統計数理(2018)
第 66 巻 第 2 号 339–351

［統計ソフトウェア］

P³：Pythonによる並列計算機用粒子フィルタライブラリ

中野 慎也[1,2]・有吉 雄哉[1,3]・樋口 知之[1,2]

(受付 2018 年 2 月 20 日；改訂 8 月 13 日；採択 10 月 2 日)

要　　旨

粒子フィルタ(PF)は，多数の粒子を用いたモンテカルロ計算に基づく状態推定手法であり，非線型，非ガウスの問題に適用できることから広く様々な目的で用いられるようになってきている．一方で，PF には，推定に必要となる粒子の数が状態変数の自由度に対して指数関数的に増大するため，計算量も指数関数的に増大してしまうという欠点がある．並列計算機の利用は，PF の計算量に対処する手段の一つとして有効であると考えられる．しかし，並列計算機を使うには並列プログラミングの知識が必要であり，また，PF には並列化の困難な処理が含まれているため，並列プログラミングの知識があるユーザにとっても，PF で高い並列化効率を実現するのは容易ではない．そこで，並列化効率の高い PF アルゴリズムを手軽に利用できるようにするために P³(Python Parallelized Particle Filter Library)という Python ライブラリを開発した．本稿では，P³ で利用できる PF の並列アルゴリズムについて述べ，構成の概要や特徴を紹介する．

キーワード：粒子フィルタ，並列計算，Python.

1. はじめに

粒子フィルタ(particle filter; 以下 PF)(Gordon et al., 1993; Kitagawa, 1993, 1996; Doucet et al., 2001)は，非線型・非ガウスの状態空間モデルに適用可能な状態推定手法で，非線型時系列解析や動画上のターゲット追跡，さらにはデータ同化など，様々な目的で用いられる．PF では，状態変数の確率分布を多数の粒子で表現し，モンテカルロ法の考え方に基づいて逐次ベイズ推定の計算を行う．しかし，PF では，状態変数の自由度(独立と見なせる状態変数の数)が大きくなると，いわゆる次元の呪いの問題が顕在化するという問題がある(Daum and Huang, 2003; Bengtsson et al., 2008; Snyder et al., 2008)．また，詳細な事後分布の情報を得たい場合には，さらに多数の粒子を用いる必要があり，12 変数の推定に 10^8 個の粒子を使用した事例もある(Nakamura et al., 2009)．PF の計算量は，少なくとも粒子数 N のオーダーになるため，PF では計算時間が大きな問題となってくる．

多数の粒子を用いて高速に推定を行う手段としては，並列計算機の利用が考えられる．しかし，マルチコア，マルチノードの並列計算機が安価に購入できる状況にある一方，それを使い

[1] 統計数理研究所：〒 190–8562 東京都立川市緑町 10–3
[2] 総合研究大学院大学 複合科学研究科：〒 240–0193 神奈川県三浦郡葉山町湘南国際村
[3] 現 日本文理大学 工学部：〒 870–0397 大分県大分市一木 1727

こなすには並列プログラミングの知識が必須であり，誰もが直ちに並列計算機上で PF を実装できるものではない．また，PF で通常用いられるフィルタリング手続きには，並列化の困難な処理が含まれているため，並列プログラミングの知識があるユーザにとっても，高い並列化効率を実現するのは容易ではない．フィルタリング手続きを改良し，高い並列化効率を実現する方法はいくつか提案されている（Bolić et al., 2005; Nakano, 2010; Nakano and Higuchi, 2010）ものの，その手続きは煩雑であり，プログラムの実装に掛かる手間の問題は依然として残る．

P^3（Python Parallelized Particle Filter Library）は，並列化効率の高い PF 手法を利用しやすくするために，Python ライブラリとして整備したものである．Python は，インタプリタ型のプログラミング言語ではあるが，numpy, scipy をはじめとする高速な数値計算用ライブラリや，統計計算，機械学習のライブラリが豊富に用意されている他，mpi4py のような並列計算の手段も用意されており，科学技術計算の分野にもかなり普及した感がある．一般的な状態空間モデルに適用可能な逐次ベイズ推定手法を実装した Python ライブラリとしては，すでに `FilterPy`（Labbe, 2015）などがあるが，P^3 では並列計算機を活用して比較的大きな規模の非線型・非ガウス状態空間モデルを扱うことを想定している．以下では，まず PF の基本的な形のアルゴリズムについて述べ，P^3 で実装されている並列計算用の PF アルゴリズムについて説明した後，P^3 の構成や特徴を述べる．

2. 粒子フィルタの基本的なアルゴリズム

2.1 非線型状態空間モデル

時刻 t_k の状態を $\boldsymbol{x}_k$，時刻 t_k に得られる観測を $\boldsymbol{y}_k$ と表す．時刻 t_{k-1} から時刻 t_k の間の状態遷移は，関数 $\boldsymbol{f}_k$ を用いて

$$\boldsymbol{x}_k = \boldsymbol{f}_k(\boldsymbol{x}_{k-1}) + \boldsymbol{v}_k \tag{2.1}$$

という形のシステムモデルで記述されるものとする．但し，確率変数 $\boldsymbol{v}_k$ は確率的な変動を表し，システムノイズと呼ばれる．一方，観測 $\boldsymbol{y}_k$ は，状態 $\boldsymbol{x}_k$ との間に以下の関係があるものとする:

$$\boldsymbol{y}_k = \boldsymbol{h}_k(\boldsymbol{x}_k) + \boldsymbol{w}_k. \tag{2.2}$$

関数 $\boldsymbol{h}$ は $\boldsymbol{x}_k$ を観測に対応づける関数，$\boldsymbol{w}_k$ は観測ノイズである．式(2.1), (2.2)をまとめて，非線型状態空間モデルと呼ぶ．

この非線型状態空間モデルは，以下に示す確率分布の形に書き直すこともできる:

$$\boldsymbol{x}_k \sim p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}), \tag{2.3a}$$

$$\boldsymbol{y}_k \sim p(\boldsymbol{y}_k|\boldsymbol{x}_k). \tag{2.3b}$$

例えば，式(2.1)は，$\boldsymbol{v}_k$，$\boldsymbol{w}_k$ がガウス分布

$$\boldsymbol{v}_k \sim \mathcal{N}(\boldsymbol{v}_k; \boldsymbol{0}, \mathsf{Q}), \tag{2.4a}$$

$$\boldsymbol{w}_k \sim \mathcal{N}(\boldsymbol{w}_k; \boldsymbol{0}, \mathsf{R}) \tag{2.4b}$$

に従うとき，

$$p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) = \mathcal{N}(\boldsymbol{x}_k; \boldsymbol{f}_k(\boldsymbol{x}_{k-1}), \mathsf{Q}), \tag{2.5a}$$

$$p(\boldsymbol{y}_k|\boldsymbol{x}_k) = \mathcal{N}(\boldsymbol{y}_k; \boldsymbol{h}_k(\boldsymbol{x}_k), \mathsf{R}) \tag{2.5b}$$

と書くことができる．

式(2.3)の形を導入すると，時刻 t_k までの観測データ $\boldsymbol{y}_{1:k} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k\}$ が与えられた時の $\boldsymbol{x}_k$ は，$t = 0$ における状態 $\boldsymbol{x}_0$ の分布 $p(\boldsymbol{x}_0)$ を与えておけば，次の 2 つの式を用いて推定することができる:

$$p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1}) = \int p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1}|\boldsymbol{y}_{1:k-1})\, d\boldsymbol{x}_{k-1}, \tag{2.6a}$$

$$p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k}) = \frac{p(\boldsymbol{y}_k|\boldsymbol{x}_k) p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1})}{\int p(\boldsymbol{y}_k|\boldsymbol{x}_k) p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1})\, d\boldsymbol{x}_k}. \tag{2.6b}$$

ここで，$p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1})$ は予測分布，$p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ はフィルタ分布と呼ばれる．式(2.6a)では予測分布を得るのにフィルタ分布が用いられ，式(2.6b)では逆にフィルタ分布を得るのに予測分布が用いられていることに注意すると，式(2.6a), (2.6b)を時刻 t_0 から交互に適用することで，各時刻のフィルタ分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ が得られる．このように式(2.6a), (2.6b)を逐次的に適用して，$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ を推定する方法を逐次ベイズ推定と呼ぶ．PF は逐次ベイズ推定の枠組みに従って $\boldsymbol{x}_k$ を推定するアルゴリズムの一つである．

2.2 粒子フィルタ(PF)の概要

PF では，確率分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1})$ や $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ を N 個の粒子で表し，式(2.6)を近似計算する．PF には様々な変形版が存在する（e.g., Doucet et al., 2001; van Leeuwen, 2009)が，最も基本的なアルゴリズムは次に述べるとおりである．

まず，式(2.6a)の計算をモンテカルロ法で行う．時刻 t_{k-1} において，$p(\boldsymbol{x}_{k-1}|\boldsymbol{y}_{1:k-1})$ に従う N 個のサンプル(粒子)$\{\boldsymbol{x}^{(i)}_{k-1|k-1}\}_{i=1}^N$ が得られていたとする．$p(\boldsymbol{x}_k|\boldsymbol{x}^{(i)}_{k-1|k-1})$ に従う乱数は，$p(\boldsymbol{v}_k)$ から生成した乱数 $\boldsymbol{v}^{(i)}_k$ を用いて

$$\boldsymbol{x}^{(i)}_{k|k-1} = \boldsymbol{f}_k(\boldsymbol{x}^{(i)}_{k-1|k-1}) + \boldsymbol{v}^{(i)}_k \tag{2.7}$$

から得られる．式(2.7)にしたがい，各 i に対して $\boldsymbol{x}^{(i)}_{k|k-1}$ を生成すれば，予測分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k-1})$ に従う N 個の粒子 $\{\boldsymbol{x}^{(i)}_{k|k-1}\}_{i=1}^N$ が得られる．

次に式(2.6b)にしたがって，フィルタ分布を求める．式(2.6b)は，$\{\boldsymbol{x}^{(i)}_{k|k-1}\}_{i=1}^N$ の各粒子に尤度 $p(\boldsymbol{y}_k|\boldsymbol{x}^{(i)}_{k|k-1})$ で重みづけすれば，フィルタ分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ の形状を表現できることを示している．ここで，各粒子 $\boldsymbol{x}^{(i)}_{k|k-1}$ が $p(\boldsymbol{y}_k|\boldsymbol{x}^{(i)}_{k|k-1})$ に比例する確率

$$\beta^{(i)}_k = \frac{p(\boldsymbol{y}_k|\boldsymbol{x}^{(i)}_{k|k-1})}{\sum_{i=1}^N p(\boldsymbol{y}_k|\boldsymbol{x}^{(i)}_{k|k-1})} \tag{2.8}$$

で抽出されるように N 回復元抽出を行い，新たに N 個の粒子の集合 $\{\boldsymbol{x}^{(i)}_{k|k}\}_{i=1}^N$ を作る．そうすると，$\{\boldsymbol{x}^{(i)}_{k|k}\}_{i=1}^N$ は，元の粒子 $\boldsymbol{x}^{(i)}_{k|k-1}$ の複製を $\beta^{(i)}_k$ にほぼ比例する個数だけ含んでおり，重みづけをしなくてもフィルタ分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ を表現していることになる．このように N 回復元抽出によって新たに N 個の粒子の集合を得る操作をリサンプリングと呼ぶ．

アルゴリズムをまとめると，以下のようになる：

(1) $t = 0$ における状態 $\boldsymbol{x}_0$ の分布 $p(\boldsymbol{x}_0)$ にしたがう乱数 $\boldsymbol{x}^{(i)}_{0|0} \sim p(\boldsymbol{x}_0)$ を N 個生成する．
(2) 毎時間ステップ k $(k = 1, \ldots, K)$ において以下を実行する．
  (a) 予測
    • 各 i $(i = 1, \ldots, N)$ について $p(\boldsymbol{v}_k)$ にしたがう乱数 $\boldsymbol{v}^{(i)}_k \sim p(\boldsymbol{v}_k)$ を N 個生成する．
    • 各粒子 i $(i = 1, \ldots, N)$ について

$$\boldsymbol{x}_{k|k-1}^{(i)} = \boldsymbol{f}_k(\boldsymbol{x}_{k-1|k-1}^{(i)}) + \boldsymbol{v}_k^{(i)}$$

により，$\boldsymbol{x}_{k-1|k-1}^{(i)}$ から $\boldsymbol{x}_{k|k-1}^{(i)}$ を生成する．

(b) フィルタリング

〈尤度・重みの計算〉 各粒子 i について，尤度 $p(\boldsymbol{y}_k|\boldsymbol{x}_{k|k-1}^{(i)})$ を計算し，重み

$$\beta_k^{(i)} = p(\boldsymbol{y}_k|\boldsymbol{x}_{k|k-1}^{(i)}) \Big/ \sum_{i=1}^{N} p(\boldsymbol{y}_k|\boldsymbol{x}_{k|k-1}^{(i)})$$

を求める．

〈リサンプリング〉 粒子の集合 $\{\boldsymbol{x}_{k|k-1}^{(1)}, \ldots, \boldsymbol{x}_{k|k-1}^{(N)}\}$ から，各粒子 $\boldsymbol{x}_{k|k-1}^{(i)}$ が $\beta_k^{(i)}$ の確率で抽出されるように復元抽出を N 回繰り返す「リサンプリング」を行い，$\{\boldsymbol{x}_{k|k}^{(1)}, \ldots, \boldsymbol{x}_{k|k}^{(N)}\}$ を生成する．

このように，尤度に比例する重みでリサンプリングを行い，フィルタ分布を計算するのが PF の基本的な形式（Gordon et al., 1993; Kitagawa, 1996）である．このような形式のアルゴリズムを特にブートストラップフィルタ，あるいはモンテカルロフィルタと呼ぶ場合もある．

3. 並列アルゴリズム

PF では，必要な粒子数 N が $\boldsymbol{x}_k$ の次元に対して指数関数的に増大する．そこで，多数の粒子を高速で処理する手段として，並列計算機の使用が考えられる．粒子フィルタのアルゴリズムにおいて，(a) の予測の手続きは，各粒子の処理が独立しているため，容易に並列化できる．しかし，(b) のフィルタリングを行う際に必要となるリサンプリングの手続きが，並列化の際に問題となる．

図 1 は，通常の粒子フィルタのリサンプリングを並列計算機上で行った場合の概念図である．ここでは，互いにメモリを共有しない複数のプロセスによって並列に処理が行われるマルチプロセス型の並列計算を考える．リサンプリングの手続きでは，重み $\beta_k^{(i)}$ の小さい（観測と合わない）粒子は破棄され，重み $\beta_k^{(i)}$ に応じて粒子の複製が生成されるため，個々のプロセスが保持する粒子数にばらつきが生じることになる．次の予測の手続きに進んだときに高い並列化効率を得るには，粒子数の増えたプロセスから粒子数の減ったプロセスに粒子を移動し，各プロセスの保持する粒子の数を均等に再配分する操作が必要になる．粒子数を決める重み $\beta_k^{(i)}$ の値にはランダム性があることから，各プロセスの粒子の増減数はランダムに決まる．したがって，粒子の再配分に伴うプロセス間通信には規則性がなく，並列処理が難しい．

そこで，高い並列化効率を実現する方法として，粒子をグループに分割してリサンプリングを行い，グループごとに異なる重みを持たせる方法が提案されている（Bolić et al., 2005）．単にグループに分けただけでは，各グループに割り当てられた個数の粒子のみで推定を行うのと同等になってしまうため，全体で高い精度の推定結果を得るには，グループ間の情報交換が必要になる．情報交換は，グループ間で粒子を部分的に交換する方法（Balasingam et al., 2011; Bai et al., 2016）や，グループ間の相互作用を考える方法（Hlinka et al., 2013; Savic et al., 2014）なども研究が進められているが，P^3 では，並列化効率を重視し，次に述べる 2 種類の手法を実装している．

3.1 階層的リサンプリング

P^3 で用意しているリサンプリング法の 1 つ目は，階層的リサンプリング（Nakano, 2010）である．この方法は，図 2 に示すように，まず各プロセス内でリサンプリングを行い，次に各プ

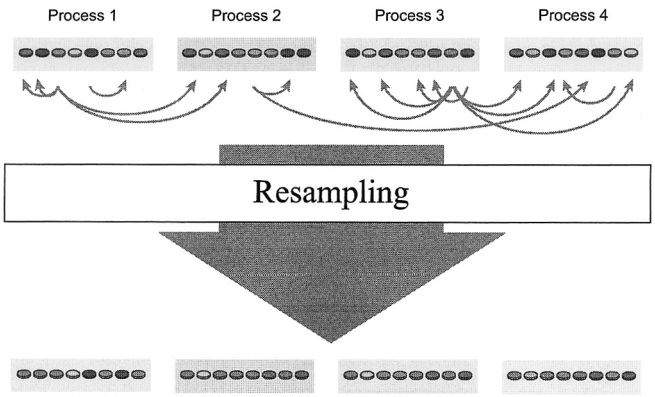

図 1. 通常のリサンプリング手法を並列計算機上で行った場合.

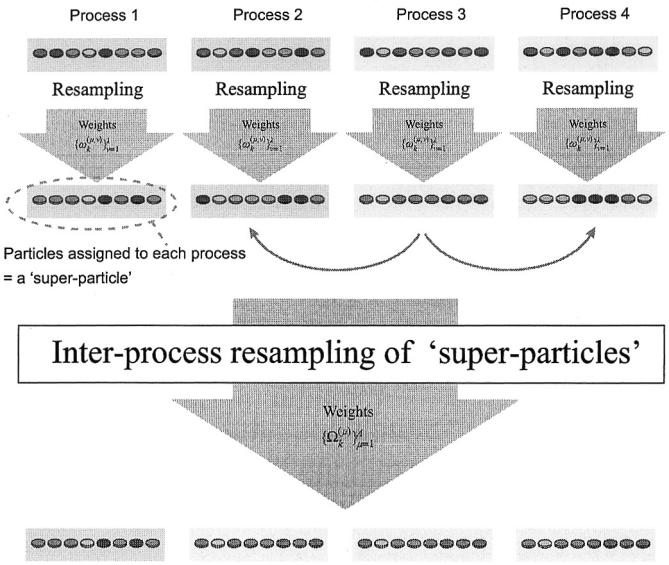

図 2. 階層的リサンプリングの概念図（Nakano, 2010）.

ロセスに属する粒子の集合を超粒子と見なして，超粒子をリサンプリングするというものである.

階層的リサンプリングを行うには，まず各プロセス μ に割り当てられた粒子の集合 G_μ の重みは

$$\Omega_k^{(\mu)} = \sum_{\nu=1}^{\lambda} \beta_k^{([\mu-1]\lambda+\nu)} \tag{3.1}$$

となる．但し，λ は各プロセスに属する粒子の数で，ここでは全プロセスに均等に λ 個の粒子が割り当てられているものとする．したがって，プロセス数を Λ として，

$$\lambda = N/\Lambda \tag{3.2}$$

である．各粒子の重み $\beta_k^{([\mu-1]\lambda+\nu)}$ を集合 G_μ の重み $\Omega_k^{(\mu)}$ で割ると，集合 G_μ 内での各粒子の重みは

$$\omega_k^{(\nu)} = \beta_k^{([\mu-1]\lambda+\nu)}/\Omega_k^{(\mu)} \tag{3.3}$$

となる．次に，各集合 G_μ の中で，重み $\{\omega_k^{(\nu)}\}$ を用いて粒子のリサンプリング (ローカル・リサンプリング) を行う．ローカル・リサンプリングの手続きは，各プロセスで閉じており，異なるプロセスのローカル・リサンプリングは並列に実行できる．最後に，各集合 G_μ を超粒子と見なし，重み $\{\Omega_k^{(\mu)}\}$ を用いてリサンプリング(メタ・リサンプリング)する．これにより，観測と合わない粒子しか保持していない集合は破棄され，観測と合う粒子の集合に置き換えられる．P^3 では，ローカル・リサンプリングの手続きを `local_resampling()` という関数で，メタ・リサンプリングの手続きを `meta_resampling()` という関数で提供している．

なお，各 μ に対する $\Omega_k^{(\mu)}$ の値のばらつきが大きくない場合は，メタ・リサンプリングを行わず，$\Omega_k^{(\mu)}$ で重み付けしたままの方がフィルタ分布 $p(\boldsymbol{x}_k|\boldsymbol{y}_{1:k})$ をよりよく表現できると考えられる．そこで P^3 では，$\Omega_k^{(\mu)}$ のばらつきを評価するためにエントロピー

$$S_{\Omega,k} = -\sum_{\mu=1}^{\Lambda} \Omega_k^{(\mu)} \log \Omega_k^{(\mu)} \tag{3.4}$$

から

$$\Lambda_{\mathrm{eff},k} = e^{S_{\Omega,k}} \tag{3.5}$$

という量を計算し，$\Lambda_{\mathrm{eff},k}$ がある閾値より小さくなった場合のみに，メタ・リサンプリングの手続きが実行される．

3.2 Alternately lattice-pattern switching(ALPS)法

P^3 で提供するリサンプリング法の 2 つ目は，alternately lattice-pattern switching(ALPS)法 (Nakano and Higuchi, 2010, 2012)である．この方法では，複数のプロセスに割り当てられた粒子の集合をまとめてグループにし，各グループの中でローカル・リサンプリングと同じ操作を適用する．グループ分けを行う際には，まず，$2m \times 2n$ 個のプロセスを図 3 に示すような 2 次元トーラス状のグラフの各ノードに割り当てる．そして，図 3 左のように，ノード(プロセス)

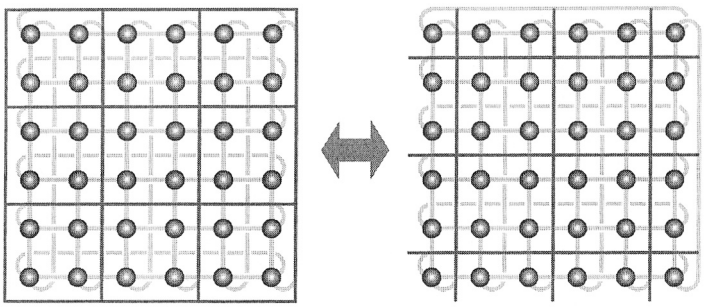

図 3. Alternate lattice-pattern switching 法の概念図 (Nakano and Higuchi, 2010)．

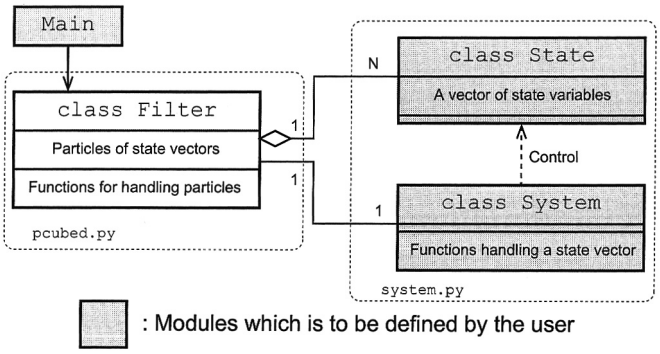

図 4. P³ の構成の概略図.

を 2 × 2 個のグループに分割する．リサンプリングは，各グループで並列に実行される．これだけでは，単なるローカル・リサンプリングと大差ないが，ALPS では，次のステップで，図 3 右に示すようにグループ分けのパターンを変え，その上でグループごとのリサンプリングを行う．ステップごとにグループ分けパターンを変えることで，観測に合う粒子の情報が全プロセスに行き渡るようにし，推定性能を向上させる．

4. P³ の構成

カルマンフィルタを適用できる線型状態空間モデルの場合，状態遷移行列，観測行列などの行列を与えれば記述できる．しかし，PF の扱う一般の非線型状態空間モデルの定義の仕方は，個々の問題によって様々であり，非線型微分方程式が使われる場合もあるし，非ガウスの確率分布が含まれることも考えられる．このような様々な場合に対応できるようにするため，P³ では，既存の Python ライブラリとの互換性には配慮せず，式(2.1), (2.2)の状態空間モデルの内容をユーザが自由に定義できるように設計されている．但し，式(2.1), (2.2)の計算を P³ のモジュールから実行できるようにするため，ユーザは，状態ベクトル，状態空間モデルの定義を `system.py` という名前のファイルに記述しておく必要がある．状態ベクトル，状態空間モデルの定義は，それぞれ `class State`，`class System` という名前で所定の形式に従って記述する．

PF を実行するために必要な関数は，ファイル `pcubed.py` 中で定義されている `class Filter` のメンバ関数の形で用意されている．並列計算やリサンプリングの処理は，`class Filter` の中に記述されており，ユーザは `class State`，`class System` を記述さえすれば，面倒なプログラミング作業を行わずとも `class Filter` 中の関数を使って状態推定の計算が実行できる．図 4 に，P³ の構成を示す．

なお，従来の逐次ベイズ推定のための計算ライブラリでは，状態ベクトルを 1 つの配列にまとめた上で計算を行う実装が多かった．しかし，高次元の状態空間モデルにおいては，状態変数のそれぞれが別々の意味を持つことが少なくない．このような場合，異なる意味を持つ状態変数は，プログラム上でも異なる変数名で扱った方が，可読性が向上すると考えられる．特に，シミュレーションモデルからシステムモデルを構成するデータ同化においては，元となるシミュレーションプログラムの変数名が流用できるため，システムモデルのプログラムを書く際にも都合がよい．そのため P³ では，状態ベクトルを配列ではなく，`class State` というクラス(構造体)の形で保持する設計になっている．

以下では，実際に P³ を使うために必要な情報として，`class State` と `class System` の定義

表 1. class System で定義すべき関数.

関数名	引数	機能
system_config	なし	事前のパラメータ設定，前処理を行う.
init_state	ydata, x	各粒子の初期値を設定する.
step_system_model	x	各粒子を 1 ステップ進めるモデル $\boldsymbol{f}_k$ に相当する.
add_noise	x	各粒子にノイズ $\boldsymbol{v}_k$ を加える.
log_likelihood	x, ydata	各粒子の対数尤度を求める.
obs_predict	x	各粒子の状態変数の値を与えたときの観測の期待値を求める.

の仕方を説明し，class Filter の内容について説明した後，メインプログラムの書き方の概要を説明する.

4.1 class State の定義

P^3 の class Filter を使う際には，予め system.py の中で class State と class System を定義する必要がある．このうち，class State は，式(2.1)に出てくる状態ベクトル $\boldsymbol{x}_k$ の全要素(つまり全状態変数)をまとめたもので，推定するべき状態変数は，すべて class State のメンバ変数として列挙する.

各メンバ変数は，numpy.ndarray とし，0 で初期化するようにする．例えば，$a_{0,k}$, $a_{1,k}$, ..., $a_{4,k}$ と $b_{0,k}$, $b_{1,k}$, $b_{2,k}$ という 8 つの変数をまとめて状態ベクトル $\boldsymbol{x}_k$ として扱いたいときは,

```
class State:
  def __init__(self):
    self.a = np.zeros((5))
    self.b = np.zeros((3))
```

のように定義する.

4.2 class System の定義

class State は，状態ベクトルの定義を与えるだけで，式(2.1)のようなシステムモデルの定義は，class System で行う．class System の中では，pcubed.py から呼ばれる関数を所定の名前で定義する必要がある．表 1 が，class System で定義すべき関数である．以下に具体的な定義の仕方を述べる.

system_config(cls)

この関数は，クラスメソッドとして定義され，事前に実行しておくべき前処理や，パラメータ値の設定をここに記述する．前処理等が必要ない場合も，system_config() をダミー関数にするなど，何らかの形で定義する必要がある.

init_state(self, ydata, x)

各粒子の初期値を設定する．引数 x は class State 型の変数であり，x のメンバ変数に状態変数の初期値が代入されるように関数 init_state() を定義する必要がある．PF において，各粒子の初期値は，初期状態の確率分布 $p(\boldsymbol{x}_0)$ にしたがう乱数で与えるので，基本的に $p(\boldsymbol{x}_0)$ にしたがう乱数が x のメンバ変数に代入されるように定義すればよい.

ydata は System.nobs の長さを持つ numpy.ndarray クラスのオブジェクトで，初期化に用いる観測データをここに与えることができる．これは，観測可能な状態変数をデータから与えるというようなことを想定している．つまり，$p(\boldsymbol{x}_0)$ ではなく，$p(\boldsymbol{x}_0|\boldsymbol{y}_0)$ から各粒子の初期値を

生成できるようになっている．

`step_system_model(self, x)`

1 ステップ分の粒子の時間発展を計算する．データ同化を行う場合，シミュレーションモデルの 1 ステップ分をここで定義することになる．

x は入力と出力を兼ねた class State の変数で，粒子 i の時刻 t_{k-1} における状態変数の値 $\boldsymbol{x}^{(i)}_{k-1|k-1}$ を x として与えると，$\boldsymbol{f}_k(\boldsymbol{x}^{(i)}_{k-1|k-1})$ の値が State クラスのオブジェクトの形で x に代入されるようにする．

`add_noise(self, x)`

粒子にシステムノイズ $\boldsymbol{v}^{(i)}_k$ を付加する．粒子の持つ状態変数の値を State クラスの形で x に入力すると，それにノイズが付加された値が x に代入されるように定義する．add_noise() を step_system() と組み合わせることで

$$\boldsymbol{x}^{(i)}_{k|k-1} = \boldsymbol{f}_k(\boldsymbol{x}^{(i)}_{k-1|k-1}) + \boldsymbol{v}^{(i)}_k$$

の計算が実行できる．

`log_likelihood(self, x, ydata)`

観測データ $\boldsymbol{y}_k$ が与えられた下での粒子 $\boldsymbol{x}^{(i)}_{k|k-1}$ の対数尤度 $p(\boldsymbol{y}_k|\boldsymbol{x}^{(i)}_{k|k-1})$ を計算する．入力として，粒子の状態変数の値を State クラスの形で x に，観測値を長さ System.nobs の numpy.ndarray クラスの形で ydata に与えるようにする．計算された対数尤度は，実数値(スカラー)の返り値として返すようにする．

4.3 class Filter の内容

class Filter は，表 2 に示す関数で構成されており，メインプログラムから，表 2 の関数を呼ぶことで粒子フィルタの計算が実現できる．以下で，各関数の役割を述べる．

`init_ensemble(nptot, yinit)`

N 個の粒子を初期化する．引数 nptot には粒子数 N(整数値)を入力し，yinit には初期化のための観測データを入力する．yinit の型は numpy.ndarray で System.nobs の長さを持つ．具体的な初期化の方法は，system.py に記述する必要がある．

`finalize()`

終了処理をする．class Filter を使い終わった後に呼ぶ．

`step_ensemble()`

各粒子を 1 ステップ進める．これは式(2.7) の $\boldsymbol{f}_k(\boldsymbol{x}^{(i)}_{k-1|k-1})$ の部分の計算に相当する．$\boldsymbol{f}_k$ の

表 2. class Filter で用意されている関数.

関数名	引数	返り値	機能
init_ensemble	iregopt, nptot, yinit	なし	N 個の粒子の初期化.
finalize	なし	なし	終了処理.
step_ensemble	なし	なし	全粒子を 1 ステップ進める.
add_noise_to_ensemble	なし	なし	全粒子にノイズを加える.
ensemble_mean	mrank	class State	全粒子の平均を返す.
dist_reweight	ydata	なし	全粒子を尤度に従って重み付けする.
local_resample	なし	なし	プロセス内でリサンプリング.
meta_resample	なし	なし	粒子の集合のメタ・リサンプリング.
regional_resample	なし	なし	ALPS によるリサンプリング.

定義は system.py に記述する．

add_noise_to_ensemble()

各粒子にシステムノイズを加える．これは式(2.7)で $\boldsymbol{v}_k^{(i)}$ を足す部分に相当する．

ensemble_mean()

粒子が保持する状態 $\boldsymbol{x}_{k|k}^{(i)}$ の平均

$$\bar{\boldsymbol{x}}_{k|k} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_{k|k}^{(i)}$$

を求める．これにより PF による $\boldsymbol{x}_k$ の推定値が得られる．

dist_reweight(ydata)

観測データ ydata を参照して，式(2.8)の重み $\beta_k^{(i)}$ を計算する．ydata は numpy.ndarray で System.nobs の長さを持つ．

local_resample()

dist_reweight で求めた重みにしたがい，各プロセス内で粒子のローカル・リサンプリングを行う．

meta_resample()

各プロセスに割り当てられた粒子の集合をまとめて超粒子と見なし，dist_reweight で求めた重みから計算される超粒子の重みにしたがって，メタ・リサンプリングを行う．但し，この関数では，超粒子の重み $\Omega_k^{(\mu)}$ のばらつきを評価するために，式(3.5)の Λ_{eff} を計算し，$\Lambda_{\text{eff}} \geq 0.5$ の時は何もしない(つまりメタ・リサンプリングを行わない)ようになっている．

regional_resample()

ALPS 法によるリサンプリングを実行する．ALPS のグループ分けパターンは，この関数を呼ぶ度に自動的に切り替えられる．

4.4 プログラムの書き方のまとめ

class Filter を用いて，PF の予測ステップを行うには，メインプログラム中で

```
Filter.step_ensemble()
Filter.add_noise_to_ensemble()
```

のように記述する．フィルタリングについては，階層的リサンプリングを用いる場合，

```
Filter.dist_reweight( ydata )
Filter.local_resample()
Filter.meta_resample()
```

のように記述すればよく，ALPS を用いる場合，

```
Filter.dist_reweight( ydata )
Filter.regional_resample()
```

のように記述すればよい．

5. おわりに

PF は，状態変数が数個程度の小規模な問題であれば容易に実装できるため，非線型・非ガウスの状態空間モデルにおいて広く活用されているが，問題の規模に対して，計算量が指数関

数的に増大するという問題がある．P^3 は，粒子フィルタの適用対象としては，比較的規模の大きい非線型の問題に対しても，並列計算機を利用して状態推定を実現できる手段を提供する．粒子フィルタを並列計算機で実行しようとすると，個々の粒子を並列処理するために並列プログラミングの知識が必要になる上，リサンプリング時に粒子を再配分する処理の実装に労力が必要となるが，P^3 では並列計算やリサンプリングの処理が抽象化されているため，ユーザは煩雑な処理を自分でプログラミングしなくても，並列化効率の高い PF アルゴリズムを利用でき，状態空間モデルの構築に専念することができると考えられる．P^3 は無償で配布している．利用を希望される方は，次のウェブサイト（http://daweb.ism.ac.jp/support/software/P-cubed/P-cubed.html）に記載の要領で申し込みいただきたい．

現在は PF のみを実装しているが，用途によっては別の手法を用いた方がよい場合もある．例えば，数百次元以上の大規模な非線型状態空間モデルを扱う場合には，PF よりもアンサンブルカルマンフィルタ（Evensen, 1994, 2003）が有効であるし，システムノイズがガウス的である場合には，混合ガウスフィルタ（Stordal et al., 2011）などが有効であると考えられる．今後は，このような他の有用な非線型逐次ベイズ推定手法を追加するなどして，機能の充実を図る予定である．

謝　辞

P^3 のプログラム開発にあたっては，科学研究費補助金基盤研究 B（課題番号: 26280010）の助成を受けた．ここに感謝の意を表する．

参 考 文 献

Bai, F., Gu, F., Hu, X. and Guo, S. (2016). Particle routing in distributed particle filters for large-scale spatial temporal systems, *IEEE Transactions on Parallel and Distributed Systems*, **27**, 481–493.

Balasingam, B., Bolić, M., Djurić, P. M. and Míguez, J. (2011). Efficient distributed resampling for particle filters, *Proceedings of IEEE International Conference of Acoustics, Speech, Signal Processing*, 3772–3775.

Bengtsson, T., Bickel, P. and Li, B. (2008). Curse-of-dimensionality revisited: Collapse of the particle filter in very large scale systems, *Probability and Statistics*: *Essays in Honors of David A. Freedman*, **2**, 316–334, Institute of Mathematical Statistics, Beachwood, Ohio.

Bolić, M., Djurić, P. M. and Hong, S. (2005). Resampling algorithms and architectures for distributed particle filters, *IEEE Transactions on Signal Processing*, **53**, 2442–2450.

Daum, F. and Huang, J. (2003). Curse of dimensionality and particle filters, *Proceedings of IEEE Aerospace Conference*, **4**, 1979–1993.

Doucet, A., de Freitas, N. and Gordon, N. (eds.) (2001). *Sequential Monte Carlo Methods in Practice*, Springer-Verlag, New York.

Evensen, G. (1994). Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics, *Journal of Geophysical Research*, **99(C5)**, 10143–10162.

Evensen, G. (2003). The ensemble Kalman filter: theoretical formulation and practical implementation, *Ocean Dynamics*, **53**, 343–367.

Gordon, N. J., Salmond, D. J. and Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation, *IEE Proceedings F*, **140**, 107–113.

Hlinka, J., Hlawatsch, F. and Djurić, P. M. (2013). Distributed particle filtering in agent networks, *IEEE Signal Processing Magazine*, **30**, 61–81.

Kitagawa, G. (1993). Monte Carlo filtering and smoothing method for non-Gaussian nonlinear state space model, Reserch Memo., No.462, The Institute of Statistical Mathematics, Tokyo.

Kitagawa, G. (1996). Monte Carlo filter and smoother for non-Gaussian nonlinear state space models, *Journal of Computational and Graphical Statistics*, **5**, 1–25.

Labbe, R. (2015). Kalman and Bayesian Filters in Python, http://filterpy.readthedocs.io (2018 年 2 月 17 日閲覧).

Nakamura, K., Yoshida, R., Nagasaki, M., Miyano, S. and Higuchi, T. (2009). Parameter estimation of in silico biological pathways with particle filtering towards peta-scale computing, *Proceedings of Pacific Symposium on Biocomputing*, **14**, 227–238.

Nakano, S. (2010). Population-based quasi-Bayesian algorithm for high-dimensional sequential problems and hierarchization of it for distributed computing environments, *Proceedings of 2010 IEEE World Congress on Computational Intelligence*.

Nakano, S. and Higuchi, T. (2010). A dynamic grouping strategy for implementation of the particle filter on a massively parallel computer, *Proceedings of 13th International Conference on Information Fusion*.

Nakano, S. and Higuchi, T. (2012). Weight adjustment of the particle filter on distributed computing system, *Proceedings of 15th International Conference on Information Fusion*, 2480–2485.

Savic, V., Wymeersch, H. and Zozo, S. (2014). Belief consensus algorithms for fast distributed target tracking in wireless sensor networks, *Signal Processing*, **95**, 149–160.

Snyder, C., Bengtsson, T., Bickel, P. and Anderson, J. (2008). Obstacles to high-dimensional particle filtering, *Monthly Weather Review*, **136**, 4629–4640.

Stordal, A. S., Karlsen, H. A., Nævdal, G., Skaug, H. J. and Vallès, B. (2011). Bridging the ensemble Kalman filter and particle filters: The adaptive Gaussian mixture filter, *Computational Geosciences*, **15**, 293–305.

van Leeuwen, P. J. (2009). Particle filtering in geophysical systems, *Monthly Weather Review*, **137**, 4089–4114.

Proceedings of the Institute of Statistical Mathematics Vol. 66, No. 2, 339–351 (2018) 351

P^3: Python Parallelized Particle Filter Library

Shin'ya Nakano[1,2], Yuya Ariyoshi[1,3] and Tomoyuki Higuchi[1,2]

[1]The Institute of Statistical Mathematics
[2]School of Multidisciplinary Science, SOKENDAI
[3]Now at Faculty of Engineering, Nippon Bunri University

Particle filter (PF) is a class of state-estimation techniques based on Monte Carlo computation that use a large number of particles. Because PF is applicable even to nonlinear and/or non-Gaussian problems, it is used for a variety of purposes. One serious problem of PF is its computational time, which is exponential in the degrees of freedom of the state vector. Parallel computing is an effective way to decrease computational time, but this approach requires skills in parallel programming. Even for experienced users, it is challenging to achieve high computational efficiency in PF computation because the PF algorithm contains a procedure difficult to parallelize. We developed a Python library named P^3(Python Parallelized Particle Filter Library), that enables us to readily use parallel-ready PF algorithms with high parallel efficiency. In this paper, we describe the parallelized PF algorithms available in P^3, as well as explaining the design and characteristics of the library.

Key words: Particle filter, parallel computing, Python.