Linux と C++ を用いた移動ロボットの自律走行シミュレーション

-PID 制御を用いた軌道追従とパーティクルフィルタを用いた自己位置推定-

赤井 直紀

2022 年 12 月 19 日

本研修では，Linux PC 上で，C++ を用いて移動ロボットの軌道追従制御と自己位置推定を行うためのシミュレーションの実装を行う．経路追従の研修では，誤差の定義が容易な関数で表現される軌道（本研修では円軌道）と，任意の点列で表現される軌道の追従を PID 制御を用いて行う．また自己位置推定のためには，ベイズフィルタの一種であるパーティクルフィルタを用いて行う．これらを通じて，ロボットを制御するための基礎的な知識，および実環境に存在する不確実性をどの様に扱うか [1] について学習する．なお Linux PC（Ubuntu），および C/C++ の基礎的な項目の使用に関しては，すでに理解があることを前提とする．

1　想定するロボットと実行環境

ロボットは，xy 平面上を移動するものとし，その状態 $\mathbf{x}$ は 2 次元の位置 x，y と姿勢 θ で表されるものとする．ロボットには，制御入力として並進速度 v と，角速度 ω を与えることができるものとする．今時刻 t において，制御入力 $\mathbf{u}_t = (v_t\ \omega_t)^\top$ が加えられたとする．このとき，ロボットの状態は以下の様に変化するものとする．

$$\begin{pmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{pmatrix} = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} + \begin{pmatrix} v_t \cos\theta_t \\ v_t \sin\theta_t \\ \omega_t \end{pmatrix} \Delta t \tag{1}$$

ここで Δt は，時刻 t から $t+1$ の間の時間間隔である．このモデルにより定まる移動体の移動方式を，差分駆動方式と呼ぶ．

本研修で用いるソースコードは，Linux ディストリビューションの 1 つである Ubuntu 20.04 を用いて開発を行った．ただし，その他の Linux ディストリビューションでも動作させることは可能であるが，実際に動作が行えるかどうかの確認は行っていたため注意すること．

次に，ソースコードの実行について確認する．practice ディレクトリ内の test ディレクトリに入り，以下のコマンドを端末上で実行せよ．

```
$ ./make.sh
$ ./test
```

これにより，ロボットが半時計周りに動く描画が確認できれば問題ない．なお，本研修では gnuplot という描画ソフトを用いて，シミュレーション結果の描画を行う．

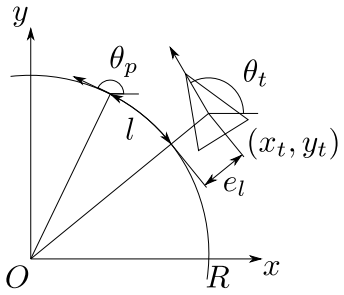

図 1　円軌道追従の概略.

2　円軌道の追従

2.1　円軌道追従の概略

まずはロボットに，半径 R の円軌道を追従させることを考える．図 1 に，円軌道追従を考える際の概略を示す．なお今回の円軌道の追従では，半時計周りに円軌道上をロボットが走行するものとする．ロボットが軌道追従を達成するためには，

- 軌道とロボットとの距離の偏差が 0 となる
- 軌道の向きとロボットの姿勢の偏差が 0 となる

という状況を満たしながら，並進の速度を加えれば良い．そこで，これらを表す偏差の計算方法について考える．

まず軌道とロボットの距離の偏差 e_l は，以下の様に定義できる．

$$e_l = \sqrt{(x_t^2 + y_t^2)} - R \tag{2}$$

この偏差を 0 にできれば，ロボットは円軌道上に存在することになる．

次に，角度の偏差 e_θ について考える．円の中心（原点）とロボット位置を結んだ線と，x 軸の成す角度は $\tan^{-1}(y_t/x_t)$ である[*1]．この直線と円の交点における半時計向きの接線ベクトルの向きの角度は，$\tan^{-1}(y_t/x_t) + \pi/2$ である．これとロボットの姿勢角 θ_t を一致させれば，ロボットは進行方向を向く．しかし実際には，ロボットは前進しながら軌道に接近するため，この地点より先の地点の円の接線ベクトルを目標姿勢角とする方が，滑らかな軌道追従を行える．この様に先の地点を目標とするとき，その距離を lookahead distance と呼ぶ．図 1 では，l が lookahead distance である．すなわち，現在位置と原点を結んだ円の交点から，円上の l 先の地点の半時計向きの接線ベクトル θ_p を求める．半径 R の円上で l 先ということは，R/l 回転した地点となるため，

$$\theta_p = \tan^{-1}\left(\frac{y_t}{x_t}\right) + \frac{R}{l} + \frac{\pi}{2} \tag{3}$$

となる．ここから，軌道の向きとロボットの姿勢角に対して定まる偏差を

$$e_\theta = \theta_p - \theta_t \tag{4}$$

[*1] 逆正接関数に関しては値域に関して注意すること．プログラムの実装においては，引数の符号を考慮して逆正接関数の値を返す atan2(y, x) を使うことが好ましい．以下本稿で逆正接関数を用いる場合は，atan2(y, x) が使われているものとし，値域は正しく考慮されているものとみなす.

と定める．なお，必ず $-\pi \leq e_\theta < \pi$ となる様に修正すること[*2]．

2.2 PID 制御を用いた軌道追従のための制御入力の決定

2.1 節で定めた誤差を基に，PID 制御を用いて制御入力の決定を行う．

$$
\begin{aligned}
v_t &= v_{\max} - K_{v,l}|e_l| + K_{v,\theta}|e_\theta| \\
\omega_t &= K_{p,l}e_l + K_{i,l}\int_0^t e_l \mathrm{d}\tau + K_{d,l}\frac{\mathrm{d}e_l}{\mathrm{d}t} + K_{p,\theta}e_\theta + K_{i,\theta}\int_0^t e_\theta \mathrm{d}\tau + K_{d,\theta}\frac{\mathrm{d}e_\theta}{\mathrm{d}t}
\end{aligned} \tag{5}
$$

ここで，$v_{\max}$ はロボットの最大の移動速度，$K_{v,l}$ と $K_{v,\theta}$ は偏差 e_l と e_θ の大きさに応じて速度を低下させるための係数，$K_{p,l}$，$K_{i,l}$,，$K_{d,l}$，$K_{p,\theta}$，$K_{i,\theta}$，$K_{d,\theta}$ はそれぞれ偏差 e_l と e_θ に対する P，I，D 制御のゲインである．差分駆動方式のロボットは，横方向には移動できないため，基本的には角速度を制御することで，目標軌道への接近，追従を達成する．偏差が大きい場合には，ロボットは速度を減速することで，軌道追従の性能を上げることができる．なお，プログラム上で離散値の微分や積分を計算することはできないため，角速度に関しては以下の様に計算を行う．

$$
\omega_t = K_{p,l}e_l + K_{d,l}\Delta e_l + K_{i,l}\sum^t e_l + K_{p,\theta}e_\theta + K_{d,\theta}\Delta e_\theta + K_{i,\theta}\sum^t e_\theta \tag{6}
$$

ここで，Δe_l と Δe_θ は，時刻 t と $t-1$ におけるそれぞれの偏差の差分，$\sum^t e_l$ と $\sum^t e_\theta$ は時刻 t までのそれぞれの偏差の総和である．

2.3 演習

2.3.1 円軌道追従プログラムの実装

circle_follow ディレクトリ内の main.cpp を以下の様に編集せよ．

- 72 行目から，円軌道とロボット位置の距離の偏差を計算せよ．
- 75 行目から，lookahead distance を考慮した円軌道の向きとロボット姿勢角の偏差を計算せよ．
- 82 行目から，上記で求めた 2 つの偏差を基に，PID 制御を用いて並進，角速度の制御入力を計算せよ．

編集を終えたら，

```
$ ./make.sh
$ ./circle_follow
```

のコマンドを実行し，実際に円軌道の追従が行えることを確認せよ．

2.3.2 制御パラメータ変更による挙動の変更の確認

circle_follow ディレクトリ内の main.cpp 内の 46 から 59 行目にかけて，軌道追従の挙動に関するパラメータが存在する．これらを変更し，以下のことを確認せよ．

[*2] 0 rad と 2π rad は同じ方向であるが，その差分を計算すると $2\pi - 0 = 2\pi$ となる．そしてこの差分を偏差として P 制御を行うと，本来なら偏差 0 であるはずなのに，偏差が 0 とならず P 制御による入力値が非零となる．この問題を回避するために，角度，特に角度の差分の計算結果は $-\pi$ から π に収まらなければならない．

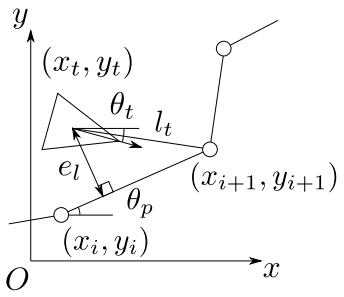

図 2　点列道追従の概略.

- x に関するロボットの初期位置を円軌道から離した場合でも，正しく経路追従できることを確認せよ.
- 最大速度を大きくした場合，P 制御だけでは追従を行うことが難しくなることを確認せよ.
- 最大速度が大きい場合でも，D 制御や I 制御を適切に加えることで円軌道を追従できることを確認せよ.
- P，D，I 制御の効果をそれぞれ述べよ.
- 通常の経路追従では，I 制御は重要な要素とならない．これは，経路の形が不定であるためである．しかし今回の円軌道追従では，I 制御は一定の効果をもたらす．その理由を述べよ．ヒント：ノイズの無い環境で円軌道を追従するためには，どの様な制御入力を定めれば良いかを考えよ.

3　点列軌道の追従

3.1　点列軌道追従の概略

2 章では，円軌道で表現される軌道追従について述べた．しかし実際には，ロボットが移動すべき経路が何かしら特定の関数の形で表現されることは少ない．一般的には，ロボットが走行する経路を一定間隔で区切り，その間隔を直線で表して経路とすることが多い．この区切った点のことをウェイポイントと呼ぶ．本章では，この軌道を点列軌道と呼び，以下この軌道追従ついて考える.

図 2 に，点列軌道追従の概略を示す．点列軌道は点 (x_i, y_i) の集合により表現される．今ロボットは i 番目のウェイポイントまで到達し，図の様に $\mathbf{x}_t = (x_t\ y_t\ \theta_t)^\top$ の状態にあるとする．点列軌道の追従を達成するためには，2 章で述べた様に，軌道とロボット位置の偏差 e_l と，軌道の向きとロボットの姿勢角の偏差 e_θ を 0 にすれば良い．ただし，円軌道の追従の際に考えた偏差の計算とは異なることに注意しなければならない.

角度に関する偏差 e_θ は，円軌道の追従の際に考えた時と同様に，$e_\theta = \theta_p - \theta_t$ で計算される．ただし，$\theta_p = \tan^{-1}\left((y_{i+1} - y_i)/(x_{i+1} - x_i)\right)$ である．e_l の計算に関しては注意が必要である．i，$i+1$ 番目の Waypoint が作る直線と，ロボット位置の点の距離を用いて，点と直線の距離の公式を用いて求めても良いが，その際は e_l の符号が求められないため，別途符号を求めなければならない．符号と偏差を同時に求める方法として，例えば i 番目の Waypoint を原点とし，その x 軸の向きを θ_p とする xy 座標系を考えることもできる．この場合には，その座標上での y の符号を変えた値が，e_l となる（新たに定めた座標上で y が正の場合，負（時計周り）の角速度を入力しなければ軌道に接近できない）．以上より求めた偏差を用いて，同様に式 (5) に用いて，PID 制御に基づく軌道追従制御を行うことができる.

また，円軌道追従のときと比べて，lookahead distance の考え方も変わる．現在のロボット位置から，$i+1$ 番目のウェイポイントまでの距離を考え，これを l_t とする．この l_t が lookahead distance より小さくなった場合，ロボットは $i+1$ 番目のウェイポイントに到達したとみなす．これにより，次回以降 $i+1$，$i+2$ 番目のウェイポイントで構成される直線に対して同様の制御を行うこととなり，lookahead distance を維持することができる．

3.2 演習

3.2.1 点列軌道追従プログラムの実装

path_follow ディレクトリ内の main.cpp を以下の様に編集せよ．

- 89 行目から，ロボット位置と経路の位置に関する偏差を計算せよ．
- 92 行目から，ロボット姿勢と軌道の角度に関する偏差を計算せよ．
- 158 行目から，lookahead distance を考慮して，必要であればウェイポイントの番号を更新せよ．

編集を終えたら，

```
$ ./make.sh
$ ./path_follow
```

のコマンドを実行し，実際に点列軌道の追従が行えることを確認せよ．

3.2.2 制御パラメータ変更による挙動の変更の確認

path_follow ディレクトリ内 main.cpp 内の 63 行目から 72 行目にかけて，軌道追従の性能に関するパラメータが存在する．これらを変更し，以下のことを確認せよ．

- lookahead distance が小さい場合，大きい場合の軌道追従の違いを確認せよ．またそれぞれ，lookahead distance が大きい，小さい場合の利点と欠点を述べよ．
- 今回考えているロボットは差動二輪方式であり，横方向への移動ができない．そのため，軌道追従を行う場合は角速度の入力を偏差に合わせて適切に選ぶ必要がある．ただし，角速度の選択だけでは，軌道追従の精度に限界が現れる．最大速度は変えずに経路追従の精度を維持したい場合，どの様なパラメータチューニングを行うことが効果的か考察せよ．ヒント：lookahead distance と式 (5)，特に並進速度に関して考えよ．

4 パーティクルフィルタを用いた自己位置推定

4.1 実世界に存在する不確実性

これまで 2 種類の軌道追従を行うことを考えてきたが，その際「ロボットの位置は取得できる」という前提で制御入力の計算を行ってきた．しかし実世界では，簡単にロボットの位置を取得することはできない．これまでのシミュレーションでは，式 (1) に示す通り，ロボットは与えられた制御入力の通り動くと仮定していたが，実際のロボットは制御入力通りに動かない．例えば，移動ロボットがモータを回転させて移動する場合には，指令値通りにモータが回転するまでの遅れ，実際の指令値まで到達しない定常偏差，車輪のスリップなど，様々な不確実性をもたらす要因が存在する．すなわち，実際の環境ではロボットの位置を簡単

に知ることはできない．本章では，この不確実性に対処しながら，ロボットの位置を求める方法に関して述べる．なお具体的には，ベイズフィルタの 1 種であるパーティクルフィルタを用いた自己位置推定法に関して述べる．

4.2 想定するシナリオ

ロボットの運動はこれまで通り，式 (1) よって表されるものとする．ただし，真のロボット位置（真値：ground truth）は，式 (1) の通りには動作しないものとする．真値 $\mathbf{x}^{\mathrm{GT}}$ は以下の様に動作するものとする．

$$\begin{pmatrix} x_{t+1}^{\mathrm{GT}} \\ y_{t+1}^{\mathrm{GT}} \\ \theta_{t+1}^{\mathrm{GT}} \end{pmatrix} = \begin{pmatrix} x_{t}^{\mathrm{GT}} \\ y_{t}^{\mathrm{GT}} \\ \theta_{t}^{\mathrm{GT}} \end{pmatrix} + \begin{pmatrix} (1+r_v)v_t \cos\theta_t^{\mathrm{GT}} \\ (1+r_v)v_t \sin\theta_t^{\mathrm{GT}} \\ (1+r_\omega)\omega_t \end{pmatrix} \Delta t \tag{7}$$

ここで r_v と r_ω は，制御入力に加わる定常的なノイズ率を表現する係数である．r_v と r_ω が非零であれば，式 (1) により更新された推定値と真値の誤差は，時間経過とともに増大していく．

またロボットは，全球測位衛星システム（Global Navigation Satellite System: GNSS）のレシーバーを持ち，自身の位置，姿勢を得ることができるものとする[*3]．GNSS により測定される姿勢 $\mathrm{x}^{\mathrm{GNSS}}$ は，真値から一定のノイズの加わったものとする．

$$\begin{pmatrix} x_{t}^{\mathrm{GNSS}} \\ y_{t}^{\mathrm{GNSS}} \\ \theta_{t}^{\mathrm{GNSS}} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(x_t^{\mathrm{GT}}, \sigma_x^2) \\ \mathcal{N}(y_t^{\mathrm{GT}}, \sigma_y^2) \\ \mathcal{N}(\theta_t^{\mathrm{GT}}, \sigma_\theta^2) \end{pmatrix} \tag{8}$$

ここで $\mathcal{N}(x, \sigma^2)$ は，平均 x，分散 σ^2 に従う正規乱数を発生させる関数であり，σ_x^2，σ_y^2，σ_θ^2 は GNSS による測位の x，y，θ 方向の分散である．

4.3 パーティクルフィルタを用いた自己位置推定

自己位置推定の目的は，制御入力列 $\mathbf{u}_{1:t}$，センサ観測列 $\mathbf{z}_{1:t}$ を用いて，現時刻のロボット位置 $\mathbf{x}_t$ を求めることである（$1:t$ は時系列データの集合を表しており，例えば $\mathbf{u}_{1:t} = (\mathbf{u}_1, \mathbf{u}_2, ..., \mathbf{u}_{:t})$ である）．今回の例では，ロボットに与えられる並進，角速度が制御入力 $\mathbf{u}_t = (v_t\ \omega_t)^\top$ であり，GNSS による測位結果が観測値 $\mathbf{z}_t = \mathbf{x}_t^{\mathrm{GNSS}}$ である．パーティクルフィルタを用いて自己位置推定を実施する場合，以下のステップを繰り返すことになる．

1. ロボットの初期位置に合わせてパーティクルをばらまく
2. 制御入力にあわせてパーティクル群の位置を更新
3. センサ観測にしたがってパーティクルの重みを計算
4. 重みを正規化し，パーティクル群の位置の重み付き平均を推定値とする
5. 不要なパーティクルの削除と有効パーティクルの複製（リサンプリング）
6. 2 へ戻る

なおパーティクルとは，位置と重みを持つもの $\mathbf{s}_t = (\mathbf{x}_t, p_t)$ である．パーティクルフィルタを端的に述べれば，多数のパーティクルをロボットが存在しそうな場所にばらまき，センサ観測を基にその良し悪しを評

[*3] 全球測位衛星システムというと GPS が馴染み深い言葉と思うが，GPS は米国の GNSS であり，GNSS の述べた方が汎用性が高い．また 1 つの GNSS レシーバーを用いた場合は位置しか得られないが，複数の GNSS レシーバーを用いることで姿勢に関しても得ることができる．

価し，良さそうなパーティクルのみを残し，自己位置推定を行うというものである．以下，それぞれの処理の実装方法に関して述べる．なおパーティクルフィルタは，再帰的ベイズフィルタとしても定式化できる．これに関しては，5 章にて解説する．

4.3.1 初期化

ロボットが軌道追従を開始する地点（初期位置 $\mathbf{x}_0$）は与えられているものとする．このとき，それぞれのパーティクルの初期位置 $\mathbf{x}_0^{[i]}$ を以下の様に定める．

$$\begin{pmatrix} x_0^{[i]} \\ y_0^{[i]} \\ \theta_0^{[i]} \end{pmatrix} = \begin{pmatrix} \mathcal{N}(x_0, \sigma_x^2) \\ \mathcal{N}(y_0, \sigma_y^2) \\ \mathcal{N}(\theta_0, \sigma_\theta^2) \end{pmatrix} \tag{9}$$

4.3.2 パーティクル位置の更新

制御入力 $\mathbf{u}_t = (v_t \ \omega_t)^\top$ が与えられたとき，パーティクル群の姿勢を以下の様に更新する．

$$\begin{pmatrix} x_{t+1}^{[i]} \\ y_{t+1}^{[i]} \\ \theta_{t+1}^{[i]} \end{pmatrix} = \begin{pmatrix} x_t^{[i]} \\ y_t^{[i]} \\ \theta_t^{[i]} \end{pmatrix} + \begin{pmatrix} v_t^{[i]} \cos\theta_t^{[i]} \\ v_t^{[i]} \sin\theta_t^{[i]} \\ \omega_t^{[i]} \end{pmatrix} \Delta t \tag{10}$$

ここで，

$$\begin{aligned} v_t^{[i]} &\sim \mathcal{N}(v_t, \sigma_v^2) \\ \omega_t^{[i]} &\sim \mathcal{N}(\omega_t, \sigma_\omega^2) \end{aligned} \tag{11}$$

である．なお今回の例では，パーティクルフィルタによる自己位置推定を行った後に，ロボットが軌道追従を行うための制御入力を決定する．そのため $\mathbf{u}_t = (v_t \ \omega_t)^\top$ としては，1 ステップ前の制御入力を用いるものとする．

4.3.3 パーティクルの重みの更新

パーティクル位置の更新を終えた後に，パーティクルの重みをセンサ観測値に合わせて更新する．パーティクルの重みは以下の様に計算できるものとする．

$$p_t^{[i]} = \frac{1}{(\sqrt{2\pi})^3\sqrt{|\Sigma^{\mathrm{GNSS}}|}} \exp\left(-\frac{1}{2}(\mathbf{z}_t - \mathbf{x}_t^{[i]})^\top \Sigma^{-1}(\mathbf{z}_t - \mathbf{x}_t^{[i]})\right) \tag{12}$$

なお，$\Sigma^{\mathrm{GNSS}} = \mathrm{diag}(\sigma_x^2, \sigma_y^2, \sigma_\theta^2)$ である．すなわち，GNSS の測位結果に近い位置に存在するパーティクル程高い重みを持つことになる．

4.3.4 重みの正規化と自己位置推定

重みの計算を終えた後に，全てのパーティクルの重みの総和が 1 となる様に正規化を行う．

$$p_t^{[i]} \leftarrow \frac{p_t^{[i]}}{\sum_{j=1}^{M} p_t^{[j]}} \tag{13}$$

ここで M はパーティクル数である．パーティクルの正規化を終えた後に，パーティクル位置の重み付き平均を自己位置推定結果とする．

$$\mathbf{x}_t = \sum_{i=1}^{M} p_t^{[i]} \mathbf{x}_t^{[i]} \tag{14}$$

なお，重み付き平均を取得するという意味で式 (14) の様に記載しているが，角度に関する重み付き計算は工夫が必要なため注意すること．

4.3.5 リサンプリング

リサンプリングの役割は，不要なパーティクルを削除し，有効なパーティクルを複製することである．これにより，過度にパーティクル群が広がることを抑えることができ，自己位置推定の精度を維持することができる．リサンプリングの方法は様々あるが，ここでは一例を紹介する．

まず，以下の様な値を計算する．

$$b_t^{[i]} = \sum_{j=1}^{i} p_t^{[j]} \tag{15}$$

$b_t^{[i]}$ は，正規化された i 番目までのパーティクルの重みの和であり，$b_t^{[M]} = 1$ である．次に，0 から 1 の間の乱数 $r^{[i]} \sim \mathrm{unif}(0,1)$ を M 回発生させる．そして，$r^{[i]} \leq b_t^{[j]}$ となるとき，i 番目のパーティクルを j 番目のパーティクルで置き換えるという操作を M 回繰り返す．この様にすることで，乱数に従いながら重みの大きなパーティクルを残すことができる．

4.4 演習

4.4.1 パーティクルフィルタを用いた自己位置推定の実装

pf_path_follow ディレクトリ内の main.cpp を以下の様に編集せよ．

- 125 行目から，パーティクル群を初期化せよ．
- 152 行目から，パーティクル群の位置を過去の制御入力に従って更新せよ．また，位置の更新後に GNSS の観測値を用いて各パーティクルの重みを計算せよ．
- 155 行目から，重み付き平均を計算せよ．
- 168 行目から，パーティクルのリサンプリングを行え．

編集を終えたら，

```
$ ./make.sh
$ ./pf_path_follow
```

のコマンドを実行し，自己位置推定を行いながら，点列軌道の追従が行えることを確認せよ．

なお，pf_path_follow を実行すると，同ディレクトリ内に sim_result.txt が生成される．これが自己位置結果を保存したログファイルとなっている．以下のコマンドを実行することで，シミュレーション結果の確認を行うことができる．

```
$ gnuplot plot.gpt
```

4.4.2 制御パラメータ変更による挙動の変更の確認

pf_path_follow ディレクトリ内の main.cpp 内のパラメータを変え，以下のことを確認せよ．

- 120 行目の usePFEst を false とし，パーティクルフィルタによる状態推定を行わず，推定位置を式 (1) のみで更新した場合，真値と推定値がどの様になるか確認せよ．

- パーティクルフィルタを用いた自己位置推定を行うことで，真値と推定値がどの様になるか確認せよ．
- GNSS に関するノイズを大きくした場合自己位置推定に失敗してしまう．その理由を述べよ．
- パーティクル数を多くすることはどの様な利点と欠点をもたらすか述べよ．

5　再帰的ベイズフィルタによる自己位置推定の定式化

最後に，再帰的ベイズフィルタによる自己位置推定の定式化に関して述べる．4 章では，パーティクルフィルタによる自己位置推定の概略のみを述べた．パーティクルフィルタによる自己位置推定は非常に直感的であり，理解がしやすく，実装も容易でありかつ性能が高いという利点がある．しかしこういったことが逆にデメリットとなり，パーティクルフィルタを直感的に改良し，パーティクルフィルタが持つ確率的な意味合いを無視してしまうことも多い．パーティクルフィルタの確率的な定式化を理解することは，パーティクルフィルタを発展，改良するにあたり非常に効果的である．

5.1　再帰的ベイズフィルタの定式化

まず，以下に示す確率分布を求めることを考える．

$$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t}) \tag{16}$$

これは，時刻 1 から t までの間の制御入力列とセンサ観測列が与えられた下で，時刻 t における自己位置に関する確率分布を求めるということである．まず，$\mathbf{z}_t$ を用いてベイズの定理を適用する．

$$\begin{aligned} p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t}) &= \frac{p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})}{p(\mathbf{z}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})} \\ &= \eta p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) \end{aligned} \tag{17}$$

なお，$p(\mathbf{z}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ は正規化のために用いられる分布であり，正規化係数 η を用いて省略した．

次に $p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ に着目する．この分布は時刻 t におけるセンサ観測値に関する分布である．これに影響を与えるのは，基本的には現時刻の位置のみである．そのため，

$$p(\mathbf{z}_t|\mathbf{x}_t, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) = p(\mathbf{z}_t|\mathbf{x}_t) \tag{18}$$

と変形できる．この分布は観測モデルと呼ばれる．

次に $p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ に着目する．今考えている問題は，連続的に移動するロボットの自己位置を推定する問題であり，過去のロボットの位置も使って現在位置を推定できる方が好ましい．そこで，$\mathbf{x}_{t-1}$ を用いて，全確率の定理を適用する．

$$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})\mathrm{d}\mathbf{x}_{t-1} \tag{19}$$

ここで $p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ に着目する．この分布は現時刻の位置に関する分布であり，これに影響を与えるのは 1 時刻前の位置と現在の制御入力のみである．そのため，

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) \tag{20}$$

と変形できる．この分布は動作モデルと呼ばれる．また $p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ に着目すると，これは 1 時刻前の位置に関する分布であり，未来の制御入力である $\mathbf{u}_t$ が影響することはない．そのため，

$$p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) = p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \tag{21}$$

と書き換えることができる．

これらの式変形の結果をまとめると次式を得る．

$$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{x}) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \mathrm{d}\mathbf{x}_{t-1} \tag{22}$$

これは，$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ に関する再起式となっている．つまり，$p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1})$ が動作モデルを用いて更新され，観測モデルを掛けることで時刻 t の分布に更新される．ベイズの定理を適用して得られたこの再起式を，再帰的ベイズフィルタと呼ぶ．

5.2 パーティクルフィルタとの対応

4 章では，パーティクルフィルタとは，制御入力を基にパーティクル群をばらまき，センサ観測を基にそれぞれの良し悪しを判断し，良さそうなパーティクルを残すことで自己位置を推定する方法であると端的に述べた．その一方で，パーティクルフィルタは，5.1 節で述べたベイズフィルタの 1 つの実装例とも呼べる．本節では，パーティクルフィルタと再帰的ベイズフィルタの対応について述べる．

まずパーティクルフィルタとは，重みを持つ有限のパーティクル集合により，任意の確率分布を近似する方法である．例えば，ある状態 $\mathbf{x}$ に関する確率分布を，以下の様に近似する．

$$p(\mathbf{x}) \simeq \sum_{i=1}^{M} p^{[i]} \delta(\mathbf{x} - \mathbf{x}^{[i]}) \tag{23}$$

$\delta(\cdot)$ はディラックのデルタ関数であり，括弧内が零のときに 1，そうでなければ 0 を返す関数である．なお，$p^{[i]}$ はパーティクルの重みであり，$\sum_{i=1}^{M} p^{[i]} = 1$ である．

今，時刻 $t-1$ までの制御入力 $\mathbf{u}_{1:t-1}$ とセンサ観測 $\mathbf{z}_{1:t-1}$ が入力され，パーティクルフィルタによる自己位置推定の処理が終えられているとする．このパーティクルの分布は，

$$p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \tag{24}$$

を近似する．ここからパーティクルフィルタによる処理を行うにあたり，まず時刻 t における制御入力 $\mathbf{u}_t$ を用いて，パーティクル群の位置の更新を行う．この更新は，式 (10) に示す操作であり，これにより更新されたパーティクル群は，

$$\int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \mathrm{d}\mathbf{x}_{t-1} \tag{25}$$

を近似する．確率分布の積の積分という演算を見たとき，初学者からすると想像が難しいものであると思う．しかしパーティクルフィルタにおいては，制御入力に基づいてパーティクル群を更新するのみで，この分布を近似できてしまうのである．

パーティクル群を更新した後に，センサ観測を用いてその重みを式 (12) を用いて計算した．この重みは，ベイズフィルタ上では，

$$p(\mathbf{z}_t|\mathbf{x}_t) \tag{26}$$

に対応する．すなわち，観測モデルを用いてパーティクルの尤度が定まるのである．なお式 (12) では正規分布を仮定しているが，それ以外の分布を仮定することも可能である．そして，重みの計算されたパーティクル群は，

$$p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \mathrm{d}\mathbf{x}_{t-1} \tag{27}$$

を近似する．そして，パーティクル群の重みを正規化することで，

$$\eta p(\mathbf{z}_t|\mathbf{x}_t) \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) \mathrm{d}\mathbf{x}_{t-1} \tag{28}$$

が近似される．これはベイズフィルタを用いて求めるべき分布 $p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ に一致する．

パーティクル群の重みを正規化した後に，式 (14) に示す様に，パーティクル位置の重み付き平均を計算し，自己位置推定結果とした．もう一度述べるが，重みの正規化されたパーティクル群は，$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ を近似しているだけである．しかし，我々の目標はあくまでロボットの正しい位置を知り，軌道追従を行うことであり．この目的達成のためには，自己位置に関する分布を知れることは不十分であり，ここから何かしらの代表値を取得する必要がある．ここで，パーティクル位置の重み付き平均を計算することは，$p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ の期待値を計算していることと一致する．すなわち式 (14) に示す計算により，パーティクル群が示す確率分布の期待値を取得できるのである．なお，リサンプリングを行ったパーティクル群も，同様に $p(\mathbf{x}_t|\mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ を近似するが，リサンプリング後のパーティクル群からは同様の期待値は計算できないことに注意すること．

5.3 演習

- 再帰的ベイズフィルタの実装方法の 1 つに，カルマンフィルタがある．カルマンフィルタについて調べ，カルマンフィルタとベイズフィルタの対応について述べよ．
- カルマンフィルタとパーティクルフィルタを比較し，それぞれの利点，欠点を述べよ．
- 本研修の自己位置推定では，xy 座標での位置と姿勢角の 3 つの要素を扱った．しかし自己位置推定を 3 次元に拡張すると，xyz 座標上の位置とその姿勢角の 6 つの要素を扱うことになる．またさらなる応用を考えると，それぞれの方向の速度や加速度も知りたいという要望もあり，推定する要素は 12，18 に増える．この場合，パーティクルフィルタによる状態（位置，速度，加速度）の推定を行うことが妥当といえない理由を述べよ．また，どの様な工夫を行えば，パーティクルフィルタによる推定を有効に保てるか考えよ．

6 おわりに

本研修では，移動ロボットの軌道追従，および簡易な自己位置推定に関して述べた．軌道追従のためには，PID 制御を用いた．自己位置推定のためには，パーティクルフィルタを用いた．本稿で述べた方法は初歩的なものであり，実応用にあたっては不具合を起こす場合も多い．より応用的な議論については，別途他の書籍を参考にされたい [2].

参考文献

[1] S. Thrun, W. Burgard, and D. Fox. “Probabilistic Robotics,” *MIT Press*, 2005.

[2] 赤井直紀．“LiDAR を用いた高度自己位置推定システム: 移動ロボットのための自己位置推定の高性能化とその実装例”，コロナ社，2022.