ベイズ最適化パッケージ
PHYSBO の使い方

Website: https://www.pasums.issp.u-tokyo.ac.jp/physbo

GitHub: https://github.com/issp-center-dev/PHYSBO

E-mail: physbo-dev@issp.u-tokyo.ac.jp

Paper: Computer Physics Communications 278,108405 (2022)

東大物性研　**本山裕一**

2022-06-20 物性研CCMS 講習会

PHYSBO のインストール

- 最新版 (v1.1.0) は Python3.6 以上が必要
- `pip install physbo` でインストール可能
  - インストール方法について詳しくは pip のマニュアルや文献を参照のこと
    - バージョン更新は `pip install -U physbo`
    - バージョン指定は `physbo==1.1.0`
    - インストール場所の変更は `install --user` や `--prefix=PATH`
  - インストールには C コンパイラ及び NumPy, Cython が必要
    - pip が十分新しければ後者は自動で入るが、だめなら手動で導入
    - NumPy 1.20 以降と 1.19 以前とでバイナリに互換性がないので注意
      - インストールに使ったバージョンと実際に使ったバージョンを合わせるのが一番安全
- MateriApps LIVE! を使う場合は最初から入っています

PHYSBO のインストール

・ インストールできたかの簡易確認

```
$ python3                       # Pythonの起動

>>> import physbo
>>> physbo.__version__  # バージョン確認
'1.1.0'
```

・ 次のエラーメッセージが出た場合（数字は違うこともある）

- `ValueError: numpy.ndarray size changed, may indicate binary incompatibility. Expected 96 from C header, got 88 from PyObject`

・ NumPy のバージョンを更新した後に PHYSBO の再インストールを試みてください

```
$ python3 -m pip uninstall physbo  # PHYSBO のアンインストール
$ python3 -m pip install -U numpy  # NumPy の更新
$ python3 -m pip install physbo    # PHYSBO の再インストール
```

PHYSBO のインストール（サンプル）

- PHYSBO を使う場合には自分でpython スクリプトを書く必要がある
  - 基本的にはサンプルスクリプトやチュートリアルを改造すればよい
  - サンプルスクリプトは pip install では入らない
    - https://github.com/issp-center-dev/PHYSBO から持ってくる
      - `git clone https://github.com/issp-center-dev/PHYSBO`
    - `examples` ディレクトリ以下にサンプルが存在
- チュートリアルは
  - https://issp-center-dev.github.io/PHYSBO/manual/master/ja/notebook/
    - もととなっている jupyter notebook は `docs/sphinx/manual/ja/source/notebook` 以下にある
- MateriApps LIVE! では `/usr/share/physbo` にexamples がインストールされている

PHYSBO のインストール(on MALIVE!)

```
# この2つは新しくMA LIVE! を準備した場合は不要です
$ sudo apt update                  # パッケージ一覧の更新
$ sudo apt install physbo          # PHYSBO の更新

$ mkdir physbo                                  # 作業用ディレクトリを作成して
$ cd physbo                                     # そこに移動
$ cp -r /usr/share/physbo/examples ./ # サンプルのコピー
$ cd examples
$ python3 simple.py                             # とりあえず実行してみる

   ... 中略 ...
best_fx: -0.0 at [0. 0. 0.]
```

NOTE: 最新版がまだ用意されていない場合には
前のページのやり方でインストールする

PHYSBO の使用法 (Basic)

- PHYSBO を使ったベイズ最適化に必要な要素は大雑把に3つ
  1. 解きたい最適化問題
     - 目的関数 y = f(x)
       - PHYSBO は最大化問題であると仮定する
     - x の探索空間（候補点の集合）
       - PHYSBO は離散空間中で最適化する
  2. ベイズ最適化のパラメータ
     - 初期データの準備
     - 獲得関数の選択
     - ガウス過程の表現能力
  3. 結果の取り出し方
     - 探索済みの点
     - 獲得関数
     - 学習したガウス過程（各点での期待値、分散）

→実際にスクリプトを見ていきましょう

PHYSBO の使用法 (Basic)

- examples/simple.py は以下の最適化問題を PHYSBO を用いて解くプログラム
  - 探索空間が3次元
    - パラメータの数が3
  - 候補点は乱数生成
- 目的関数が

$$f(\vec{x}) = -\sum_{i=1}^{3} x_i^2$$

次ページから解説

```
import numpy as np
import physbo

# 候補点の集合を生成する
D = 3     # 探索パラメータの数（パラメータ空間の次元）
N = 1000  # 候補点の数
test_X = np.random.randn(N, D)  # 正規分布
test_X[0,:] = 0.0 # 真の答えをつくっておく

# 目的関数 （二次関数）
# 探索点の番号の配列から対応する目的関数の値の配列を返す
def simulator(actions):
    return -np.sum(test_X[actions, :] ** 2, axis=1)

# 最適化を行うクラス policy の初期化
policy = physbo.search.discrete.policy(test_X)

# ランダム探索 (10 個)
policy.random_search(max_num_probes=10, simulator=simulator)

# ベイズ探索 (40 回)
policy.bayes_search(
    max_num_probes=40, simulator=simulator, score="EI"
)

# これまでの最適解を表示
best_fx, best_actions = policy.history.export_sequence_best_fx()
print(f"best_fx: {best_fx[-1]} at {test_X[best_actions[-1], :]}")
```

PHYSBO の使用法 (Basic)

- PHYSBO はD 次元パラメータ空間中のN 個の点（候補点）の集合から、目的関数 f(x) の値が**最大**となる点を探索する
  - **候補点の集合** は N 行 D 列の行列 (np.ndarray)として表現する (test_X)
  - **目的関数**は候補点の番号 (action) の配列から対応する値の配列を返す関数として表現する (simulator)

```
import numpy as np
import physbo

# 候補点の集合を生成する
D = 3      # 探索パラメータの数（パラメータ空間の次元）
N = 1000   # 候補点の数
test_X = np.random.randn(N, D)  # 正規分布
test_X[0,:] = 0.0 # 真の答えをつくっておく

# 目的関数 （二次関数）
# 探索点の番号の配列から対応する目的関数の値の配列を返す
def simulator(actions):
    return -np.sum(test_X[actions, :] ** 2, axis=1)

# 最適化を行うクラス policy の初期化
policy = physbo.search.discrete.policy(test_X)

# ランダム探索 (10 個)
policy.random_search(max_num_probes=10, simulator=simulator)

# ベイズ探索 (40 回)
policy.bayes_search(
    max_num_probes=40, simulator=simulator, score="EI"
)

# これまでの最適解を表示
best_fx, best_actions = policy.history.export_sequence_best_fx()
print(f"best_fx: {best_fx[-1]} at {test_X[best_actions[-1], :]}")
```

PHYSBO の使用法 (Basic)

- physbo.search.discrete.policy が探索を行うクラス
- policy.random_search で候補点集合からランダムに選び出し、目的関数の値を計算する
- 結果はpolicy が覚えている

```
import numpy as np
import physbo

# 候補点の集合を生成する
D = 3     # 探索パラメータの数（パラメータ空間の次元）
N = 1000  # 候補点の数
test_X = np.random.randn(N, D)  # 正規分布
test_X[0,:] = 0.0 # 真の答えをつくっておく

# 目的関数 （二次関数）
# 探索点の番号の配列から対応する目的関数の値の配列を返す
def simulator(actions):
    return -np.sum(test_X[actions, :] ** 2, axis=1)

# 最適化を行うクラス policy の初期化
policy = physbo.search.discrete.policy(test_X)

# ランダム探索 (10 個)
policy.random_search(max_num_probes=10, simulator=simulator)

# ベイズ探索 (40 回)
policy.bayes_search(
    max_num_probes=40, simulator=simulator, score="EI"
)

# これまでの最適解を表示
best_fx, best_actions = policy.history.export_sequence_best_fx()
print(f"best_fx: {best_fx[-1]} at {test_X[best_actions[-1], :]}")
```

PHYSBO の使用法 (Basic)

- policy.bayes_search でベイズ的に探索する
- 40ステップ行う
- 獲得関数はEI
- 結果はpolicy が覚えている

```
import numpy as np
import physbo

# 候補点の集合を生成する
D = 3     # 探索パラメータの数（パラメータ空間の次元）
N = 1000  # 候補点の数
test_X = np.random.randn(N, D)  # 正規分布
test_X[0,:] = 0.0 # 真の答えをつくっておく

# 目的関数 （二次関数）
# 探索点の番号の配列から対応する目的関数の値の配列を返す
def simulator(actions):
    return -np.sum(test_X[actions, :] ** 2, axis=1)

# 最適化を行うクラス policy の初期化
policy = physbo.search.discrete.policy(test_X)

# ランダム探索 (10 個)
policy.random_search(max_num_probes=10, simulator=simulator)

# ベイズ探索 (40 回)
policy.bayes_search(
    max_num_probes=40, simulator=simulator, score="EI"
)

# これまでの最適解を表示
best_fx, best_actions = policy.history.export_sequence_best_fx()
print(f"best_fx: {best_fx[-1]} at {test_X[best_actions[-1], :]}")
```

PHYSBO の使用法 (Basic)

- policy.history が過去の探索結果
- history.export_sequence_best_fx は、各ステップごとに、それまでの最適解（値と番号）を出力する
- 最後のステップは -1 で取れる

```
import numpy as np
import physbo

# 候補点の集合を生成する
D = 3     # 探索パラメータの数（パラメータ空間の次元）
N = 1000  # 候補点の数
test_X = np.random.randn(N, D)  # 正規分布
test_X[0,:] = 0.0 # 真の答えをつくっておく

# 目的関数 （二次関数）
# 探索点の番号の配列から対応する目的関数の値の配列を返す
def simulator(actions):
    return -np.sum(test_X[actions, :] ** 2, axis=1)

# 最適化を行うクラス policy の初期化
policy = physbo.search.discrete.policy(test_X)

# ランダム探索 (10 個)
policy.random_search(max_num_probes=10, simulator=simulator)

# ベイズ探索 (40 回)
policy.bayes_search(
    max_num_probes=40, simulator=simulator, score="EI"
)

# これまでの最適解を表示
best_fx, best_actions = policy.history.export_sequence_best_fx()
print(f"best_fx: {best_fx[-1]} at {test_X[best_actions[-1], :]}")
```

PHYSBO の使用法 (Basic)

- policy.random_search の出力
- policy.bayes_search の出力
  - まず、ガウス過程のハイパーパラメータ学習を行う
  - その後にベイズ最適化
- 最終結果
  - 「候補点の集合からの最適解」であることに注意
    - 今回は候補点の中に「真の解」を手で追加してある

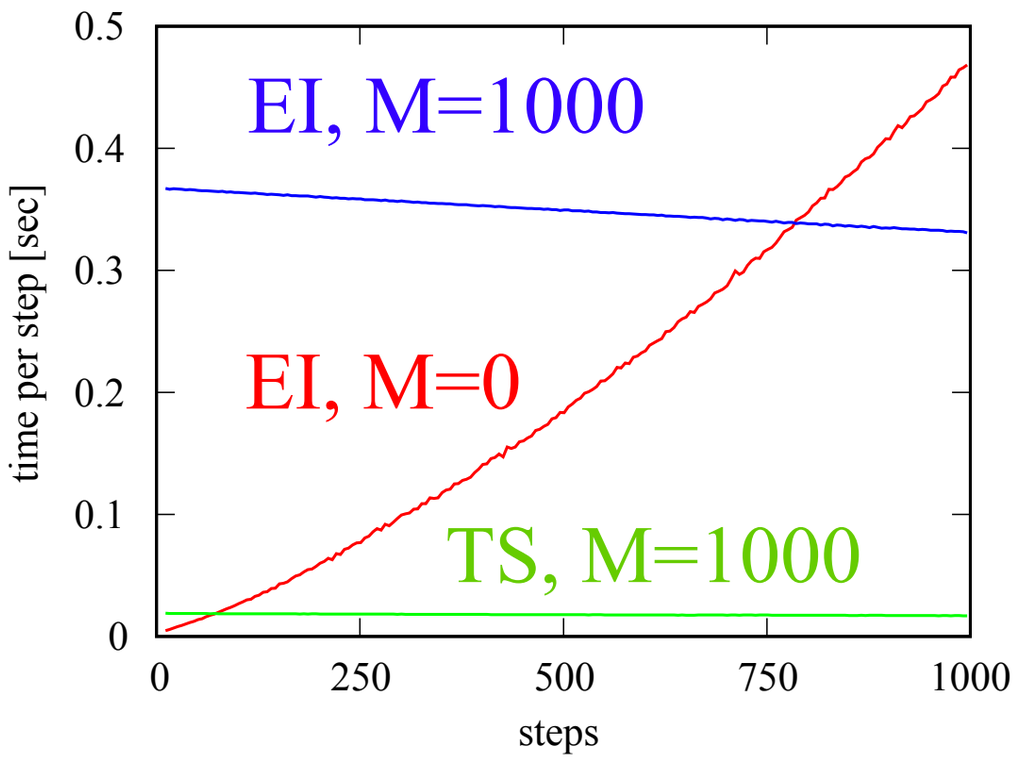
```
$ python3 simple.py
0001-th step: f(x) = -1.491918 (action=144)
   current best f(x) = -1.491918 (best action=144)
                  (中略)
0010-th step: f(x) = -0.172312 (action=531)
   current best f(x) = -0.172312 (best action=531)


Start the initial hyper parameter searching ...
Done

Start the hyper parameter learning ...
0 -th epoch marginal likelihood 20.066856052528934
                  (中略)
500 -th epoch marginal likelihood 18.20028024628816
Done

0011-th step: f(x) = -0.906000 (action=690)
   current best f(x) = -0.172312 (best action=531)
                  (中略)
0050-th step: f(x) = -5.873065 (action=830)
   current best f(x) = -0.000000 (best action=0)

best_fx: -0.0 at [0. 0. 0.]
```

PHYSBO の使用法 (Advanced)

- 目的関数の評価は自分でやりたい
  - 例：化学組成を入力として実際に合成し、測定した物理量（Tc など）を最大化したい
  - チュートリアルの「**インタラクティブに実行する**」および「**既存の計算結果を読み込んで実行する**」を参照
  - bayes_search に simulator を渡さないと、この関数は次に測定すべき候補点（の番号）を返す
    - `next_actions = policy.bayes_search(max_num_probes=1)`
  - 評価結果 `t = simulator(next_actions)` は `policy.write` で登録する
    - `policy.write(actions, t)`
  - 既知の測定結果（候補点番号と目的関数の組）は `policy` の初期化時にも渡せる
    - `discrete.policy(test_X, initial_data=(actions, fs))`
- 複数の目的関数を同時に扱いたい（多目的最適化）
  - `examples/multiple.py`
  - 公式ドキュメントを参照

PHYSBO の使用法 (Advanced)

- 高速化したい（その1）
  - **random feature map**を用いる（本スライドの付録やマニュアルの「アルゴリズム」を参照)
    - ランダム基底関数の数 M を num_rand_basis として渡す
    - `policy.bayes_search(num_rand_basis=1000)`
      - デフォルトは0 （「普通の」ガウス過程回帰を用いる）
      - Mが大きいほどモデル関数の表現能力が上がる
- N=10000 個の探索空間から次の候補点を求めるのにかかった時間（ステップあたり）
- M>0 では測定済み点数に対してほぼ定数
  - 測定済み点数が大きいときに効果的
- 獲得関数をトンプソンサンプリング("TS") にするとより効果的



PHYSBO の使用法 (Advanced)

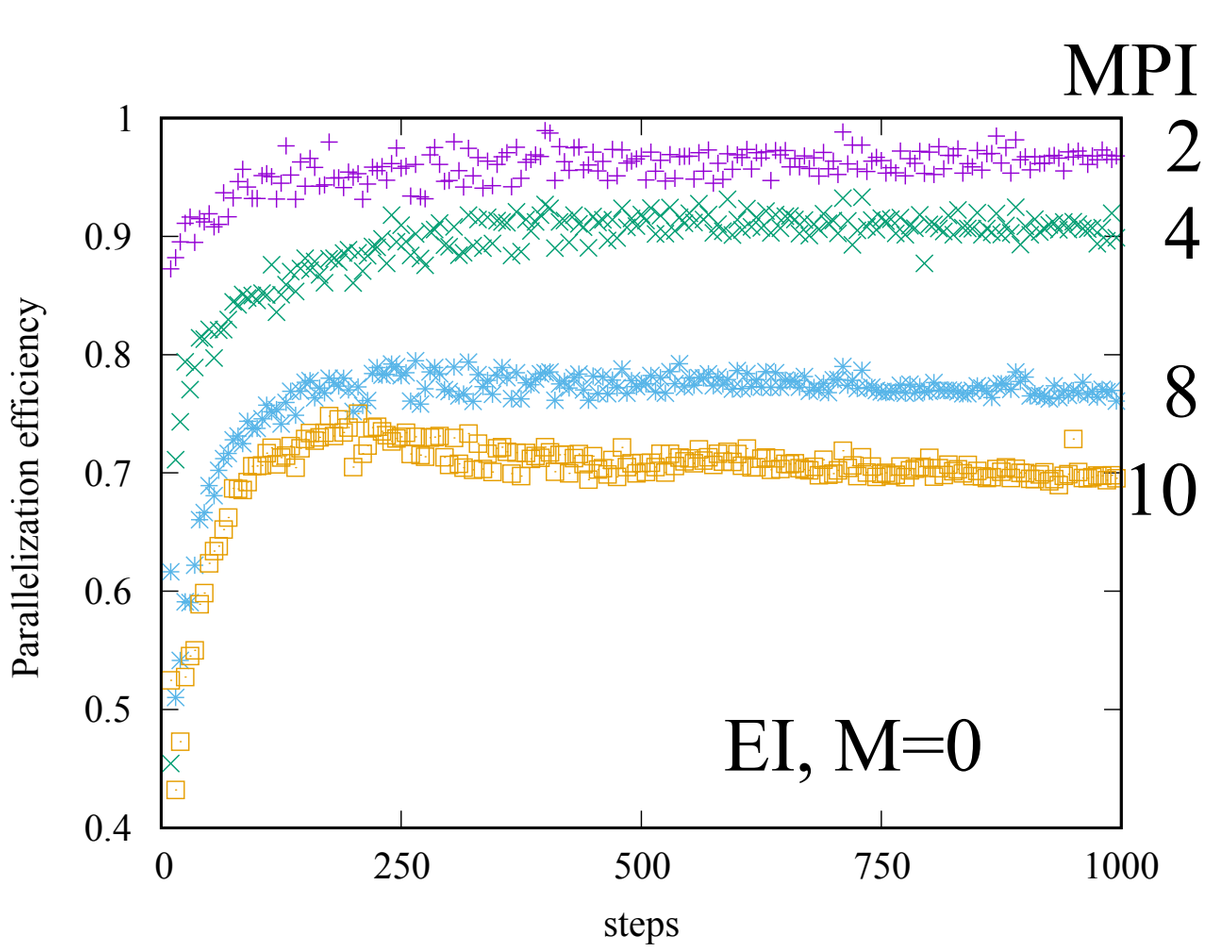
- 高速化したい（その2)
  - 獲得関数の最適化に関して、MPI 並列可能 (mpi4py を利用)
    - 高次元など、候補点が多い場合に有効
    - policy を作るときにMPIコミュニケータ (mpi4py.MPI.Comm) を渡す
      - ほとんどのケースでは全系（いわゆる MPI_COMM_WORLD）でよい

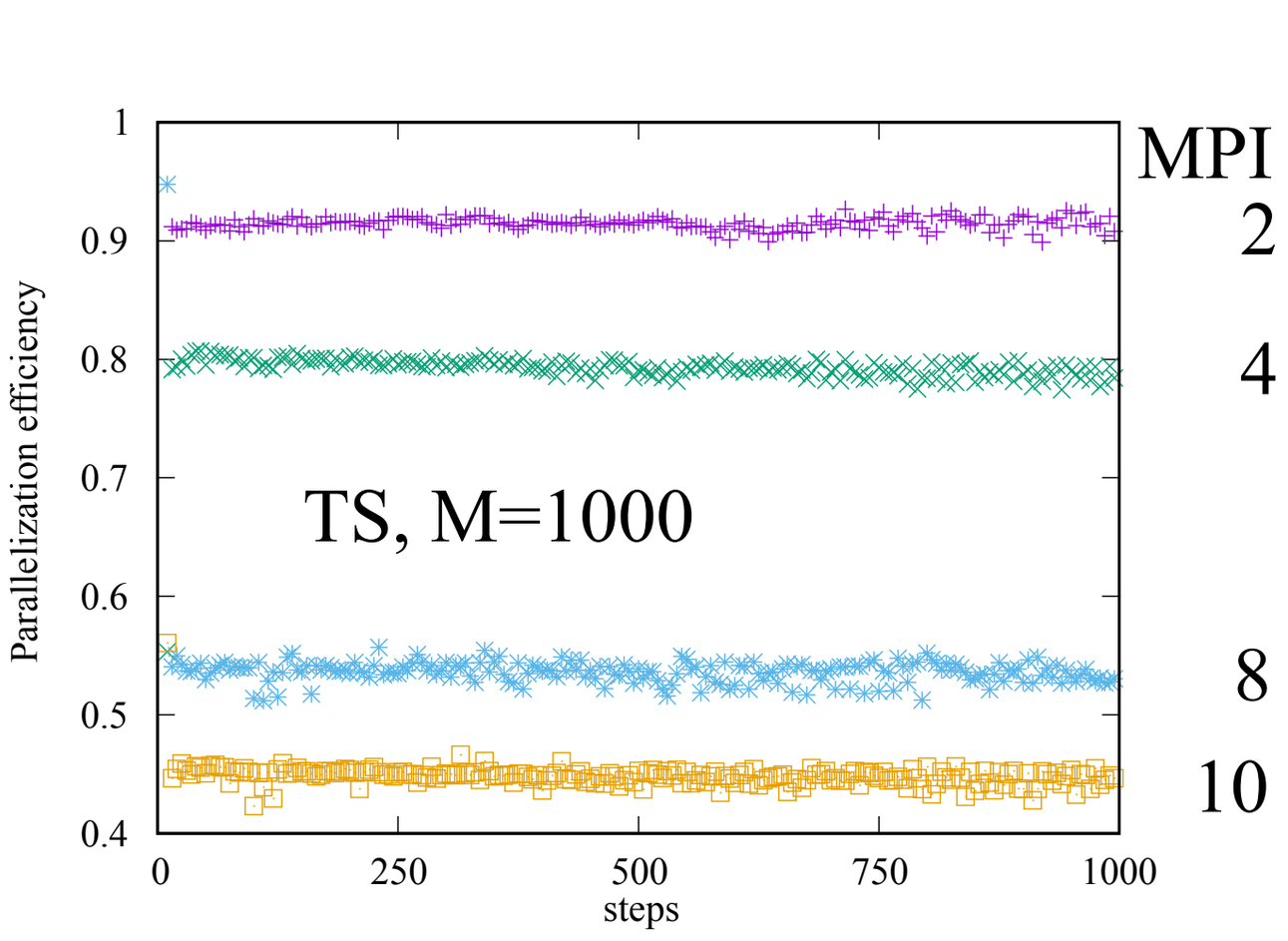
```
physbo.search.discrete.policy(test_X, comm=MPI.COMM_WORLD)
$ mpiexec -np 2 python3 simple.py
```

- 現状では、目的関数の計算は rank==0 のものだけが行う
- MateriApps LIVE! で試したい場合
  - VirtualBox の設定で利用する物理コアの数を変える
    - 一旦MateriApps LIVE! からログアウトする
    - 仮想マシン一覧から設定するマシンを右クリック→設定→システム→プロセッサー

PHYSBO の使用法 (Advanced)

- 高速化したい（その2)
  - 獲得関数の最適化に関して、MPI 並列可能 (mpi4py を利用)
  - 並列化効率を示したものが下図



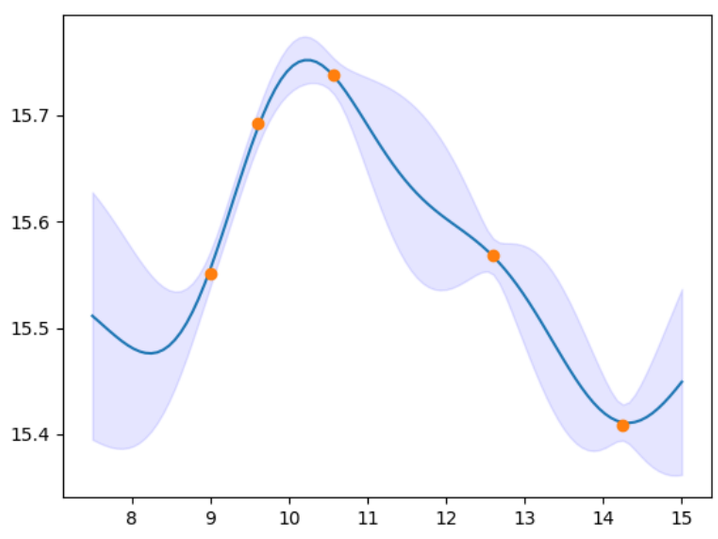


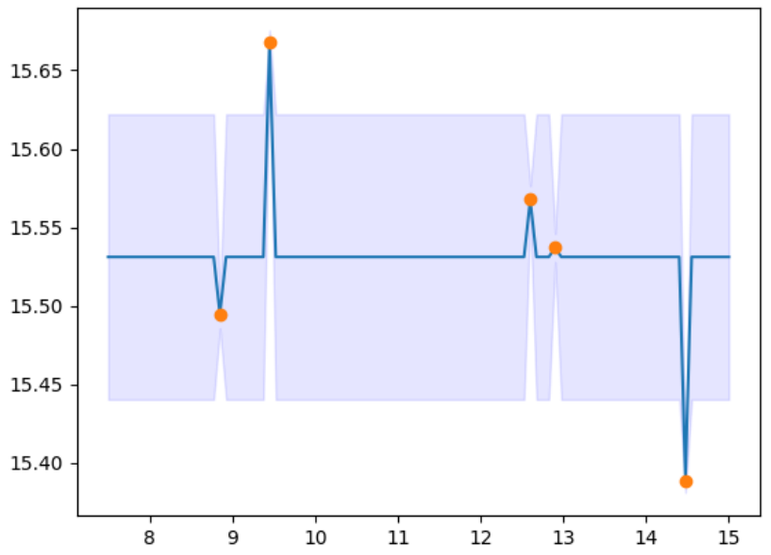
PHYSBO の使用法 (Advanced)

- 学習したモデル関数や獲得関数を見てみたい
  - 期待値
    - `policy.get_post_fmean(xs=test_X)`
  - 分散
    - `policy.get_post_fcov(xs=test_X)`
  - 獲得関数
    - `policy.get_score(mode="EI", xs=test_X)`
- もちろん `xs` や`mode` は学習に使ったものと異なっていても良い
- `examples/simple_score.py` がサンプルスクリプト
  - 結果を愚直に `print` しているだけ
- （単純な）可視化の例としてはチュートリアルの「PHYSBO の基本」

PHYSBO の使用法 (Advanced)

- ハイパーパラメータの学習について
  - ガウスカーネルの幅 η と測定にかかる誤差 α という2つのハイパーパラメータがある
  - PHYSBO では自動的に学習する（最尤法）
    - 初期データが偏っていた場合、失敗することもある
- 下は同じ関数から別々に 5点取ってきてハイパーパラメータ学習したもの
  - 右の状態では定数関数しか表現できない





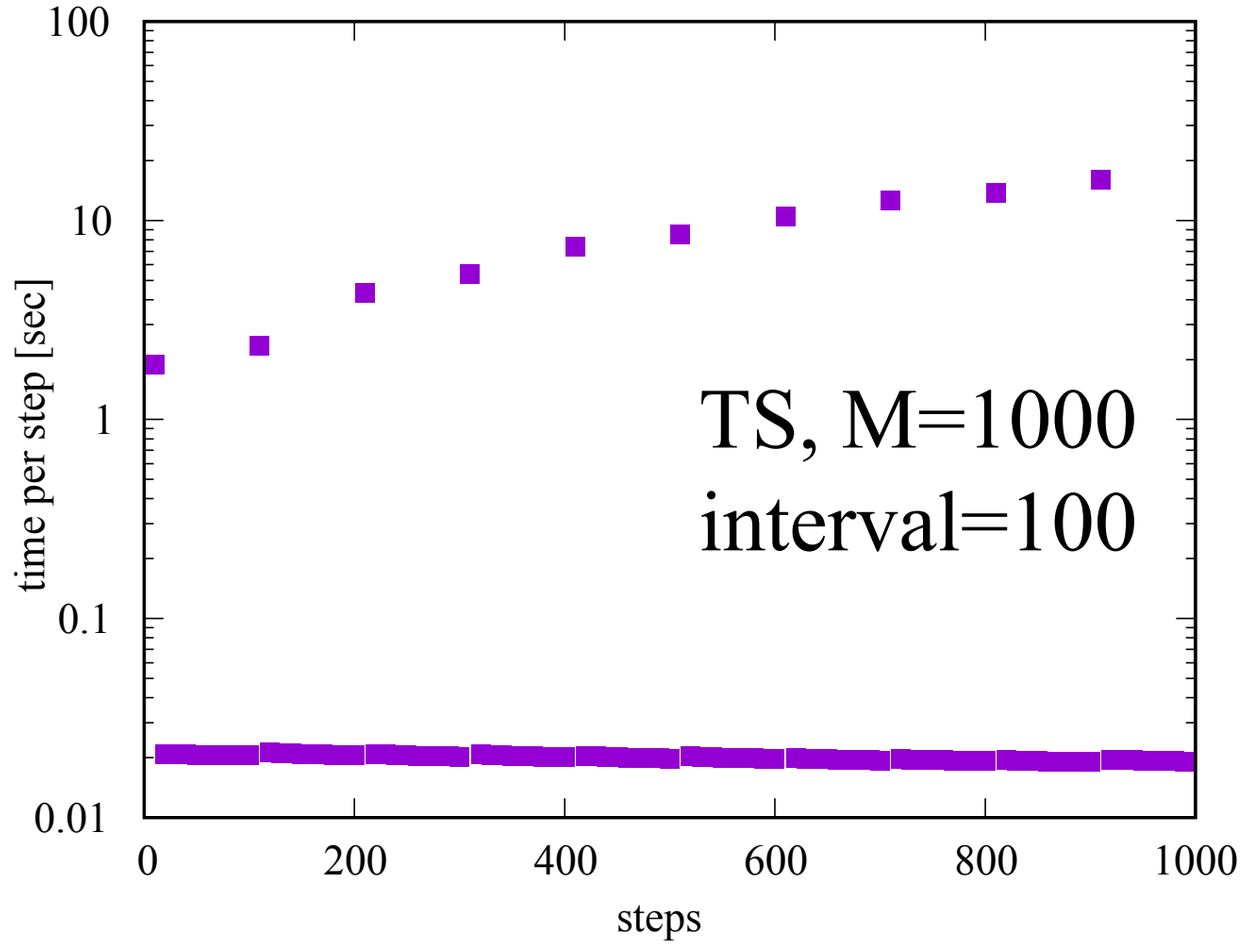
PHYSBO の使用法 (Advanced)

- ハイパーパラメータの再学習
  - データ点を増やしてから学習しなおすとたいていは直る
  - bayes_search のキーワード引数 interval でハイパーパラメータの再学習頻度を指定可能
    - policy.bayes_search(interval=10)
- デフォルトは0 で、この場合は最初に学習したのち再学習しない
  - 最初に失敗すると最後までダメ
- 負数を渡すと最初の学習も行わない
  - 獲得関数など、途中経過を出力した後に続行したいときには有用

PHYSBO の使用法 (Advanced)

- ハイパーパラメータの再学習
  - ハイパーパラメータ学習はモデル関数や獲得関数の計算と比べて、文字通り桁違いに時間がかかることに注意
    - 最初の学習の段階で可視化をしておくのが良い

- 右図はステップごとの所要時間
- 100 回に1回行っているハイパーパラメータ学習は、次候補選定と比べて100倍重い



演習例

- **モデル関数を可視化する（おすすめ）**
  - まずは 1パラメータの探索から始めるのがよいでしょう
- **獲得関数の種類やランダム基底の数、再学習頻度、並列化数を変えて遊んでみる**
  - `policy.history.time_total` などでステップごとの経過時間を取得できます
  - `examples/simple_time.py` に matplotlib を用いた例があります
- **他のシミュレーションソフトウェアと連携する（時間内にやるのは難しいかも）**
  - DFT ソフトウェア (例: Quantum ESPRESSO) と組み合わせて安定構造を探索する
    - 例： Si について、エネルギーが最小となるような格子定数を探る
  - 模型ソルバ（例: HΦ）と組み合わせてハミルトニアンのパラメータ探索
    - 例：スピン模型について、与えられた磁化曲線を再現するようなパラメータ（結合定数など）を探る
      - https://ma.issp.u-tokyo.ac.jp/app-post/2179?appid=1432

付録

ベイズ最適化の流れ

獲得関数

ガウス過程

ガウスカーネル

線形ベイズ回帰

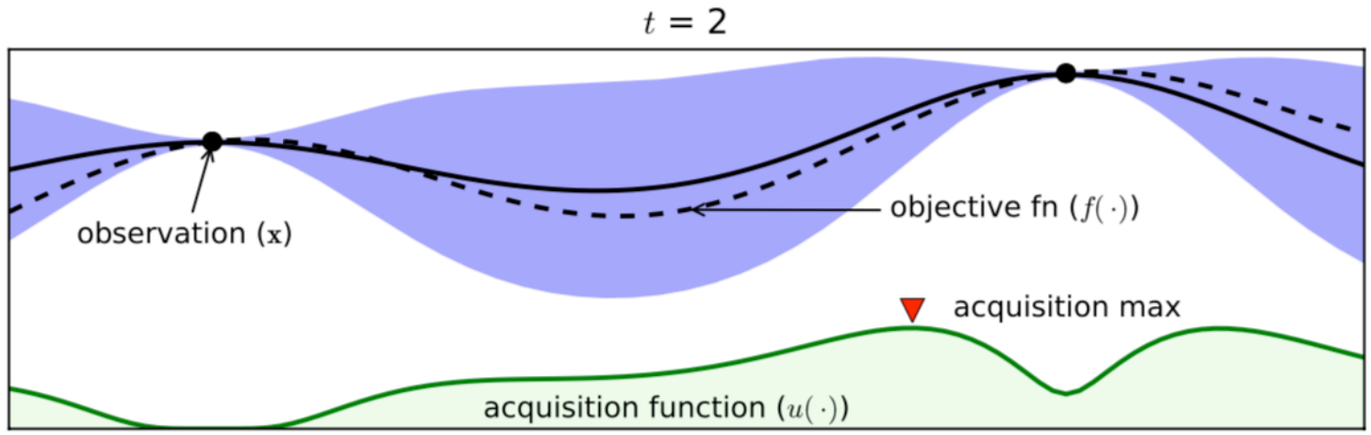
ベイズ最適化(Bayesian Optimization)

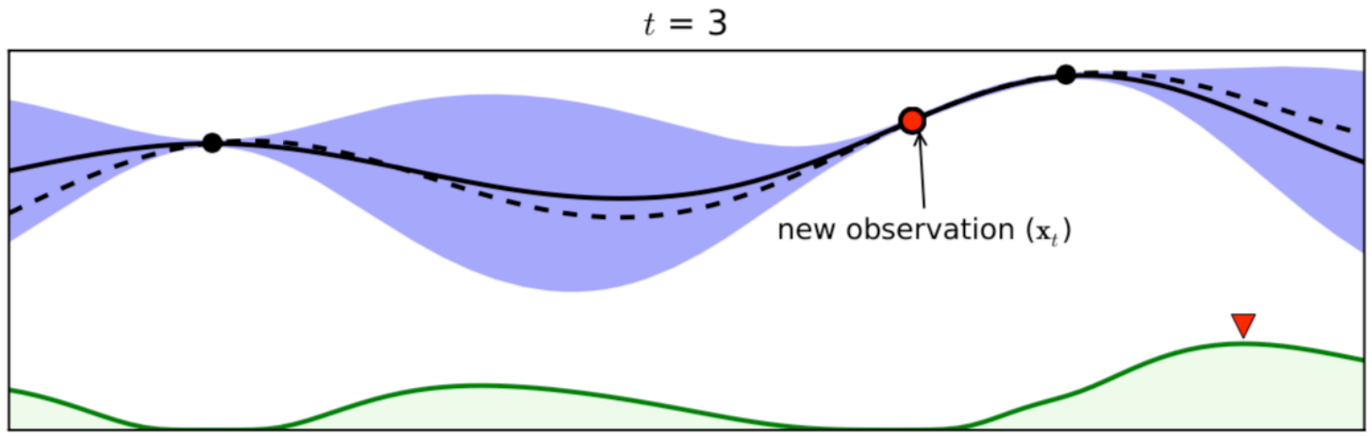
- f(x) の評価が難しい（コスト高）であるときに特に有用な最適化手法
- 目的関数 f(x) を、計算しやすいモデル関数 g(x) で近似する
  - x_i を数点適当にサンプルし、 $y_i=f(x_i)$ を計算
  - (x_i, y_i) のセットを用いて g(x) を学習する（フィッティングする）
  - g(x) を最小化・最大化するような点 x' を次の観測点として、 y'=f(x') を計算、学習データに追加して g(x) を再学習する
  - 適当な回数繰り返す
- 一般的にはg(x) として、ガウス過程を事前分布とした事後分布を用いる
  - g(x) という関数の分布をベイズ推定するという意味で「ベイズ」最適化
- 実際にはg(x) そのものではなく、 g(x) の期待値・標準偏差から計算される獲得関数（スコア関数）を最小化・最大化する
  - 「利用」と「探索」のトレードオフ

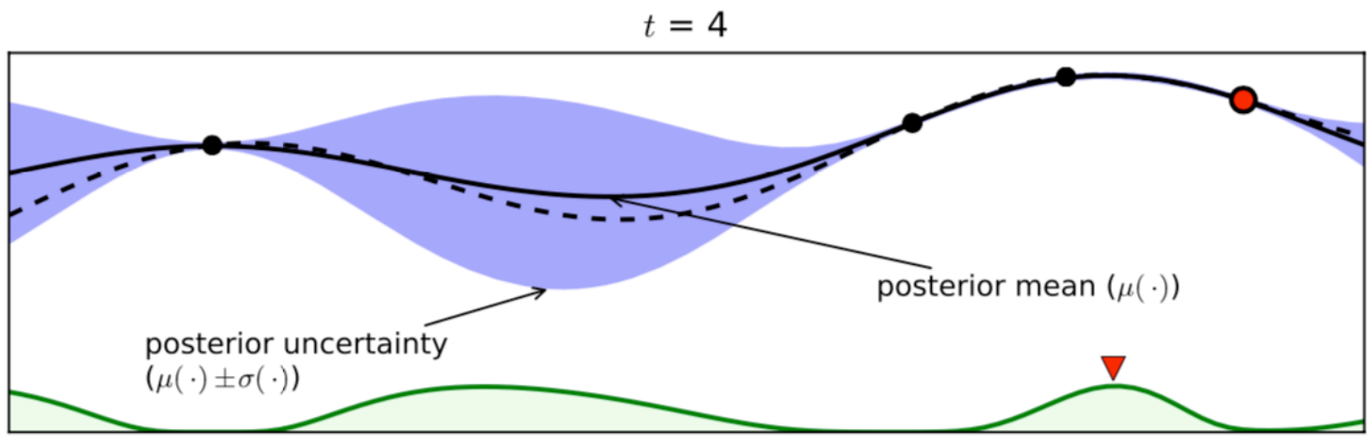
ベイズ最適化の流れ

taken from E. Brochu, et al., arXiv:1012.2599

- 破線が真の目的関数（未知）
- 黒丸・赤丸が測定値
- 実線が推定の期待値
- 青が期待値の不確かさ
- 緑が獲得関数
- 三角が獲得関数の最大
- 獲得関数は、 g(x) の期待値と分散とから定義される、候補点の「スコア」







獲得関数(acquisition function)

- 事後分布 g(x) の期待値 $\mu(x)$ と分散 $\sigma^2(x)$ から定義される、点 x を測定すると「どのくらい最適化が進むか（嬉しいか）」を表す関数
  - PHYSBO では score とも呼ばれる
  - コレを最大化するような x を次に測定すれば良い
    - PHYSBO ではこの最大化問題を簡単にするために x の候補を離散集合で与え、全探索する
- 単目的関数最適化においてPHYSBO で定義されているのは次の3つ
  - PI: これまでの最適値を更新する確率
  - EI: これまでの最適値がどれくらい更新されるかの期待値
  - TS: 事後分布g(x) から実際にサンプルした値

ガウス過程

- 関数 f(x) について、n 個の任意の点 {x_1, x_2, ..., x_n} に対する値の同時確率分布が次のようなn 次元の正規分布となるとき、f(x) はガウス過程 GP(m,k)である

$$\begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix} \sim N\left(\begin{pmatrix} m(x_1) \\ m(x_2) \\ \vdots \\ m(x_n) \end{pmatrix}, \begin{pmatrix} k(x_1,x_1) & k(x_1,x_2) & \dots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \dots & k(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_n,x_1) & k(x_2,x_n) & \dots & k(x_n,x_n) \end{pmatrix} \right)$$

  - とても大雑把には、n 次元の正規分布を n→∞ に飛ばした極限
  - 関数を∞ 次元のベクトルだとみなせるようなもの
- 以下では分散共分散行列をK と書く
- k をカーネル関数と呼ぶ

ガウス過程回帰

- n 組の既知データから、別の座標x* に対する未知データy*の値を推測したい
- 関数 y = f(x) が、あるカーネル関数 k を持つガウス過程であると仮定すると、y* の平均μ_c及び分散σ_cは以下のように計算できる

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

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

  - ここで**y** は既知データ $\mathbf{y} = [y_1, y_2, \ldots, y_n]^T$
  - **k** はカーネルの組 $\mathbf{k}(x^*) = [k(x^*, x_1), k(x^*, x_2), \ldots, k(x^*, x_n)]^T$
  - σ は回帰のハイパーパラメータ（測定にかかる誤差の大きさ）
- 計算量はO(n^3) （逆行列の計算）
  - x* にはよらないので、一度やれば使い回せる

カーネル関数

- カーネル関数k 自体の選択も重要
  - 2点がどのくらい「近い（＝互いに影響を与える）」か
- PHYSBO ではガウスカーネルを用いる
  - Squared Exponential (SE) とかよばれることも
- random feature map （次ページ）がわかっているためコレをつかっている

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

- η はカーネル関数をチューニングするハイパーパラメータ
  - 大きいほど距離に鈍感
  - 回帰のパラメータ σ ともども、実行中にチューニング可能
    - PHYSBO は最尤法を用いている

特徴空間への写像ϕ

- 次のようなパラメータ付けられた関数z を考える

$$z_{\vec{\omega},b}(\vec{x}) = \sqrt{2}\cos(\vec{\omega}\cdot\vec{x}+b)$$

- ここで ωはD 次元正規分布 N(0, I_D) に従う確率変数で、b は一様分布 [0, 2π) に従う確率変数

- zの積について、ωとb に関する期待値はガウスカーネルに一致する

$$E\left[z_{\vec{\omega},b}(\vec{x})z_{\vec{\omega},b}(\vec{x}')\right]_{\vec{\omega},b} = \exp\left[-\frac{1}{2\eta^2}\|\vec{x}-\vec{x}'\|^2\right] = k(\vec{x},\vec{x}')$$

- x からM 次元特徴空間への次の写像ϕを考えると

$$\vec{\phi}(\vec{x}) = \left[z_{\vec{\omega}_1,b_1}(\vec{x}/\eta),\ldots,z_{\vec{\omega}_M,b_M}(\vec{x}/\eta)\right]^T$$

- ϕの内積でガウスカーネルを近似可能（M→∞ 極限で厳密 ∵大数の法則）

$$k(\vec{x},\vec{x}') \simeq \vec{\phi}(\vec{x})^T\vec{\phi}(\vec{x}')$$

ベイズ線形回帰

- ϕがガウスカーネルを（近似的に）生成するので、もともと考えていたガウスカーネルガウス過程回帰は次のϕによるベイズ線形回帰の双対となる

$$y = \vec{w}^T \vec{\phi}(\vec{x})$$

- 学習データD のもとで、係数ベクトルw の事後分布は次のガウス分布になる

$$p(\vec{w}|D) = N(\vec{\mu}, \Sigma)$$

$$\vec{\mu} = \left[\Phi\Phi^T + \sigma^2 I\right]^{-1} \Phi\vec{y}$$

$$\Sigma = \sigma^2 \left[\Phi\Phi^T + \sigma^2 I\right]^{-1}$$

$$\Phi = \left(\vec{\phi}(\vec{x}_1), \ldots, \vec{\phi}(\vec{x}_n)\right)$$

トンプソンサンプリング

- 事後分布に従って係数ベクトルを一つサンプリングして w* とする
- トンプソンサンプリングにおいて、x における獲得関数 TS(x) は、 y の事後分布からのサンプリング値 y* になるが、これは次の単純な形になる

$$y^* = \vec{w}^* \cdot \vec{\phi}(\vec{x})$$

- ひとたび w* を決めてしまえば、獲得関数の計算がO(M) でできる

w* のサンプリング

- w の事後分布を簡単に書くために、次のM次元対称行列A を導入する

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

- w の事後分布は　$p(\vec{w}|D) = N(\frac{1}{\sigma^2}A^{-1}\Phi\vec{y}, A^{-1})$
- Aの逆行列計算はnaive にはO(M^3)
- データが増えた場合、A の更新はΦ に列が増えることによってなされる

$$A' = A + \frac{1}{\sigma^2}\vec{\phi}(\vec{x}')\vec{\phi}(\vec{x}')^T$$

- あらかじめA をコレスキー分解 ($A=L^T L$)しておくことでA^{-1} の計算コストが $O(M^2)$ になる