Implementazione di un Particle Filter per la Localizzazione di un Carrello Elevatore

Francesco Caligiuri
Matricola: 146666
Scienze Informatiche
Anno Accademico: 2024/2025

Platforms and Algorithms for Autonomous Veichles

Indice

1 Introduzione

In questo progetto, è stato implementato un Particle Filter per la localizzazione di un carrello elevatore utilizzando dati LiDAR e landmark come riferimento. L'obiettivo è sviluppare un filtro in grado di stimare con precisione la posizione del veicolo durante l'intera simulazione, analizzando l'effetto di diversi parametri sul suo comportamento.

2 Implementazione del Particle Filter

Il Particle Filter è stato implementato seguendo i passaggi fondamentali:

- Inizializzazione delle particelle
- Predizione dello stato delle particelle
- Aggiornamento dei pesi delle particelle in base alle osservazioni
- Resampling delle particelle in base ai pesi aggiornati

Di seguito, vengono spiegate le funzioni implementate, evidenziando le parti di codice aggiunte e le scelte fatte.

2.1 Inizializzazione delle Particelle

Sono stati implementati due metodi di inizializzazione:

2.1.1 Inizializzazione Casuale (`init_random()`)

La funzione `init_random()` inizializza le particelle in posizioni casuali entro i limiti della mappa. Questo è utile quando non si dispone di una stima iniziale precisa.

Listing 2.1: Funzione `init_random()` con commenti

```
1  void ParticleFilter::init_random(double std[], int nParticles) {
2      num_particles = nParticles;
3
4      // Definizione dei limiti della mappa (da adattare in base alla
           mappa utilizzata)
5      double x_min = 0.0;
6      double x_max = 10.0;
7      double y_min = 0.0;
8      double y_max = 10.0;
9
10     // Distribuzioni uniformi per x, y e theta
11     std::uniform_real_distribution<double> dist_x(x_min, x_max);
12     std::uniform_real_distribution<double> dist_y(y_min, y_max);
13     std::uniform_real_distribution<double> dist_theta(-M_PI, M_PI);
14
15     // Inizializzazione delle particelle
16     for (int i = 0; i < num_particles; ++i) {
17         Particle p;
18         p.x = dist_x(gen);          // Posizione x casuale
19         p.y = dist_y(gen);          // Posizione y casuale
```

```
20          p.theta = dist_theta(gen);  // Orientamento theta casuale
21          particles.push_back(p);
22      }
23
24      is_initialized = true;  // Flag di inizializzazione impostato a true
25  }
```

Scelte Fatte:

- Sono state utilizzate distribuzioni uniformi per inizializzare le posizioni x, y e l'orientamento θ delle particelle all'interno dei limiti definiti della mappa.
- Questo approccio garantisce una copertura uniforme dell'area di interesse quando non si ha una stima iniziale.

2.1.2 Inizializzazione Basata su Stima Iniziale (`init()`)

La funzione `init()` inizializza le particelle attorno a una stima iniziale (x, y, θ), aggiungendo rumore gaussiano per rappresentare l'incertezza.

Listing 2.2: Funzione `init()` con commenti

```
1  void ParticleFilter::init(double x, double y, double theta, double std
       [], int nParticles) {
2      num_particles = nParticles;
3
4      // Distribuzioni normali centrate sulla stima iniziale
5      std::normal_distribution<double> dist_x(x, std[0]);       // Rumore
           su x
6      std::normal_distribution<double> dist_y(y, std[1]);       // Rumore
           su y
7      std::normal_distribution<double> dist_theta(theta, std[2]); //
           Rumore su theta
8
9      // Inizializzazione delle particelle
10     for (int i = 0; i < num_particles; ++i) {
11         Particle p;
12         p.x = dist_x(gen);          // Posizione x con rumore
13         p.y = dist_y(gen);          // Posizione y con rumore
14         p.theta = dist_theta(gen);  // Orientamento theta con rumore
15         particles.push_back(p);
16     }
17
18     is_initialized = true;  // Flag di inizializzazione impostato a true
19 }
```

Scelte Fatte:

- Sono state utilizzate distribuzioni normali centrate sulla stima iniziale, con deviazioni standard specificate da σ_{init}.
- Questo permette di inizializzare le particelle attorno alla posizione stimata, riflettendo l'incertezza iniziale.

2.2 Predizione dello Stato delle Particelle (`prediction()`)

La funzione `prediction()` aggiorna lo stato delle particelle in base al modello di movimento e aggiunge rumore gaussiano.

Listing 2.3: Funzione `prediction()` con commenti

```
void ParticleFilter::prediction(double delta_t, double std_pos[], double
    velocity, double yaw_rate) {
    // Creazione delle distribuzioni per il rumore gaussiano
    std::normal_distribution<double> dist_x(0, std_pos[0]);      //
       Rumore su x
    std::normal_distribution<double> dist_y(0, std_pos[1]);      //
       Rumore su y
    std::normal_distribution<double> dist_theta(0, std_pos[2]);  //
       Rumore su theta

    for (auto& particle : particles) {
        if (fabs(yaw_rate) < 1e-5) {
            // Movimento rettilineo (yaw_rate prossimo a zero)
            particle.x += velocity * delta_t * cos(particle.theta);
            particle.y += velocity * delta_t * sin(particle.theta);
            // Orientamento theta rimane invariato
        } else {
            // Movimento rotazionale
            particle.x += (velocity / yaw_rate) * (sin(particle.theta +
               yaw_rate * delta_t) - sin(particle.theta));
            particle.y += (velocity / yaw_rate) * (-cos(particle.theta +
                yaw_rate * delta_t) + cos(particle.theta));
        }

        particle.theta += yaw_rate * delta_t;  // Aggiornamento di theta

        // Normalizzazione dell'angolo theta nell'intervallo [-pi, pi]
        while (particle.theta > M_PI) particle.theta -= 2.0 * M_PI;
        while (particle.theta < -M_PI) particle.theta += 2.0 * M_PI;

        // Aggiunta del rumore gaussiano
        particle.x += dist_x(gen);
        particle.y += dist_y(gen);
        particle.theta += dist_theta(gen);
    }
}
```

Scelte Fatte:

- È stata gestita separatamente la situazione in cui il tasso di rotazione ($\dot{\psi}$) è molto piccolo (movimento rettilineo) e il caso generale (movimento curvilineo).
- L'aggiunta del rumore gaussiano dopo l'aggiornamento dello stato garantisce che tutte le particelle siano soggette all'incertezza del movimento.

- È stata effettuata la normalizzazione dell'angolo θ per mantenerlo nell'intervallo $[-\pi, \pi]$.

2.3 Aggiornamento dei Pesi delle Particelle (`updateWeights()`)

La funzione `updateWeights()` aggiorna i pesi delle particelle in base alla probabilità delle osservazioni date le posizioni dei landmark.

Listing 2.4: Funzione `updateWeights()` con commenti

```
1  void ParticleFilter::updateWeights(double std_landmark[], std::vector<
       LandmarkObs> observations, Map map_landmarks) {
2      for (auto& particle : particles) {
3          // Passo 1: Trasformazione delle osservazioni nel sistema di
               riferimento della mappa
4          std::vector<LandmarkObs> transformed_observations;
5          for (const auto& obs : observations) {
6              transformed_observations.push_back(transformation(obs,
                   particle));
7          }
8
9          // Passo 2: Associazione delle osservazioni ai landmark della
               mappa
10         std::vector<LandmarkObs> mapLandmarks;
11         for (const auto& lm : map_landmarks.landmark_list) {
12             mapLandmarks.push_back(LandmarkObs{lm.id_i, lm.x_f, lm.y_f})
                   ;
13         }
14         dataAssociation(mapLandmarks, transformed_observations);
15
16         // Passo 3: Aggiornamento dei pesi delle particelle
17         particle.weight = 1.0;
18         double sigma_x = std_landmark[0];
19         double sigma_y = std_landmark[1];
20         double gauss_norm = 1 / (2 * M_PI * sigma_x * sigma_y);
21
22         for (const auto& obs : transformed_observations) {
23             // Ricerca del landmark associato
24             LandmarkObs landmark;
25             for (const auto& lm : mapLandmarks) {
26                 if (lm.id == obs.id) {
27                     landmark = lm;
28                     break;
29                 }
30             }
31
32             // Calcolo della differenza tra osservazione e landmark
33             double dx = obs.x - landmark.x;
34             double dy = obs.y - landmark.y;
35
```

```
36              // Calcolo dell'esponente della distribuzione gaussiana
37              double exponent = (dx * dx) / (2 * sigma_x * sigma_x) + (dy
                    * dy) / (2 * sigma_y * sigma_y);
38
39              // Calcolo del peso usando la distribuzione gaussiana
                    multivariata
40              double weight = gauss_norm * exp(-exponent);
41
42              // Aggiornamento del peso della particella
43              particle.weight *= weight;
44          }
45      }
46  }
```

Scelte Fatte:

- Le osservazioni sono state trasformate dal sistema di riferimento del veicolo a quello globale utilizzando la funzione `transformation()`.
- L'associazione dei dati viene effettuata nella funzione `dataAssociation()`, associando ogni osservazione al landmark più vicino.
- I pesi delle particelle vengono aggiornati calcolando la probabilità delle osservazioni utilizzando la distribuzione gaussiana multivariata.

2.4 Resampling delle Particelle (`resample()`)

La funzione `resample()` esegue il resampling delle particelle in base ai loro pesi.

Listing 2.5: Funzione `resample()` con commenti

```
1  void ParticleFilter::resample() {
2      std::vector<Particle> new_particles;
3      std::vector<double> weights;
4
5      // Estrazione dei pesi delle particelle
6      for (const auto& particle : particles) {
7          weights.push_back(particle.weight);
8      }
9
10     // Distribuzioni per la selezione casuale
11     std::uniform_real_distribution<double> dist_double(0.0, *max_element
           (weights.begin(), weights.end()));
12     std::uniform_int_distribution<int> dist_int(0, num_particles - 1);
13
14     int index = dist_int(gen);  // Indice iniziale casuale
15     double beta = 0.0;
16
17     // Resampling stocastico universale (ruota)
18     for (int i = 0; i < num_particles; ++i) {
```

```
19          beta += dist_double(gen) * 2.0;
20          while (beta > weights[index]) {
21              beta -= weights[index];
22              index = (index + 1) % num_particles;
23          }
24          new_particles.push_back(particles[index]);
25      }
26
27      particles = new_particles;  // Aggiornamento delle particelle
28  }
```

Scelte Fatte:

- È stato implementato il metodo di resampling basato sulla ruota (*Resampling Stocastico Universale*).
- Questo metodo seleziona le particelle proporzionalmente al loro peso, mantenendo quelle con peso maggiore.

3 Esperimenti e Risultati

Sono stati eseguiti tre esperimenti variando il numero di particelle e il rumore, per osservare l'effetto su precisione e stabilità del filtro. Per ciascun caso, sono state prodotte immagini che mostrano il comportamento del filtro in diverse configurazioni.

3.1 Esperimento 1: Variazione del Numero di Particelle

Configurazione:

- Numero di particelle: $N = 50$, $N = 200$, $N = 500$
- Rumore di movimento: $\sigma_{pos} = [0.15, 0.15, 0.15]$
- Rumore del sensore: $\sigma_{landmark} = [0.3, 0.3]$

3.1.1 Risultati e Analisi

Con $N = 50$ particelle, si osserva un notevole tremolio e incertezza nella localizzazione (Figura 3.1). La traiettoria stimata è instabile e presenta deviazioni significative dalla traiettoria reale.

Aumentando il numero di particelle a $N = 200$, il tremolio è notevolmente ridotto (Figura 3.2). Tuttavia, permane una certa incertezza nei movimenti più imprevedibili, specialmente durante le curve strette.

Con $N = 500$ particelle, l'incertezza è ulteriormente diminuita (Figura 3.3), e la traiettoria stimata aderisce meglio a quella reale. Tuttavia, l'aumento del numero di particelle comporta un incremento significativo della computazione e del tempo di elaborazione.

Conclusione:

Un numero maggiore di particelle migliora la precisione della localizzazione, ma aumenta il costo computazionale. Un compromesso ragionevole è utilizzare $N = 200$ particelle per bilanciare precisione ed efficienza.

3.2 Esperimento 2: Variazione del Rumore di Movimento

Configurazione:

- Numero di particelle: $N = 200$
- Rumore di movimento:
  - Caso A: $\sigma_{pos} = [0.15, 0.15, 0.15]$
  - Caso B: $\sigma_{pos} = [0.05, 0.05, 0.05]$
  - Caso C: $\sigma_{pos} = [0.01, 0.01, 0.01]$
- Rumore del sensore: $\sigma_{landmark} = [0.3, 0.3]$

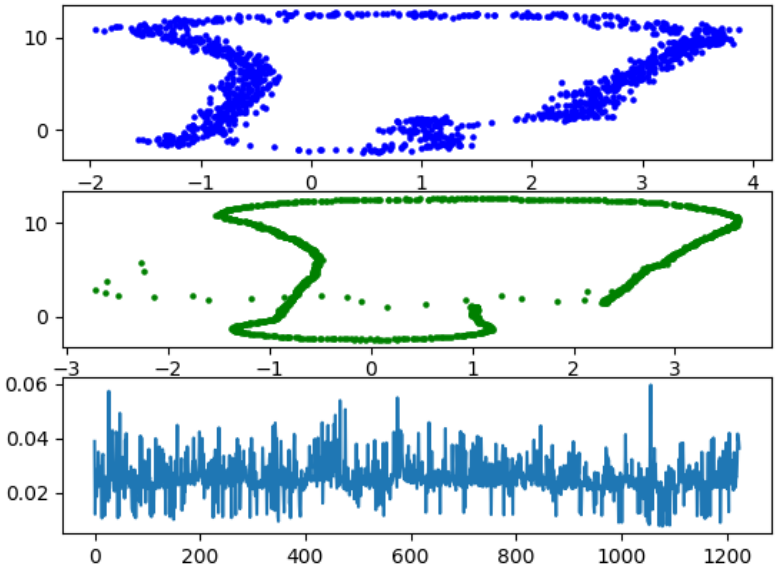

Figura 3.1: Traiettoria stimata con $N = 50$ particelle

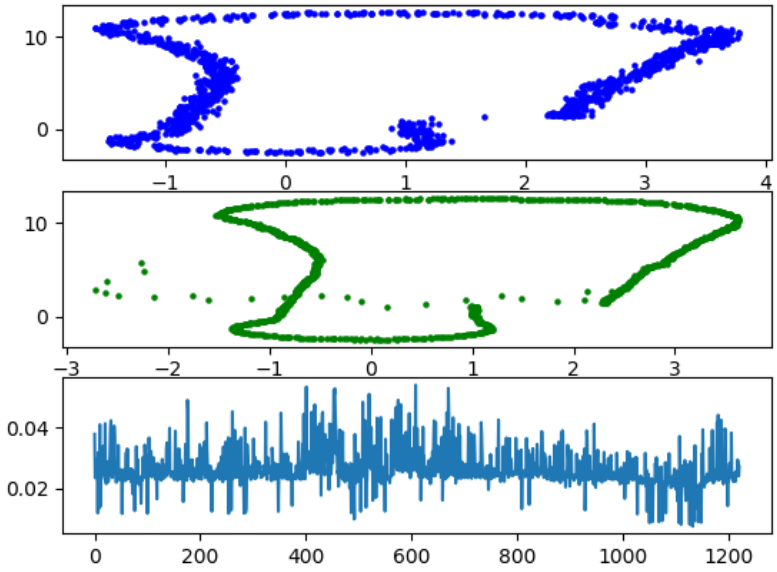

Figura 3.2: Traiettoria stimata con $N = 200$ particelle

3.2.1 Risultati e Analisi

Con $\sigma_{pos} = 0.15$ (Figura 3.4), si osserva poco tremolio, ma nelle curve il filtro perde leggermente precisione, mostrando una deviazione dalla traiettoria reale.

Riducendo il rumore a $\sigma_{pos} = 0.05$ (Figura 3.5), la stabilità aumenta e il tremolio è ulteriormente ridotto. Questa configurazione sembra offrire la soluzione ottimale, bilanciando stabilità e adattabilità.

Con $\sigma_{pos} = 0.01$ (Figura 3.6), il filtro è molto stabile, ma la ridotta variabilità impedisce alle particelle di adattarsi ai cambiamenti improvvisi. Questo porta a una perdita completa della navetta, con le particelle che si spostano su una traiettoria errata. La mancanza di flessibilità impedisce di trovare i landmark, compromettendo la localizzazione.

Conclusione:

Ridurre il rumore di movimento aumenta la stabilità delle predizioni, ma un rumore troppo basso limita la capacità del filtro di adattarsi ai cambiamenti. Un valore di $\sigma_{pos} = 0.05$ offre un buon compromesso tra stabilità e flessibilità.

3.3 Esperimento 3: Variazione del Rumore del Sensore

Configurazione:

- Numero di particelle: $N = 200$
- Rumore di movimento: $\sigma_{pos} = [0.15, 0.15, 0.15]$
- Rumore del sensore:
  - Caso A: $\sigma_{landmark} = [0.5, 0.5]$
  - Caso B: $\sigma_{landmark} = [0.3, 0.3]$
  - Caso C: $\sigma_{landmark} = [0.1, 0.1]$

3.3.1 Risultati e Analisi

Con $\sigma_{landmark} = 0.5$ (Figura 3.7), si riscontra una minore stabilità nella localizzazione. Il filtro è meno preciso nell'associare le osservazioni ai landmark, a causa dell'elevato rumore del sensore.

Con $\sigma_{landmark} = 0.3$ (Figura 3.8), la stabilità aumenta rispetto al caso precedente. Il filtro riesce a localizzare il veicolo con maggiore precisione, mantenendo un tempo di computazione accettabile.

Riducendo ulteriormente il rumore a $\sigma_{landmark} = 0.1$ (Figura 3.9), si ottiene la massima stabilità e precisione nella localizzazione. Tuttavia, il costo computazionale aumenta significativamente rispetto alle soluzioni precedenti, a causa della maggiore sensibilità alle misurazioni del sensore.

Conclusione:

Una riduzione del rumore del sensore migliora la precisione della localizzazione, ma comporta un aumento del tempo di elaborazione. Un valore di $\sigma_{landmark} = 0.3$ rappresenta un buon compromesso tra stabilità e efficienza computazionale.

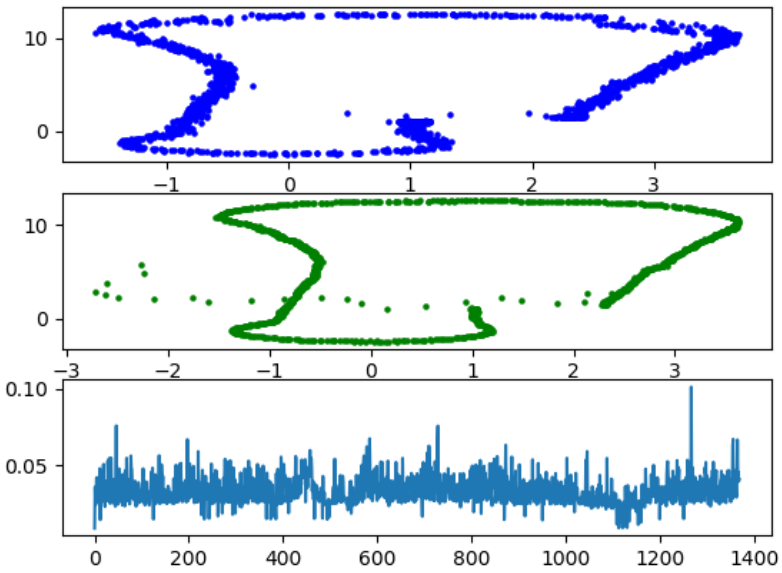

Figura 3.3: Traiettoria stimata con $N = 500$ particelle

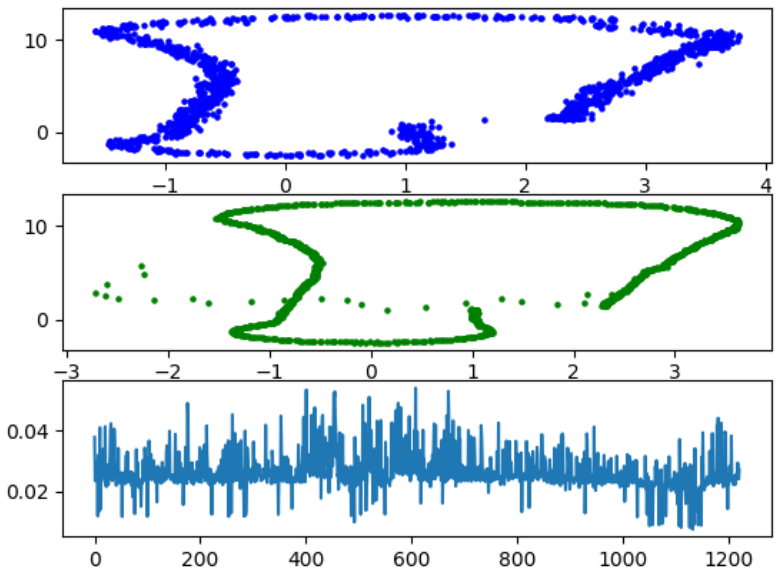

Figura 3.4: Traiettoria stimata con $\sigma_{pos} = 0.15$

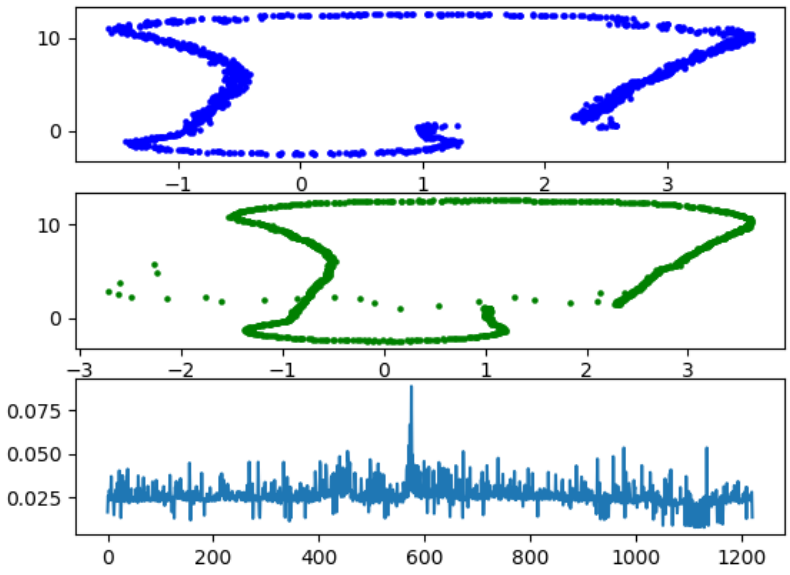

Figura 3.5: Traiettoria stimata con $\sigma_{pos} = 0.05$

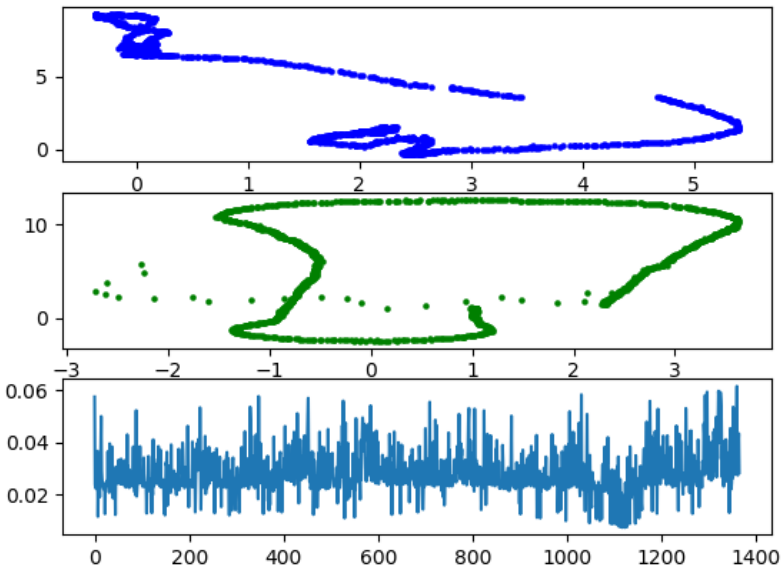

Figura 3.6: Traiettoria stimata con $\sigma_{pos} = 0.01$

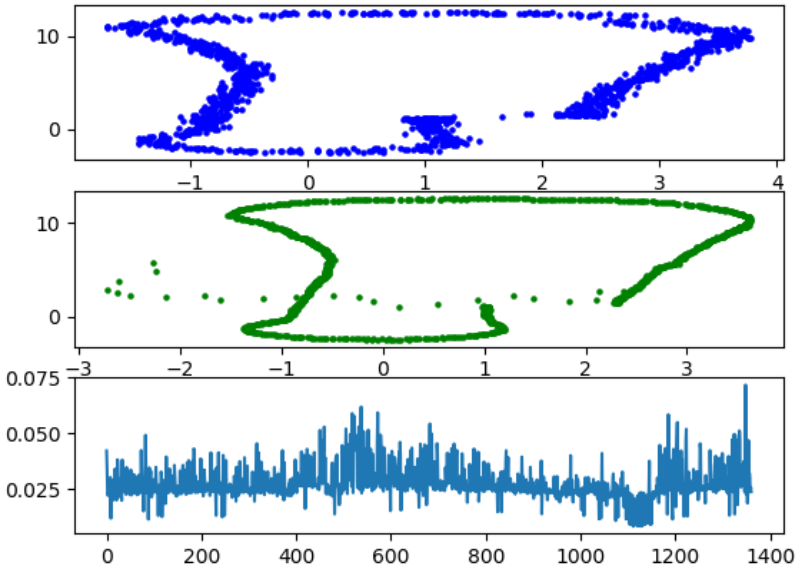

Figura 3.7: Traiettoria stimata con $\sigma_{landmark} = 0.5$

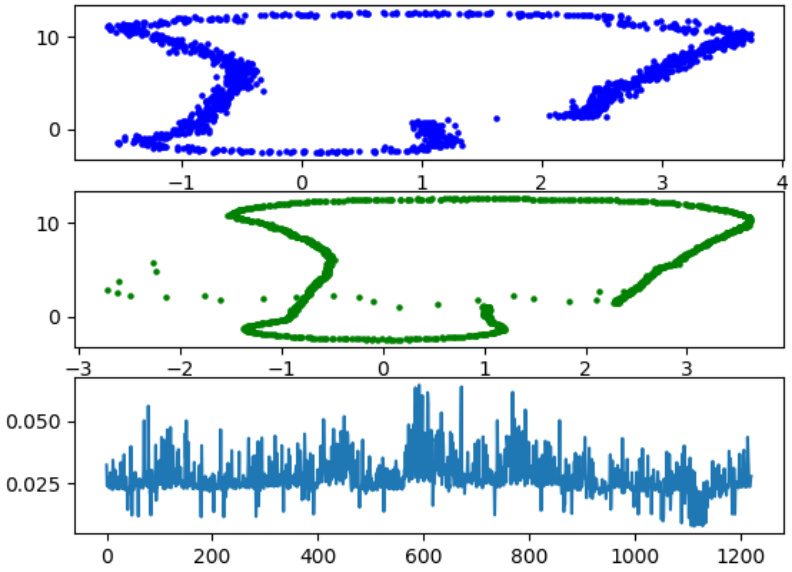

Figura 3.8: Traiettoria stimata con $\sigma_{landmark} = 0.3$

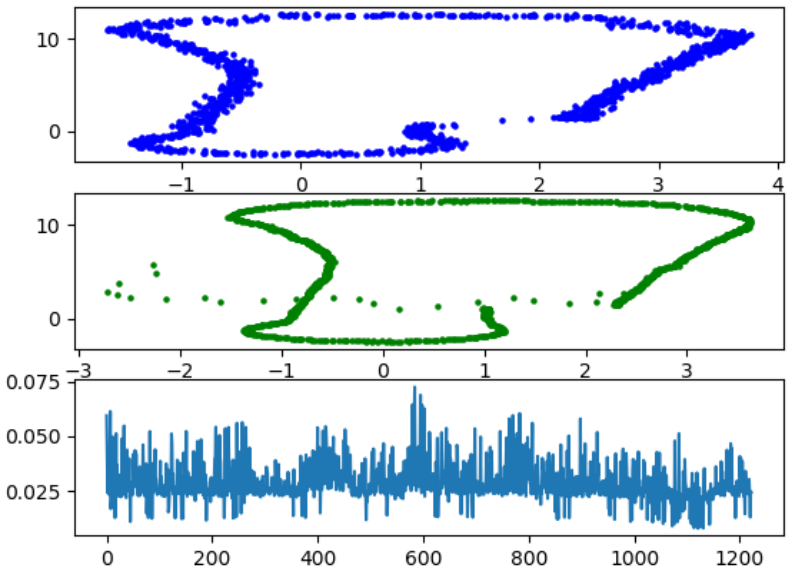

Figura 3.9: Traiettoria stimata con $\sigma_{landmark} = 0.1$

4 Conclusioni

È stato implementato con successo un Particle Filter per la localizzazione di un carrello elevatore utilizzando dati LiDAR e landmark. Le scelte fatte nelle funzioni implementate e l'analisi dei parametri hanno permesso di ottenere una localizzazione abbastanza precisa e stabile.

Dagli esperimenti condotti, è emerso che:

- Aumentare il numero di particelle migliora la precisione ma aumenta il tempo di elaborazione. Un valore di $N = 200$ offre un buon equilibrio.
- Ridurre il rumore di movimento a $\sigma_{pos} = 0.05$ diminuisce il tremolio delle predizioni senza compromettere l'adattabilità.
- Un rumore del sensore di $\sigma_{landmark} = 0.3$ garantisce una buona stabilità con un costo computazionale accettabile.

In futuro, potrebbe essere considerata l'implementazione di metodi di resampling più avanzati e ulteriori ottimizzazioni del codice per migliorare le prestazioni.