ベイズ最適化パッケージ
PHYSBO

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

physbo-dev@issp.u-tokyo.ac.jp

東大物性研　**本山裕一**、**吉見一慶**、**川島直輝**

物材機構・東大新領域　**田村亮**

横市大　**寺山慧**

Magne-Max Capital Management **植野剛**

東大新領域　**津田宏治**

2021-02-22 物性研ワークショップ「物性科学におけるデータ科学の今と未来」

目次 (クリックするとジャンプします)

最適化問題

- 研究・開発では（以外でも）たくさんの最適化問題がある
  - 計算結果と実験との違いを最小化するような理論・モデルのパラメータを知りたい
  - 材料の組成・実験の条件を変えて、物性値（磁化・転移温度・etc）ができるだけ良いものを探したい

$x^* = \mathrm{argmax}_{\vec{x}} f(\vec{x})$　　本発表ではf(x) を目的関数と呼ぶ

- コレを数値的に解く（最大・最小のf を与えるxを探す）アルゴリズムはたくさんある
  - 勾配法
  - （古典・量子）アニーリング
  - **ベイズ最適化**
  - etc...

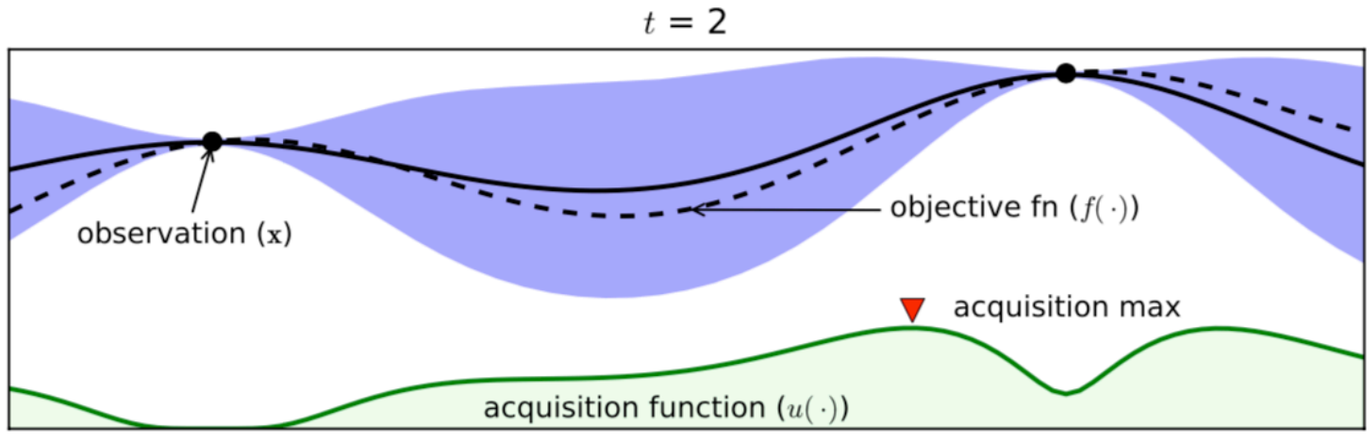
ベイズ最適化 (Bayesian Optimization, BO)

- 目的関数 f(x) の評価が高コスト（時間、資金、etc）なときに有効
  - できるだけ試行回数を減らしたい
  - できるだけ過去の結果$(\vec{x}_i, f(\vec{x}_i))$ を利用したい
- 過去の結果を用いて、ブラックボックスである fを、より扱いやすいモデル関数 gで推定する
  - g を解析し、次にf を評価すべき点を提案する
- モデル関数g はガウス過程に従うと仮定する（→付録）
  - 任意のx でg(x)の期待値及び分散（期待値の不確かさ）が計算可能
    - 期待値と分散の組み合わせ（獲得関数）を用いて次の点を提案する

BO の流れ

- 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

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



獲得関数

- 期待値と分散から、「次にこの点を計算すると良さそうだよ」というスコアを計算するのが獲得関数
- 以下の**活用**と**探索**をバランス良く取り入れた獲得関数が重要
- 期待値が一番大きい場所を選ぶ（**活用**）のが一つの選択肢
  - 良い結果が得られる可能性は高い
  - 同じような場所ばかり選ばれがち、情報量が少なくなる
- （期待値の）分散が大きい＝不確かさが大きい場所を選ぶ（**探索**）のがもう一つの選択肢
  - 大雑把にはこれまで調べたことのない領域
  - 測定したときに得られる情報量が大きい
  - 見当違いのところを探索することもある

獲得関数の例

ガウス過程ではすべて計算可能！

- Probability of Improvement (PI)
  - 最高記録を更新する確率　$\mathrm{Prob}[g > g_{\mathrm{best}}]$
- Expected of Improvement (EI)
  - 最高記録の更新幅の期待値　$\mathrm{E}\left[\max\left(0, g - g_{\mathrm{best}}\right)\right]$
- Upper Confidence Bound (UCB)
  - 期待値μと不確かさσの加重平均　$\mu + \kappa\sigma$
    - ハイパーパラメータκを含む
- Thompson sampling (TS)
  - 事後分布 g(x) からサンプリングした生の値　$g \sim N(\mu, \sigma^2)$

目次（クリックするとジャンプします）

PHYSBOの概要

- ベイズ最適化パッケージとして、PHYSBO (optimization tool for **PHYS**ics based on **B**ayesian **O**ptimization)を開発・公開している
  - https://github.com/issp-center-dev/PHYSBO
  - COMBO (**COM**mon **B**ayesian **O**ptimization library) のフォーク
    - https://github.com/tsudalab/combo
- ライセンスは GNU GPL v3 (COMBO は MIT)
- 物性研ソフトウェア高度化プロジェクト2020 年度課題
  - ドキュメント拡充
  - Python3 対応
  - 機能追加（並列化、多目的最適化機能の移植など）

PHYSBOの特徴

- データ数に依存しない速度でガウス過程回帰を実行可能
  - ガウスカーネルを生成する特徴空間への写像ϕを用いた線形ベイズ回帰を用いて計算する
- 獲得関数の計算および次候補選定を高速に実行可能（→参考文献・付録）
  - トンプソンサンプリング
  - コレスキー分解を用いたfast update
  - MPI を用いた自明並列化
- 複数の目的関数の同時最適化（多目的最適化）が可能

PHYSBO のインストール

- 最新版 (v0.3.0) 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==0.3.0`
    - 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 （現在の最新） 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

PHYSBO の使用法 (Basic)

- 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
  - 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[int(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[int(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[int(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[int(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、それまでの最適解を出力する
- Tips
  - 最後のステップは -1 で取れる
  - best_actions はfloat を返すのでint に直す
    - 次のバージョンまでに修正します

```
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[int(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

```
$ python3 example.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)

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

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)

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

PHYSBO の使用法 (Advanced)

- 目的関数の評価は自分でやりたい
  - 例：実験で測定したい
  - bayes_search に simulator を渡さない場合、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)`
  - 既知の測定結果（候補点番号と目的関数の組）を初期化時に渡しておく
    - `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）
  - 公式ドキュメントを参照

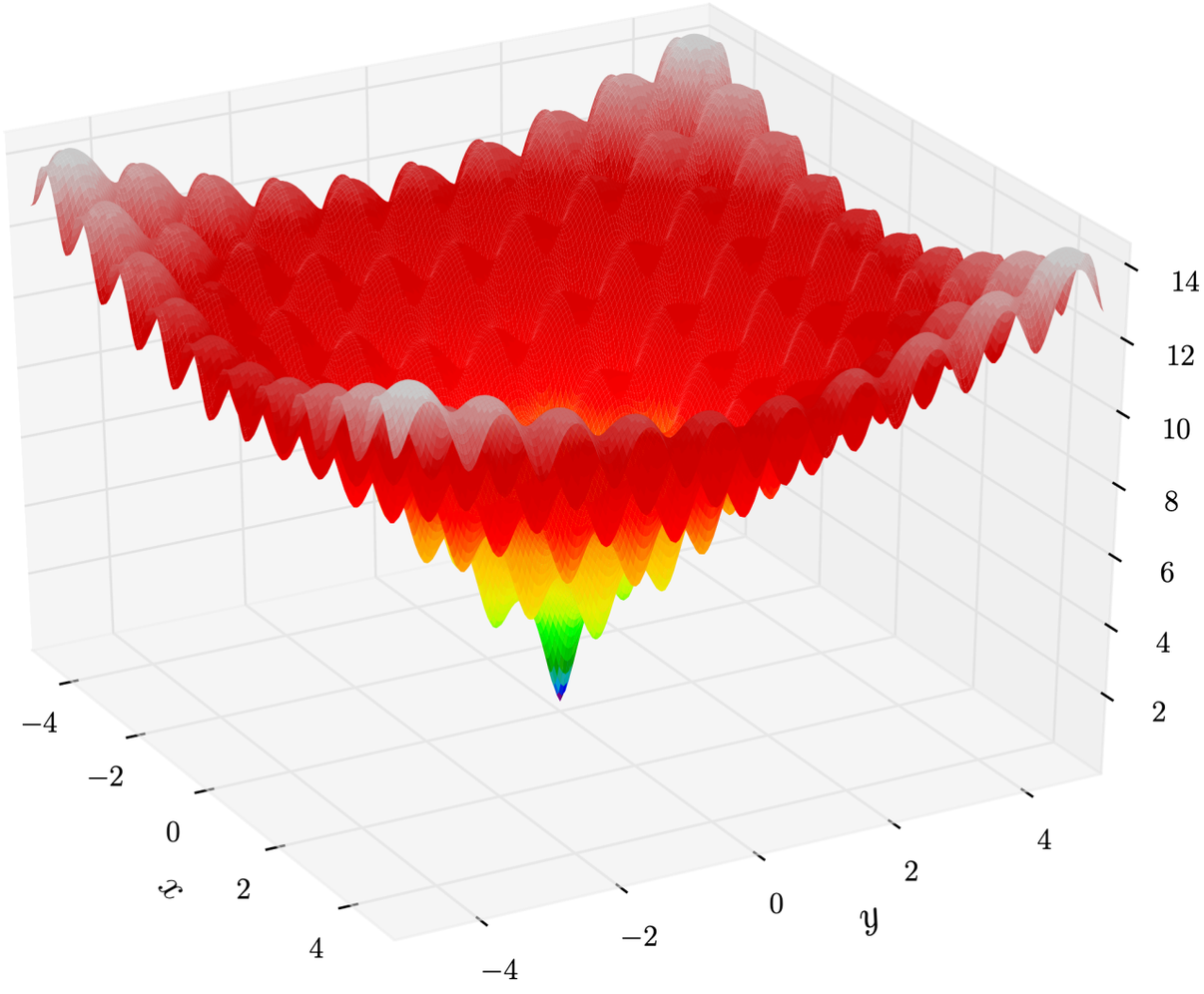
PHYSBO の使用法 (Advanced)

- 高速化したい
  - 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
      - `physbo.search.discrete.policy(test_X, comm=MPI.COMM_WORLD)`
  - 現状では、目的関数の計算は rank==0 のものだけが行う
- random feature mapを用いる
  - policy.bayes_search(num_rand_basis=1000)
    - デフォルトは0 （「普通の」ガウス過程回帰）
  - 測定済みデータが多い場合に有効

目次（クリックするとジャンプします）

ベンチマーク：セットアップ

- 問題のセットアップ
  - 目的関数は Ackley 関数
    - ベンチマーク用の関数
    - 大域最適解 (0,0,...,0) の周りに
      大量の局所最適解
  - 探索パラメータ空間
    - 2次元
    - 探索範囲は両次元とも [-5,5]
- 測定するもの
  - ステップあたりの時間
  - 開始からの時間
  - 目的関数の計算は獲得関数の計算と比較して非常に軽いので、計算時間は無視できる
  - 計算全体のフレームワークとしてはpy2dmat を用いている



taken from wikimedia commons (CC BY-SA 3.0)

ベンチマーク：セットアップ

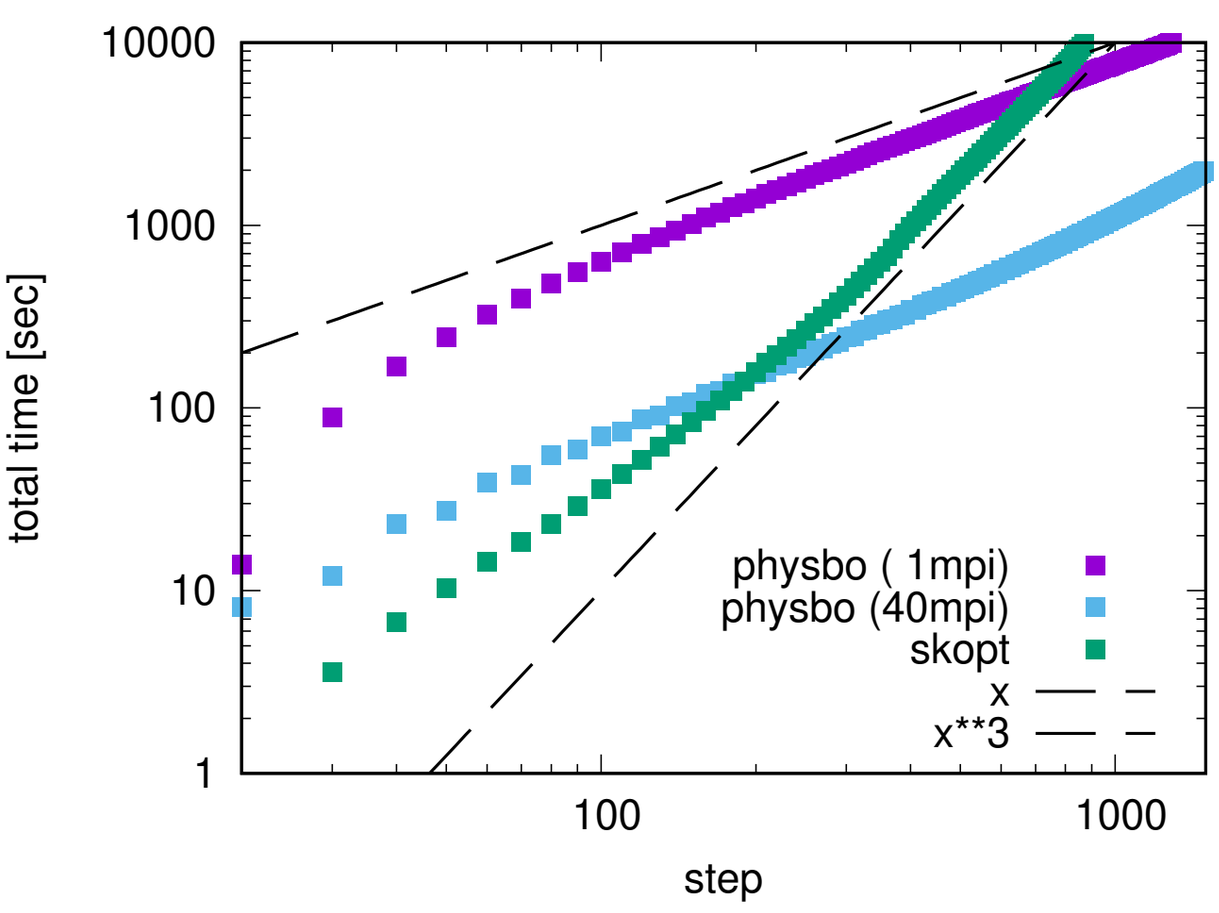
- PHYSBO
  - 双対基底の数は M = 5000
  - 獲得関数
    - Thompson Sampling (TS)
      - 計算量は O(M)
    - Expected Improvement (EI)
      - O(M^2)
- 20個のランダムサンプルからスタート
  - 101x101=10201 個のグリッド点で獲得関数を計算し、最良のものを選ぶ
    - 逐次計算
    - MPI40プロセスによる自明並列
- 20回の suggestion ごとにハイパーパラメータを更新（再学習）

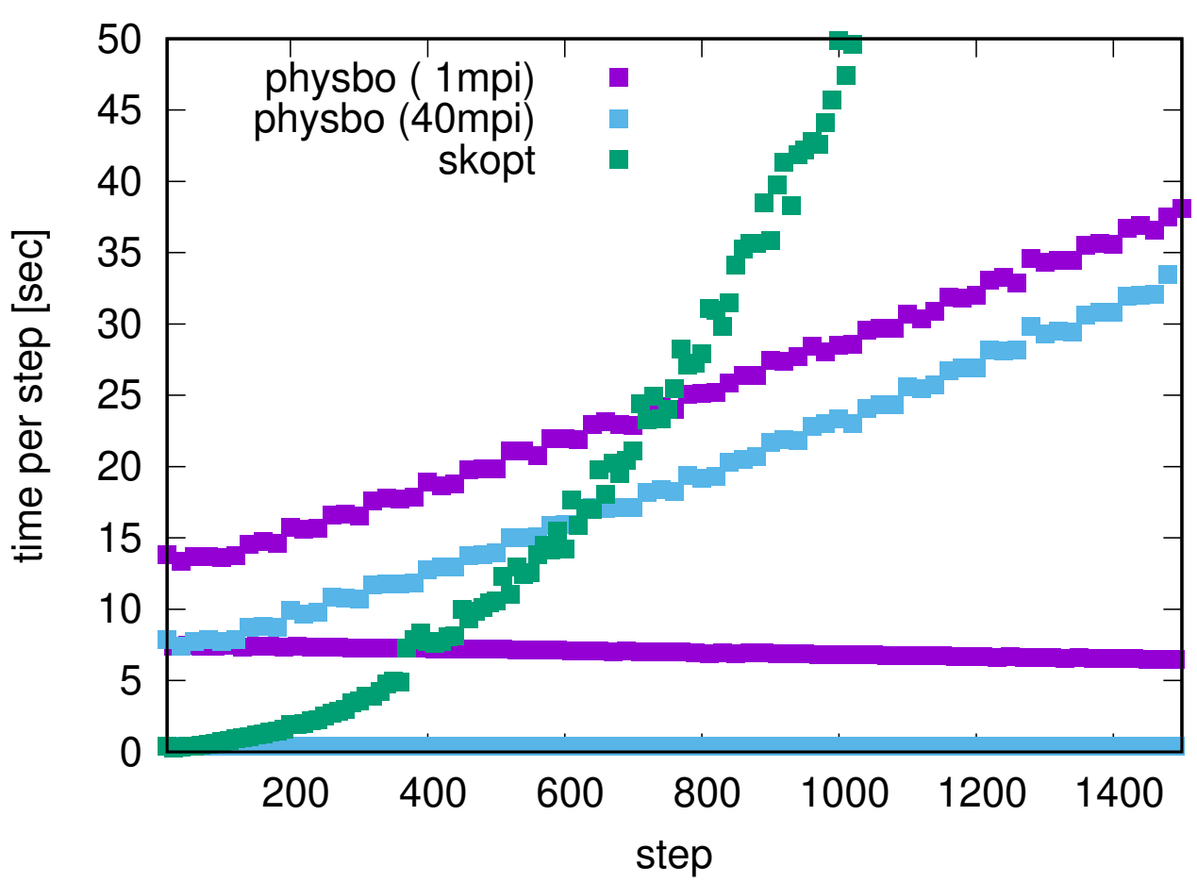
ベンチマーク：セットアップ

- Scikit-optimize (skopt)
  - https://scikit-optimize.github.io/stable/
  - scikit-learn をベースにしたライブラリで、BO を含む
  - 獲得関数
    - EI
      - 他にもいくつかあり、デフォルトも別だが、 PHYSBO との比較のためにEI を利用する
- 20 個のランダムサンプルからスタート
- 準ニュートン法 (L-BFGS) を用いて獲得関数を最適化する

ベンチマーク1: PHYSBO vs skopt

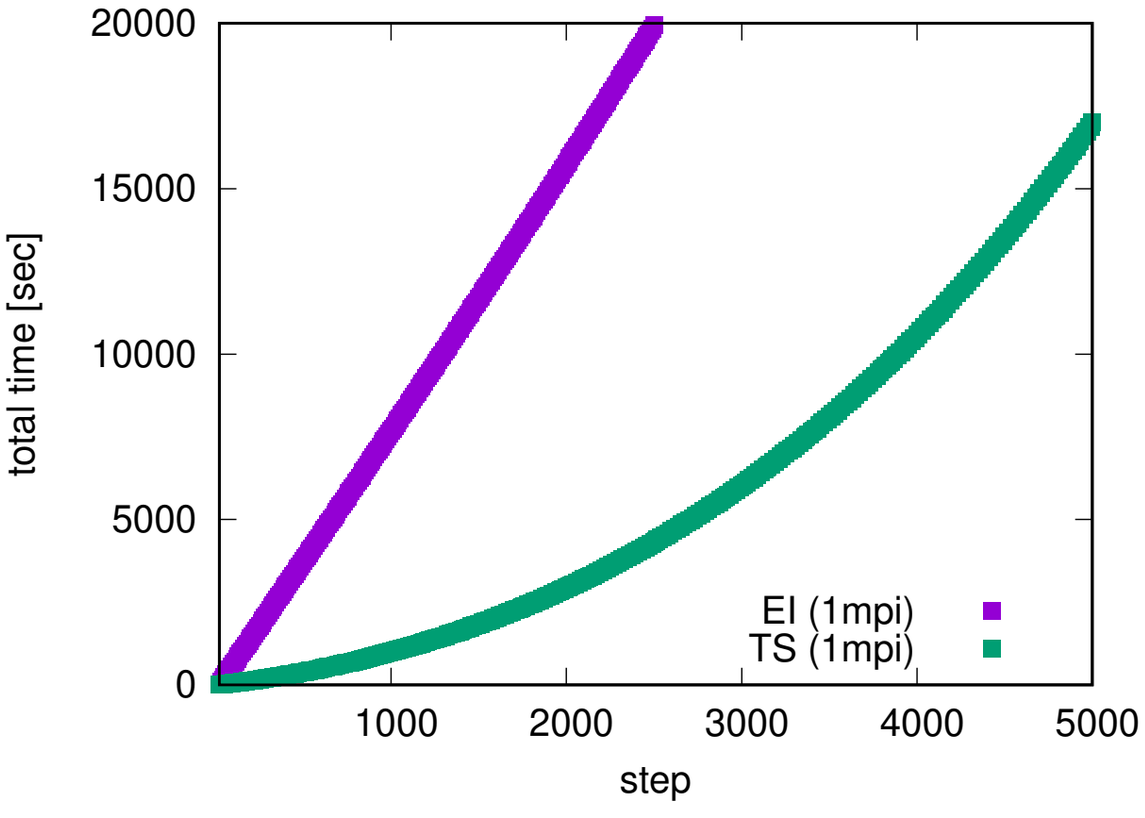
- （左）開始からの経過時間と（右）ステップあたりの時間
  - （軽量化のためプロット点を1/10 に間引いてある）
- ステップあたりの時間で、PHYSBO の結果が2つあるように見えるのは、20ステップごとに行っているハイパーパラメータ学習が含まれているため
- 最初はskopt が早いが、ステップ数 t を増やすとオーダーの違いにより逆転

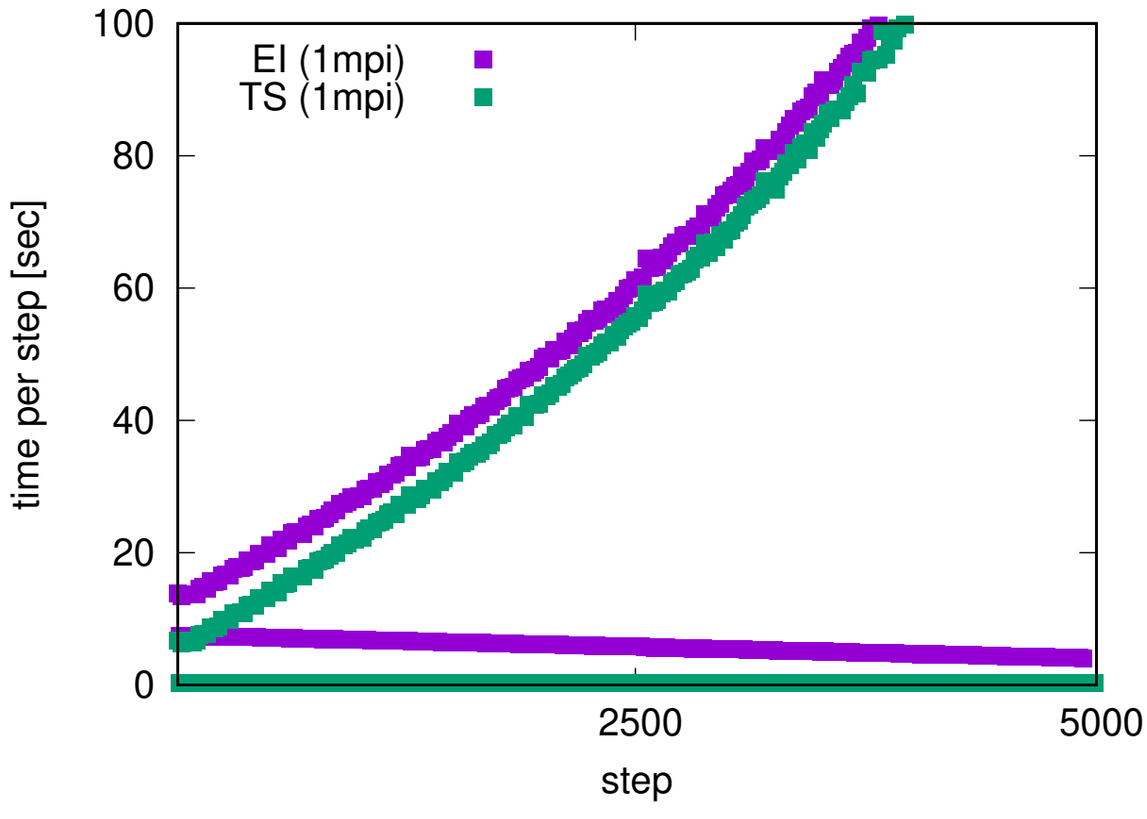




ベンチマーク2: EI vs TS

- （左）開始からの経過時間と（右）ステップあたりの時間
  - （軽量化のためプロット点を1/10 に間引いてある）
- ステップあたりの時間で、結果が2つあるように見えるのは、20ステップごとに行っているハイパーパラメータ学習が含まれているため
- TS は普段のステップが早すぎて、ハイパーパラメータ学習の時間 O(t)が律速になる
  - 累計時間がO(t^2) になってしまう（が、それでも早いので問題はない）
  - EI もそうなると思うが、たどり着くまでにかなりの時間がかかる

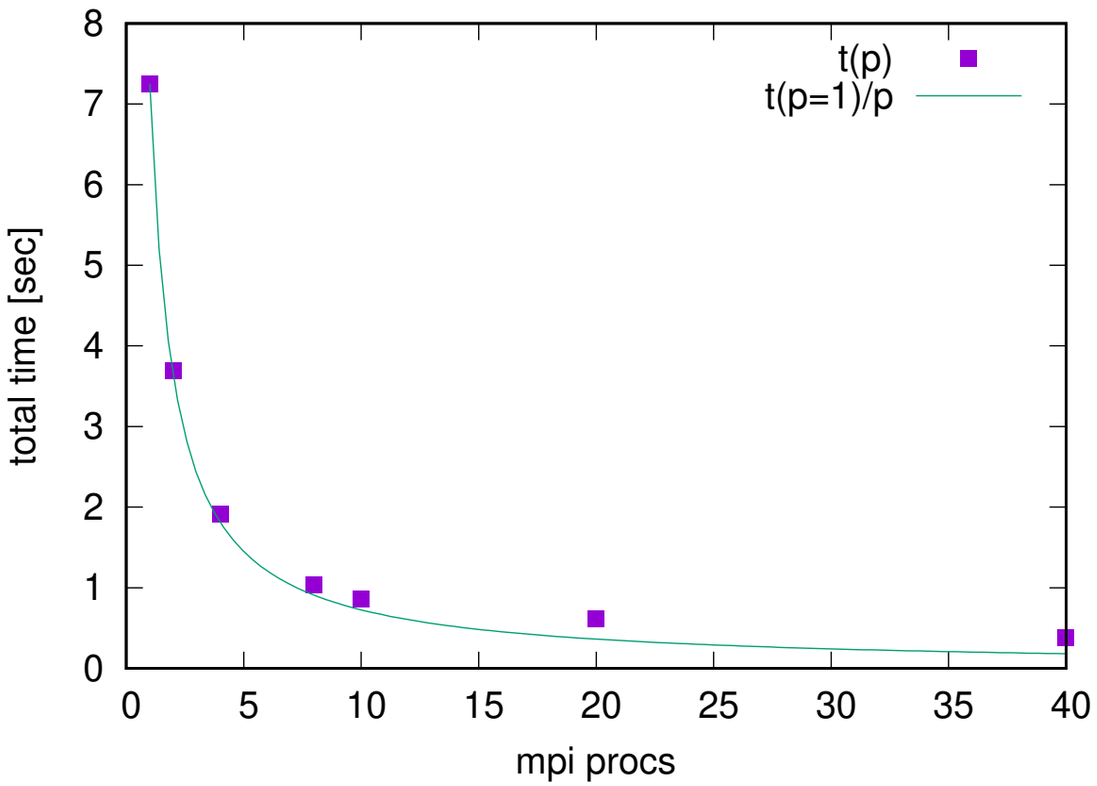




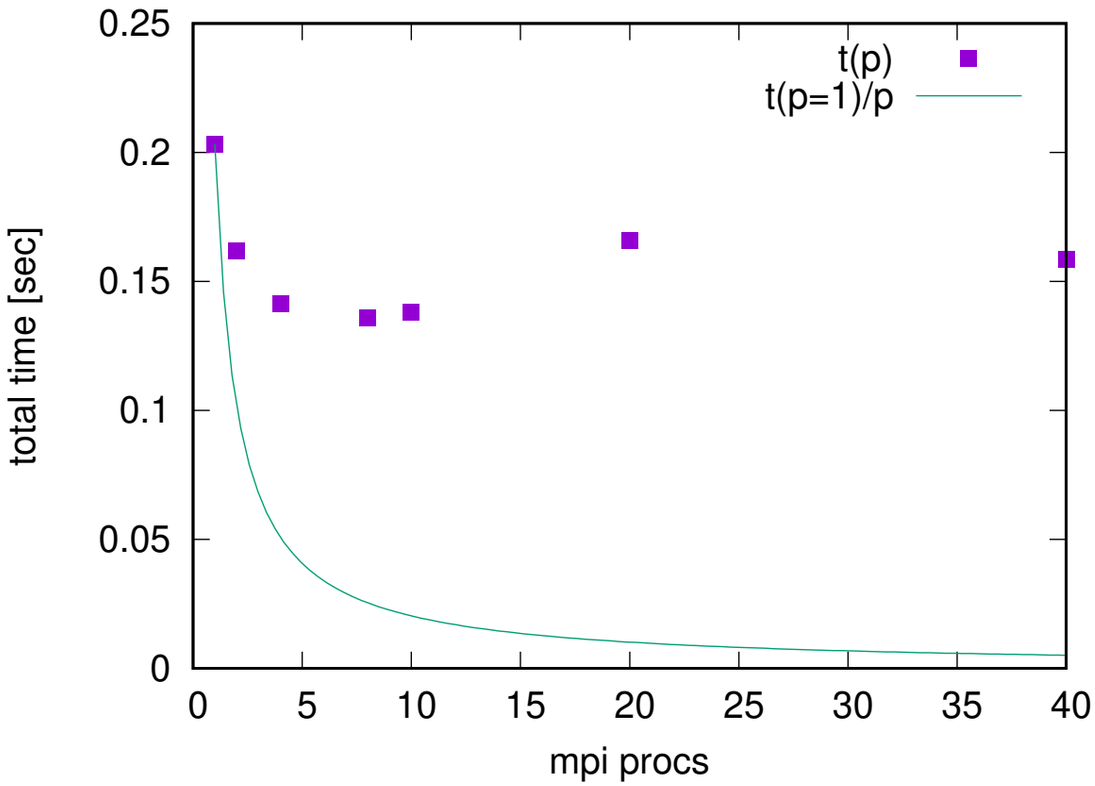
ベンチマーク3: 並列化

- 次に測定すべき点を見つけるために、全グリッド点で獲得関数を計算している
  - この部分はMPI による自明並列可能
  - トンプソンサンプリング w/ ランダム基底など、獲得関数の計算が十分高速な場合、効率は出ない（おそらく通信）
    - 逆に、候補点集合の数を増やせることを示唆
  - ハイパーパラメータ学習のほうが律速になる（未並列化・今後の課題）
    - 下のプロットには含まれていない

EI でのステップあたりの時間



TS でのステップあたりの時間



目次（クリックするとジャンプします）

まとめ

- 最適化問題にベイズ最適化を適用すると、少ない試行回数でより良い結果を得ることができる
- ベイズ最適化を実装したプログラムパッケージは多数公開されている
- 我々が開発・公開しているPHYSBO もそのひとつ
  - 既知データ数の増大に対する計算量の増大を抑えているのが特徴
- 本スライドではPHYSBO の使い方やベンチマーク計算を紹介した
- 質問や要望は下記の GitHub の Issue もしくは下記メールアドレスまで！

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

physbo-dev@issp.u-tokyo.ac.jp

参考文献（教科書・レビュー）

- ベイズ最適化（およびそこで使われるガウス過程）
  - E. Brochu, et al. arXiv:1012.2599 (2010) [web]
  - 佐藤一誠 「ベイズ的最適化 (Bayesian Optimization) の入門とその応用-」 (2015) [web]
- ガウス過程およびカーネル法について
  - C. M. Bishop "Pattern Recognition and Machine Learning", Springer-Verlag NY (2006) [web]
  - C. E. Rasmussen and C. K. I. Williams "Gaussian Processes for Machine Learning" The MIT Press (2006) [web]
  - 持橋大地・大羽成征「ガウス過程と機械学習」講談社 (2019) [web]
- COMBO/PHYSBO でなされている高速化について
  - T. Ueno, et al., Materials Discovery 4, 18-21 (2016) [web]
  - PHYSBO マニュアル (develop版) [web]

目次（クリックするとジャンプします）

ガウス過程

- 関数 f(x) について、n 個の任意の点 {x1, x2, ..., xn} 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

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

- 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

目次（クリックするとジャンプします）

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