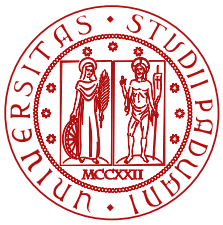

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Un sistema di navigazione probabilistico per robot mobili autonomi

Relatore:

Prof. Alberto Pretto

Laureando:

Paul Marin

ANNO ACCADEMICO 2023-2024

Data di laurea 25/09/2024

Abstract

Questa tesi presenta l'implementazione di un sistema di localizzazione per robot mobili basato su un filtro particellare (particle filter), progettato per un robot mobile dotato di sensori LiDAR e di odometria delle ruote. L'obiettivo primario di questa tesi è ottenere una localizzazione robusta e accurata in ambienti complessi e possibilmente dinamici. Il filtro particellare viene utilizzato per la sua efficacia nel gestire distribuzioni di probabilità multimodali e non gaussiane tipiche del mondo reale. Il sistema proposto integra i dati LiDAR per la percezione dell'ambiente e l'odometria delle ruote per la predizione del movimento, combinando queste fonti di informazione per migliorare l'accuratezza della localizzazione. Il particle filter stima la posa (posizione 2D ed orientazione) del robot gestendo un insieme di particelle, ognuna della quali rappresenta un'ipotesi di posa. Più probabile una regione, maggiore è la densità di particelle in tale regione. Le particelle vengono pesate integrando i dati LiDAR per permetterne il ricampionamento, affinando così la localizzazione nel tempo. Il sistema è stato implementato in C++ e testato su dataset pubblici. I risultati sperimentali ottenuti dimostrano la capacità del sistema di fornire una localizzazione accurata e robusta.

Keywords:

Mobile robots, Navigation, Particle filters

Indice

1 Introduzione **4**

1.1 Fondamenti della robotica mobile 4

1.2 Il robot aspirapolvere come applicazione nella robotica mobile 5

1.3 L'importanza della navigazione e della localizzazione 6

1.4 Scopo della tesi 9

2 Metodi probabilistici e particle filter **10**

2.1 Concetti preliminari della robotica probabilistica 11

2.1.1 Inferenza Probabilistica 11

2.1.2 Belief 15

2.2 Approccio Bayesiano e filtro di Bayes 16

2.3 Particle filter con metodo Monte Carlo 19

3 Localizzazione Monte Carlo e modelli di movimento e sensoriali **24**

3.1 Introduzione all'algoritmo di localizzazione Monte Carlo 24

3.2 Modello di movimento 26

3.3 Modello di misurazione sensoriale 27

4 Implementazione e risultati sperimentali **31**

4.1 Tecnologie e scelte progettuali 31

4.2 Implementazione del modello di misurazione sensoriale 34

4.3 Implementazione dell'algoritmo MCL 37

4.3.1 Aggiornamento del peso delle particelle 42

4.3.2 Ricampionamento delle particelle 43

4.4 Risultati Sperimentali 48

4.4.1 Risultati su fr079.png 52

4.4.2 Risultati su belgioioso.png 54

4.4.3 Risultati su orebro.png 56

5 Conclusioni **58**

Capitolo 1

Introduzione

1.1 Fondamenti della robotica mobile

La robotica mobile nasce dall'esigenza di sostituire l'uomo nell'esecuzione di compiti impegnativi, ripetitivi e in ambienti difficili. Questo disciplina, seppur giovane, si basa su molteplici rami dell'ingegneria tra cui, ingegneria meccanica, dell'automazione, elettronica, cosi come su campi interdisciplinari che combinano scienze cognitive e ingegneria, come l'informatica e l'intelligenza artificiale.
A differenza dei robot fissi, progettati per svolgere compiti in un'area ben determinata, i robot mobili sono in grado di spostarsi da un luogo all'altro e interagire in modo dinamico con l'ambiente.
L'obiettivo principale di questo settore è di ottenere un comportamento in grado di rispondere autonomamente alle sfide incontrate nell'ambiente in cui il robot viene utilizzato.
I recenti progressi in vari campi come la sensoristica, grazie ai sensori sempre più performanti e accurati, l'elaborazione dei dati in tempo reale, realizzata tramite chip sempre più performanti, e l'intelligenza artificiale hanno sbloccato nuove possibilità per i robot mobili.
La tecnologia moderna permette quindi ai robot di percepire l'ambiente circostante con maggiore precisione ed efficienza, elaborando le informazioni e permettendo loro di prendere decisioni autonome più velocemente.

Per poter comprendere meglio le caratteristiche dei robot mobili è utile elencare i principali aspetti che definiscono questa tipologia di robot.

Esse sono suddivise in caratteristiche tecniche e di progettazione:

- **La mobilità**, una caratteristica tecnica, sottintesa dato che si parla di robot mobili. I robot mobili sono dotati di diversi meccanismi di movimento che possono variare da ruote a cingoli o altri sistemi progettati specificamente per adattarsi all'ambiente in cui questi devono operare.

- **I sensori**, rappresentano un'altra caratteristica tecnica importante per i robot mobili. Possono usare uno oppure una combinazione di sensori come telecamere, LiDAR (Light Detection and Ranging), ultrasuoni e giroscopi. Lo scopo principale dei sensori è di rilevare ostacoli misurando distanze e raccogliendo dati ambientali riuscendo in questo modo di facilitare la navigazione.

- **La localizzazione e la navigazione** rappresentano caratteristiche chiave della progettazione per poter rendere autonomo un robot mobile. Mentre la mobilità e i sensori sono importanti per il movimento e la percezione dell'ambiente, la localizzazione permette al robot di determinare la propria posizione utilizzando tecniche come il filtro di Kalman e il particle filter.
  La navigazione invece, consente al robot di pianificare e seguire percorsi ottimali per raggiungere gli obiettivi prefissati, evitando gli ostacoli.
  Queste due funzionalità sono strettamente correlate, dato che una precisa localizzazione è essenziale per una pianificazione del percorso efficace e una navigazione accurata.

- **Gli algoritmi di controllo** permettono di gestire il movimento e l'interazione del robot con l'ambiente circostante, consentendo al robot di adattarsi in tempo reale a cambiamenti o situazioni impreviste.

1.2 Il robot aspirapolvere come applicazione nella robotica mobile

Un esempio tipico di robot mobile è rappresentato dai robot aspirapolvere, che stanno diventando sempre più comuni nelle nostre abitazioni. Questi dispositivi, oltre ad essere equipaggiati con ruote motorizzate, sono dotati di vari sensori.
Uno dei sensori più comuni nei robot aspirapolvere di fascia alta è il sensore LiDAR, che emette impulsi laser a intervalli regolari, permettendo di ottenere una scansione 2D dell'ambiente circostante. Inoltre, i dati sensoriali raccolti dal LiDAR vengono forniti a un insieme di algoritmi e tecniche utilizzati in robotica, chiamato SLAM (Simultaneous Localization and Mapping), che permettono al robot di localizzarsi all'interno di un ambiente sconosciuto e contemporaneamente di creare la mappa

dell'ambiente stesso. Dato che si tratta di un robot aspirapolvere, una delle sfide principali in un sistema indoor è rappresentata dalla stima precisa della posizione del robot, stima che non può essere aggiornata tramite GPS (Global Positioning System), in quanto i segnali satellitari non sono affidabili all'interno degli edifici. Per superare questa limitazione, all'interno dello SLAM, per la parte di localizzazione del robot, viene implementato uno dei vari algoritmi di localizzazione a seconda delle specifiche esigenze o delle caratteristiche dell'ambiente. Uno di questi è la localizzazione Monte Carlo o MCL (Monte Carlo Localization), un algoritmo probabilistico che utilizza un insieme di particelle per stimare la posizione del robot e che costituisce l'oggetto di studio del presente elaborato.

Figura 1.1: Robot aspirapolvere moderno

1.3 L'importanza della navigazione e della localizzazione

Avendo analizzato il robot aspirapolvere come esempio di robotica mobile, possiamo notare che esso si può suddividere in una parte hardware composta da vari sensori e un sistema di movimento, e da una parte software, dedicata all'elaborazione dei dati raccolti, alla localizzazione, alla navigazione e al controllo delle operazioni. Per chiarire ulteriormente la differenza tra navigazione e localizzazione, la navigazione comprende un insieme di algoritmi che consentono al robot di decidere come spostarsi all'interno della mappa, mentre la localizzazione mobile del robot consiste nella stima della posa attraverso l'acquisizione dei dati dai vari sensori e successivamente la rielaborazione delle informazioni ottenute dall'ambiente circostante.
Incentrando l'attenzione sul problema della movimentazione del robot, è importante considerare tutti i sistemi di localizzazione che permetto di percepire la sua posa.

Per **posa** si intende l'insieme delle coordinate (x,y) e il suo orientamento (θ) rispetto al sistema di coordinate della mappa. Determinare la posa di un robot rappresenta un compito complesso ma essenziale, dato che non è possibile misurarla direttamente, e richiede quindi di affrontare alcuni problemi per stimare con precisione la sua posizione.
Per comprendere meglio i principali fattori legati alla localizzazione, introduciamo alcuni concetti di base sui problemi di localizzazione e la loro complessità.

Le tipologie di localizzazione possono essere categorizzate in base al modo in cui vengono raccolti e utilizzati i dati per stimare la posizione del robot, alle informazioni iniziali che il robot ha sulla propria posizione, alla natura dell'ambiente in cui opera e in base alla gestione del movimento del robot durante il processo di localizzazione.
Considerando il modo in cui vengono e utilizzati i dati raccolti, è possibile distinguere tra sistemi di localizzazione **assoluta** e sistemi di localizzazione **relativa**. Nella prima, la stima della posizione del robot viene ottenuta esclusivamente dalle misurazioni raccolte in quell'istante, trattate in modo indipendente e senza integrare le informazioni passate. Tale procedura non si verifica invece per la localizzazione relativa, in cui la posizione corrente viene calcolata come un aggiornamento continuo rispetto a quella precedente, rendendo la stima fortemente influenzata dalle posizioni assunte nel tempo.
Basandosi sulla conoscenza iniziale del robot, questi problemi si distinguono tra localizzazione **locale** e **globale**.
La localizzazione **locale** presuppone che la posa iniziale del robot sia nota e l'obiettivo principale consiste nel tracciarla con precisione, correggendo i piccoli errori dovuti al rumore dei sensori e al movimento. Questo metodo, nel mondo della robotica, è associato al concetto di *Position Tracking*. Il **Position tracking** è la situazione in cui il robot dispone di una stima della posizione iniziale e mantiene una traccia accurata della propria posizione durante gli spostamenti, basandosi sull'analisi dei dati sensoriali raccolti.
Al contrario, nella localizzazione **globale** la posa iniziale del robot è sconosciuta, rendendo il processo più complesso. In questa situazione, il robot deve stimare da zero la propria posizione all'interno dell'ambiente. Questo processo, noto come *Global Localization*, si verifica quando il robot cerca di stimare la sua posizione assoluta all'interno della mappa, basandosi esclusivamente sull'analisi dei dati sensoriali di cui dispone.
Un'altra categoria di localizzazione, come discusso precedentemente, riguarda la **natura dell'ambiente** in cui il robot opera. In questa categoria, possiamo distinguere tra **ambienti statici** e **ambienti dinamici**.

Negli ambienti statici l'unica variabile è la posa del robot, mentre gli oggetti circostanti rimangono fissi e invariati. Questo rende la localizzazione più semplice grazie alla staticità dell'ambiente e al fatto che l'incertezza è principalmente dovuta ai movimenti del robot. Tale situazione si traduce nel fatto che, in ambienti stabili, le stime probabilistiche risultano più precise grazie all'assenza di cambiamenti esterni che complicano il processo. Al contrario, negli ambienti dinamici, oltre al movimento della posizione del robot, si aggiungono variabili supplementari, come la posizione degli oggetti circostanti, che può cambiare continuamente nel tempo. Questi cambiamenti influenzano in particolar modo le letture dei sensori e complicano la localizzazione, rendendo necessario aggiornare la mappa in modo continuo per tenere conto delle delle modifiche e ridurre le incertezze aggiuntive.
Esempi di tali cambiamenti includono il movimento di persone, variazioni nella luce del giorno nel caso di robot dotati di telecamere e spostamenti di mobili o porte.
Inoltre si possono introdurre altri due tipi di localizzazione, **passiva** e **attiva**, i quali definiscono come il robot interagisce con l'ambiente per determinare la posizione. Nella localizzazione attiva, l'algoritmo interviene direttamente nel controllo dei movimenti del robot con l'obiettivo di migliorare la precisione della sua posizione per minimizzare gli errori di localizzazione ed evitare situazioni impreviste.
Nel contesto della localizzazione passiva invece, il sistema di localizzazione si limita a osservare i movimenti del robot, senza intervenire. Per fare un esempio, si può considerare un robot che svolge le sue attività principali muovendosi in maniera casuale, senza prestare particolare attenzione della propria posizione all'interno dell'ambiente.

In definitiva la precisione della localizzazione è un elemento importante per garantire risultati affidabili in diverse applicazioni dei robot mobili, come la pulizia domestica, la navigazione autonoma e altre aree di utilizzo. Per affrontare le difficoltà nella localizzazione, verranno esplorati concetti come l'incertezza associata alla localizzazione, la probabilità condizionata, da cui deriva la regola di Bayes e il particle filter con il metodo di Monte Carlo. Questi strumenti saranno trattati in dettaglio nei capitoli successivi.

In questo elaborato, verrà affrontato il problema della **localizzazione globale**, in cui la posa iniziale del robot è sconosciuta. L'algoritmo, basato su un particle filter e implementato in linguaggio C++, distribuisce inizialmente le particelle casualmente all'interno della mappa e sfrutta i dati di odometria e le letture LiDAR per correggere e aggiornare continuamente la stima della posa. Questo processo consente al robot di convergere progressivamente verso la posa corretta, anche in assenza di una stima iniziale accurata. Il sistema opera in un **ambiente statico**, in cui la mappa

di input rimane invariata, rendendo la localizzazione più semplice grazie all'assenza di oggetti mobili o cambiamenti strutturali. La localizzazione implementata è di tipo **relativa**, in quanto aggiorna la posizione del robot basandosi sui movimenti precedenti e sulle nuove letture sensoriali. Infine, il sistema adotta una **localizzazione passiva**, in cui il robot raccoglie dati sensoriali senza influenzare attivamente il proprio movimento per ottimizzare la localizzazione.

1.4 Scopo della tesi

In questo elaborato, l'obiettivo finale è quello di implementare un sistema di localizzazione basato sul metodo Monte Carlo e di testare le sue varie applicazioni in ambienti diversi, realizzando dei grafici e confrontando le varie stime delle posizioni calcolate, con i metodi della media delle particelle e la stima della densità con Kernel (kernel density estimation, KDE). Verrà inoltre presentata un'attenta analisi in cui verranno testate diverse configurazioni dei parametri, tra cui:

- **Numero di particelle**: rappresenta il numero di particelle che verranno utilizzate per il funzionamento dell'algoritmo.

- **Deviazione standard per la traslazione**: rappresenta la misura della dispersione delle posizioni stimate delle particelle lungo una certa direzione (ad esempio, x o y). Essa quantifica quanto le particelle deviano dalla loro posizione durante il movimento traslazionale del robot.

- **Deviazione standard per la rotazione**: rappresenta la misura della dispersione degli angoli stimati delle particelle durante un movimento rotazionale del robot. Questa metrica quantifica quanto le particelle deviano dall'angolo dopo una rotazione.

- **Limite inferiore di una distribuzione Gaussiana**: rappresenta il limite inferiore che viene applicato a una distribuzione Gaussiana, tenendo conto della differenza tra il valore letto del LiDAR e la stima prevista in quella posizione per una particella nel calcolo del peso della stessa.

Nei prossimi capitoli verranno spiegati tutti gli elementi che compongono un particle filter e i loro scopi.

Capitolo 2

Metodi probabilistici e particle filter

La robotica probabilistica ('probabilistic robotics') è un approccio che tiene conto dell'incertezza sia nella percezione sia nelle azioni del robot. Il concetto fondamentale è di rappresentare in modo chiaro l'incertezza utilizzando la teoria della probabilità. Questo approccio consente al robot di navigare e prendere decisioni basate sulle informazioni ricevute dai sensori, anche quando queste informazioni sono parziali o affette da rumore, migliorando in questo modo le capacità di localizzazione e movimento del robot in ambienti di una certa complessità o in continua evoluzione. Gli algoritmi probabilistici hanno l'obiettivo di rappresentare le distribuzioni di probabilità relative allo stato del robot, considerando tutte le configurazioni possibili. Questo approccio permette di gestire le ambiguità in una via completamente matematica.
La pianificazione delle azioni viene realizzata in modo da minimizzare, ad ogni spostamento, l'incertezza della posa assunta dal robot. In conclusione, la robotica probabilistica integra percezione, pianificazione e controllo, permettendo al robot di adattarsi dinamicamente ai dati sensoriali e ridurre progressivamente l'incertezza sulla propria posizione.
Passiamo ora alla descrizione dei temi fondamentali della robotica probabilistica che permettono di comprendere i modelli probabilistici della localizzazione dei robot mobili.

2.1 Concetti preliminari della robotica probabilistica

2.1.1 Inferenza Probabilistica

L'inferenza probabilistica è il metodo attraverso il quale si utilizzano modelli probabilistici per dedurre informazioni sulle variabili di interesse o per effettuare previsioni, basandosi su dati osservati e probabilità condizionate.
Nella robotica probabilistica, le misurazioni dei sensori, i controlli e gli stati che il robot e l'ambiente possono assumere vengono trattati come variabili casuali. Queste vengono utilizzate per rappresentare caratteristiche del sistema robotico come letture sensoriali, posizione e orientamento, che consentono poi di gestire l'incertezza nelle azioni del robot. Per affrontare questo compito complesso, vengono impiegati strumenti matematici che consentono di aggiornare in modo continuo le stime dello stato del robot.

Variabili aleatorie

Le variabili aleatorie (o casuali) sono grandezze che possono assumere diversi valori in seguito a eventi casuali. Le variabili casuali possono essere di due tipi principali:

- **Discrete**: Le variabili possono assumere un insieme finito o numerabile di valori.
- **Continue**: Le variabili possono assumere un insieme infinito di valori all'interno di un intervallo.

Se per esempio indichiamo con X una variabile aleatoria, il valore specifico che esso può assumere è denotato con x. Un esempio classico di variabile casuale è il lancio di una moneta, dove X può assumere i valori "*testa*" o "*croce*". Se lo spazio di tutti i valori che X può assumere è discreto, come nel caso in cui X sia l'esito di un lancio di moneta, scriviamo :

$$p(X = x) \tag{2.1}$$

per indicare la probabilità che la variabile casuale X assume il valore x.
Ad esempio, una moneta equa è caratterizzata da:

$$p(X = testa) = p(X = croce) = \frac{1}{2} \tag{2.2}$$

Le variabili discrete godono della proprietà che la somma delle probabilità associate

ai valori possibili deve essere pari a 1. Questo significa che la probabilità totale di tutti i valori che una variabile discreta può assumere soddisfa la seguente condizione:

$$\sum_{x} p(X = x) = 1 \tag{2.3}$$

Nel caso di variabili casuali continue, come nella localizzazione robotica, si utilizzano funzioni di densità di probabilità. Una delle funzione di densità più comuni è quella della distribuzione normale unidimensionale con media μ e varianza σ^2. Questa distribuzione è data dalla seguente funzione gaussiana:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{2.4}$$

Le distribuzioni normali vengono spesso abbreviate come $\mathcal{N}(x; \mu, \sigma^2)$, che specifica la variabile casuale, la sua media e la sua varianza.
La distribuzione normale (2.4) assume che x sia una valore scalare. Spesso, come anche nel nostro elaborato, x è un vettore multidimensionale. Le distribuzioni normali su vettori vengono chiamate multivariate. Le distribuzioni normali multivariate sono caratterizzate da funzioni di densità della seguente forma :

$$p(x) = \frac{1}{\sqrt{det(2\pi\Sigma)}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)) \tag{2.5}$$

In questo caso μ rappresenta il vettore delle medie mentre Σ è una matrice simmetrica chiamata matrice di covarianza. Il valore T indica la trasposizione di un vettore. Analogamente a quanto avviene per le variabili discrete, anche per la distribuzione normale o distribuzione normale multivariata l'integrale della funzione di densità su tutto il dominio deve essere pari a 1:

$$\int p(x)dx = 1 \tag{2.6}$$

Inoltre, le probabilità, sia per le variabili discrete che per quelle continue, sono sempre non negative cioè $p(X = x) \geq 0$

Come ultimo argomento, introdurremo la **distribuzione congiunta**, concetto importante per l'introduzione alla teorema di Bayes. La distribuzione congiunta di due variabili casuali X e Y è espressa come :

$$p(x, y) = p(X = x \; e \; Y = y) \tag{2.7}$$

Questa formula rappresenta la probabilità che X assuma il valore x e Y il valore y. Se X e Y sono indipendenti :

$$p(x, y) = p(x) \cdot p(y) \tag{2.8}$$

Spesso, le variabili casuali portano informazioni l'una sull'altra. Se conosciamo già il valore di $Y = y$ e vogliamo determinare la probabilità che $X = x$, utilizziamo la **probabilità condizionata**:

$$p(x \mid y) = \frac{p(x, y)}{p(y)} \quad \text{con} \quad p(y) > 0 \tag{2.9}$$

Se X e Y sono indipendenti, la probabilità condizionata diventa $p(x)$. Un fatto rilevante è il **teorema della probabilità totale**, che afferma:

$$p(x) = \sum_{y} p(x \mid y)p(y) \quad \text{(caso discreto)} \tag{2.10}$$

$$p(x) = \int p(x \mid y)p(y)dy \quad \text{(caso continuo)} \tag{2.11}$$

Se $p(x \mid y)$ o $p(y)$ sono pari a zero, definiamo il prodotto $p(x \mid y) \cdot p(y)$ come zero, indipendentemente dal valore del fattore rimanente.

Come illustrato in precedenza, l'inferenza probabilistica è un metodo utile ad estrarre informazioni sulle variabili di interesse o per effettuare previsioni. Tra i vari approcci di inferenza probabilistica, l'inferenza bayesiana, basata sul teorema di Bayes, ha un ruolo fondamentale nella robotica probabilistica. Questa importanza è dovuta alla sua abilità di aggiornare in modo preciso le stime dello stato del robot utilizzando i dati osservati.

Teorema di Bayes:

Il Teorema di Bayes consente di determinare la probabilità di uno stato x dato uno stato y, collegando la probabilità condizionata $p(x \mid y)$ alla probabilità condizionata inversa $p(y \mid x)$, a condizione che $p(y) > 0$.

La formula è:

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)} = \frac{p(y \mid x)\, p(x)}{\sum_{x'} p(y \mid x')\, p(x')} \quad \text{(caso discreto)} \tag{2.12}$$

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)} = \frac{p(y \mid x)\, p(x)}{\int p(y \mid x')\, p(x')\, dx'} \quad \text{(caso continuo)} \tag{2.13}$$

dove :

- $p(x)$ è la distribuzione di probabilità a priori o probabilità marginale di x. 'A priori' significa che non tiene conto di nessuna informazione riguardo il dato y (ad esempio, una misura del sensore).
- $p(x \mid y)$ è la distribuzione di probabilità a posteriori di x, dato y. Indica come la nostra conoscenza su x viene aggiornata dopo aver incorporato il dato y.
- $p(y \mid x)$ è la distribuzione di probabilità condizionata di y, dato x. Specifica la probabilità di osservare i dati y assumendo che x sia noto.
- $p(y)$ è la distribuzione di probabilità a priori di y, e serve come costante di normalizzazione per assicurare che la somma delle probabilità a posteriori sia pari a 1.

Un'osservazione importante è che il denominatore $p(y)$ non dipende da x, quindi $p(y)^{-1}$ è lo stesso per ogni valore di x nella distribuzione a posteriori delle due equazioni sopra. Per questo motivo, $p(y)^{-1}$ è spesso scritto come una variabile di normalizzazione e genericamente denotato con η:

$$p(x \mid y) = \eta\, p(y \mid x)\, p(x). \tag{2.14}$$

Le regole discusse finora riguardanti l'indipendenza condizionale saranno utili per comprendere il concetto di **'belief'**, che introdurremo a breve. È possibile condizionare le regole discusse su altre variabili casuali, come ad esempio Z. Condizionare la regola di Bayes su $Z = z$ porta alla seguente formula:

$$p(x \mid y, z) = \frac{p(y \mid x, z)\, p(x \mid z)}{p(y \mid z)} \tag{2.15}$$

Analogamente, condizionare la regola per le probabilità di variabili indipendenti (2.8) su z produce:

$$p(x, y \mid z) = p(x \mid z)\, p(y \mid z) \tag{2.16}$$

Tale relazione è conosciuta come indipendenza condizionale. Essa si applica ogni volta che una variabile y non fornisce informazioni aggiuntive su una variabile x dato il valore di un'altra variabile z.

2.1.2 Belief

Per fornire una descrizione probabilistica della posa del robot, l'algoritmo presentato in questo elaborato assume che il robot abbia inizialmente a disposizione la mappa metrica dell'ambiente, ma che non conosca la propria posizione all'interno di essa.
Per descrivere in modo efficace la posa del robot all'interno dell'ambiente è importante introdurre il concetto di **Belief**.
La **Belief** (credenza) rappresenta la conoscenza interna del robot del proprio stato, ovvero la percezione di se stesso all'interno dell'ambiente di lavoro. Nei problemi reali di localizzazione, il robot spesso non può misurare direttamente la propria posizione, sopratutto quando essa è definita in un sistema di coordinate globali. A meno che il robot non disponga di un dispositivo GPS, non può conoscere con esattezza la propria posa iniziale e, in nei momenti successivi, può ottenere solo una stima probabilistica di dove si trova.
La robotica probabilistica rappresenta la **belief** attraverso distribuzioni di probabilità condizionate. Queste distribuzioni associano una probabilità a ciascuna possibile ipotesi riguardante lo stato reale del robot. Essa indica la probabilità che il robot si trovi in una certa posizione o stato, basandosi sulle informazioni disponibili.
Definendo:

- x_t**:** una variabile di stato al tempo t
- $z_{1:t}$**:** le misurazioni del sensore fino al tempo t
- $u_{1:t}$**:** le azioni di controllo eseguite dal robot fino al tempo t

La credenza su x_t, indicata con **bel(x_t)**, rappresenta la probabilità a **posteriori** dello stato x_t ed è espressa come:

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) \tag{2.17}$$

Viene chiamata a posteriori perché la belief è stata calcolata **dopo** aver incorporato la misurazione z_t.
Vedremo come nel contesto del filtro di Bayes, possiamo anche definire una probabilità a posteriori **prima** di incorporare la misurazione corrente di z_t, che viene calcolata subito dopo aver eseguito il controllo u_t.
Questa probabilità a posteriori viene definita come:

$$\overline{bel}(x_t) = p(x_t \mid z_{1:t-1}, u_{1:t}) \tag{2.18}$$

Questa distribuzione è comunemente denominata **previsione**, in quanto fornisce una stima della posa del robot al tempo t, basandosi sulle informazioni disponibili dopo l'esecuzione di un'azione o spostamento, ma prima dell'analisi delle letture dei sensori cioè prima della nuova misurazione z_t. Successivamente, l'integrazione della misurazione z_t permette il calcolo di $bel(x_t)$, un processo definito come correzione o aggiornamento della misurazione.

2.2 Approccio Bayesiano e filtro di Bayes

Una volta introdotta la teoria Bayesiana e il concetto di credenza (Belief), possiamo ora descrivere l'algoritmo implementato per la stima della posa del robot utilizzando il filtro Bayesiano. Esso rappresenta il modello teorico generale per la stima dello stato di un sistema dinamico.
Nel nostro elaborato, il particle filter fornisce una soluzione pratica al filtro di Bayes, utilizzando il metodo di Monte Carlo per approssimare la distribuzione di probabilità e gestire l'incertezza in spazi continui e complessi.
L'algoritmo calcola la distribuzione di probabilità, nota come Belief (bel), a partire dalle misure sensoriali e dalle azioni compiute dal robot. Essendo un filtro ricorsivo, esso aggiorna la credenza $bel(x_t)$ al tempo t utilizzando la credenza precedente $bel(x_{t-1})$, integrando le nuove informazioni provenienti dai sensori e dal modello di movimento.
Questo processo consente di mantenere una stima continua della posa del robot, adattandola in modo dinamico alle incertezze e alle variazioni dell'ambiente.
L'input del filtro è costituito dalla credenza $bel(x_{t-1})$ al tempo $t-1$, insieme al controllo più recente u_t e alle nuove osservazioni provenienti dai sensori z_t, mentre la l'output è la credenza aggiornata $bel(x_t)$ al tempo In seguito viene riportato lo schema generale del filtro Bayesiano in pseudocodice.

Algorithm Bayes_filter($bel(x_{t-1}), u_t, z_t$**):**

Algorithm Bayes_filter($bel(x_{t-1}), u_t, z_t$**):**

for all x_t do

$$\begin{aligned} \overline{bel}(x_t) &= \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, dx \\ bel(x_t) &= \eta\, p(z_t \mid x_t)\, \overline{bel}(x_t) \end{aligned} \tag{2.19}$$

endfor

return $bel(x_t)$

L'algoritmo del filtro Bayesiano possiede due punti chiave: predizione (o aggiornamento del controllo) e aggiornamento delle misure.

1. Nella prima, il filtro di Bayes elabora il controllo u_t. La credenza predetta $\overline{bel}(x_t)$ si calcola integrando il prodotto della credenza precedente $bel(x_{t-1})$ e la probabilità che u_t induca una transizione da x_{t-1} a x_t.

2. Nella seconda, l'algoritmo aggiorna la credenza predetta $\overline{bel}(x_t)$ moltiplicandola per la probabilità della nuova osservazione $p(z_t \mid x_t)$.
   Il prodotto risultante può non essere una probabilità valida, in quanto potrebbe non integrare a 1. Per correggere questo, il risultato viene normalizzato utilizzando una costante η, ottenendo cosi la credenza finale $bel(x_t)$.

Come osservazione finale dell'algoritmo, per calcolare la credenza posteriore in modo ricorsivo è essenziale avere una credenza iniziale $bel(x_0)$ al tempo $t = 0$. Se il valore di x_0 è noto con certezza, $bel(x_0)$ è inizializzata con una distribuzione a massa puntiforme, che concentra tutta la massa di probabilità sul valore esatto di x_0, assegnando probabilità zero agli altri valori. Se il valore non è noto, si usa una distribuzione uniforme. Se si dispone di una conoscenza parziale del valore iniziale, si può utilizzare una distribuzione che non è uniforme e che riflette tale incertezza.
In poche parole, l'algoritmo del filtro di Bayes è un algoritmo ricorsivo per calcolare la credenza posteriore, e per aggiornare la credenza in modo efficace è necessario avere una credenza iniziale $bel(x_0)$ al tempo $t = 0$.
Per limitare la necessità di considerare la storia completa del sistema, l'algoritmo del filtro di Bayes è stato costruito sulla base dell'assunzione di Markov, che rappresenta la condizione di 'assenza di memoria'. Questa assunzione implica che lo stato attuale rappresenta un riassunto completo del passato, semplificando così la rappresentazione della storia del robot.
Tuttavia, nella pratica, l'assunzione di Markov non è sempre applicabile dato che esistono dinamiche ambientali ed imprecisioni nei modelli probabilistici che possono violare la stessa. Nonostante queste possibili violazioni, i filtri Bayesiani si sono dimostrati generalmente efficaci. Per questo motivo, è essenziale definire con precisione lo stato x_t per ridurre al minimo l'influenza dei fattori non considerati nel modello di localizzazione.

L'algoritmo del filtro di Bayes, sebbene teoricamente valido, è spesso impraticabile in scenari reali a causa dell'elevata complessità computazionale. Calcolare e gestire la distribuzione completa della probabilità diventa oneroso in spazi di stato ad alta dimensionalità, il che rende indispensabile l'uso di metodi approssimativi per

applicazioni pratiche. Per rispondere a queste esigenze, sono emerse diverse varianti del filtro di Bayes, suddivisibili in due categorie principali: filtri gaussiani e filtri non parametrici.
Questi approcci offrono soluzioni alternative per bilanciare la complessità computazionale e l'accuratezza delle stime in contesti reali.

Elenchiamo alcuni dei filtri di maggior rilevanza, evidenziandone le caratteristiche principali:

- Filtri Gaussiani come, il filtro di Kalman o Filtri di Kalman esteso:
  - Efficienza computazionale: Il filtro di Kalman è progettato per modelli lineari, mentre il filtro di Kalman esteso è progettato per modelli non lineari. Entrambi sono rapidi nei tempi di calcolo, anche se il filtro di Kalman esteso richiede più risorse per la linearizzazione.
  - Accuratezza dell'approssimazione: Il filtro di Kalman è altamente preciso per distribuzioni unimodali e modelli gaussiani, ma limitato per distribuzioni multimodali, mentre il filtro di Kalman Esteso può avere una ridotta accuratezza in presenza di forti non linearità.
  - Facilità di implementazione: Il filtro di Kalman è relativamente facile da implementare grazie alla sua struttura lineare, mentre il filtro di Kalman esteso è moderatamente complesso da implementare richiedendo una particolare attenzione nella linearizzazione e nella gestione degli errori.
- Filtri non parametrici, come il particle filter:
  - Efficienza computazionale: Richiede un tempo computazionale maggiore, specialmente con un alto numero di particelle, ma offre flessibilità nella rappresentazione delle distribuzioni.
  - Accuratezza dell'approssimazione: In grado di rappresentare distribuzioni complesse e multimodali con alta precisione, sebbene la precisione dipenda fortemente dal numero di particelle utilizzate.
  - Facilità di implementazione: Relativamente facile da implementare per modelli non lineari, ma la scelta e l'ottimizzazione del numero di particelle possono influenzare significativamente le prestazioni.

Questi filtri rappresentano soluzioni diverse per le sfide di modellazione e calcolo nella robotica probabilistica, ognuno con i propri vantaggi e compromessi.

2.3 Particle filter con metodo Monte Carlo

Nel paragrafo precedente, abbiamo introdotto l'approccio bayesiano come una struttura rigorosa per affrontare problemi di stima dello stato nei sistemi dinamici, basato sulla costruzione di una funzione di densità di probabilità che integra tutte le informazioni disponibili.
Per i sistemi non lineari e non gaussiani, come presente nel nostro elaborato, l'assenza di soluzioni esatte richiede l'uso di metodi approssimativi, necessari per una gestione efficace dell'incertezza, soprattutto nelle applicazioni reali. Per affrontare questi problemi nel mondo della robotica viene utilizzato il particle filter, una variante non parametrica del filtro di Bayes.
L'idea centrale del particle filter è che approssima la distribuzione a posteriori $bel(x_t)$ attraverso un insieme di campioni estratti da una densità di probabilità, utilizzando il *campionamento per importanza*. Questo campionamento assegna un peso a ciascuna particella in base alla sua rilevanza rispetto alle osservazioni sensoriali. L'approccio in cui un insieme di campioni e i loro pesi vengono utilizzati per approssimare la distribuzione a posteriori, concentrando le particelle nelle regioni di maggiore probabilità, rappresenta il metodo di Monte Carlo.
Grazie al campionamento per importanza, il particle filter sfrutta simulazioni ripetute per ottenere una stima accurata dello stato.
In seguito, introduciamo alcune notazioni prima di presentare il codice. Nei particle filter, i campioni di una distribuzione a posteriori sono chiamati *particelle* e sono come:

$$\mathcal{X} := \{x^{[1]}, x^{[2]}, \dots, x^{[M]}\} \tag{2.20}$$

Ogni particella $x^{[m]}$ (con $1 \leq m \leq M$) è una concreta rappresentazione dello stato al tempo t, cioè un'ipotesi su quale possa essere il vero stato del mondo al tempo t. Analogamente a come avviene per il robot, ogni particella ha una propria posa definita da coordinate (x, y) e un angolo θ, che descrivono posizione e orientamento rispetto al sistema di riferimento.
Il numero di particelle nel set $\mathcal{X}_t$, indicato con M, è generalmente elevato, ma può variare in funzione del tempo t o altre grandezze legate alla credenza bel(x_t).
I filtri a particelle approssimano la credenza bel(x_t) tramite il set di particelle $\mathcal{X}_t$, dove la probabilità che uno stato ipotizzato x_t venga incluso nel set è proporzionale alla sua distribuzione a posteriori del filtro di Bayes bel(x_t):

$$x_t^{[m]} \sim p(x_t \mid z_{1:t}, u_{1:t}) \tag{2.21}$$

Presentiamo in seguito l'algoritmo del particle filter in pseudocodice:

Algorithm Particle_filter($\mathcal{X}_{t-1}, u_t, z_t$):

$$
\begin{aligned}
&\overline{\mathcal{X}}_t = \mathcal{X}_t = \emptyset \\
&\text{for } m = 1 \text{ to } M \text{ do} \\
&\qquad \text{sample } x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]}) \\
&\qquad w_t^{[m]} = p(z_t \mid x_t^{[m]}) \\
&\qquad \overline{\mathcal{X}}_t = \overline{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle \\
&\text{endfor} \\
&\text{for } m = 1 \text{ to } M \text{ do} \\
&\qquad \text{draw } i \text{ with probability } \propto w_t^{[i]} \\
&\qquad \text{add } x_t^{[i]} \text{ to } \mathcal{X}_t \\
&\text{endfor} \\
&\text{return } \mathcal{X}_t
\end{aligned}
\tag{2.22}
$$

Si osserva che l'algoritmo del particle filter aggiorna ricorsivamente $\mathcal{X}_t$ a partire da $\mathcal{X}_{t-1}$ e che una maggiore densità di campioni in una regione dello spazio degli stati aumenta la probabilità che il vero stato si trovi in quella regione.

Le fasi principali del particle filter:

Dall'algoritmo presentato, osserviamo che il particle filter esegue una sequenza di operazioni ripetute a ogni intervallo di tempo, che comprende tre fasi principali:

- predizione
- aggiornamento
- ricampionamento chiamato anche ricampionamento per importanza

Questi passaggi vengono eseguiti in modo continuo per stimare con precisione lo stato del sistema. La fase di ricampionamento, nel presente elaborato, è stata implementata tramite il metodo di ricampionamento a bassa varianza, che si distingue dal metodo di ricampionamento per importanza. Questo metodo consente di selezionare in modo più efficiente le particelle con pesi maggiori, ottenendo una distribuzione più accurata e bilanciata delle particelle nel sistema rispetto al metodo proposto nel particle filter.

1. Predizione

   Nella prima fase, l'algoritmo predice lo stato del sistema per ciascuna particella utilizzando il modello dinamico e i comandi di movimento. Partendo dallo stato della particella al tempo $t-1$ e dal comando di controllo u_t, viene generato un nuovo stato per ogni particella:

   $$x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]}) \tag{2.23}$$

   Il campione (sample) risultante è numerato con m, indicando che è generato a partire dalla m-esima particella dell'insieme $\mathcal{X}_{t-1}$.
   L'insieme di particelle ottenuto dall'iterazione di questo passo M volte rappresenta $\overline{bel}(x_t)$. La predizione si basa sulla transizione dello stato $p(x_t \mid u_t, x_{t-1}^{[m]})$ che descrive come il sistema evolve nel tempo in risposta ai comandi di movimento. Come si può notare dal codice, in questa fase si considerano solo i comandi di movimento e lo stato precedente, escludendo le osservazioni sensoriali.

2. Aggiornamento

   Nella seconda fase, l'algoritmo aggiorna lo stato del sistema basandosi sulle osservazioni sensoriali. Il campionamento per importanza viene utilizzato per assegnare i pesi alle particelle in base alla loro coerenza con i dati raccolti. Per ogni particella $x_t^{[m]}$, viene calcolato il cosiddetto fattore di importanza, denotato da $w_t^{[m]}$. Questo fattore rappresenta la probabilità di osservare i dati z_t dato lo stato della particella, cioè $w_t^{[m]} = p(z_t \mid x_t^{[m]})$.
   Le particelle più compatibili con le osservazioni ricevono pesi maggiori, mentre quelle meno compatibili hanno pesi minori. Se interpretiamo $w_t^{[m]}$ come il peso di una particella, l'insieme di particelle pesate fornice una rappresentazione approssimativa del belief del filtro di Bayes $bel(x_t)$.
   Successivamente, fuori dalla fase di aggiornamento, $\overline{\mathcal{X}}_t$ viene aggiornato temporaneamente con le particelle e i pesi associati. Questo set $\overline{\mathcal{X}}_t$ viene utilizzato per memorizzare le particelle campionate $x_t^{[m]}$ e i loro pesi $w_t^{[m]}$, calcolati nella fase di aggiornamento.

3. Ricampionamento

   Nell'ultima fase, l'algoritmo esegue il ricampionamento per evitare la degenerazione del set di particelle, ovvero la situazione in cui la maggior parte delle particelle ha pesi molto bassi e contribuisce poco all'approssimazione

della distribuzione posteriore. In questa fase, le particelle con pesi bassi vengono scartate, mentre quelle con pesi maggiori vengono selezionate più frequentemente. In definitiva, il ricampionamento per importanza assicura che le particelle selezionate rappresentino meglio la distribuzione posteriore.
Durante il ricampionamento, l'algoritmo genera un nuovo insieme di particelle $\mathcal{X}_t$ campionando dall'insieme temporaneo $\overline{\mathcal{X}}_t$ proporzionalmente ai pesi $w_t^{[m]}$.
Incorporando i pesi di importanza durante il ricampionamento, l'algoritmo seleziona le particelle che meglio rappresentano la distribuzione posteriore. Prima del ricampionamento, le particelle erano distribuite secondo $\overline{bel}(x_t)$, mentre dopo il ricampionamento rappresentano approssimativamente la belief come $bel(x_t) = \eta \cdot p(z_t|x_t^{[m]}) \cdot \overline{bel}(x_t^{[m]})$.
Dopo il ricampionamento, i pesi delle particelle vengono reimpostati e l'algoritmo riprende il ciclo di predizione e aggiornamento a ogni iterazione successiva.
Questo meccanismo permette di ottimizzare l'uso delle risorse computazionali, concentrandole nelle aree più rilevanti dello spazio degli stati, riducendo cosi la necessità di un numero eccessivo di particelle per approssimare correttamente la distribuzione posteriore.

In sintesi, il funzionamento del particle filter si basa sull'idea di mantenere, a ogni istante, un insieme di possibili stati del sistema, ciascuno leggermente diverso dagli altri. Quando una nuova misurazione dei sensori diventa disponibile, ogni stato candidato viene valutato in base alla sua coerenza con i dati osservati. Gli stati con valutazioni migliori vengono selezionati, duplicati e leggermente modificati, generando così una nuova serie di stati che rappresentano più accuratamente la realtà del sistema.

Principali fonti di errore nel particle filter:

Il particle filter, sebbene sia uno strumento efficace nella stima degli stati, presenta alcune limitazioni essendo un metodo di approssimazione. Di seguito vengono descritte alcune delle principali fonti di errore che possono compromettere le prestazioni dell'algoritmo:

- **Numero limitato di particelle:** Utilizzando poche particelle si introduce un errore sistematico (bias) nel calcolo della distribuzione posteriore, ovvero una tendenza costante a produrre stime imprecise. Per esempio, se viene utilizzata una sola particella cioè M = 1, il ciclo for verrà eseguito una sola volta e $\overline{\mathcal{X}}_t$ conterrà una sola particella, campionata dal modello di movimento. Il ricampionamento accetterà questa particella indipendentemente dal suo fattore di importanza $w_t^{[m]}$, ignorando quindi le misurazioni z_t.

Con M = 1, durante la fase di ricampionamento, si verifica la normalizzazione implicita, cioè il peso $w_t^{[m]}$ diventa il proprio normalizzatore, e quindi $p(\text{estrarre } x^{[m]}) = 1$. Questo effetto diventa meno rilevante man mano che aumenta il numero di particelle.

- **Casualità del ricampionamento:** Un'ulteriore fonte di errore è legata alla casualità nella fase di ricampionamento. Ad esempio, quando il robot rimane fermo, la diversità delle particelle può ridursi, portando a stime imprecise. Per affrontare questo problema, si possono adottare due strategie:
  - ridurre la frequenza del ricampionamento quando il robot è fermo
  - utilizzare il ricampionamento a bassa varianza per mantenere la diversità delle particelle e migliorare l'accuratezza delle stime.
- **Perdita di particelle a causa di spazi complessi:** Nel particle filter, la perdita di particelle può verificarsi quando il numero di particelle M è insufficiente rispetto alla varietà di stati possibili in una mappa molto grande o complessa. Per limitare questo problema, si possono aggiungere particelle casuali dopo ogni ricampionamento, anche se ciò può ridurre l'accuratezza della stima posteriore.

Capitolo 3

Localizzazione Monte Carlo e modelli di movimento e sensoriali

3.1 Introduzione all'algoritmo di localizzazione Monte Carlo

Nel capitolo precedente sono stati approfonditi i concetti dell'inferenza, della belief come rappresentazione probabilistica dello stato del robot e del teorema di Bayes. L'introduzione di questi concetti ci ha permesso di comprendere meglio che la posizione del robot non viene descritta tramite un singolo punto, ma attraverso una distribuzione di probabilità.
Questo concetto rappresenta la localizzazione probabilistica, dalla quale si è sviluppata la localizzazione ricorsiva, che rappresenta un processo continuo di aggiornamento della stima probabilistica della posizione del robot attraverso algoritmi ricorsivi, come il filtro di Bayes o il particle filter.
Successivamente, l'attenzione si è focalizzata in particolare sul particle filter basato sul metodo di Monte Carlo, un algoritmo che non richiede di definire una forma precisa per la distribuzione della belief, ma utilizza un insieme di campioni (o particelle) per approssimarla, fornendo cosi una stima probabilistica della posizione del robot anche in presenza di incertezze e rumore.

Introduciamo in questo capitolo l'algoritmo di localizzazione Monte Carlo (MCL), una variante del particle filter applicata alla localizzazione robotica. L'algoritmo MCL integra l'uso della **mappa** dell'ambiente come vincolo aggiuntivo, migliorando l'accuratezza nella stima della posizione del robot. La differenza principale consiste nel modo in cui le fasi di predizione e correzione vengono implementate. Sebbene entrambi gli algoritmi utilizzino queste due fasi, nel MCL esse vengono potenziate dall'integrazione della mappa dell'ambiente.

Nella fase di **predizione**, nel caso dell'algoritmo MCL, la mappa dell'ambiente introduce un vincolo aggiuntivo che limita le posizioni delle particelle alle aree consentite, escludendo quelle che si troverebbero in zone non valide, come ostacoli o al di fuori dei confini della mappa.
Nella fase di **correzione**, mentre nel particle filter con metodo di Monte Carlo il processo si basa unicamente sulla coerenza con i dati sensoriali, nel caso di MCL i pesi delle particelle sono influenzati anche dalla mappa. In questo modo, vengono 'privilegiate' le particelle che non solo sono coerenti con le osservazioni sensoriali, ma che si trovano anche in posizioni valide all'interno dell'ambiente mappato.

In seguito, l'algoritmo di localizzazione Monte Carlo in pseudocodice:

Algorithm MCL($\mathcal{X}_{t-1}, u_t, z_t, m$):

$$
\begin{aligned}
&\overline{\mathcal{X}}_t = \mathcal{X}_t = \emptyset \\
&\text{for } m = 1 \text{ to } M \text{ do} \\
&\qquad x_t^{[m]} = \textbf{sample\_motion\_model}(u_t, x_{t-1}^{[m]}) \\
&\qquad w_t^{[m]} = \textbf{measurement\_model}(z_t, x_t^{[m]}, m) \\
&\qquad \overline{\mathcal{X}}_t = \overline{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle \\
&\text{endfor} \\
&\text{for } m = 1 \text{ to } M \text{ do} \\
&\qquad \text{draw } i \text{ with probability } \propto w_t^{[i]} \\
&\qquad \text{add } x_t^{[i]} \text{ to } \mathcal{X}_t \\
&\text{endfor} \\
&\text{return } \mathcal{X}_t
\end{aligned}
\tag{3.1}
$$

L'algoritmo inizia creando una nuova rappresentazione dello stato $\overline{\mathcal{X}}_t$, inizialmente vuota, come avviene nel particle filter con il metodo di Monte Carlo. Questo insieme accoglierà le particelle aggiornate che rappresentano le possibili posizioni del robot. Come menzionato in precedenza, la differenza tra il particle filter e l'algoritmo MCL si trova nella fase di predizione, dove l'algoritmo MCL utilizza un modello di movimento campione del robot, presente nell'algoritmo come **sample_motion_model**, e nella fase di correzione, che si basa su un modello di misurazione che integra le osservazioni sensoriali e le informazioni della mappa, presente nell'algoritmo come **measurement_model**.
Passiamo ora all'analisi dei due modelli, la cui implementazione sarà presentata nel capitolo successivo.

3.2 Modello di movimento

Il modello di movimento o motion model descrive come un robot passa da uno stato x_{t-1} a uno stato x_t in risposta a un comando u_t tenendo conto dell'incertezza associata al movimento. Come nel caso delle particelle, in un contesto probabilistico, il modello di movimento viene rappresentato dalla probabilità $p(x_t \mid u_t, x_{t-1})$ che descrive dove potrebbe trovarsi il robot in futuro in base ai comandi ricevuti e alla posizione precedente. La probabilità viene utilizzata nella fase di previsione del filtro di Bayes, consentendo di aggiornare la stima dello stato del robot tenendo conto dell'incertezza e della variabilità nei movimenti. Esistono diversi modelli di movimento che descrivono come un robot passa da uno stato all'altro in base alle variabili considerate. Di seguito presentiamo alcuni esempi, come il modello di movimento basato sulla velocità o motion model velocity, che si basa sulla velocità del robot. In questo modello, il robot viene controllato tramite due velocità: una traslazionale (v_t) e una rotazionale (ω_t). Un altro possibile modello di movimento è il modello di movimento basato sull'odometria o motion model odometry, che si basa sulle misurazioni odometriche del robot. In questo modello, il movimento viene stimato attraverso una sequenza di tre passaggi: una rotazione iniziale (δ_{rot1}), una traslazione (δ_{trans}) e una seconda rotazione finale (δ_{rot2}). Queste grandezze sono calcolate utilizzando le differenze relative tra le pose odometriche del robot nei momenti $t-1$ e t.

In questo elaborato è stata scelta l'implementazione di un modello di movimento semplice con rumore aggiunto all'interno del metodo **updateParticlePos()**, che aggiorna ogni particella con le variazioni odometriche e il rumore. Questo modello aggiorna la posizione delle particelle basandosi sui dati di odometria del robot, tenendo conto dell'incertezza attraverso l'aggiunta di rumore casuale.

Il modello applica i cambiamenti di posizione $(\Delta x, \Delta y)$ e orientamento $(\Delta\theta)$ alla posizione corrente di ogni particella, per simulare la variazione della posizione del robot in base alla sua odometria.

Per simulare l'incertezza, è stato aggiunto rumore casuale utilizzando distribuzioni gaussiane. Questo rumore è stato applicato alle coordinate x e y per la **traslazione** e all'angolo θ per la **rotazione**. Il rumore è stato generato con una distribuzione gaussiana centrata su 0, con deviazioni standard specifiche per la traslazione (**std_dev_t**) e per la rotazione (**std_dev_r**).

Considerando nel contesto probabilistico il modello di movimento descritto soltanto dalla $p(x_t \mid u_t, x_{t-1})$ allora teniamo conto del movimento del robot nel suo sistema locale.

Interazione del modello di movimento con la mappa:

Nel contesto probabilistico descritto sopra, il modello di movimento si basa esclusivamente sulla probabilità di transizione $p(x_t \mid u_t, x_{t-1})$, considerando il movimento del robot senza informazioni specifiche sull'ambiente. Nell'algoritmo descritto nell'elaborato, viene fornita in input una mappa m. Di conseguenza, è necessario adattare il modello di movimento per incorporare le informazioni ambientali, in modo da permettere al robot di navigare efficacemente, tenendo conto delle aree percorribili e degli ostacoli presenti.
Con l'integrazione della mappa, la probabilità di transizione diventa:

$$p(x_t \mid u_t, x_{t-1}, m) = \eta \cdot p(x_t \mid u_t, x_{t-1}) \cdot p(x_t \mid m) \tag{3.2}$$

3.3 Modello di misurazione sensoriale

La parte sensoristica occupa un ruolo fondamentale nella soluzione di tutte le problematiche legate alla localizzazione del robot, queste tecnologie sono discusse in maniera più approfondita nel capitolo Sensors for Mobile Robots del libro "Encyclopedia of Robotics" [1] di cui diamo una breve introduzione.

I sensori possono essere classificati in due grandi famiglie in base al **tipo di informazione** che raccoglie:

- **Sensori propriocettivi:** Questi sensori forniscono informazioni sulla configurazione interna del robot indipendentemente dall'ambiente esterno. Essi rilevano grandezze come la posizione delle ruote, la velocità e l'orientamento del robot stesso. Esempi di sensori : encoder (per misurare la rotazione delle ruote), giroscopi (per rilevare l'orientamento del robot), accelerometri (per misurare le accelerazioni e le variazioni di velocità).

- **Sensori eterocettivi:** Questi sensori misurano grandezze esterne al robot, come la distanza dagli ostacoli o la forma degli oggetti presenti nell'ambiente. Esempi di questi sensori : LiDAR (laser scanner), telecamere (RGB e di profondità), sensori a ultrasuoni (sonar), radar.

I sensori possono essere classificati in due grandi famiglie in base al **modo in cui** il sensore raccoglie le informazioni:

- **Sensori attivi:** Questi sensori raccolgono informazioni emettendo energia sotto forma di luce, onde sonore oppure onde elettromagnetiche, rilevando la risposta o il riflesso di tale energia. Esempi di sensori: LiDAR, sonar e radar.

- **Sensori passivi:** Questi sensori invece raccolgono informazioni rilevando l'energia già presente nell'ambiente, come la luce o il calore. Sono chiamati passivi proprio perché non emettono energia, ma raccolgono quella disponibile. Esempi di sensori: Telecamere RGB e a infrarossi.

Tra tutti i sensori menzionati, ci concentriamo in particolare sul LiDAR, sensore utilizzato nell'algoritmo di questo elaborato. I sensori LiDAR (Light Detection and Ranging) sono dei sensori eterocettivi attivi che utilizzano laser per misurare la distanza dagli oggetti circostanti.
Esistono tre principali tipi di LiDAR:

- **LiDAR a onda continua (Continuous Wave)**: Sensore che misura la distanza confrontando la differenza di fase tra l'onda continua emessa dal sensore e quella riflessa da un oggetto incontrato lungo la linea di vista del sensore.
  Esempio: Il sensore SICK LMS 100.

- **LiDAR a impulsi (Pulse-Based)**: Sensore che misura la distanza calcolando il tempo che un impulso di luce impiega per raggiungere un oggetto e tornare indietro al sensore.
  Esempio: Velodyne Alpha Puck LiDAR.

- **LiDAR a stato solido (Solid-State)**: Sensore che non usa parti meccaniche per deviare i fasci laser, ma impiega un laser pulsato che alimenta un array di fase ottico composto da antenne ottiche ravvicinate. Il controllo della fase di ogni antenna dirige il fascio nella direzione desiderata.
  Esempio: Quanergy S3-2.

Concentrando l'attenzione sul modello di misurazione, possiamo analizzare la fase di **correzione** dell'algoritmo MCL. Il modello di misurazione o measurement model descrive come i sensori del robot percepiscono l'ambiente circostante e come queste misurazioni vengono utilizzate per aggiornare la stima della posizione del robot.

Nel contesto della robotica probabilistica, il modello di misurazione descrive la probabilità di ottenere una misurazione sensoriale z_t, data la posa del robot x_t e la mappa m. Per i sensori di distanza come il LIDAR, questa relazione è formalmente rappresentata come una distribuzione condizionale:

$$p(z_t, x_t, m) \tag{3.3}$$

A differenza del particle filter, nel caso della localizzazione Monte Carlo, i pesi delle particelle sono influenzati non solo dai dati sensoriali, ma anche dalla mappa. Ogni particella rappresenta una possibile ipotesi sulla posizione del robot. La validità di ciascuna particella viene valutata attraverso un processo di **ray casting**, che simula il comportamento del sensore LiDAR.

Per una migliore comprensione, introduciamo in seguito il processo di **correzione** tramite ray casting:

- **Ray casting delle particelle**: Per ciascuna particella, il sistema esegue il ray casting, proiettando raggi dalla posizione ipotetica della particella, simulando cosi le misurazione che il robot effettuerebbe se si trovasse in quella posizione.
- **Scansione reale del robot**: In parallelo, il robot esegue una scansione LiDAR nell'ambiente reale, ottenendo in questo modo le **misurazioni reali** delle distanze dagli ostacoli.
- **Confronto delle misurazioni**: Le misurazioni simulate per ciascuna particella vengono confrontate con quelle reali ottenute dal robot. Se le due misurazioni sono ben allineate, significa che la particella rappresenta una posizione probabile del robot. Maggiore è l'allineamento tra le misurazioni, maggiore sarà l'incremento del peso della particella.
- **Correzione della distribuzione delle particelle**: Le particelle che presentano misurazioni simulate distanti da quelle reali ricevono un peso minore e vengono scartate durante il processo di ricampionamento. Al contrario, le particelle con pesi maggiori, caratterizzate da misurazioni coerenti con quelle reali, vengono conservate e ricampiate per affinare ulteriormente la stima della posizione del robot.

Nell'elaborato è stato scelto di utilizzare un modello matematico di misurazione sensoriale semplificato rispetto alla implementazione più complessa introdotta nel libro "Probabilistic Robotics" [2], di cui diamo visione degli ultimi due punti precedenti con le seguenti equazioni.

Fase di confronto delle misurazioni:
L'equazione scelta per definire la probabilità della posizione di una particella, tramite una Gaussiana centrata sulla distanza simulata del beam LiDAR della particella,

è la seguente:

$$p(z_t^k \mid x_t, m) = \max\left(\exp\left(-\frac{(z_t^k - \hat{z}_t^k)^2}{2\sigma^2}\right), \text{min\_prob}\right) \tag{3.4}$$

Dove:

- z_t^k: è la lettura reale del LiDAR per il beam k
- $\hat{z}_t^k$: è la distanza simulata del LiDAR in base alla posizione (x, y, θ) della particella
- σ: è la deviazione standard della traslazione della particella
- `min_prob`: è la costante a cui verrà verrà saturata la Gaussiana

Fase di correzione della distribuzione delle particelle:
In questa fase il peso totale della particella viene calcolato come la somma delle probabilità normalizzate sul numero totale di misurazioni.

$$w(x_t) = \frac{1}{K}\sum_{k=1}^{K} p(z_t^k \mid x_t, m) \tag{3.5}$$

Dove K è il numero totale delle particelle.

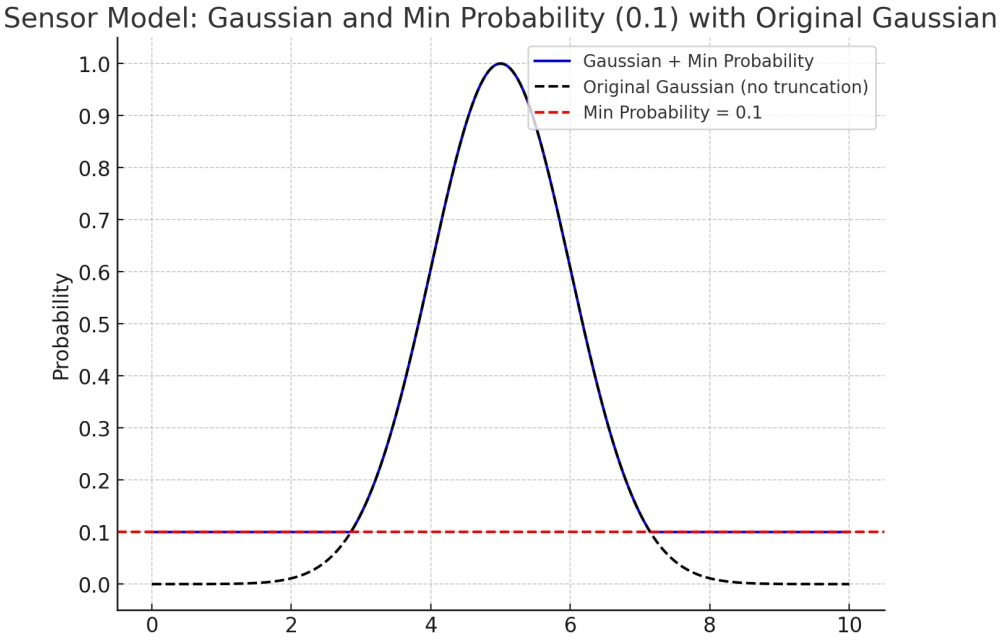

Figura 3.1: Gaussiana normalizzata a 1 e saturata a min_prob

I dettagli tecnici di questo processo verranno discussi al capitolo successivo dove si andrà a vedere punto per punto l'implementazione e le tecnologie usate.

Capitolo 4

Implementazione e risultati sperimentali

4.1 Tecnologie e scelte progettuali

Prima di descrivere l'implementazione effettiva dell'algoritmo MCL bisogna fare alcune premesse riguardanti le tecnologie utilizzate e le scelte progettuali che sono state fatte.
L'input iniziale della implementazione si basa su letture reali fornite da esperimenti che sono stati condotti in vari laboratori tra i quali citiamo Freiburg del quale verranno mostrati i risultati dell'implementazione.
Queste letture sono state realizzate utilizzando un robot mobile dotato di LiDAR a 180 gradi modello SICK, creando così dei dataset indoor che sono stati salvati e verranno utilizzati come base di partenza per l'algoritmo.

Figura 4.1: Modello del robot utilizzato per le sperimentazioni

I risultati di queste letture sono inseriti in input al programma tramite file con estensione .clf (CARMEN Logfile) formattato nel seguente modo:

```
# CARMEN Logfile
# file format is one message per line
# message_name [message contents] ipc_timestamp ipc_hostname
    logger_timestamp
# message formats defined: PARAM SYNC ODOM FLASER RLASER
# PARAM param_name param_value
# SYNC tagname
# ODOM x y theta tv rv accel
# FLASER num_readings [range_readings] x y theta odom_x odom_y
    odom_theta
# RLASER num_readings [range_readings] x y theta odom_x odom_y
    odom_theta
# TRUEPOS true_x true_y true_theta odom_x odom_y odom_theta
# robot: stayton
```

Per ogni esperimento in questi laboratori esistono 2 file, quello che contine le letture fatte dal robot ed il file "GT" che invece contiene le posizioni "Ground Truth" ovvero le posizioni esatte del robot. Questi due file contengono molte informazioni e parametri circa l'esperimento effettuato, ma i dati che interessano l'implementazione sono le letture di odometria e di LiDAR che seguono uno specifico formato di cui al seguito diamo un esempio.

```
ODOM -3.034287 8.291214 -3.120965 0.000000 0.000000 0.000000
    1211.530144 magnum 0.016900
FLASER 360 1.65 ... 1.00 -2.994779 8.291967 -3.122499 -3.034772
    8.291204 -3.122499 1211.720330 magnum 0.227623
```

Come tecnologia principale per l'implementazione è stato scelto di utilizzare il C++ perché, in quanto linguaggio compilato, si presta bene ad essere usato per calcoli di grandi dimensione, inoltre ha una ottima implementazione della libreria di OpenCV, tecnologia di computer vision che è stata usata per la visualizzazione grafica dell'algoritmo e per testare le varie implementazioni dello stesso.
Dopo questa introduzione iniziale elenchiamo le strutture dati utilizzate per interpretare i log dei file clf ed utilizzarli nel codice.

```
struct Data
{
  virtual ~Data() = default;
  double timestamp;
};
```

Listing 4.1: Struttura base del Data

```
struct OdomData : public Data
{
  double x, y, theta;
  double tv, rv, accel;
};
```

Listing 4.2: Struttura della odometria

Per maggiori informazioni su OpenCV: `https://opencv.org`

```
struct LidarData : public Data
{
  int num_readings;
  std::vector<double>
   range_readings;
  double x, y, theta;
  double odom_x, odom_y,
   odom_theta;
};
```

Listing 4.3: Struttura del lidar

Queste prime tre strutture sono fondamentali per il progetto dato che è stato creato un parser, di cui tralasceremo l'implementazione per semplicità, utilizzato per interpretare i dati del .clf e mappare le linee contenute al suo interno in queste strutture permettendo così di lavorare con dati ben strutturati. Un'altra classe fondamentale di cui non discuteremo l'implementazione è la classe Map, il cui scopo è quello di creare e ed eseguire operazioni su una mappa virtuale utilizzando la libreria di OpenCV, permettendoci così di visualizzare passo per passo quello che è il comportamento dell'algoritmo di localizzazione con il metodo di Monte Carlo.

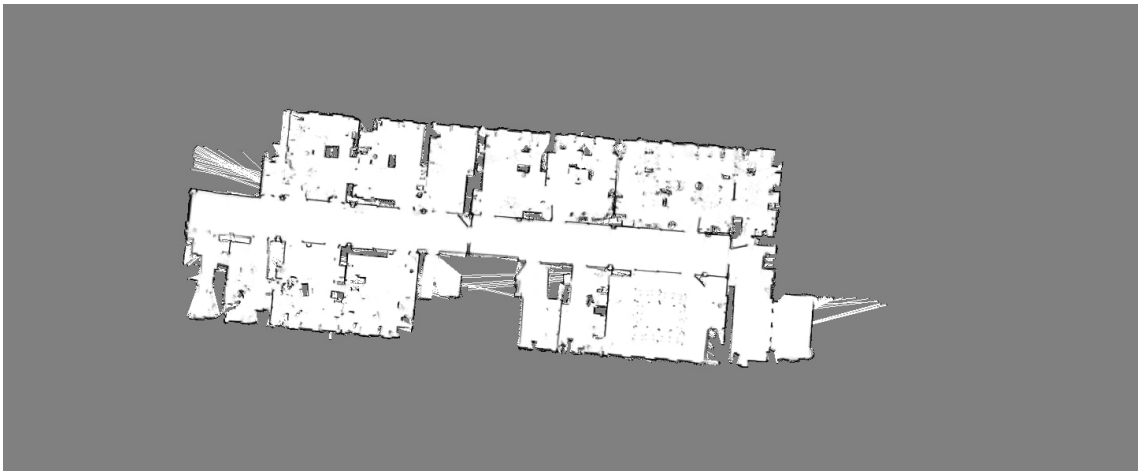

Figura 4.2: Mappa di Freiburg generata dalle letture gt

L'insieme di questi elementi abilita quindi ad implementare e fare simulazioni dell'algoritmo in modo visivo ed interpretabile.

4.2 Implementazione del modello di misurazione sensoriale

Riprendendo la definizione data di modello sensoriale del capitolo precedente, nella reale implementazione dell'elaborato, la simulazione del LiDAR si basa su un sensore a onda continua come il SICK LMS 100. Il codice riportato nella seguente pagina, facente parte della classe Map, simula una **scansione LiDAR** per una data posizione x, y e orientamento θ su una mappa. Esso viene usato per il processo di ray casting delle particelle. Il codice simula una proiezione di raggi dalla posizione ipotetica di ogni particella. Il processo di simulazione si basa su tre elementi chiave:

- **LidarData:** La struttura che memorizza le informazioni relative alla simulazione, comprese la posizione x, y e θ delle particelle, oltre alle distanze rilevate dagli ostacoli.

- **cell_size_:** Definisce di quanto i raggi avanzano ad ogni passo nella simulazione, con incrementi pari a $cell\_size\_$. Questo simula il movimento dei raggi attraverso la mappa.

- **isObstacle():** Le coordinate reali del raggio vengono trasformate in coordinate di pixel della mappa utilizzando il metodo **point2Pix()**.
  Se il raggio "colpisce" un pixel che rappresenta un ostacolo (verificato tramite il metodo **isObstacle()**), la distanza viene registrata e il raggio si interrompe. In caso contrario, se non viene "colpito" nessun ostacolo entro il max_range, la distanza registrata è pari al valore massimo.
  Il metodo isObstacle() è strettamente legato alla struttura della mappa, in quanto verifica se un pixel della mappa rappresenta un ostacolo o meno. Un ostacolo è considerato quando un pixel è nell'intervallo che va da nero a grigio, altrimenti è un pixel disponibile.

Questo metodo verrà utilizzato successivamente nell'implementazione **updateParticlesWeight()** per l'algoritmo di localizzazione.

```
LidarData Map::getLidarScan(double x, double y, double yaw){
  LidarData lidarData;
  // Set the position and orientation
  lidarData.x = x;
  lidarData.y = y;
  lidarData.theta = yaw;
  // Set odom values to the same as the position and orientation
  lidarData.odom_x = x;
  lidarData.odom_y = y;
  lidarData.odom_theta = yaw;
  lidarData.num_readings = 360;
  // Simulate LIDAR range readings
  lidarData.range_readings.resize(lidarData.num_readings);
  const double max_range = lidar_max_range_;
  const double angle_increment = M_PI / lidarData.num_readings;
  for (int i = 0; i < lidarData.num_readings; ++i){
    double angle = yaw - M_PI_2 + i * angle_increment;
    double cos_angle = std::cos(angle);
    double sin_angle = std::sin(angle);
    double distance = 0.0;
    bool hit = false;
    while (distance < max_range){
      distance += cell_size_;
      double ray_x = x + distance * cos_angle;
      double ray_y = y + distance * sin_angle;
      cv::Point2i map_point = point2Pix(cv::Point2d(ray_x, ray_y));
      if (map_point.x >= 0 && map_point.x < map_.cols && map_point.
   y >= 0 && map_point.y < map_.rows){
        uchar pixel_value = map_.at<uchar>(map_point.y, map_point.x
   );
        if (isObstacle(pixel_value)){
          hit = true;
          break;
        }
      } else {
        break; // Ray is out of map bounds
      }
    }
    if (!hit) {
      distance = max_range;
    }
    lidarData.range_readings[i] = distance;
  }
  return lidarData;
}
```

Il comportamento finale della funzione sulla mappa mostrata in precedenza, disegnando i dati grazie all'utilizzo di OpenCV è il seguente

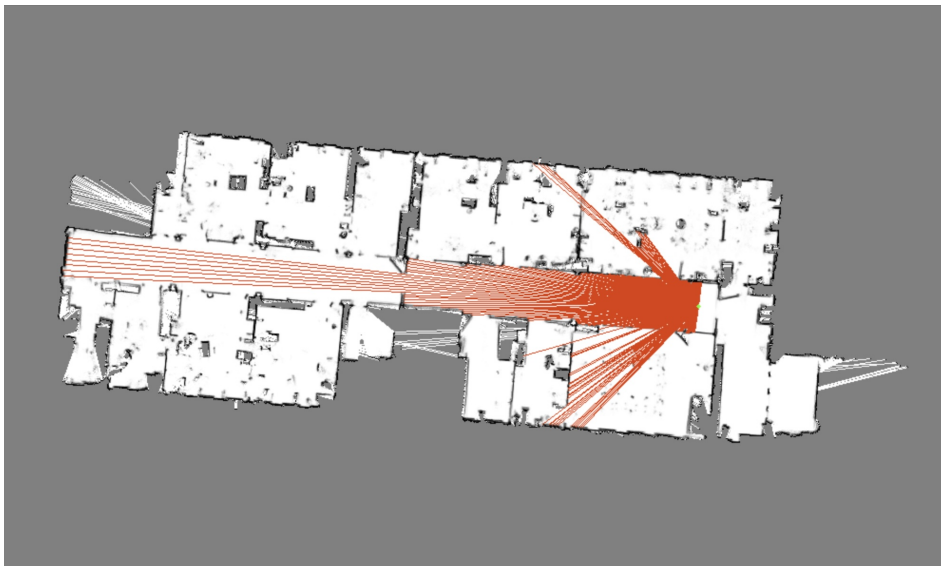

Figura 4.3: Esempio di ray casting

4.3 Implementazione dell'algoritmo MCL

Vediamo quindi l'implementazione dello pseudo codice descritto al capitolo precedente discutendo le varie funzioni che compongono l'implementazione reale.
Con la premessa di inserire nel main del programma al suo avvio i seguenti file:

- **map.png**: Il file png di una delle mappe precedentemente generate
- **log_file.clf**: Il file di log con le letture effettuate sulla mappa scelta
- **log_file_gt.clf**: Il file di log con le letture GT effettuate sulla mappa scelta

e di usare il parser per la corretta interpretazione dei file, definiamo quindi alcune variabili che servono per il funzionamento di base.

```
bool first_data = true; //flag to check first loop iteration
OdomData cur_odom, prev_odom; // current and previous reading
//Shared pointers with the data from the files
std::shared_ptr<OdomData> p_odom;
std::shared_ptr<LidarData> p_lidar;
//Initial delta position and theta
cv::Point2d delta_p(0, 0);
double delta_theta;
//Instance of ParticleFilter class
ParticleFilter pf(map);
// Set the std_dev_t, std_dev_r ad min_prob
pf.setNumParticles(num_particles);
pf.setStdDevT(std_dev_t);
pf.setStdDevR(std_dev_r);
pf.setMinProb(min_prob);
// Vectors to store positions for the metrics
std::vector<cv::Point2d> estimated_kde_positions;
std::vector<cv::Point2d> estimated_mean_positions;
```

Breve descrizione dei valori che vengono impostati sul particle filter:

- **num_particles**: è il numero di particelle con il quale si vuole eseguire la simulazione.
- **std_dev_t**: è la deviazione standard per la traslazione delle particelle in metri, quindi l'errore che verrà applicato alle particelle dopo lo spostamento del robot.
- **std_dev_r**: è la deviazione standard per la rotazione delle particelle in radianti, quindi l'errore che verrà applicato alle particelle dopo la rotazione del robot.
- **min_prob**: è il valore inferiore a cui la gaussiana che calcola il peso delle particelle verrà saturata

Andiamo quindi a vedere l'implementazione reale dell'algoritmo basato sui dati descritti.

```
// Main loop of the particle filter
while (auto data = parser.next()){
  if (p_odom = std::dynamic_pointer_cast<OdomData>(data)) {
    cur_odom = *p_odom;
  } else if (p_lidar = std::dynamic_pointer_cast<LidarData>(data))
   {
    cur_odom.x = p_lidar->odom_x;
    cur_odom.y = p_lidar->odom_y;
    cur_odom.theta = p_lidar->theta;
  }
  if (first_data){
    pf.initialize();
    first_data = false;
  } else {
    auto tmp_delta_p = rotatePoint(cv::Point2d(cur_odom.x -
   prev_odom.x, cur_odom.y - prev_odom.y), -prev_odom.theta);
    auto tmp_delta_theta = normalizeAngle(cur_odom.theta -
   prev_odom.theta);
    prev_odom = cur_odom;
    delta_p += tmp_delta_p;
    delta_theta = normalizeAngle(tmp_delta_theta + delta_theta);
    pf.updateParticlesPos(delta_p.x, delta_p.y, delta_theta);
    delta_p = cv::Point2d(0, 0);
    delta_theta = 0;
    if (p_lidar = std::dynamic_pointer_cast<LidarData>(data)){
      pf.updateParticlesWeight(p_lidar);
      pf.resampleParticles();
      auto estimateKDEPosition = pf.estimateStateKDE();
      auto estimateMeanPosition = pf.estimateStateMean();
      estimated_kde_positions.emplace_back(
        estimateKDEPosition.x, estimateKDEPosition.y
      );
      estimated_mean_positions.emplace_back(
        estimateMeanPosition.x, estimateMeanPosition.y
      );
    }
  }
}
```

Come si può notare l'algoritmo cicla sul parser e per ogni linea contenuta nel file clf interpreta la riga stessa e se questa contiene letture di odometria allora va ad impostare la variabile `cur_odom` con i valori letti, altrimenti vengono impostati con i valori di odometria della lettura LiDAR.

Dopo questo passaggio iniziale comincia il vero e proprio lavoro dell'algoritmo. Se è una prima lettura viene inizializzato il particle filter usando la sua funzione di `initialize()`, definita come segue:

```
void ParticleFilter::initialize(){
  particles.reserve(num_particles_);
  sampleRandomParticles(particles, num_particles_);
  for (auto& particle : particles)
    particle.weight = 1.0 / num_particles_;
}
```

Che a sua volta utilizza la funzione di `sampleRandomParticles()` che va ad inizializzare le particelle in maniera casuale sulla mappa.

```
void ParticleFilter::sampleRandomParticles(
  std::vector<Particle>& p,
  int num_part,
  bool concatenate)
{
  std::default_random_engine generator;
  if (!concatenate) p.clear();
  std::normal_distribution<double> dist_theta(0.0, 2.0 * M_PI);
  for (int i = 0; i < num_part; ++i){
    cv::Point2i pixel_position = map_.randomPositionInMap();
    cv::Point2d metersPosition = map_.pixToMeters(pixel_position);
    Particle particle;
    particle.x = metersPosition.x;
    particle.y = metersPosition.y;
    particle.theta = dist_theta(generator);
    particle.weight = 0;
    p.push_back(particle);
  }
}
```

Disegnando la mappa fino a questo punto del programma si ottiene questo risultato:

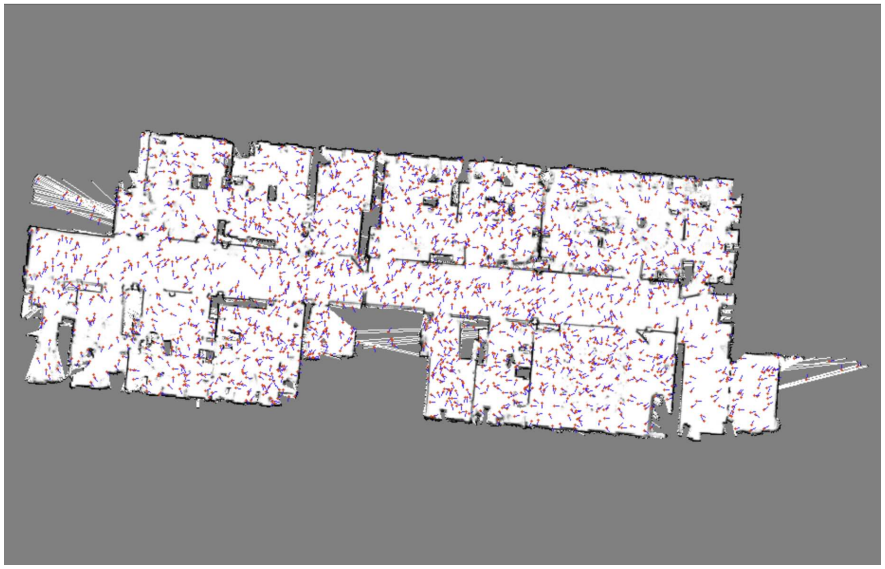

Figura 4.4: Mappa con il posizionamento iniziale delle particelle

Analizziamo ora il comportamento a regime della implementazione descrivendo punto per punto quello che succede nel corpo del ramo else, cioè se non è una prima lettura.
L'algoritmo per ogni lettura da qui in poi calcola una variabile `tmp_delta_p` in cui viene inserito un valore temporaneo della posizione utilizzando la funzione di supporto `rotatePoint()` che semplicemente applica una rotazione al punto in 2D a cui viene data in input la differenza tra la precedente lettura e quella attuale insieme al valore della rotazione.

```
inline cv::Point2d rotatePoint(cv::Point2d p, double yaw){
  cv::Point2d rot_p;
  rot_p.x = cos(yaw) * p.x - sin(yaw) * p.y;
  rot_p.y = sin(yaw) * p.x + cos(yaw) * p.y;
  return rot_p;
}
```

Dopodichè viene calcolata la varibaile `tmp_delta_theta` tramite la differenza dei theta della lettura attuale e della precedente il cui valore viene normalizzato usando la funzione `normalizeAngle()` il cui nome è autoesplicativo.

```
inline double normalizeAngle(double angle){
  while (angle > M_PI){
    angle -= 2.0 * M_PI;
  }
  while (angle < -M_PI){
    angle += 2.0 * M_PI;
  }
  return angle;
}
```

Alla fine di questo processo viene aggiornata la lettura precedente con quella attuale. Vengono quindi aggiornati valori del `delta_p` e del `delta_theta` per essere dati in input alla funzione di `updateParticlePost()`. Questa è una delle funzioni più importanti del particle filter, quella che aggiorna la posizione delle particelle ad ogni iterazione, l'effettiva implmentazione del Motion Model.

Per il motion model si è scelto di fare una implementazione semplificata che va ad aggiornare le particelle con i valori del delta.

```
void ParticleFilter::updateParticlesPos(
  double delta_x,
  double delta_y,
  double delta_theta)
{
  double delta_t = sqrt(delta_x * delta_x + delta_y * delta_y);
  std::default_random_engine generator(rnd_seed_);
  std::normal_distribution<double> noise_x(0.0, std_dev_t_);
  std::normal_distribution<double> noise_y(0.0, std_dev_t_);
  std::normal_distribution<double> noise_theta(0.0, std_dev_r_);
  delta_theta = normalizeAngle(delta_theta) ;
  for (auto& particle : particles){
    auto delta_p = rotatePoint(cv::Point2d(delta_x, delta_y),
   particle.theta);
    particle.x += delta_p.x + noise_x(generator);
    particle.y += delta_p.y + noise_y(generator);
    particle.theta += delta_theta + noise_theta(generator);
    // Normalize angle
    particle.theta = normalizeAngle(particle.theta);
  }
}
```

L'implementazione fa uso della `normal_distribution` per creare dei valori random di errore basati sulle deviazioni standard che sono state definite nella inizializzazione del particle filter. Questi vengono poi sommati ai valori in delta delle posizioni x, y e θ delle letture prima di aggiornare la posizione x, y e θ della particella stessa.

Esistono diversi modi di calcolare la posizione delle particelle, un'altra possibile implementazione poteva essere quella del motion model basato sull'odometria. Questa strada è stata scelta per via della sua implementazione semplificata, facile da comprende, avendo anche tenuto conto che una implementazione più sofisticata non avrebbe portato molto valore allo scopo finale dell'elaborato.

Dopo aver finito di fare l'aggiornamento della posizione delle particelle, l'algoritmo resetta i valori di default di `delta_p` e `delta_theta` dichiarati nel Main del programma. Le operazioni descritte fino ad ora vengono fatte sia per le letture LiDAR che per gli spostamenti di odometria. Cominciamo ora ad analizzare nel dettaglio quando una lettura è LiDAR quali sono le operazioni da eseguire. Come si può notare dal codice del programma, nel caso di una lettura LiDAR vengono eseguite le seguenti funzioni:

- **updateParticlesWeight()**
- **resampleParticles()**
- **estimatePositions()**

4.3.1 Aggiornamento del peso delle particelle

La funzione di aggiornamento del peso utilizza la lettura LiDAR per generare un peso per ogni particella, grazie al metodo di `getParticleWeight()`

```
void ParticleFilter::updateParticlesWeight(
  std::shared_ptr<LidarData> lidar_data)
{
  for (auto& particle : particles){
    particle.weight = getParticleWeight(
      particle.x,
      particle.y,
      particle.theta,
      lidar_data
    );
  }
}
```

Vediamo quindi nella prossima pagina l'implementazione della funzione ausiliaria `getParticleWeight()`

```
double ParticleFilter::getParticleWeight(double x, double y, double
    theta, std::shared_ptr<LidarData> lidar_data)
{
  double total_probability = 0.0;
  // Simulate LIDAR scan for the particle's position and
   orientation
  auto particle_lidar_scan = map_.getLidarScan(x, y, theta);
  // Calculate the total probability based on the Gaussian function
  int num_readings = lidar_data->range_readings.size();
  for (int i = 0; i < num_readings; ++i) {
    total_probability += gauss_func(lidar_data->range_readings[i],
   particle_lidar_scan.range_readings[i],std_dev_t_);
  }
  return total_probability / num_readings;
}
```

Questo metodo a sua volta utilizza la funzione definita nella classe Map: `getLidarScan()` discusso al paragrafo precedente.
Il peso delle particelle è calcolato grazie all'utilizzo di una funzione gaussiana saturata ad un valore `min_prob`, che era stato definito nell'inizializzazione del particle filter. Diamo quindi l'implementazione, a cui viene data in input valore per valore la lettura del LiDAR dal file clf ed il valore del range delle letture simulate in precedenza dalla funzione della mappa.

```
double ParticleFilter::gauss_func(double x, double mean, double
    std_dev){
  double exponent = -((x - mean) * (x - mean)) / (2 * std_dev *
   std_dev);
  double val = exp(exponent);
  return (val < min_prob_) ? min_prob_ : val;
}
```

A questo punto dell'algoritmo tutte le particelle avranno un nuovo peso basato sulla loro nuova posizione.

4.3.2 Ricampionamento delle particelle

Bisogna quindi fare il ricampionamento delle particelle in base al loro nuovo peso. Si è deciso di implementare il `"Low Variance Resamping"`, che ha 2 vantaggi principali:

- Minimizza la perdita di diversità nelle particelle
- Riduce il rischio che solo poche particelle vengano selezionate ripetutamente

```cpp
void ParticleFilter::resampleParticles(){
  // Step 1: Create a new container for resampled particles and reserve space
  std::vector<Particle> resampled_particles;
  resampled_particles.reserve(num_particles_);
  // Step 2: Calculate the total weight sum and normalize weights in one loop
  double total_weight = 0.0;
  for (auto& particle : particles){
    if (map_.isOutside(cv::Point2d(particle.x, particle.y)) || map_.isObstacle(cv::Point2d(particle.x, particle.y))){
        particle.weight = 0;
    }
    total_weight += particle.weight;
  }
  // Step 3: Check for zero total weight and handle the error
  if (total_weight == 0.0){
    initialize(); // Reinitializes all particles
    return;
  }
  // Step 4: Normalize the weights of all particles
  double normalization_factor = 1.0 / total_weight;
  for (auto& particle : particles){
    particle.weight *= normalization_factor;
  }
  // Step 5: Perform low variance resampling
  std::default_random_engine random_generator(rnd_seed_);
  std::uniform_real_distribution<double> random_dist(0.0, 1.0 / num_particles_);
  double random_offset = random_dist(random_generator);
  double cumulative_weight = particles[0].weight;
  int index = 0;
  int num_rnd_part = static_cast<int>(num_particles_ * rnd_part_perc_);
  for (int m = 0; m < num_particles_ - num_rnd_part; ++m){
    double target_weight = random_offset + m * (1.0 / num_particles_);
    while (target_weight > cumulative_weight && index < num_particles_ - 1){
      ++index;
      cumulative_weight += particles[index].weight;
    }
    resampled_particles.push_back(particles[index]);
  }


```

```
  // Step 6: Add a percentage of random particles
  sampleRandomParticles(resampled_particles, num_rnd_part, true);
  // Step 7: Replace old particles with the resampled particles
  particles = std::move(resampled_particles);
  // Step 8: Reset the weights of all particles to uniform
   distribution
  double uniform_weight = 1.0 / num_particles_;
  for (auto& particle : particles){
    particle.weight = uniform_weight;
  }
}
```

A questo punto tutte le particelle sono state ricampionate in base al loro peso, questo ci permette infine di calcolare la posizione stimata per la particella con peso maggiore. Facendo questo si vuole ottenere una lettura da poter confrontare con i dati reali del file "GT". Discuteremo tuttavia questo confronto quando andremo a vedere i risultati delle metriche sperimentali.
Per implementare la stima della posizione sono state implementati due diversi algoritmi, uno basato sulla media dei pesi ed uno basato sulla stima della densità con Kernel (`Kernel Density Estimation`), metodo che stima una distribuzione di probabilità continua utilizzando i dati campione, sommando funzioni kernel (come gaussiane) centrate su ciascun punto per mostrare dove i valori sono più probabili. Vediamo quindi le due implementazioni.

```
Particle ParticleFilter::estimateStateKDE() const {
  if (particles.empty()){
    std::cerr << "Error: No particles available for state
   estimation." << std::endl;
    throw std::runtime_error("No particles available for state
   estimation");
  }
  Particle mode_particle;
  double max_density = -std::numeric_limits<double>::infinity();
  // Determine the optimal bandwidth (optional, you can set this
   dynamically)
  double bandwidth = 1.0;
  for (const auto& particle : particles) {
    double density = estimateDensity(particle, bandwidth);
    if (density > max_density){
      max_density = density;
      mode_particle = particle;
    }
  }
  return mode_particle;
}
```

```
// Kernel Density Estimation (KDE) function
double ParticleFilter::estimateDensity(const Particle& p, double
    bandwidth) const {
  double density = 0.0;
  for (const auto& other_particle : particles){
    double distance_squared =
    (p.x - other_particle.x) * (p.x - other_particle.x) +
        (p.y - other_particle.y) * (p.y - other_particle.y);
    density += gaussianKernel(distance_squared, bandwidth);
  }
  // Normalize density by the number of particles
  density /= static_cast<double>(particles.size());
  return density;
}

// Gaussian kernel function
double ParticleFilter::gaussianKernel(double distance_squared,
    double bandwidth) const {
  // Adding a small epsilon to bandwidth to avoid division by zero
  const double epsilon = 1e-9;
  double adjusted_bandwidth = std::max(bandwidth, epsilon);
  double exponent = -distance_squared / (2 * adjusted_bandwidth *
    adjusted_bandwidth);
  return std::exp(exponent) / (adjusted_bandwidth * std::sqrt(2 *
    M_PI));
}
```

Implementazione della stima basata sulla media dei pesi

```
Particle ParticleFilter::estimateStateMean() const {
  double sum_x = 0.0;
  double sum_y = 0.0;
  double sum_sin_theta = 0.0;
  double sum_cos_theta = 0.0;
  for (const auto& particle : particles){
    sum_x += particle.x;
    sum_y += particle.y;
    sum_sin_theta += std::sin(particle.theta);
    sum_cos_theta += std::cos(particle.theta);
  }
  // Compute the weighted averages
  Particle estimate;
  estimate.x = sum_x / particles.size();
  estimate.y = sum_y / particles.size();
  estimate.theta = std::atan2(sum_sin_theta, sum_cos_theta);
  return estimate;
}
```

Finita questa fase, il ciclo while è a regime e continuerà ad effettuare le operazioni descritte fin ora per tutte le letture del file clf fino al suo esaurimento.
Volendo disegnare sulla mappa questo comportamento il risultato che si ottiene con il ciclo a regime è il seguente, prestando bene attenzione al fatto che questa volta sono state disegnate sulla mappa anche:

- La traiettoria del robot basata sulle letture del file "GT"
- La posizione robot stesso con un cerchio rosso
- La posizione stimata del robot con un cerchio verde

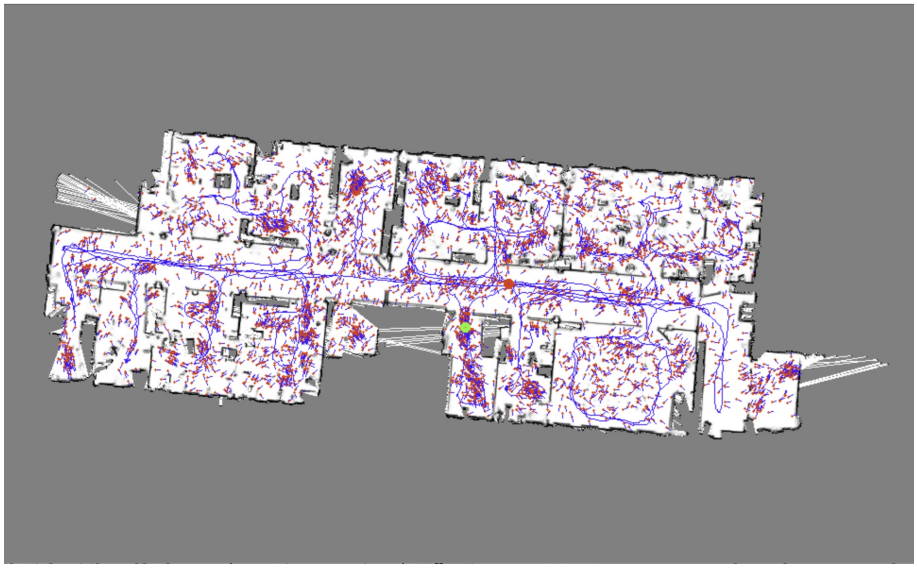

Figura 4.5: Mappa di Freiburg dopo poche iterazioni del ciclo

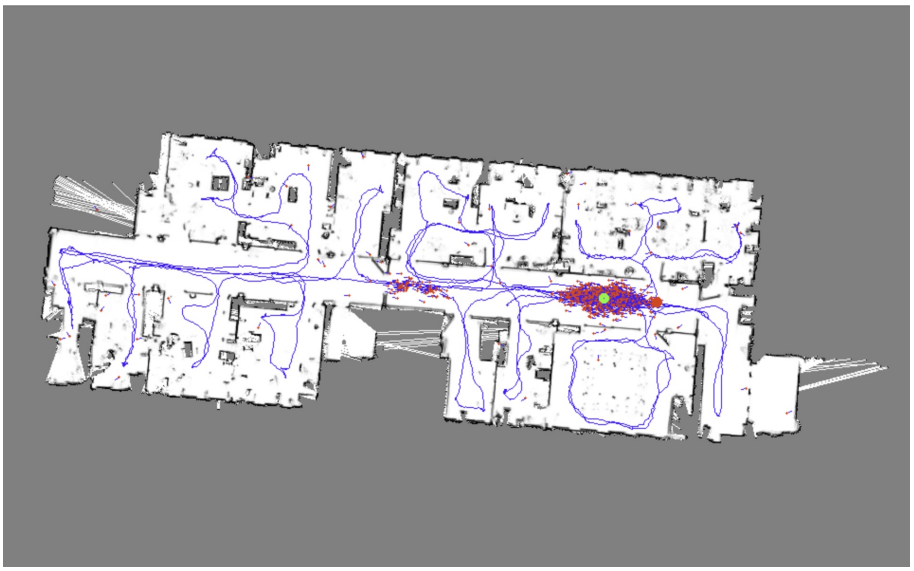

Figura 4.6: Mappa di Freiburg con le particelle che stanno andando in convergenza

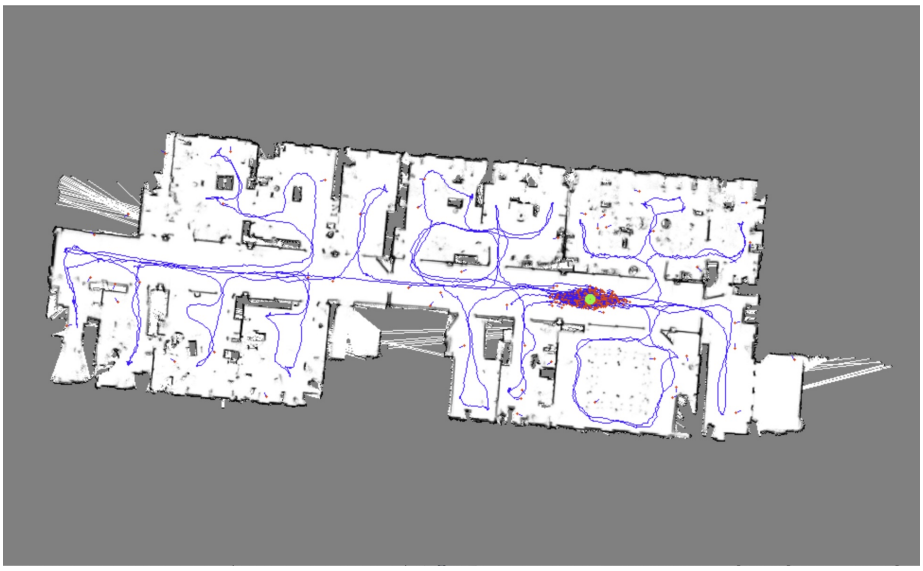

Figura 4.7: Mappa di Freiburg in convergenza

4.4 Risultati Sperimentali

In questa sezione vediamo come le due funzioni di stima della posizione vengono confrontate con la posizione reale del robot. Per fare ciò è stato deciso di utilizzare un algoritmo che implementa il modello matematico di una spezzata.
In matematica, una spezzata è una figura geometrica costituita da una sequenza di segmenti consecutivi, detti lati della spezzata, che collegano una serie di punti distinti, detti vertici della spezzata.
Inoltre questo l'algoritmo usa una scala di rapporto di convergenza che è una misura utilizzata per quantificare la proporzione di stime che sono considerate "convergenti" rispetto al totale delle stime effettuate. Il rapporto di convergenza viene calcolato confrontando le stime generate dall'algoritmo (sia le stime basate su KDE che quelle basate sulla media) con le vere posizioni del robot. Questa scala di convergenza va da 0 a 1, dove 1 rappresenta la convergenza entro il valore di threshold in questo caso impostato a 1 metro.

$$\text{Rapporto di convergenza} = \frac{\textbf{Numero di stime convergenti}}{\textbf{Numero totale di stime}}$$

Le prove sono state fatte seguendo una logica particolare, prima di tutto si è deciso di fare diverse simulazioni dell' algoritmo MCL su 3 diverse mappe in base a 4 fattori:

- **Numero di particelle**: con i seguenti range, 1000, 3000, 5000
- **Deviazione standard della traslazione**: in metri da 0.05 a 0.1 con 0.01 di aumento ad ogni iterazione

- **Deviazione standard della rotazione**: in radianti da 0.01 a 0.04 con 0.01 di aumento ad ogni iterazione
- **Saturazione della funzione gaussiana del peso**: min_prob da 0 a 1 con 0.025 di aumento ad ogni iterazione

Queste prove sono state salvate in un file dove ogni riga ha questo formato:

```
t = 0.05, r = 0.01, min_prob = 0, kde = [(0.0614455, 3.51321)...],
    mean = [(0.0614455, 3.51321)];
```

dove t è il valore di traslazione, r il valore della rotazione, kde e mean sono array con le posizioni x e y. Questo file contiene tutti i valori calcolati per ogni fattore descritto sopra, per un totale di 120 simulazioni per ogni range di particelle.
Grazie a questo file si riesce quindi a trovare qual'è la configurazione migliore e calcolare su questa un plot che mostra visivamente quale è il range di convergenza entro il metro di distanza su ogni mappa.
Il valore del threshold è stato scelto per mostrare la affidabilità dell'implementazione. Un metro di divergenza dalla posizione reale è grande abbastanza da vedere il comportamento corretto, tuttavia si può configurare e fare prove anche per valori più stretti dove l'affidabilità potrebbe scendere fino a non essere più una implementazione corretta.
Da questo file viene creata una struttura definita come segue:

```
struct EstimationData {
    std::vector<Particle> kde_estimations;
    std::vector<Particle> mean_estimations;
};
using Estimations = std::map<std::tuple<double, double, double>,
    EstimationData>;
```

Per semplicità tralasciamo sia l'implementazione del parser, sia quella dell'algoritmo che trova la migliore simulazione.
Andiamo quindi a vedere quali sono i grafici generati per le seguenti mappe:

- **fr079.png**
- **belgioioso.png**
- **orebro.png**

Di cui alla pagina seguente vedremo due illustrazioni per completezza.

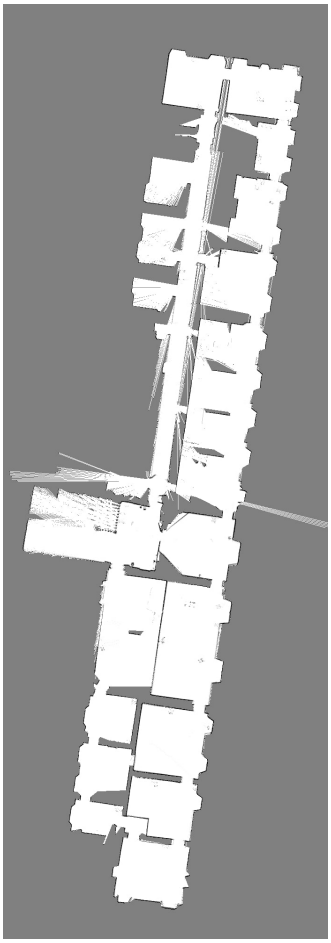

Figura 4.8: Mappa Belgioioso

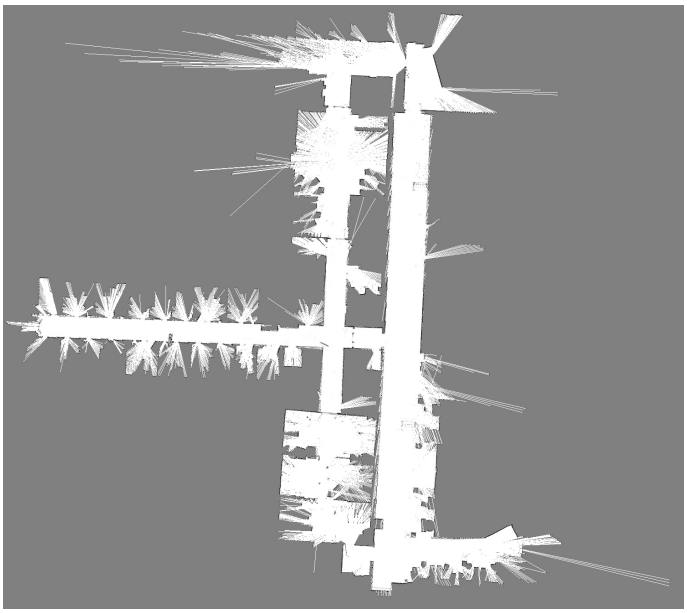

Figura 4.9: Mappa Orebro

I risultati sono stati ottenuti facendo le prove con le seguenti specifiche tecniche dell'hardware:

- **Processore (CPU)**: Apple M1 chip
  - 8-core CPU con 4 performance cores e 4 efficiency cores
  - Architettura ARM
  - Frequenza base: 3.2 GHz
  - 5-nanometer process technology
- **Memoria (RAM)**: 16 GB di memoria unificata (LPDDR4X)
  - Velocità della memoria: 4266 MHz
  - Memoria condivisa tra CPU e GPU
  - Alta efficienza energetica grazie alla memoria unificata

Vengono fornite queste informazioni dato che nei paragrafi successivi si andrà a presentare una sitma del tempo medio delle fasi di predizione e correzione di una singola esecuzione dell'intero algoritmo MCL. Per effettuare tuttavia le 120 simulazioni già discusse è stato scelto di usare un sistema multi-thread, grazie all'architettura multicore del processore M1 Pro e alla memoria unificata tra CPU e GPU, il sistema multi-thread sfrutta appieno le capacità del dispositivo, permettendo di eseguire le simulazioni in parallelo e ottenere i risultati in tempi significativamente inferiori rispetto a un'esecuzione sequenziale.

4.4.1 Risultati su fr079.png

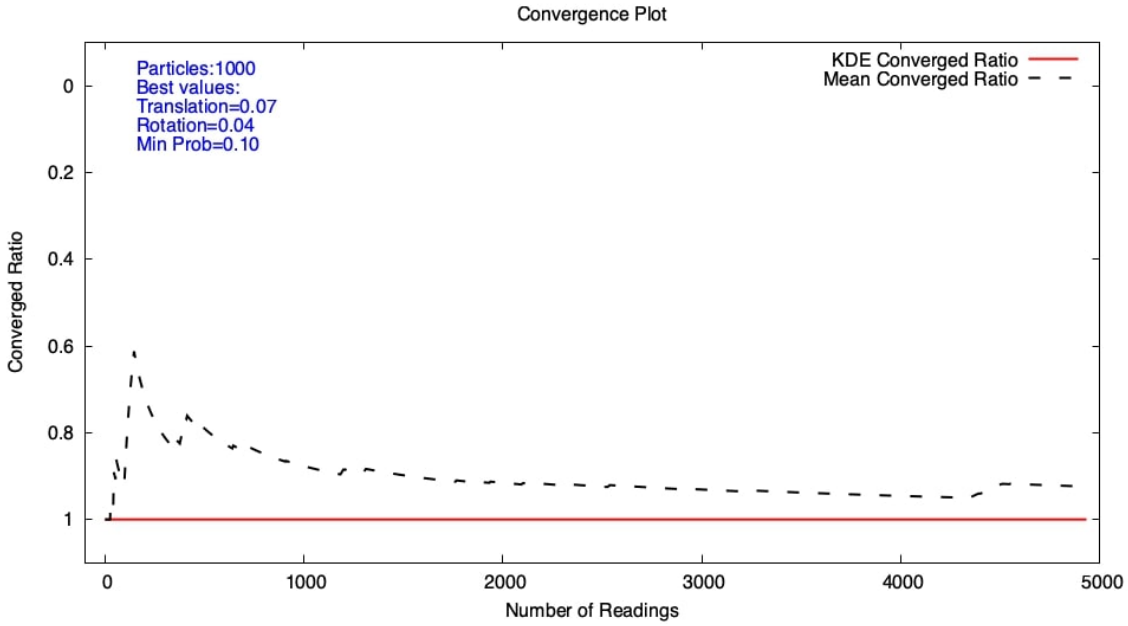

Figura 4.10: Risultati Freiburg per 1000 particelle

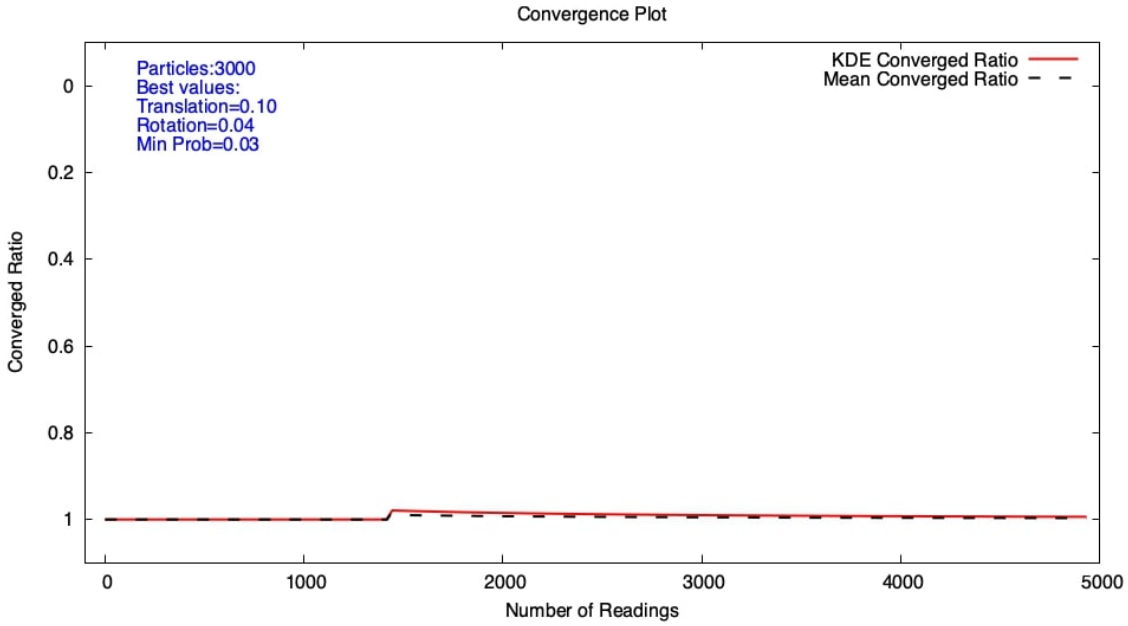

Figura 4.11: Risultati Freiburg per 3000 particelle

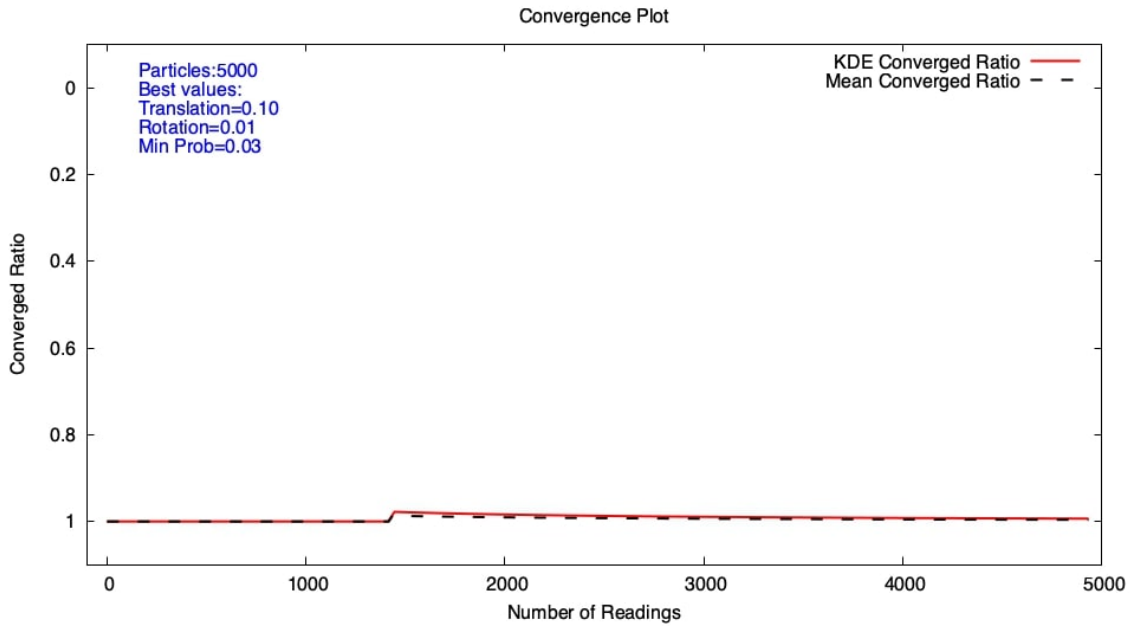

Figura 4.12: Risultati Freiburg per 5000 particelle

Analizzando i grafici sulla mappa di Freiburg, si può notare come all'aumentare del numero di particelle la convergenza entro il threshold di un metro aumenta sia per le stime in KDE che per quelle basate sulla media.

Vengono riportati inoltre i parametri della simulazione con il rapporto di convergenza migliore. Interessante notare anche che c'è un picco ad un certo punto delle letture il che sta a significare che la stima è uscita dalla convergenza per poi piano piano riallinearsi.

Numero Particelle	**Tempo Medio di Predizione (s)**	**Tempo Medio di Correzione (s)**
1000	5.71955e-05	0.0349773
3000	0.000179261	0.127778
5000	0.00030607	0.242607

Tabella 4.1: Tempi di esecuzione fr079.png

4.4.2 Risultati su belgioioso.png

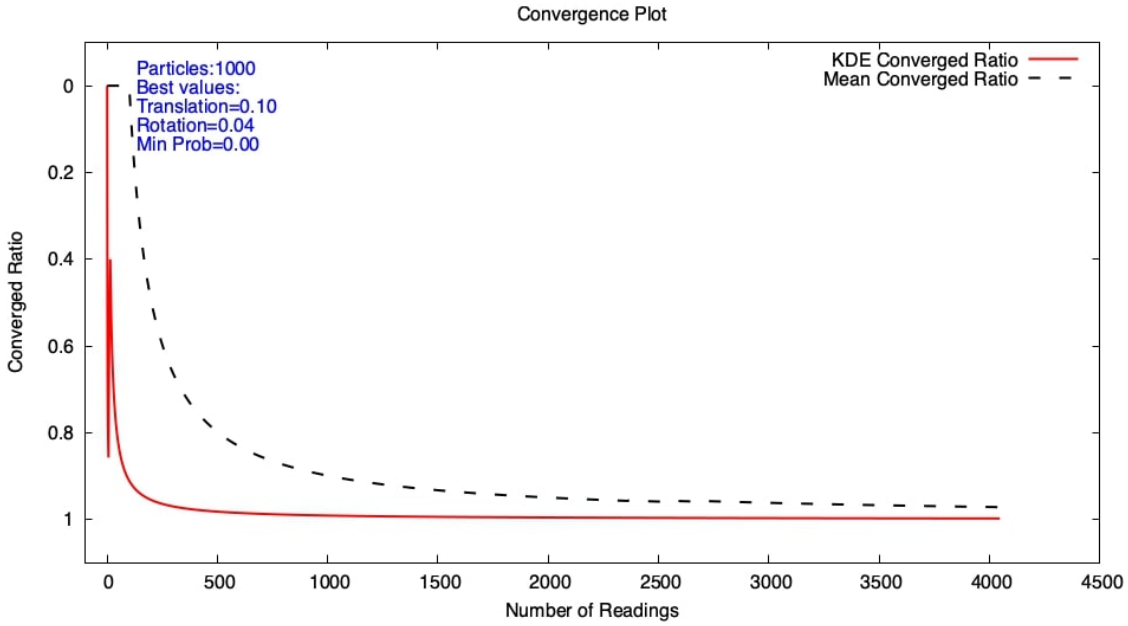

Figura 4.13: Risultati Belgioioso per 1000 particelle

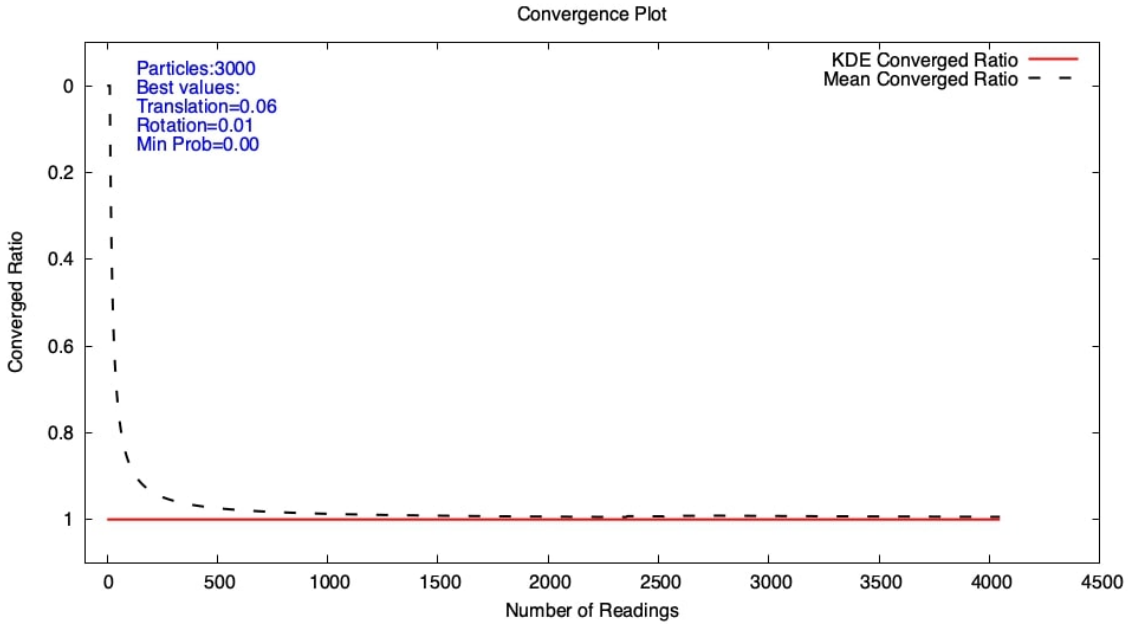

Figura 4.14: Risultati Belgioioso per 3000 particelle

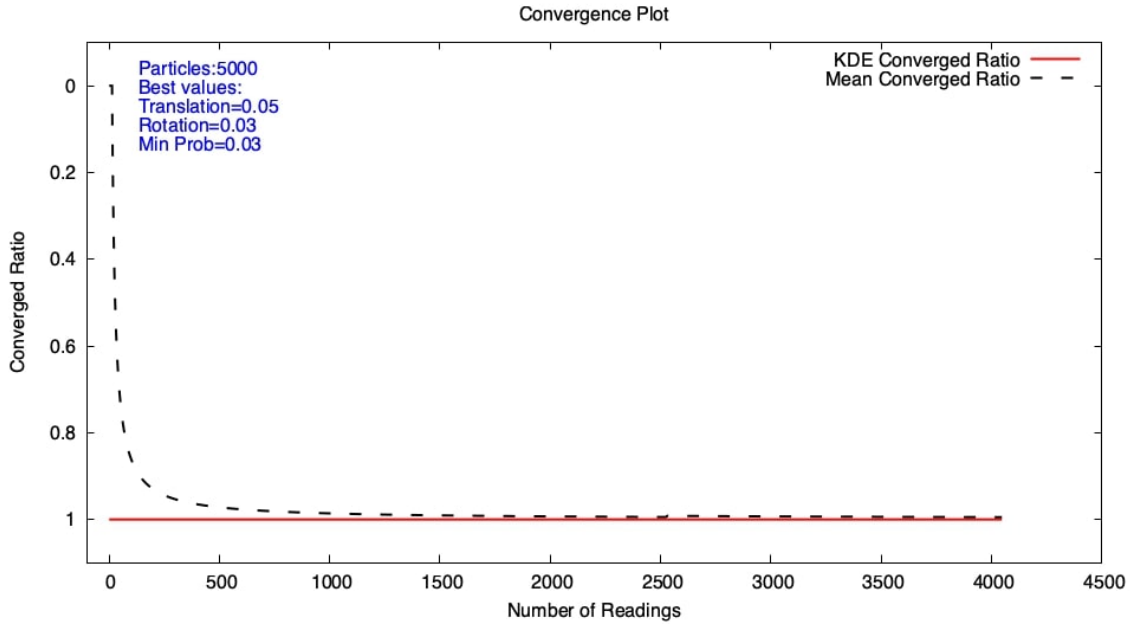

Figura 4.15: Risultati Belgioioso per 5000 particelle

Analizzando i grafici di questa mappa possiamo notare subito che la media si comporta diversamente, ed arriva in convergenza più tardi rispetto alla stima KDE. Questo perchè ci sono stati più gruppi di particella con un valore del peso alto.

Anche da questi grafici si può notare che all'aumento del numero di particelle la convergenza cambia, addirittura nel caso delle simulazioni con 3000 particelle e delle 5000 particelle le stime sono sempre state entro il metro di convergenza dall'inizio alla fine.

Numero Particelle	**Tempo Medio di Predizione (s)**	**Tempo Medio di Correzione (s)**
1000	5.69211e-05	0.0425227
3000	0.000177684	0.166317
5000	0.000298361	0.281057

Tabella 4.2: Tempi di esecuzione belgioioso.png

4.4.3 Risultati su orebro.png

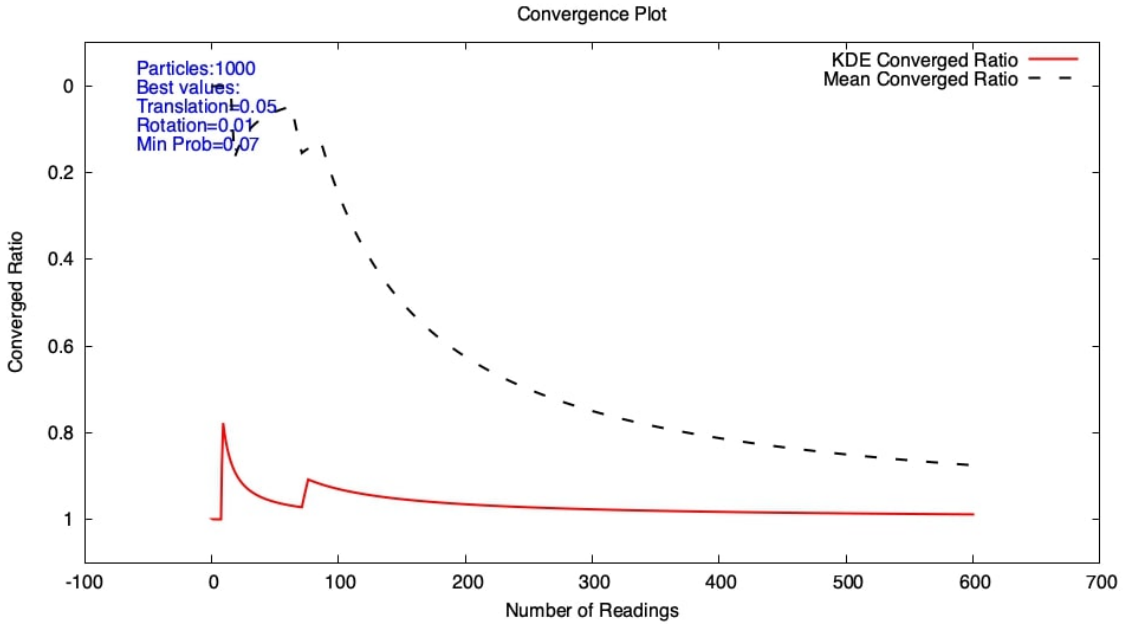

Figura 4.16: Risultati Orebro per 1000 particelle

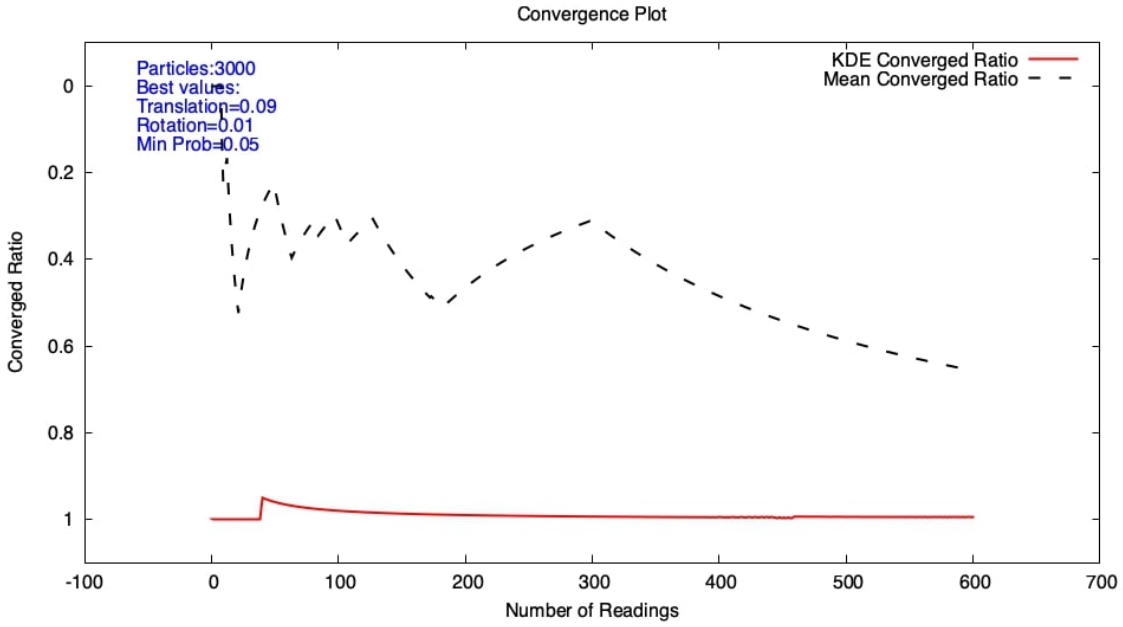

Figura 4.17: Risultati Orebro per 3000 particelle

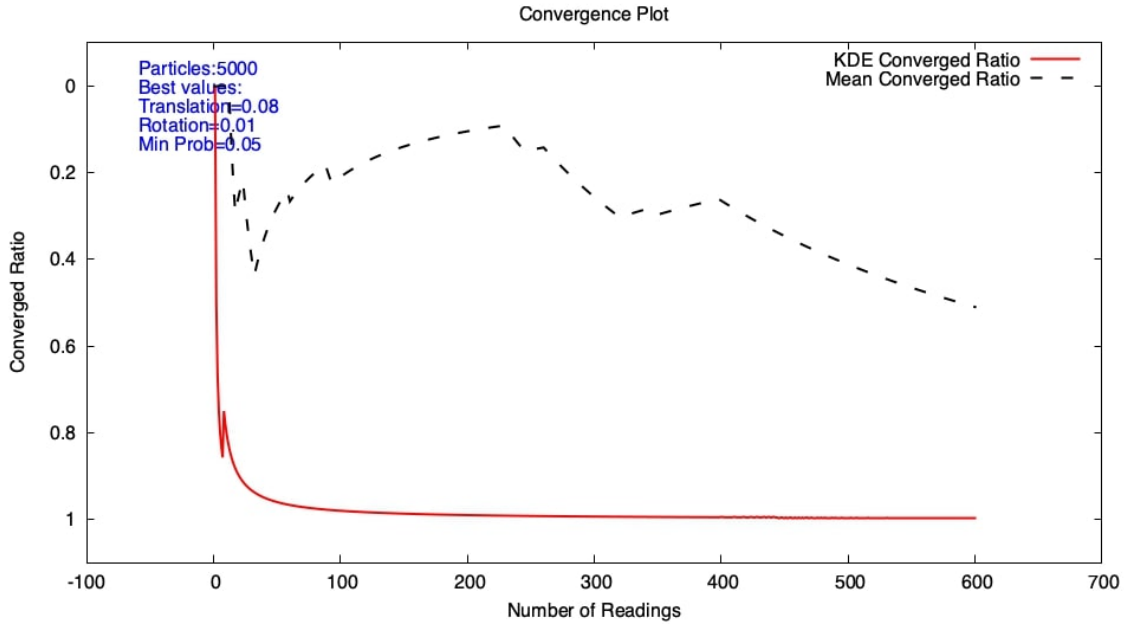

Figura 4.18: Risultati Orebro per 5000 particelle

I risultati su questa mappa sono più interessanti, si può notare come la media non vada mai in convergenza, questo perché come discusso prima ci sono vari gruppi di particelle con un peso alto. Si vede bene anche come con 5000 particelle le stime KDE vadano in convergenza come una funzione limite dopo poche iterazioni.

Si nota inoltre che su questa mappa in tutte le simulazioni c'è un picco in cui la stima perde la convergenza, probabilmente perché durante quelle iterazioni un gruppo di particelle diverso rispetto a quello corretto va ad ottenere un alto valore del peso disturbando momentaneamente la convergenza.

Numero Particelle	**Tempo Medio di Predizione (s)**	**Tempo Medio di Correzione (s)**
1000	6.01013e-05	0.0494658
3000	0.000188452	0.153323
5000	0.000307488	0.27377

Tabella 4.3: Tempi di esecuzione orebro.png

Capitolo 5

Conclusioni

In questa tesi abbiamo dimostrato l'efficacia di un particle filter gestito con il metodo Monte Carlo (MCL) applicato alla localizzazione di robot mobili in ambienti statici. Attraverso simulazioni condotte su diverse mappe, utilizzando un numero variabile di particelle, è stato possibile evidenziare come l'aumento del numero di particelle migliori significativamente la precisione della stima della posizione del robot, specialmente su mappe complesse.

Il confronto tra le diverse mappe Freiburg, Belgioioso e Orebro, ha sottolineato come il comportamento dell'algoritmo vari in base alle caratteristiche dell'ambiente e del numero di particelle utilizzate. In particolare, i risultati ottenuti mostrano che all'aumentare del numero di particelle si ottiene una convergenza più rapida e accurata, come evidenziato dalle simulazioni con 5000 particelle che hanno mantenuto una stima entro il metro di tolleranza per la maggior parte delle posizioni.

Inoltre, l'utilizzo di due diversi metodi di stima, uno basato sulla distribuzione KDE e l'altro sulla media delle particelle, ha permesso di ottenere una visione più completa del comportamento del robot. I risultati hanno indicato che in alcuni casi il metodo basato sulla media può risultare meno preciso rispetto al metodo KDE, specialmente in situazioni in cui ci sono più gruppi di particelle con pesi elevati.

L'aumento delle particelle tuttavia è limitante dal punto di vista computazionale, dato che ad un incremento delle particelle si riscontra un aumento sostanziale del tempo impiegato per la conclusione dell'algoritmo.

Nonostante l'efficacia dell'implementazione attuale, ci sono quindi possibili miglioramenti di cui possiamo prendere nota:

- **Motion model:** Un primo possibile miglioramento riguarda l'adozione di un motion model più sofisticato. In questo lavoro è stato utilizzato un modello semplice con rumore aggiunto, ma si potrebbero esplorare altre implementazioni come il modello basato sull'odometria o il velocity motion model, che potrebbero fornire una stima più accurata in scenari dinamici.

- **MCL Adattivo:** Un altro miglioramento potrebbe essere l'implementazione di una variante adattativa dell'algoritmo di localizzazione Monte Carlo. Questa variante prevede l'aggiunta di campioni casuali, con il numero di campioni determinato dal confronto tra la probabilità a breve termine e quella a lungo termine delle misurazioni sensoriali. Ciò permetterebbe al sistema di adattarsi meglio ai cambiamenti improvvisi nell'ambiente, migliorando la robustezza del sistema di localizzazione.

- **Sensori aggiuntivi:** L'integrazione di sensori supplementari, come gli encoder, i giroscopi, le telecamere o i sensori a ultrasuoni, rappresenta un ulteriore miglioramento per l'algoritmo MCL. Questi sensori forniscono informazioni aggiuntive che, combinate con i dati del LiDAR, permettono di ridurre ulteriormente l'incertezza nella stima della posizione del robot, migliorando la precisione e l'affidabilità del sistema di localizzazione.

In conclusione, l'implementazione discussa ha dimostrato una buona efficacia nella localizzazione del robot in ambienti statici, con margini di miglioramento legati all'adozione di motion model più complessi e varianti adattative dell'algoritmo MCL. L'integrazione di sensori aggiuntivi come quelli discussi in precedenza potrebbe ulteriormente incrementare la precisione e la robustezza del sistema in scenari reali più complessi.

Bibliografia

[1] H. Andreasson, G. Grisetti, T. Stoyanov e A. Pretto. «Sensors for Mobile Robots». In: *Encyclopedia of Robotics*. A cura di M.H. Ang, O. Khatib e B. Siciliano. Springer, Berlin, Heidelberg, 2020.

[2] Sebastian Thrun, Wolfram Burgard e Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

Link Utili

- Slamevaluation, mappe e informazioni utili:
  `http://ais.informatik.uni-freiburg.de/slamevaluation/index.php`
- OpenCV informazioni utili:
  `https://opencv.org`

Elenco delle figure

1.1 Robot aspirapolvere moderno 6

3.1 Gaussiana normalizzata a 1 e saturata a min_prob 30

4.1 Modello del robot utilizzato per le sperimentazioni 31
4.2 Mappa di Freiburg generata dalle letture gt 33
4.3 Esempio di ray casting 36
4.4 Mappa con il posizionamento iniziale delle particelle 40
4.5 Mappa di Freiburg dopo poche iterazioni del ciclo 47
4.6 Mappa di Freiburg con le particelle che stanno andando in convergenza 47
4.7 Mappa di Freiburg in convergenza 48
4.8 Mappa Belgioioso 50
4.9 Mappa Orebro 50
4.10 Risultati Freiburg per 1000 particelle 52
4.11 Risultati Freiburg per 3000 particelle 52
4.12 Risultati Freiburg per 5000 particelle 53
4.13 Risultati Belgioioso per 1000 particelle 54
4.14 Risultati Belgioioso per 3000 particelle 54
4.15 Risultati Belgioioso per 5000 particelle 55
4.16 Risultati Orebro per 1000 particelle 56
4.17 Risultati Orebro per 3000 particelle 56
4.18 Risultati Orebro per 5000 particelle 57