PHYSBO

リリース ***1.0.1***

PHYSBO developers

2021 年 **05** 月 **24** 日

Contents:

第 1 章

はじめに

1.1 PHYSBO とは

PHYSBO(optimization tool for PHYSics based on Bayesian Optimization) は、高速でスケーラブルなベイズ最適化 (Bayesian optimization) のための Python ライブラリです。COMBO(COMmon Baysian Optimization) をもとに、主に物性分野の研究者をターゲットに開発されました。物理、化学、材料分野において、データ駆動的な実験計画アルゴリズムによって科学的発見を加速する、という試みが多く行われています。ベイズ最適化は、このような科学的発見を加速するために有効なツールです。ベイズ最適化は、複雑なシミュレーションや、実世界における実験タスクなど、目的関数値（特性値など）の評価に大きなコストがかかるような場合に利用できる手法です。つまり、「できるだけ少ない実験・シミュレーション回数でより良い目的関数値（材料特性など）を持つパラメータ（材料の組成、構造、プロセスやシミュレーションパラメータなど）を見つけ出す」ことが、ベイズ最適化によって解かれる問題です。ベイズ最適化では、探索するパラメータの候補をあらかじめリストアップし、候補の中から目的関数値が最大と考えられる候補を機械学習（ガウス過程回帰を利用）による予測をうまく利用することで選定します。その候補に対して実験・シミュレーションを行い目的関数値を評価します。機械学習による選定・実験シミュレーションによる評価を繰り返すことにより、少ない回数での最適化が可能となります。一方で、一般的にベイズ最適化は計算コストが高く、scikit-learn 等のスタンダードな実装では、多くのデータを扱うことが困難です。PHYSBO では以下の特徴により、高いスケーラビリティを実現しています。

- Thompson Sampling
- random feature map
- one-rank Cholesky update
- automatic hyperparameter tuning

技術的な詳細については、こちらの文献 を参照して下さい。

1.2 PHYSBO の引用

PHYSBO を引用する際には、以下の文献を引用してください、

Tsuyoshi Ueno, Trevor David Rhone, Zhufeng Hou, Teruyasu Mizoguchi and Koji Tsuda, COMBO: An Efficient Bayesian Optimization Library for Materials Science, Materials Discovery 4, 18-21 (2016). Available from https://doi.org/10.1016/j.md.2016.04.001

Bibtex は以下の通りです。

```
@article{Ueno2016,
title = "COMBO: An Efficient Bayesian Optimization Library for Materials Science ",
journal = "Materials Discovery",
volume = "4",
pages = "18-21",
year = "2016",
doi = "http://dx.doi.org/10.1016/j.md.2016.04.001",
url = "http://www.sciencedirect.com/science/article/pii/S2352924516300035",
author = "Tsuyoshi Ueno and Trevor David Rhone and Zhufeng Hou and Teruyasu Mizoguchi␣
↪and Koji Tsuda",
}
```

1.3 主な開発者

- ver. 1.0-
  - 田村 亮 (物質・材料研究機構 国際ナノアーキテクトニクス研究拠点)
  - 寺山 慧 (横浜市立大学大学院 生命医科学研究科)
  - 津田 宏治 (東京大学大学院 新領域創成科学研究科)
  - 植野 剛 (株式会社 Magne-Max Capital Management)
  - 本山 裕一 (東京大学 物性研究所)
  - 吉見 一慶 (東京大学 物性研究所)
  - 川島 直輝 (東京大学 物性研究所)
- ver. 0.1-0.3
  - 田村 亮 (物質・材料研究機構 国際ナノアーキテクトニクス研究拠点)
  - 寺山 慧 (横浜市立大学大学院 生命医科学研究科)
  - 津田 宏治 (東京大学大学院 新領域創成科学研究科)

- 本山 裕一 (東京大学 物性研究所)
- 吉見 一慶 (東京大学 物性研究所)
- 川島 直輝 (東京大学 物性研究所)

1.4 ライセンス

本ソフトウェアのプログラムパッケージおよびソースコード一式は GNU General Public License version 3（GPL v3）に準じて配布されています。

本ソフトウェアは 2020 年度 東京大学物性研究所 ソフトウェア高度化プロジェクトの支援を受け開発されました。

第 2 章

基本的な使用方法

2.1 インストール

2.1.1 実行環境・必要なパッケージ

PHYSBO の実行環境・必要なパッケージは以下の通りです。

- Python >= 3.6
- numpy
- scipy

2.1.2 ダウンロード・インストール

- `PyPI` からのインストール（推奨）

```
$ pip3 install physbo
```

  - NumPy などの依存パッケージも同時にインストールされます。
  - `--user` オプションを追加するとユーザのホームディレクトリ以下にインストールされます。

```
$ pip3 install --user physbo
```

- ソースコードからのインストール（開発者向け）

1. 本体のダウンロード

   ソースファイルをダウンロードするか、以下のように github レポジトリをクローンしてください。

```
$ git clone https://github.com/issp-center-dev/PHYSBO
```

2. pip を 19.0 以上に更新

```
$ pip3 install -U pip

- ここで ``pip3`` が入っていない場合には ``python3 -m ensurepip`` でインストール可能です
```

3. インストール

```
$ cd PHYSBO
$ pip3 install --user ./
```

2.1.3 アンインストール

1. 以下のコマンドを実行します。

```
$ pip uninstall physbo
```

2.2 PHYSBO の基本構造

PHYSBO は以下のような構成になっています (第 2 階層まで表示)。

各モジュールは以下のような構成で作成されています。

- `blm` :Baysean linear model に関するモジュール
- `gp` :Gaussian Process に関するモジュール
- `opt` :最適化に関するモジュール
- `search` :最適解を探索するためのモジュール
- `predictor.py` :predictor の抽象クラス
- `variable.py` :physbo で用いる変数関連について定義されたクラス
- `misc` : その他 (探索空間を正規化するためのモジュールなど)

各モジュールの詳細については API リファレンスを参考にしてください。

2.3 計算の流れ

ベイズ最適化は、複雑なシミュレーションや、実世界における実験タスクなど、目的関数の評価に大きなコストがかかるような最適化問題に適しています。PHYSBO では以下の手順により最適化を実行します (それぞれの詳細はチュートリアルおよび API リファレンスを参考にしてください)。

1. 探索空間の定義

   N: 探索候補の数 , d: 入力パラメータの次元数 とした時、探索候補である各パラメータセット (d 次元のベクトル) を定義します。パラメータセットは全ての候補をリストアップしておく必要があります。

2. simulator の定義

   上で定義した探索候補に対して、各探索候補の目的関数値（材料特性値など最適化したい値）を与える simulator を定義します。PHYSBO では、最適化の方向は「目的関数の最大化」になります。そのため，目的関数を最小化したい場合、simulator から返す値にマイナスをかけることで実行できます。

3. 最適化の実行

   最初に、最適化の policy をセットします (探索空間はこの段階で引数として policy に渡されます)。最適化方法は、以下の 2 種類から選択します。

   - *random_search*
   - *bayes_search*

   random_search では、探索空間からランダムにパラメータを選び、その中で最大となる目的関数を探します。ベイズ最適化を行うための前処理として初期パラメータ群を用意するために使用します。*bayes_search* は、ベイズ最適化を行います。ベイズ最適化での score: 獲得関数 (acquisition function) の種類は、以下のいずれかから指定します。

   - TS (Thompson Sampling): 学習されたガウス過程の事後確率分布から回帰関数を 1 つサンプリングし、それを用いた予測が最大となる点を候補として選択します。
   - EI (Expected Improvement): ガウス過程による予測値と現状での最大値との差の期待値が最大となる点を候補として選択します。
   - PI (Probability of Improvement): 現状での最大値を超える確率が最大となる点を候補として選択します。

   ガウス過程に関する詳細については アルゴリズム に記載してあります。その他、各手法の詳細については、こちらの文献 およびその参考文献を参照して下さい。

   これらのメソッドに先ほど定義した simulator と探索ステップ数を指定すると、探索ステップ数だけ以下のループが回ります。

   i). パラメータ候補の中から次に実行するパラメータを選択

   ii). 選択されたパラメータで simulator を実行

i) で返されるパラメータはデフォルトでは 1 つですが、1 ステップで複数のパラメータを返すことも可能です。詳しくはチュートリアルの「複数候補を一度に探索する」の項目を参照してください。また、上記のループを PHYSBO の中で回すのではなく、i) と ii) を別個に外部から制御することも可能です。つまり、PHYSBO から次に実行するパラメータを提案し、その目的関数値を PHYBO の外部で何らかの形で評価し（例えば、数値計算ではなく、実験による評価など）、それを PHYSBO の外部で何らかの形で提案し、評価値を PHYSBO に登録する、という手順が可能です。詳しくは、チュートリアルの「インタラクティブに実行する」の項目を参照してください。

4. 結果の確認

探索結果 res は history クラスのオブジェクト (physbo.search.discrete.results.history) として返されます。以下より探索結果を参照します。

- res.fx : simulator (目的関数) の評価値の履歴。
- res.chosen_actions: simulator を評価したときの action ID(パラメータ) の履歴。
- fbest, best_action= res.export_all_sequence_best_fx(): simulator を評価した全タイミングにおけるベスト値とその action ID(パラメータ) の履歴。
- res.total_num_search: simulator のトータル評価数。

また、探索結果は save メソッドにより外部ファイルに保存でき、load メソッドを用いて出力した結果をロードすることができます。使用方法の詳細はチュートリアルをご覧ください。

第 3 章

チュートリアル

ここでは、PHYSBO のチュートリアルを通してその使用方法を紹介します。

3.1 PHYSBO の基本

3.1.1 はじめに

本チュートリアルでは例として、一次元の関数の最小値を求める例題を解きます。 はじめに、PHYSBO をインポートします。

[1]:
```
import physbo
```

3.1.2 探索候補データの準備

最初に関数を探索する空間を定義します。 以下の例では、探索空間 X を x_min = -2.0 から x_max = 2.0 まで window_num=10001 分割で刻んだグリッドで定義しています。 なお、X は window_num x d の ndarray 形式にする必要があります (d は次元数、この場合は 1 次元)。そのため、reshape を行って変形しています。

[2]:
```
#In
import numpy as np
import scipy
import physbo
import itertools

#In
#Create candidate
window_num=10001
x_max = 2.0
x_min = -2.0
```

(次のページに続く)

(前のページからの続き)

```
X = np.linspace(x_min,x_max,window_num).reshape(window_num, 1)
```

3.1.3 simulator クラスの定義

目的関数を定義するための simulator クラスをここで定義します。

今回は f(x) = 3 x^4 + 4 x ^3 + 1.0 が最小となる x を探索するという問題設定にしています (答えは x=-1.0)。

simulator クラスでは、__call__関数を定義します (初期変数などがある場合は__init__を定義します)。 action は探索空間の中から取り出すグリッドの index 番号を示しており、複数の候補を一度に計算できるように一般的に ndarray の形式を取っています。 今回は一つの候補のみを毎回計算するため、action_idx=action[0] として X から候補点を一つ選んでいます。 **PHYSBO では目的関数値が最大となる**ものを求める仕様になっているため、候補点での f(x) の値に-1 をかけたものを返しています。

[3]:
```
# Declare the class for calling the simulator.
class simulator:

    def __call__(self, action):
        action_idx = action[0]
        x = X[action_idx][0]
        fx = 3.0*x**4 + 4.0*x**3 + 1.0
        fx_list.append(fx)
        x_list.append(X[action_idx][0])

        print ("**********************")
        print ("Present optimum interactions")

        print ("x_opt=", x_list[np.argmin(np.array(fx_list))])

        return -fx
```

3.1.4 最適化の実行

policy のセット

まず、最適化の policy をセットします。

test_X に探索候補の行列 (numpy.array) を指定します。

[4]:
```
# policy のセット
policy = physbo.search.discrete.policy(test_X=X)
```

(次のページに続く)

(前のページからの続き)

```
# シード値のセット
policy.set_seed(0)
```

policy をセットした段階では、まだ最適化は行われません。 policy に対して以下のメソッドを実行することで、最適化を行います。

- random_search
- bayes_search

これらのメソッドに先ほど定義した simulator と探索ステップ数を指定すると、探索ステップ数だけ以下のループが回ります。

i) パラメータ候補の中から次に実行するパラメータを選択

ii) 選択されたパラメータで simulator を実行

iii) で返されるパラメータはデフォルトでは 1 つですが、1 ステップで複数のパラメータを返すことも可能です。 詳しくは「複数候補を一度に探索する」の項目を参照してください。

また、上記のループを PHYSBO の中で回すのではなく、i) と ii) を別個に外部から制御することも可能です。つまり、PHYSBO から次に実行するパラメータを提案し、その目的関数値を PHYBO の外部で何らかの形で評価し（例えば、数値計算ではなく、実験による評価など）、それを PHYSBO の外部で何らかの形で提案し、評価値を PHYSBO に登録する、という手順が可能です。詳しくは、チュートリアルの「インタラクティブに実行する」の項目を参照してください。

ランダムサーチ

まず初めに、ランダムサーチを行ってみましょう。

ベイズ最適化の実行には、目的関数値が 2 つ以上求まっている必要があるため（初期に必要なデータ数は、最適化したい問題、パラメータの次元 d に依存して変わります）、まずランダムサーチを実行します。

引数

- max_num_probes: 探索ステップ数
- simulator: 目的関数のシミュレータ (simulator クラスのオブジェクト)

```
[ ]: fx_list=[]
x_list = []
res = policy.random_search(max_num_probes=20, simulator=simulator())
```

実行すると、各ステップの目的関数値とその action ID、現在までのベスト値とその action ID に関する情報が以下のように出力されます。

```
0020-th step: f(x) = -19.075990 (action=8288)
   current best f(x) = -0.150313 (best action=2949)
```

ベイズ最適化

続いて、ベイズ最適化を以下のように実行します。

引数

- `max_num_probes`: 探索ステップ数
- `simulator`: 目的関数のシミュレータ (simulator クラスのオブジェクト)
- `score`: 獲得関数 (acquisition function) のタイプ。以下のいずれかを指定します。
  - TS (Thompson Sampling)
  - EI (Expected Improvement)
  - PI (Probability of Improvement)
- `interval`: 指定したインターバルごとに、ハイパーパラメータを学習します。負の値を指定すると、ハイパーパラメータの学習は行われません。0 を指定すると、ハイパーパラメータの学習は最初のステップでのみ行われます。
- `num_rand_basis`: 基底関数の数。0 を指定すると、Bayesian linear model を利用しない通常のガウス過程が使用されます。

```
[ ]: res = policy.bayes_search(max_num_probes=50, simulator=simulator(), score='TS',
                                                    interval=0, num_rand_basis=500)
```

3.1.5 結果の確認

探索結果 res は history クラスのオブジェクト (`physbo.search.discrete.results.history`) として返されます。
以下より探索結果を参照します。

- `res.fx` : simulator (目的関数) の評価値の履歴。
- `res.chosen_actions`: simulator を評価したときの action ID (パラメータ) の履歴。
- `fbest, best_action= res.export_all_sequence_best_fx()`: simulator を評価した全タイミングにおけるベスト値とその action ID (パラメータ) の履歴。
- `res.total_num_search`: simulator のトータル評価数。

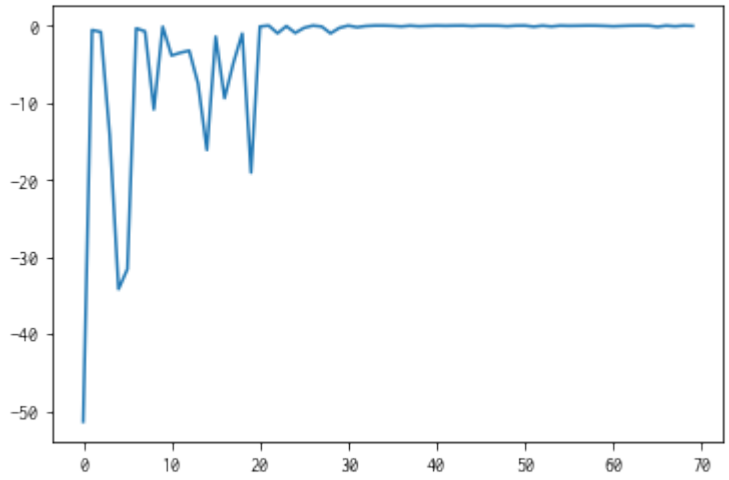
各ステップでの目的関数値と、ベスト値の推移をプロットしてみましょう。

`res.fx, best_fx` はそれぞれ `res.total_num_search` までの範囲を指定します。

[7]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

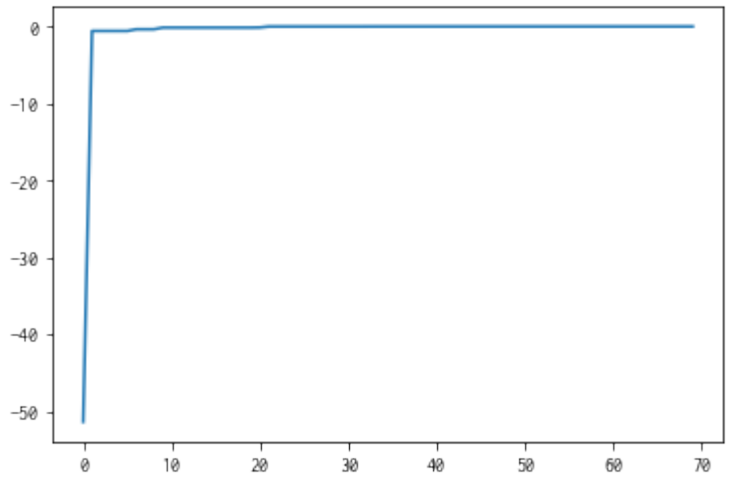
[8]:
```
plt.plot(res.fx[0:res.total_num_search])
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7ff618e0d820>]
```



[9]:
```
best_fx, best_action = res.export_all_sequence_best_fx()
plt.plot(best_fx)
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7ff618f0ff70>]
```



3.1.6 結果のシリアライズ

探索結果は save メソッドにより外部ファイルに保存できます。

[10]:
```
res.save('search_result.npz')
```

[11]:
```
del res
```

保存した結果ファイルは以下のようにロードすることができます。

[12]:
```
res = physbo.search.discrete.results.history()
res.load('search_result.npz')
```

最後に、一番よいスコアを持つ候補は以下のようにして表示することができます。正しい解 x=-1 に行き着いていることがわかります。

[13]:
```
print(X[int(best_action[-1])])
```

```
[-1.002]
```

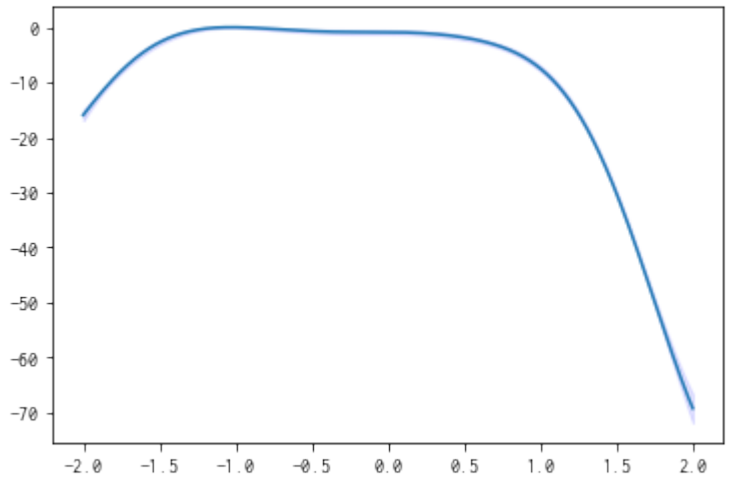
3.1.7 回帰

get_post_fmean, get_post_fcov メソッドでガウス過程（事後分布）の期待値と分散を計算可能です。

[14]:
```
mean = policy.get_post_fmean(X)
var = policy.get_post_fcov(X)
std = np.sqrt(var)

x = X[:,0]
fig, ax = plt.subplots()
ax.plot(x, mean)
ax.fill_between(x, (mean-std), (mean+std), color='b', alpha=.1)
```

[14]:
```
<matplotlib.collections.PolyCollection at 0x7ff608adc3d0>
```

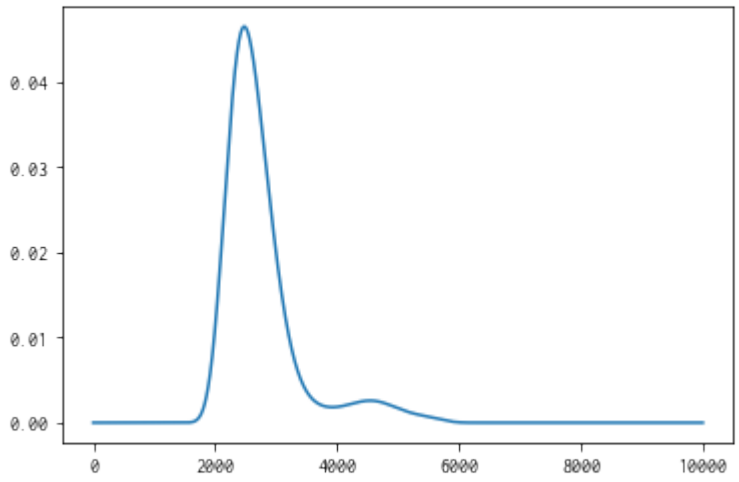


3.1.8 獲得関数

get_score メソッドで獲得関数を計算可能です。

[15]:
```
score = policy.get_score(mode="EI", xs=X)
plt.plot(score)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7ff608b3d070>]
```



3.1.9 並列化

PHYSBO は全候補点に対する獲得関数の計算を MPI を用いて並列化出来ます。 MPI 並列には `mpi4py` を用います。

並列化を有効化するには、 `policy` のコンストラクタのキーワード引数 `comm` に MPI コミュニケータ、たとえば `MPI.COMM_WORLD` を渡してください。

[16]:
```
# from mpi4py import MPI
# policy = physbo.search.discrete.policy(X=test_X, comm=MPI.COMM_WORLD)
```

3.2 ガウス過程

PHYSBO ではガウス過程回帰を実行しながらベイズ最適化を行なっています。

そのため、学習データが与えられた際にガウス過程回帰を実行することもでき、学習済みモデルを利用したテストデータの予測も行うことができます。

ここでは、その手順について紹介します。

3.2.1 探索候補データの準備

本チュートリアルでは例として、Cu の安定した界面構造の探索問題を扱います。 目的関数の評価にあたる構造緩和計算には、実際には 1 回あたり数時間といったオーダーの時間を要しますが、本チュートリアルでは既に評価済みの値を使用します。問題設定については、以下の文献を参照してください。 S. Kiyohara, H. Oda, K. Tsuda and T. Mizoguchi, “Acceleration of stable interface structure searching using a kriging approach”, Jpn. J. Appl. Phys. 55, 045502 (2016).

データセットファイル s5-210.csv を `data` ディレクトリ以下に保存し、次のように読み出します。

[1]:
```
import physbo

import numpy as np

def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering( X )
```

3.2.2 学習データの定義

対象データのうち、ランダムに選んだ 1 割をトレーニングデータとして利用し、別のランダムに選んだ 1 割をテストデータとして利用します。

[2]:
```
N = len(t)
Ntrain = int(N*0.1)
Ntest = min(int(N*0.1), N-Ntrain)

id_all   = np.random.choice(N, N, replace=False)
id_train  = id_all[0:Ntrain]
id_test = id_all[Ntrain:Ntrain+Ntest]

X_train = X[id_train]
X_test = X[id_test]

t_train = t[id_train]
t_test = t[id_test]

print("Ntrain =", Ntrain)
print("Ntest =", Ntest)
```

```
Ntrain = 1798
Ntest = 1798
```

3.2.3 ガウス過程の学習と予測

以下のプロセスでガウス過程を学習し、テストデータの予測を行います。

1. ガウス過程のモデルを生成します。
2. X_train（学習データのパラメータ）, t_train（学習データの目的関数値）を用いてモデルを学習します。
3. 学習されたモデルを用いてテストデータ (X_test) に対する予測を実行します。

共分散の定義 (ガウシアン)

[3]:
```
cov = physbo.gp.cov.gauss( X_train.shape[1],ard = False )
```

平均の定義

[4]:
```
mean = physbo.gp.mean.const()
```

尤度関数の定義 (ガウシアン)

[5]:
```
lik = physbo.gp.lik.gauss()
```

ガウス過程モデルの生成

[6]:
```
gp = physbo.gp.model(lik=lik,mean=mean,cov=cov)
config = physbo.misc.set_config()
```

ガウス過程モデルを学習

[7]:
```
gp.fit(X_train, t_train, config)
```

```
Start the initial hyper parameter searching ...
Done

Start the hyper parameter learning ...
0 -th epoch marginal likelihood 16051.69452976001
50 -th epoch marginal likelihood 4551.39626443153
100 -th epoch marginal likelihood 2141.377872729846
150 -th epoch marginal likelihood 595.8361411907399
200 -th epoch marginal likelihood -373.23922309413774
250 -th epoch marginal likelihood -929.2472009209496
300 -th epoch marginal likelihood -1273.8727959019732
350 -th epoch marginal likelihood -1413.2553901394206
400 -th epoch marginal likelihood -1477.3889625983586
450 -th epoch marginal likelihood -1525.339082571432
500 -th epoch marginal likelihood -1539.787541261617
Done
```

学習されたガウス過程におけるパラメタを出力

[8]:
```
gp.print_params()
```

```
likelihood parameter =   [-2.57036368]
mean parameter in GP prior:  [-1.0654197]
covariance parameter in GP prior:  [-0.59460765 -2.4232173 ]
```

テストデータの平均値（予測値）および分散を計算

[9]:
```
gp.prepare(X_train, t_train)
fmean = gp.get_post_fmean(X_train, X_test)
fcov = gp.get_post_fcov(X_train, X_test)
```

予測の結果

[10]:
```
fmean
```

```
[10]: array([-1.01865696, -0.98356729, -0.97797072, ..., -0.99978278,
             -0.98145533, -0.9956255 ])
```

分散の結果

```
[11]: fcov
```

```
[11]: array([0.00046688, 0.0010622 , 0.0006136 , ..., 0.00043492, 0.0005969 ,
             0.00053435])
```

予測値の平均二乗誤差の出力

```
[12]: np.mean((fmean-t_test)**2)
```

```
[12]: 0.004179032574484333
```

3.2.4 訓練済みモデルによる予測

学習済みモデルのパラメタを gp_params として読み出し、これを用いた予測を行います。

gp_params および学習データ (X_train, t_train) を記憶しておくことで、訓練済みモデルによる予測が可能となります。

学習されたパラメタを準備 (学習の直後に行う必要あり)

```
[13]: #学習したパラメタを 1 次元配列として準備
gp_params =  np.append(np.append(gp.lik.params, gp.prior.mean.params), gp.prior.cov.
↪params)

gp_params
```

```
[13]: array([-2.57036368, -1.0654197 , -0.59460765, -2.4232173 ])
```

学習に利用したモデルと同様のモデルを gp として準備

```
[14]: #共分散の定義 （ガウシアン）
cov = physbo.gp.cov.gauss( X_train.shape[1],ard = False )

#平均の定義
mean = physbo.gp.mean.const()

#尤度関数の定義 （ガウシアン）
lik = physbo.gp.lik.gauss()

#ガウス過程モデルの生成
gp = physbo.gp.model(lik=lik,mean=mean,cov=cov)
```

学習済みのパラメタをモデルに入力し予測を実行

[15]:
```
#学習済みのパラメタをガウス過程に入力
gp.set_params(gp_params)


#テストデータの平均値（予測値）および分散を計算
gp.prepare(X_train, t_train)
fmean = gp.get_post_fmean(X_train, X_test)
fcov = gp.get_post_fcov(X_train, X_test)
```

予測の結果

[16]:
```
fmean
```

[16]:
```
array([-1.01865696, -0.98356729, -0.97797072, ..., -0.99978278,
       -0.98145533, -0.9956255 ])
```

分散の結果

[17]:
```
fcov
```

[17]:
```
array([0.00046688, 0.0010622 , 0.0006136 , ..., 0.00043492, 0.0005969 ,
       0.00053435])
```

予測値の平均二乗誤差の出力

[18]:
```
np.mean((fmean-t_test)**2)
```

[18]:
```
0.004179032574484333
```

[ ]:
```

```

3.3 インタラクティブに実行する

以下の流れで、PHYSBO をインタラクティブに実行することができます。

1. PHYSBO から次に実行するパラメータを得ます。
2. PHYSBO の外部で評価値を得ます。
3. 評価値を PHYSBO に登録します。

例えば、以下の様な場合に適しています。

- 人手による実験を行い、その評価値を PHYSBO に与えたい。
- simulator の実行を別プロセスで行うなど、柔軟に実行制御を行いたい。

3.3.1 探索候補データの準備

これまでのチュートリアルと同様、データセットファイル s5-210.csv を data ディレクトリ以下に保存し、次のように読み出します。

[1]:
```
import physbo

import numpy as np


def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.3.2 simulator の定義

[2]:
```
class simulator:
    def __init__( self ):
        _, self.t = load_data()

    def __call__( self, action ):
        return self.t[action]
```

3.3.3 最適化の実行

[3]:
```
# policy のセット
policy = physbo.search.discrete.policy(test_X=X)

# シード値のセット
policy.set_seed( 0 )
```

各探索ステップでは以下の処理を行っています。

1. `max_num_probes=1, simulator=None` として random_search または bayes_search を実行して action ID (パラメータ) を得る。
2. `t = simulator(actions)` により評価値 (の array) を得る。
3. `policy.write(actions, t)` により action ID (パラメータ) に対する評価値を登録する。

4. physbo.search.utility.show_search_results により履歴を表示する。

以下では、ランダムサンプリングを 2 回（1st, and 2nd steps）、ベイズ最適化による提案を 2 回（3rd, and 4th steps）を行います。

[ ]:
```
simulator = simulator()

''' 1st step (random sampling) '''
actions = policy.random_search(max_num_probes=1, simulator=None)
t  = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 2nd step (random sampling) '''
actions = policy.random_search(max_num_probes=1, simulator=None)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 3rd step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0,
  num_rand_basis = 5000)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

''' 4-th step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0,
  num_rand_basis = 5000)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)
```

3.3.4 中断と再開

以下の predictor, training, history を外部ファイルに保存することで、最適化プロセスを中断し、途中から再開することができます。

- predictor: 目的関数の予測モデル
- training: predictor の学習に用いるデータ (physbo.variable オブジェクト)
- history: 最適化実行の履歴 (physbo.search.discrete.results.history オブジェクト)

[5]:
```
policy.save(file_history='history.npz', file_training='training.npz', file_predictor=
↪'predictor.dump')
```

[ ]:
```
# policy を削除
del policy

# 保存した policy をロード
policy = physbo.search.discrete.policy(test_X=X)
policy.load(file_history='history.npz', file_training='training.npz', file_predictor=
↪'predictor.dump')

''' 5-th step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1, simulator=None, score='EI', interval=0,
  num_rand_basis = 5000)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)

# predictor と training を個別に指定することも可
''' 6-th step (bayesian optimization) '''
actions = policy.bayes_search(max_num_probes=1,
                                          predictor=policy.predictor,␣
↪training=policy.training,
                                          simulator=None, score='EI', interval=0, ␣
↪num_rand_basis = 5000)
t = simulator(actions)
policy.write(actions, t)
physbo.search.utility.show_search_results(policy.history, 10)
```

[ ]:
```

```

3.4 既存の計算結果を読み込んで実行する

以下の流れで、既存の action ID(パラメータ) とその評価値を読み込み、PHYSBO を実行することができます。

1. 外部ファイルを読み込み、既存の action ID(パラメータ) と評価値を読み込む。
2. action ID(パラメータ) と評価値を PHYSBO に登録する。
3. PHYSBO から次に実行するパラメータを得る。

時間制限の関係上、PHYSBO をずっと開いたままにできないため、インタラクティブに実行できないといった場合に、利用することができます。

3.4.1 探索候補データの準備

これまでのチュートリアルと同様、データセットファイル s5-210.csv を data ディレクトリ以下に保存し、次のように読み出します。

[1]:
```
import physbo

import numpy as np


def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.4.2 事前に計算したデータの用意

上述の load_data 関数では全ての X と t が格納されています。ここでは事前に計算したとして、actoin ID のリストをランダムに 20 個取得し、その評価値を得ます。

[2]:
```
import random
random.seed(0)
calculated_ids = random.sample(range(t.size), 20)
print(calculated_ids)
t_initial = t[calculated_ids]
```

```
[12623, 13781, 1326, 8484, 16753, 15922, 13268, 9938, 15617, 11732, 7157, 16537, 4563,␣
↪9235, 4579, 3107, 8208, 17451, 4815, 10162]
```

3.4.3 action ID(パラメータ) と評価値を PHYSBO に登録

policy の初期変数　initial_data に calculated_ids と t[calculated_ids] をリストとして登録します。

[3]:
```
# policy のセット
policy = physbo.search.discrete.policy(test_X=X, initial_data=[calculated_ids, t_
↪initial])

# シード値のセット
policy.set_seed( 0 )
```

3.4.4 PHYSBO から次に実行するパラメータを取得

ベイズ最適化を行い、次の候補点を得ます。

[4]:
```
actions = policy.bayes_search(max_num_probes=1, simulator=None, score="TS", interval=0,
  num_rand_basis = 5000)
print(actions, X[actions])
```

```
Start the initial hyper parameter searching ...
Done

Start the hyper parameter learning ...
0 -th epoch marginal likelihood -20.09302189053099
50 -th epoch marginal likelihood -23.11964735598211
100 -th epoch marginal likelihood -24.83020118385076
150 -th epoch marginal likelihood -25.817906570042602
200 -th epoch marginal likelihood -26.42342027124426
250 -th epoch marginal likelihood -26.822598600211865
300 -th epoch marginal likelihood -27.10872736571494
350 -th epoch marginal likelihood -27.331572599126865
400 -th epoch marginal likelihood -27.517235815448124
450 -th epoch marginal likelihood -27.67892333553869
500 -th epoch marginal likelihood -27.82299469827059
Done

[73] [[-1.6680279  -1.46385011  1.68585446]]
```

得られた候補点について外部で計算を行い、ファイルに actions とそのスコアを登録する。再びファイルを読み込み、ベイズ最適化を実行し次の候補点を得るというプロセスを繰り返すことで、ベイズ最適化を進めることができます。

3.5 複数候補を一度に探索する

1 回の探索ステップで、2 つ以上の候補を一度に評価する場合のチュートリアルです。

3.5.1 探索候補データの準備

これまでのチュートリアルと同様、データセットファイル s5-210.csv を data ディレクトリ以下に保存し、次のように読み出します。

[1]:
```
import physbo

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

(次のページに続く)

(前のページからの続き)

```
def load_data():
    A =  np.asarray(np.loadtxt('data/s5-210.csv',skiprows=1, delimiter=',') )
    X = A[:,0:3]
    t  = -A[:,3]
    return X, t

X, t = load_data()
X = physbo.misc.centering(X)
```

3.5.2 simulator の定義

後述の num_search_each_probe を 2 以上にした場合、action には action ID の array が入力されます。
各 action ID に対応した評価値のリストを返すように定義してください。

基本チュートリアルと simulator の定義は同じですが、 t は numpy.array であり、action に array が入力されると self.t[action] も array になる点に留意してください。

[2]:
```
class simulator:
    def __init__( self ):
        _, self.t = load_data()

    def __call__( self, action ):
        return self.t[action]
```

simulator の実行例

[3]:
```
sim = simulator()
sim([1,12,123])
```

[3]:
```
array([-1.01487066, -1.22884748, -1.05572838])
```

3.5.3 最適化の実行

[4]:
```
# policy のセット
policy = physbo.search.discrete.policy(test_X=X)

# シード値のセット
policy.set_seed( 0 )
```

num_search_each_probe によって、各探索ステップにおいて評価する候補数を指定することができます。

下記の実行例だと、ランダムサーチにより 2 × 10 = 20 回、ベイズ最適化により 8 × 10 = 80 回 simulator を評価することになります。

引数

- max_num_probes: 探索ステップ数
- num_search_each_probe: 各探索ステップにおいて評価する候補数

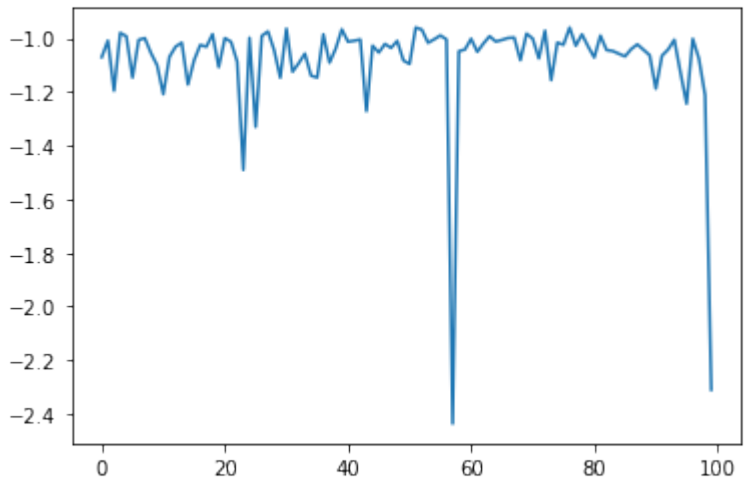
[ ]:
```
res = policy.random_search(max_num_probes=2, num_search_each_probe=10,␣
↪simulator=simulator())

res = policy.bayes_search(max_num_probes=8, num_search_each_probe=10,␣
↪simulator=simulator(), score='EI',
                                                    interval=2, num_rand_basis=100)
```

3.5.4 結果の確認

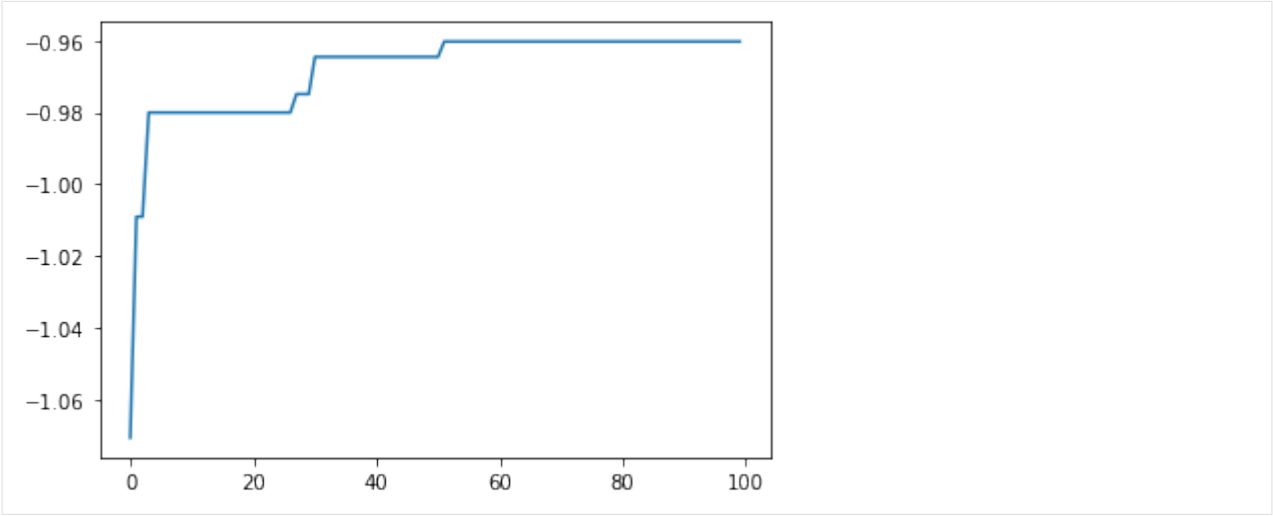
[6]:
```
plt.plot(res.fx[0:res.total_num_search])
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7fe958b7fdc0>]
```



[7]:
```
best_fx, best_action = res.export_all_sequence_best_fx()
plt.plot(best_fx)
```

[7]:
```
[<matplotlib.lines.Line2D at 0x7fe9400e1e50>]
```



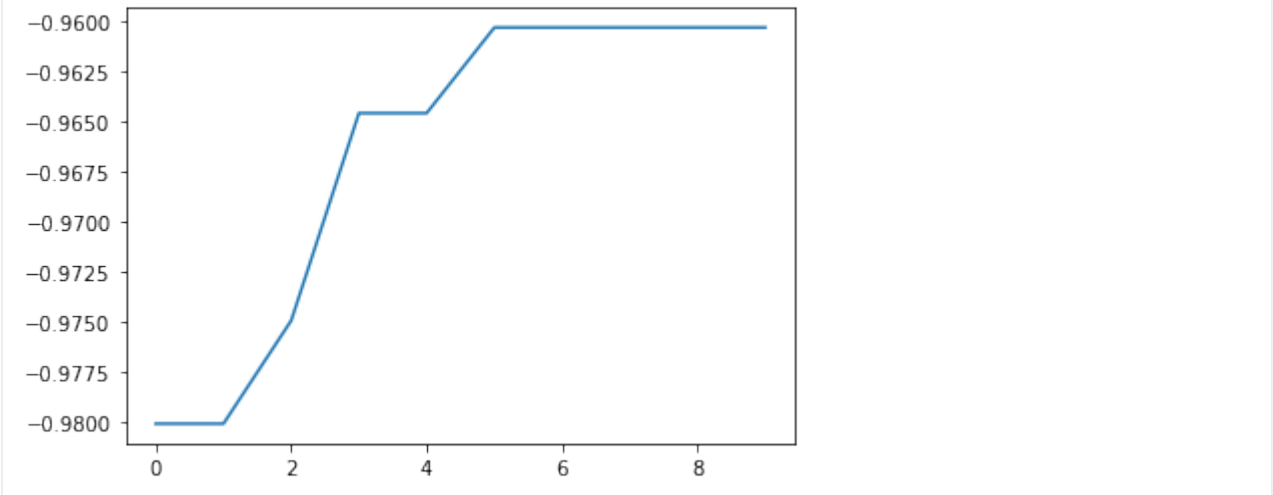
`res.export_sequence_best_fx()` により、各ステップまでに得られたベスト値とその action の履歴 を得られます。

`res.export_all_sequence_best_fx()` との違いは、simulator の評価毎ではなく、探索ステップ毎の情報になるという点です。

(今回の場合は合計ステップ数は 10, 評価数は 100 です)

[8]:
```
best_fx, best_action = res.export_sequence_best_fx()
plt.plot(best_fx)
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7fe9789468b0>]
```



3.6 多目的最適化

最適化したい目的関数が複数 (p 個) ある場合は、多目的最適化をおこないます。
本チュートリアルでは、「解」は目的関数の組 $y = (y_1(x), y_2(x), \ldots, y_p(x))$ を意味することに注意してください。
解の大小関係 $\prec$ を以下のように定義します。

$$y \prec y^{'} \Longleftrightarrow \forall\, i \leq p, y_i \leq y_i^{'} \wedge \exists\, j \leq p, y_j < y_j^{'}$$

（最大化問題における）**パレート解** とは、上記の大小関係の上で、自身よりも大きな解がないような解を指します。
すなわち、任意の目的関数の値を改善しようとした場合、他の目的関数のうちどれかひとつは悪化するような解です。
目的関数間にトレードオフが存在する場合は、パレート解は複数存在するため、それらを効率的に求めることが課題となります。

PHYSBO では、パレート解を効率的に求めるためのベイズ最適化手法を実装しています。

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import physbo
%matplotlib inline
```

3.6.1 テスト関数

本チュートリアルでは、多目的最適化のベンチマーク関数である VLMOP2 を利用します。
目的関数の数は 2 つです。

$$y_1(\vec{x}) = 1 - \exp\left[-\sum_{i=1}^{N}\left(x_i - 1/\sqrt{N}\right)^2\right]$$
$$y_2(\vec{x}) = 1 - \exp\left[-\sum_{i=1}^{N}\left(x_i + 1/\sqrt{N}\right)^2\right]$$

y_1 と y_2 はそれぞれ $x_1 = x_2 = \cdots x_N = 1/\sqrt{N}$ と $x_1 = x_2 = \cdots x_N = -1/\sqrt{N}$ に最小値があり、その値はともに 0 です。また、上界は 1 です。

PHYSBO は最大化問題を仮定するため、-1 を掛けたものをあらためて目的関数とします。

- 参考文献

– Van Veldhuizen, David A. Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. No. AFIT/DS/ENG/99-01. AIR FORCE INST OF TECH WRIGHT-PATTERSONAFB OH SCHOOL OF ENGINEERING, 1999.

[2]:
```
def vlmop2_minus(x):
    n = x.shape[1]
    y1 = 1 - np.exp(-1 * np.sum((x - 1/np.sqrt(n)) ** 2, axis = 1))
    y2 = 1 - np.exp(-1 * np.sum((x + 1/np.sqrt(n)) ** 2, axis = 1))

    return np.c_[-y1, -y2]
```

3.6.2 探索候補データの準備

入力空間 x は 2 次元とし、[-2, 2] × [-2, 2] の上で候補点をグリッド状に生成します。

[3]:
```
import itertools
a = np.linspace(-2,2,101)
test_X = np.array(list(itertools.product(a, a)))
```

[4]:
```
test_X
```

[4]:
```
array([[-2.  , -2.  ],
       [-2.  , -1.96],
       [-2.  , -1.92],
       ...,
       [ 2.  ,  1.92],
       [ 2.  ,  1.96],
       [ 2.  ,  2.  ]])
```

[5]:
```
test_X.shape
```

[5]:
```
(10201, 2)
```

3.6.3 simulator の定義

[6]:
```
class simulator(object):
    def __init__(self, X):
        self.t = vlmop2_minus(X)

    def __call__( self, action):
        return self.t[action]
```

[7]:
```
simu = simulator(test_X)
```

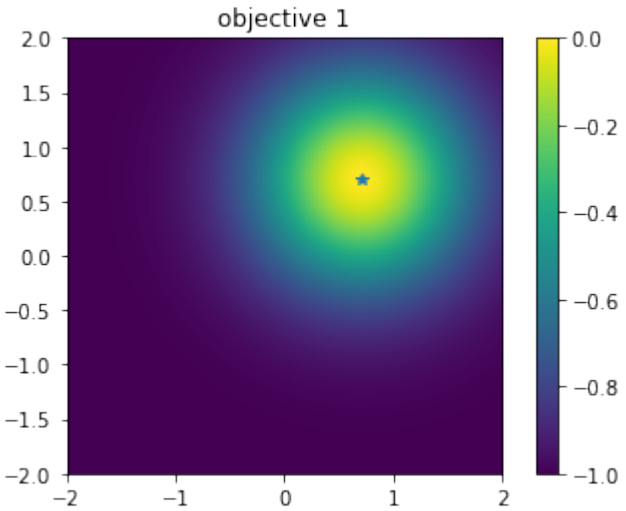
3.6.4 関数のプロット

2 つの目的関数をそれぞれプロットしてみましょう。

1 つ目の目的関数は右上にピークがあり、2 つ目の目的関数は左下にピークがあるようなトレードオフがある状態となっています。（星はピークの位置です）

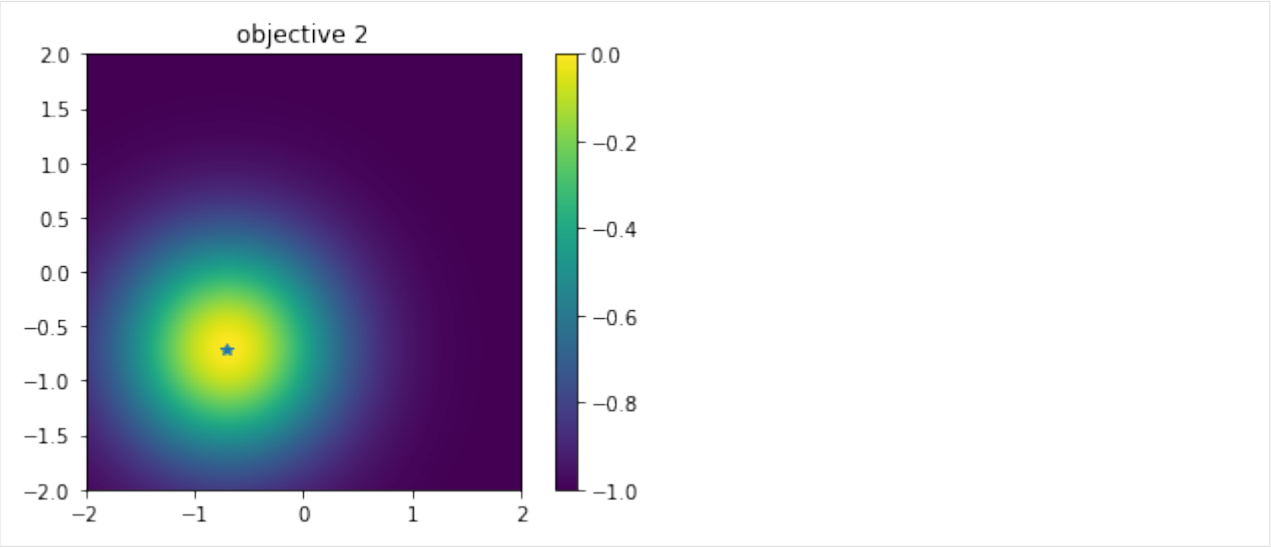
1 つ目の目的関数

[8]:
```
plt.figure()
plt.imshow(simu.t[:,0].reshape((101,101)), vmin=-1.0, vmax=0.0, origin="lower",␣
↪extent=[-2.0, 2.0, -2.0, 2.0])
plt.title("objective 1")
plt.colorbar()
plt.plot([1.0/np.sqrt(2.0)], [1.0/np.sqrt(2.0)], '*')
plt.show()
```



2 つ目の目的関数

[9]:
```
# plot objective 2
plt.figure()
plt.imshow(simu.t[:,1].reshape((101,101)), vmin=-1.0, vmax=0.0, origin="lower",␣
↪extent=[-2.0, 2.0, -2.0, 2.0])
plt.title("objective 2")
plt.colorbar()
plt.plot([-1.0/np.sqrt(2.0)], [-1.0/np.sqrt(2.0)], '*')
plt.show()
```



3.6.5 最適化の実行

policy のセット

多目的最適化用の `physbo.search.discrete_multi.policy` を利用します。

`num_objectives` に目的関数の数を指定してください。

[10]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)
```

通常の `physbo.search.discrete.policy` (目的関数が 1 つの場合）と同じく、`random_search` または `bayes_search` メソッドを呼ぶことで最適化を行います。

基本的な API や利用方法は `discrete.policy` とおおよそ共通しています。

ランダムサーチ

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

res_random = policy.random_search(max_num_probes=50, simulator=simu)
```

目的関数の評価値 (の array) とそのときの action ID が表示されます。

また、パレート解集合 (Pareto set) が更新されたときにメッセージを表示します。

Pareto set が更新された際に中身を表示したい場合は、`disp_pareto_set=True` と指定します。

Pareto set は 1 つ目の目的関数値の昇順でソートされています。また、steps はパレート解が追加された際のステップ数を示しています。

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)
res_random = policy.random_search(max_num_probes=50, simulator=simu, disp_pareto_
↪set=True)
```

結果の確認

評価値の履歴

[ ]:
```
res_random.fx[0:res_random.num_runs]
```

パレート解の取得

[14]:
```
front, front_num = res_random.export_pareto_front()
front, front_num
```

[14]:
```
(array([[-0.95713719, -0.09067194],
        [-0.92633083, -0.29208351],
        [-0.63329589, -0.63329589],
        [-0.52191048, -0.72845916],
        [-0.26132949, -0.87913689],
        [-0.17190645, -0.91382463]]),
 array([40,  3, 19, 16, 29, 41]))
```

解（評価値）のプロット

これ以降、図示する空間が $y = (y_1, y_2)$ であり $x = (x_1, x_2)$ ではないことにあらためて注意してください。

赤のプロットがパレート解です。

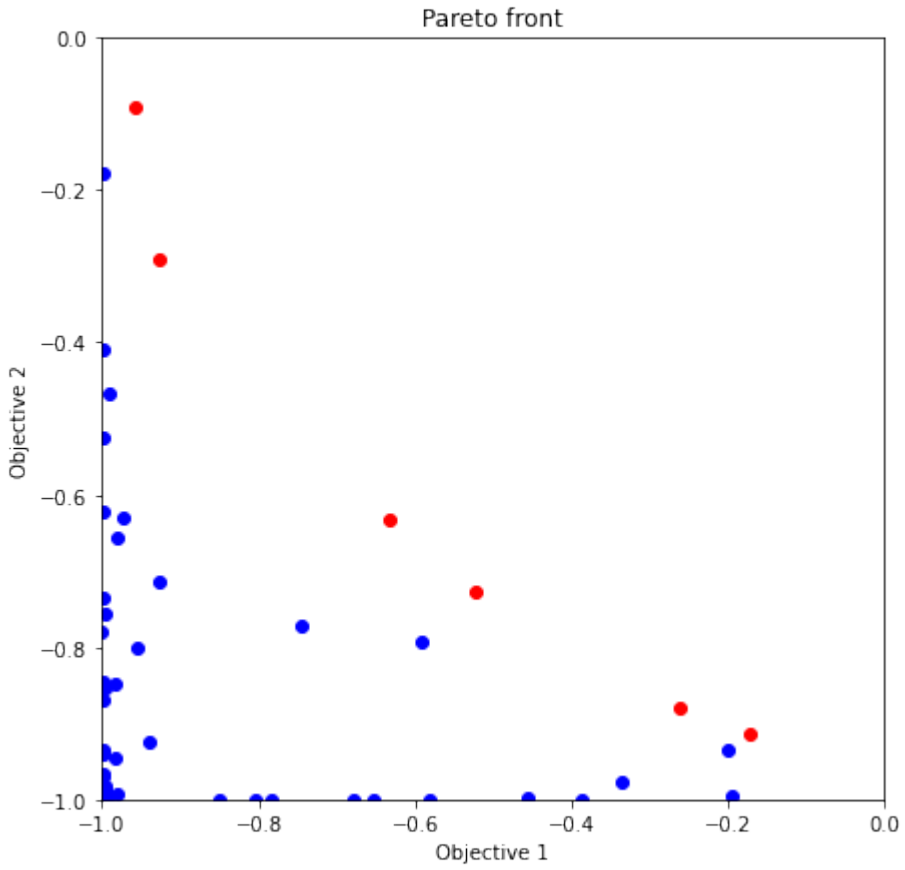
[15]:
```
def plot_pareto_front(res):
    front, front_num = res.export_pareto_front()
    dominated = [i for i in range(res.num_runs) if i not in front_num]
    points = res.fx[dominated, :]
```

(次のページに続く)

(前のページからの続き)

```
    plt.figure(figsize=(7, 7))
    plt.scatter(res.fx[dominated,0], res.fx[dominated,1], c = "blue")
    plt.scatter(front[:, 0], front[:, 1], c = "red")
    plt.title('Pareto front')
    plt.xlabel('Objective 1')
    plt.ylabel('Objective 2')
    plt.xlim([-1.0,0.0])
    plt.ylim([-1.0,0.0])
```

[16]:
```
plot_pareto_front(res_random)
```



劣解領域 (dominated region) の体積を計算

パレート解ではない解、すなわち、「自らよりも優れた解 y' が存在する解 y 」 を劣解と呼びます ($\exists y' y \prec y'$)。

解空間（の部分空間）のうち、劣解の占める空間である劣解領域の体積は、多目的最適化の結果を示す指標のひとつです。

この値が大きいほど、より良いパレート解が多く求まっていることになります。

`res_random.pareto.volume_in_dominance(ref_min, ref_max)` は、`ref_min, ref_max` で指定された矩形 (hyper-rectangle) 中の劣解領域体積を計算します。

[17]:
```
res_random.pareto.volume_in_dominance([-1,-1],[0,0])
```

[17]:
```
0.2376881844865093
```

3.6.6 ベイズ最適化

多目的の場合の `bayes_search` では、`score` には以下のいずれかを指定します。

- HVPI (Hypervolume-based Probability of Improvement)
- EHVI (Expected Hyper-Volume Improvement)
- TS (Thompson Sampling)

以下、score を変えてそれぞれ 50 回 (ランダムサーチ 10 回 + ベイズ最適化 40 回) 評価を行います。

HVPI (Hypervolume-based Probability of Improvement)

多次元の目的関数空間における非劣解領域 (non-dominated region) の改善確率を score として求めます。

- 参考文献
  - Couckuyt, Ivo, Dirk Deschrijver, and Tom Dhaene. "Fast calculation of multiobjective probability of improvement and expected improvement criteria for Pareto optimization." Journal of Global Optimization 60.3 (2014): 575-594.

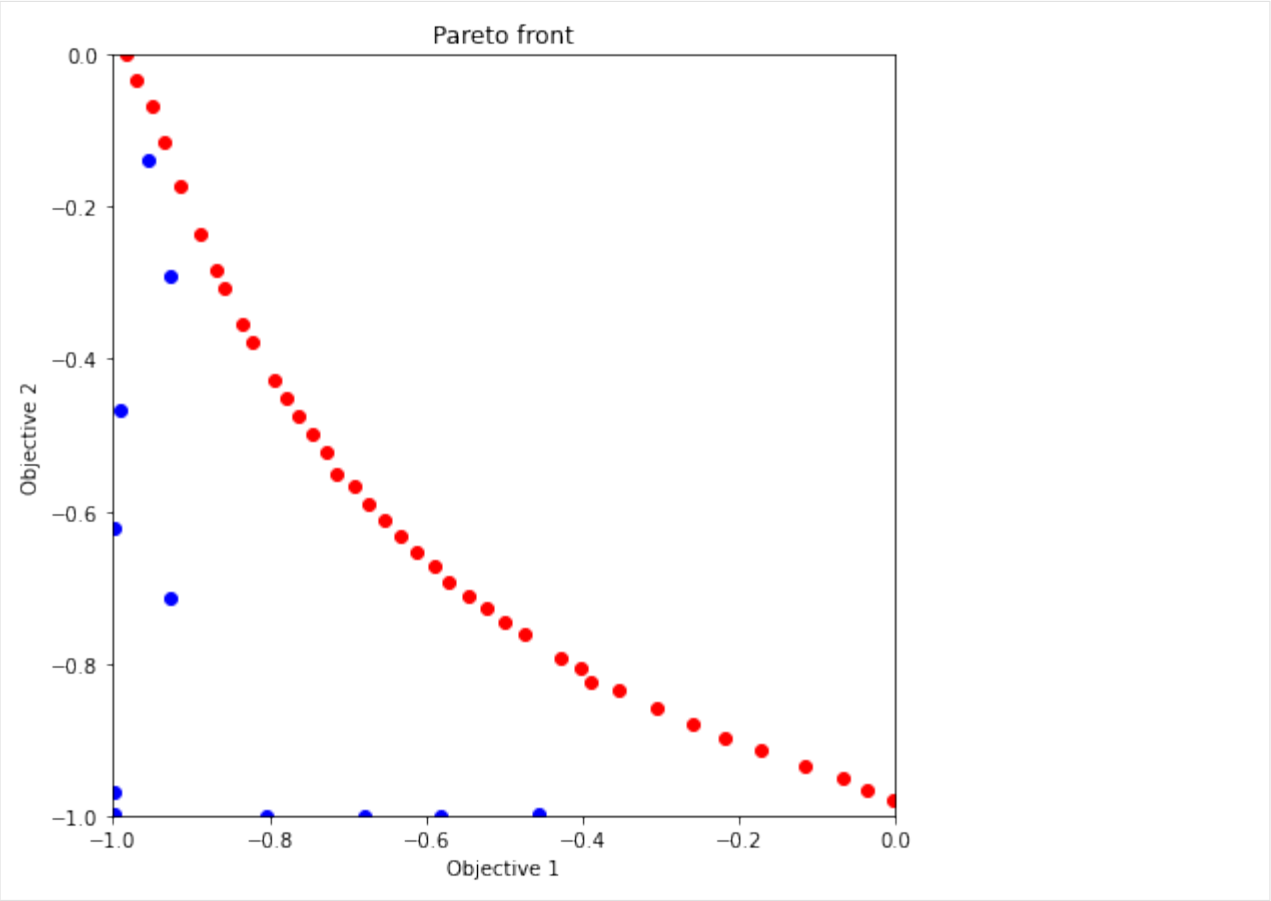
[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

policy.random_search(max_num_probes=10, simulator=simu)
res_HVPI = policy.bayes_search(max_num_probes=40, simulator=simu, score='HVPI',␣
↪interval=10)
```

パレート解のプロット

ランダムサンプリングと比較して、パレート解が多く求まっていることが分かります。

[19]:
```
plot_pareto_front(res_HVPI)
```



劣解領域体積

[20]:
```
res_HVPI.pareto.volume_in_dominance([-1,-1],[0,0])
```

[20]:
```
0.32877907991633726
```

EHVI (Expected Hyper-Volume Improvement)

多次元の目的関数空間における非劣解領域 (non-dominated region) の改善期待値を score として求めます。

- 参考文献
  - Couckuyt, Ivo, Dirk Deschrijver, and Tom Dhaene. “Fast calculation of multiobjective probability of improvement and expected improvement criteria for Pareto optimization.” Journal of Global Optimization 60.3 (2014): 575-594.

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)
```

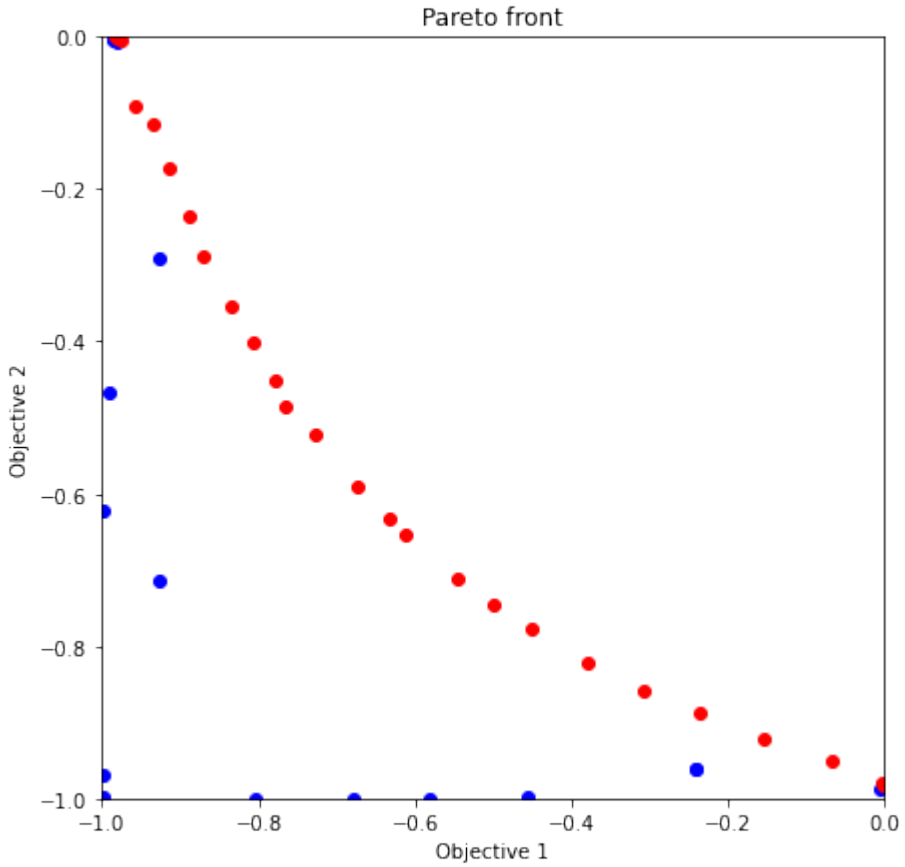
(次のページに続く)

(前のページからの続き)

```
policy.random_search(max_num_probes=10, simulator=simu)
res_EHVI = policy.bayes_search(max_num_probes=40, simulator=simu, score='EHVI',␣
↪interval=10)
```

パレート解のプロット

[22]:
```
plot_pareto_front(res_EHVI)
```



劣解領域体積

[23]:
```
res_EHVI.pareto.volume_in_dominance([-1,-1],[0,0])
```

[23]:
```
0.3200467412741881
```

TS (Thompson Sampling)

単目的の場合の Thompson Sampling では、各候補 (test_X) について、目的関数の事後分布からサンプリングを行い、値が最大となる候補を次の探索点として推薦します。

多目的の場合は、サンプリングした値についてパレートルールの上で最大となる候補、つまりパレート最適な候補の中からランダムに 1 つ選択して次の探索点とします。

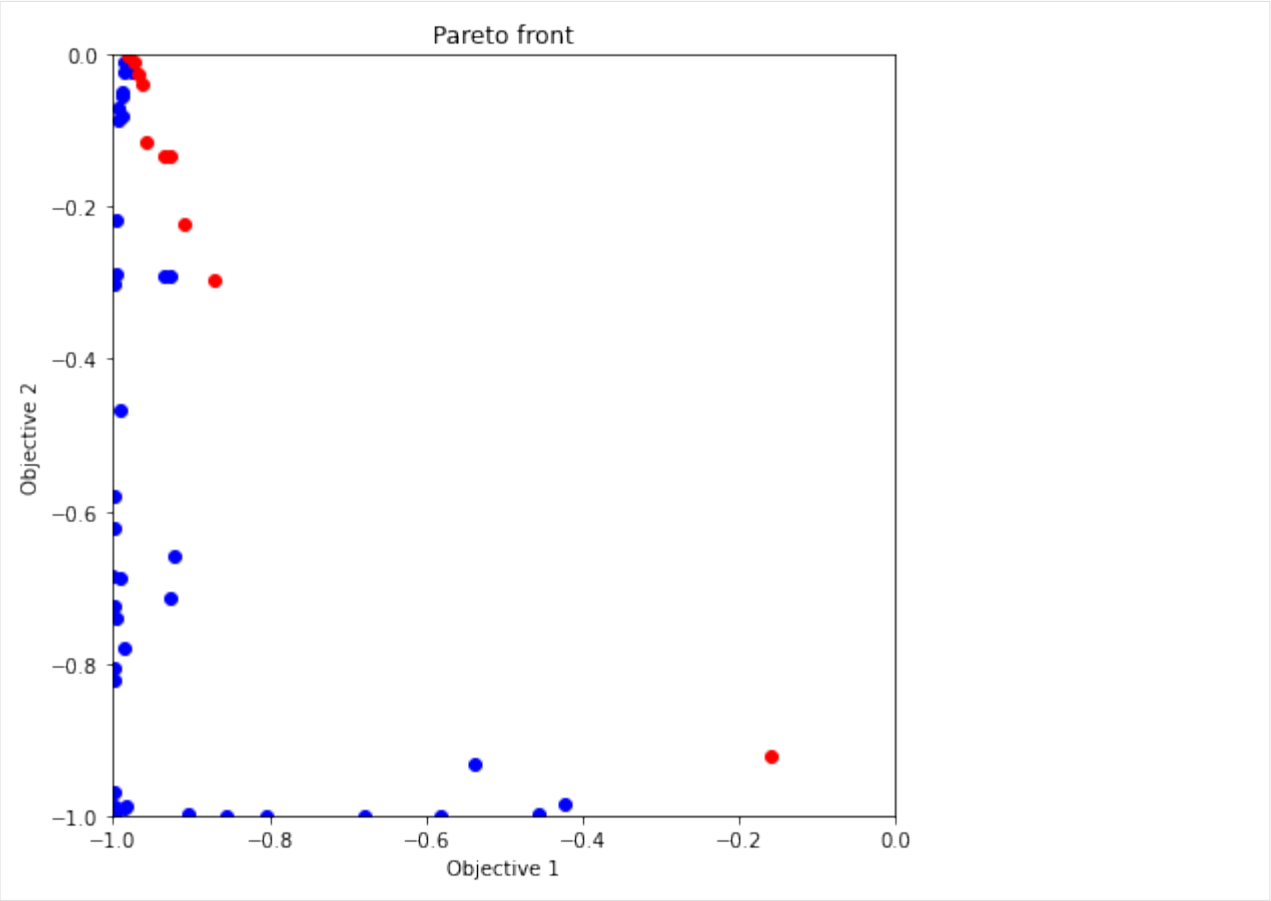
- 参考文献
  - Yahyaa, Saba Q., and Bernard Manderick. “Thompson sampling for multi-objective multi-armed bandits problem.” Proc. Eur. Symp. Artif. Neural Netw., Comput. Intell. Mach. Learn.. 2015.

[ ]:
```
policy = physbo.search.discrete_multi.policy(test_X=test_X, num_objectives=2)
policy.set_seed(0)

policy.random_search(max_num_probes=10, simulator=simu)
res_TS = policy.bayes_search(max_num_probes=40, simulator=simu, score='TS',␣
↪interval=10, num_rand_basis=5000)
```

パレート解のプロット

[25]:
```
plot_pareto_front(res_TS)
```



劣解領域体積

[26]:
```
res_TS.pareto.volume_in_dominance([-1,-1],[0,0])
```

[26]:
```
0.16415446221006114
```

3.6.7 付録：全探索

random_search で max_num_probes に全データ数 (N = test_X.shape[0]) を渡すと手軽に全探索できます。

全データの評価には時間がかかるため、あらかじめデータ数を減らしておきます。

[ ]:
```
test_X_sparse = np.array(list(itertools.product(np.linspace(-2, 2, 21), repeat=2)))
simu_sparse = simulator(test_X_sparse)

policy = physbo.search.discrete_multi.policy(test_X=test_X_sparse, num_objectives=2)
```

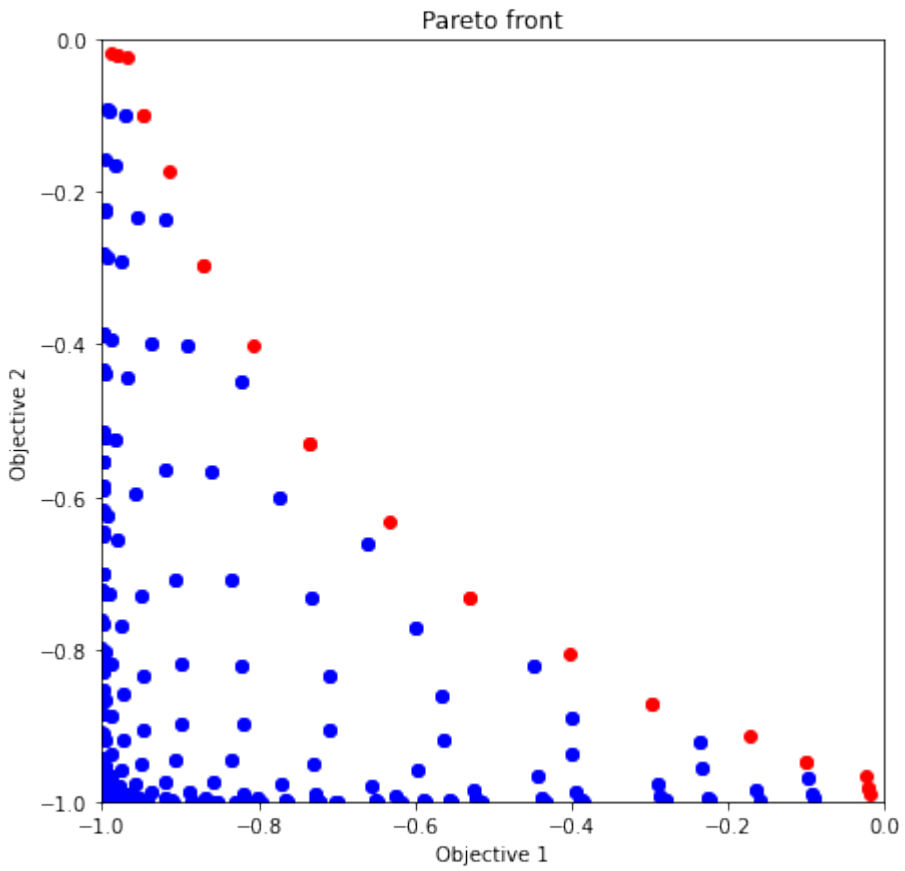
(次のページに続く)

(前のページからの続き)

```
policy.set_seed(0)

N = test_X_sparse.shape[0]
res_all = policy.random_search(max_num_probes=N, simulator=simu_sparse)
```

[28]:
```
plot_pareto_front(res_all)
```



[29]:
```
res_all.pareto.volume_in_dominance([-1,-1],[0,0])
```

[29]:
```
0.30051687493437484
```

第 4 章

アルゴリズム

ここでは、ベイズ最適化に関する説明を行います。技術的な詳細については、こちらの文献 を参照してください。

4.1 ベイズ最適化

ベイズ最適化は、複雑なシミュレーションや、実世界における実験タスクなど、目的関数（特性値など）の評価に大きなコストがかかるような場合に利用できる手法です。つまり、「できるだけ少ない実験・シミュレーション回数でより良い目的関数（材料特性など）を持つ説明変数（材料の組成、構造、プロセスやシミュレーションパラメータなど）を見つけ出す」ことが、ベイズ最適化によって解かれる問題です。ベイズ最適化では、探索する説明変数（ベクトル $\mathbf{x}$ で表す）の候補をあらかじめリストアップした状況からスタートします。そして、候補の中から目的関数 y が良くなると考えられる候補を、機械学習（ガウス過程回帰を利用）による予測をうまく利用することで選定します。その候補に対して実験・シミュレーションを行い目的関数の値を評価します。機械学習による選定・実験シミュレーションによる評価を繰り返すことにより、できるだけ少ない回数で最適化が可能となります。

ベイズ最適化のアルゴリズムの詳細を以下に示します。

- ステップ 1 ：初期化

探索したい空間をあらかじめ用意します。つまり、候補となる材料の組成・構造・プロセスやシミュレーションパラメータ等を、ベクトル $\mathbf{x}$ で表現しリストアップします。この段階では、目的関数の値はわかっていません。このうち初期状態としていくつかの候補を選び、実験またはシミュレーションによって目的関数の値 y を見積もります。これにより、説明変数 $\mathbf{x}$ と目的関数 y が揃った学習データ $D = \{\mathbf{x}_i, y_i\}_{(i=1,\cdots,N)}$ が得られます。

- ステップ 2 ：候補選定

学習データを用いて、ガウス過程を学習します。ガウス過程によれば、任意の $\mathbf{x}$ における予測値の平均を $\mu_c(\mathbf{x})$ 、分散を $\sigma_c(\mathbf{x})$ とすると、

$$\mu_c(\mathbf{x}) = \mathbf{k}(\mathbf{x})^T (K + \sigma^2 I)^{-1} \mathbf{y}$$
$$\sigma_c(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) + \sigma^2 - \mathbf{k}(\mathbf{x})^T (K + \sigma^2 I)^{-1} \mathbf{k}(\mathbf{x})$$

となります。ただし、 $k(\mathbf{x}, \mathbf{x}')$ はカーネルと呼ばれる関数であり、 2 つのベクトルの類似度を表します。一般に、以下のガウスカーネルが使われます。

$$k(\mathbf{x}, \mathbf{x}') = \exp\left[-\frac{1}{2\eta^2}||\mathbf{x} - \mathbf{x}'||^2\right]$$

また、このカーネル関数を利用し、 $\mathbf{k}(\mathbf{x})$ および K は以下のように計算されます。

$$\mathbf{k}(\mathbf{x}) = (k(\mathbf{x}_1, \mathbf{x}), k(\mathbf{x}_2, \mathbf{x}), \cdots, k(\mathbf{x}_N, \mathbf{x}))^\top$$

$$K = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \dots & k(\mathbf{x}_1, \mathbf{x}_N) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \dots & k(\mathbf{x}_2, \mathbf{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}_1) & k(\mathbf{x}_N, \mathbf{x}_2) & \dots & k(\mathbf{x}_N, \mathbf{x}_N) \end{pmatrix}$$

まだ実験やシミュレーションを行っていない候補全てに対して、予測値 $\mu_c(\mathbf{x})$ および予測の不確かさに関連する分散 $\sigma_c(\mathbf{x})$ を見積もります。これを用いて獲得関数を計算し、目的関数の値がまだわかっていない候補の中から、獲得関数を最大化する候補 $\mathbf{x}^*$ を選定します。このとき、 σ および η はハイパーパラメタと呼ばれ、PHYSBO では最適な値が自動で設定されます。

獲得関数として、例えば、最大改善確率 (PI : Probability of Improvement)、最大期待改善率 (EI : Expected Improvement) が有用です。PI によるスコアは次のように定義される。

$$\mathrm{PI}(\mathbf{x}) = \Phi(z(\mathbf{x})), \quad z(\mathbf{x}) = \frac{\mu_c(\mathbf{x}) - y_{\max}}{\sigma_c(\mathbf{x})}$$

ここで $\Phi(\cdot)$ は累積分布関数です。PI スコアは、現在得られている y の最大値 $y_{\max}$ を超える確率を表します。さらに、EI によるスコアは、予測値と現在の最大値 $y_{\max}$ との差の期待値であり、次式で与えられます。

$$\mathrm{EI}(\mathbf{x}) = [\mu_c(\mathbf{x}) - y_{\max}]\Phi(z(\mathbf{x})) + \sigma_c(\mathbf{x})\phi(z(\mathbf{x})), \quad z(\mathbf{x}) = \frac{\mu_c(\mathbf{x}) - y_{\max}}{\sigma_c(\mathbf{x})}$$

ここで $\phi(\cdot)$ は確率密度関数です。

- ステップ 3 ：実験

ステップ 2 で選定された獲得関数が最大となる候補 $\mathbf{x}^*$ に対して実験またはシミュレーションを行い、目的関数値 y を見積もります。これにより学習データが一つ追加されます。このステップ 2 、 3 を繰り返すことで、スコアのよい候補を探索します。

4.2 PHYSBO によるベイズ最適化の高速化

PHYSBO では、random feature map、トンプソンサンプリング、コレスキー分解を利用することで、ベイズ最適化の高速化を実現しています。まず、random feature map について説明します。random feature map $\phi(\mathbf{x})$ を導入することでガウスカーネル $k(\mathbf{x}, \mathbf{x}')$ を以下のように近似しています。

$$k(\mathbf{x}, \mathbf{x}') = \exp\left[-\frac{1}{2\eta^2}\|\mathbf{x} - \mathbf{x}'\|\right]^2 \simeq \phi(\mathbf{x})^\top \phi(\mathbf{x}')$$

$$\phi(\mathbf{x}) = (z_{\omega_1, b_1}(\mathbf{x}/\eta), ..., z_{\omega_l, b_l}(\mathbf{x}/\eta))^\top$$

ただし、 $z_{\omega,b}(\mathbf{x}) = \sqrt{2}\cos(\boldsymbol{\omega}^\top \mathbf{x} + b)$ としています。このとき、 $\boldsymbol{\omega}$ は $p(\boldsymbol{\omega}) = (2\pi)^{-d/2}\exp(-\|\boldsymbol{\omega}\|^2/2)$ より生成され、 b は $[0, 2\pi]$ から一様に選ばれます。この近似は、 $l \to \infty$ の極限で厳密に成立し、 l の値が random feature map の次元となります。

このとき Φ を、以下のように学習データのベクトル $\mathbf{x}$ による $\phi(\mathbf{x}_i)$ を各列に持つ l 行 n 列行列とします。

$$\Phi = (\phi(\mathbf{x}_1), ..., \phi(\mathbf{x}_n))$$

すると、

$$\mathbf{k}(\mathbf{x}) = \Phi^\top \phi(\mathbf{x})$$
$$K = \Phi^\top \Phi$$

という関係が成立することがわかります。

次に、トンプソンサンプリングを利用することで、候補の予測にかかる計算時間を $O(l)$ にする手法について紹介します。EI や PI を利用すると、分散を評価する必要があるため $O(l^2)$ になってしまうことに注意が必要です。トンプソンサンプリングを行うために、以下で定義されるベイズ線形モデルを利用します。

$$y = \mathbf{w}^\top \phi(\mathbf{x})$$

ただし、この $\phi(\mathbf{x})$ は前述した random feature map であり、 $\mathbf{w}$ は係数ベクトルです。ガウス過程では、学習データ D があたえられたとき、この $\mathbf{w}$ が以下のガウス分布に従うように決まります。

$$p(\mathbf{w}|D) = \mathcal{N}(\boldsymbol{\mu}, \Sigma)$$
$$\boldsymbol{\mu} = (\Phi\Phi^\top + \sigma^2 I)^{-1}\Phi\mathbf{y}$$
$$\Sigma = \sigma^2(\Phi\Phi^\top + \sigma^2 I)^{-1}$$

トンプソンサンプリングでは、この事後確率分布にしたがって係数ベクトルを一つサンプリングし、それを $\mathbf{w}^*$ とすることで、獲得関数を

$$\mathrm{TS}(\mathbf{x}) = \mathbf{w}^{*\top}\phi(\mathbf{x})$$

と表す。これを最大とする $\mathbf{x}^*$ が次の候補として選出されます。このとき、 $\phi(\mathbf{x})$ は l 次元ベクトルなため、獲得関数の計算は $O(l)$ で実行できます。

次に $\mathbf{w}$ のサンプリングの高速化について紹介します。行列 A を以下のように定義します。

$$A = \frac{1}{\sigma^2}\Phi\Phi^\top + I$$

すると、事後確率分布は、

$$p(\mathbf{w}|D) = \mathcal{N}\left(\frac{1}{\sigma^2}A^{-1}\Phi\mathbf{y}, A^{-1}\right)$$

と表すことができます。そのため、 $\mathbf{w}$ をサンプリングするためには、 A^{-1} の計算が必要となります。ここで、ベイズ最適化のイテレーションにおいて、新しく $(\mathbf{x}', y')$ が加わった場合について考えます。このデータの追加により、行列 A は、

$$A' = A + \frac{1}{\sigma^2}\phi(\mathbf{x}')\phi(\mathbf{x}')^\top$$

と更新されます。この更新は、コレスキー分解 ($A = L^\top L$) を用いることで、 A^{-1} の計算にかかる時間を $O(l^2)$ にすることができます。もし、 A^{-1} をイテレーションごとに毎回計算すると $O(l^3)$ の計算が必要になります。実際、 $\mathbf{w}$ をサンプリングする際は、

$$\mathbf{w}^* = \boldsymbol{\mu} + \mathbf{w}_0$$

とし、 $\mathbf{w}_0$ を $\mathcal{N}(0, A^{-1})$ からサンプリングします。また、 $\boldsymbol{\mu}$ は、

$$L^\top L\boldsymbol{\mu} = \frac{1}{\sigma^2}\Phi\mathbf{y}$$

を解くことで得られます。これらの技術を利用することで、学習データ数に対してほぼ線形の計算時間を実現しています。

第 5 章

謝辞

PHYSBO は東京大学物性研究所 ソフトウェア高度化プロジェクト (2020 年度) の支援を受け開発されました。この場を借りて感謝します。

第 6 章

お問い合わせ

PHYSBO に関するお問い合わせはこちらにお寄せください。

- バグ報告

  PHYSBO のバグ関連の報告は GitHub の Issues で受け付けています。

  バグを早期に解決するため、報告時には次のガイドラインに従ってください。

  - 使用している PHYSBO のバージョンを指定してください。
  - インストールに問題がある場合には、使用しているオペレーティングシステムとコンパイラの情報についてお知らせください。
  - 実行に問題が生じた場合は、実行に使用した入力ファイルとその出力を記載してください。

- その他

  研究に関連するトピックなど GitHub の Issues で相談しづらいことを問い合わせる際には、以下の連絡先にコンタクトをしてください。

  E-mail: `physbo-dev__at__issp.u-tokyo.ac.jp` (_at_を@に変更してください)