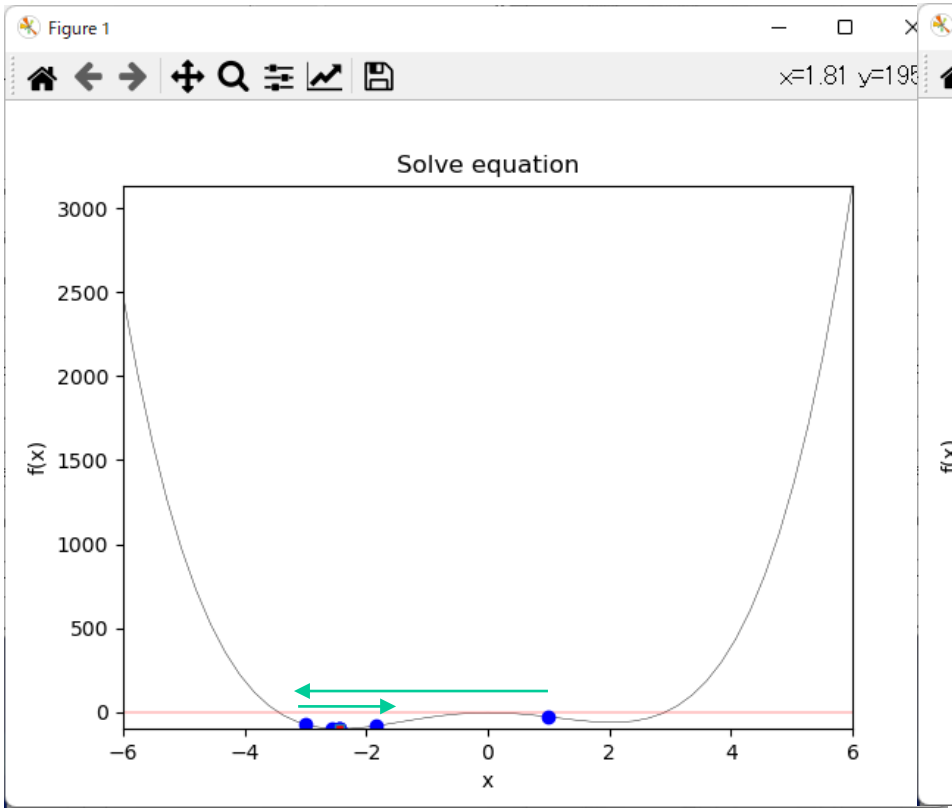
Find minimum / maximum point by Newton-Raphson method

```
Iter    0: x:       -1 =>      0.6212, dx =     1.621   f=      -32.5
Iter    1: x:   0.6212 =>     -0.1602, dx =   -0.7814   f=     -13.15
Iter    2: x:  -0.1602 =>     0.01625, dx =    0.1764   f=     -3.934
Iter    3: x:  0.01625 =>     0.01669, dx = 0.0004443   f=     -2.992
Iter    4: x:  0.01669 =>     0.01669, dx =  2.125e-08   f=     -2.992
Iter    5: x:  0.01669 =>     0.01669, dx =  2.126e-13   f=     -2.992
  Success: Convergence reached: dx =   2.126e-13 < eps =    1e-10
```

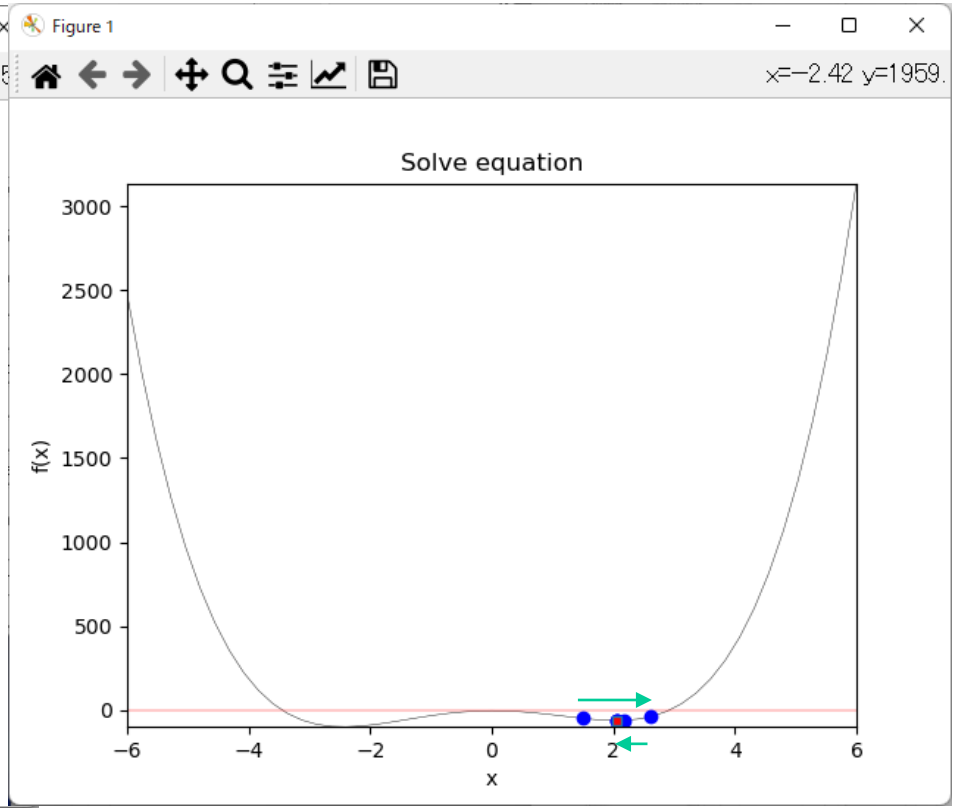


初期値の効果

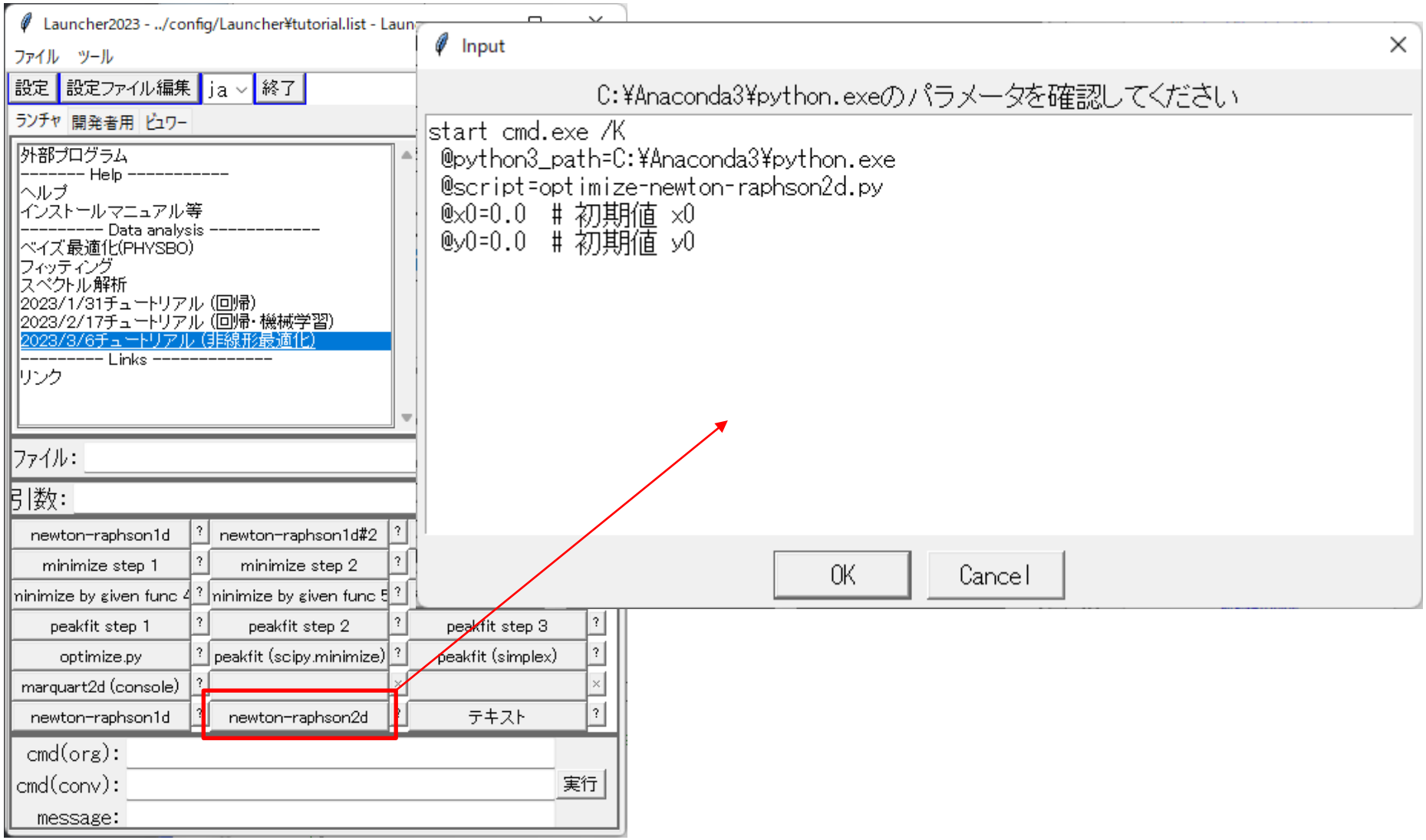
初期値 x0=1.0



初期値 x0=1.5

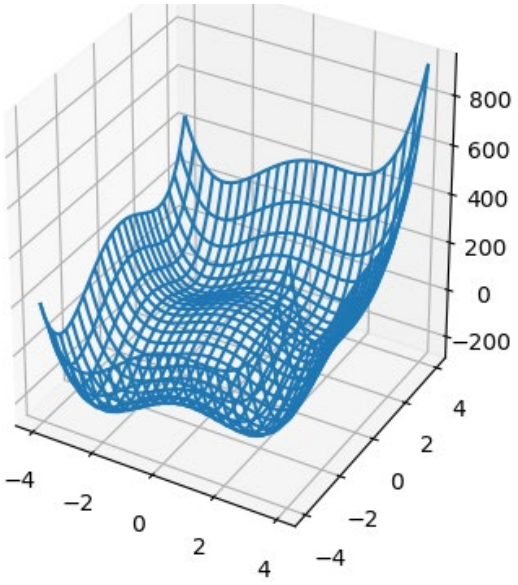


2変数Newton-Raphson法

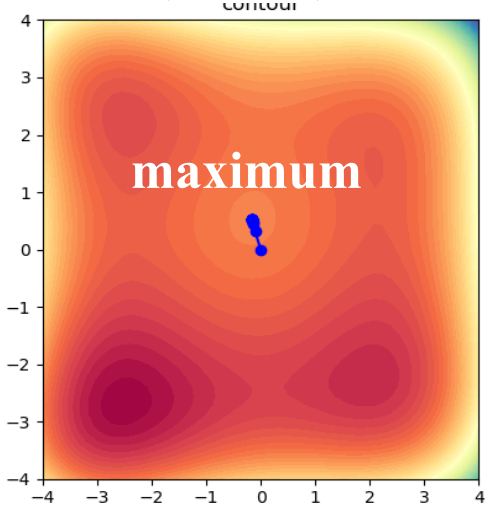


2変数Newton-Raphson法

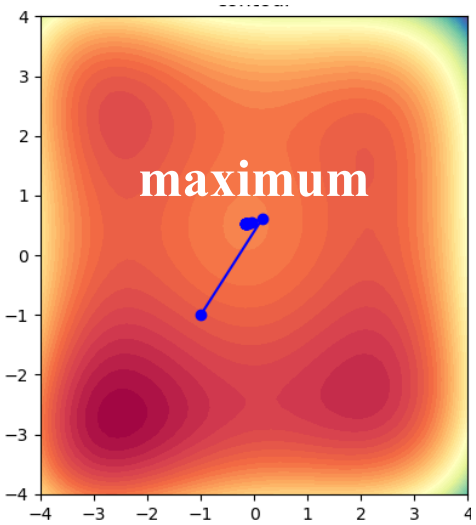
$F(x,y) = -3.0 - 10x - 30x^2 + 1.5x^3 + 3x^4 + 30y - 30y^2 + 3y^4 + 3xy^2$



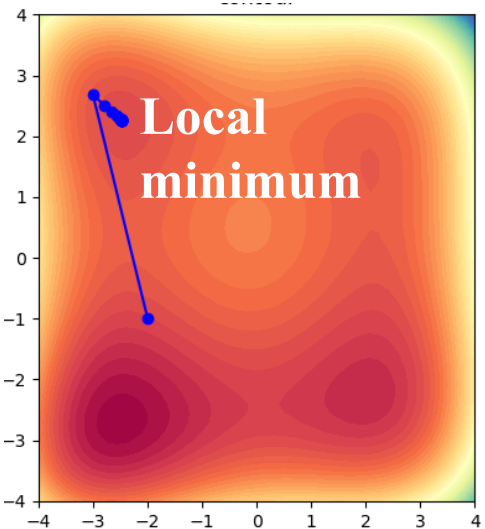
From (0.0 0.0) Newton



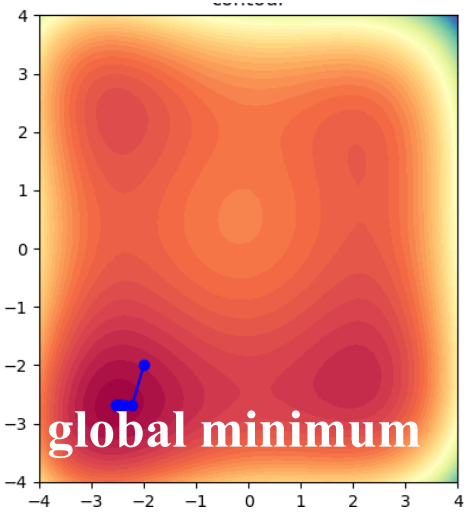
From (-1.0 -1.0)



From (-2.0 -1.0)



From (-2.0 -2.0)



準Newton法

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

最小化関数 $F(x_l)$

繰り返し計算: $x_l{}^{(i+1)} = x_l{}^{(i)} - \left(\partial^2 F/\partial x_k \partial x_{k\prime}\right)^{-1}(\partial F/\partial x_k)$

$F''_{kk\prime} = \partial^2 F/\partial x_k \partial x_{k\prime}$: **Hessian (ヘッセ) 行列**

Newton法の問題:

(1) Hessian行列は二次行列のため、計算に時間がかかる

(2) Hessian行列の固有値は負になることもある => 極大値を探索

(3) 発散しやすい

準Newton法:

(1,2) Hessian行列の計算を過去の一次微分の値を使って近似

(3) 探索方向 $-(\partial^2 F/\partial x_k \partial x_{k\prime})^{-1}(\partial F/\partial x_k)$ に沿って線形探索を行う

Davidon-Fletcher-Powell (DFP) 法

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

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

探索方向 d は $B^{(k)}d = -\nabla F(x_l{}^{(k)})$ で決定

DFP法: 準Newton法の最初の定式化

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

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

$$- \frac{s^{(k)T}\cdot\left(y^{(k)} - B^{(k)}s^{(k)}\right)}{\left(s^{(k)T}\cdot y^{(k)}\right)^2} y^{(k)} \cdot y^{(k)T}$$

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

Broyden-Fletcher-Goldfarb-Shanno (BFGS) 法

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

BFGS法：準Newton法でも最も有効と認められている

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

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

アルゴリズム：

STEP 0: 初期値 $x^{(0)}$、初期行列 $B^{(0)}$ (通常は単位行列) を与える。

STEP 1: $B^{(k)}d = -\nabla F\left(x_l{}^{(k)}\right)$ から探索方向 $d^{(k)}$ を求める

STEP 2: 直線探索によって、$d^{(k)}$方向のステップ幅 $\alpha^{(k)}$ を決める

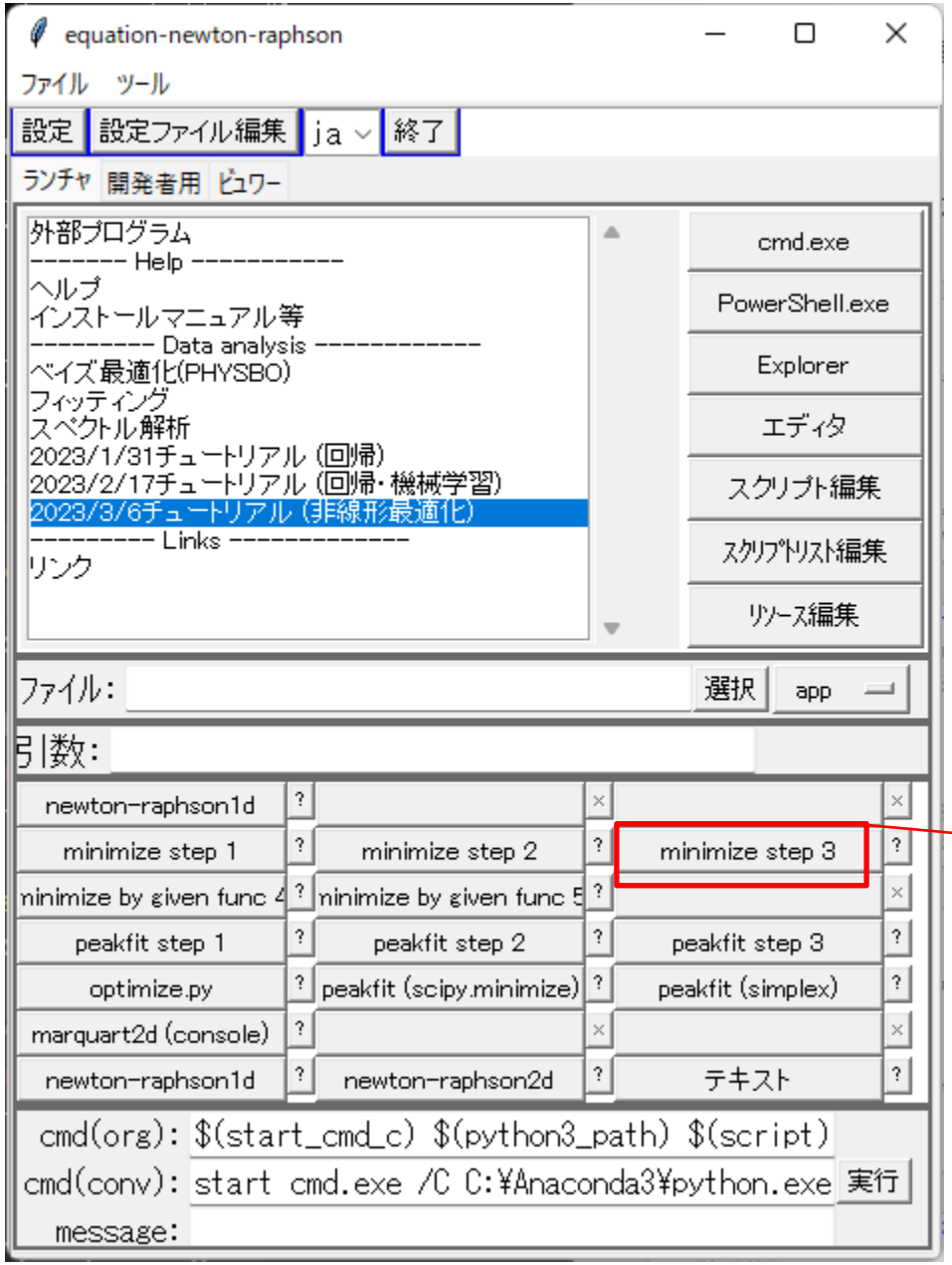
STEP 3: $x^{(k+1)} = x^{(k)} + \alpha^{(k)}d^{(k)}$ とする

STEP 4: 収束したら計算終了。

収束条件が満たされていなければSTEP 5へ

STEP 5: $s^{(k)}, y^{(k)}$を計算し、$B^{(k+1)}$ を求め、STEP 1へ

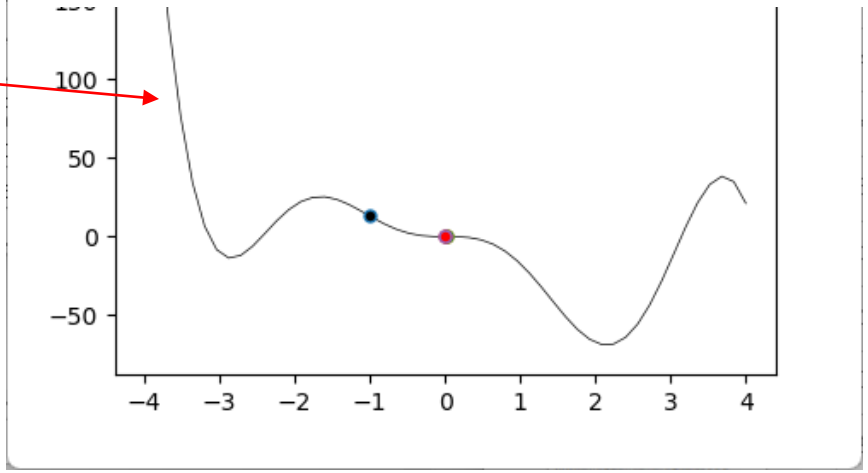
最適化: 準Newton法 (BFGS法)



方程式: (x[0] - 20.0 * x[0]**2 - 3.5 * x[0]**3 + 2.0 * x[0]**4) * sin(x[0]) = 0
初期値: x0 = -1

initial values: [-1.0]
tol : 1e-05
maxiter : 100

Minimize:
callback 0: xk=[0.01] func=7.996386725333053e-05
callback 1: xk=[0.00778691] func=5.1179322352247074e-05
callback 2: xk=[0.00192424] func=3.560146791253761e-06
callback 3: xk=[0.00077673] func=5.939336549108224e-07
Warning: Desired error not necessarily achieved due to precision loss.
Current function value: 0.000001
Iterations: 4
Function evaluations: 57
Gradient evaluations: 45



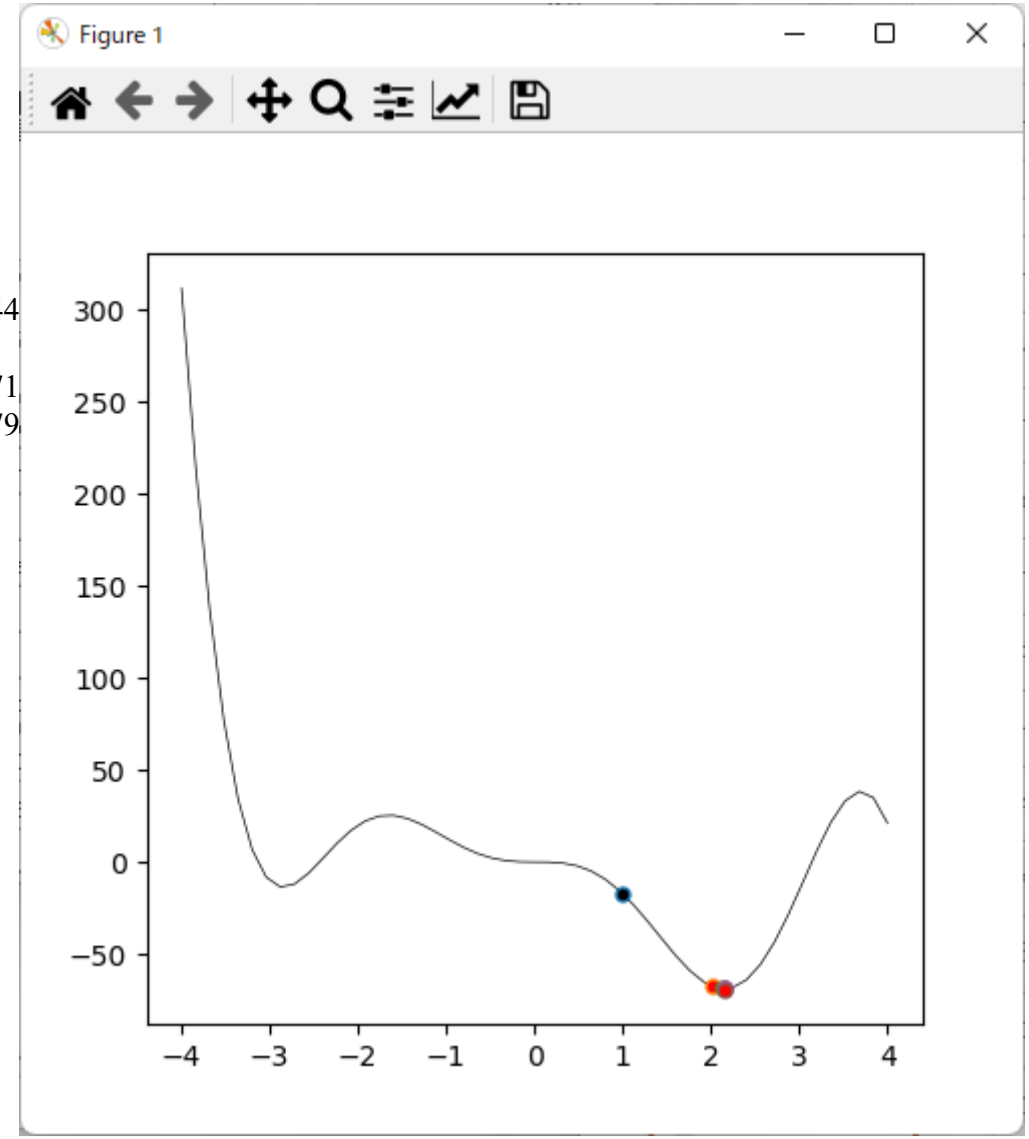
初期値の効果

方程式: (x[0] - 20.0 * x[0]**2 - 3.5 * x[0]**3 + 2.0 * x[0]**4) * sin(x[0]) = 0

初期値: x0 = 1

method : bfgs
initial values: [1.0]
tol : 1e-05
maxiter : 100

Minimize:
callback 0: xk=[2.01] func=-67.49195841203205
callback 1: xk=[2.13738861] func=-68.90437158642544
callback 2: xk=[2.15188914] func=-68.918031615511
callback 3: xk=[2.15093374] func=-68.91809739205871
callback 4: xk=[2.15093886] func=-68.91809740396579
Optimization terminated successfully.
Current function value: -68.918097
Iterations: 5
Function evaluations: 7
Gradient evaluations: 7



再急降下法 (Steepest decent method: SD)

1次微分だけから極小値を探す

考え方: 最小値は $-(\partial F(x_i)/\partial x_i)$ の方位にある

$$x_i^{(k+1)} = x_i^{(k)} - \alpha \partial F(x_i^{(k)})/\partial x_i$$

$F(x_i^{(k)})$ の最小値を見つけるには
適切な α を選択する必要がある

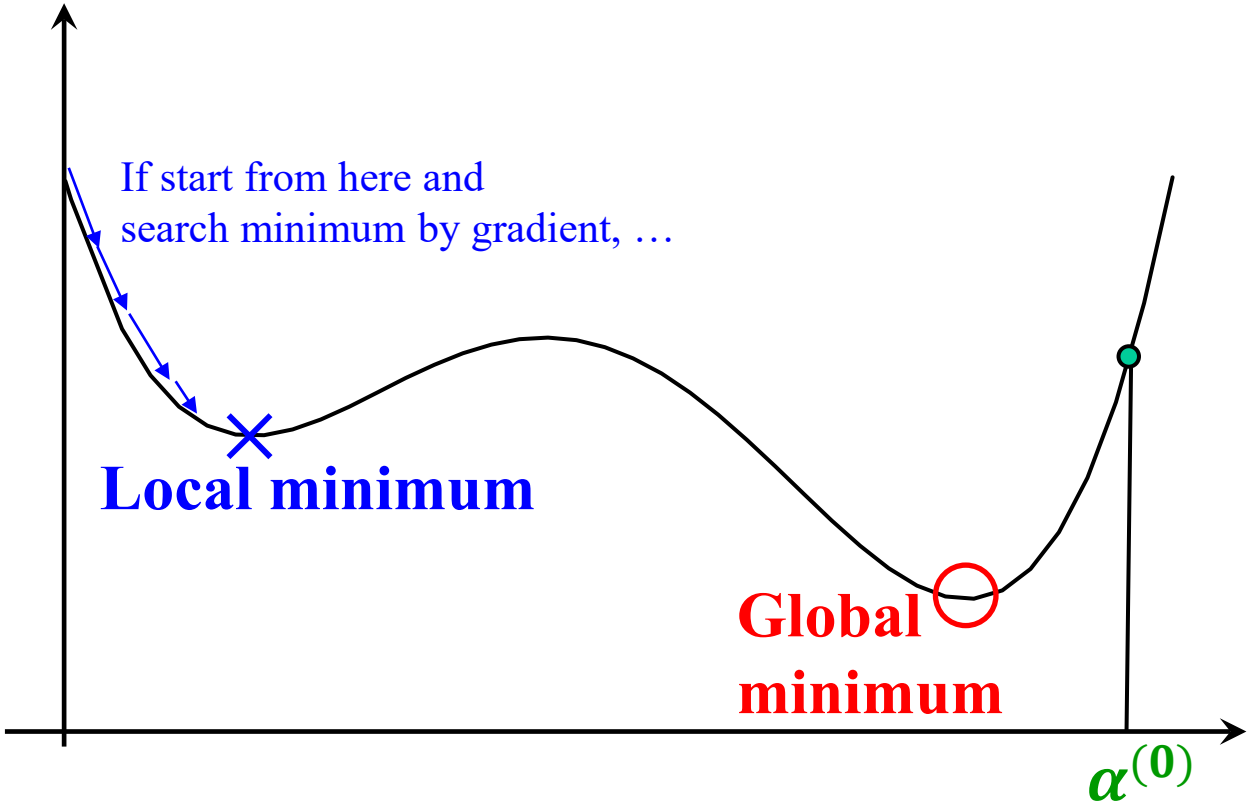
α の選び方の選択肢:

(i) 単純な方法: 小さい α を選ぶ

(ii) 直接探索: 直線探索法

Armijo / Wolfe 条件

Global minimum (大域的最小値) vs local minimum (極小値)



How to avoid to be trapped by local minimum:

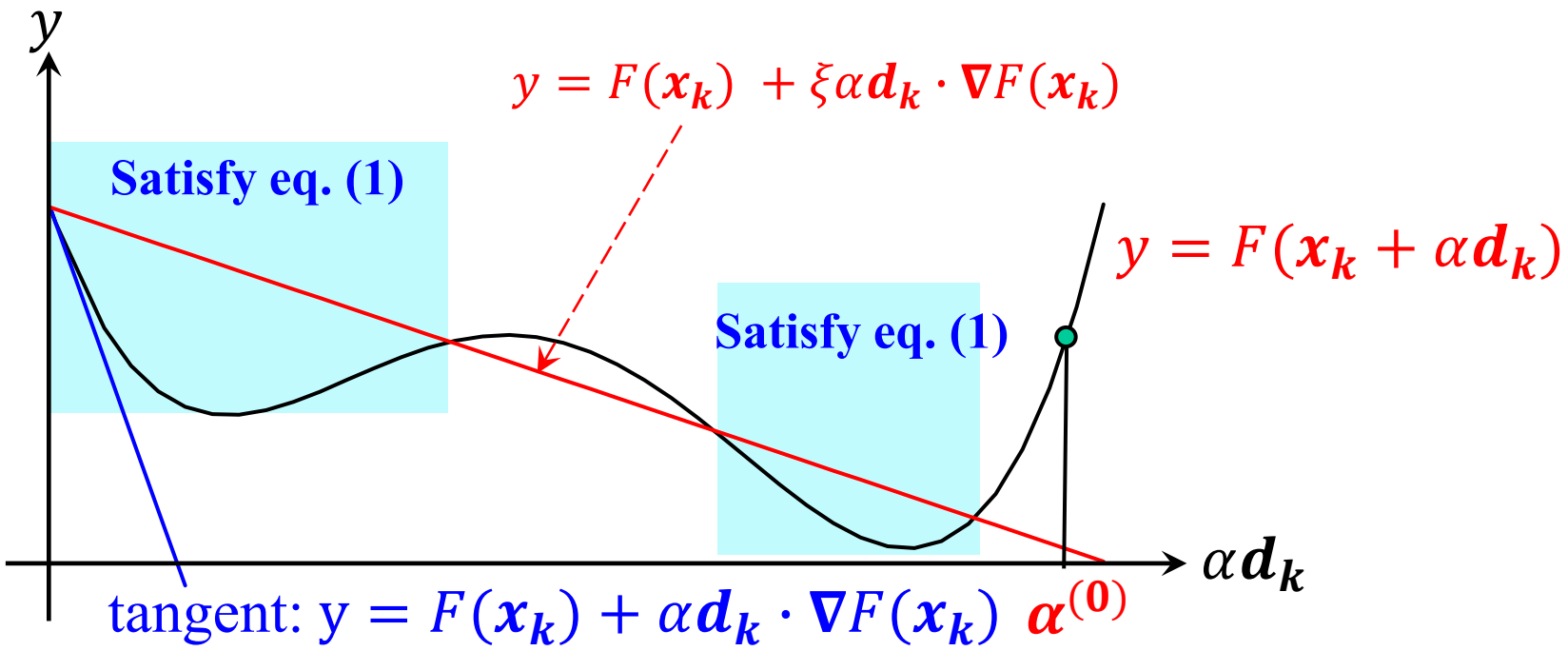
1. **Employ a large initial search range**
2. **Not use a direct value of gradient**

Line search (直線探索法): Armijo condition

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Armijo (アルミホ) condition (eq. (1)) and algorism:

1. **Provide initial x_k, choose constant ξ and τ ($0 < \xi < 1$, $0 < \tau < 1$)**
2. **Find search direction d_k** (e.g., by steepest descent method)
3. **Find $\alpha > 0$ so as to satisfy** $F(x_k + \alpha d_k) \leq F(x_k) + \xi \alpha d_k \cdot \nabla F(x_k)$ **(1)**
   (i) $\beta_{k,0} = 1, i = 0$
   (ii) if $F(x_k + \beta_{k,i} d_k) \leq F(x_k) + \xi \beta_{k,i} d_k \cdot \nabla F(x_k)$ go to **step 4**, or go to (iii)
   (iii) $\beta_{k,i+1} = \tau \beta_{k,i}$ and go to (ii)
4. $\alpha = \beta_{k,i}$

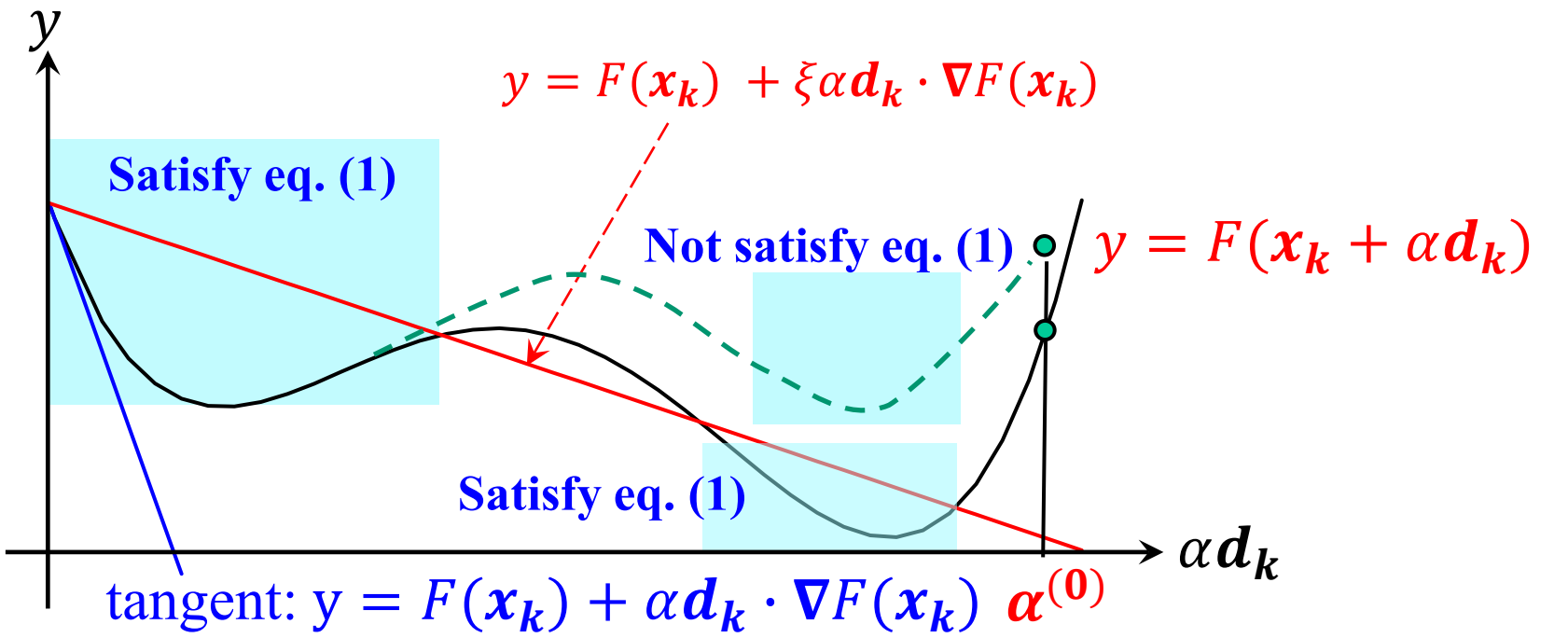


Line search (直線探索法): Armijo condition

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Armijo (アルミホ) condition (eq. (1)) and algorism:

1. **Provide initial x_k, choose constant ξ and τ ($0 < \xi < 1$, $0 < \tau < 1$)**
2. **Find search direction d_k** (e.g., by steepest descent method)
3. **Find $\alpha > 0$ so as to satisfy** $F(x_k + \alpha d_k) \leq F(x_k) + \xi \alpha d_k \cdot \nabla F(x_k)$ **(1)**
   (i) $\beta_{k,0} = 1, i = 0$
   (ii) if $F(x_k + \beta_{k,i} d_k) \leq F(x_k) + \xi \beta_{k,i} d_k \cdot \nabla F(x_k)$ go to **step 4**, or go to (iii)
   (iii) $\beta_{k,i+1} = \tau \beta_{k,i}$ and go to (ii)
4. $\alpha = \beta_{k,i}$



Line search (直線探索法): Wolfe condition

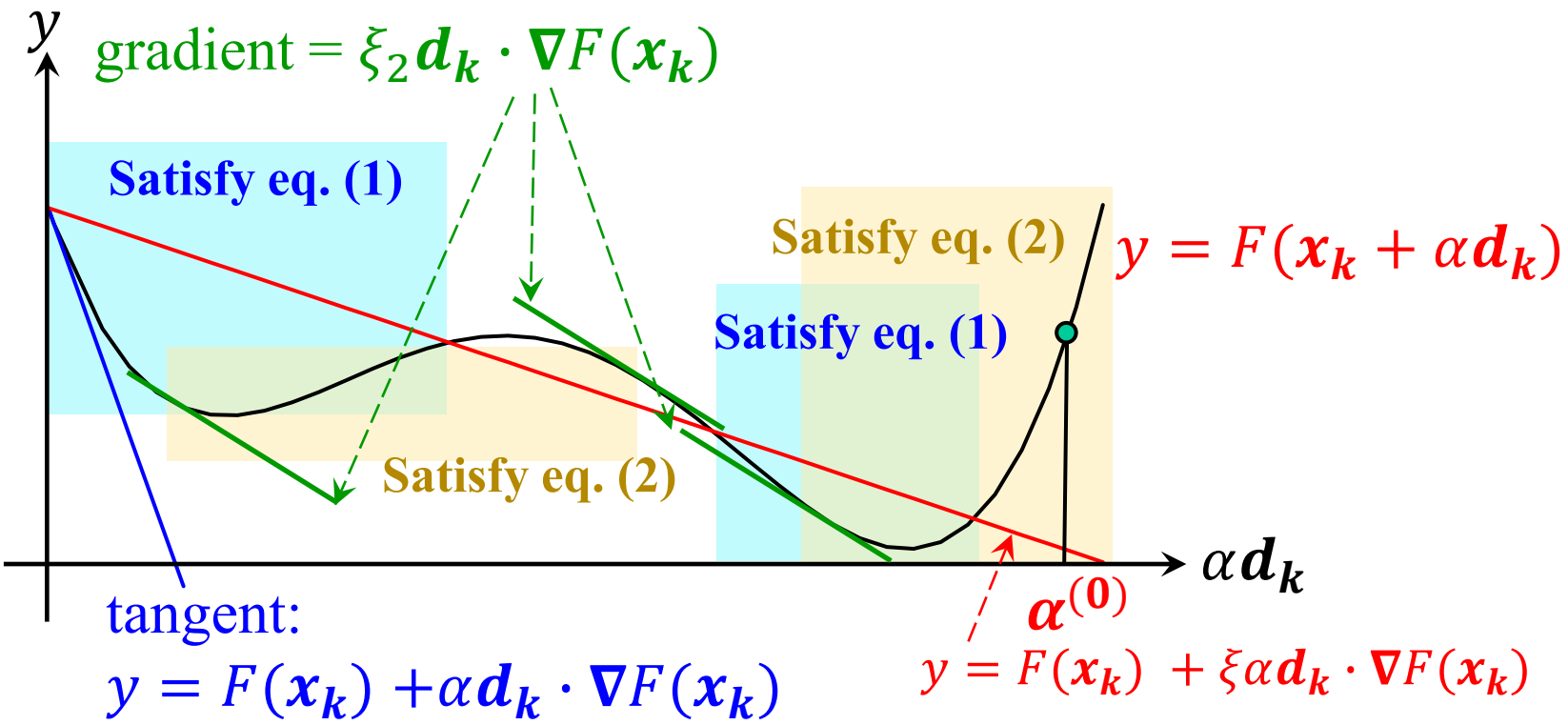
矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Wolfe (ウルフ) condition:

1. **Find search direction** $\boldsymbol{d_k}$ (e.g., by steepest descent method)
2. **Choose constants** ξ_1 **and** ξ_2 **that satisfy** $0 < \xi_1 < \xi_2 < 1$
3. **Find** $\alpha > 0$ **so as to satisfy:**

$$F(\boldsymbol{x_k} + \alpha \boldsymbol{d_k}) \leq F(\boldsymbol{x_k}) + \xi \alpha \boldsymbol{d_k} \cdot \nabla F(\boldsymbol{x_k}) \quad (1)$$

$$\xi_2 \boldsymbol{d_k} \cdot \nabla F(\boldsymbol{x_k}) \leq \boldsymbol{d_k} \cdot \nabla F(\boldsymbol{x_k} + \alpha \boldsymbol{d_k}) \quad (2)$$



Steepest Descend (SD) method (最急降下法)

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

Search minimum only by first derivatives. Simplest one among gradient methods

- **SD: $F(x, y)$ would decrease in the vector $-(dF/dx_i)dx_i$**

  $x_i^{(k+1)} = x_i^{(k)} - \alpha(dF/dx_i)$

  α_k may be a small constant step

  or determined by a line search method

***ex.* in right figure:**

$F(x_1, x_2) = 5x_1^2 + x_2^2$, initial $x_1 = 0.7, x_2 = 1.5$

- **Newton method**

  One cycle calculation provides the final solution for quadratic problems

  楕円問題の場合は一度目の計算で最適値に到達

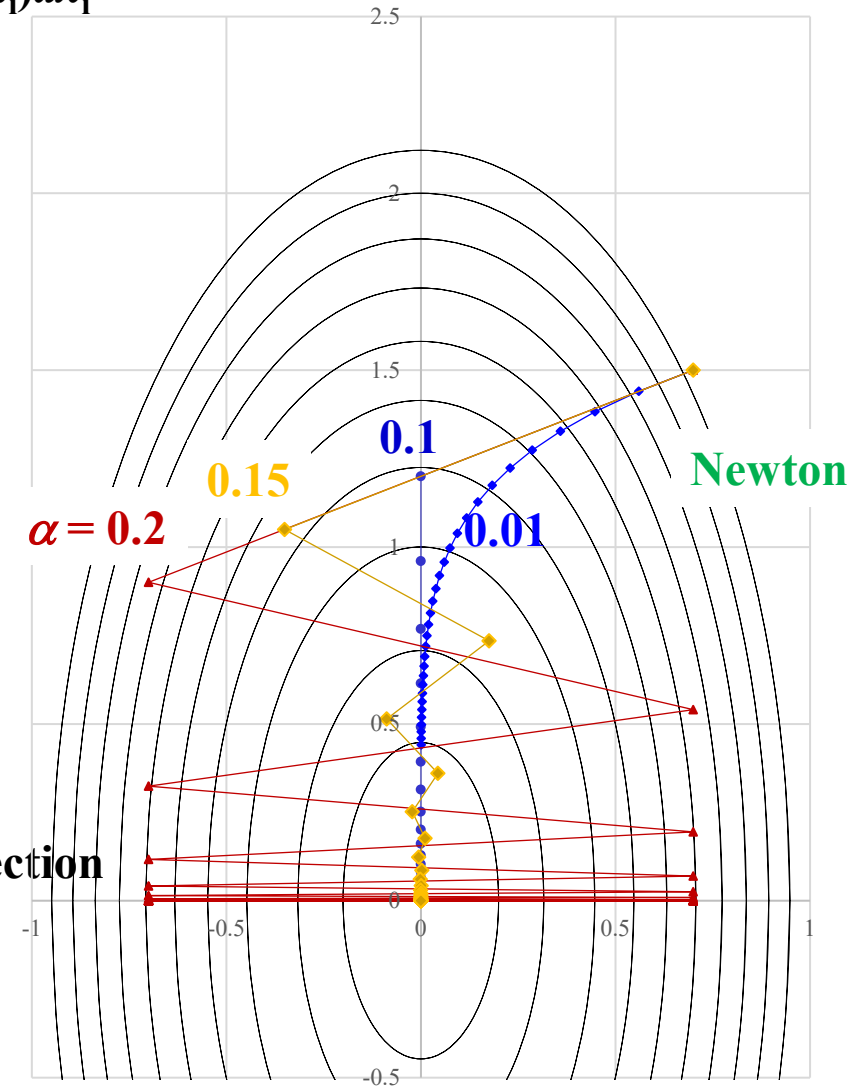
- **SD method**

  α = 0.3: Diverged (not shown in the graph)

  0.2, 0.15: Converged, but oscillated

  0.1: Reach final solution by one cycle calculation

  0.01: Not oscillated, but slowly converged

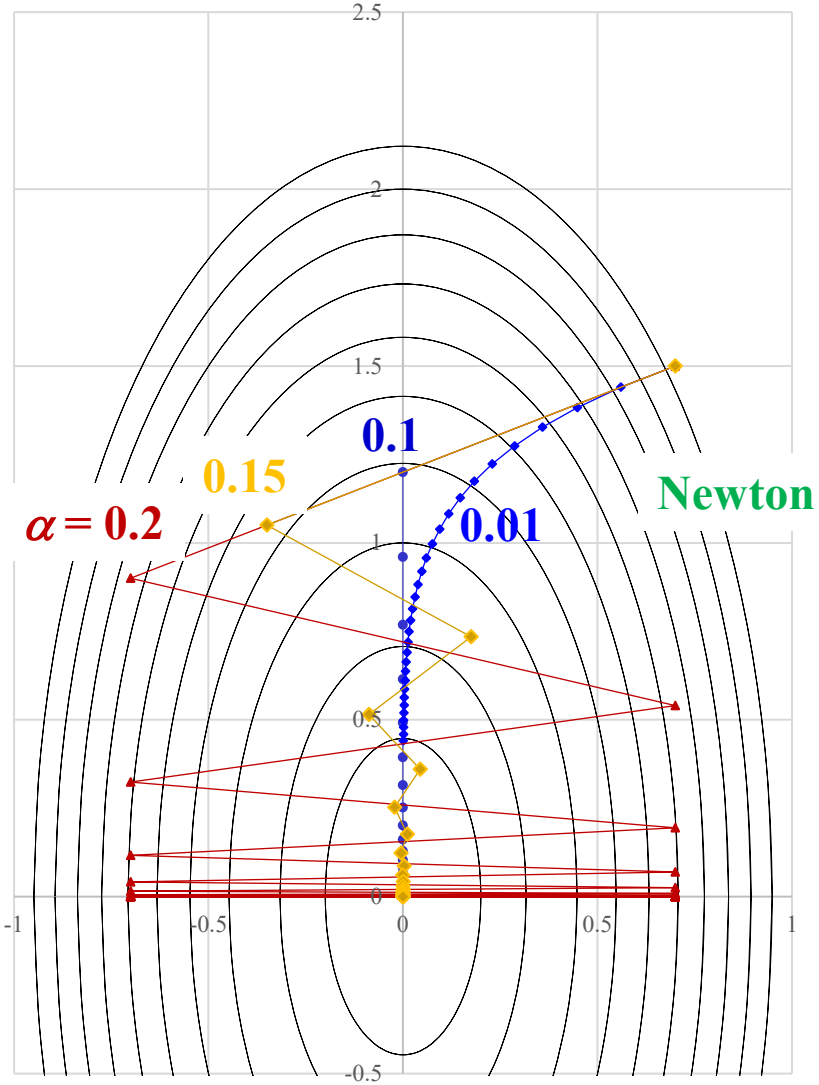


Problem: If $F(x_i)$ is highly anisotropic, the SD direction would be different largely from the minimization direction $F(x_i)$ が大きく非対称な場合、最急勾配方向は最小値方向とは大きく異なることがある

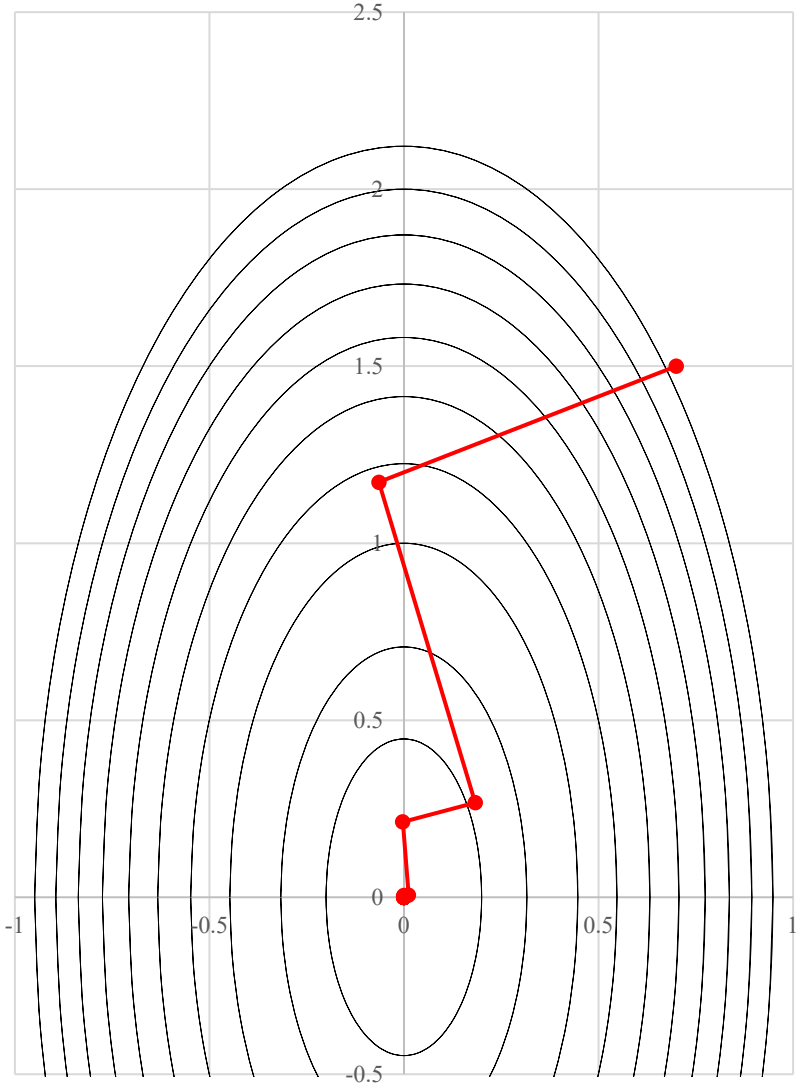
=> **Conjugate Gradient (CG)** method (共役勾配法)

Steepest descend method

直線探索法を使わず、
αを固定する

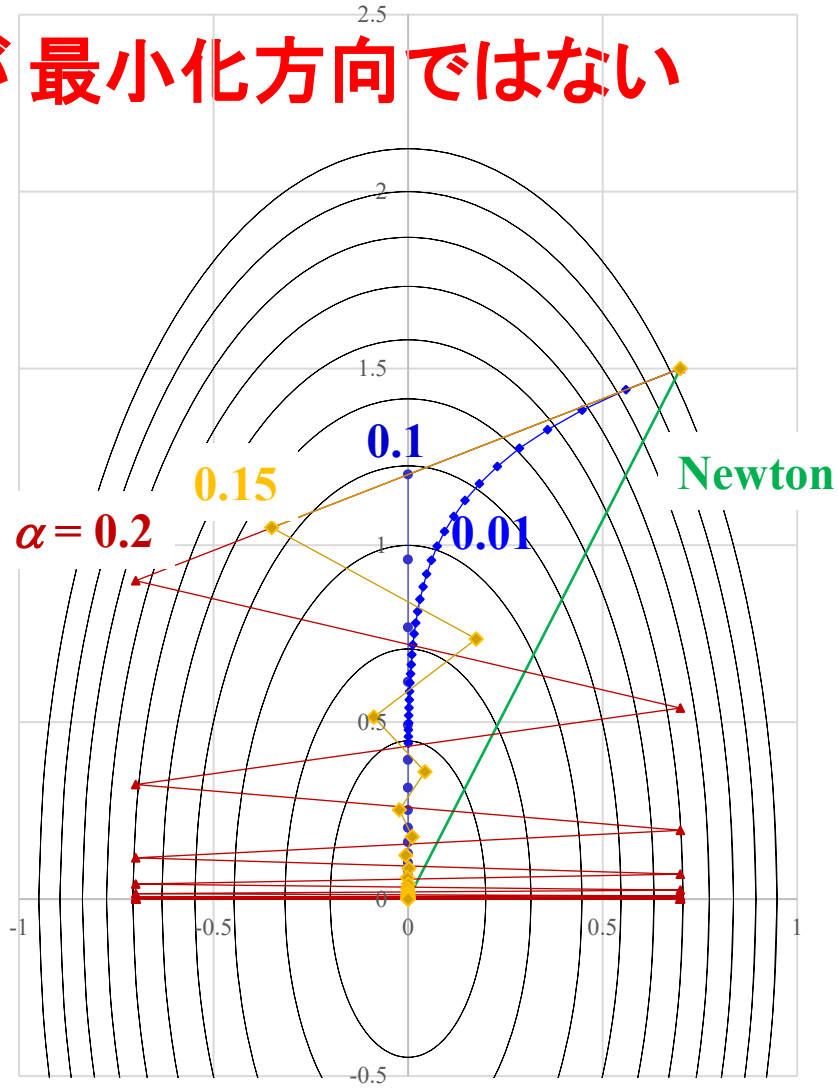


直線探索法



SD vs. Newton-Raphson methods

最急降下方向 が 最小化方向ではない



SD法を、最小化方向を使うように改良する

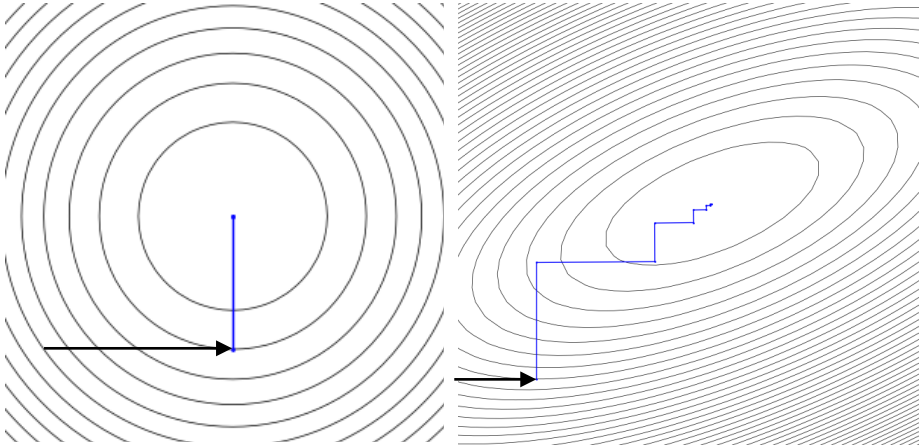
共役勾配 (Conjugate Gradient) 法

矢部博, 工学基礎 最適化とその応用, 数理工学社 (2006)

行列 A に対してベクトル u, v が $u^t Av = 0$ であるとき、u と v は互いに共役の関係にあるという

・共役な探索方向に沿って正確な直線探索を実行していけば、
有限回の反復で2次関数の最小解に到達することが期待される

等高線が円の場合、変数個数回の探索で最小値に到達できるが・・・



共役なベクトルと楕円ー円変換

$u^{\mathrm{T}}\mathrm{P}^{\mathrm{T}}\mathrm{P}v = u^{\mathrm{T}}\mathrm{A}v = 0$

1. 初期値 x_0 を与える
2. 初期探索方向 d を再急降下方向にとる

$$\mathbf{d} = -\nabla f$$

3. 直接探索法に従ってα_kを決め、

$$x_{k+1} = x_k + \alpha_k d_k$$

を計算する

4. 探索方向を

$$\mathbf{d}_k = \mathbf{d}_{k-1} - \frac{\mathbf{d}_{k-1} \cdot \nabla f(\mathbf{x}_k)}{\mathbf{d}_{k-1} \cdot \mathbf{d}_{k-1}} \nabla f(\mathbf{x}_k)$$

に更新する

5. 3,4を繰り返して収束させる

4.では、$\mathbf{d}_k$は $\mathbf{d}_i$ ($i = 1, \ldots, k$-1) のすべてに直交するため、このループは有限回しか実行できない => 時々$\mathbf{d}_k$をリセット

Marquart法

N個の変数 x_i をもつm個の関数 $f_j(x_i)$の自乗和

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

の最小値(最大値)を求める

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

と近似すると、

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

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

$$\boldsymbol{\delta}\mathbf{x}=-\left(\mathbf{A}^{\mathbf{t}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$ **Gauss-Newton法**

Levenberg-Marquart法

$$\boldsymbol{\delta}\mathbf{x}=-\left(\mathbf{A}^{\mathbf{t}}\mathbf{A}+\lambda I\right)^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$ **λの最適値がよくわからない**

$$\boldsymbol{\delta}\mathbf{x}=-\left(\mathbf{A}^{\mathbf{t}}\mathbf{A}+\lambda\,\mathrm{diag}(\mathbf{A}^{\mathbf{t}}\mathbf{A})\right)^{-1}\mathbf{A}^{\mathbf{t}}\left(f_j\right)$$ **A行列の対角和に比例させる**

直線探索法

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）

単体: n次元空間で(n+1) 個の頂点が作る多面体)

$F(x_i)$ を最小化する

1. $(n+1)$ 個の初期点 x_i $(i = 1, 2, \cdots, n+1) \Rightarrow F(x_i) > F(x_{i'})$ $(i < i')$ となるように
   ソートし、$x_h = x_1, x_l = x_{n+1}$ とする
2. 最大値をとる点以外の平均を $x_G = \sum_{i=2} x_i / n$ とする
3. 新しい x は $x_1 - x_G$ 線上において次の規則でみつける

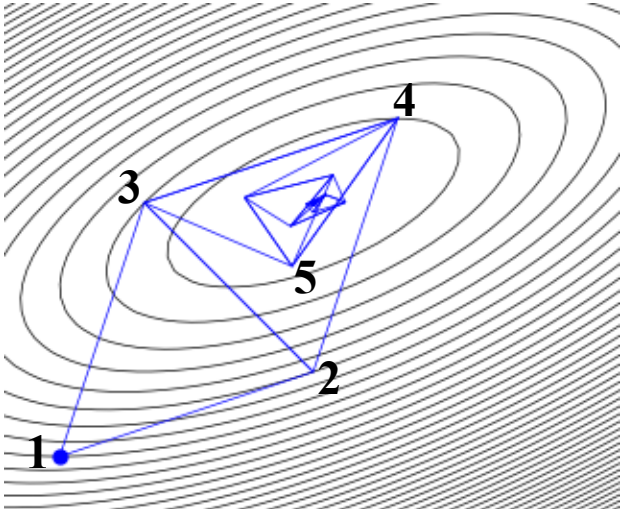
(i) Reflection (鏡映) : $x_R = (1+\alpha)x_G - \alpha x_1$ $(\alpha > 0, ex.\ 1.0)$

(ii) Expansion (拡大) : $x_E = \gamma x_r + (1-\gamma)x_G$ $(\gamma > 0, ex.\ 2.0)$

(iii) Contraction (収縮) : $x_C = \beta x_1 + (1-\beta)x_G$ $(0 < \beta < 1, ex.\ 0.5)$

(iv) Reduction (縮小) : $x_{RD} = (x_1 + x_l) / 2$

4. (i) – (iv) のうち、最初に $F(x) < F(x_1)$ を満たす
   x で x_1 を置き換える
5. 2 – 4 を繰り返す



最小化のpythonプログラム: Simplex法

scipy.optimize.minimize()

[tkProg]¥tkprog_tutorial¥optimize¥minimize_tutorial.py

```
# Simplex法を選択。微分を計算する必要がない
method = "nelder-mead"

# 最小化する関数
def minimize_func(x):
    return 2.0 * (x[0] - 3.0)**2 + (x[1] - 1.0)**4 + 2.0

def main():
# 最小化を実行
    res = minimize(minimize_func, x0s, method = method, tol = tol, options = {'maxiter':maxiter, "disp":True})
    print("")
    if res.success:
        print(f"Function takes the minimum")
# 最小値
        print(f"   at y={res.fun}")
# 最小値を取るときの x
        print(f"   with x={res.x}")
# 収束するまでの繰り返し数
        print(f" iteration: {res.nit}")
    else:
        print(f"Function did not converge")
        print(res)

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

scipy.optimize.minimize()で使えるアルゴリズム

[tkProg]¥tkprog_tutorial¥optimize¥minimize_tutorial2.py

#nelder-mead　Downhill simplex
#powell　　Modified Powell
#cg　　conjugate gradient (Polak-Ribiere method)
#bfgs　　BFGS法
#newton-cg　　Newton-CG
#trust-ncg　　信頼領域 Newton-CG 法
#dogleg　　信頼領域 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)

最小化のpythonプログラム: 勾配法

scipy.optimize.minimize()

[tkProg]¥tkprog_tutorial¥optimize¥minimize_tutorial2.py

```
# BFGS法を選択。1次微分を計算する関数が必要
method = "bfgs"

# 最小化する関数
def minimize_func(x):
    return 2.0 * (x[0] - 3.0)**2 + (x[1] - 1.0)**4 + 2.0

# 1次微分を計算する関数
# 中央差分で微分を計算
def diff1(x):
     for i in range(nvar):
        xx = x.copy()
        xx[i] = x[i] - h
        ym = minimize_func(xx)

        xx[i] = x[i] + h
        yp = minimize_func(xx)
        diff[i] = (yp - ym) / 2.0 / h
    return diff

def main():
# 最小化を実行。jacに1次微分を計算する関数を渡す
 res = minimize(minimize_func, x0s, method = method, jac = diff1, callback = callback,
                tol = tol, options = {'maxiter':maxiter, "disp":True})
#    print("")
```

jac に ‘3-point’ を渡すと3点法で微分を計算してくれるはず

数値微分: 中央差分を取る

前進差分: $\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x)}{h}$ $x + h/2$ の微分値が最も精確に出る

中央差分: $\frac{df(x)}{dx} \sim \frac{f(x+h)-f(x-h)}{2h}$ この式を使う

二階微分

$$\frac{d^2f(x)}{dx^2} \sim = \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$$

高次の公式

3点公式

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

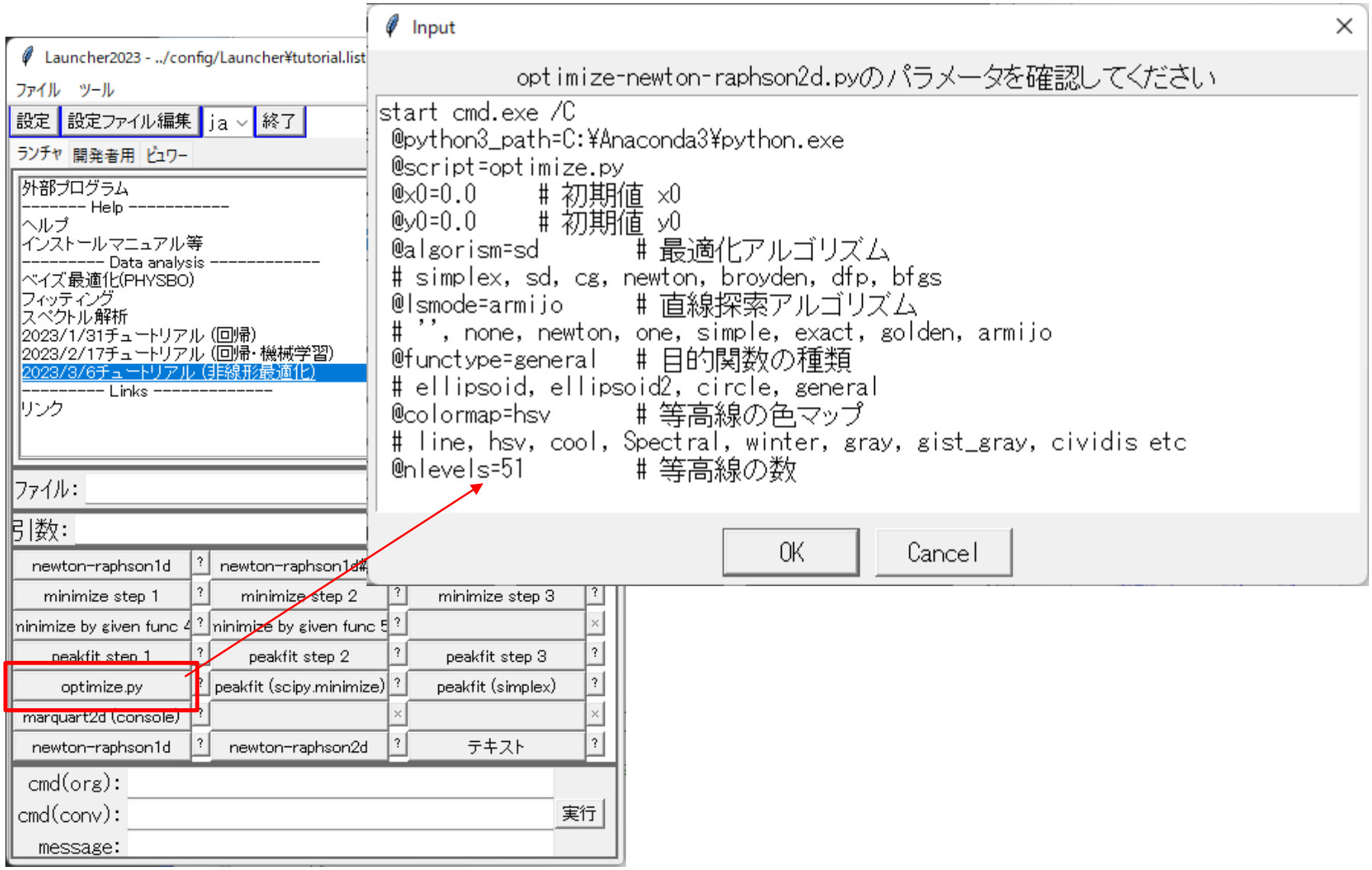
5点公式

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

7点公式

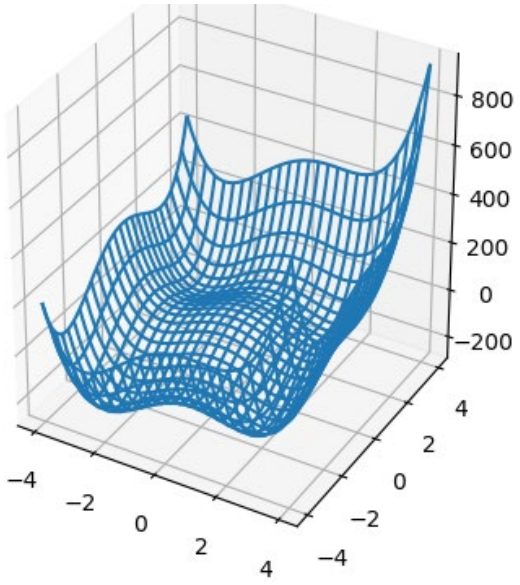
$$f'(a) = \frac{1}{h}\left\{\frac{1}{60}f(a+3h) - \frac{3}{20}f(a+2h) + \frac{3}{4}f(a+h) - \frac{3}{4}f(a-h) + \frac{3}{20}f(a-2h) - \frac{1}{60}f(a-3h)\right\} + \frac{1}{140}f^{(7)}(a)h^6 + \cdots$$

2変数最適化: アルゴリズムの比較

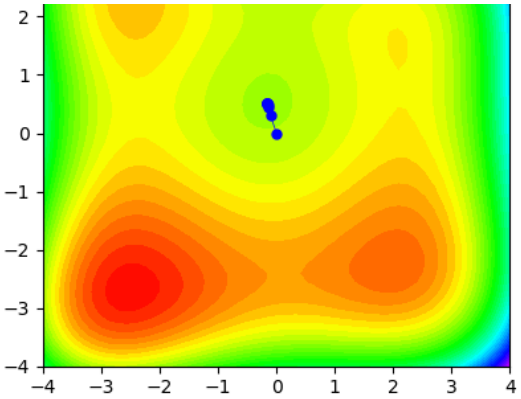


2変数最適化: アルゴリズムの比較

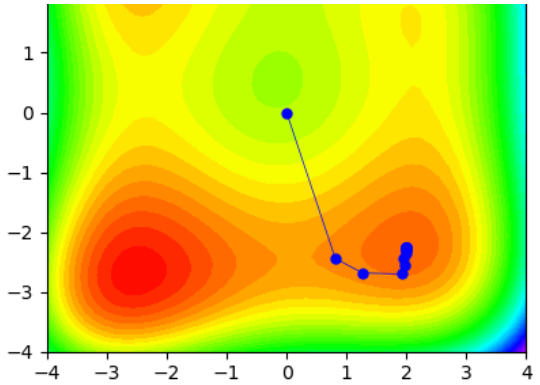
[tkProg]¥tkprog_tutorial¥optimize¥optimize.py



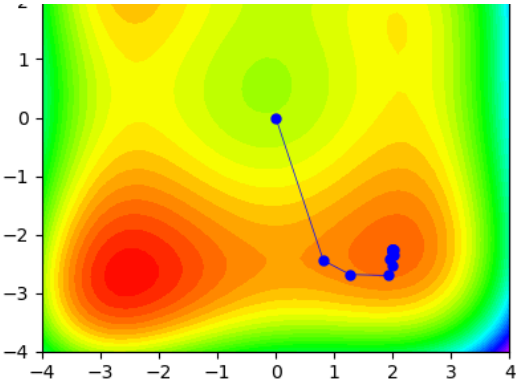
From (0.0 0.0) Newton
algorism=newton
lsmode=newton



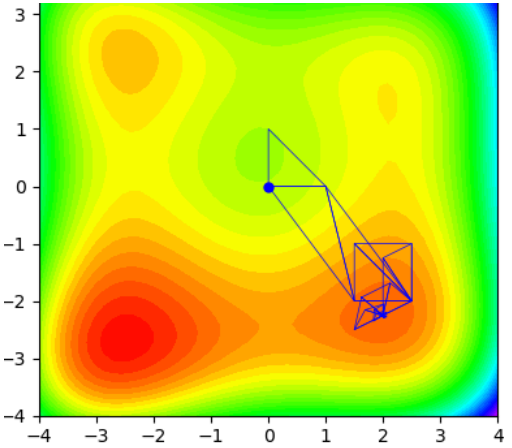
From (0.0 0.0) DFP
algorism=dfp
lsmode=golden



From (0.0 0.0) BFGS
algorism=bfgs
lsmode=golden



From (0.0 0.0) Simplex
algorism=simplex



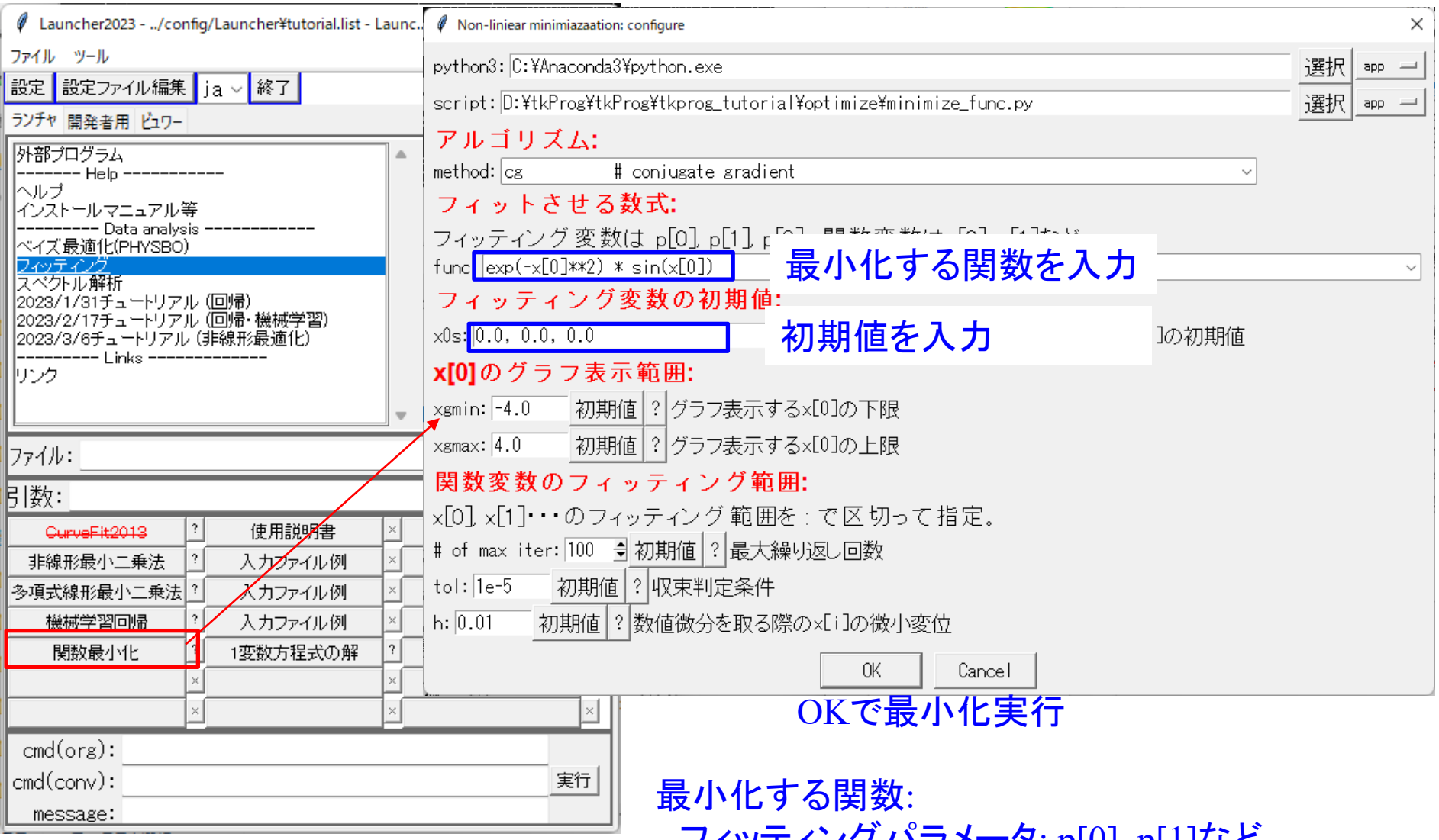
Main algorism:

Newton, DFP, BFGS
SD, CG
Simplex

Line search:

Golden, Armijo
exact/newton (NR法)

一般関数の最小化



OKで最小化実行

最小化する関数:

フィッティングパラメータ: p[0], p[1]など

関数変数: x[0], x[1]など

2変数関数の最小化

Non-liniear minimiazaation: configure

python3: C:¥Anaconda3¥python.exe 選択 app

script: D:¥tkProg¥tkProg¥tkprog_tutorial¥optimize¥minimize_func.py 選択 app

アルゴリズム:

method: cg # conjugate gradient

フィットさせる数式:

フィッティング変数は p[0], p[1], p[2]、関数変数はx[0], x[1]など。

func: exp(-x[0]**2) * sin(x[0]+x[1]) / (1 + (x[1]+1)**2)

フィッティング変数の初期値:

x0s: 0.0, 0.0, 0.0 初期値 ? x[i]の初期

x[0]のグラフ表示範囲:

xgmin: -4.0 初期値 ? グラフ表示するx[0]の下限

xgmax: 4.0 初期値 ? グラフ表示するx[0]の上限

関数変数のフィッティング範囲:

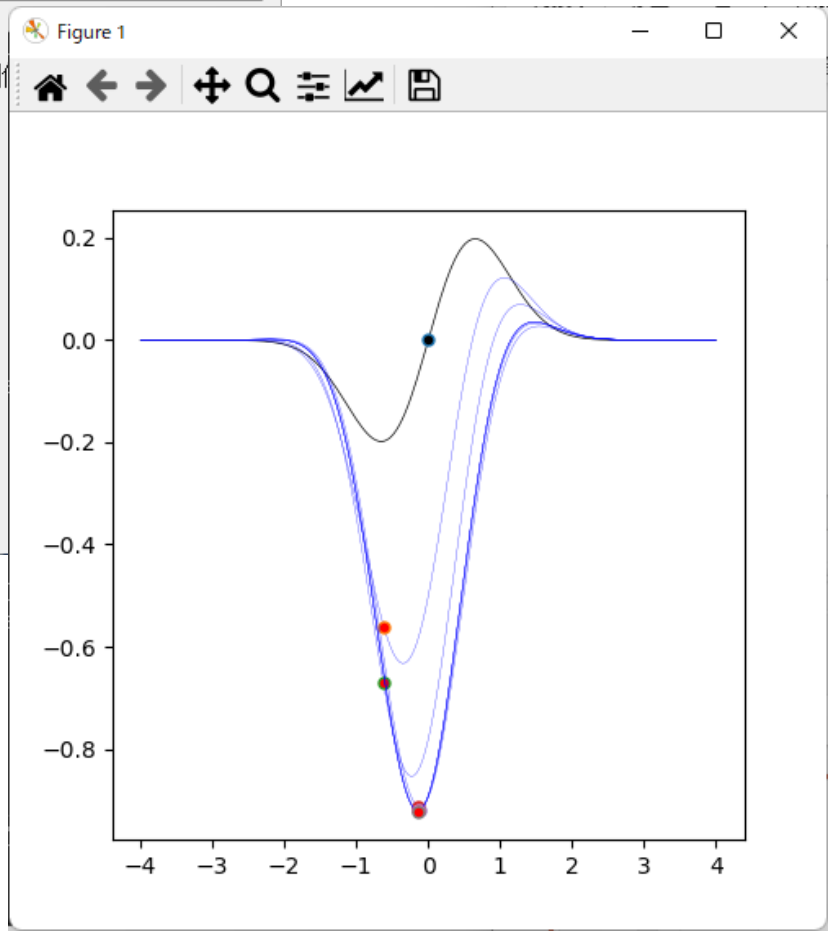
x[0], x[1]・・・のフィッティング範囲を：で区切って指定。

# of max iter: 100 初期値 ? 最大繰り返し回数

tol: 1e-5 初期値 ? 収束判定条件

h: 0.01 初期値 ? 数値微分を取る際のx[i]の微小変位

OK Cancel



分光解析に使われるプロファイルモデル

Lorentz関数

$$I_L(x) = \frac{1}{1+\left[(x-x_0)/w\right]^2}$$

w: 半値半幅

Gauss関数

$$I_G(x) = \frac{1}{a_w w \pi^{1/2}} \exp\left\{-\left[(x-x_0)/(a_w w)\right]^2\right\}$$

$$a_w = (\ln 2)^{-1/2} = 0.832554611$$

Voigt関数:

Lotentz型の固有スペクトルに
他の要因のGauss型の広がりが重なる
畳み込み積分 (Convolution) であらわされる

$$I_V(x) = \int_{-\infty}^{\infty} I_G(x') I_L(x-x') dx'$$

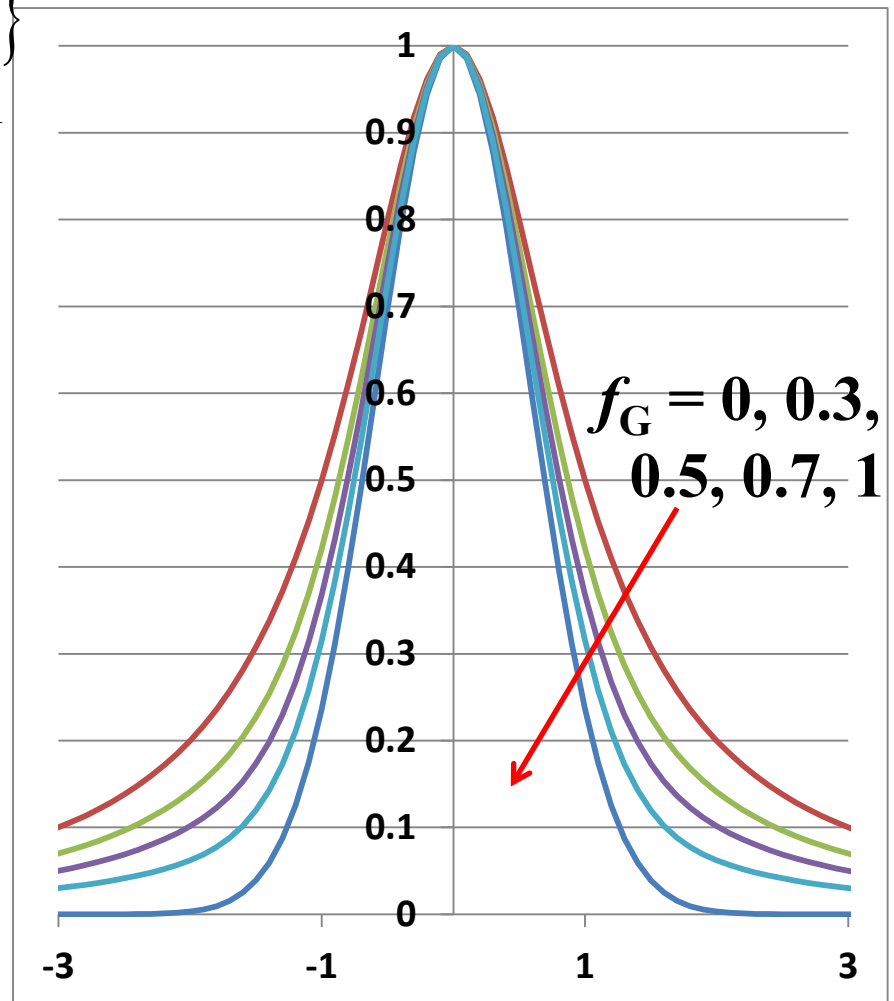
$$= \frac{a_V}{\pi} \int_{-\infty}^{\infty} \frac{\exp(-x'^2)}{a_V{}^2 + (x-x')^2} dx'$$

Pseudo-Voigt関数:

Voigt関数の簡略版

$$I_{PV}(x) = f_G I_G(x) + (1-f_G) I_L(x)$$

f_G: Gauss関数分率



カーブフィット

一般的な目的関数の例: 複数のpseudo-Voigt関数の和

$$F(x_{o,k}, f_{G,k}, f_{L,k} w_{G,k}, w_{L,k}) = \sum_{data} \left(y_i - I_{pV}(x_i) \right)^2$$

$$I_{pV,k}(x_i) = f_{G,k} I_{G,k}(x_i) + f_{L,k} I_{L,k}(x_i)$$

$$I_{G,k}(x_i) = \frac{1}{a_w w_{G,k} \pi^{1/2}} \exp\left\{ -\left[\left(x_i - x_{0,k} \right) / \left(a_w w_{G,k} \right) \right]^2 \right\}$$

$$I_{L,k}(x_i) = \frac{1}{1 + \left[\left(x_i - x_{o,k} \right) / w_{L,k} \right]^2}$$

Program: peakfit-scipy-simple.py

Simplex法を使っているので、微分の計算の必要がない。もっとも簡単なプログラム

```
# パラメータリスト params を与えて x におけるピーク強度を計算
def ycal(x, params):
  return intensity

# xのリスト xd とパラメータリスト params を与えて y のリストを返す
def ycal_list(xd, params):
   return ylist

# yの入力値のリスト yd とパラメータリスト params から 目的関数 S2 を計算
def CalS2(params):
   return S2

def main():
   df = pd.read_excel(infile, engine = 'openpyxl')          #データの読み込み
   xd = df[labels[0]].to_numpy()
   yd = df[labels[1]].to_numpy()

   yini = ycal_list(xd, x0)                                 # 初期値からピーク形状を計算

   res = minimize(CalS2, x0, method = method, tol = tol, options = {'maxiter':maxiter, "disp":True})
                                                            # 初期値を与えて最適化。method = "cg"で共役購買法を使っている

   yopt = ycal_list(xd, res.x)                              # 最適化したパラメータからピーク形状を計算
```

最適化の途中過程を確認したい場合は、minimizeにcallbackを渡す

Program: peakfit-scipy-minimize.py

勾配法の場合、微分計算が必要なので、minimize()にjacとして1次微分リストを返す関数 diff1 を渡す

```
# パラメータリスト params を与えて x におけるピーク強度を計算
def ycal(x, params):
  return intensity

# xのリスト xd とパラメータリスト params を与えて y のリストを返す
def ycal_list(xd, params):
   return ylist

# yの入力値のリスト yd とパラメータリスト params から 目的関数 S2 を計算
def CalS2(params):
   return S2

# i番目のパラメータ ai について、パラメータリスト params から1次微分 diff1 = dS2 / dai を計算
def diff1i(i, params):
   return diff1

# パラメータリスト params から1次微分 dS2 / dai ベクトル diffs を返す
def diff1(params):
   return diffs

def main():
   df = pd.read_excel(infile, engine = 'openpyxl')          #データの読み込み
   xd = df[labels[0]].to_numpy()
   yd = df[labels[1]].to_numpy()

   yini = ycal_list(xd, x0)                                 # 初期値からピーク形状を計算

   res = minimize(CalS2, x0, jac = diff1, method = method, tol = tol, callback = callback,
         options = {'maxiter':maxiter, "disp":True})        # 初期値を与えて最適化。method = "cg"で共役購買法を使っている

   yopt = ycal_list(xd, res.x)                              # 最適化したパラメータからピーク形状を計算
```

Program: peakfit-scipy-minimize.py

Callback関数の使い方

```
# callback関数: ある関数から呼び出される関数
# scipy.minimize()のcallback関数では、更新されたパラメータのリストが引数 xk として渡される
# 他の引数は渡されないので、global変数にするか、callbackの指定でlambda関数を定義する
iter = 0
xiter  = []
yfmin  = []
def callback(xk):
# global変数を変更する場合は、global宣言が必要。そうでないと、local変数として扱われる
    global iter, xiter, yfmin

    fmin = CalS2(xk)
    print("callback  {}: xk={}".format(iter, xk))
    print("    fmin={}".format(fmin))
    iter += 1
    xiter.append(iter)
    yfmin.append(fmin)


def main():
…
    res = minimize(CalS2, x0, jac = diff1, method = method, tol = tol, callback = callback,
            options = {'maxiter':maxiter, "disp":True})        # 初期値を与えて最適化。method = "cg"で共役購買法を使っている
```

Python tips: scipy.minimize()のcallback

Callback関数で繰り返し回数などを表示する方法

例: [tkProg]¥tkprog_tutorial¥optimize¥minimize_func1d.py

- **Global変数を使う**

```
iter = 0
def callback(xk):
    global iter                                    # 関数内でglobal変数を書き換える場合には global宣言が必要

    fmin = minimize_func(xk)
    print(f"callback {iter}: xk={xk}  func={fmin}")
    iter += 1

res = minimize(minimize_func, x0s, jac = diff1, method = method, tol = tol,
               callback = callback,
               options = {'maxiter':maxiter, "disp":True})
```

- **classを使う。__call__ を使うと、クラスのインスタンスを関数として呼び出せる**

```
class call_back():
    def __init__(self):
        self.iter = 0
    def __call__(self, xk):
        fmin = minimize_func(xk)
        print(f"callback {self.iter}: xk={xk}  func={fmin}")
        self.iter += 1

cb = call_back()
res = minimize(minimize_func, x0s, jac = diff1, method = method, tol = tol,
               callback = cb,                                  # callbackとして cb(x) が呼び出される
               options = {'maxiter':maxiter, "disp":True})
```

Python tips: Early Stopさせる方法

scikit-optimizeの場合: EarlyStopperで最適化を中断する
https://qiita.com/saiaron/items/8bf6cdf411b48a5ea59e

scipy.minimize() では、callback() が False を返したら Early Stopする

1. 割と簡単でお薦め: グラフウィンドウを閉じて終了させる
   例: [tkProg]¥tkprog_tutorial¥optimize¥lsq_func.py
   (1) 最初にグラフを表示したときにウインドウハンドルを保存
   callback.window = plt.get_current_fig_manager().window
   (2) callback()でウインドウハンドルを取得し、(1)と比較。
   異なっていたら False を返して Early Stopさせる

2. ちょっと複雑だけど正攻法: グラフウィンドウにボタンを表示、ボタンを押して終了させる
   https://note.com/evjunior/n/n73c43c1b546f などを参考にボタンとクリックイベント関数を設定
   参考: [tkProg]¥tkprog_tutorial¥optimize¥lsq_func.py で 'button' と 'subplots_adjust' を検索
   (1) ボタンを押したら stop_flag をTrueにする
   (2) callback()が呼ばれたら、stop_flag が Trueなら False を返して Early Stopさせる
   (3) tight_layout() を呼んだら subplots_adjust() を再設定する

3. ファイル検出: 面倒くさい
   callbackで os.path.exists('STOP') を実行し、STOPファイルがあったら中断
4. キーボード入力: 間違ってキー入力する可能性。入力するキーを覚えていないといけない
   keyboardモジュール keyboard.read_key()
   https://kuku81kuku81.hatenablog.com/entry/2022/06/16_python_keyboard_AvailableKeyTypes
5. 割り込み (Ctrl+C): scipyと共存しない
   https://kusanohitoshi.blogspot.com/2017/01/pythonctrl-c.html

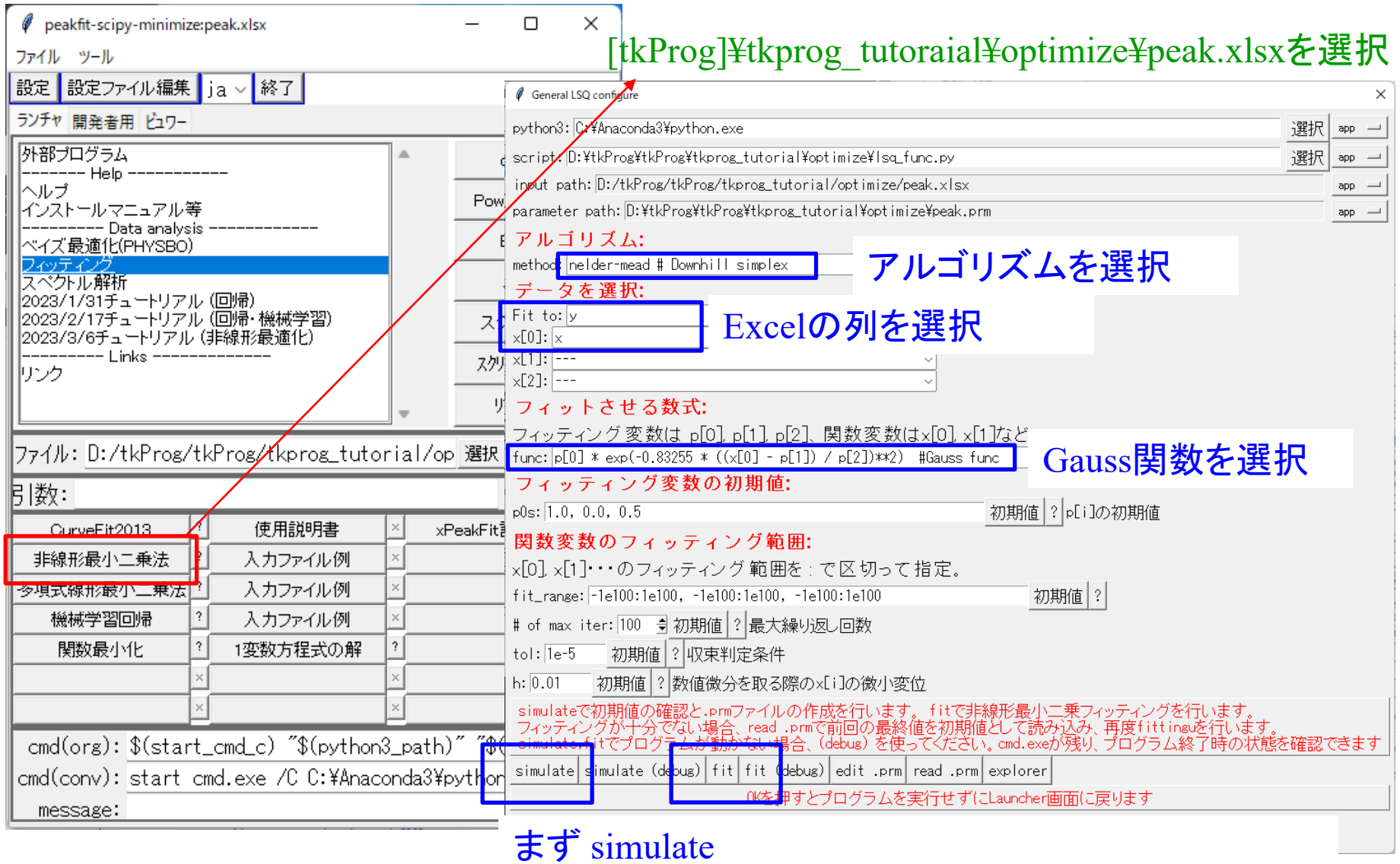
Python tips: print()の出力先を変える方法

https://ja.stackoverflow.com/questions/42919/python3-%E6%A8%99%E6%BA%96%E9%96%A2%E6%95%B0%E3%81%AB%E5%89%B2%E3%82%8A%E8%BE%BC%E3%81%BF%E5%87%A6%E7%90%86%E3%82%92%E5%85%A5%E3%82%8C%E3%81%9F%E3%81%84

builtinsモジュールに組み込み関数が入っているので、置き換える
ただし、import builtins を行ってbuiltins.printを置き換えたファイルにだけ有効

```
import builtins

# もとの組込みprint関数の値を保存しておく
print_original = print

# 自前のprint用関数を定義する
def my_print(*args, **kwargs):
 original _print("# カスタム版printが呼ばれました")     # 追加したい処理
    return orig_ original(*args , **kwargs)                    # 元のprintの呼び出し

# 組込み識別子のprintをmy_printに変える
builtins.print = my_print

print("hello, world")
```

Peakfit



まず simulate
初期値を修正してから fit

Ex.: Curve fit to Gaussian peak

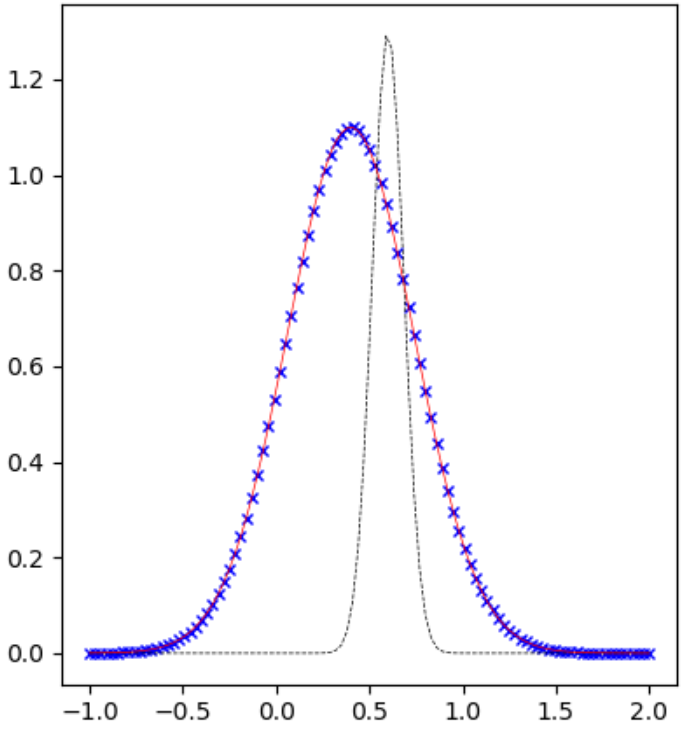
Usage: python peakfit-scipy-minimize.py I0 x0 w

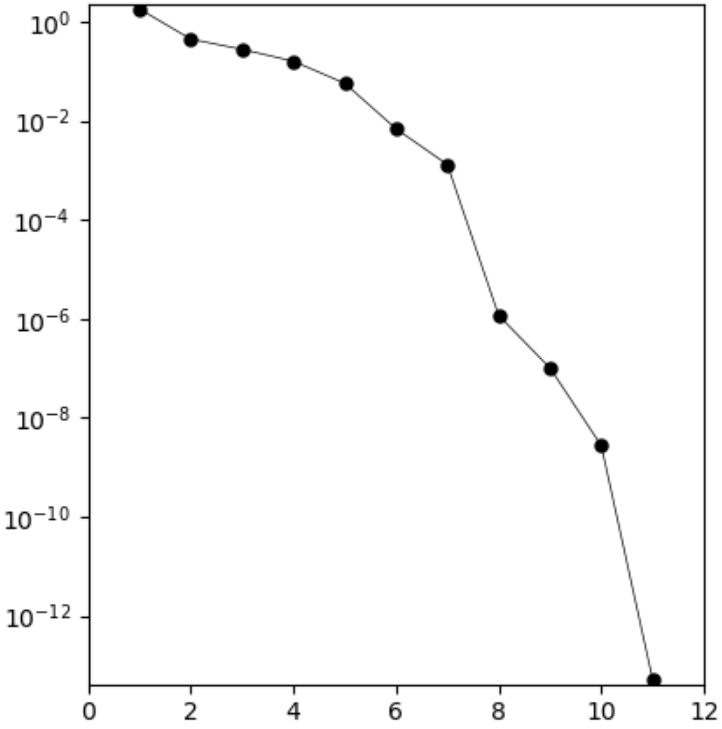
uses **scipy.minimize()** function, employs conjugate gradient method

python peakfit-scipy-minimize.py

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.6, w = 0.1



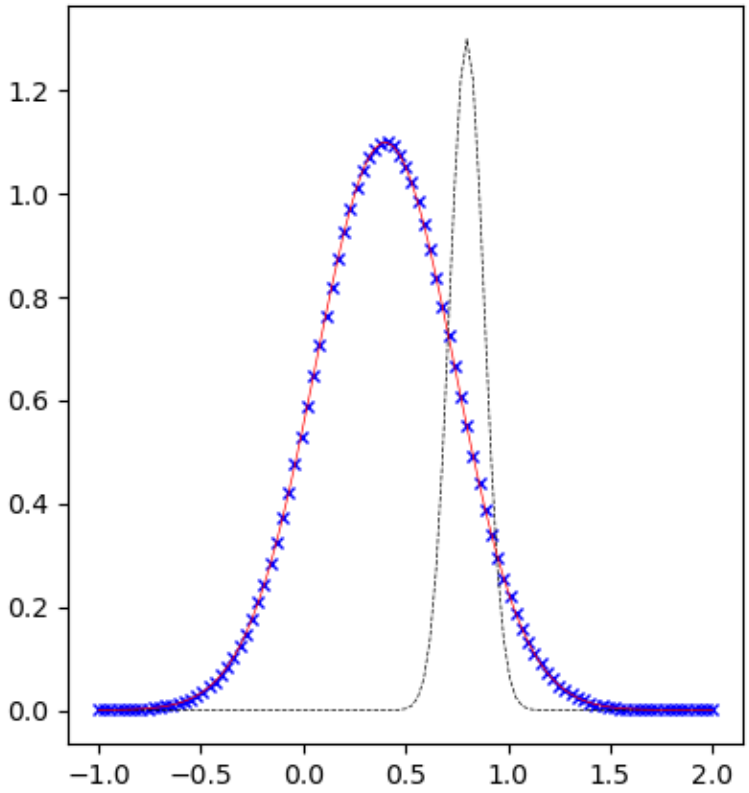


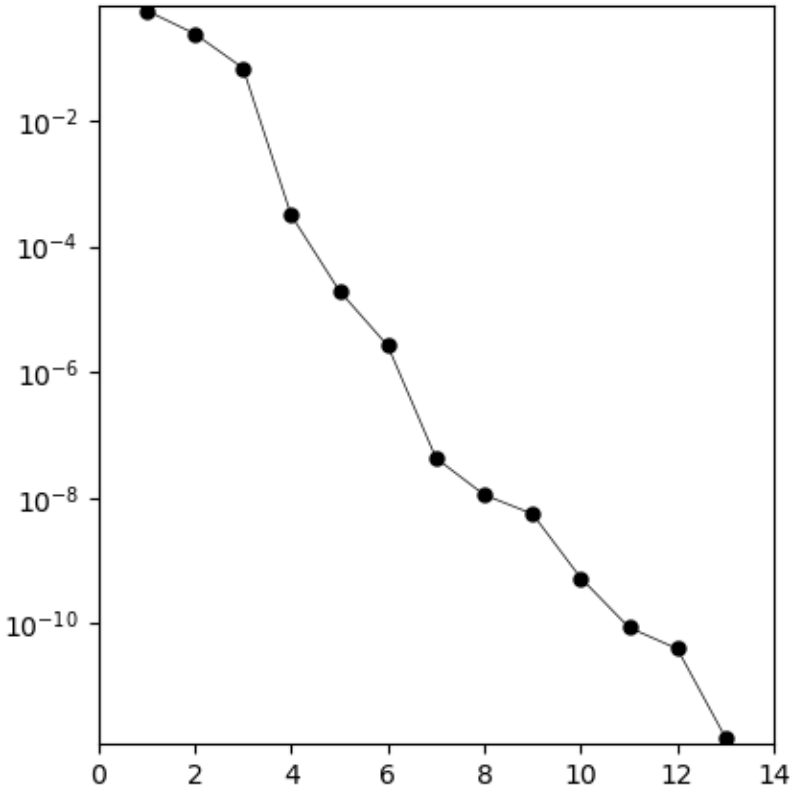
Converging range

python peakfit-scipy-minimize.py 1.3 0.8 0.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.8, w = 0.1



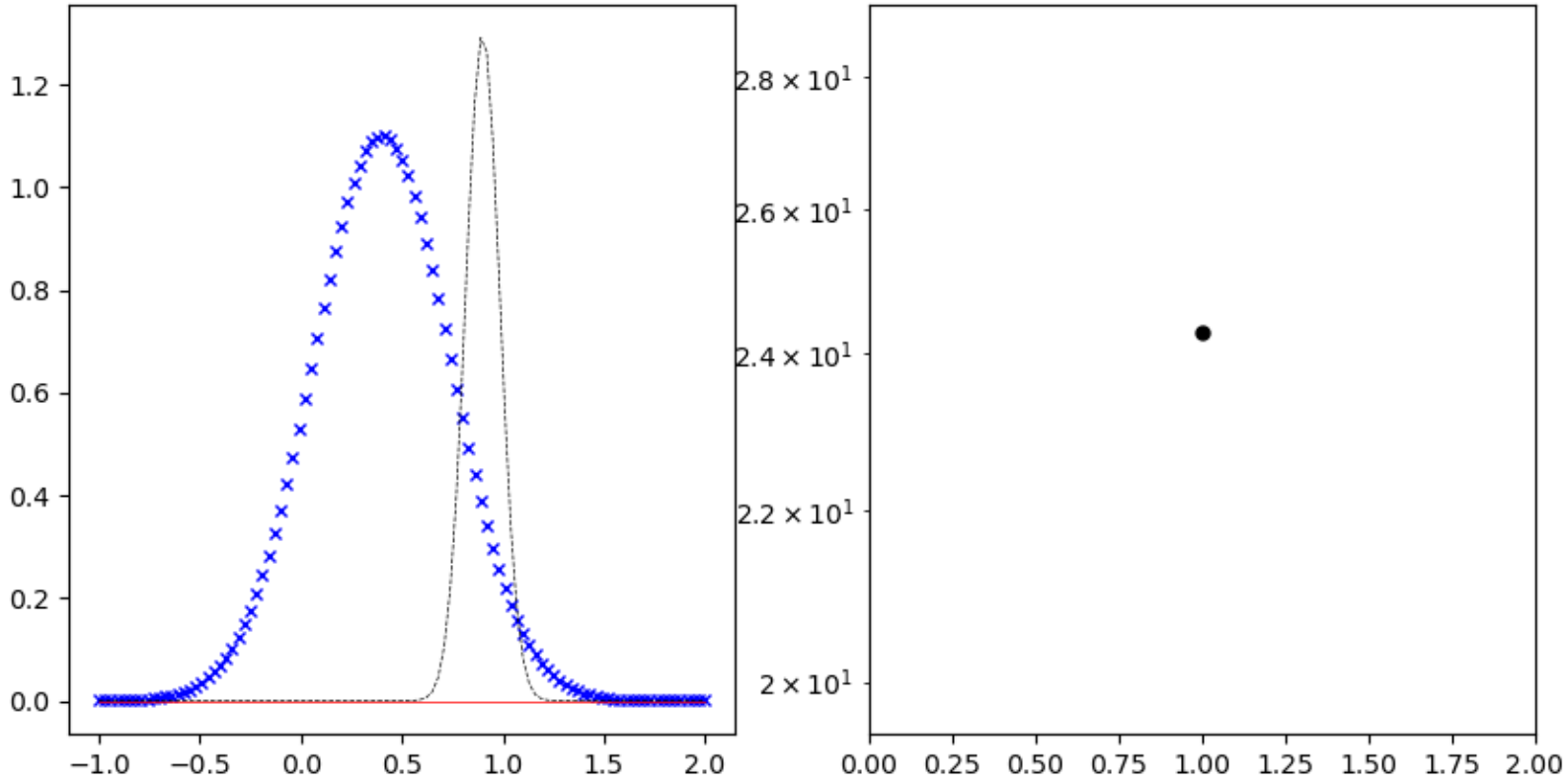


Diverged

python peakfit-scipy-minimize.py 1.3 0.9 0.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 1.3, x0 = 0.9, w = 0.1



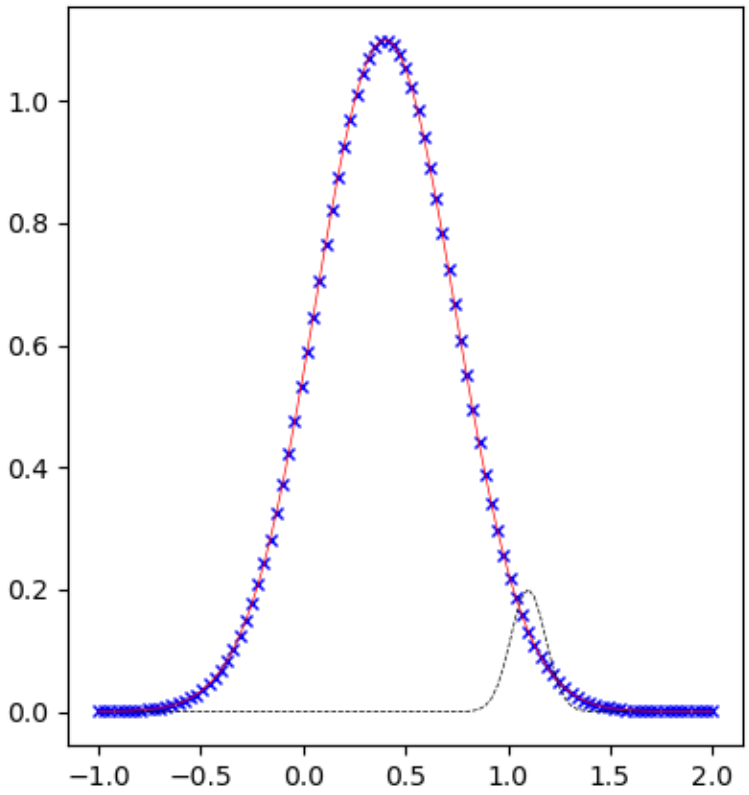
Initial peak must be in the FWHM of the target peak

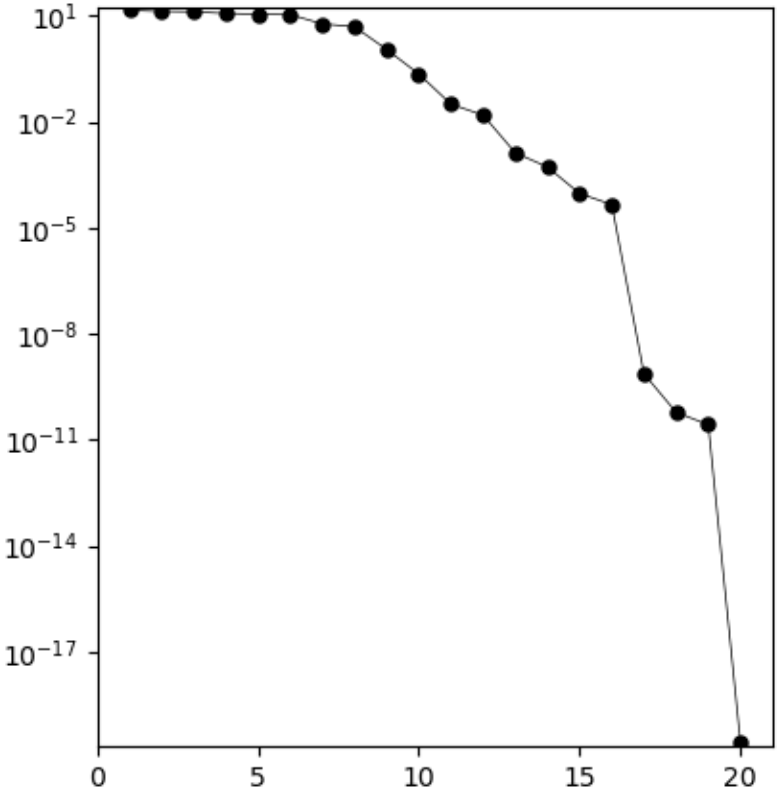
Converged

python peakfit-scipy-minimize.py 0.2 1.1 0.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 0.2, x0 = 1.1, w = 0.1





If I0 is close to the target curve, it can be converged

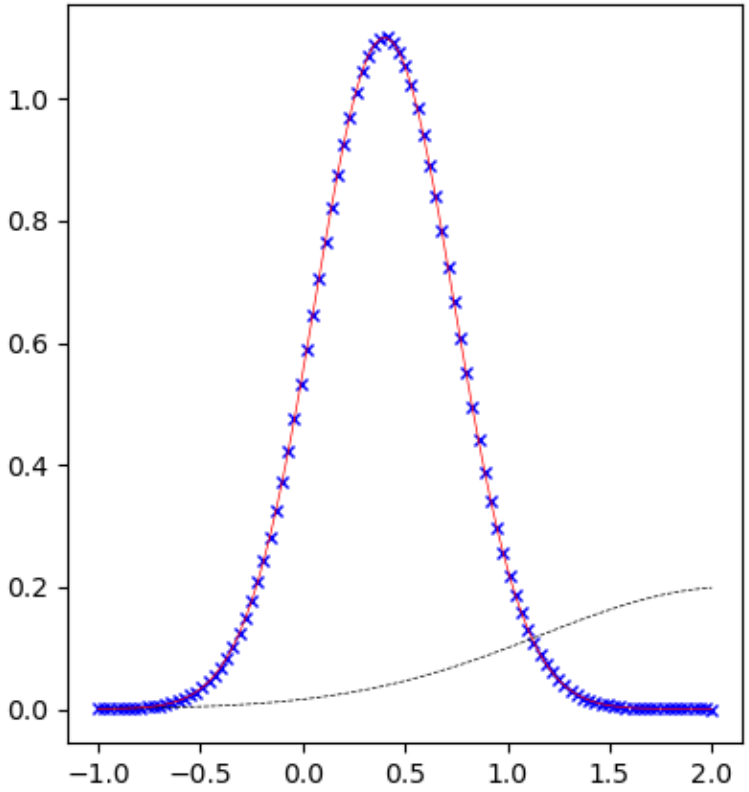
even if the initial peak position is out of the FWHM of the target peak

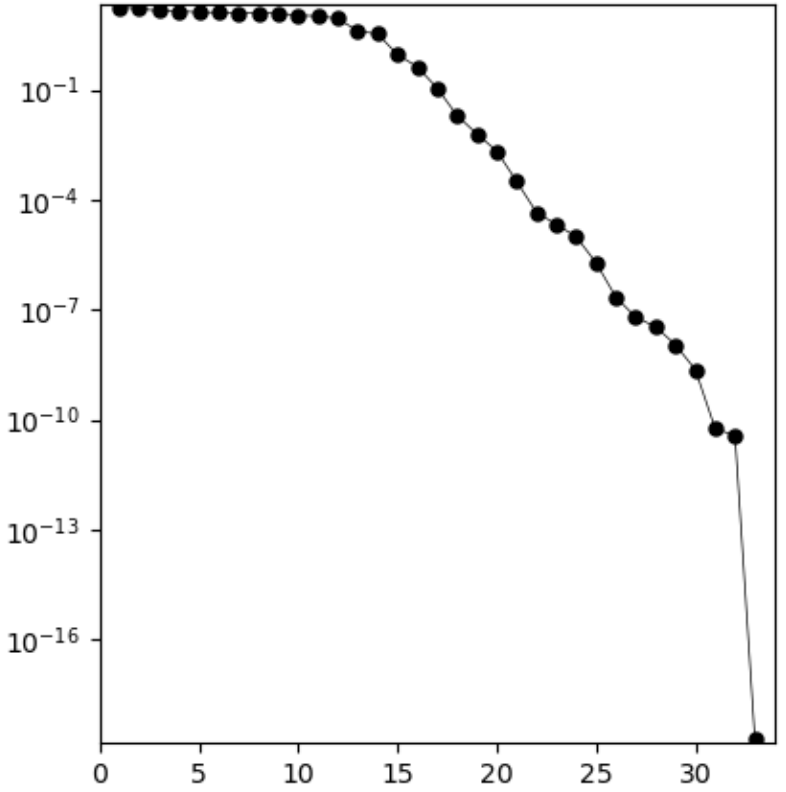
Converged

python peakfit-scipy-minimize.py 0.2 2.1 1.1

Target peak: Gaussian function, I0 = 1.1, x0 = 0.4, w = 0.4

default: I0 = 0.2, x0 = 2.1, w = 1.1



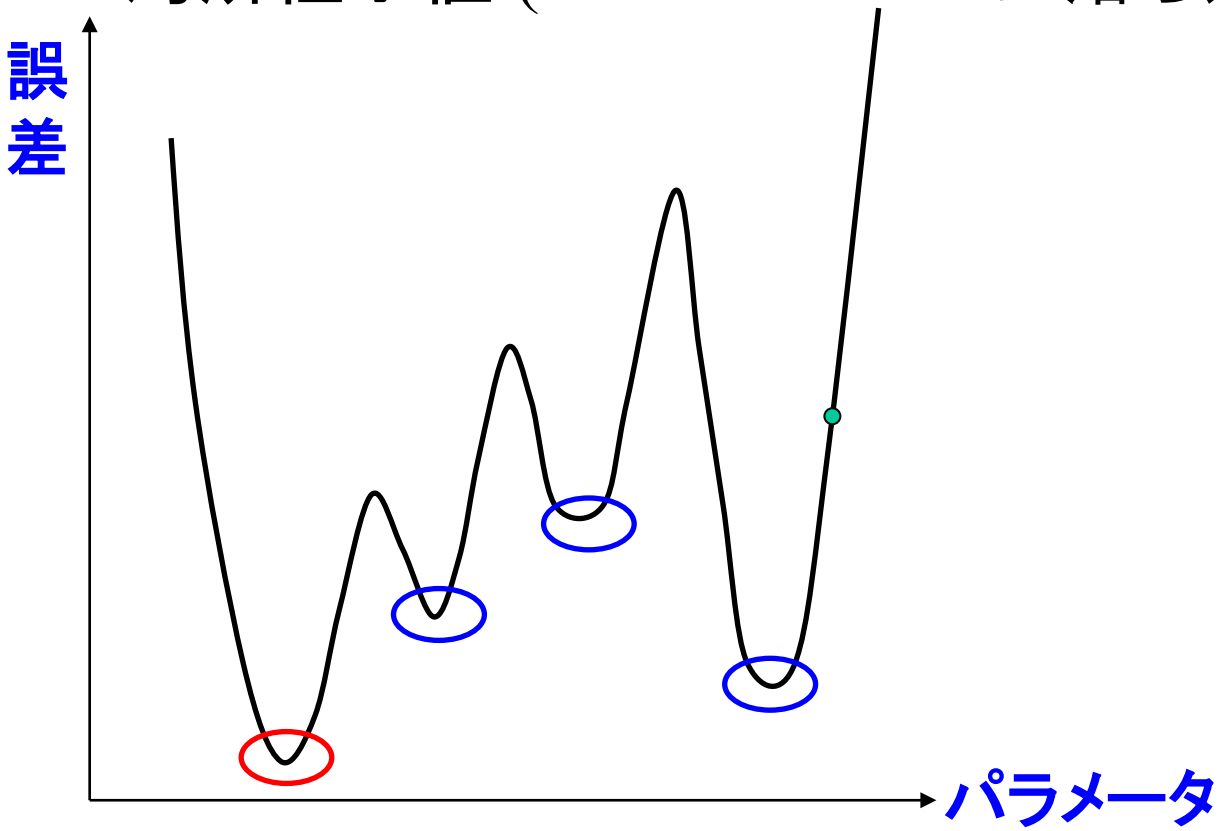


If peaks are overlapped satisfactory, can be converged

非線形 (多値) 方程式の注意

- 解が複数ある場合も
- ほとんどの場合、1度の計算で最適解を求めることは無理
  - 収束したことを確認する
  - 大域最小値を求める

⇔局所極小値 (local minimumに落ち込む)



非線形最適化アルゴルリズムの傾向

	A	B
収束速度	×	○
収束安定性	○	×
安定収束範囲	○	×
使い方	第一段階	第二段階

A: 単体 (Simplex) 法

A,B: 共役勾配法 (Conjugate Gradient: CG)

B: 最急降下法 (Steepest Descent: SD)

B: Newton-Raphson法　・準Newton法

- **Davidson-Fletcher-Powell (DFP)**
- **Broyden-Fletcher-Goldfarb-Shanno (BFGS)**