UNIVERSITÀ DI FIRENZE
DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Ingegneria Informatica

Controllo di copertura dinamico per il monitoraggio cooperativo in tempo reale

Relatore

Giorgio Battistelli

Candidato

Ivan Necerini

Correlatore

Nicola Forti

Anno accademico 2023 / 2024

Indice

1 Introduzione

Il controllo di copertura dinamico è un problema centrale in molti campi applicativi, tra cui la robotica, la sorveglianza, la gestione delle emergenze e le operazioni militari. L'obiettivo principale di questo problema è quello di garantire che un'area di interesse sia monitorata in modo efficace e continuo da una flotta di agenti mobili, come droni o veicoli autonomi, equipaggiati con sensori. *Questo compito diventa particolarmente complesso e, interessante e pratico quando l'ambiente è dinamico e i punti di interesse (targets) sono in movimento.* Il controllo dinamico della copertura è dunque un tipo di controllo cooperativo che richiede a un sistema multi-agente di monitorare dinamicamente un'area di interesse nel tempo (2). Questo tipo di controllo è ampiamente utilizzato in una varietà di applicazioni, come esplorazione, ricognizione, sorveglianza e tracciamento dei bersagli.

Nella letteratura, esistono due principali categorie di controllo della copertura: statica e dinamica. Il controllo della copertura statica si occupa di determinare il posizionamento ottimale degli agenti in modo che le aree di interesse siano completamente coperte dall'unione delle zone di rilevamento degli agenti. Al contrario, il controllo della copertura dinamica sfrutta la mobilità degli agenti per monitorare continuamente le aree di interesse nel tempo. In altre parole, ogni area deve essere monitorata da agenti mobili per un periodo sufficiente a garantire una copertura efficace. Il controllo della copertura statica, tuttavia, presuppone implicitamente la disponibilità di un numero sufficiente di agenti o la presenza di aree di interesse di dimensioni ridotte. In caso contrario, non vi è alcuna garanzia che tutte le aree possano essere completamente coperte. Queste limitazioni possono essere superate con il controllo della copertura dinamica, che permette il monitoraggio di vaste aree di interesse utilizzando un numero ridotto di agenti.
Tuttavia, sia nel controllo della copertura statica che dinamica, è spesso necessario disporre di modelli dell'ambiente e della cinematica/dinamica degli agenti per sviluppare leggi di controllo del movimento efficaci (2).

Ottenere modelli accurati può risultare restrittivo e addirittura irrealizzabile in molte applicazioni pratiche. Per questo motivo, il presente lavoro è fortemente orientato a sviluppare un approccio privo di modelli, in cui gli agenti possano apprendere direttamente dalle interazioni con l'ambiente, ottenendo così un controllo dinamico della copertura.

Il Reinforcement Learning (RL) si presenta come una soluzione promettente per affrontare il problema del controllo di copertura dinamico. L'RL è un processo decisionale sequenziale in cui un agente interagisce con l'ambiente e apprende da esso, sviluppando politiche ottimali *senza la necessità di modelli predittivi espliciti.*
Tuttavia, l'RL tradizionale è principalmente limitato a spazi di azione discreti. Per affrontare problemi con spazi di azione continui o dati ad alta dimensione, il Deep Reinforcement Learning (DRL) integra reti neurali, offrendo soluzioni più efficaci. Ad esempio, i robot possono apprendere politiche di controllo direttamente dai dati delle telecamere piuttosto che da controller ingegnerizzati manualmente o da caratteristiche a bassa dimensione degli stati del robot. Recen-

temente, il focus della ricerca si è spostato verso il Multi-Agent Deep Reinforcement Learning (MADRL), poiché più agenti sono in grado di affrontare compiti complessi in modo più efficiente rispetto a un singolo agente. Il MADRL trova applicazioni in vari ambiti, tra cui giochi online multiplayer, robot cooperativi, controllo del traffico e sistemi di controllo dell'energia. Anche il controllo di copertura dinamico ha beneficiato dell'applicazione di tecniche MADRL, come dimostrato in vari studi.
Nel contesto di questo progetto, l'implementazione di algoritmi basati su DRL e MADRL rappresenta una possibile espansione futura. Questi approcci potrebbero migliorare ulteriormente le prestazioni del sistema multi-agente, consentendo agli agenti di apprendere politiche di controllo più sofisticate e coordinarsi in modo più efficace.

In questo lavoro, ci concentriamo sull'implementazione di un algoritmo di controllo di copertura basato sull'ascesa del gradiente *(Gradient Ascent Algorithm)*, una tecnica di Reinforcement Learning (RL) che permette agli agenti di apprendere politiche di controllo ottimali attraverso l'interazione con l'ambiente. Questo approccio è particolarmente adatto per scenari in cui non si dispone di un modello completo dell'ambiente o dove le condizioni cambiano rapidamente nel tempo.
L'algoritmo di ascesa del gradiente è stato scelto per la sua capacità di ottimizzare direttamente una funzione obiettivo, in questo caso l'indice di copertura totale, che quantifica la qualità della copertura fornita dagli agenti sui targets. Questo metodo è stato esteso per considerare le limitazioni pratiche, come le capacità di rilevamento e comunicazione limitate degli agenti, nonché la necessità di mantenere una copertura sufficiente su tutti i targets durante l'intera durata della missione.

Il presente lavoro mira a sviluppare e implementare un algoritmo di controllo di copertura dinamico che sfrutti i principi del Reinforcement Learning e dell'ascesa del gradiente. Gli agenti mobili (droni) devono monitorare una serie di targets mobili, garantendo che ogni target sia coperto cumulativamente a un livello desiderato nel tempo. La formulazione matematica del problema, le soluzioni proposte e i dettagli dell'implementazione dell'algoritmo saranno discussi nelle sezioni successive.

1.1 Descrizione del problema

In questo lavoro, un piccolo gruppo di agenti mobili (*UAVs, droni*) con capacità di rilevamento limitate, posti ad un'altezza fissa, ha l'obiettivo di monitorare dinamicamente un'area popolata da un gran numero di punti di interesse (*Targets mobili*). Lo scenario di riferimento potrebbe essere, ad esempio, un'area portuale dove i targets sono rappresentati da imbarcazioni che si muovono continuamente nel tempo.

Figura 1: Possibile scenario di un controllo della copertura dinamico (2).

Il monitoraggio dinamico di un'area di interesse richiede che, istante per istante, i droni forniscano una visione accurata (o almeno sufficiente) di tutti i target mobili presenti. L'obiettivo generale può variare, come ad esempio l'evitare collisioni tra le imbarcazioni.
Per raggiungere questo obiettivo, è fondamentale garantire una *copertura sufficiente* dell'area. Nel contesto di questo esperimento, le traiettorie dei target saranno predefinite e verranno assegnate posizioni iniziali ai droni. **Il compito consiste dunque nello scrivere un algoritmo di copertura per determinare le traiettorie dei droni in modo tale che, al termine delle traiettorie dei target, venga raggiunto un grado sufficiente di copertura dell'area.** (vedi 2).
I droni dovranno quindi "seguire" i target, assicurando una buona visione dell'area di interesse per tutta la durata dell'esperimento. La definizione dettagliata del *grado di copertura sufficiente* verrà fornita nella sezione 1.3 della tesi.

1.2 Formulazione matematica del problema

Consideriamo M punti di interesse (***Targets dinamici***) distribuiti in uno spazio di lavoro bidimensionale W. Le posizioni dei targets sono denotate da $q_j(t) \in \mathbb{R}^2$, con $j \in \{1, \ldots, M\}$. Un gruppo di N ***Agenti mobili, dinamici*** (*Agents*), ad un'altezza fissata, è incaricato di monitorare continuamente questi targets. Le posizioni degli agenti sono denotate da $p_i(t) \in \mathbb{R}^2$, con $i \in \{1, \ldots, N\}$.
Si presume che ogni agente abbia una zona di rilevamento limitata, modellata da un'area discoidale con un certo *raggio di copertura* $r \in \mathbb{R}^+$ centrata su se stesso.
Poiché la qualità del rilevamento generalmente degrada con la distanza dal target, il rilevamento basato sulla distanza dell'agente i sul target j è caratterizzato da una *FUNZIONE DI MISURAZIONE:*

$$E_{ij}(p_i(t), q_j(t)) = \begin{cases} M_P \left(\frac{(l_{ij}(t) - r)^2}{r^4} \right), & \text{se } l_{ij}(t) \leq r^2, \\ 0, & \text{se } l_{ij}(t) > r^2, \end{cases} \tag{1}$$

dove $l_{ij}(t) = \|p_i(t) - q_j(t)\|^2$ è la distanza, all'istante t, tra il drone i e il target j, e $M_P \in \mathbb{R}^+$ è una qualità di rilevamento di picco. Il modello di rilevamento indica che la qualità del rilevamento del target j raggiunge il picco quando l'agente i coincide con esso, e diminuisce monotonamente quando l'agente i si allontana da esso (2).

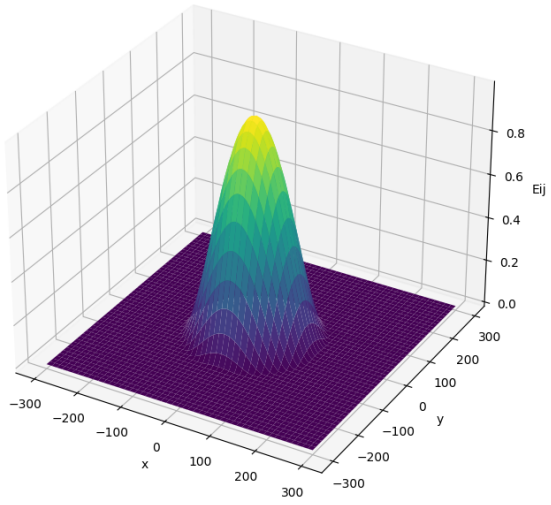

Figura 2: Funzione di misurazione, mp=1, r=150.

1.3 Soluzioni proposte

In questa sottosezione viene definita la procedura matematica che stabilisce il *grado di copertura sufficiente*, menzionato nel paragrafo 1.1.

Il controllo della copertura efficace basato su *RL* (Reinforcement Learning) in questo lavoro ha l'obiettivo di guidare gli agenti mobili affinché garantiscano che ogni target nella regione di interesse sia coperto cumulativamente a un livello desiderato nel tempo (2). Per ogni target j, viene definito un *indice di copertura* $E_j(t)$, che quantifica la qualità della copertura al tempo t fornita dai vari agenti. Ogni agente contribuisce all'indice di copertura in base alla propria vicinanza al target, secondo la seguente funzione di misurazione:

$$E_j(t) = \sum_{i=1}^{N} E_{ij}\left(p_i(t), q_j(t)\right), \quad j \text{ fisso} \tag{2}$$

Come indicato nella sezione 1.2, la funzione di misurazione $E_{ij}(p_i(t), q_j(t))$ varia nell'intervallo $[0, M_P]$. Di conseguenza, l'indice di copertura $E_j(t)$ apparterrà all'intervallo $[0, M_P \cdot N]$.

Successivamente, viene definito un *indice di copertura totale* $E(t)$, espresso come:

$$E(t) = \sum_{j=1}^{M} \sigma(E_j(t)) \tag{3}$$

dove:

$$\sigma(E_j(t)) = \frac{\tanh(E_j(t) - E^*) + 1}{2} \tag{4}$$

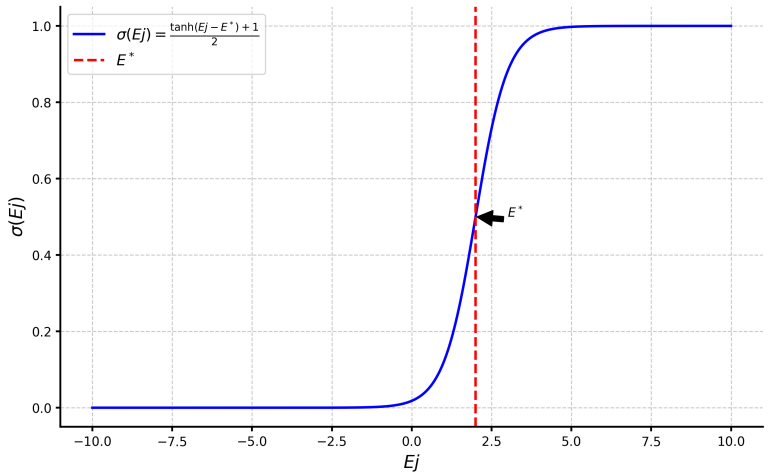

Figura 3: Funzione sigmoidale, $E^* = 2$

La definizione di queste funzioni e indici di copertura è fondamentale per raggiungere l'obiettivo del problema. La soglia E^* rappresenta un *lower bound index (LB)*, ovvero ogni indice di copertura di un target generico j al tempo t ($E_j(t)$) deve essere maggiore di questa soglia per considerare "accettabile" la copertura fornita dagli agenti. È importante notare che I PARAMETRI SONO FORTEMENTE INTERDIPENDENTI; infatti, in base a M_P, varia l'intervallo in cui è contenuta la funzione di misurazione $E_{ij}(p_i(t), q_j(t))$ e, di conseguenza, anche gli indici di copertura ($E_j(t)$). È quindi necessario settare i vari parametri caso per caso. Nella simulazione effettuata (vedi 4), sono stati fissati i seguenti parametri:

- $N = 4$. Rappresenta il numero degli agenti.
- $M = 10$. Rappresenta il numero dei target.
- $Mp = 1$. Definisce l'intervallo in cui è contenuta la funzione di misurazione e anche gli indici di copertura.
- $E^* = 2$. Definisce una soglia minima per gli indici di copertura.
- *duration* $= 150$. Durata in secondi delle traiettorie dei target.

Considerando questi parametri numerici, possiamo affermare che la funzione di misurazione varia tra 0 e 1, mentre gli indici di copertura variano tra 0 e 4. Impostare la soglia E^* a 2 implica che, nel complesso, desideriamo che almeno 2 agenti su 4 coprano ciascun target. Per raggiungere il valore 2, è possibile che 2 agenti siano esattamente sopra i target e abbiano quindi copertura massima, oppure che la somma delle funzioni di misurazione complessive sia pari a 2 (ogni agente copre in minima parte quel target, sommando le coperture si raggiunge la soglia).

La funzione sigmoidale è stata definita per tener conto di questo ragionamento sulla soglia. L'indice di copertura totale somma i singoli indici di copertura a cui è applicata la sigmoide, in modo che, durante il calcolo del gradiente (vedi 2), tutti i contributi degli agenti siano considerati.

Durante la simulazione, verranno generate delle posizioni iniziali degli agenti tali da avere una copertura leggermente inferiore al sufficiente nella situazione iniziale. L'obiettivo sarà quello di sviluppare un algoritmo di copertura dinamica (che verrà analizzato nella sezione 2) per creare le traiettorie degli agenti e indicare loro come muoversi, al fine di ottenere un *indice di copertura totale* finale significativamente maggiore rispetto a quello iniziale $E(\text{duration} - 1) > E(0)$.
Può succedere, come verrà analizzato in modo esaustivo nella sezione 4, che alcuni indici di copertura singoli al tempo finale $E_{ij}(\text{duration}-1)$ siano azzerati o prossimi allo zero. Questo non rappresenta un problema, ma è un comportamento previsto dell'algoritmo di ascesa del gradiente (trattato nella sezione 2), che punta a massimizzare l'indice di copertura totale. Di conseguenza, alcuni target potrebbero non essere coperti nell'istante finale; tuttavia, ciò avvantaggia molti altri target che verranno coperti in maniera ottimale, contribuendo così ad aumentare l'indice totale.

2 Algoritmi

Come anticipato nella sezione 1, l'obiettivo di questo progetto è sviluppare un algoritmo di copertura dinamico (**dynamic coverage algorithm**) per determinare le traiettorie degli agenti, al fine di massimizzare l'indice di copertura totale E.

Gli algoritmi proposti si basano tutti sul **metodo di ascesa del gradiente** come fondamento implementativo. Un algoritmo basato sull'ascesa del gradiente è una tecnica di ottimizzazione iterativa utilizzata per individuare il massimo di una funzione obiettivo. L'idea principale consiste nell'aggiornare iterativamente le variabili nella direzione del gradiente della funzione obiettivo rispetto a queste variabili, al fine di aumentare il valore della funzione (4).

Nelle sezioni seguenti verranno illustrati tre differenti tipi di algoritmi di copertura dinamica basati sull'ascesa del gradiente. Le versioni seconda e terza saranno estese per risolvere problemi di collisione tra gli agenti nell'istante finale.

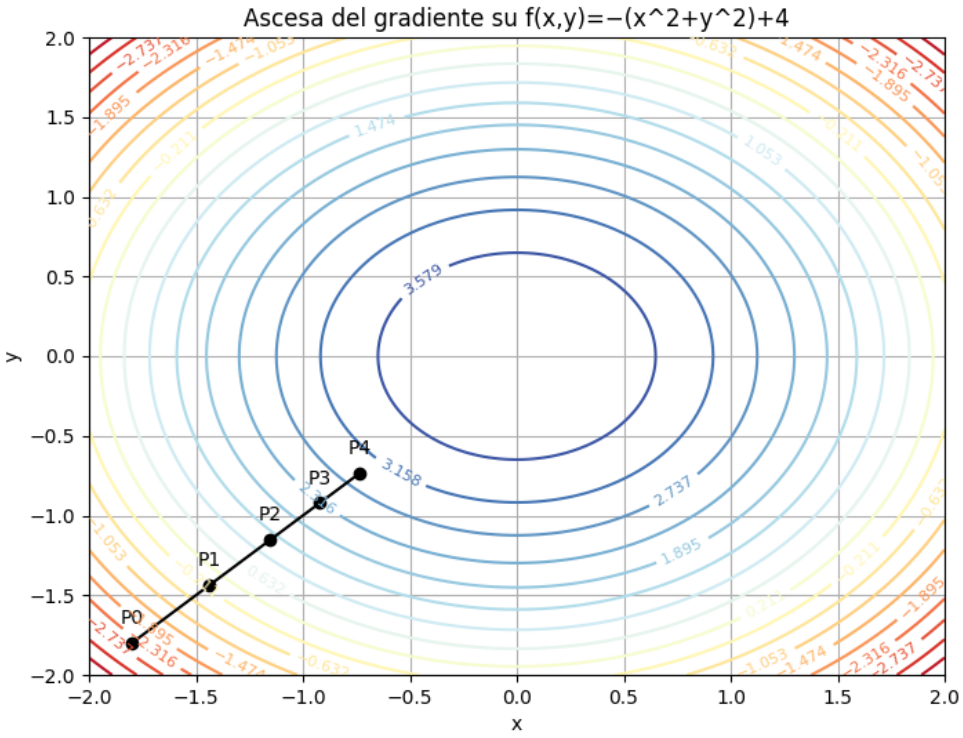

Figura 4: Ascesa del gradiente

2.1 Algoritmo di coverage V1: algoritmo di ascesa del gradiente classico

Nell'ambito del presente progetto di controllo della copertura dinamica, l'algoritmo di ascesa del gradiente viene impiegato per aggiornare le posizioni degli agenti mobili, come droni, determinando le loro traiettorie in modo da massimizzare una funzione obiettivo che rappresenta la copertura totale dell'area di interesse, ovvero l'indice di copertura totale $E(t)$. Gli agenti calcolano il gradiente della funzione di copertura rispetto alle loro posizioni e a quelle dei target al tempo corrente. Successivamente, aggiornano le loro traiettorie seguendo la direzione del gradiente, migliorando iterativamente la copertura complessiva dell'area.

Algorithm 1: V1: Algoritmo di coverage dinamico con ascesa del gradiente

Result: Massimizzare la copertura totale E

for ogni t **do**
  for ogni agente i **do**
    Decidere la direzione in cui muovere l'agente per massimizzare la copertura;
    $$\mu_i = \frac{\partial E}{\partial p_i}\bigg|_{p_i=p_i(t),q_i=q_i(t)} (p_1, \ldots, p_N, q_1, \ldots, q_N)$$
    L'agente si muove con passo di salita ϵ;
    $p_i(t+1) = p_i(t) + \epsilon\mu_i(t)$;
  end
end

In questo contesto, ϵ rappresenta il passo di salita. Questo parametro è cruciale per l'aggiornamento delle posizioni degli agenti mobili ed è direttamente responsabile della velocità con cui gli agenti si muovono lungo la traiettoria determinata dal gradiente della funzione obiettivo.

Come descritto in dettaglio nella sezione 1.1, è possibile che al termine dell'esecuzione dell'algoritmo alcuni indici di copertura individuali $E_{ij}(\text{duration} - 1)$ risultino pari a zero o prossimi allo zero. Questo fenomeno non rappresenta un'anomalia, ma un comportamento previsto dell'algoritmo di ascesa del gradiente, il cui obiettivo principale è la massimizzazione dell'indice di copertura totale. Di conseguenza, alcuni target potrebbero non essere coperti al termine della simulazione.

Un problema emergente in questa situazione è che gli agenti, per massimizzare la copertura, possono convergere verso posizioni finali molto vicine tra loro al tempo $t = \text{duration}$, aumentando il rischio di collisione. Questo è un problema ben noto nel contesto del coverage dinamico, dove la priorità alla copertura totale può portare a densità elevate di agenti in aree specifiche.

Per mitigare il rischio di collisione e migliorare la distribuzione spaziale degli agenti, sono state sviluppate due ulteriori versioni dell'algoritmo. Queste versioni estendono la versione base dell'algoritmo di ascesa del gradiente, introducendo meccanismi che favoriscono una separazione adeguata tra gli agenti, garantendo una copertura efficace senza incorrere in collisioni esagerate.

Nel contesto del controllo di copertura dinamica, il problema della collisione tra agenti è un aspetto cruciale da gestire. Quando più agenti, come droni, operano nello stesso spazio per monitorare una serie di punti di interesse, c'è un rischio significativo che le loro traiettorie si intersechino, portando a collisioni. Questo è particolarmente problematico alla fine dell'esecuzione dell'algoritmo di copertura, quando gli agenti tendono a convergere verso le aree con alta densità di target.
Per affrontare questo problema, diverse strategie di controllo dell'evitamento delle collisioni (*collision avoidance*) sono state proposte in letteratura. Alcuni approcci utilizzano tecniche di ottimizzazione che includono vincoli di distanza minima tra gli agenti, mentre altri implementano algoritmi di controllo distribuito che permettono agli agenti di coordinarsi per evitare le collisioni in tempo reale. Un esempio di queste tecniche è l'uso di funzioni potenziali che generano forze repulsive tra gli agenti quando si avvicinano troppo l'uno all'altro (5) (come fatto in V3: sezione 2.3).

2.2 Algoritmo V2: aggiunta di un moto Browniano

In questa seconda versione dell'algoritmo, le posizioni degli agenti vengono aggiornate non solo utilizzando il gradiente della funzione obiettivo, ma anche includendo una componente stocastica derivante dal **moto browniano**. Questo approccio introduce una variazione casuale nelle traiettorie degli agenti, migliorando la loro capacità di esplorazione e riducendo il rischio di convergenza in posizioni subottimali.

Algorithm 2: V2: Algoritmo di coverage dinamico, aggiunta moto di Browniano

Result: Massimizzare la copertura totale E

for ogni t **do**
  Calcolo della componente Browniana $\eta_i(t)$
  for ogni agente i **do**
    Decidere la direzione in cui muovere l'agente per massimizzare la copertura;

$$\mu_i = \frac{\partial E}{\partial p_i}\bigg|_{p_i=p_i(t),q_i=q_i(t)} (p_1, \ldots, p_N, q_1, \ldots, q_N)$$

    L'agente si muove con passo di salita ϵ;
    $p_i(t+1) = p_i(t) + \epsilon\mu_i(t) + \eta_i(t)$;
  end
end

Il moto browniano, noto anche come movimento aleatorio, è un modello matematico utilizzato per descrivere il comportamento di particelle che si muovono in modo casuale in un fluido. Questo fenomeno prende il nome dal botanico Robert Brown, che per primo osservò il movimento irregolare di particelle microscopiche in sospensione nel 1827. Il moto browniano è caratterizzato da una traiettoria irregolare e imprevedibile, dove ogni passo successivo è indipendente dal precedente e segue una distribuzione di probabilità specifica (6).

Il moto browniano può essere modellato matematicamente come un processo stocastico, spesso descritto da equazioni differenziali stocastiche. In particolare, il modello di Wiener è comunemente utilizzato per rappresentare il moto browniano. Questo modello considera la traiettoria di una particella come una somma di incrementi casuali, ciascuno distribuito normalmente con media zero e varianza proporzionale al tempo. La funzione di densità di probabilità per la posizione della particella a un tempo futuro è quindi una distribuzione gaussiana, il cui centro si sposta con il tempo e la cui larghezza aumenta proporzionalmente alla radice quadrata del tempo (6).

Nel contesto degli algoritmi di ottimizzazione e controllo, il moto browniano può essere utilizzato per introdurre una componente stocastica nei movimenti degli agenti. Questa componente stocastica permette una migliore esplorazione dello spazio delle soluzioni, contribuendo a evitare il problema della convergenza a soluzioni subottimali. In particolare, l'introduzione di una variazione casuale nei movimenti degli agenti consente loro di esplorare aree più ampie dello spazio di ricerca, aumentando così la probabilità di trovare soluzioni globali ottimali. Ad esempio, l'algoritmo di ottimizzazione basato sul moto browniano dei gas (GBMO) combina i principi fisici del moto browniano con strategie di ottimizzazione per migliorare la ricerca di soluzioni ottimali in spazi complessi (7). Altri studi dimostrano che il moto browniano può essere combinato con altri algoritmi di ottimizzazione, come l'algoritmo del volo della farfalla e l'ottimizzazione degli stormi di particelle, per migliorare ulteriormente le prestazioni e la convergenza (8), (9).

La componente browniana $\eta_i(t)$ è definita come:

$$\eta_i(t) = \begin{bmatrix} \text{gauss}_i(t) \cdot \cos(\theta_i(t)) \\ \text{gauss}_i(t) \cdot \sin(\theta_i(t)) \end{bmatrix} \tag{5}$$

dove:

$$\theta_i(t) \sim \mathcal{U}(0, 2\pi) \tag{6}$$

rappresenta la direzione casuale, distribuita uniformemente nell'intervallo $[0, 2\pi]$.

$$\text{gauss}_i(t) \sim \mathcal{N}(\mu, \sigma^2) \tag{7}$$

rappresenta l'intensità del moto browniano, distribuita secondo una distribuzione gaussiana con media μ e varianza σ^2.

Questa componente aggiuntiva introduce una variazione casuale nel movimento degli agenti, permettendo una migliore esplorazione dello spazio e contribuendo a evitare che gli agenti si raggruppino in posizioni subottimali.

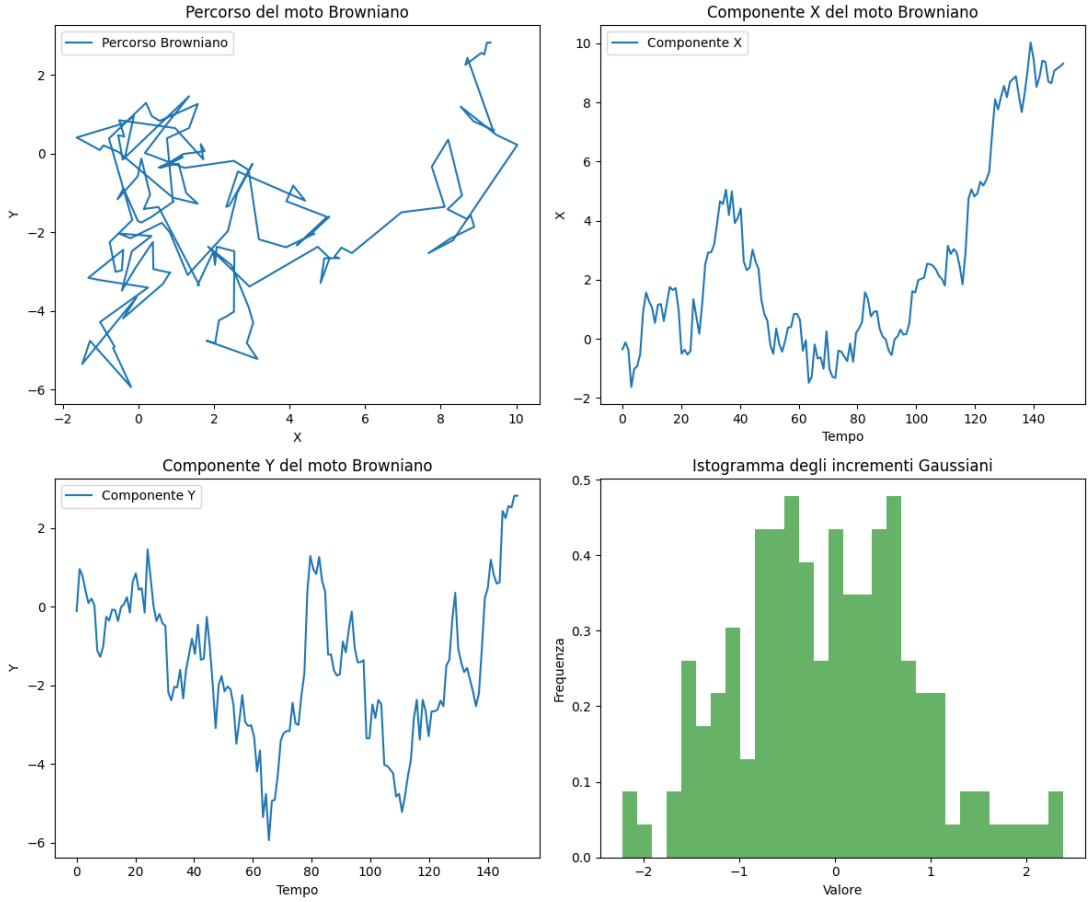

Figura 5: Grafici del moto Browniano

La Figura 5 illustra diverse caratteristiche del moto browniano. Il grafico in alto a sinistra mostra la traiettoria in due dimensioni di una particella che segue un moto browniano, evidenziando la natura casuale e imprevedibile del percorso. I grafici in alto a destra e in basso a sinistra mostrano rispettivamente le componenti X e Y del moto browniano nel tempo, che evidenziano come queste componenti variano casualmente. Infine, l'istogramma in basso a destra rappresenta la distribuzione degli incrementi gaussiani, confermando che questi incrementi seguono una distribuzione normale con media zero.

2.3 Algoritmo V3: aggiunta di un potenziale repulsivo

In questa terza versione dell'algoritmo, le posizioni degli agenti vengono aggiornate non solo utilizzando il gradiente della funzione obiettivo, ma anche includendo una componente di **potenziale repulsivo**. Questo approccio introduce una forza repulsiva tra gli agenti quando la distanza tra di loro è inferiore a una certa soglia δ. L'obiettivo principale di questa componente è evitare collisioni tra gli agenti durante la loro operatività, migliorando la sicurezza e l'efficacia del controllo della copertura.

Algorithm 3: V3: Algoritmo di coverage dinamico, aggiunta di un potenziale repulsivo

Result: Massimizzare la copertura totale E

for ogni t **do**
  for ogni agente i **do**
    Calcolo del potenziale repulsivo
    Decidere la direzione in cui muovere l'agente per massimizzare la copertura;

$$\mu_i = \frac{\partial E}{\partial p_i}\bigg|_{p_i=p_i(t),q_i=q_i(t)} (p_1, \ldots, p_N, q_1, \ldots, q_N)$$

    L'agente si muove con passo di salita ϵ;
    $p_i(t+1) = p_i(t) + \epsilon\mu_i(t) + potRep$;
  end
end

Il potenziale repulsivo totale $potRep$ è la somma delle componenti repulsive calcolate per tutti gli agenti k diversi da i:

$$potRep = \sum_{k \neq i} \eta_i^k(t) \tag{8}$$

dove la componente repulsiva $\eta_i^k(t)$ è calcolata come segue:

$$\eta_i^k(t) = \begin{cases} \left[\frac{p_i(t)-p_k(t)}{d_{ik}(t)}\right] \cdot \left[\frac{\delta - d_{ik}(t)}{d_{ik}(t)}\right] & \text{se } d_{ik}(t) \leq \delta \\ 0 & \text{se } d_{ik}(t) > \delta \end{cases} \tag{9}$$

dove $d_{ik}(t) = \|p_i(t) - p_k(t)\|$ rappresenta la distanza tra l'agente i e l'agente k al tempo t.

Se la distanza è inferiore alla soglia δ, viene calcolata una forza repulsiva. Il primo termine in blu è il versore che indica la direzione di allontanamento dall'agente k. Il secondo termine in viola rappresenta l'intensità della forza repulsiva, che aumenta inversamente proporzionale alla distanza tra gli agenti: più i e k sono vicini, maggiore è la forza repulsiva.
Questo fattore è critico per garantire che la forza repulsiva sia efficace solo quando necessario, senza interferire con il compito principale di massimizzare la copertura.

Viene riportato il grafico dell' intensità della forza repulsiva che mostra come l'intensità della forza repulsiva decresca all'aumentare della distanza tra gli agenti i e k. Quando la distanza è inferiore a una certa soglia δ, la forza repulsiva è alta, mentre diminuisce rapidamente man mano che la distanza aumenta, tendendo a zero quando la distanza è maggiore di δ.

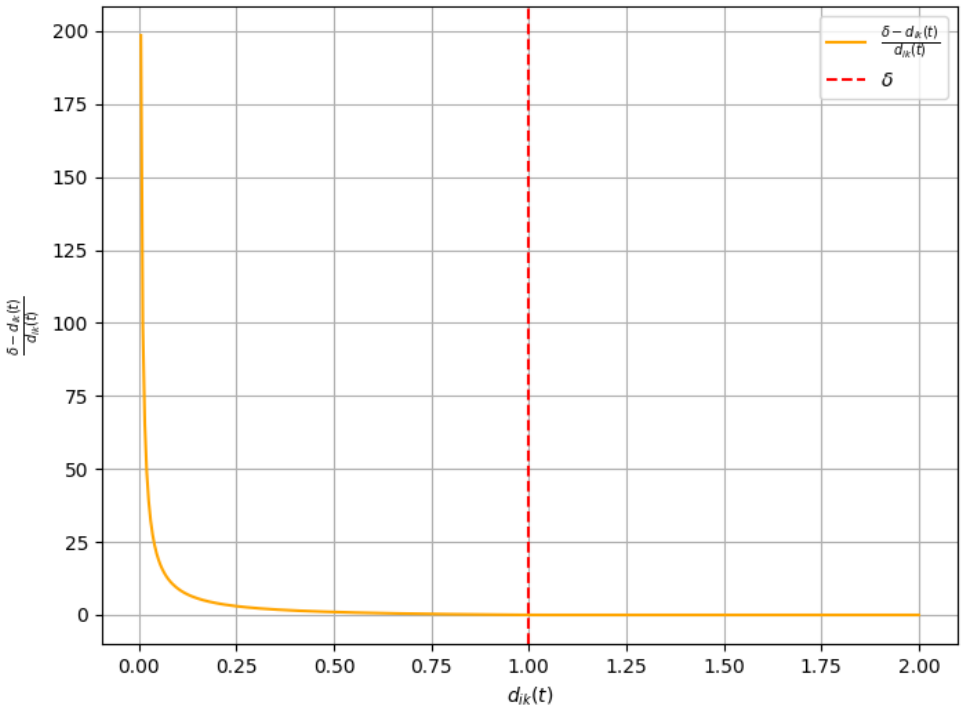

Figura 6: Intensità del potenziale repulsivo

Il potenziale repulsivo è una tecnica ampiamente utilizzata nel campo della robotica e del controllo multi-agente per prevenire le collisioni. La forza repulsiva cresce man mano che gli agenti si avvicinano tra loro, inducendo un movimento che li spinge a distanziarsi. Questo metodo garantisce che gli agenti mantengano una distanza di sicurezza reciproca, riducendo il rischio di collisioni durante il movimento.
Secondo Hao et al., il potenziale repulsivo è fondamentale per il controllo di formazione e la manutenzione della connettività nei sistemi multi-agente. La forza repulsiva viene calcolata in base alla distanza tra gli agenti, aumentando esponenzialmente quando la distanza scende sotto una soglia di sicurezza, generando un'azione di repulsione che allontana gli agenti l'uno dall'altro (10). Questo meccanismo è particolarmente utile nel contesto del coverage dinamico, dove gli agenti devono cooperare per coprire un'area di interesse senza collidere tra loro.

In un altro studio, Jiang et al. descrivono come l'utilizzo di potenziali repulsivi possa migliorare l'efficacia della copertura dinamica, specialmente in scenari complessi con molti agenti che operano in prossimità. I potenziali repulsivi permettono agli agenti di distribuire uniformemente lo spazio di copertura, ottimizzando l'efficienza e riducendo i rischi di collisione (11).
Inoltre, Nagy et al. hanno condotto uno studio sul comportamento dei sistemi multi-agente mobili durante la costruzione di campi di potenziale, dimostrando che le forze repulsive aiutano a mantenere gli agenti separati e ad evitare collisioni (12). L'integrazione del controllo reattivo e telerobotico nei sistemi robotici multi-agente è stata esplorata da Arkin e Ali, evidenziando come le forze repulsive tra robot permettono una convergenza sicura e coordinata (13). Infine, Mabrouk e McInnes hanno introdotto un modello per simulare il comportamento emergente dei robot a sciame, utilizzando campi di potenziale basati su forze attrattive e repulsive (14).

Questo approccio assicura che ogni agente consideri la presenza degli altri agenti nel suo ambiente e regoli il proprio movimento di conseguenza, evitando collisioni e mantenendo una copertura efficace dell'area di interesse.

3 Implementazione degli algoritmi

L'intero progetto di tesi è stato sviluppato utilizzando il linguaggio di programmazione Python. In questa sezione, verranno illustrati SOLO I PUNTI CRUCIALI DELLA LOGICA DEL PROGRAMMA, partendo dalle operazioni preliminari fino ad arrivare alla reale implementazione degli algoritmi di coverage discussi nella sezione 2.
Per l'analisi del codice completo è possibile fare riferimento a (Source code).
Il codice è strutturato come mostrato nelle figure 7 e 8.

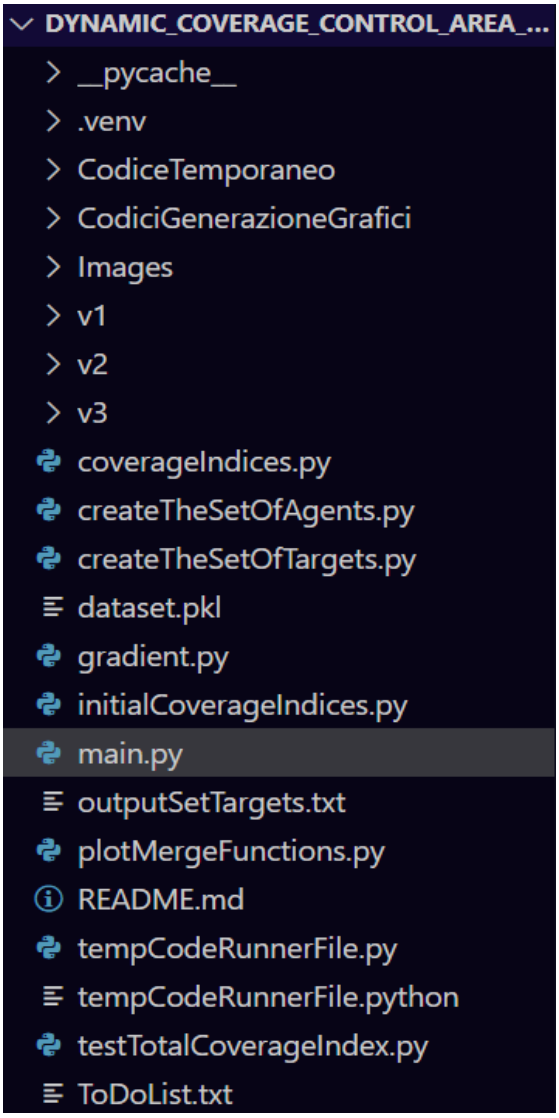

Figura 7: Organizzazione del codice

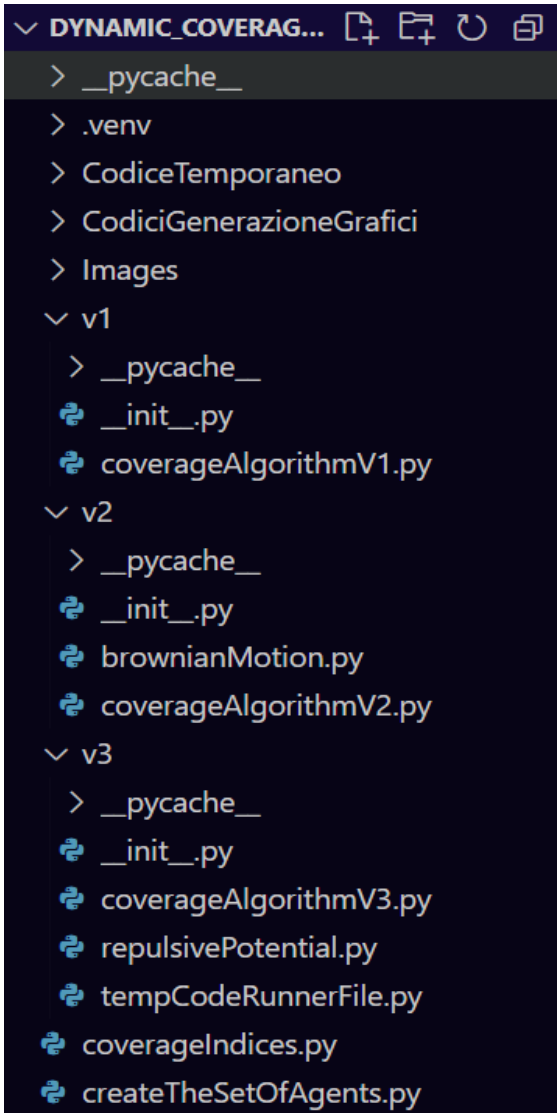

Figura 8: Zoom nelle cartelle degli algoritmi

3.1 Operazioni preliminari

L'unico dato in input di questo progetto è un dataset, salvato in un file *.pickle*. Come illustrato in figura 9, il dataset contiene le traiettorie dei target. Una volta caricato, il dataset (50 traiettorie in una regione di 800 x 800) si presenta nella seguente struttura:

- `np.shape(dataset[i][0])=(351, 4)`
- `np.shape(dataset[i][1])=(351, 2)`

dove i è l'indice della traiettoria e 0,1 indica stato vero (0) con 4 componenti (posizione x, posizione y, velocità x, velocità y) e misure (1) con 2 componenti della posizione, in tutto ciò *l'intervallo di campionamento è di 1 secondo*. Quindi il dataset è una lista di tuple, ognuna di esse contiene due array numpy, uno per rappresentare lo stato vero della traiettoria e l'altro per le **misurazioni** (che rappresentano i dati effettivi delle coordinate della traiettoria considerando lo stato vero + un rumore di misura additivo).

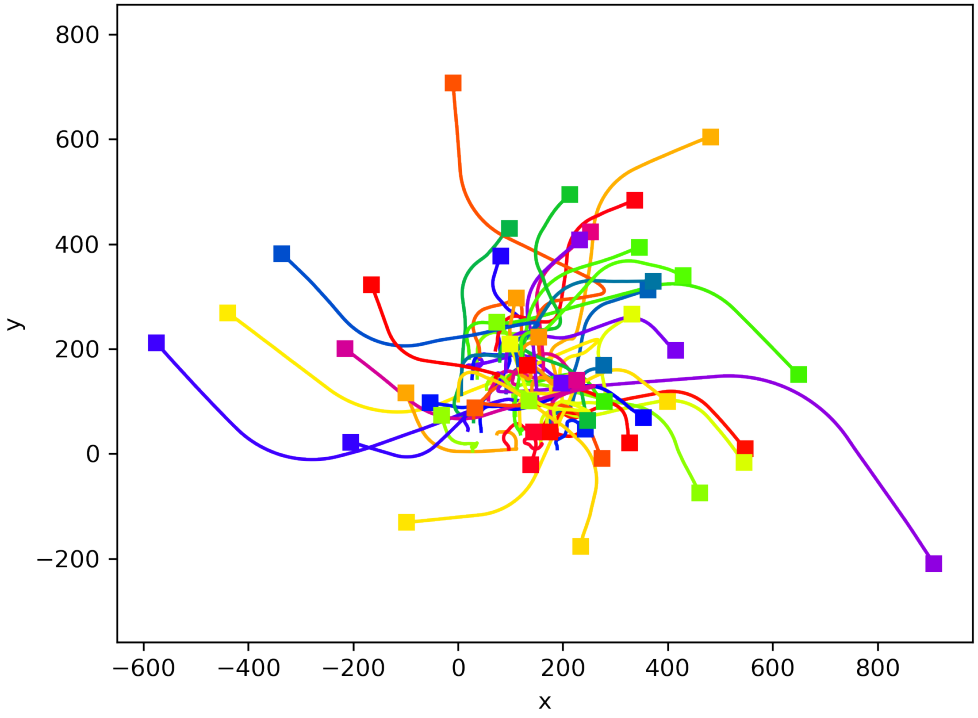

Figura 9: Dataset traiettorie dei target

Per simulare il progetto, ad ogni esecuzione del programma vengono selezionate casualmente 10 traiettorie (*numTargets*) della durata di 150 secondi (*duration*). Maggiori dettagli sul settaggio

dei parametri sono disponibili nella sezione 4. Di seguito è riportato il cuore della funzione responsabile della creazione del dataset di traiettorie dei target utilizzate. È importante notare che viene estratta solo la parte delle **misurazioni**, che è quella di nostro interesse. Ovviamente prima di chiamare la funzione, nel main, il dataset è stato estratto dal file pickle (vedi (15)).

Snippet 3.1: Metodo "buildTheSet()" di *createTheSetOfTargets*

```
def buildTheSet(numTrajectories: int, inputDataset: list, duration:
    int):
    selectedTrajectories = []
    try:
        for _ in range(numTrajectories):
            random_trajectory_index = np.random.choice(len(
                inputDataset))
            # Ottieni la traiettoria corrispondente all'indice
                casuale
            random_trajectory = inputDataset[
                random_trajectory_index]
            # Tronca la traiettoria ai primi "duration" secondi
            truncated_real_state = random_trajectory[0][:duration]
            truncated_measurement = random_trajectory[1][:duration]
            # Aggiungi la traiettoria troncata alla lista delle
                traiettorie selezionate
            selectedTrajectories.append((truncated_real_state,
                truncated_measurement))
    except Exception as e:
        print("Errore durante la creazione dello scenario:", e)

    return selectedTrajectories
```

Snippet 3.2: Estrazione delle misure nel *main*

```
# Estrarre solo la parte truncated_measurement
    targets = [truncated_measurement for _, truncated_measurement
        in createdDatasetOfTargets]
    targetTrajectories = np.array(targets)
```

La struttura dell'array *targetTrajectories* sarà dunque [numTargets]x[duration]x[2] dove 2 rappresenta le due coordinate x e y.

Successivamente, è necessario andare a creare delle posizioni iniziali degli agenti, dei quali, per adesso, non è presente nessuna informazione. Attraverso una verifica, eseguita creando e

visualizzando il grafico delle posizioni iniziali dei target, è stato rilevato che queste sono tutte comprese nell'area [0,200]x[0,200] (*initialAreaSize*). uesta informazione consente di posizionare gli agenti inizialmente in modo uniforme all'interno di questa area. Nel progetto sono stati utilizzati 10 target e 4 agenti (*numAgents*), ma l'approccio è valido anche per un numero diverso di agenti (aspetto migliorabile, come discusso nella sezione 5).

Snippet 3.3: Metodo "generateInitialAgentPositions()" di *createTheSetOfAgents*

```
if numAgents == 4:
        initialAgentPositions = [(50, 50), (50, 150), (150, 50),
            (150, 150)]
    else:
        # Calcola la dimensione della griglia
        grid_size = int(np.sqrt(numAgents))
        if grid_size * grid_size < numAgents:
            grid_size += 1
        # Calcola le distanze tra gli agenti per coprire l'area 200
            x200
        x_spacing = (initialAreaSize - 100) / (grid_size - 1)
        y_spacing = (initialAreaSize - 100) / (grid_size - 1)
        initialAgentPositions = []
        # Genera le posizioni degli agenti distribuendoli
            uniformemente
        for i in range(grid_size):
            for j in range(grid_size):
                if len(initialAgentPositions) < numAgents:
                    x = 50 + j * x_spacing
                    y = 50 + i * y_spacing
                    initialAgentPositions.append((x, y))
    return np.array(initialAgentPositions, dtype=float)
```

La struttura dell'array *initialAgentsPositions* sarà dunque [numAgents]x[2].

Come ultima operazione "preliminare", ma non meno importante, è necessario calcolare gli indici di copertura all'istante iniziale $E_j(0)$ per tutti i target j e anche l'indice di copertura totale iniziale $E(0)$.
Questi calcoli sono fondamentali per effettuare i confronti necessari a dimostrare il corretto funzionamento del progetto, ovvero provare che l'indice di copertura totale finale $E(duration-1)$ è significativamente maggiore rispetto a quello iniziale.
Il codice per questi calcoli è contenuto nel modulo *initialCoverageIndices.py* e i vari metodi implementati rispecchiano precisamente ciò che è stato descritto nella formulazione del problema (1.2) e nelle soluzioni proposte (1.3).

3.2 Descrizione del codice (e tabelle con pseudocodice)

Sono state descritte tutte le operazioni preliminari necessarie per impostare il progetto, tra cui l'estrazione delle traiettorie dei target, la creazione delle posizioni iniziali degli agenti e il calcolo degli indici di copertura iniziali. Questi passaggi preparatori sono fondamentali per l'implementazione degli algoritmi di copertura dinamica descritti nella sezione 2.
In questa sezione, verrà illustrata l'effettiva implementazione degli algoritmi di coverage, discutendo gli aspetti più significativi.

3.2.1 Versione 1

In questa sottosezione viene illustrato il **cuore del progetto**: l'algoritmo di copertura dinamica basato sull'ascesa del gradiente. L'implementazione segue fedelmente lo pseudocodice descritto in sezione 2.1, come mostrato nel codice sottostante.

Snippet 3.4: Metodo "coverageAlgorithmV1()" di *v1.coverageAlgorithmV1*

```
1  def coverageAlgorithmV1(targetsTrajectories: list,
       agentsInitialPosition: list, r, mp, lb, h, NUMAGENTS, NUMSECONDS
       , EPSILON):
2      agentsInitialPositions = np.array(agentsInitialPosition)
3
4      agentsTrajectories = np.zeros((NUMSECONDS, NUMAGENTS, 2))
5
6      agentsTrajectories[0] = agentsInitialPositions
7
8      for t in range(NUMSECONDS-1):
9          gradients_t = gradientOfCoverageIndex(targetsTrajectories,
               agentsTrajectories[t], t, r, mp, lb, h)
10
11         for i in range(NUMAGENTS):
12             agentsTrajectories[t+1][i][0] = (agentsTrajectories[t][
                   i][0]) + (EPSILON*gradients_t[i][0])
13             agentsTrajectories[t+1][i][1] = (agentsTrajectories[t][
                   i][1]) + (EPSILON*gradients_t[i][1])
14     return agentsTrajectories
```

Come si può notare, all'algoritmo vengono passati in ingresso vari parametri fondamentali, i quali verranno descritti meglio in sezione 4. Successivamente, viene inizializzato l'array numpy che conterrà le traiettorie degli agenti, con struttura [DURATION]x[NUMAGENTS]x[2].
Ad ogni istante di tempo, corrispondente a un secondo (che rappresenta l'intervallo di campionamento), viene calcolato il gradiente dell'indice di copertura totale $E(t)$ utilizzando il metodo *gradientOfCoverageIndex()*, che calcola anche i singoli indici di copertura. Successivamente,

vengono predette e salvate le future posizioni degli agenti.
In questo modo, alla fine dell'algoritmo, l'array numpy *agentsTrajectories* conterrà le traiettorie complete degli agenti, che seguono i target al fine di massimizzare l'indice di copertura totale.

E' interessante approfondire come è stato implementato il metodo *gradientOfCoverageIndex()*.

Inizialmente veniva utilizzato il metodo alle differenza finite. Il metodo delle differenze finite è una tecnica numerica utilizzata per approssimare le derivate di una funzione, particolarmente utile quando la derivata analitica è difficilmente calcolabile o si lavora con dati discreti. Per calcolare il gradiente di una funzione $f(x)$ in un punto x, si utilizza una piccola perturbazione h. Una formula comune è la differenza finita in avanti:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{10}$$

Dove:

- $f'(x)$ è la derivata approssimata della funzione f nel punto x
- h è un piccolo incremento, chiamato passo, che rappresenta la distanza tra i punti utilizzati per calcolare la differenza finita. Il valore di h deve essere sufficientemente piccolo per garantire una buona approssimazione, ma non troppo piccolo da causare problemi numerici, come la cancellazione numerica (ed errori numerici)

Per funzioni in più variabili, il gradiente ∇f può essere calcolato utilizzando le differenze finite per ogni componente. Ad esempio, per una funzione $f(x_1, x_2, \ldots, x_n)$, il gradiente è un vettore delle derivate parziali:

$$\nabla f \approx \left(\frac{f(x_1 + h, x_2, .., x_n) - f(x_1, x_2, .., x_n)}{h}, \cdots, \frac{f(x_1, .., x_n + h) - f(x_1, \ldots, x_n)}{h} \right) \tag{11}$$

Questa tecnica è ben documentata e utilizzata in vari contesti ingegneristici e scientifici (16; 17).

Il metodo delle differenze finite, purtroppo, si è rivelato insufficiente per il nostro scopo, in quanto ha introdotto numerose problematiche. In particolare, il calcolo del gradiente ha mostrato oscillazioni troppo brusche e anomalie significative. Per superare queste difficoltà, abbiamo adottato *PyTorch*, una libreria open-source per il machine learning.
Il codice della funzione è riportato di seguito; per ulteriori dettagli, si rimanda il lettore alla sezione 3.3.4.

Snippet 3.5: Metodo "gradientOfCoverageIndex()" di *gradient*

```
def gradientOfCoverageIndex(targets, agentsPosition, t, r, mp, lb,
    h):
    agentsPosition_torch = torch.tensor(agentsPosition,
        requires_grad=True, dtype=torch.float32)

    coverageIndices = calculateCoverageIndices(targets,
        agentsPosition_torch, t, r, mp)

    totalCoverageIndex_t = calculateTotalCoverageIndex(
        coverageIndices, lb)

    # Assicuriamoci che totalCoverageIndex_t richieda gradienti
    totalCoverageIndex_t = totalCoverageIndex_t.requires_grad_(True
        )

    totalCoverageIndex_t.backward()

    # Verifica che i gradienti siano calcolati
    if agentsPosition_torch.grad is None:
        raise RuntimeError("I gradienti non sono stati calcolati
            correttamente.")

    gradients_t = agentsPosition_torch.grad.detach().numpy()

    return gradients_t
```

3.2.2 Versione 2

Come descritto ampiamente nella sezione 2.1, per mitigare il rischio di collisione e migliorare la distribuzione spaziale degli agenti, nella seconda versione dell'algoritmo è stato introdotto un moto Browniano. Di seguito è riportata l'implementazione, dove si notano le differenze e le aggiunte rispetto all'algoritmo precedente.

Snippet 3.6: Metodo "coverageAlgorithmV2()" di *v2.coverageAlgorithmV2*

```
def coverageAlgorithmV1(targetsTrajectories: list,
    agentsInitialPosition: list, r, mp, lb, h, NUMAGENTS, NUMSECONDS
    , EPSILON):
    agentsInitialPositions = np.array(agentsInitialPosition)

    agentsTrajectories = np.zeros((NUMSECONDS, NUMAGENTS, 2))

    agentsTrajectories[0] = agentsInitialPositions

    #Creazione delle traiettorie del moto browniano per la durata
        specificata
    m_i_t = creationOfBrownianMotionTrajectories(NUMSECONDS)

    for t in range(NUMSECONDS-1):
        gradients_t = gradientOfCoverageIndex(targetsTrajectories,
            agentsTrajectories[t], t, r, mp, lb, h)

        for i in range(NUMAGENTS):
            agentsTrajectories[t+1][i][0] = (agentsTrajectories[t][
                i][0]) + (EPSILON*gradients_t[i][0]) + m_i_t[t][0]
            agentsTrajectories[t+1][i][1] = (agentsTrajectories[t][
                i][1]) + (EPSILON*gradients_t[i][1]) + m_i_t[t][1]
    return agentsTrajectories
```

Come descritto in sezione 2.2, in questo algoritmo vengono introdotte delle variazioni casuali nelle traiettorie degli agenti secondo un moto Browniano

Snippet 3.7: Metodo "brownianMotionDef()" di *v2.brownianMotion*

```
def brownianMotionDef(t, mu, sigma, n):
    theta_i = uniformDistribution(n)

    gauss_i = normalDistribution(mu, sigma, n)

    x = gauss_i * np.cos(theta_i)
    y = gauss_i * np.sin(theta_i)

    return np.vstack((x, y)).T
```

Il metodo *brownianMotionDef* calcola il moto browniano per un insieme di punti temporali. Questo metodo genera una sequenza di vettori di movimento (coordinate x e y) utilizzando

distribuzioni gaussiana e uniforme per determinare le intensità e le direzioni dei movimenti.

- `Angoli direzionali casuali`: Viene chiamata la funzione *uniformDistribution(n)*, che restituisce n campioni dalla distribuzione uniforme $U(0, 2\pi)$. Questi campioni rappresentano gli angoli direzionali casuali del movimento browniano.
- `Intensità del movimento`: Viene chiamata la funzione *normalDistribution(mu, sigma, n)*, che restituisce n campioni dalla distribuzione gaussiana $N(\mu, \sigma^2)$. Questi campioni rappresentano le intensità (lunghezze) casuali del movimento.
- `Coordinate x e y`: Le coordinate x e y dei vettori di movimento vengono calcolate utilizzando le funzioni trigonometriche cos e sin per determinare rispettivamente le componenti orizzontali e verticali del vettore.
- `Matrice finale`: Le coordinate x e y vengono combinate in una matrice 2D utilizzando *np.vstack((x, y))*, che impila gli array x e y verticalmente. La trasposizione della matrice $(.T)$ assicura che ogni riga rappresenti un vettore $[x, y]$ per ogni punto temporale. Il risultato è una matrice con forma $(n, 2)$, dove n è il numero di campioni.

Il metodo restituisce una matrice [numAgents]x[2] con i vettori di movimento per ogni punto temporale, consentendo la simulazione del moto browniano in un piano bidimensionale.

3.2.3 Versione 3

Un'altra variante dell'algoritmo base, descritta nella sezione 2.3, prevede l'aggiornamento delle posizioni degli agenti utilizzando non solo il gradiente della funzione obiettivo, ma anche una componente di **potenziale repulsivo**. Questa componente aggiuntiva aiuta a prevenire le collisioni tra gli agenti, migliorando la distribuzione spaziale e l'efficacia complessiva del controllo di copertura.

Snippet 3.8: Metodo "coverageAlgorithmV3()" di *v3.coverageAlgorithmV3*

```
1  def coverageAlgorithmV1(targetsTrajectories: list,
       agentsInitialPosition: list, r, mp, lb, h, NUMAGENTS, NUMSECONDS
       , EPSILON, DELTA):
2      agentsInitialPositions = np.array(agentsInitialPosition)
3
4      agentsTrajectories = np.zeros((NUMSECONDS, NUMAGENTS, 2))
5
6      agentsTrajectories[0] = agentsInitialPositions
7
8      for t in range(NUMSECONDS-1):
9          gradients_t = gradientOfCoverageIndex(targetsTrajectories,
               agentsTrajectories[t], t, r, mp, lb, h)
10
```

```
        for i in range(NUMAGENTS):
            potRep = calculateRepulsivePotential(i,
                agentsTrajectories[t], DELTA)
            agentsTrajectories[t+1][i][0] = (agentsTrajectories[t][
                i][0]) + (EPSILON*gradients_t[i][0]) + potRep[0]
            agentsTrajectories[t+1][i][1] = (agentsTrajectories[t][
                i][1]) + (EPSILON*gradients_t[i][1]) + potRep[1]
    return agentsTrajectories
```

Ogni secondo (intervallo di campionamento), per ciascun agente i, viene calcolato un potenziale repulsivo che contribuisce all'aggiornamento delle loro posizioni. Come descritto nella sezione 2.3, il potenziale repulsivo è determinato nel seguente modo:

Snippet 3.9: Metodo "calculateRepulsivePotential()" di *repulsivePotential*

```
def calculateRepulsivePotential(i, agentsPositions, delta):
    potRep = np.zeros_like(agentsPositions[i])

    for k in range(len(agentsPositions)):
        if k != i:
            diff = agentsPositions[i] - agentsPositions[k]
            d_ik = np.sqrt(diff[0]**2 + diff[1]**2)

            if d_ik <= delta:
                eta_ik = ((agentsPositions[i] - agentsPositions[k])
                    / d_ik) * (delta - d_ik) / d_ik
            else:
                eta_ik = np.zeros_like(agentsPositions[i])
            potRep += eta_ik

    return potRep
```

Il metodo *calculateRepulsivePotential()* calcola il potenziale repulsivo per un agente specifico all'interno di un gruppo di agenti per evitare collisioni. La logica del codice segue perfettamente quanto esposto nell'equazione 9.
Quando il valore ritornato dal metodo è negativo, indica che l'agente deve allontanarsi dagli altri agenti vicini, un comportamento desiderato per evitare collisioni. Quando il potenziale ha componenti negative, la forza repulsiva spinge l'agente nella direzione opposta, aumentando la distanza tra gli agenti.

3.3 Librerie Python utilizzate

Di seguito vengono elencate ed illustrate le librerie Python (degne di nota) utilizzate per raccogliere, elaborare ed analizzare i dati necessari alla scrittura degli algoritmi di copertura. L'implementazione è stata realizzata utilizzando Python nella versione 3.11.3.

3.3.1 NumPy

NumPy è una libreria fondamentale per il calcolo scientifico in Python, che fornisce supporto per array e matrici multidimensionali, oltre a numerose funzioni matematiche (18). Grazie alla sua implementazione in C, NumPy è altamente efficiente e viene utilizzata in una vasta gamma di applicazioni scientifiche e ingegneristiche.
Nel progetto di tesi, NumPy è stato utilizzato per diverse operazioni chiave:

- `Gestione delle traiettorie`:
  - Caricamento e manipolazione del dataset delle traiettorie, organizzate in array multidimensionali.
  - *File di riferimento:* *createTheSetOfTargets.py*.
- `Inizializzazione della posizione degli agenti`:
  - Generazione delle posizioni iniziali degli agenti all'interno di un'area specificata utilizzando `np.random.uniform`.
  - *File di riferimento:* *createTheSetOfAgents.py*.
- `Calcolo degli indici di copertura`:
  - Calcolo efficiente degli indici di copertura iniziali tramite operazioni vettoriali e matriciali.
  - *File di riferimento:* *initialCoverageIndices.py*.
- `Implementazione degli algoritmi di coverage`:
  - Aggiornamento delle posizioni degli agenti, calcolo dei gradienti e delle componenti browniane negli algoritmi di coverage.
  - *File di riferimento:* *coverageAlgorithmV1/2/3.py*
- `Operazioni matematiche avanzate`:
  - Calcolo delle distanze, interpolazioni e altre operazioni matematiche avanzate necessarie per gli algoritmi implementati.

NumPy è stato cruciale per gestire array, calcolare distanze e gradienti, dimostrando la sua importanza e versatilità nel progetto di tesi.

3.3.2 Pickle

Pickle è una libreria standard di Python che permette la serializzazione e deserializzazione di oggetti Python. La serializzazione è il processo di conversione di un oggetto in una sequenza di byte per memorizzarlo in un file o trasferirlo attraverso una rete. La deserializzazione è l'operazione inversa, ossia convertire la sequenza di byte in un oggetto Python. Nel progetto, 'pickle' è stato utilizzato per:

- `Caricamento delle traiettorie dei target`:
  - Utilizzato per caricare un dataset di traiettorie salvato in un file `.pickle`.
  - *File di riferimento:* *`createTheSetOfTargets.py`*.

3.3.3 Matplotlib

Matplotlib è una libreria di plotting per Python che fornisce un framework flessibile e potente per la visualizzazione di dati. Consente di creare grafici statici, animati e interattivi (20). Nel progetto, Matplotlib è stata utilizzata per:

- `Visualizzazione delle traiettorie`:
  - Tracciare e visualizzare le traiettorie dei target e degli agenti durante la simulazione.
  - *File di riferimento:* *`coverageAlgorithmV1/2/3.py`*.
- `Grafici degli indici di copertura`:
  - Creare grafici che mostrano l'evoluzione degli indici di copertura nel tempo, facilitando l'analisi dei risultati.
  - *File di riferimento:* *`coverageIndices.py`*.
- `Grafici utili alla relazione`:
  - Generare tutti i grafici illustrati nella relazione per migliorarne la comprensione e l'impatto visivo.
  - *File di riferimento:* *`CodiciGenerazioneGrafici`*.

3.3.4 Pytorch

PyTorch è una libreria open-source per il machine learning sviluppata da Facebook's AI Research lab. È utilizzata principalmente per applicazioni di deep learning e reti neurali, grazie alla sua flessibilità, velocità di esecuzione e supporto per il calcolo su GPU. PyTorch offre un'interfaccia intuitiva per la creazione di modelli complessi e fornisce potenti strumenti per il calcolo automatico del gradiente, rendendola una scelta ideale per la ricerca e lo sviluppo in ambito di intelligenza artificiale.

PyTorch è particolarmente potente per il *calcolo del gradiente*, che è essenziale per il training dei modelli di machine learning e deep learning. Ecco come PyTorch facilita questo processo:

- `Autograd`: PyTorch include una libreria di differenziazione automatica chiamata Autograd, che traccia tutte le operazioni che coinvolgono i tensori e costruisce dinamicamente un grafo computazionale. Ogni nodo del grafo rappresenta una variabile tensoriale e i bordi rappresentano le operazioni che producono nuovi tensori.
- `Backward Propagation`: Una volta costruito il grafo computazionale, PyTorch può calcolare automaticamente i gradienti rispetto a qualsiasi variabile utilizzando il metodo di backpropagation. Chiamando `.backward()` su un tensore, PyTorch calcola il gradiente di quel tensore rispetto a tutti i tensori che hanno contribuito alla sua creazione.
- `Efficienza`: PyTorch è ottimizzato per sfruttare le capacità delle GPU, rendendo il calcolo dei gradienti estremamente efficiente, cruciale per l'allenamento di modelli di deep learning su grandi dataset.
- `Supporto per le Funzioni Custom`: Gli utenti possono definire nuove funzioni custom e PyTorch può automaticamente calcolarne i gradienti, utile per implementare nuovi algoritmi di machine learning che richiedono operazioni personalizzate.
- `Interfaccia Intuitiva`: PyTorch offre un'interfaccia intuitiva e Pythonica che rende semplice scrivere e comprendere i modelli, facilitando il processo di calcolo del gradiente e il debug.

In sintesi, PyTorch semplifica il calcolo del gradiente attraverso la sua libreria Autograd, permettendo una definizione dinamica e efficiente dei grafi computazionali, supportata da accelerazione hardware. Questa funzionalità è fondamentale per l'allenamento e l'ottimizzazione dei modelli di machine learning e deep learning (vedi (21)).

Nel contesto del progetto, PyTorch è stato utilizzato principalmente per calcolare i gradienti, migliorando l'efficienza e la precisione rispetto al metodo delle differenze finite. Di seguito vengono descritti i vari usi specifici di PyTorch nel codice.

- `Calcolo Automatico del Gradiente`:
  - PyTorch è stato utilizzato per calcolare i gradienti delle funzioni obiettivo in modo efficiente e preciso.
  - La funzione `gradientOfCoverageIndex` utilizza PyTorch per derivare i gradienti, migliorando la precisione e riducendo le oscillazioni rispetto alle differenze finite.
  - *File di riferimento:* ***`gradient.py`***.
- `Definizione e Manipolazione dei Tensors`:
  - PyTorch permette la creazione e manipolazione di tensors, che sono l'equivalente dei arrays in NumPy, ma ottimizzati per il calcolo su GPU.
  - Utilizzato per rappresentare e gestire le traiettorie e le posizioni degli agenti e dei target durante l'esecuzione degli algoritmi.

- *File di riferimento:* `coverageIndices.py`.

- `Integrazione con Altri Strumenti`:
  - PyTorch si integra facilmente con altre librerie utilizzate nel progetto, come NumPy e Matplotlib, permettendo un workflow fluido per l'analisi e la visualizzazione dei dati.
  - Questa integrazione ha facilitato la transizione dei dati tra le varie fasi di calcolo, visualizzazione e analisi.

PyTorch si è rivelato fondamentale per l'implementazione efficiente e precisa degli algoritmi di copertura dinamica, dimostrando la sua versatilità e potenza nel contesto del machine learning e dell'intelligenza artificiale.

Vediamo come Pytorch è stata utilizzata nel modulo *gradient.py* per calcolare il gradiente dell'indice di copertura totale:

Snippet 3.10: Metodo "gradientOfCoverageIndex()" di *gradient*

```
def gradientOfCoverageIndex(targets, agentsPosition, t, r, mp, lb,
    h):
    # Conversione della posizione degli agenti in un tensore
        PyTorch e abilitazione del calcolo dei gradienti
    agentsPosition_torch = torch.tensor(agentsPosition,
        requires_grad=True, dtype=torch.float32)
    # Calcolo degli indici di copertura dei target
    coverageIndices = calculateCoverageIndices(targets,
        agentsPosition_torch, t, r, mp)
    # Calcolo dell'indice di copertura totale
    totalCoverageIndex_t = calculateTotalCoverageIndex(
        coverageIndices, lb)
    # Assicurarsi che l'indice di copertura totale richieda
        gradienti
    totalCoverageIndex_t = totalCoverageIndex_t.requires_grad_(True
        )
    # Esecuzione della backpropagation per calcolare i gradienti
    totalCoverageIndex_t.backward()
    # Verifica se i gradienti sono stati calcolati correttamente
    if agentsPosition_torch.grad is None:
        raise RuntimeError("Gradienti calcolati erroneamente")
    # Detach i gradienti dal grafo computazionale di PyTorch e
        convertirli in un array NumPy
    gradients_t = agentsPosition_torch.grad.detach().numpy()
    return gradients_t
```

4 Simulazione e discussione dei risultati

fare sotto indice

5 Conclusioni

Riferimenti bibliografici

[1] J. Cortes, S. Martinez, T. Karatas and F. Bullo, "Coverage control for mobile sensing networks," in IEEE Transactions on Robotics and Automation, vol. 20, no. 2, pp. 243-255, April 2004, doi: 10.1109/TRA.2004.824698. keywords: Sensor phenomena and characterization;Remotely operated vehicles;Mobile robots;Temperature sensors;Biosensors;Animals;Optimization methods;Partitioning algorithms;Prototypes;Infrared sensors, https://ieeexplore.ieee.org/document/1284411

[2] S. Meng and Z. Kan, "Deep Reinforcement Learning-Based Effective Coverage Control With Connectivity Constraints," in IEEE Control Systems Letters, vol. 6, pp. 283-288, 2022, doi: 10.1109/LCSYS.2021.3070850. keywords: Sensors;Monitoring;Dynamics;Task analysis;Reinforcement learning;Vehicle dynamics;Training;Coverage control;deep reinforcement learning;multi-agent systems, https://ieeexplore.ieee.org/document/9395182

[3] Hübel, Nico. (2008). Coverage Control with Information Decay in Dynamic Environments. IFAC Proceedings Volumes. 41. 4180-4185. 10.3182/20080706-5-KR-1001.00703. https://www.researchgate.net/publication/254995455_Coverage_Control_with_Information_Decay_in_Dynamic_Environments

[4] Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press. https://www.deeplearningbook.org/

[5] Stipanović, D.M., Hokayem, P.F., Spong, M.W., & Šiljak, D.D. (2007). Cooperative avoidance control for multi-agent systems. *Journal of Dynamic Systems, Measurement, and Control*, 129, 699-707. doi:10.1115/1.2764502. https://asmedigitalcollection.asme.org/dynamicsystems/article-abstract/129/5/699/446752/Cooperative-Avoidance-Control-for-Multiagent

[6] E. Nelson, *Dynamical Theories of Brownian Motion*, Princeton University Press, 1967. https://web.math.princeton.edu/~nelson/books/bmotion.pdf

[7] M. Abdechiri, M. R. Meybodi, H. Bahrami, *Gases Brownian motion optimization: an algorithm for optimization (GBMO)*, Applied Soft Computing, 2013. https://www.sciencedirect.com/science/article/pii/S1568494612001834

[8] K. Sowjanya, S. K. Injeti, *Investigation of butterfly optimization and gases Brownian motion optimization algorithms for optimal multilevel image thresholding*, Expert Systems with Applications, 2021. https://www.sciencedirect.com/science/article/pii/S095741742100717X

[9] Q. Xiaohong, Q. Xiaohui, *An evolutionary particle swarm optimizer based on fractal Brownian motion*, Journal of Computational Intelligence, 2012. https://www.ingentaconnect.com/contentone/asp/jcies/2012/00000001/00000001/art00023

[10] H. Hao, et al., *Potential Field Method for Multi-Agent Formation Control with Collision Avoidance*, MDPI, 2020.
https://www.mdpi.com/2076-3417/10/20/7259

[11] J. Jiang, et al., *Enhanced Dynamic Coverage Using Repulsive Potentials in Multi-Agent Systems*, MDPI, 2021.
https://www.mdpi.com/2079-9292/10/8/950

[12] I. Nagy, *Behaviour Study of a Multi-Agent Mobile Robot System during Potential Field Building*, Acta Polytechnica Hungarica, 2009.
https://www.researchgate.net/publication/45087275_Behaviour_Study_of_a_Multi-Agent_Mobile_Robot_System_during_Potential_Field_Building

[13] R. Arkin, K. Ali, *Integration of Reactive and Telerobotic Control in Multi-Agent Robotic Systems*, Proc. Simulation of Adaptive Behavior, 1994.
https://www-robotics.jpl.nasa.gov/media/documents/sab94.pdf

[14] M.H. Mabrouk, C.R. McInnes, *Social Potential Model to Simulate Emergent Behaviour for Swarm Robots*, 13th International Conference on Climbing and Walking Robots, 2008.
https://strathprints.strath.ac.uk/7528/6/strathprints007528.pdf

[15] https://docs.python.it/html/lib/module-pickle.html

[16] Joe D. Hoffman, *Numerical Methods for Engineers and Scientists*, 2nd Edition, CRC Press, 2001.
https://www.crcpress.com/Numerical-Methods-for-Engineers-and-Scientists-Second-Edition/Hoffman/p/book/9780824704438

[17] James F. Epperson, *An Introduction to Numerical Methods and Analysis*, 2nd Edition, Wiley, 2013.
https://www.wiley.com/en-us/An+Introduction+to+Numerical+Methods+and+Analysis%2C+2nd+Edition-p-9781118367599

[18] NumPy Developers, *NumPy: The fundamental package for scientific computing with Python*, https://numpy.org/doc/stable/.

[19] Python Software Foundation, *pickle - Python object serialization*, https://docs.python.org/3/library/pickle.html.

[20] John D. Hunter, *Matplotlib: A 2D Graphics Environment*, 2007. https://matplotlib.org/stable/contents.html

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, Adam Lerer. *Automatic differentiation in PyTorch*, 2017. https://pytorch.org.

Ringraziamenti

Qui puoi scrivere i tuoi ringraziamenti.