

# ベイズ最適化パッケージ PHYSBO の使い方

<https://github.com/issp-center-dev/PHYSBO>

<https://issp-center-dev.github.io/PHYSBO/manual/master/ja/index.html>

[physbo-dev@issp.u-tokyo.ac.jp](mailto:physbo-dev@issp.u-tokyo.ac.jp)

東大物性研 本山裕一

# PHYSBO のインストール

- 最新版 (v1.0.1) は Python3.6 以上が必要
  - v0.1.0 は Python2.7 で動作 (COMBO +bugfix +document)
- `pip install physbo` でインストール可能
  - インストール方法については 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==1.0.1`
    - 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 の使用法（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)(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: -0.0023413821289715096 at [0.00970778 0.01407324
0.04526682]
# 乱数を固定していないので数字は異なる
```

# 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

# 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 (100 回)
policy.bayes_search(
    max_num_probes=100, 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

# 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（100 回）
policy.bayes_search(
    max_num_probes=100, 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

# 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 (100 回)
policy.bayes_search(
    max_num_probes=100, 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
- 100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

# 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 (100 回)
policy.bayes_search(
    max_num_probes=100, 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

# 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（100 回）
policy.bayes_search(
    max_num_probes=100, 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

```
$ python3 simple.py
0001-th step: f(x) = -5.936448 (action=631)
current best f(x) = -5.936448 (best action=631)
(中略)
```

```
0010-th step: f(x) = -0.944487 (action=117)
current best f(x) = -0.226037 (best action=64)
```

- `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

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

```
Start the hyper parameter learning ...
0 -th epoch marginal likelihood 18.1115054749814
(中略)
```

```
500 -th epoch marginal likelihood 16.661922004077017
Done
```

- その後にベイズ最適化

```
0011-th step: f(x) = -0.017750 (action=452)
current best f(x) = -0.017750 (best action=452)
(中略)
```

```
0110-th step: f(x) = -0.336775 (action=112)
current best f(x) = -0.017399 (best action=404)
```

- 最終結果

```
best_fx: -0.017399043740363704 at [-0.10011779  0.06464974  0.05653215]
```

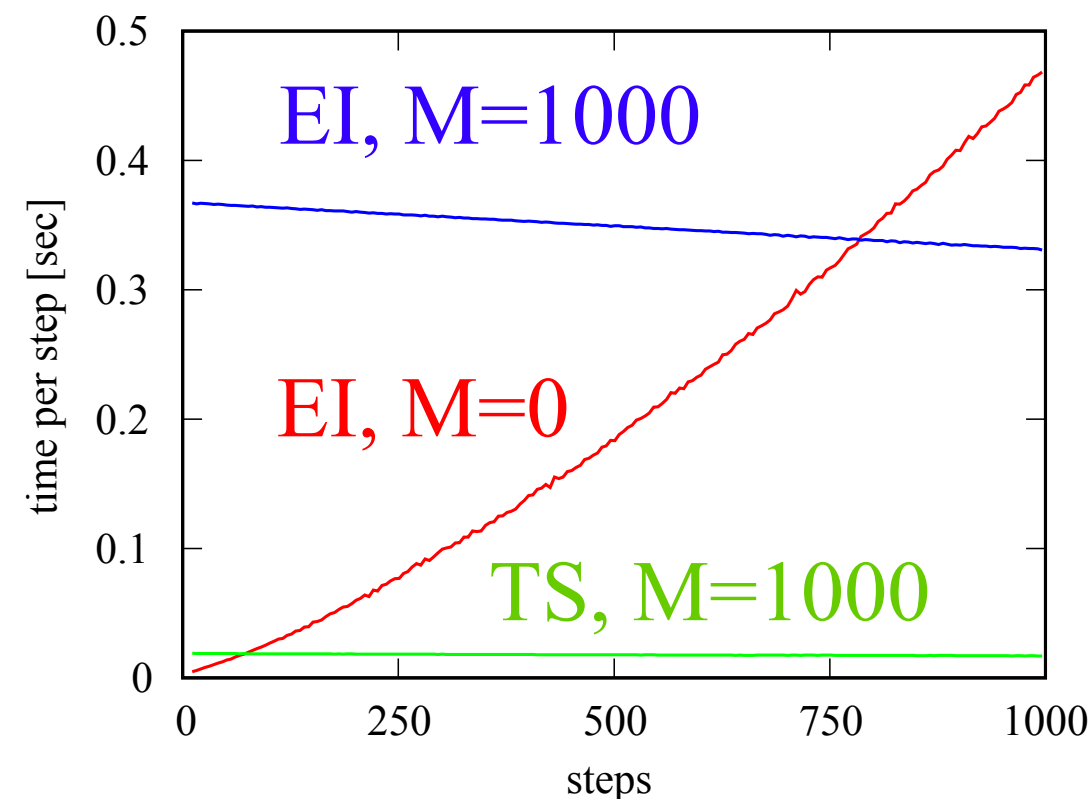
- 「候補点の集合からの最適解」であることに注意

# PHYSBO の使用法 (Advanced)

- 目的関数の評価は自分でやりたい
  - チュートリアル「[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
  - 例：実験で測定したい
  - 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 (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

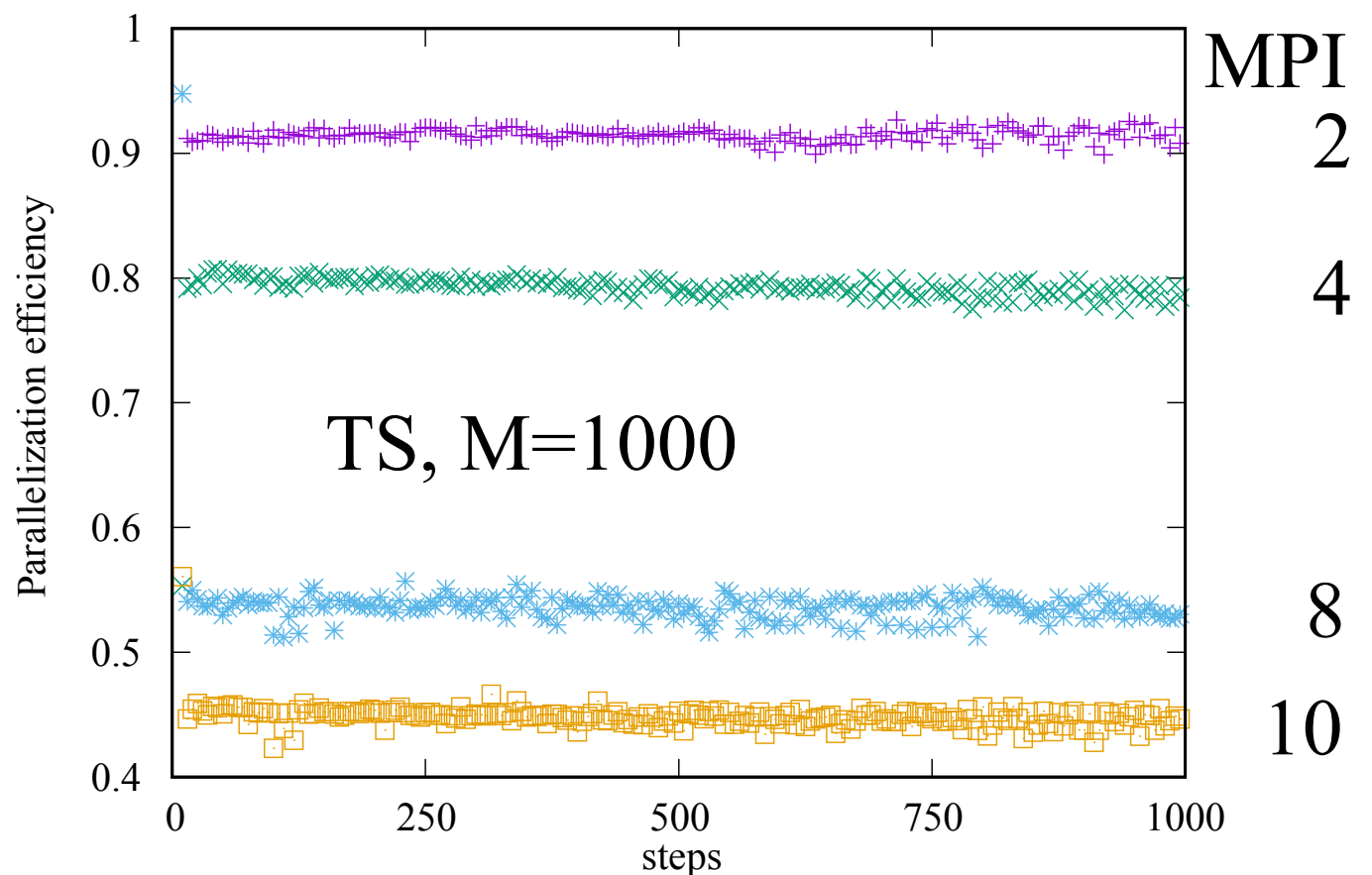
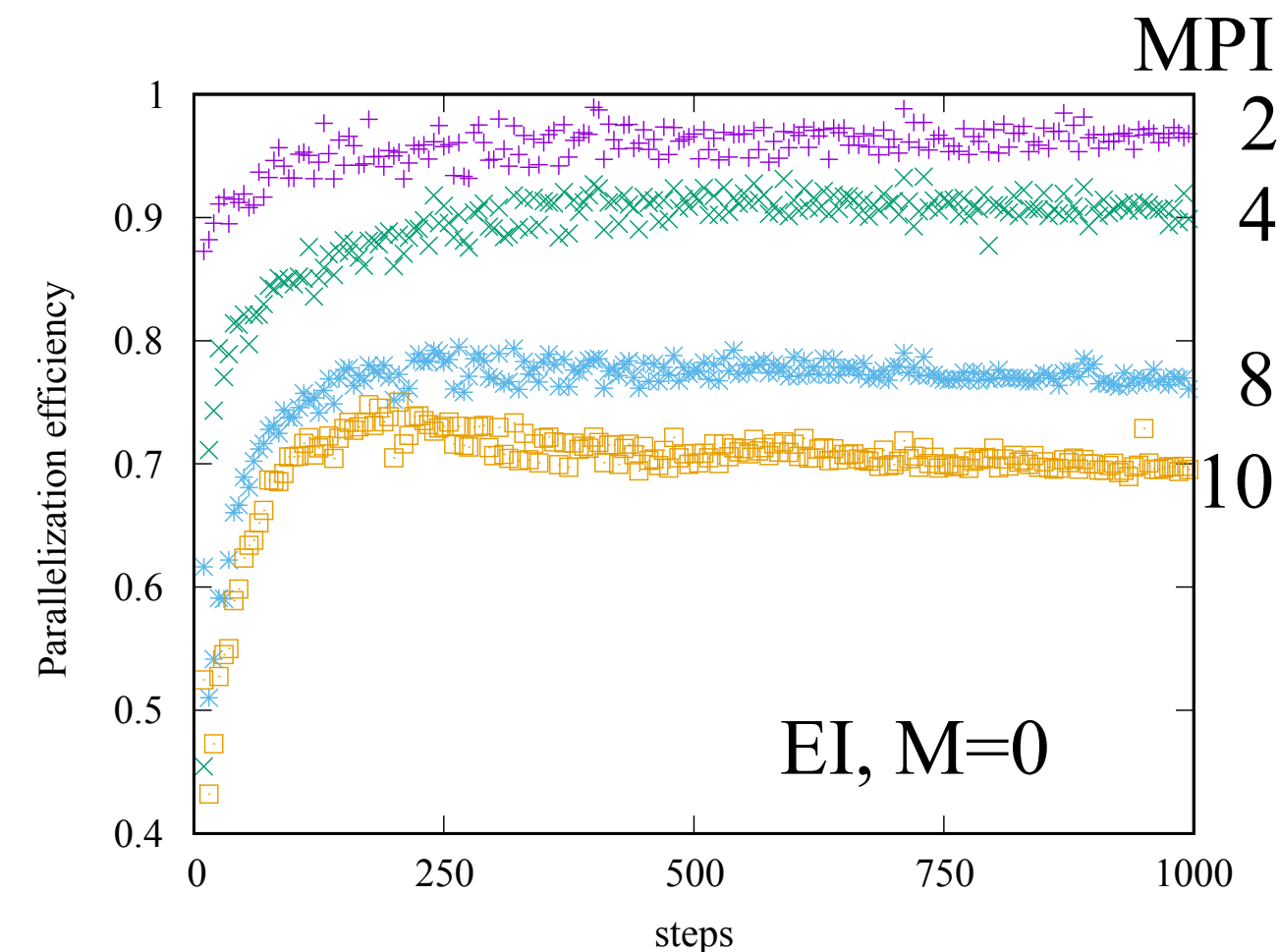


# PHYSBO の使用法 (Advanced)

- 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コミュニケーター `mpi4py.MPI.Comm` を渡す
  - `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

# 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



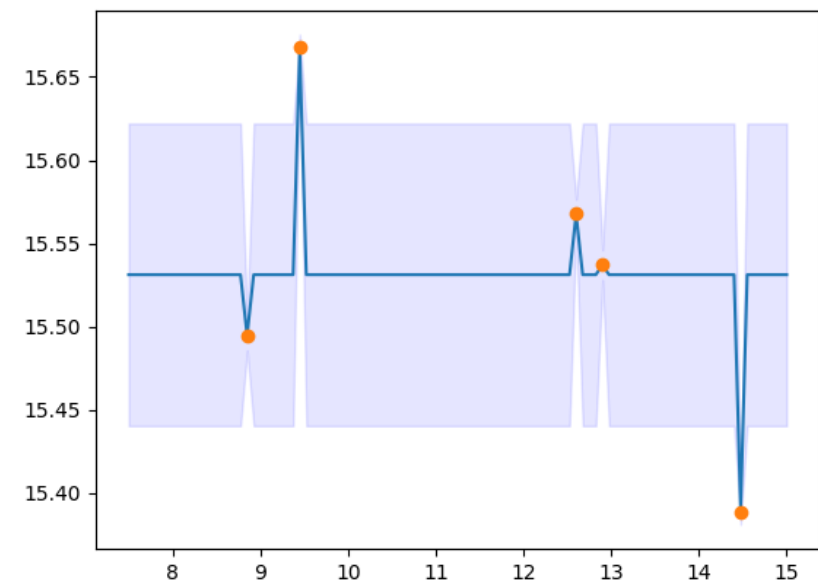
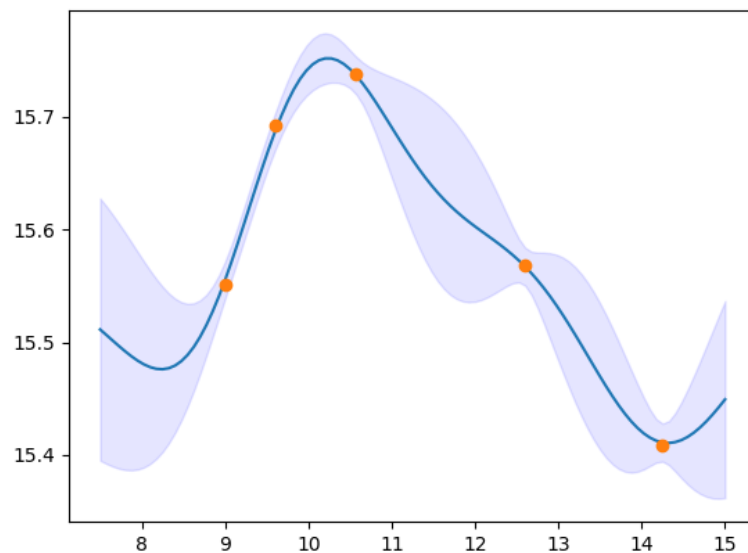
# PHYSBO の使用法 (Advanced)

- 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「[PHYSBO の基本](#)」



# PHYSBO の使用法 (Advanced)

- ハイパーパラメータの学習
  - ガウスカーネルの幅  $\eta$  と測定にかかる誤差  $\sigma$  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)
    - 初期データが少なく、偏っている場合、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
    - 右の状態では恒等関数しか表現できない

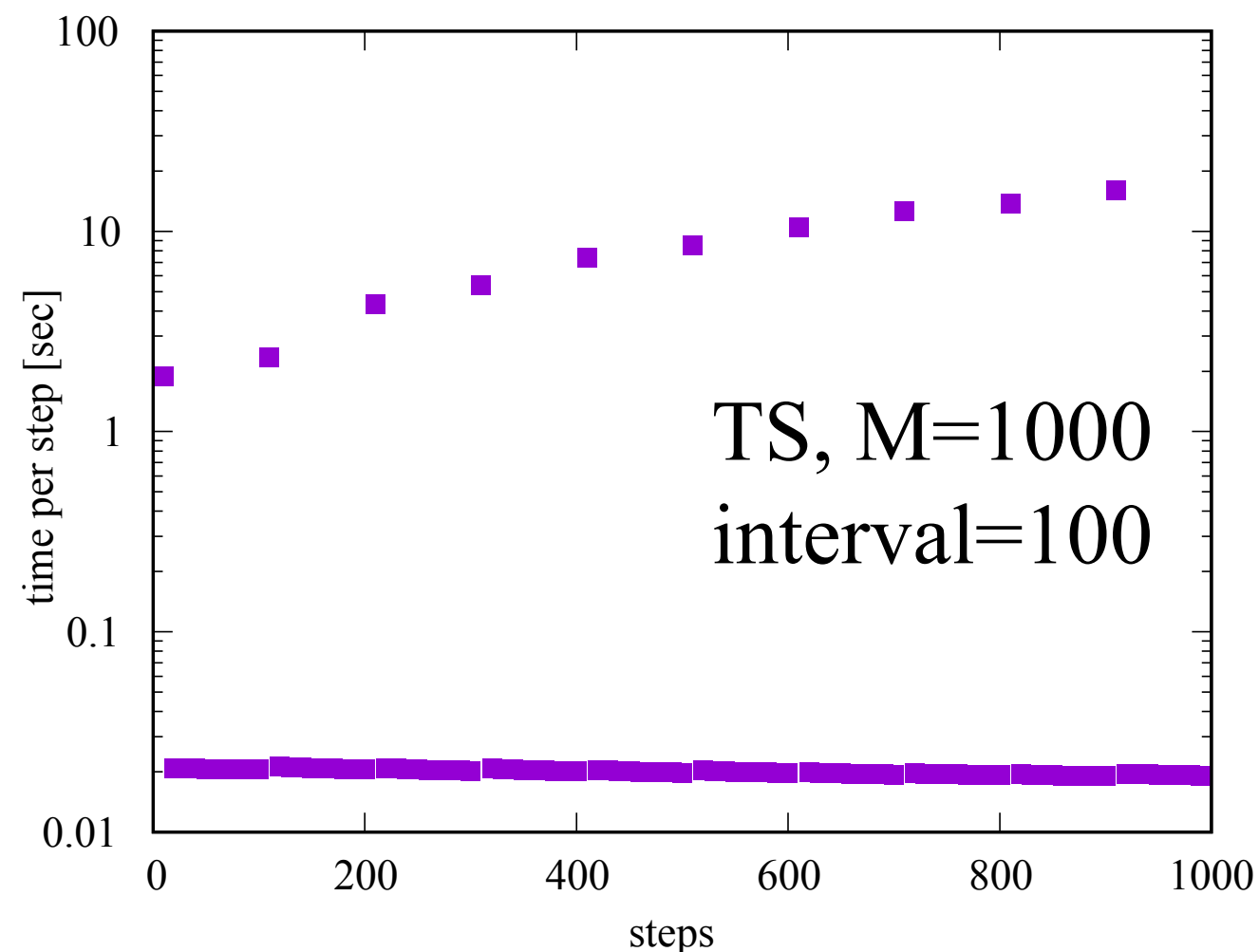


# PHYSBO の使用法 (Advanced)

- 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
- v1.1 では安全のため、デフォルト値を 1 に変更する予定
  - つまり各ステップで毎回再学習する

# 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
  - UNIX の`time` コマンドやPython の`time.time` 関数を使ってみましょう
  - 今回のスライドにあるような細かい経過時間の出力は将来バージョンで入ります
- 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）
  - 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

ガウス過程  
ガウスカーネル  
線形ベイズ回帰

# ガウス過程

- 関数  $f(x)$  について、 $n$  個の任意の点  $\{x_1, x_2, \dots, x_n\}$  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

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

- とても大雑把には、 $n$  次元の正規分布を  $n \rightarrow \infty$  に飛ばした極限
- 関数を  $\infty$  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  $\mu_c$  及び分散  $\sigma_c$  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  $\mathbf{y}$  は既知データ  $\mathbf{y} = [y_1, y_2, \dots, y_n]^T$
- $\mathbf{k}$  はカーネルの組  $\mathbf{k}(x^*) = [k(x^*, x_1), k(x^*, x_2), \dots, k(x^*, x_n)]^T$
- $\sigma$  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

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

- $\eta$  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  $\sigma$  とともに、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 $\phi$

- 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  $\omega$  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  $[0, 2\pi)$  に従う確率変数
- $z$  の積について、 $\omega$  と  $b$  に関する期待値はガウスカーネルに一致する

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

- $x$  から  $M$  次元特徴空間への次の写像  $\phi$  を考えると

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

- $\phi$  の内積でガウスカーネルを近似可能 ( $M \rightarrow \infty$  極限で厳密  $\because$  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

- $\phi$  がガウスクアーネルを（近似的に）生成するので、もともと考えていたガウスクアーネルガウス過程回帰は次の  $\phi$  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

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

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

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

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

# 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  $O(M)$  ができる

# $w^*$ 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 $\Phi$  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