



UNIVERSITÀ DEGLI STUDI DI TORINO

Dipartimento di informatica

Corso di laurea triennale in informatica

Progettazione ed implementazione di algoritmi di Formation Control per reti di agenti mobili

Primo relatore
Torta Gianluca

Candidato
Magrì Daniela

Secondo relatore
Audrito Giorgio

Correlatore
Bortoluzzi Daniele

Anno Accademico 2022/2023

Ciascuno di noi ha un potenziale illimitato, è solo una questione di quanto lontano sei disposto a spingerti.

- Daniela Magrì

ABSTRACT

Questa tesi esplora due approcci di **Formation Control** in reti di agenti mobili: **Circular Position** e **Circular Force**. Il loro scopo è quello di creare una formazione circolare di agenti *slave* attorno ad un agente *master*.

Nel primo, ogni agente slave calcola il proprio vettore velocità basato su un algoritmo di posizionamento circolare, mentre nel secondo, la formazione è ottenuta attraverso forze elastiche. Entrambi gli approcci sono approfonditamente analizzati, evidenziando vantaggi e svantaggi. Una comparazione dettagliata rivela le sfide e le opportunità di ciascun metodo, contribuendo a una migliore comprensione delle dinamiche di coordinamento in reti di agenti mobili.

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

2 novembre 2023

Indice

1	Introduzione	2
1.1	Navigazione autonoma di UV in scenari agricoli	2
1.2	Schemi cooperativi UV	3
1.3	Algoritmi di Formation Control ottimali	4
2	Aggregate Programming	6
2.1	Field Calculus Contracts	7
2.2	Resilient coordination operators	8
2.3	Developer APIs	9
3	FCPP	11
3.1	Architettura software	12
3.2	Componenti	12
3.3	Simulatore	14
3.4	Come vengono espresse le funzioni aggregate	15
3.4.1	Linguaggio di programmazione	16
3.4.2	Espressioni Aggregate	16
3.4.3	Funzioni aggregate built-in	17
3.5	Esercizi preparatori	18
3.5.1	Livello facile	18
3.5.2	Livello medio	21
3.5.3	Livello difficile	22
4	Circular Forces	26
4.1	Introduzione	26
4.2	Problema	27
4.3	Configurazione	28
4.4	Descrizione dell'algoritmo	30
4.4.1	Inizializzazione	31
4.4.2	Applicazione delle forze	33
4.4.3	Calcolo del margine d'errore	35
4.5	Analisi	40
4.5.1	Test <i>Nominal</i>	40
4.5.2	Test <i>Slow Slaves Always Fast Master</i>	45
4.5.3	Test <i>Large Radius</i>	50
4.5.4	Test <i>Busy Formation</i>	52
4.6	Valutazione conclusiva	55
5	Circular Position	57
5.1	Introduzione	57
5.2	Problema	57
5.3	Configurazione	59
5.4	Descrizione dell'algoritmo	61

5.4.1	Inizializzazione	63
5.4.2	Controllo dell'indice	64
5.4.3	Calcolo della posizione esatta	67
5.4.4	Applicazione delle forze	70
5.4.5	Calcolo del margine d'errore	72
5.5	Analisi	73
5.5.1	Test <i>Nominal</i>	73
5.5.2	Test <i>Slow Slaves Always Fast Master</i>	77
5.5.3	Test <i>Large Radius</i>	80
5.5.4	Test <i>Busy Formation</i>	83
5.6	Valutazione Conclusiva	87
6	Analisi Comparativa	89
6.1	Analisi comparativa: Test <i>Nominal</i>	89
6.1.1	Consumi energetici dei dispositivi	89
6.1.2	Valutazione della circonferenza costruita	91
7	Conclusione	93
7.1	Possibili Evoluzioni e Ottimizzazioni	93
7.1.1	Movimento del <i>Master</i>	93
7.1.2	Revisione del sistema di <i>Collision Avoidance</i>	93
7.1.3	Considerazioni sull'analisi svolta	94

Elenco delle figure

1.1	<i>Architetture cooperative</i>	3
2.1	<i>Reti di agenti mobili interconnessi</i>	7
2.2	<i>Livelli aggregati di astrazione della programmazione</i>	7
3.1	<i>Architettura software di FCPP</i>	12
3.2	<i>Presentazione dei componenti FCPP</i>	13
3.3	<i>Simulatore grafico</i>	14
3.4	<i>Presentazione delle simulation features di FCPP</i>	15
3.5	<i>Sintassi delle funzioni aggregate in FCPP</i>	15
3.6	<i>Exercises</i>	18
3.7	<i>Exercise 1</i>	19
3.8	<i>Prototipo di execution stack</i>	20
3.9	<i>Vettori velocità</i>	22
3.10	<i>Exercise 7</i>	24
4.1	<i>Campo vettoriale generato dal metodo dei potenziali artificiali</i>	26
4.2	<i>Legge di Stokes applicata dall'attrito viscoso</i>	31
4.3	<i>Legge di Hooke applicata dalla forza elastica</i>	34
4.4	<i>Circonferenza formata da Circular Forces rispetto ad una formazione teorica</i>	36
4.5	<i>Velocità e forze interagenti in Circular Forces test Nominal</i>	40
4.6	<i>Circonferenza equamente divisa durante la simulazione Nominal con Circular Forces</i>	41
4.7	<i>Errore delle posizioni in Circular Forces test Nominal</i>	43
4.8	<i>Errore delle distanze in Circular Forces test Nominal</i>	44
4.9	<i>Risultato finale del test Nominal con Circular Forces</i>	45
4.10	<i>Velocità e forze interagenti in Circular Forces test Slow Slaves Always Fast Master</i>	46
4.11	<i>Adattabilità di Circular Forces nel test Slow Slaves Always Fast Master</i>	47
4.12	<i>Errore delle distanze in Circular Forces test Slow Slaves Always Fast Master</i>	49
4.13	<i>Velocità e forze interagenti in Circular Forces test Large Radius</i>	50
4.14	<i>Comunicazione al tempo iniziale tra slave in Circular Forces test Large Radius</i>	51
4.15	<i>Comunicazione ad un tempo successivo tra slave in Circular Forces test Large Radius</i>	51
4.16	<i>Errore delle distanze in Circular Forces test Large Radius</i>	52
4.17	<i>Velocità e forze interagenti in Circular Forces test Busy Formation</i>	53
4.18	<i>Errore nella costruzione della circonferenza in Circular Forces test Busy Formation</i>	53
4.19	<i>Errore delle distanze in Circular Forces test Busy Formation</i>	54
4.20	<i>Errore delle distanze tra vicini fisici in Circular Forces test Busy Formation</i>	54

5.1	<i>Circonferenza formata da Circular Position rispetto ad una formazione teorica</i>	68
5.2	<i>Rappresentazione grafica del meccanismo di traslazione della circonferenza su un sistema di assi cartesiani</i>	69
5.3	<i>Concetto di Corona Circolare</i>	70
5.4	<i>Velocità e forze interagenti in Circular Position test Nominal</i>	73
5.5	<i>Fase 1: circonferenza in costruzione durante la simulazione Nominal con Circular Position</i>	74
5.6	<i>Fase 2: circonferenza in costruzione durante la simulazione Nominal con Circular Position</i>	75
5.7	<i>Fase 3: circonferenza in costruzione durante la simulazione Nominal con Circular Position</i>	76
5.8	<i>Errore delle posizioni in Circular Position test Nominal</i>	77
5.9	<i>Risultato finale del test Nominal con Circular Position</i>	77
5.10	<i>Velocità e forze interagenti in Circular Position test Slow Slaves Always Fast Master</i>	78
5.11	<i>Errore delle posizioni in Circular Position test Slow Slaves Always Fast Master</i>	78
5.12	<i>Errore delle distanze in Circular Position test Slow Slaves Always Fast Master</i>	79
5.13	<i>Accelerazioni posizionali di Circular Position nel test Slow Slaves Always Fast Master</i>	79
5.14	<i>Analisi dell'accelerazione angolare aumentata di Circular Position nel test Slow Slaves Always Fast Master</i>	80
5.15	<i>Velocità e forze interagenti in Circular Position test Large Radius</i>	81
5.16	<i>Errore delle posizioni in Circular Position test Large Radius</i>	81
5.17	<i>Errore delle distanze in Circular Position test Large Radius</i>	82
5.18	<i>Fenomeno di dispersione in Circular Position test Large Radius</i>	82
5.19	<i>Risultato finale del test Large Radius con Circular Position</i>	83
5.20	<i>Velocità e forze interagenti in Circular Position test Busy Formation</i>	84
5.21	<i>Diverse implementazioni della regione di attivazione del sistema Collision Avoidance in Circular Position nel test Busy Formation</i>	85
5.22	<i>Forze applicate con un solo livello all'interno della regione di attivazione del sistema Collision Avoidance in Circular Position nel test Busy Formation</i>	85
5.23	<i>Errore delle posizioni in Circular Position test Busy Formation</i>	86
5.24	<i>Errore delle distanze in Circular Position test Busy Formation</i>	86
5.25	<i>Risultato finale del test Busy Formation con Circular Position</i>	87
6.1	<i>Consumo di energia a confronto test Nominal</i>	89
6.2	<i>Errore delle distanze a confronto test Nominal</i>	91

Elenco delle tabelle

3.1	<i>FCPP components</i>	13
4.1	<i>Dati relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato sulle forze applicate nel test Nominal con Circular Forces</i>	41
4.2	<i>Dati relativi alle distanze catturati durante un fenomeno di picco grafico rilevato sulle forze applicate nel test Nominal con Circular Forces</i>	42
4.3	<i>Dati relativi alle distanze catturati durante un fenomeno di minimo grafico sulle forze applicate nel test Slow Slaves Always Fast Master failed con Circular Forces</i>	48
4.4	<i>Dati relativi alle distanze catturati durante un fenomeno di minimo grafico sulle forze applicate nel test Slow Slaves Always Fast Master failed rispetto alla versione solution con Circular Forces</i>	49
4.5	<i>Dati relativi alle distanze catturati in presenza di buchi nella formazione nel test Busy Formation con Circular Forces</i>	54
5.1	<i>Logica del ricalcolo dell'indice</i>	66
5.2	<i>Esempio di applicazione del meccanismo di traslazione della circonferenza</i>	69
5.3	<i>Dati fase 1 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position</i>	74
5.4	<i>Dati fase 2 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position</i>	75
5.5	<i>Dati fase 3 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position</i>	76
5.6	<i>Dati relativi all'errore delle posizioni catturati al tempo massimo di simulazione preso in considerazione nei grafici del test Large Radius con Circular Position</i>	82

Capitolo 1

Introduzione

Nel contesto dell’evoluzione delle tecnologie digitali e dell’automazione, le reti di agenti mobili stanno emergendo come un campo di ricerca con applicazioni in una vasta gamma di settori, tra cui la robotica, la logistica, il monitoraggio ambientale e molto altro ancora. La capacità di coordinare e controllare gruppi di agenti mobili in modo efficiente e cooperativo è essenziale per sfruttare appieno il potenziale di tali sistemi. Questa tesi si concentra sulla progettazione ed implementazione di algoritmi di **Formation Control**, un aspetto fondamentale nell’ambito delle reti di agenti mobili.

La **Formation Control**, si riferisce alla capacità di stabilire e mantenere una disposizione geometrica desiderata tra gli agenti mobili, consentendo loro di lavorare insieme in modo coordinato per raggiungere obiettivi comuni. Questo concetto è utile ad esempio per il monitoraggio aereo con droni o la navigazione di team di robot mobili in spazi con restrizioni. Questa tesi si propone di esplorare in profondità la progettazione, l’implementazione e l’ottimizzazione di algoritmi di **Formation Control** per reti di agenti mobili. Tali algoritmi sono realizzati sfruttando l’*Aggregate Programming* utilizzando il framework *FCPP*, sviluppato presso il Dipartimento di Informatica dell’Università degli Studi di Torino.

Saranno affrontate sfide complesse legate alla comunicazione tra agenti e alla dinamica del sistema. Sarà inoltre data particolare attenzione all’applicazione pratica di tali algoritmi in diversi scenari, valutando le prestazioni in base a metriche rilevanti. Attraverso l’attenta selezione di operatori del *Field Calculus*, è possibile mantenere la coerenza tra le operazioni aggregate e locali di un programma. Ogni comportamento del programma a livello aggregato può essere implementato mediante semplici regole di interazione locali.

1.1 Navigazione autonoma di UV in scenari agricoli

Il Dipartimento di Informatica presso cui si è svolto il lavoro di ricerca oggetto di questa tesi, ha instaurato partnership accademiche con il Dipartimento di Scienze Agrarie, Forestali e Alimentari nello specifico in aree di studio che riguardano reti di agenti mobili.

L’**agricoltura 4.0**[1] comprende un insieme di tecnologie che integrano sensori, sistemi informativi avanzati, macchinari sofisticati e una gestione basata su dati, il tutto con l’obiettivo di massimizzare l’efficienza produttiva, tenendo conto delle variazioni e incertezze che caratterizzano i sistemi agricoli. L’impiego di veicoli autonomi rappresenta una leva significativa per migliorare l’efficienza, consentendo una rapida ed efficace esecuzione di una vasta gamma di attività in campo. In particolare, si aprono ampie possibilità di ottimizzazione attraverso la promozione della cooperazione e dell’azione di **UV** (*Unmanned Vehicles*) al fine di condurre operazioni in campo in maniera precisa ed efficiente dal punto di vista temporale.

I contesti operativi che richiedono l’impiego di sistemi UV includono:

- Terreni pianeggianti occupati da coltivazioni con coperture omogenee (*come campi di grano o di riso*), nei quali è necessario condurre operazioni sopra le colture senza interagire con il terreno;
- Vigneti fortemente inclinati o altri campi che non sono accessibili tramite trattori e attrezzature standard.

L'impiego simultaneo di più veicoli autonomi, non pilotati, con l'obiettivo di eseguire compiti complessi ha dimostrato la sua efficacia in numerose applicazioni, sia in simulazioni che in situazioni pratiche. Questi sistemi hanno dimostrato di offrire prestazioni superiori rispetto ai sistemi monolitici in termini di: **flessibilità, ottimizzazione dei tempi e dei costi operativi, miglioramento della sicurezza e riduzione degli eventi di guasto.**[2]

1.2 Schemi cooperativi UV

Come evidenziato da una serie di casi documentati sia nella letteratura scientifica che nelle applicazioni commerciali[3], quando gruppi di robot sono coinvolti in attività agricole, possono essere composti da agenti omogenei o eterogenei. Riguardo alle strategie di assegnazione delle attività e alle architetture operative, i sistemi **UV** autonomi e collaborativi possono essere categorizzati come segue:

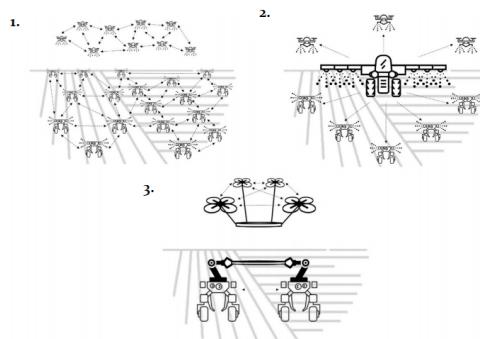


Figura 1.1: *Architetture cooperative*

1. **Architettura peer-to-peer:** le macchine coinvolte presentano tipicamente similitudini o sono identiche, e il compito complessivo viene suddiviso in numerose attività parallele, ciascuna delle quali viene eseguita individualmente da ciascuna macchina;
2. **Architettura master-slave:** una o più macchine robotiche sono governate da un *master*. In questo caso, a ciascun partecipante viene assegnato un compito specifico, che deve essere coordinato in modo adeguato per raggiungere l'obiettivo complessivo;
3. **Architettura team robots:** è richiesta un'azione congiunta di più robot per portare a termine un singolo compito.

Le tipologie di sistemi di veicoli autonomi cooperativi possono essere suddivise anche in base al livello di interazione tra gli agenti coinvolti e alle strategie di controllo implementate. In questa prospettiva, si possono individuare le seguenti categorie:

- **Multi-agent systems:** gli **UV** autonomi fanno parte di un sistema più complesso, in cui sono coinvolti anche altri agenti (ad esempio, computer, sensori in campo, operatori umani, ecc.) che cooperano simultaneamente tra loro;

- **Swarm robots:** un insieme di **UV** autonomi con strategie di controllo più semplici rispetto ad altre categorie, e una elevata capacità di interazione tra loro;
- **Shared-world approach:** Una variante del *multi-agent systems*, contraddistinta dalla possibilità di cooperazioni ritardate, ossia cooperazioni che avvengono in fasi successive. Quando gli agenti operano in fasi successive, possono essere assegnati a missioni completamente diverse, condividendo informazioni strategiche essenziali per il successo delle operazioni, ad esempio informazioni riguardanti l'ambiente in cui stanno agendo.

1.3 Algoritmi di Formation Control ottimali

Per garantire l'autonomia degli **UV**, l'aspetto più importante è assicurare la capacità di generare percorsi ottimali che tengano conto della posizione corrente, del rispetto delle attività della missione, nonché del soddisfacimento dei vincoli ambientali e di sicurezza. I criteri per determinare il percorso ottimale si basano su una o più di queste caratteristiche: **la minimizzazione del rischio di collisioni, l'adempimento continuo degli obiettivi, la massimizzazione della copertura del perimetro e la minimizzazione dei requisiti energetici**. Il percorso ottimale viene svolto da una formazione di dispositivi.

In questo lavoro vi è un dispositivo principale, identificato come ***master***, che segue un percorso che si presuppone sia ottimale e i dispositivi ausiliari, noti come ***slave***, cooperano tra loro per stabilire e mantenere una formazione attorno al dispositivo ***master***.

Questo lavoro di ricerca mira a sviluppare algoritmi di Formation Control ottimali, utilizzando un'architettura ***master-slave*** e un elevata interazione tra i dispositivi sviluppando così uno ***swarm*** di robot.

Capitolo 2

Aggregate Programming

L'aumento progressivo del numero di dispositivi interconnessi implica un incremento dei costi associati alla manutenzione dei sistemi distribuiti.

Ciò crea notevoli sfide nell'implementazione di servizi software su scala globale attraverso l'approccio tradizionale di programmazione individuale per ciascun agente, spingendo la ricerca di soluzioni mirate a migliorare l'autonomia dei sistemi informatici e ridurre la loro complessità.

In questo contesto, la **programmazione aggregata** emerge come un approccio di rilievo: essa si basa sulla composizione funzionale di blocchi di comportamento collettivo riutilizzabili, con l'obiettivo di ottenere in modo efficiente comportamenti complessi e resilienti in reti dinamiche. Questa metodologia consente l'analisi di proprietà di fondamentale importanza, come l'**autostabilizzazione** e l'**indipendenza dalla densità**. Tali proprietà vengono inizialmente studiate su blocchi di base, quali il **broadcast**, la **stima delle distanze** e l'**aggregazione dei dati**, prima di essere trasferite e validate su sistemi più complessi realizzati mediante composizione funzionale.

La programmazione aggregata trova le sue radici concettuali nel *Field Calculus*, che si presenta come un linguaggio di programmazione funzionale minimale progettato per la specifica e la composizione di comportamenti collettivi.

Riassumendo, attraverso l'impiego di costrutti di *Field Calculus* e l'utilizzo delle API fornite dai blocchi predefiniti, la programmazione aggregata potrebbe rivelarsi un catalizzatore fondamentale per liberare appieno il potenziale dell'Internet of Things (IoT). Questo consentirebbe la definizione concisa di servizi distribuiti altamente complessi, consentendo loro di essere incapsulati, modulari e componibili tra di loro.

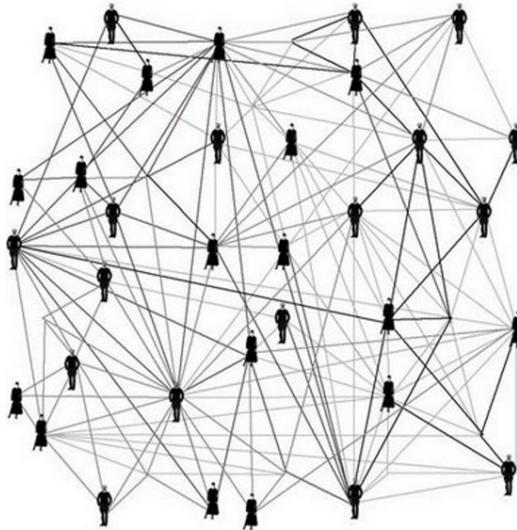


Figura 2.1: Reti di agenti mobili interconnessi

Tramite vari livelli di astrazione, la programmazione aggregata riesce a mitigare la complessità coinvolta nella coordinazione distribuita in ambienti di reti IoT. Vediamo ora più nel dettaglio quali sono e come sono strutturati questi livelli:

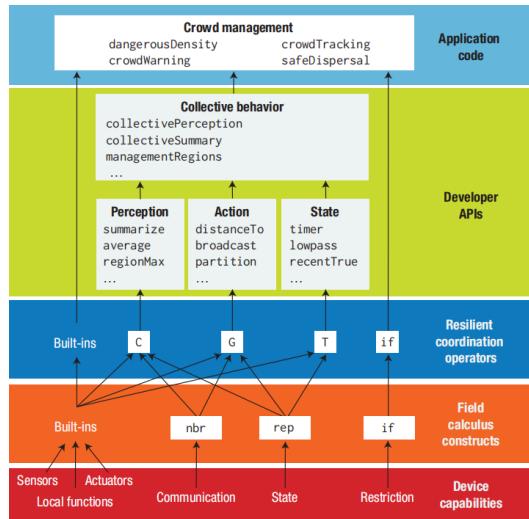


Figura 2.2: Livelli aggregati di astrazione della programmazione

2.1 Field Calculus Constructs

Questo strato, situato come secondo strato nella Figura 2.2 [4], rappresenta il punto in cui le interfacce di programmazione aggregata interagiscono con l'ambiente esterno costituito dall'infrastruttura dei dispositivi e dai servizi software non aggregati, che insieme formano il livello più basso del sistema.

L'astrazione chiave fornita dal concetto di *Field Calculus* è basata sulla nozione di **campo**, una concettualizzazione ispirata a concetti fisici. In questa prospettiva, ogni dispositivo connesso in rete è mappato su un valore locale all'interno di un campo. Nel contesto del

Field Calculus, ogni entità, che sia un'espressione, un valore o una variabile, è trattata come un campo, ad esempio: un insieme di sensori di temperatura contribuisce a creare un campo di valori di temperatura ambiente, gli accelerometri presenti negli smartphone contribuiscono a un campo che rappresenta le direzioni di movimento, e un'applicazione di notifica genera un campo di messaggi visualizzati sui dispositivi mobili... La manipolazione e la creazione dei campi avvengono attraverso l'uso di quattro costrutti:

1. Le funzioni, espresse come **b(e1,...en)**, operano attraverso l'applicazione della funzione b agli argomenti e1...en. Queste funzioni **"built-ins"** rappresentano funzioni matematiche, logiche o algoritmiche prive di stato, nonché sensori o attuatori, e possono includere funzioni definite dall'utente;
2. La clausola **"rep(x ← v) s1;...;sn"** introduce una variabile di stato locale denominata x, la quale viene inizializzata con il valore v e periodicamente aggiornata con il risultato dell'esecuzione delle istruzioni contenute nel suo blocco s1;...;sn. Questa procedura consente di definire un campo che evolve dinamicamente nel corso del tempo;
3. L'operazione **"nbr(s)"** effettua una raccolta di valori più recenti di ciascun dispositivo provenienti da tutti i dispositivi adiacenti, incluso il dispositivo stesso. Alcune funzioni *built-in* operano poi una sintesi di tali mappe: per esempio, la funzione **minHood(m)** determina il valore minimo presente nel field m...;
4. L'istruzione **"if(e)s1;...;sn else s1';...;sm"** suddivide la rete in due regioni: in quella in cui l'espressione è valutata come vera, vengono quindi eseguite le istruzioni s1;...;sn, mentre nell'altra regione, vengono eseguite s1';...;sm'. È fondamentale notare che questa suddivisione comporta la separazione dei rami e impedisce loro di avere interazioni.

2.2 Resilient coordination operators

Il successivo livello di astrazione all'interno del framework di programmazione aggregata introduce la resilienza e identifica una serie di operatori di base generali destinati a essere utilizzati nelle applicazioni di coordinamento resilienti. Questo strato, posizionato al centro nella Figura 1.2, comprende meccanismi di coordinamento che dimostrano proprietà di auto-stabilizzazione, ovvero la capacità di adattarsi reattivamente ai cambiamenti nella struttura della rete o nei valori di input.

Una possibile raccolta di operatori di base comprende tre operatori di coordinamento generalizzati, oltre a "if" e le funzioni *built-in*. I tre operatori sono:

1. L'operazione **G(source, init, metric, accumulate)** rappresenta una operazione di "spreading" che offre una generalizzazione delle operazioni di misurazione delle distanze, trasmissione broadcast e proiezione. Questa operazione svolge due compiti principali: innanzitutto, calcola un campo di distanze minime da una regione di origine specificata (indicata mediante un campo booleano, source) utilizzando una metrica fornita. Successivamente, essa propaga i valori lungo il gradiente di distanza risultante, partendo da un valore iniziale (init) e accumulando ulteriori valori lungo tale gradiente utilizzando una funzione di accumulo (accumulate);
2. L'operazione **C(potential, accumulate, local, null)** si occupa dell'accumulazione di informazioni in direzione della sorgente lungo il gradiente di un campo di potenziale. Inizialmente, partendo da un elemento neutro idempotente (null), il valore locale viene combinato con i valori "in salita" utilizzando una funzione di accumulo commutativa e associativa (accumulate), risultando in un valore cumulativo presso la sorgente;

3. L'operazione **T(initial, floor, decay)** descrive un processo di conto alla rovescia flessibile caratterizzato da un tasso che può variare nel tempo. La funzione "decay" riduce progressivamente il valore iniziale (initial) fino a raggiungere il valore minimo consentito (floor).

2.3 Developer APIs

Le librerie sviluppate mediante l'utilizzo degli operatori di base possono impiegare e combinare tali operatori per costituire un'API (Application Programming Interface) pragmatica e di facile utilizzo. Tali librerie rappresentano il penultimo strato nella struttura illustrata nella Figura 1.2, su cui si basa la scrittura del codice delle applicazioni. Facciamo ora un esempio, molte funzioni di diffusione di azioni e informazioni distribuite possono basarsi su G ad esempio:

Il calcolo che stima la distanza da uno o più dispositivi sorgente designati

```
def distanceTo(source) {
    G(source, 0, () -> {nbrRange},
        (v) -> {v + nbrRange})
}
```

Application code

Le API svolgono un ruolo cruciale nel consentire agli sviluppatori di creare e gestire servizi IoT complessi in ambienti di coordinazione distribuita (come il servizio **"Crowd management"**¹), garantendo allo stesso tempo resilienza e sicurezza. Il loro utilizzo agevola la programmazione aggregata fornendo strumenti per l'implementazione di comportamenti sofisticati all'interno di reti dinamiche.

La programmazione aggregata stratificata offre quindi la prospettiva di un ecosistema software efficiente per lo sviluppo di servizi IoT distribuiti!

¹"**Crowd management**" si riferisce alla gestione e al controllo di grandi folle di persone durante eventi, situazioni o luoghi in cui si prevede una concentrazione elevata di individui. Un esempio pratico di applicazione di tale servizio è **Alchemist simulation of a Crowd Safety Service**, una simulazione in Alchemist di un servizio di sicurezza della folla in esecuzione su 2500 dispositivi.

Capitolo 3

FCPP

Alcune delle attuali implementazioni di *Field Calculus* sono basate sulla Java Virtual Machine (JVM) e soffrono di alcuni problemi di prestazioni, limitando la loro idoneità in scenari in cui le prestazioni sono critiche o le risorse computazionali sono scarsamente disponibili, come nel caso dei sistemi di sensori wireless (WSN) e dell'Internet of Things (IoT). FCPP[5] è stata recentemente proposta come una nuova implementazione di *Field Calculus* ed è stata progettata seguendo un approccio basato su componenti per favorire l'estendibilità in diversi contesti. Utilizzando i template C++, consente ottimizzazioni in fase di compilazione e riduce al minimo l'overhead prestazionale.

FCPP dimostra un notevole miglioramento delle prestazioni rispetto alle implementazioni precedenti mantenendo un alto livello di astrazione.

Attualmente, la libreria FCPP è focalizzata sulla simulazione di sistemi distribuiti ed è già in grado di ottimizzare notevolmente i processi di simulazione, accelerando lo sviluppo di nuovi algoritmi distribuiti. Inoltre, le sue caratteristiche permettono una flessibile estensione per coprire ulteriori scenari applicativi, specialmente quelli per i quali gli strumenti esistenti risultano inadeguati. In particolare, FCPP offre vantaggi significativi in due ambiti specifici:

1. Nelle implementazioni su sistemi basati su microcontroller, spesso caratterizzati da limitate prestazioni e difficoltà di utilizzo con le implementazioni attuali;
2. Nelle applicazioni cloud self-organising, dove la scalabilità tramite parallelismo dettagliato è essenziale e miglioramenti prestazionali si traducono in una riduzione dei costi.

3.1 Architettura software

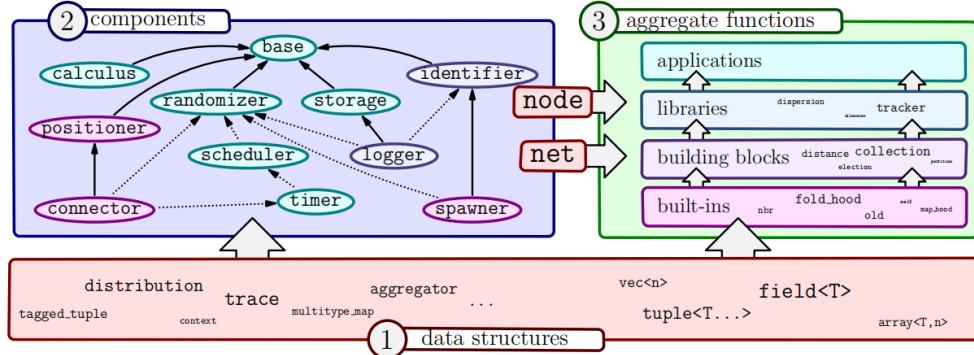


Figura 3.1: Architettura software di FCPP

La libreria FCPP è strutturata in tre livelli concettuali principali, ognuno con un ruolo specifico.

Nel primo livello, troviamo le strutture dati di uso generale in C++, che sono utilizzate sia internamente dal secondo livello per l'implementazione dei suoi componenti, sia esternamente per specificare le opzioni di configurazione. Inoltre, in questo livello, sono definite strutture dati fondamentali per il terzo livello delle funzioni aggregate, tra cui una estensione delle tipiche array e tuple in C++ che consentono operazioni su singoli componenti, una classe `vec<n>` per rappresentare vettori fisici, e soprattutto la classe `field<T>`, che implementa il concetto di campo di vicinato e supporta operazioni di mappatura e aggregazione punto per punto.

Il secondo livello è dedicato alla definizione di astrazioni per i nodi (singoli dispositivi) e per le reti di interconnessione che formano.

In un'applicazione FCPP, i tipi `nodo` e `rete` vengono creati combinando una sequenza specifica di componenti. Ogni componente offre funzionalità specifiche e questa combinazione avviene in uno stile simile a **Mixin**¹, consentendo una composizione flessibile e personalizzabile. I componenti sono implementate come classi template per consentire la composizione in fase di compilazione.

Il terzo strato della libreria fornisce l'attuazione concreta dei programmi *Field Calculus* sotto forma di funzioni aggregate, le quali accettano come parametri un nodo ed un identificatore del punto in cui la funzione è stata invocata. Questo identificatore è necessario per l'implementazione del meccanismo di allineamento automatico del *Field Calculus*.

In sintesi, la libreria FCPP offre un'architettura modulare con un'ampia gamma di strutture dati e componenti che semplificano lo sviluppo di applicazioni distribuite.

3.2 Componenti

Vediamo ora nel dettaglio come è strutturato il secondo livello concettuale di FCPP:

¹**Mixin** è una classe che fornisce metodi e attributi che possono essere utilizzati da altre classi senza la necessità di una gerarchia di ereditarietà.

Tabella 3.1: *FCPP components*

Component	Provides
base	basic update interface, node unique identifiers, reference to net objects in nodes
calculus	stack trace, context and exports, to be used indirectly through aggregate functions
identifier	manages a collection of node objects indexed by UID (creation, removal, access)
logger	periodically aggregates values from a storage component across nodes into a given stream
randomizer	random number generator, functions producing random integer or floating point values
scheduler	automatically schedules rounds
storage	holds a collection of values in a tuple-like fashion
timer	maintains temporal information and allows interactive scheduling of rounds

Positioner, Connector e Spawner, fanno parte delle features della simulazione di cui parleremo più avanti.

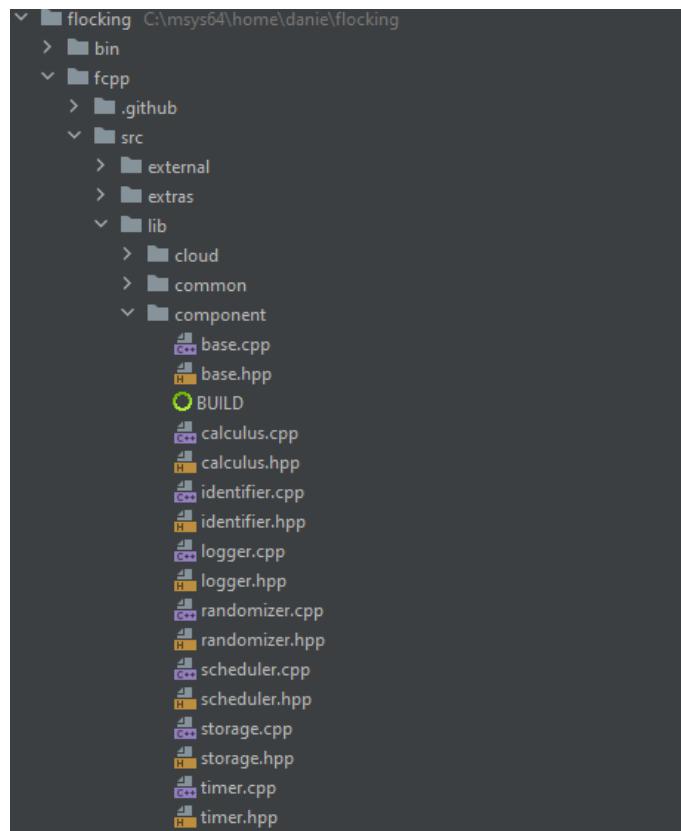


Figura 3.2: *Presentazione dei componenti FCPP*

3.3 Simulatore

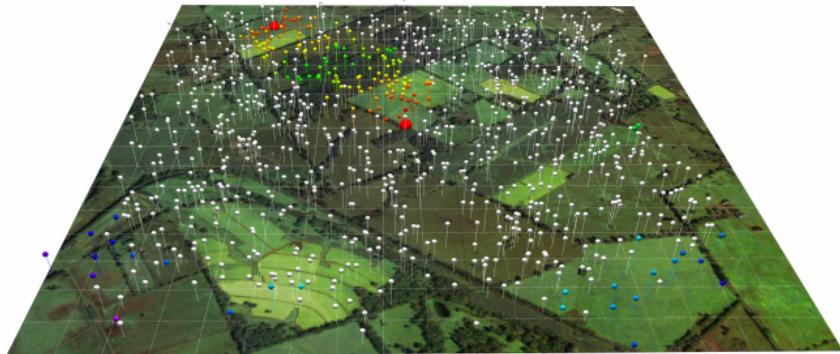


Figura 3.3: *Simulatore grafico*

- Elementi programmabili (con supporto aggregato);
- Fisica 2D e 3D con accelerazione e attrito;
- Modelli semplificati di connessione wireless;
- Dispiegamenti geometrici casuali o regolari.

In precedenza, abbiamo menzionato i concetti di **Positioner**, **Connector** e **Spawner**, che costituiscono componenti fondamentali nella fase di inizializzazione della simulazione nell'ambito di FCPP.

Connector simula trasmissioni periodiche di messaggi, determinando se la connessione è possibile in base alla posizione fisica del dispositivo simulata in un dato momento.

Positioner è il processo di evoluzione fisica dei vettori di *posizione*, *velocità* e *accelerazione* tenendo conto degli effetti dell'*attrito* nell'ambito di uno spazio *bi(o tri)-dimensionale*.

Infine **Spawner** è essenziale nel processo di creazione autonoma di nodi con un identificatore, utilizzando potenzialmente distribuzioni stocastiche.

Per ulteriori approfondimenti visitare: [FCPP GitHub](#)

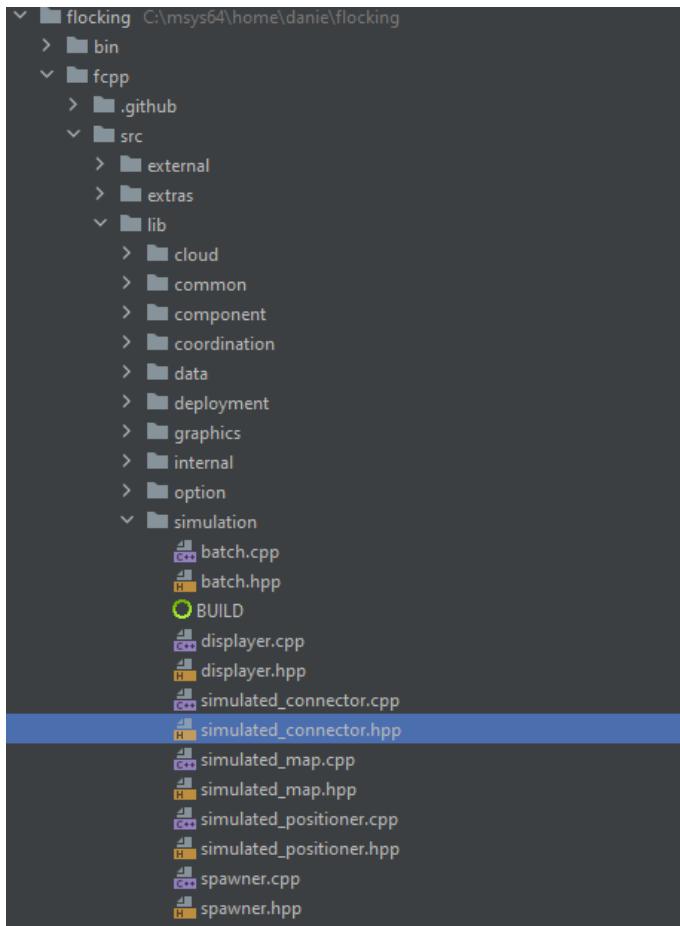


Figura 3.4: *Presentazione delle simulation features di FCPP*

3.4 Come vengono espresse le funzioni aggregate

aggregate function declaration $F ::= P \ t \ d(\text{ARGS}, t \ x*) \ \{\text{CODE } i\}$	type parameter declaration $P ::= \text{FUN} \mid \text{GEN}(T*)$ aggregate expression $e ::= x \mid \ell \mid t\{e*\} \mid [\&](t \ x*) -> t\{i\} \mid e ? e : e \mid e \circ e \mid ue \mid p(e*) \mid \text{node.c}(e*) \mid f(\text{CALL}, e*)$
type $t ::= T \mid bt \mid tt<t*, \ell * >$	aggregate function $f ::= b \mid d$
built-in aggregate functions $b ::= \text{self} \mid \text{mod_self} \mid \text{map_hood} \mid \text{fold_hood} \mid \text{old} \mid \text{nbr}$	

Figura 3.5: *Sintassi delle funzioni aggregate in FCPP*

3.4.1 Linguaggio di programmazione

Una sintassi vincolata è delineata nella Figura 1.9 per le funzioni aggregate all'interno del contesto di FCPP, con l'obiettivo di limitare le possibilità espressive del linguaggio ad un insieme essenziale di costrutti che rispecchiano quelli presenti in *Field Calculus*.

Seguendo la notazione delle *espressioni regolari*², il simbolo $*$ indica un elemento che può ripetersi, possibilmente anche zero volte, con ripetizioni sempre separate da virgole.

Una dichiarazione di funzione aggregata è introdotta tramite **FUN** oppure tramite **GEN(T*)** seguita da una serie di dichiarazioni di variabili di tipo; seguono poi specifiche su: il tipo di ritorno t , il nome della funzione d , un elenco tra parentesi di argomenti separati da virgole (preceduto da ARGs) e un'istruzione aggregata i preceduta dalla parola chiave **CODE**. La definizione dei tipi segue la sintassi di C++, dove t può essere una variabile di tipo T, un tipo "standard" bt (come bool, int, double) o un tipo "non standard" tt (come field, array, tuple) con argomenti specifici $tt < t^*, l^* >$ (es. field<double>, array<int, 3> o tuple<vec<2>, bool, field<double>>).

3.4.2 Espressioni Aggregate

Le espressioni aggregate possono essere:

- Una variabile x (sia un argomento di funzione che sia definito in una dichiarazione di tipo **let** - come descritto successivamente);
- Un valore letterale l esprimibile in C++ (ad esempio, un numero intero o floating-point, un valore booleano, stringhe letterali, ecc.);
- Un oggetto di tipo t creato mediante una chiamata al costruttore della classe **t(e*)**, con argomenti e ;
- Un operatore unario u (come ad esempio, $-$, \sim , $!$, ecc.) applicato ad e ;
- Un operatore binario e o e (ad esempio, $+$, $*$, and, or, ecc.);
- Una chiamata di funzione **p(e*)**, dove p rappresenta una funzione di base in C++, la quale non dipende dalle informazioni dei nodi né dagli scambi di messaggi;
- Una chiamata di funzione di componente **node.c(e*)**, dove c rappresenta una funzione fornita da uno dei componenti (descritti precedentemente nella sezione Componenti) che dipende dalle informazioni dei nodi ma non dai messaggi;
- Una chiamata di funzione aggregata **f(CALL, e*)**, in cui f può essere un nome definito di funzione aggregata d oppure una funzione aggregata built-in b (corrispondente a manipolazioni di campi o costrutti di Field Calculus);
- Una funzione anonima $[&](t\ x^*) \rightarrow t_r(return\ e;)$, con argomenti t x , tipo di ritorno t_r e corpo e : poiché le funzioni anonne catturano le variabili definite nel loro contesto, esse catturano anche la variabile del nodo, consentendo che il loro corpo sia un'espressione aggregata, nonostante vengano applicate come funzioni pure;
- Un'espressione di branching condizionale $e_{guard} ? e_{\top} : e_{\perp}$, tale che e_{\top} venga valutato e restituito se e_{guard} è valutato come vero, mentre e_{\perp} viene valutato e restituito se e_{guard} è valutato come falso.

²Le espressioni regolari sono sequenze di caratteri usate per cercare e manipolare testi in modo flessibile e preciso, fornendo un potente strumento per la gestione dei dati basati su modelli.

3.4.3 Funzioni aggregate built-in

Nel confrontare la sintassi delineata nella Figura 1.9 con quella del Field Calculus, emerge una notevole somiglianza nella struttura, sebbene presentino alcune differenze sintattiche. In particolare, le strutture di coordinamento **nbr**, **rep** (e le relative varianti) che rappresentano concetti primitivi all'interno del Field Calculus sono qui modellate attraverso le funzioni aggregate integrate **old** ed **nbr**. Queste due funzioni incorporate vengono sovraccaricate per gestire diverse segnature:

- La funzione **old(CALL, v₀, v)**, dove v_0 e v sono di tipo t , estrae il valore fornito come secondo argomento v dalla fase di calcolo precedente (introducendo un ritardo di un ciclo), predefinendo il valore a v_0 se il valore atteso non è disponibile;
- **old(CALL, v)** rappresenta una scorciatoia per **old(CALL, v, v)**;
- **old(CALL, v₀, f)** corrisponde all'operatore *rep* del Field Calculus e calcola il risultato dell'applicazione di una funzione f (che può essere sia una funzione aggregata anonima che una funzione pura) con segnatura $(t) \rightarrow t$ al valore complessivo della funzione **old** nell'iterazione di calcolo precedente (usando v_0 se tale valore non è disponibile);
- **nbr(CALL, v₀, v)**, con v_0 e v di tipo t , restituisce il campo di valori forniti come secondo argomento v nell'iterazione di calcolo precedente dei nodi vicini, predefinendo il valore a v_0 per il nodo corrente se non è disponibile;
- **nbr(CALL, v)** è una scorciatoia per **nbr(CALL, v, v)** e coincide con l'operatore *nbr* del Field Calculus;
- **nbr(CALL, v₀, f)** calcola il risultato dell'applicazione di una funzione f (che può essere sia una funzione aggregata anonima che una funzione pura) con segnatura $(field<t>) \rightarrow t$ al campo di valori della funzione *nbr* nell'iterazione di calcolo precedente dei nodi vicini (usando v_0 per il nodo corrente se tale valore precedente non è disponibile per esso).

Le funzioni aggregate aggiuntive comprendono:

- **self(CALL, φ)**, restituisce il valore $\phi(i)$ estratto dal campo ϕ dei vicini al nodo corrente (identificato come $i = node.uid$);
- **mod_self(CALL, φ, v)**, aggiorna $\phi(i)$ a v nel campo ϕ di tipo $field<t>$, dove i è il nodo corrente;
- **map_hood(CALL, f, v*)**, applica f in modo puntuale a una sequenza di valori locali oppure di campo v^* ;
- **fold_hood(CALL, f, φ)**, combina i valori nel dominio di ϕ di tipo $field<t>$ utilizzando l'operatore binario f di tipo $(t, t) \rightarrow t$, ottenendo un singolo valore di tipo t ;
- **fold_hood(CALL, f, φ, v)**, esegue una combinazione ϕ come sopra, utilizzando v invece del valore ϕ per il dispositivo corrente.

3.5 Esercizi preparatori

Procediamo ora a mettere in atto le conoscenze teoriche fin ora acquisite, con l'obiettivo di sviluppare le competenze necessarie per elaborare algoritmi aggregati.

Per conseguire tale obiettivo, sono stati condotti specifici esercizi preparatori, i quali rappresentano un prerequisito essenziale. Tali esercizi sono documentati e reperibili nel sito web dedicato al programma di formazione, denominato **FCPP exercises**. Gli esercizi sono stati eseguiti in sequenza, poiché ogni esercizio costituisce un presupposto per la comprensione e l'esecuzione dell'esercizio successivo, riflettendo così una struttura di dipendenza intrinseca tra di essi.

Al fine di effettuare una valutazione e un'analisi sistematica degli esercizi proposti, è stato deciso di suddividerli in tre categorie basate sul loro livello di complessità.

```
* EXERCISES:
* Expand the MAIN function below to compute the following:
*
* 1) The number of neighbour devices.
*
* 2) The maximum number of neighbour devices ever witnessed by the current device.
*
* 3) The maximum number of neighbour devices ever witnessed by any device in the network.
*
* 4) Move towards the neighbour with the lowest number of neighbours.
*
* Every exercise above is designed to help solving the following one.
*
* In order to check whether what you computed is correct, you may display the computed
* quantities as node qualities through tags `node_color`, `node_size` and `node_shape`.
* You can also save your computed quantities in additional specific node attributes:
* towards this end, you should both add a tag in namespace tags above, then list it
* (together with the corresponding data type) in the `tuple_store` option below.
*
* BONUS EXERCISES:
*
* 5) Move away from the neighbour with the highest number of neighbours.
*
* 6) Move as if the device was attracted by the neighbour with the lowest number of neighbours,
* and repulsed by the neighbour with the highest number of neighbours.
*
* 7) Move as if the device was repulsed by every neighbour, and by the four walls of the
* rectangular box between points [0,0] and [500,500].
*
* HINTS:
*
* - In the first few exercises, start by reasoning on when/where to use `nbr` (collecting from
* neighbours) and `old` (collecting from the past).
*
* - In order to move a device, you need to set a velocity vector through something like
* `node.velocity() = make_vec(0,0)` .
*
* - Coordinates are available through `node.position()`. Coordinates can be composed as physical
* vectors: `[1,3] + [2,-1] == [3,2]` , `[2,4] * 0.5 == [1,2]` .
*
* - In the last two exercises, you can model attraction/repulsion using the classical inverse square law.
* More precisely, if `v` is the vector between two objects, the resulting force is `v / |v|^3` where
* `|v| = sqrt(v.x^2 + v.y^2)` . In FCPP, `norm(v)` is available for computing `|v|`.
```

Figura 3.6: *Exercises*

3.5.1 Livello facile

I tre esercizi iniziali costituiscono una serie di attività preliminari finalizzate a facilitare la comprensione dei principi della programmazione aggregata. Analizziamo le soluzioni di ciascun esercizio per comprendere i ragionamenti e il procedimento svolto per arrivarcì.

Esercizio 1

```
//! Ex1. The number of neighbour devices for the current node.
node.storage(node_numberOfNeighbours{}) = count_hood(CALL);
```

Ciascun nodo è dotato di uno spazio di archiviazione dedicato, **storage**, all'interno del quale sono previste strutture dati vuote predisposte per l'annotazione e la categorizzazione dei dati

pertinenti. Questa configurazione agevola l'accesso e la gestione efficiente delle informazioni associate a ciascun nodo, migliorando così la facilità d'uso e l'organizzazione delle risorse informative. Si dichiara nel seguente modo:

```
using store_t = tuple_store<
    node_color,           color,
    node_size,            double,
    node_shape,           shape
>;
```

node_color, *node_size* e *node_shape* sono **tags**, aggiungere un elemento nello *storage* implica quindi aggiungere un *tag* nell'apposito namespace come segue:

```
namespace tags {
    //! @brief Color of the current node.
    struct node_color {};
    //! @brief Size of the current node.
    struct node_size {};
    //! @brief Shape of the current node.
    struct node_shape {};
    // ... add more as needed, here and in the tuple_store<...> option below
}
```

node.storage(node.numberOfNeighbours) → È stato introdotto un tag specifico denominato "node.numberOfNeighbour" al fine di archiviare il risultato della soluzione ottenuta, consentendo così la sua visualizzazione durante il corso della simulazione, allo scopo di verificare la correttezza. Sostanzialmente *storage* viene usato come strumento di debug.

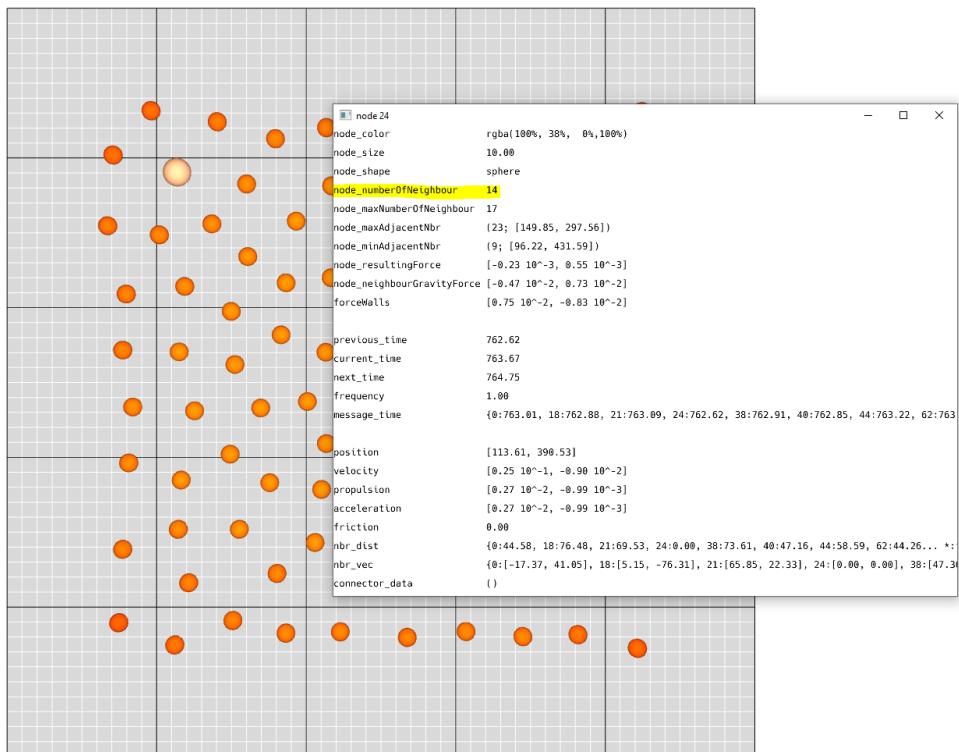


Figura 3.7: *Exercise 1*

L'effettiva soluzione dell'esercizio è data dalla funzione aggregata **count_hood(CALL)**, analizziamola.

Cos'è CALL? Il termine *CALL* rappresenta la posizione relativa all'invocazione della sudetta funzione all'interno della pila di esecuzione durante la simulazione. La funzione di libreria *count.hood(CALL)* fa parte della famiglia delle funzioni aggregate di *fold.hood*, aggrega quindi un campo di vicinato e in questo caso, tramite *count.hood*, effettua il conteggio degli elementi nel dato campo.

Gli elementi contenuti nel campo di vicinato, definito come **field**, rappresentano numericamente la quantità di dispositivi adiacenti al nodo chiamante all'interno della rete. Tale conteggio include il nodo chiamante stesso nel calcolo.

Esercizio 2

```
//! Ex2. The maximum number of neighbour devices ever witnessed by the current node.
node.storage(node_maxNumberOfNeighbour{}) = old(CALL, 0, [&](int b){
    return max(b, (int)count_hood(CALL));
});
```

Nel precedente esercizio, abbiamo applicato una funzione basata sul **Neighbourhood Interaction** (interazione tra vicini), mentre nell'attuale esercizio stiamo adottando un paradigma differente, noto come **Local History management** (gestione della storia locale). Tale scelta metodologica richiede un approfondimento:

- **Neighbourhood Interaction**, si riferisce alla dinamica d'interazione e comunicazione che si verifica tra nodi all'interno di un campo di vicinato;
- **Local History management**, si riferisce alla gestione e alla conservazione delle informazioni storiche relative ad uno specifico nodo.

In ciascun round di esecuzione, la funzione *old* calcola e restituisce il valore massimo ottenuto attraverso l'espressione *max(b, (int)count_hood(CALL))*. Tale risultato viene assegnato e memorizzato in una variabile denominata *b*. Va notato che, in caso esista già un valore all'interno di *b*, esso verrà sovrascritto dal nuovo massimo calcolato tramite l'espressione. L'effettiva valutazione di questa espressione implica il confronto tra il massimo valore precedentemente presente in *b* (ossia, il valore massimo ottenuto nel round precedente dato che il nuovo valore viene sovrascritto nel momento in cui l'espressione ritorna un risultato, quindi successivamente) e il valore corrispondente al numero di vicini nel round di esecuzione corrente, il quale viene ottenuto mediante l'uso della funzione *count.hood(CALL)* descritta in precedenza.

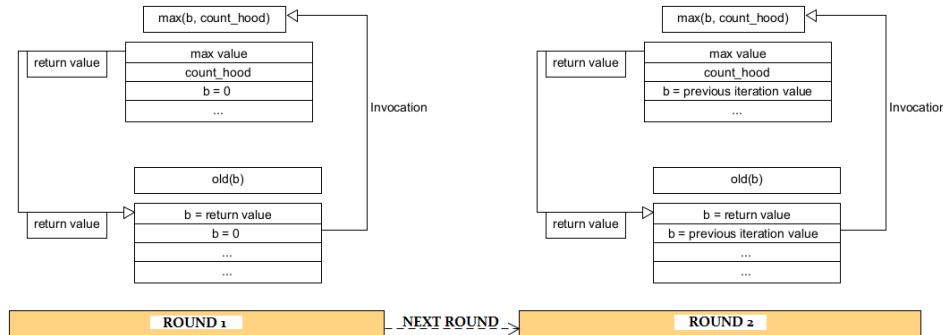


Figura 3.8: Prototipo di execution stack

Esercizio 3

```
//! Ex3. The maximum number of neighbour devices ever witnessed by any
/// device in the network.
int maxAdjacent = nbr(CALL, 0, [&](field<int> b){
    int v = count_hood(CALL);
    return max_hood(CALL, b, v);
});
node.storage(node_maxAdjacentNbr{}) = maxAdjacent;
color newColor;
node.storage(node_color{}) = newColor.hsva(maxAdjacent, 1, 1);
```

L'esercizio tre, in considerazione dei principi precedentemente affrontati, li integra in un unico procedimento. Tuttavia, è necessario effettuare una puntualizzazione.

La procedura di aggiornamento della variabile *b* segue il medesimo approccio descritto nell'esercizio precedente mediante la funzione *old*. La distinzione principale consiste nel fatto che *b* in questo contesto è un campo di vicinato (tipo **field**). Pertanto, nell'espressione aggregata, viene introdotto un ulteriore passaggio: **max_hood**, una funzione inclusa nell'insieme delle funzioni aggregate di **fold_hood**, che ha la finalità di ridurre il campo di vicinato *b* in un unico valore, mediante il massimo tra le due quantità date: il valore ottenuto dalla funzione *count_hood* ed il valore attualmente contenuto in *b*.

3.5.2 Livello medio

Gli esercizi quattro e cinque focalizzano la loro attenzione sulla manipolazione e l'apprendimento delle corrette configurazioni delle velocità di spostamento assegnate a ciascun dispositivo.

Esercizio 4

```
//! Ex4. Move towards the neighbour with the lowest number of neighbours.
tuple<int, vec<2>> t = make_tuple((int)count_hood(CALL), node.position());
field<tuple<int, vec<2>>> b = nbr(CALL, t);
node.storage(node_minAdjacentNbr{}) = min_hood(CALL, b);
node.velocity() = (get<1>(node.storage(node_minAdjacentNbr{})))
    - node.position()/100;
```

Negli esercizi precedenti, l'impiego di funzioni aggregate con il parametro "[&] **tipo nome**" (*in C++ si tratta di una funzione lambda, o anonima*) consentiva l'aggiornamento progressivo di una variabile in ogni ciclo di esecuzione, basandosi sulla storia di un dispositivo conservata nello stack. Come si può osservare dall'estratto di codice sopra menzionato, in questo esercizio e nei successivi, tale concetto non viene più impiegato né si fa uso della storia di un nodo.

Focalizziamo ora la nostra attenzione sull'ultima istruzione di codice, in cui la velocità viene determinata come un vettore di spostamento caratterizzato da un verso e da una direzione, ottenuti dalla differenza tra le due posizioni specificate.

In termini più pratici, se immaginiamo una freccia che rappresenta *node.position()* e una seconda freccia che rappresenta *get<1>(node.storage(node_minAdjacentNbr))* (posizione del nodo con il minor numero di vicini), allora il risultato di *get<1>(node.storage(node_minAdjacentNbr)) - node.position()* sarà una freccia che parte dalla testa di *node.position()* (nell'immagine sottostante corrisponde al **punto di applicazione**) e punta alla testa di *get<1>(node.storage(node_minAdjacentNbr))*, indicando la direzione dello spostamento da **node.position()** a **get<1>(node.storage(node_minAdjacentNbr))**.

Infine il vettore risultante viene scalato dividendolo per 100; questa operazione non è significativa, è implementata con l'intento di modulare la velocità impostata, introducendo un effetto di rallentamento.

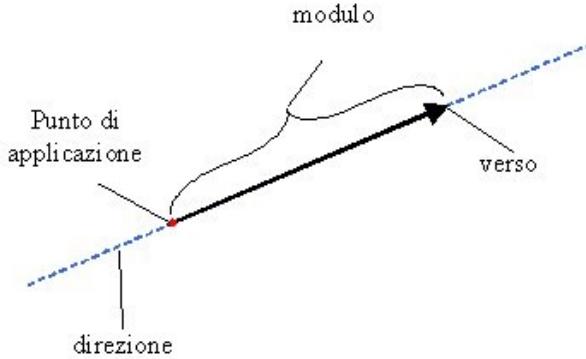


Figura 3.9: Vettori velocità

Quanto appena descritto è uno dei principi chiave, fondamentali su cui si baserà il lavoro di **Formation Control**.

Esercizio 5

```
//! Ex5. Move away from the neighbour with the highest number of neighbours.
tuple<int, vec<2>> t = make_tuple((int)count_hood(CALL), node.position());
field<tuple<int, vec<2>>> b = nbr(CALL, t);
node.storage(node_maxAdjacentNbr{}) = max_hood(CALL, b);
node.velocity() = (node.position() - get<1>(node.storage(node_maxAdjacentNbr{}))) /100;
```

Questo esercizio rappresenta una reinterpretazione di quello precedente, in cui le medesime operazioni vengono eseguite in modo speculare, riflesso rispetto all'esercizio quattro.

3.5.3 Livello difficile

Gli esercizi sei e sette si concentrano sull'analisi e la comprensione di leggi fisiche specifiche che rivestiranno un ruolo fondamentale nel contesto della **Formation Control**.

Esercizio 6

```
/*! Ex6. Move as if the device was attracted by the neighbour with the
lowest number of neighbours, and repulsed by the neighbour with the
highest number of neighbours.*/
vec<2> force = make_vec(0,0);
if(get<1>(node.storage(node_minAdjacentNbr{})) != node.position()) {
    vec<2> betweenMin = get<1>(node.storage(node_minAdjacentNbr{}))
        - node.position();
    force += (betweenMin/pow(norm(betweenMin), 3));
}
if(get<1>(node.storage(node_maxAdjacentNbr{})) != node.position()) {
    vec<2> betweenMax = -(get<1>(node.storage(node_maxAdjacentNbr{}))
        - node.position());
```

```

        force += (betweenMax/pow(norm(betweenMax), 3));
    }
node.storage(node_resultingForce{}) = force;
node.propulsion() = node.storage(node_resultingForce{});

```

Studiamo il codice passo dopo passo:

1. Inizialmente, viene creato un vettore *force* per rappresentare la **forza risultante** che agirà sul dispositivo;
2. Viene calcolato il **vettore distanza** *betweenMin* tra la posizione del vicino con il minor numero di vicini e la posizione del dispositivo corrente secondo la regola dei vettori utilizzata per l'esercizio quattro;
3. Viene calcolata la **forza di attrazione** tra i due dispositivi presi in considerazione nel punto precedente tramite la legge dell'inverso del quadrato come segue:

$$\frac{distance\ Vector}{|distance\ Vector|^3}$$

4. Il medesimo procedimento del punto 3 e 4 (*invertito*), è utilizzato per calcolare la forza repulsiva nei confronti del dispositivo con il più alto numero di vicini;
5. Infine, la forza risultante viene utilizzata per impostare la propulsione ³ del dispositivo, influenzando così il suo movimento nel sistema distribuito.

Esercizio 7

```

/*! Ex7. Move as if the device was repulsed by every neighbour, and by
the four walls of the rectangular box between points [0,0] and [500,500].*/
node.storage(node_neighbourGravityForce{}) = sum_hood(CALL, map_hood([](
vec<2> v, double m){
    double d = norm(v);
    return v * -(m) / pow(d, 3);
},node.nbr_vec(), nbr(CALL, node.storage(node_size{}))), vec<2>{}));
vec<2> forceW = make_vec(0,0);
for(int i = 1; i <= 500; i++){
    vec<2> b = -(make_vec(i,0) - node.position());
    vec<2> h = -(make_vec(0,i) - node.position());
    forceW += (h / pow(norm(h), 3)) + (b / pow(norm(b), 3));
}
for(int j = 500; j > 0; j--){
    vec<2> bb = -(make_vec(j,500) - node.position());
    vec<2> hh = -(make_vec(500,j) - node.position());
    forceW += (hh / pow(norm(hh), 3)) + (bb / pow(norm(bb), 3));
}
node.storage(forceWalls{}) = forceW;
node.propulsion() = node.storage(node_neighbourGravityForce{})
                    + node.storage(forceWalls__);
/*! Simulazione di una sorta di attrito che tende a decelerare, moltiplicando
per 0.1 riduco del 10% la velocità ad ogni round...*/
if(node.velocity() > 0){

```

³**Differenza tra propulsione e velocità:** la propulsione (o accelerazione) è responsabile del cambiamento della velocità di un oggetto, aumentandola o diminuendola, mentre la velocità è una misura istantanea del tasso di spostamento

```

        node.velocity() -= 0.1 * node.velocity();
    }

```

In questa fase finale degli esercizi preparatori, i nodi sono in grado di eseguire in simulazione un comportamento visivamente più realistico.

Tale risultato è stato ottenuto attraverso l'applicazione dell'insieme di competenze pratiche acquisite nei precedenti esercizi.

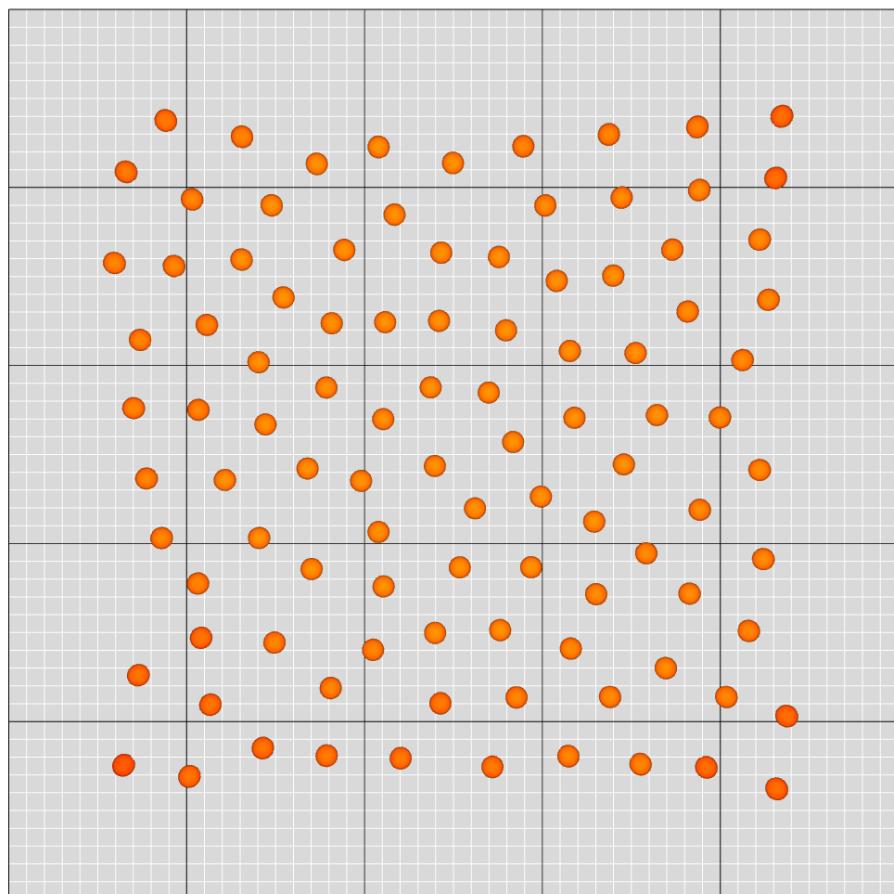


Figura 3.10: *Exercise 7*

I dispositivi si respingono tra loro e un singolo dispositivo viene respinto dalle pareti che delimitano lo spazio. Ciò determina una diffusione degli stessi dispositivi nello spazio, analogamente al comportamento delle molecole in un gas.

Per ulteriori approfondimenti riguardo le funzioni di libreria visitare: **[documentazione FCPP](#)**.

Capitolo 4

Circular Forces

4.1 Introduzione

In questa strategia, ci ispiriamo al concetto di **campo di potenziale artificiale**[6] al fine di affrontare problemi di **avoidance**¹ nell'ambito della navigazione autonoma di agenti mobili. Nello specifico un potenziale è costituito da un **potenziale attrattivo**, generato dall'agente *master*, mentre gli altri agenti *slave* contribuiscono ad un **potenziale repulsivo**. Questi due potenziali lavorano insieme per guidare l'agente *slave* preso in considerazione lungo un percorso sicuro con l'obiettivo di formare una circonferenza attorno al *master*.

Il controllo dell'agente risulta dal fatto che il potenziale artificiale definisce un campo vettoriale (*sull'area di lavoro*), in cui ciascun punto ha un vettore di velocità associato.

In altre parole l'agente segue il *flusso* di questo campo vettoriale, formato da **gradienti di velocità**², per raggiungere il suo obiettivo senza collisioni.

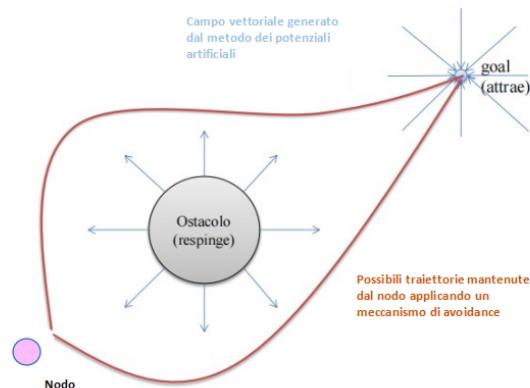


Figura 4.1: Campo vettoriale generato dal metodo dei potenziali artificiali

¹L'**evitamento (o avoidance)** è il processo mediante il quale un sistema o un agente prende misure o compie azioni specifiche per evitare collisioni, ostacoli o situazioni indesiderate mentre si muove o interagisce all'interno di un ambiente.

²**Gradienti di velocità:** è una quantità vettoriale che ha una direzione e un modulo associati e rappresenta la direzione e l'intensità della massima variazione con cui la velocità di un oggetto cambia quando ci si sposta in una determinata direzione nello spazio. Spesso associata all'accelerazione di un oggetto o alla forza che agisce su di esso.

Analiticamente è necessario definire una funzione potenziale come la combinazione di due funzioni: una caratterizzata dal potenziale attrattivo, l'altra dal potenziale repulsivo.

Lavoreremo su uno spazio a due dimensioni in cui il sistema di riferimento, che permette di individuare la posizione dell'agente, è stabilito dalle coordinate (x, y) definite in un vettore v .

Il campo potenziale all'interno del quale si muove l'agente sarà quindi una funzione scalare $\mathbf{U}(\mathbf{v})$ da \mathbf{R}^2 in \mathbf{R} generata dalla sovrapposizione dei potenziali repulsivi e attrattivi:

$$U(v) = U_a(v) + U_r(v)$$

A sua volta il potenziale repulsivo è visto come la sommatoria delle singole forze repulsive date dai singoli ostacoli:

$$U_r(v) = \sum_{i=1}^{N_{obst}} U_{r_i}(v)$$

Gli algoritmi implementati traggono ispirazione dalle formule esposte, anche se non riproducono esattamente la stessa struttura. Non si adotta la definizione di una funzione potenziale $U(v)$ che prescrive le velocità per ciascun punto, ma piuttosto si lavora direttamente con le forze e le accelerazioni.

4.2 Problema

La nostra ricerca è incentrata sulla creazione di una configurazione circolare intorno ad un nodo principale chiamato **Master**, circondato da un gruppo di nodi ausiliari di nome **Slave**. Questo sistema è guidato da forze elastiche che mirano a creare e mantenere una determinata configurazione spaziale:

1. **Forza Elastica tra Master e Slave:** viene introdotta una forza elastica con una costante di rigidità che chiameremo *hardnessMasterSlave* che agisce tra il nodo *Master* e ciascun nodo *Slave*. Questa forza è progettata per mantenere i nodi *Slave* ad una distanza *R* predefinita dal nodo *Master*. La forza è proporzionale alla discrepanza tra la distanza attuale e la distanza desiderata;
2. **Forza Elastica tra i nodi Slave:** viene implementata un'ulteriore forza elastica con una costante di rigidità che chiameremo *hardnessSlaveSlave* che agisce tra i nodi *Slave* stessi. Questa forza è attiva solo quando la distanza tra due nodi *Slave* è inferiore ad un valore distanza specificato. In tal caso, la forza elastica agisce per separare i nodi *Slave*, evitando che si avvicinino troppo l'uno all'altro, altrimenti questa forza è nulla.

4.3 Configurazione

Esaminiamo la fase di configurazione del programma proposto per il problema descritto.

```
/// File: circularForces_setup.hpp

using store_t = tuple_store<
    /// Date without prefix "node_" is used for plot
    node_color,           color,
    node_size,            double,
    node_shape,           shape,
    node_isMaster,        bool,
    node_isSlave,         bool,
    node_posMaster,       tuple<bool, vec<2>>,
    node_numberOfSlaves, int,
    node_myElasticForceSlaves, vec<2>,
    node_myElasticForceMaster, vec<2>,

    elastic_force_slaves, double,
    elastic_force_master, double,
    slaves_velocity,      double,
    master_velocity,      double,

    /// Prospettiva uno per il calcolo del margine d'errore
    node_exactExpectedPosition,   vec<2>,
    node_indexPosition,          int,
    position_error,              double,

    /// Prospettiva due per il calcolo del margine d'errore
    expected_dist_master_slave, double,
    effective_dist_master_slave, double,
    max_dist_master_slave,       double,
    min_dist_master_slave,       double,
    expected_dist_slave_slave,   double,
    max_dist_slave_slave,        double,
    min_dist_slave_slave,        double,

    /// next e previous slave sono gli slave
    /// vicini fisicamente ad uno specifico slave
    /// nel perimetro della circonferenza
    next_slave_distance,         double,
    previous_slave_distance,     double
>

/// File: circular_forces.hpp

/// The maximum communication range between nodes.
constexpr size_t communication_range = 200;
/// Constant minimum number of nodes to form a circle
constexpr int minNodesToFormCircle = 5;
/// Distance master-slave
constexpr double distanceMasterSlave = communication_range -
```

```

((communication_range / 100) * 60);

///! Number of people in the area.
constexpr int node_num = minNodesToFormCircle + 5;
///! Id master.
constexpr int masterUid = 0;
///! Constant reductor of velocity for master max velocity
constexpr int reductorMaster = 100;
///! Constant pi-greco usata per i calcoli della distanza Slave-Slave.
constexpr double pi = 3.14159265358979323846;
///! Hardness constant for elastic force Master-Slave.
constexpr double hardnessMasterSlave = (distanceMasterSlave / 10000) * 2;
///! Hardness constant for elastic force Slave-Slave.
constexpr double hardnessSlaveSlave = 0.005;
///! @brief Percentage for velocity reduction round by round
constexpr double reduction = 0.025;
///! @brief Percentage reductor of velocity for master
constexpr double reductorMaster = 0.5;
///! @brief Percentage increment force to keep up with master
constexpr double incrementForce = 50;
///! @brief Max corner for master movement in rectangle
constexpr vec<2> maxCorner = make_vec(0, 0);
///! @brief Min corner for master movement in rectangle
constexpr vec<2> minCorner = make_vec(500, 500);
///! @brief Max value period for master moviment in rectangle
constexpr double maxPeriod = 0.1;
///! @brief Max slaves velocity
constexpr int maxVelocitySlaves = 6;
///! @brief Max master velocity
constexpr int masterVelocity = 5;

```

circularForces_setup

Nel file **"circularForces_setup.hpp"** vengono inizializzati gli elementi necessari per un algoritmo di controllo mirato a generare una circonferenza conformemente alle direttive delineate nel problema descritto. Ora esaminiamo più approfonditamente il significato di tali elementi impiegati.

Vengono assegnati ruoli specifici a ciascun dispositivo mediante l'utilizzo di due variabili booleane: **node_isMaster** e **node_isSlave**. È fondamentale stabilire in modo inequivocabile la posizione del dispositivo principale (il *master*). Pertanto, la posizione del *master*, **node_posMaster**, è rappresentata da un vettore associato ad una variabile booleana. Come vedremo in seguito, solo il dispositivo *master* ha il privilegio di comunicare la propria posizione.

La configurazione della formazione subisce un'evoluzione dinamica, ne consegue che il numero di dispositivi *slave* può variare ad ogni iterazione, l'elemento **node_numberOfSlaves** denota il conteggio attuale dei dispositivi *slave* presenti all'interno della formazione in un dato momento. Questa quantità non è statica, poiché non tutti i dispositivi *slave* diventano immediatamente parte della circonferenza. Tale appartenenza dipende dalla posizione di ciascun dispositivo *slave* rispetto al dispositivo *master*, considerando un *range* di comunicazione specifico. Di conseguenza, un dispositivo *slave* diventa parte della circonferenza solamente quando riesce a stabilire una comunicazione con il dispositivo *master*.

Questi tre elementi iniziali costituiscono i pilastri su cui si fonda ogni calcolo successivo.

Abbiamo precedentemente discusso delle forze coinvolte per la risoluzione del problema, e le variabili **node_myElasticForceSlaves** e **node_myElasticForceMaster** rappresentano queste forze. Ora concentriamoci sulla definizione degli elementi successivi.

Le restanti variabili costituiscono un meccanismo di monitoraggio dell'errore. Come precedentemente discusso, l'approccio utilizzato da *Circular Forces* nell'implementazione dello **swarm** non si basa su calcoli altamente precisi, il che comporta la presenza di un **margine di errore** che viene esaminato da due prospettive differenti:

- Determinare con precisione la posizione richiesta per formare una circonferenza perfetta attraverso l'applicazione di calcoli trigonometrici e valutare l'errore in base alla differenza tra la posizione attuale e quella accuratamente identificata;
- Calcolare i valori massimi e minimi relativi alla distanza tra il dispositivo principale e quelli che lo circondano. Affinché il margine d'errore sia nullo, questi due valori devono essere uguali. Inoltre, stabilire la distanza tra il dispositivo slave e i dispositivi slave immediatamente adiacenti ad esso. Anche in questo caso, per ottenere un margine d'errore nullo, tali valori devono essere identici.

circular_forces

Nel file ”**circular_forces**”, vengono dichiarate delle costanti, elencate nell’estratto di codice sopra, che saranno oggetto di modifica al fine di condurre esperimenti e testare il codice. Queste costanti sono essenziali per la fase di inizializzazione e la gestione complessiva del sistema. Inoltre, all’interno di questo file si trova il nucleo centrale dell’algoritmo.

4.4 Descrizione dell’algoritmo

```
MAIN() {
    using namespace tags;
    node.storage(node_color{}) = color(PINK);

    //! Fase 1
    initialization(CALL);

    if (node.storage(node_isSlave{})) {

        //! Fase 2
        resultingForce(CALL);

        //! Fase 3
        errorCalculator(CALL);
    }

    //! Stabilizzazione delle forze interagenti attraverso l’uso
    //! di un attrito simulato
    if(node.velocity() > 0){
        node.velocity() -= node.velocity() * reduction;
    }

    //! Continuazione della fase 3
    distanceDevices(CALL);
    if(node.storage(node_isSlave{})) {
        node.storage(slaves_velocity{}) = norm(node.velocity());
    }
}
```

```

    }else{
        node.storage(master_velocity{}) = norm(node.velocity());
    }
}

```

Il programma può essere suddiviso in tre fasi distinte:

1. **Inizializzazione**;
2. **Applicazione delle forze**;
3. **Calcolo del margine d'errore**.

Stabilizzazione delle forze interagenti

La velocità dei dispositivi viene predeterminata durante la fase di *inizializzazione*, per quanto riguarda il dispositivo *master*, e durante la fase di *applicazione delle forze*, per quanto riguarda invece i dispositivi *slave*, consentendo la specifica di una velocità desiderata che sarà successivamente calibrata tramite l'utilizzo di un **attrito viscoso** simulato.

Questo tipo di attrito è implementato mediante il fattore di riduzione **reduction**, configurato inizialmente.

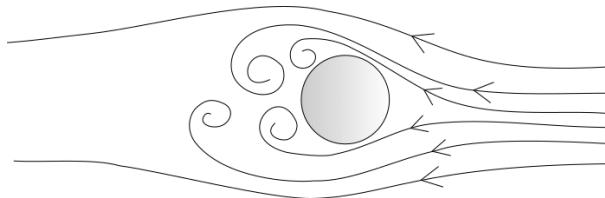


Figura 4.2: *Legge di Stokes applicata dall'attrito viscoso*

L'**attrito viscoso** è una forza di attrito che è proporzionale alla velocità dell'oggetto. In questo caso, la velocità degli agenti viene moltiplicata per il valore di **reductor** per ottenere una riduzione nella velocità.

Quando il valore **reductor** è 0, non c'è attrito e la velocità degli agenti rimane invariata. Tuttavia, quando il valore di **reductor** aumenta, l'attrito aumenta e la velocità degli agenti viene ridotta in modo proporzionale a questo valore. Questo meccanismo consente di controllare la velocità dei dispositivi in modo dinamico in base alle condizioni dell'ambiente o della formazione.

La descrizione precedente si riferisce ad una applicazione semplificata della legge di Stokes, la quale stabilisce la relazione tra la forza di attrito viscoso interagente su una sfera in movimento attraverso un fluido viscoso e variabili come la viscosità del fluido, il raggio della sfera e la sua velocità rispetto al fluido.

Questo concetto è ampiamente impiegato all'interno delle nostre strategie di controllo algorithmico.

4.4.1 Inizializzazione

La fase iniziale del programma consiste nell'effettuare l'inizializzazione dei parametri necessari per condurre i calcoli successivi. Esaminiamo ora come garantire l'affidabilità e l'accuratezza dei dati.

```

FUN void initialization(ARGs) {
    using namespace tags;
    tuple<bool, vec<2>> t = make_tuple(false, make_vec(0, 0));

```

```

        if (node.uid == masterUid) {
            node.storage(node_isMaster{}) = true;
            node.storage(node_isSlave{}) = false;
            node.storage(node_color{}) = color(LIGHT_BLUE);
            t = make_tuple(true, node.position());
        } else {
            node.storage(node_isMaster{}) = false;
            node.storage(node_isSlave{}) = true;
        }

        int slaves = (int) count_hood(CALL) - 1;
        if (node.storage(node_isSlave{})) {
            slaves = 0;
        }
        field<int> f = nbr(CALL, 0, slaves);
        node.storage(node_numberOfSlaves{}) = max_hood(CALL, f);

        field < tuple < bool, vec < 2>>> ff = nbr(CALL, t);
        tuple<bool, vec<2>> m = max_hood(CALL, ff);
        node.storage(node_posMaster{}) = m;

        //! Vedremo il seguito di tale funzione poi ...
    }
}

```

La selezione del leader è di natura semplice e non richiede un meccanismo formale di elezione del leader. Il leader sarà determinato in base all'identificativo (*uid*) scelto durante la configurazione delle costanti. In questo modo, le variabili di ruolo vengono inizializzate in modo appropriato, e va notato come la posizione del leader scelto viene comunicata, attraverso un flag che lo identifica.

Ciò è dovuto al fatto che gli *slave* potrebbero trasmettere informazioni errate. Ad esempio, la struttura del codice per la condivisione della posizione del *master*, senza alcun controllo, potrebbe portare a situazioni in cui tutti gli *slave* condividono la posizione del *master*, anche se questa informazione potrebbe non essere sempre aggiornata.

Cosa succederebbe se un nodo *slave* ricevesse un'informazione non aggiornata?
 Nel caso in cui questi dati non aggiornati risultassero essere superiori alla posizione effettiva del *master*, senza un adeguato controllo sulla validità e sull'attualità delle informazioni trasmesse dal *master*, potrebbe verificarsi una serie di conseguenze indesiderate. Alcuni nodi *slave* potrebbero interrompere la loro comunicazione con il vero *master*, erroneamente pensando che la posizione comunicata sia la corretta, oppure potrebbero sempre erroneamente riconoscere un altro nodo *slave* come il *master* e posizionarsi attorno a quest'ultimo. Un approccio simile è adottato per quanto riguarda la notifica del numero di nodi slave presenti nella formazione, sempre con l'obiettivo di mantenere la sincronizzazione.

È importante sottolineare che tali dati vengono aggiornati con una frequenza massima di una volta per ogni ciclo operativo (round).

```

if (node.storage(node_isMaster{})) {
    rectangle_walk(CALL, minCorner, maxCorner, masterVelocity, maxPeriod);
    if (node.velocity() > 0){
        node.velocity() -= node.velocity() * reductorMaster;
    }
}

```

In questa ultima parte della funzione di inizializzazione, viene impostata la velocità del nodo *master*.

Movimento del *Master*

Il nodo *master* viene spostato utilizzando la funzione **rectangle_walk**, funzione di libreria. Questa funzione fa sì che il nodo *master* segua un percorso casuale all'interno di un rettangolo, entro i limiti dei vertici massimi e minimi previamente definiti, mantenendo una velocità costante.

La riduzione della velocità tramite il concetto di attrito, nel contesto del nodo *master*, non viene impiegata al fine di stabilizzare il suo moto. Invece, essa viene utilizzata per mettere in luce le disparità di efficienza tra i nodi *slave* e il nodo *master*. In seguito, esamineremo dettagliatamente questo caratteristica.

4.4.2 Applicazione delle forze

La fase intermedia di applicazione delle forze del programma riveste un ruolo centrale nella soluzione del problema in esame. Attraverso la funzione qui di seguito presentata, si crea la formazione circolare richiesta, un passo cruciale nell'attuazione della nostra strategia.

```

FUN void resultingForce(ARGS){
    using namespace tags;
    if(get<0>(node.storage(node_posMaster{}))) {

        //! Forza interagente tra master e slave
        vec<2> v = get<1>(node.storage(node_posMaster{})) - node.position();
        node.storage(node_myElasticForceMaster{}) = v * ((1 - distanceMasterSlave
                                                        / (norm(v))) * hardnessMasterSlave);
        node.storage(elastic_force_master{}) = norm(node.storage(
            node_myElasticForceMaster{}) * incrementForce);
        node.propulsion() = node.storage(node_myElasticForceMaster{});

        //! Forza interagente tra slave
        double distanceSlaveSlave = (2 * distanceMasterSlave * pi)
                                      / node.storage(node_numberOfSlaves{});
        node.storage(expected_dist_slave_slave{}) = distanceSlaveSlave;
        node.storage(node_myElasticForceSlaves{}) = sum_hood(CALL,
            map_hood([](vec<2> v, double d, double l, double h) {
                if(d < 1)
                    return v * ((1 - l / (norm(v))) * h);
                else
                    return make_vec(0, 0);
            }, node.nbr_vec(), node.nbr_dist(), distanceSlaveSlave, hardnessSlaveSlave),
            vec<2> {});
        node.storage(elastic_force_slaves{}) = norm(node.storage(
            node_myElasticForceSlaves{}) * incrementForce);
        node.propulsion() += node.storage(node_myElasticForceSlaves{});

        //! Se date le forze applicate la velocità dello slave eccede il limite
        //! massimo viene riportata alla massima velocità consentita tramite
        //! l'operazione sottostante
        if(norm(node.velocity()) > maxVelocitySlaves){
            node.velocity() *= maxVelocitySlaves/norm(node.velocity());
        }
    }
}

```

Prima di procedere all'analisi delle formule impiegate per determinare le forze interagenti tra i dispositivi, è opportuno esaminare i concetti fondamentali su cui si basa questa funzione. Consideriamo ora un contesto in cui due agenti devono affrontare individualmente una sfida di *tracking* verso un obiettivo comune (\rightarrow *disporsi nel perimetro della circonferenza*), mentre contemporaneamente devono gestire in tempo reale la questione della prevenzione delle *collisioni* per evitare impatti. Ogni agente, quindi, percepisce l'altro agente come un ostacolo mobile da evitare caratterizzato da coordinate conosciute.

Desideriamo sviluppare una strategia di controllo, analoga al concetto dei potenziali artificiali, al fine di gestire e risolvere contemporaneamente sia il problema del *tracking*, sia la questione della prevenzione delle collisioni. Definiamo tale strategia come strategia di **Obstacle Avoidance**, per cui l'obiettivo consiste nel raggiungere il perimetro della circonferenza, mantenendo una distanza minima dagli ostacoli predefinita.

Per raggiungere il nostro scopo, è stato deciso di integrare nel sistema una **forza elastica**, seguendo i principi fondamentali della **legge di Hooke**.

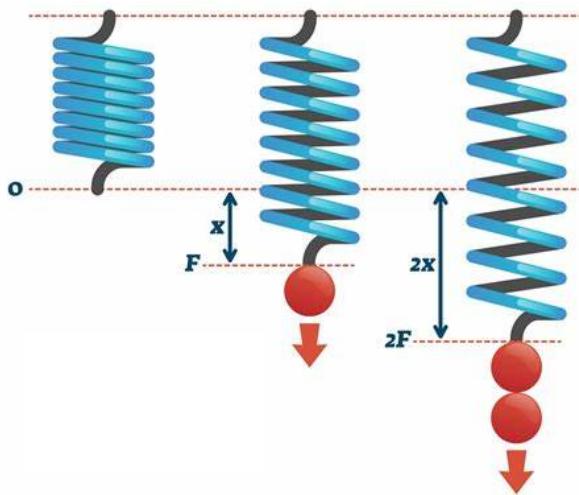


Figura 4.3: Legge di Hooke applicata dalla forza elastica

Forza interagente tra master e slave

$$x = \mathbf{v} \cdot \left(1 - \frac{\text{distanceMasterSlave}}{\|\mathbf{v}\|} \right)$$

$$k = \text{hardnessMasterSlave}$$

$$f = x \cdot k$$

Tale formula può essere descritta come segue:

1. \mathbf{v} rappresenta un vettore distanza tra il nodo *master* e il nodo *slave*;
2. **distanceMasterSlave** è la distanza desiderata tra il nodo *master* e il nodo *slave*;
3. **norm(v)** rappresenta la norma del vettore v , ossia la lunghezza della distanza tra i due dispositivi;

4. Il rapporto **distanceMasterSlave / (norm(v))** rappresenta la frazione della distanza corrente rappresentata della distanza desiderata. Se essa è 1 non occorre applicare alcuna forza;
5. **hardnessMasterSlave** è un parametro di elasticità o durezza, che indica quanto rigido sia il materiale della molla virtuale, ovvero quanto intensa debba essere la forza elastica tra i nodi.

Riassumendo, la formula complessiva calcola la forza elastica tra il nodo *master* e il nodo *slave* moltiplicando il vettore *v* per un fattore che tiene conto della distanza desiderata tra i nodi e del parametro di elasticità.

Questo calcolo rappresenta una forza che cerca di ricondurre i nodi alla loro posizione di riferimento, con l'intensità della forza regolata dalla durezza del materiale e dalla distanza tra i nodi.

Forza interagente tra slave

La formula utilizzata per il calcolo della forza interagente è identica a quella precedentemente descritta. Tuttavia, è importante concentrarsi sui procedimenti preliminari al calcolo della forza, in particolare sull'inizializzazione dinamica di alcuni dei parametri necessari per effettuare tale calcolo.

$$distanceSlaveSlave = \frac{2 \cdot distanceMasterSlave \cdot pi}{node.storage(node.numberOfSlaves)}$$

Tale formula rappresenta un'approssimazione della suddivisione di una circonferenza e può essere descritta come segue:

1. **2 * distanceMasterSlave** rappresenta la lunghezza totale della circonferenza di un cerchio con un raggio uguale;
2. **pi** è il simbolo per il valore costante di π , utilizzato per calcolare la lunghezza della circonferenza di un cerchio;
3. **node.storage(node.numberOfSlaves)** rappresenta il numero totale di parti in cui si desidera suddividere la circonferenza.

La formula nel suo complesso calcola la dimensione di ciascuna parte o intervallo in cui la circonferenza verrà divisa in modo uniforme, tenendo conto del numero specifico di parti desiderate.

Questo calcolo riveste un ruolo importante nel contesto della suddivisione equa di una circonferenza allo scopo di formare lo *swarm* in modo uniforme.

4.4.3 Calcolo del margine d'errore

Abbiamo precedentemente menzionato due diverse prospettive per comprendere l'errore, ognuna delle quali implica un procedimento di calcolo distinto.

Errore delle posizioni

```
FUN void errorCalculator(ARGS){
    using namespace tags;
    if(get<0>(node.storage(node_posMaster{}))) {
```

```

//! Calcolo delle possibili posizioni occupabili
vec<2> arrayOfExactLocation[node.storage(node_numberOfSlaves{})];
for(int i = 1; i <= node.storage(node_numberOfSlaves{}); i++){
    double radiant = i * ((2 * pi) / node.storage(node_numberOfSlaves__));
    double x = (std::sin(radiant)) * distanceMasterSlave;
    double y = (std::cos(radiant)) * distanceMasterSlave;
    vec<2> exactLocation = make_vec(x, y) +
        get<1>(node.storage(node_posMaster{}));
    arrayOfExactLocation[i-1] = exactLocation;
}

//! Intercettamento della posizione occupata
int index = 1;
double minError = distance(arrayOfExactLocation[0], node.position());
vec<2> expectedPosition = arrayOfExactLocation[0];
for(int i = 0; i < node.storage(node_numberOfSlaves{}); i++){
    double currentError = distance(arrayOfExactLocation[i], node.position());
    if(compare(currentError, minError)){
        index = i + 1;
        expectedPosition = arrayOfExactLocation[i];
        minError = currentError;
    }
}
node.storage(position_error{}) = minError;
node.storage(node_exactExpectedPosition{}) = expectedPosition;
node.storage(node_indexPosition{}) = index;
}
}

```

Illustriamo il codice attraverso la seguente analisi.

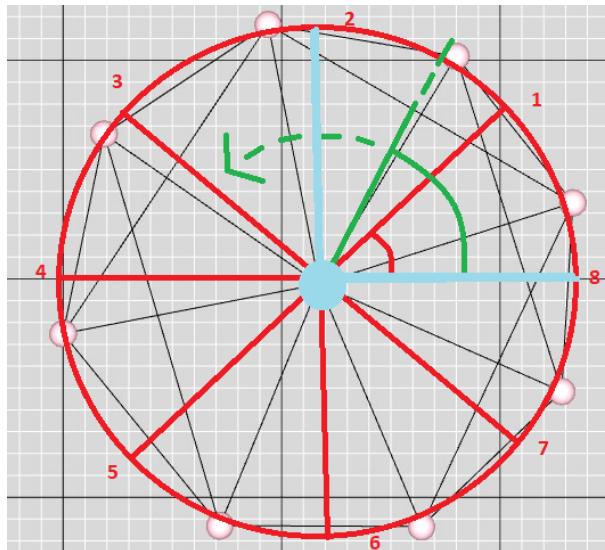


Figura 4.4: Circonferenza formata da Circular Forces rispetto ad una formazione teorica

Abbiamo già introdotto la formula per dividere equamente la nostra circonferenza, per capire l’assegnamento degli angoli, enunciamo formalmente la formula per la **divisione angolare della circonferenza**. In forma generale, questa formula può essere espressa come:

$$\text{Angolo Centrale (in radianti)} = \frac{\text{Ampiezza Angolare Totale}}{\text{Numero di Divisioni}}$$

Dove:

- L’**Angolo Centrale** rappresenta l’ampiezza angolare tra ciascuna divisione della circonferenza;
- L’**Aampiezza Angolare Totale** rappresenta l’ampiezza angolare dell’intera circonferenza, che è 2π radianti o 360° ;
- Il **Numero di Divisioni** è il numero desiderato di parti in cui si desidera suddividere la circonferenza.

Successivamente, per assegnare un angolo distintivo a ciascuna posizione, si moltiplica l’indice della posizione i (dove i varia da 1 al *numero di divisioni*) per l’*angolo centrale* che etichetteremo come β . Questa operazione assegna a ciascuna posizione un angolo univoco sulla circonferenza.

Matematicamente, l’angolo θ_i assegnato alla posizione i è dato da:

$$\theta_i = i \cdot \beta$$

In questo modo, **ogni posizione sulla circonferenza viene associata a un angolo distintivo** θ_i ottenuto dalla moltiplicazione dell’indice i per l’angolo centrale β .

Adesso siamo in possesso delle conoscenze necessarie per fornire una spiegazione dettagliata della Figura 4.4.

In tale contesto, l’assegnazione degli indici agli angoli segue una sequenza predefinita conforme alla rappresentazione grafica. Un nodo determina il proprio indice considerando l’angolo più prossimo alla sua posizione. La rotazione rappresenta un nuovo aspetto da considerare, introducendo variazioni di posizione o orientamento nell’ambito del problema in esame. È possibile eseguire un numero infinito di rotazioni variando di conseguenza la corretta costruzione della circonferenza. Inoltre, l’errore di posizione calcolato sulla base della distanza tra l’angolo θ_i e il nodo i non sarà costante, poiché i nodi non hanno una posizione fissa e possono cambiare l’angolo di riferimento ad ogni iterazione.

Ricapitolando, la precisione del calcolo dell’errore di posizione dipende dalla considerazione congiunta della **posizione attesa** (*expected position*), dell’**errore di posizione** (*position error*) e, soprattutto dell’**indice**. Nello specifico, l’informazione relativa alla *posizione attesa* espressa tramite la variabile **node.storage(node_exactExpectedPosition)** indica la posizione dell’angolo più prossimo considerato. Nel contempo, l’*errore di posizione*, rappresentato dalla variabile **node.storage(position_error)** quantifica la distanza tra la posizione attesa e la posizione corrente del nodo. È importante notare che tale calcolo si riferisce ad una specifica rotazione, catturando così le variazioni di posizione nel contesto di tale rotazione particolare.

Errore delle distanze

```
FUN void distanceDevices(ARGS){
    using namespace tags;
```

```

node.storage(effective_dist_master_slave{}) = distance(
get<1>(node.storage(node_posMaster{})), node.position());

node.storage(min_dist_master_slave{}) = min_hood(CALL, mux(
node.nbr_uid() == masterUid && node.storage(node_isSlave{}) &&
get<0>(node.storage(node_posMaster{})), node.nbr_dist(),
(double)communication_range), (double)communication_range);

node.storage(max_dist_master_slave{}) = max_hood(CALL, mux(
node.nbr_uid() == masterUid && node.storage(node_isSlave{}) &&
get<0>(node.storage(node_posMaster{})), node.nbr_dist(), 0.0), 0.0);

node.storage(max_dist_slave_slave{}) = max_hood(CALL, mux(nbr(CALL,
node.storage(node_isSlave{}) && get<0>(node.storage(node_posMaster{}))), 
node.nbr_dist(), 0.0), 0.0);

node.storage(min_dist_slave_slave{}) = min_hood(CALL, mux(nbr(CALL,
node.storage(node_isSlave{}) && get<0>(node.storage(node_posMaster{}))), 
node.nbr_dist(), (2 * distanceMasterSlave * pi)), (2 * distanceMasterSlave * pi));

if (node.storage(node_isSlave{}) && get<0>(node.storage(node_posMaster{}))) {
    tuple<int, vec<2>> t = make_tuple(node.storage(node_indexPosition{}),
                                           node.position());
    field<tuple<int, vec<2>>, f = nbr(CALL, t);

    node.storage(previous_slave_distance{}) = max_hood(CALL, map_hood([] 
(tuple<int, vec<2>> adjacentSlave, int myIndex, vec<2> myPos, int maxIndex) {
        if(get<0>(adjacentSlave) == (myIndex + 1) && myIndex < maxIndex){
            return distance(get<1>(adjacentSlave), myPos);
        } else if(get<0>(adjacentSlave) == 1 && myIndex == maxIndex){
            return distance(get<1>(adjacentSlave), myPos);
        } else{
            return 0.0;
        }
    }, f, node.storage(node_indexPosition{}), node.position(),
    node.storage(node_numberOfSlaves{})), double {});

    node.storage(next_slave_distance{}) = max_hood(CALL, map_hood([] 
(tuple<int, vec<2>> adjacentSlave, int myIndex, vec<2> myPos, int maxIndex) {
        if (get<0>(adjacentSlave) == (myIndex - 1) && myIndex > 1) {
            return distance(get<1>(adjacentSlave), myPos);
        } else if (get<0>(adjacentSlave) == maxIndex && myIndex == 1) {
            return distance(get<1>(adjacentSlave), myPos);
        } else{
            return 0.0;
        }
    }, f, node.storage(node_indexPosition{}), node.position(),
    node.storage(node_numberOfSlaves{})), double {});

} else{
    node.storage(previous_slave_distance{}) = (double)communication_range;
    node.storage(next_slave_distance{}) = (double)communication_range;
}

```

}

Il secondo tipo di errore considerato si basa sulla valutazione dell'errore mediante l'**analisi delle distanze**, in particolare vengono analizzate le distanze estreme (distanza massima e minima) tra *master-slave* e *slave-slave*, la distanza tra un dato *slave* e l'immediato vicino, sempre *slave*, il suo successivo e in maniera speculare il suo precedente.

La situazione ideale è che le distanze massime e minime tra gli elementi siano simili o poco discostate tra loro e tra la distanza desiderata. In altre parole, l'errore è minimo quando la differenza tra la distanza massima e la distanza minima è ridotta. Stesso ragionamento vale per le distanze calcolate tra *slave* immediatamente adiacenti.

Come identifichiamo i nodo slave fisicamente vicini, ovvero il successivo e il precedente rispetto ad uno slave centrale?

Per rispondere alla domanda sfruttiamo gli indici assegnati precedentemente ad ogni slave. Poniamo come centro il nodo con indice n , avremo che in ogni istante vale:

$$prev_n = n - 1, \text{ oppure } prev_1 = \text{numero degli slave}$$

$$next_n = n + 1, \text{ oppure } next_1 = 1$$

4.5 Analisi

Ora, procediamo ad esaminare l'esecuzione pratica del programma per comprenderne il funzionamento in dettaglio. In questa fase, miriamo a condurre un'analisi empirica dell'applicazione per valutarne l'efficacia, l'efficienza e il comportamento in diverse situazioni. Questo processo coinvolge l'esecuzione del programma, l'osservazione dei risultati prodotti e l'interpretazione dei dati raccolti al fine di trarre conclusioni sull'adeguatezza e sulle prestazioni dell'applicazione. Questa analisi pratica è fondamentale per determinare se il programma soddisfa gli obiettivi prefissati e per identificare eventuali aree di miglioramento o correzioni necessarie.

Test

Al fine di condurre una valutazione dettagliata del comportamento del programma, pianifichiamo di eseguire una serie di test sperimentali in cui verranno variati i parametri di configurazione precedentemente definiti. Questi test sperimentali costituiranno un'indagine sistematica delle risposte dell'applicazione alle variazioni dei parametri, consentendo così un'analisi più approfondita del suo comportamento e delle sue prestazioni in diversi contesti. Tale metodologia sperimentale ci permetterà di esaminare in modo critico l'adattabilità del programma alle diverse configurazioni e di raccogliere dati che saranno successivamente utilizzati per trarre conclusioni significative sulla sua affidabilità e sul suo comportamento in situazioni di variazione dei parametri.

4.5.1 Test *Nominal*

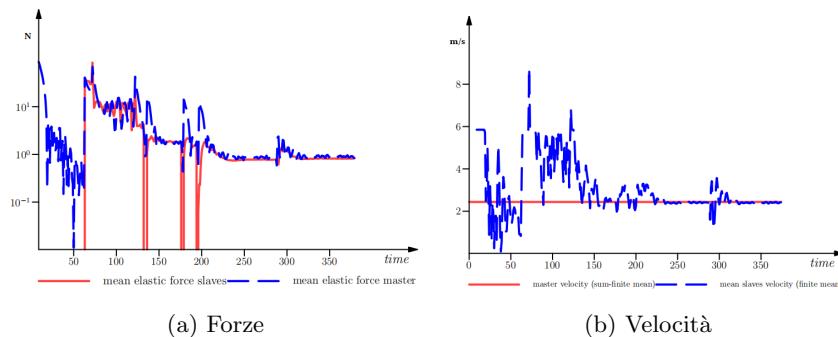


Figura 4.5: *Velocità e forze interagenti in Circular Forces test Nominal*

Identifichiamo gli elementi che saranno soggetti a variazioni rispetto alla configurazione iniziale³, la quale è stata la base di studio per lo sviluppo dell'algoritmo.

Configurazione Nominal:

```
distanceMasterSlave = 50% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

³Configurazione iniziale: enunciata nella sezione 4.3

Esaminiamo attentamente i picchi di forza uguale a $0N$ osservati nel primo grafico, che si originano dalla forza applicata tra i dispositivi *slave*, e cerchiamo di comprendere le ragioni sottese a tali picchi.

Prendiamo in considerazione un picco specifico rilevato durante la simulazione e riportiamo i dati pertinenti registrati in quel momento.

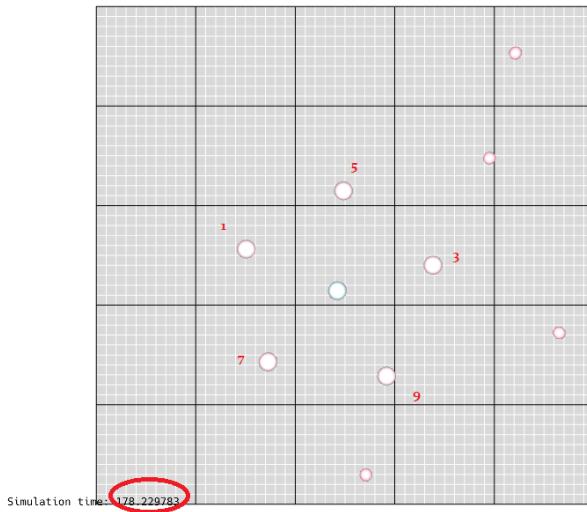


Figura 4.6: Circonferenza equamente divisa durante la simulazione Nominal con Circular Forces

Tabella 4.1: Dati relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato sulle forze applicate nel test Nominal con Circular Forces

Identificativo slave	Elastic force slaves
1	0.00N
3	0.00N
5	0.00N
7	0.00N
9	$0.55 \times 10^{-1}N$

Ricordiamo che, secondo la configurazione iniziale, il nodo *master* è identificato con il valore 0, e i nodi *slave* circostanti sono rappresentati dagli identificativi elencati nella tabella di riferimento.

All'istante preso in considerazione, in figura 4.6, non tutti i 9 *slave* sono parte della formazione ma solo 5 di essi.

Osserviamo che, per la maggior parte dei nodi *slave*, la forza elastica applicata tra di essi risulta essere nulla. Questa circostanza si verifica quando⁴ i nodi *slave* hanno praticamente raggiunto una configurazione ottimale, determinata approssimativamente mediante il calcolo della distanza necessaria per garantire una suddivisione equa della circonferenza.

Prendiamo ora in considerazione ulteriori dati relativi alle distanze calcolate allo stesso tempo di simulazione catturato.

⁴Nel codice: la forza viene applicata solo quando la distanza tra slave è minore di quella calcolata secondo la formula della suddivisione equa della circonferenza.

Tabella 4.2: Dati relativi alle distanze catturati durante un fenomeno di picco grafico rilevato sulle forze applicate nel test Nominal con Circular Forces

Identificativo <i>slave</i>	Effective dist <i>master-slave</i>	Expected dist <i>master-slave</i>	Expected dist <i>slave-slave</i>	Max dist <i>slave-slave</i>	Min dist <i>slave-slave</i>	Next dist <i>slave</i>	Previous dist <i>slave</i>
1	100.47m	100.00m	104.71m	192.53m	113.95m	115.32m	113.91m
3	99.60m	100.00m	104.71m	195.21m	117.16m	117.24m	120.74m
5	100.43m	100.00m	104.71m	188.18m	114.19m	114.10m	116.94m
7	99.81m	100.00m	104.71m	191.60m	115.59m	151.10m	115.64m
9	99.11m	100.00m	104.71m	193.78m	102.43m	120.85m	102.36m

Esaminiamo l' **effective dist master-slave** e notiamo che presenta deviazioni minime con l' **expected dist master-slave**, suggerendo che la forza applicata tra il nodo *master* e i nodi *slave* sta svolgendo efficacemente la sua funzione nel mantenere la distanza desiderata. È fondamentale garantire una calibrazione precisa tra questa forza applicata e le velocità assunte dai dispositivi, poiché variazioni eccessive nella velocità possono comportare un notevole aumento o una drastica diminuzione delle distanze tra il nodo *master* e i nodi *slave*.

Procediamo all'analisi delle distanze tra i nodi *slave*, con particolare attenzione alle distanze minime e le distanze tra un nodo *slave* specifico e quelli posizionati fisicamente vicini: **previous slave** e **next slave**.

Questo approccio si basa sulla constatazione che **con un raggio della circonferenza inferiore al range di comunicazione di almeno il 50%**, le distanze massime prendono in considerazione il fatto che un nodo *slave* può stabilire collegamenti con altri nodi *slave* lontani all'interno della circonferenza. Le distanze massime coinvolgono quindi distanze con nodi cui non vi è una vicinanza fisica all'interno del perimetro della circonferenza o che addirittura possono comprendere distanze con nodi all'interno dello *swarm*, quindi che hanno effettuato la loro prima comunicazione con il nodo *master*, ma che devono ancora raggiungere il perimetro della circonferenza⁵.

Osserviamo che le distanze minime con le distanze tra il **previous slave** e **next slave** rispetto uno specifico *slave* mostrano deviazioni minime tra di loro. Questa coerenza suggerisce una configurazione adeguata per quanto riguarda le interazioni tra i nodi *slave*.

La distanza prevista per numero pari a 5 *slave* all'interno della circonferenza, è all'incirca 125m. Un **errore nelle distanze** si verifica tra il nodo 7 e il nodo 9. In particolare, il valore **next dist slave** del nodo 7 è pari a 151.10m, conseguentemente il valore **previous dist slave** del nodo 9 è di 102.36m.

Nella tabella, la stima della distanza attesa tiene in considerazione l'aggiunta di un nuovo *slave* in procinto di entrare nel perimetro della circonferenza.

L'**errore nelle distanze** in questo caso è identificato dato che la distanza tra il nodo 9 e il nodo 7 risulta essere inferiore rispetto a quanto calcolato tenendo in considerazione l'arrivo del nuovo *slave*. Poiché tale valore è leggermente inferiore alla distanza prevista, la forza viene applicata, seppur in misura modesta.

Nei restanti casi, l' **expected dist slave-slave** non viene considerata, poiché, come già anticipato, questa misura tiene conto di 6 dispositivi *slave* all'interno della circonferenza, nonostante nella figura 2.6 ne siano presenti solo 5. Questa differenza è dovuta all'inclusione di un nuovo nodo *slave* nell'istante preso in considerazione, il quale ha appena stabilito la

⁵**Delay tra l'entrata di un nodo nello swarm e la sua entrata nel perimetro della circonferenza:** Un breve lasso di tempo trascorre tra il momento in cui un agente *slave* entra nello *swarm*, ovvero stabilisce la sua prima comunicazione con il nodo *master*, e il momento in cui viene attivata la forza attrattiva verso il *goal* (→ raggiungere il perimetro della circonferenza).

sua prima connessione con il dispositivo *master* e sta per essere integrato nel perimetro della circonferenza.

Per riassumere, l'assenza o la presenza di forze molto deboli in *Circular Forces* indica che, per qualche ragione, le distanze misurate sono in linea, o superiori, rispetto alle distanze attese.

Andiamo avanti a confermare questa situazione mediante l'acquisizione e l'analisi di ulteriori grafici relativi al calcolo del margine d'errore.

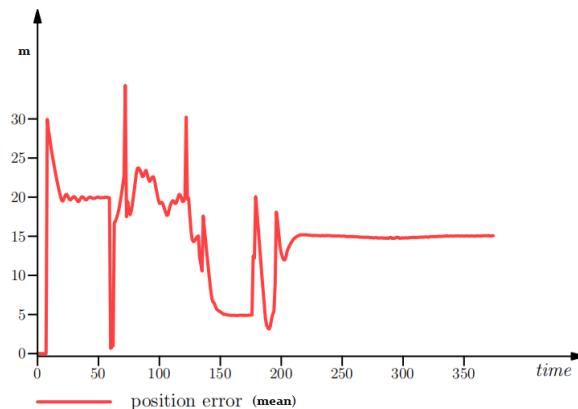


Figura 4.7: *Errore delle posizioni in Circular Forces test Nominal*

È importante tenere presente che l'errore di posizione non tiene conto delle possibili rotazioni che le configurazioni di nodi *slave* potrebbero assumere durante il processo di costruzione rispetto alla configurazione geometricamente ideale calcolata matematicamente. I picchi osservati nel grafico rappresentano in questo contesto una transizione nella rotazione, la quale è causata dall'inserimento di un nuovo dispositivo *slave* all'interno della circonferenza. Quando il valore dell'errore di posizione è vicino allo zero vuol dire che la configurazione dello *swarm* in quell'istante si avvicina alla configurazione perfetta calcolata matematicamente. Al contrario quando il valore dell'errore di posizione è alto vuol dire che la rotazione assunta dallo *swarm* identifica una configurazione lontana rispetto a quella perfetta. L'errore di posizione viene calcolato in base alla distanza tra un dato *slave* e l'angolo di riferimento⁶ preso da lui in considerazione, in relazione all'indice di posizione assegnatogli in un dato istante.

Questa variazione continua fino a giungere a uno stato di stabilità, dovuto dalla cessazione di nuove incorporazioni.

La rotazione della configurazione stabilita al termine del tempo di simulazione preso in considerazione, dista 15m dalla configurazione perfetta calcolata matematicamente.

⁶ **Angolo di riferimento:** l'angolo più vicino in termini di distanza rispetto la posizione del nodo *slave* all'interno del perimetro della circonferenza in un dato istante. La posizione di un nodo *slave* nel perimetro varia nell'istante in cui un nuovo nodo *slave* raggiunge il perimetro della circonferenza

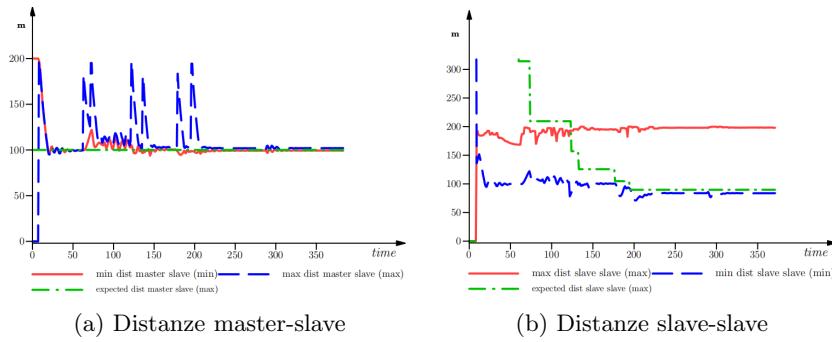


Figura 4.8: *Errore delle distanze in Circular Forces test Nominal*

Per confermare la situazione precedentemente descritta, consideriamo in particolare le *distanza minime evidenziate nei grafici* al momento temporale 178 della simulazione.

Come osservabile nel primo grafico, la misurazione della **distanza minima master-slave** è congruente con la distanza prevista con il nodo *master* e stessa cosa vale nel secondo grafico, la **distanza minima slave-slave** e la distanza attesa tra *slave* coincidono tra loro.

La decisione di adottare questo metodo di valutazione dei dati nel primo grafico è motivata dalla considerevole crescita delle distanza massime al momento in cui un nodo *slave* fuori dalla struttura stabilisce la sua prima comunicazione con i dispositivi all'interno della circonferenza. Per quanto riguarda i picchi nel grafico delle distanze *master-slave*, costituiscono il preciso istante in cui un nuovo nodo *slave* sta per essere incorporato all'interno della circonferenza.

Invece, nel secondo grafico, come già precedentemente esplicato, la distanza massima *slave-slave* tratta la distanza tra *slave* non immediatamente vicini fisicamente: un dispositivo *slave* può stabilire connessioni con ipoteticamente tutti i dispositivi all'interno dello *swarm* quanto il *range di comunicazione* glielo permette.

Al tempo massimo di simulazione considerato in questi grafici, la circonferenza non ha raggiunto la sua completa realizzazione, ovvero incorporando tutti i 10 nodi presenti in simulazione.

Vediamo il risultato finale del test *Nominal*, in cui la circonferenza raggiunge la sua configurazione finale e completa.

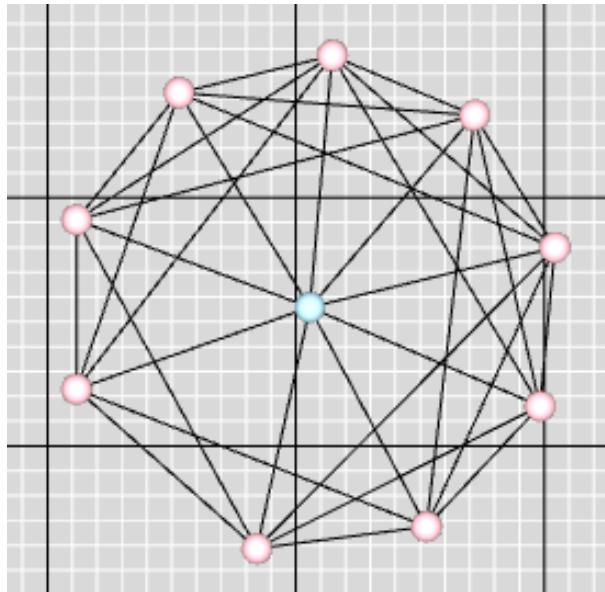


Figura 4.9: *Risultato finale del test Nominal con Circular Forces*

4.5.2 Test *Slow Slaves Always Fast Master*

In questo esperimento, ci concentreremo su un aspetto specifico delle caratteristiche di *Circular Forces*, ovvero la sua adattabilità in uno scenario in cui tutti i dispositivi si spostano alla stessa velocità. Il nostro obiettivo è valutare quanto i dispositivi *slave* siano meno efficienti rispetto al dispositivo *master* quando operano alla stessa velocità. Nell'esperimento precedente, abbiamo osservato che quando il dispositivo *master* viaggia a una velocità inferiore rispetto ai dispositivi *slave*, il sistema funziona senza problemi.

È importante notare che ai dispositivi *slave* viene assegnata un'accelerazione, mentre al dispositivo *master* viene assegnata direttamente una velocità che viene successivamente regolata mediante una riduzione. In questo nuovo test, esploreremo la situazione in cui il dispositivo *master* non è soggetto a limitazioni e si muove alla stessa velocità dei dispositivi *slave*. Prendiamo in considerazione un elemento precedentemente costante, **incrementForce**, che ora verrà manipolata come una variabile al fine di correggere l'errore associato ai dispositivi *slave* rispetto alle velocità applicate.

In seguito, osserveremo come *Circular Forces* sarà in grado di adattare in modo ottimale questa variabile in risposta alle sfide presentate dal test.

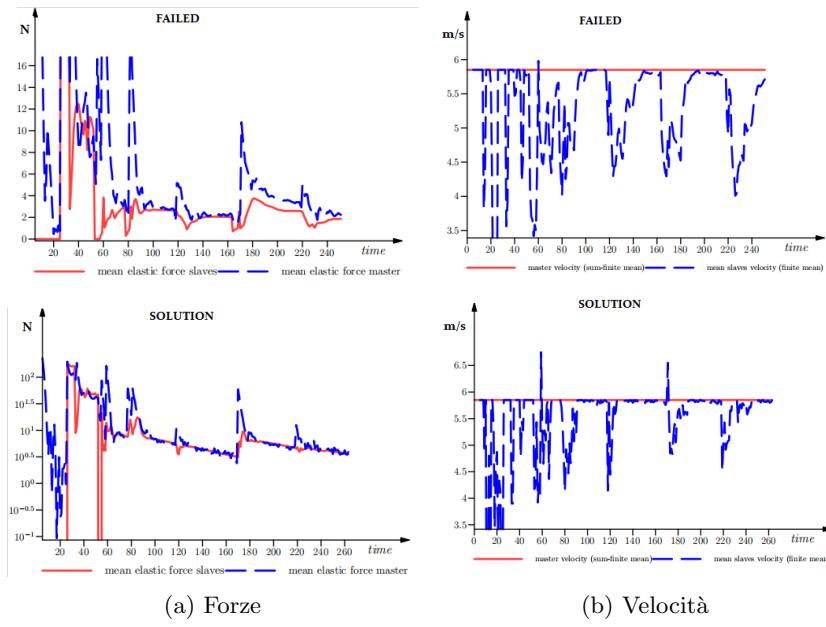


Figura 4.10: *Velocità e forze interagenti in Circular Forces test Slow Slaves Always Fast Master*

Configurazione Slow Slaves Always Fast Master failed:

```
distanceMasterSlave = 50% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.0;

//! Variabile che ci permette di adattare Circular Forces al test
//! Tale assegnazione è quella di default...
incrementForce = 50;
```

```
maxVelocitySlaves = 6;
masterVelocity = 6;
```

Configurazione Slow Slaves Always Fast Master solution:

```
distanceMasterSlave = 50% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.0;

//! Variabile che ci permette di adattare Circular Forces al test
//! Tale assegnazione è quella di adattabilità...
incrementForce = 250;

maxVelocitySlaves = 6;
masterVelocity = 6;
```

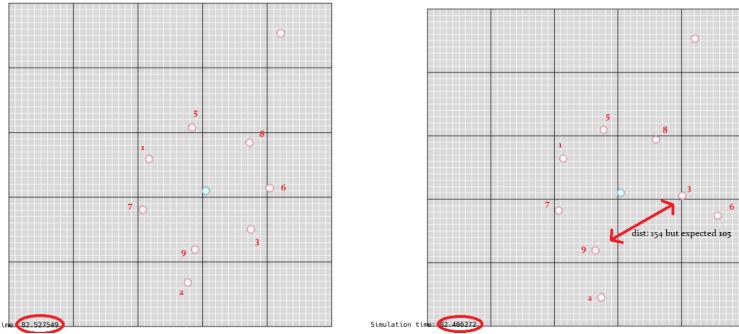
Nella parte superiore della figura 4.10 sono presentati i grafici relativi al test *Slow Slaves Always Fast Master failed*, mentre nella sezione inferiore sono riportati i grafici riguardanti il test *Slow Slaves Always Fast Master solution*.

Nei grafici *failed*, possiamo osservare che le forze applicate sono notevolmente inferiori rispetto a quelle dei grafici *solution*. Inoltre, la velocità dei dispositivi *slave* risulta spesso significativamente inferiore.

I picchi osservati nei grafici delle velocità, generalmente, sono il risultato dell'entrata di un nodo all'interno della circonferenza, in risposta all'attrazione generata dal *goal*⁷, seguita da un adattamento della velocità. È importante notare che questi picchi sono più frequenti e pronunciati nel grafico del test *Slow Slaves Always Fast Master solution* in confronto alle simulazioni precedenti, principalmente a causa dell'applicazione di una forza notevolmente superiore rispetto ai precedenti test.

Questi risultati suggeriscono che nel test *Slow Slaves Always Fast Master failed*, i dispositivi *slave* non riescono a seguire i movimenti del dispositivo *master*.

Analizziamo questo fenomeno con l'utilizzo del simulatore grafico.



(a) Slow Slaves Always Fast Master solution (b) Slow Slaves Always Fast Master failed

Figura 4.11: Adattabilità di Circular Forces nel test Slow Slaves Always Fast Master

È stato identificato il momento temporale nella simulazione prossimo al tempo 82, in cui si verifica una significativa discrepanza tra le forze applicate nei grafici.

Ricordiamo che, come precedentemente spiegato, la forza repulsiva tra *slave*, manifesta la sua influenza solamente quando la distanza tra di essi risulta inferiore a quella prevista in base ai calcoli derivanti da una distribuzione equa su una circonferenza. Analizziamo ora i dati relativi al test *Slow Slaves Always Fast Master* al tempo di simulazione catturato.

⁷Goal (o obiettivo): posizionarsi nel perimetro della circonferenza

Tabella 4.3: Dati relativi alle distanze catturati durante un fenomeno di minimo grafico sulle forze applicate nel test *Slow Slaves Always Fast Master failed con Circular Forces*

Identificativo <i>slave</i>	Expected dist <i>slave-slave</i>	Min dist <i>slave-slave</i>
1	78.53m	77.81m
2 nodo entrante	78.53m	71.13m
3	78.53m	60.19m
5	78.53m	77.83m
6 nodo entrante	78.53m	60.68m
7	78.53m	81.74m
8	78.53m	83.16m
9	78.53m	70.31m

La distanza attesa è calcolata considerando anche i due nuovi nodi *slave* che stanno per entrare nel sistema suddividendo così la circonferenza per 8 *slaves*, anche se all'interno nel dato istante sono presenti 6 *slaves*.

Generalmente, in questo contesto, sarebbe necessario applicare forze in ogni istante significative al fine di mantenere la formazione rispetto al movimento del *master*. Le forze applicate dovrebbero essere amplificate durante l'introduzione di un nuovo dispositivo nel sistema *swarm*, poiché le *distanze preesistenti*, se rispettate, sono suddivise in maniera uniforme lungo il perimetro della circonferenza rispetto al numero degli *slave* in formazione. Il nodo entrante deve competere per stabilire la sua posizione e le corrette distanze tramite l'applicazione di forze.

Successivamente, queste forze dovrebbero propagarsi verso tutti i dispositivi *slave* all'interno della circonferenza, generando un'instabilità temporanea causata dall'integrazione del nuovo dispositivo *slave* nel sistema.

L'andamento delle forze nel grafico del test *Slow Slaves Always Fast Master failed* al tempo 82 non rispecchia le previsioni formulate in base alle considerazioni precedenti, e ciò è attribuibile alle *distanze preesistenti* nel sistema.

Sorge quindi il seguente interrogativo: **perché i nodi *slave* all'interno della circonferenza, ancor prima dell'arrivo dei nuovi *slave*, presentano una distanza minima approssimativamente uguale (o addirittura inferiore) a quella calcolata in base alla nuova distanza attesa tenendo conto dei due nuovi *slave* in arrivo?**

La risposta è che i nodi *slave* fanno fatica a mantenere la formazione rispetto la velocità del *master*, la distanza prevista per un numero di *slave* pari a 6 (non contando i nodi entranti) è 105m all'incirca.

Al fine di ottenere una suddivisione equa della circonferenza nella figura 4.11 b, bisognerebbe basare le distanze minime apportate in tabella con il valore della distanza prevista calcolata pari a 105m. Le distanze minime tra *slave* dovrebbero quindi essere calibrate prendendo come riferimento tale valore, il fatto che siano inferiori indica la presenza di errori nelle distanze.

I dispositivi *slave* si distanziano troppo tra loro in direzione del movimento del *master* e tendono a raggrupparsi nella direzione opposta avvicinandosi troppo tra loro.

Vediamo ora i dati alla versione del test *solution*.

Tabella 4.4: Dati relativi alle distanze catturati durante un fenomeno di minimo grafico sulle forze applicate nel test *Slow Slaves Always Fast Master failed* rispetto alla versione *solution* con *Circular Forces*

Identificativo slave	Expected dist slave	Min dist slave-slave
1	78.53m	77.81m
2 nodo entrante	78.53m	55.41m
3	78.53m	76.30m
5	78.53m	83.48m
6	78.53m	77.41m
7	78.53m	80.00m
8	78.53m	78.82m
9	78.53m	56.18m

Tenendo presente che la distanza attesa su cui basarsi, non conteggiando il nodo entrante, è pari all'incirca a 90 notiamo un margine d'errore inferiore secondo le distanze minime rispetto allo scenario precedente.

Notiamo inoltre che in questo contesto, vi è un solo nodo in fase di ingresso, a differenza della situazione precedente in cui ce n'erano due. Questa variazione è dovuta al fatto che le forze applicate sono di intensità superiore, il che determina una maggiore attrazione del nodo entrante verso il suo *goal*. Un incremento dell'attrazione comporta una maggiore celerità nel raggiungere il perimetro della circonferenza, causando inoltre, quando l'accelerazione è troppo elevata, dei picchi nel grafico della velocità.

Procediamo ora nell'analisi comparativa dei dati, integrando grafici concernenti le distanze in questione.

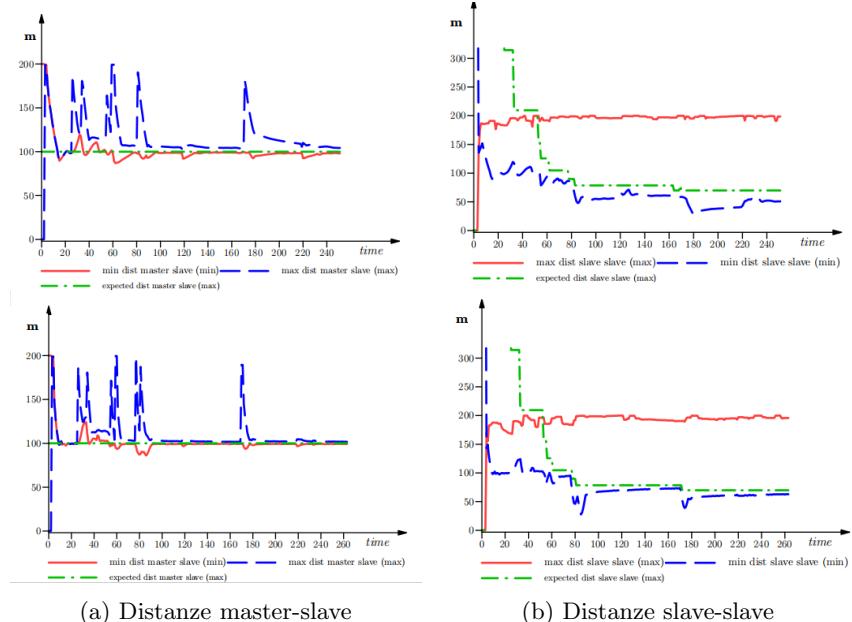


Figura 4.12: Errore delle distanze in Circular Forces test *Slow Slaves Always Fast Master*

Come può essere dedotto dalla porzione inferiore dei grafici relativa al test *Slow Slaves Always Fast Master solution*, i dati dimostrano con maggiore frequenza una vicinanza più marcata alle distanze obiettivo desiderate.

4.5.3 Test *Large Radius*

In questo test, condurremo un'analisi dell'algoritmo *Circular Forces* in un contesto in cui il raggio della circonferenza è notevolmente esteso. Ciò comporterà un adattamento delle dimensioni delle forze applicate.

Risulterà che questo scenario rappresenta l'ambiente in cui l'algoritmo manifesta la sua massima efficienza.

Configurazione Large Radius:

```
distanceMasterSlave = 75% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

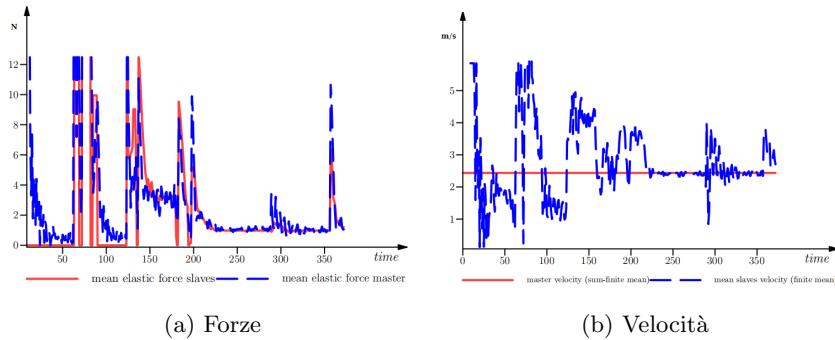


Figura 4.13: *Velocità e forze interagenti in Circular Forces test Large Radius*

Le forze applicate in questa situazione risultano di entità inferiore rispetto al test precedente, suggerendo che, in un contesto caratterizzato da un raggio esteso e, di conseguenza, da ampie distanze tra i dispositivi, è richiesta una quantità minore di forza per mantenere la formazione rispetto al movimento del master.

Dalla visualizzazione del grafico, è possibile notare che l'intervallo in cui le forze si annullano inizialmente è più esteso. In base all'analisi effettuata in precedenza l'intervallo temporale in cui le forze risultano nulle è sempre molto breve.

In questa contesto, l'azzeramento delle forze per un tempo così prolungato rappresenta un periodo in cui i dispositivi *slave* all'interno della circonferenza sono posizionati e non riescono a comunicare tra loro.

Il range di comunicazione, inizialmente, è inferiore alla distanza attesa tra i dispositivi *slave*. Pertanto, inizialmente, i dispositivi *slave* comunicano nel perimetro della circonferenza solo fino a quando il range di comunicazione glielo consente.

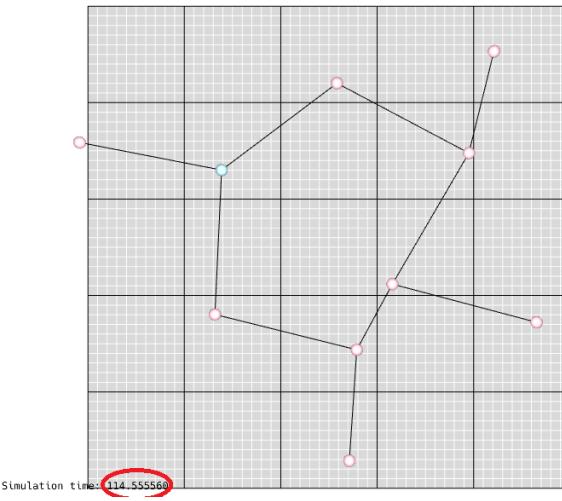


Figura 4.14: Comunicazione al tempo iniziale tra slave in Circular Forces test Large Radius

Come si vede in figura 4.14, i dispositivi *slave* non interagiscono tra loro e quindi non applicano forze repulsive. Al tempo di simulazione catturato la distanza attesa tra *slave* è pari a $314m$ e il *range di comunicazione* configurato è $200m$.

I dispositivi *slave* cominceranno a comunicare con **certezza** tra loro all'interno della circonferenza quando la distanza attesa calcolata tra loro sarà all'incirca il 75% del *range di comunicazione*, quindi con numero di *slave* nello *swarm* pari a 7.

La comunicazione nel contesto di *Circular Forces* è importante al fine di determinare la (*eventuale*) forza repulsiva che gli *slave* esercitano tra loro.

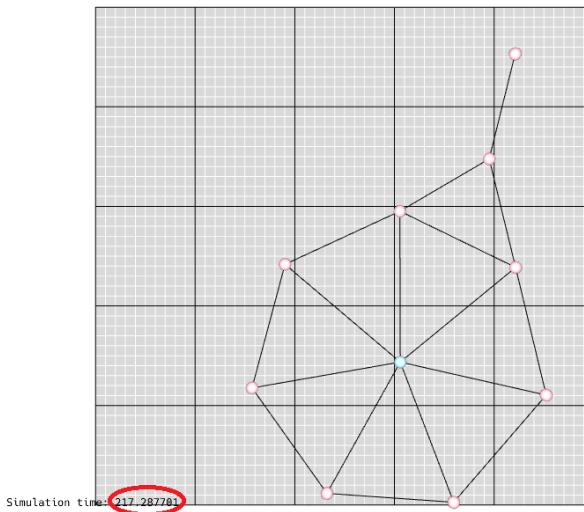


Figura 4.15: Comunicazione ad un tempo successivo tra slave in Circular Forces test Large Radius

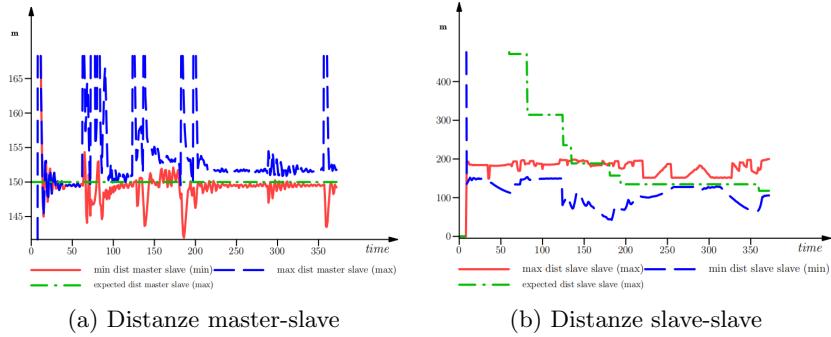


Figura 4.16: *Errore delle distanze in Circular Forces test Large Radius*

Per quanto riguarda la distanza tra il dispositivo *master* e gli *slave*, l'ampio raggio della circonferenza rispetto al *range di comunicazione* non compromette i valori risultanti. Ciò è dovuto al fatto che, per condividere le informazioni relative alle distanze, è sufficiente essere all'interno dello *swarm*, e l'ingresso nello *swarm* avviene mediante la comunicazione diretta con il dispositivo *master*.

Si deduce che all'interno del perimetro della circonferenza, è essenziale instaurare solo la comunicazione con il dispositivo *master*, non importa se i nodi *slave*, al fine della costruzione della formazione, comunichino in ogni istante tra loro. Tuttavia, per garantire una maggiore precisione, è consigliabile che essi stabiliscano una comunicazione con almeno i dispositivi fisicamente vicini a loro lungo il perimetro.

Infine, si può dedurre che nel grafico delle distanze tra i dispositivi *slave*, in questo specifico test, la distanza minima e massima corrispondono alle valutazioni del *previous* e *next slave* oggetto di studio nei test precedenti. Ciò è dovuto al fatto che i nodi *slave* nello *swarm*, quando riescono a comunicare, stabiliscono comunicazioni principalmente solo con i nodi *slave* vicini fisicamente a loro.

Fino a circa il momento temporale 210, i dati delle distanze tra *slave* non possono essere considerati affidabili, poiché la distanza prevista supera il *range di comunicazione* e di conseguenza le comunicazioni tra *slave* non sono costanti. A partire dal momento temporale 210, la distanza prevista è di 134.57m, calcolata attraverso la suddivisione equa della circonferenza in 7 parti. Questo valore di distanza è inferiore al 75% del *range di comunicazione*, limite precedentemente individuato per garantire una connessione affidabile tra i dispositivi. Riassumendo, quando i nodi *slave* instaurano comunicazioni tra di loro, rilevano le distanze e le diffondono; quando la comunicazione cessa, tali distanze vengono ripristinate ai valori predefiniti e vengono utilizzati come riferimento gli ultimi dati rilevati e trattenuti dal nodo *master* per la creazione dei grafici.

4.5.4 Test *Busy Formation*

In quest'ultimo test valuteremo *Circular Forces* nello scenario in cui dovrà disporre numerosi dispositivi *slave* all'interno di un perimetro limitato.

L'applicazione di *Circular Forces* si dimostra inefficace e non è stata individuata una configurazione in cui il suo funzionamento sia adeguato.

Configurazione Busy Formation:

```
distanceMasterSlave = 30% of communication_range;
node_num = 25;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

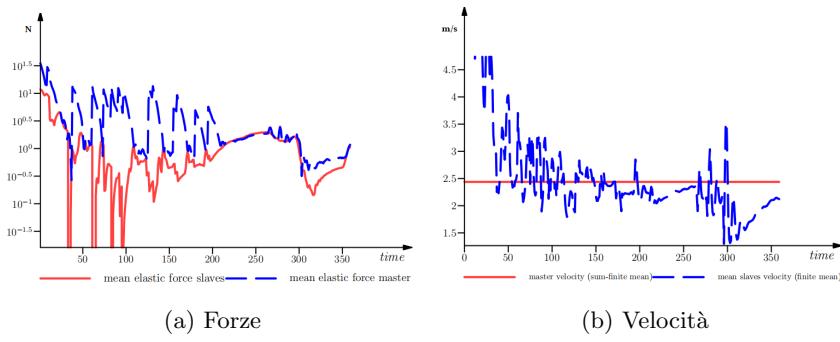


Figura 4.17: Velocità e forze interagenti in Circular Forces test Busy Formation

Fino a quando il numero di dispositivi slave all'interno della circonferenza è limitato, all'incirca fino al tempo 60, l'algoritmo opera in modo congruente.

Sappiamo che i picchi osservati nel grafico delle forze rappresentano il momento antecedente all'introduzione di un nuovo dispositivo *slave* nella formazione. Questo è dovuto al fatto che le distanze effettive tra *slave* calibrate in base alle distanze attese nel periodo immediatamente precedente alla prima comunicazione del dispositivo *slave* entrante, sono certamente superiori rispetto alle nuove distanze previste, a meno che non vi siano errori di distanza. **Ricordiamo che le forze vengono applicate unicamente quando la distanza effettiva è inferiore alla distanza attesa.**

Una volta che il nodo *slave* entrante raggiunge il perimetro, le forze vengono nuovamente applicate in maggiore quantità con regolarità al fine di mantenere la formazione in movimento.

Il primo aspetto da considerare è che, rispetto al grafico delle forze nei precedenti test, nel grafico delle forze del test *Busy Formation* non vi è questa regolarità dal momento 60 in poi. Questo fenomeno indica instabilità riguardo alle forze applicate rispetto al movimento del *master*.

Analizziamo ora un istante di tempo successivo a 160 dove le forze sono applicate costantemente, si presuppone al fine di mantenere la formazione rispetto al movimento del master.

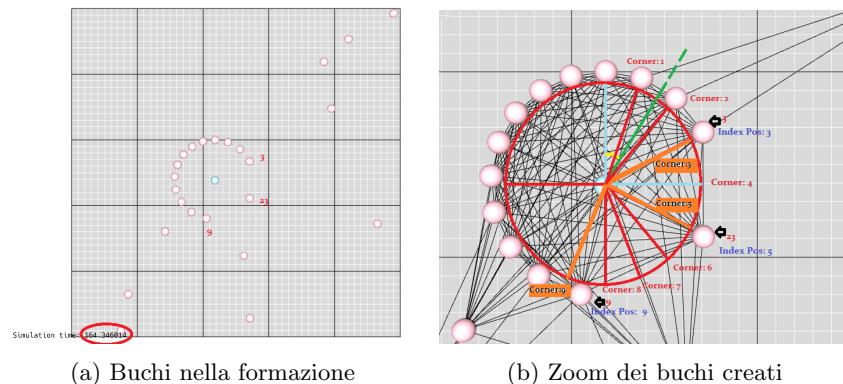


Figura 4.18: Errore nella costruzione della circonferenza in Circular Forces test Busy Formation

Tabella 4.5: Dati relativi alle distanze catturati in presenza di buchi nella formazione nel test Busy Formation con Circular Forces

Identificativo slave	Indice di posizione	Expected dist slave-slave	Next dist slave	Previous dist slave	Min dist slave-slave
3	3	26.91m	23.55m	0.00m	23.55m
23	5	26.91m	0.00m	0.00m	56.17m
9	9	26.91m	0.00m	66.81m	24.97m

La circonferenza è stata suddivisa in modo equo in base al numero di dispositivi *slave* presenti nel perimetro. Vi sono 4 vertici non assegnati, mentre simmetricamente vi è una marcata sovrapposizione di indici nella porzione della circonferenza compatta.

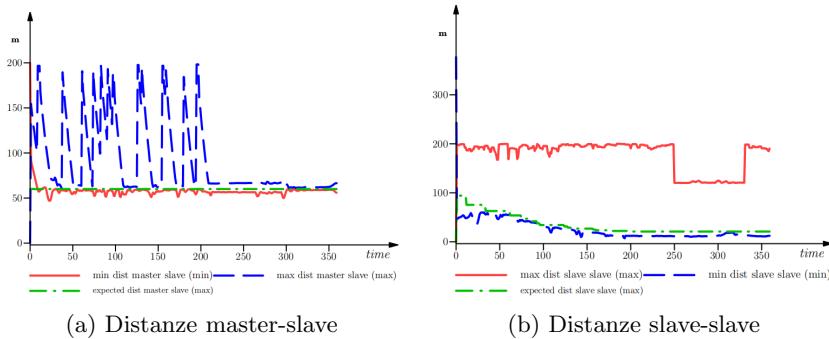


Figura 4.19: Errore delle distanze in Circular Forces test Busy Formation

Osserviamo che la distanza minima *slave-slave* risulta marginalmente inferiore rispetto all'attesa.

In questo contesto, procediamo all'introduzione di un grafico addizionale con l'obiettivo di analizzare il fenomeno della presenza di spazi vuoti lungo il perimetro della circonferenza. In precedenza, abbiamo descritto la valutazione della precisione della circonferenza mediante la determinazione della minima distanza tra i dispositivi *slave* presenti lungo il perimetro. Questa misura della distanza tiene in considerazione le relazioni tra tutti i dispositivi all'interno della circonferenza. Al contrario, la distanza considerata tra un nodo *slave*, il suo predecessore e il suo successore, riflette una situazione specifica di un dato nodo *slave* con i vicini fisici.

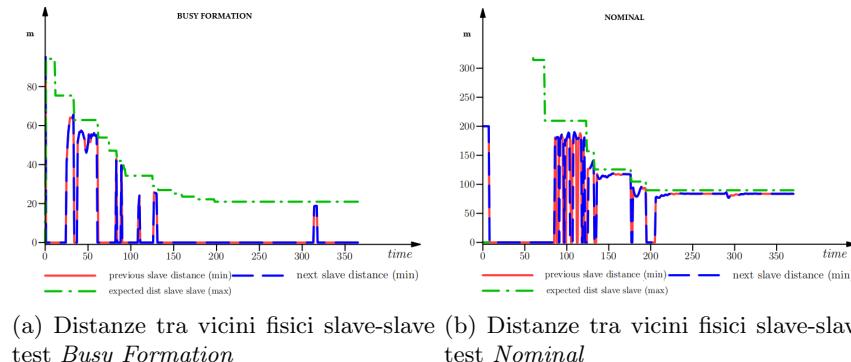


Figura 4.20: Errore delle distanze tra vicini fisici in Circular Forces test Busy Formation

Analizziamo il grafico delle distanze minime tra i dispositivi *previous* e *next slave*. Quando quest'ultime sono pari a 0, come evidenziato nella tabella 4.5, ciò indica l'assenza dei dispositivi vicini fisicamente ad uno specifico *slave*. È stato esaminato precedentemente che questa situazione suggerisce che in base alla suddivisione trigonometrica della circonferenza, sono presenti spazi vuoti nella formazione.

In sé, questa situazione non costituisce un problema fino a quando questi spazi vuoti vengono successivamente occupati. Sappiamo che l'introduzione di un nuovo dispositivo *slave* all'interno della formazione genera temporanei squilibri sotto diversi aspetti.

Eseguiamo un confronto tra il grafico dell'errore delle distanze tra i vicini fisici in due test:

- Test *Nominal*, in cui la circonferenza viene costruita in modo corretto *senza la presenza di buchi* lungo il perimetro. Notiamo che se vi sono buchi questi vengono successivamente riempiti. In questo contesto i buchi rappresentano il momento di entrata di un nuovo dispositivo *slave* nello *swarm*;
- Test *Busy Formation*, in cui la circonferenza viene costruita in modo scorretto *con la presenza di buchi* lungo il perimetro. In questo contesto i buchi non vengono mai riempiti.

4.6 Valutazione conclusiva

Abbiamo osservato che l'algoritmo *Circular Forces* è in grado di adattarsi alla maggior parte degli scenari considerati, mantenendo un grado di precisione accettabile nella costruzione e nel mantenimento della formazione. Ogni test ha rivelato caratteristiche e aspetti distintivi dell'algoritmo. Nel prossimo capitolo, verrà sviluppato un secondo algoritmo che cercherà di ridurre il margine d'errore e ottimizzare il *consumo di energia* dei dispositivi in relazione alla forza applicata per la distanza percorsa.

Capitolo 5

Circular Position

5.1 Introduzione

Intendiamo sviluppare una seconda versione dell'algoritmo *Circular Forces* che utilizzi principi teorici basati sulla trigonometria di una circonferenza e il concetto di potenziale artificiale in contesti differenti, ovvero rispettivamente per la formazione circolare ed un sistema di **Collision Avoidance**.

Nella **formazione circolare**, *Circular Position* sfrutta i principi trigonometrici di una circonferenza nel seguente modo: gli agenti cercano di posizionarsi in modo tale che le loro posizioni calcolate in maniera esatta formino una struttura a forma di circonferenza, impiegando calcoli trigonometrici per determinare gli angoli e le distanze necessarie tra di loro.

Nel sistema di **Collision Avoidance** implementato, *Circular Position* impiega il concetto del potenziale artificiale. In questo caso, il concetto di potenziale artificiale è utilizzato per rilevare la presenza di altri agenti nelle vicinanze e **generare forze di repulsione che impediscono le collisioni durante il raggiungimento di una specifica posizione identificata nel perimetro della circonferenza**. Successivamente, verrà fornita una spiegazione dettagliata di questo meccanismo, evidenziando le differenze rispetto all'algoritmo *Circular Forces* precedentemente introdotto.

5.2 Problema

Questa versione è basata sul **codice di MacroSwarm**.

L'idea di *MacroSwarm* prevede che il nodo *master* calcoli i vettori velocità per i propri nodi *slave*. Questa procedura è spiegata nel seguente modo:

1. **Disposizione ordinata dei nodi slave:** i nodi *slave* vengono disposti in modo ordinato attorno al nodo *master*, ad una distanza R fissa;
2. **Suddivisione degli angoli in radianti:** l'algoritmo divide l'angolo completo di 360° (corrispondente a 2π radianti) in base al numero di nodi *slave* presenti, calcolando così l'angolo da formare con il nodo *master* per ciascun *slave*;
3. **Calcolo del vettore velocità da parte del nodo master:**
 - Il nodo *master* calcola il vettore velocità per ciascun nodo *slave* utilizzando il raggio della circonferenza e l'angolo in radianti precedentemente calcolato;
 - L'ascissa del vettore velocità viene ottenuta moltiplicando il seno dell'angolo per il raggio, mentre l'ordinata viene ottenuta moltiplicando il coseno dell'angolo per il raggio;

- Al vettore ottenuto al passo precedente viene poi sommato il vettore distanza tra il nodo *master* e il nodo *slave*.
4. **Normalizzazione del vettore risultante:** il vettore finale ottenuto dal calcolo viene normalizzato dividendo ciascuna delle sue componenti per la sua norma, risultando in un versore (un vettore di modulo unitario);
5. **Distribuzione delle informazioni sulla rete:** La mappa dei vettori velocità (associando ciascun *slave* al proprio versore velocità) viene distribuita sulla rete in modo che ciascun nodo possa successivamente recuperare il proprio versore velocità.

Considerazioni:

- L'attuale calcolo dei vettori velocità è **centralizzato** sul nodo *master*. È stato riformulato l'algoritmo di *MacroSwarm* in modo che ciascun nodo *slave* calcoli autonomamente il proprio vettore di **accelerazione**, utilizzando la conoscenza della posizione del nodo *master* e altre informazioni rilevanti;
- È importante notare che la condivisione completa dei vettori velocità tra tutti i nodi *slave* può risultare onerosa in termini di comunicazione. Nell'algoritmo proposto ogni nodo *slave* calcolerà autonomamente il proprio vettore di **accelerazione** senza alcuna necessità di sapere quello degli altri dispositivi;
- L'algoritmo include un sistema di **collision avoidance** per garantire che i nodi non si scontrino durante i loro movimenti, migliorando così l'affidabilità della formazione e la sicurezza del sistema.

5.3 Configurazione

Esaminiamo la fase di configurazione del programma proposto per il problema descritto.

/// File: circularPosition_setup.hpp

```
using store_t = tuple_store<
    /// Date without prefix "node_" is used for plot
    node_color,                      color,
    node_size,                        double,
    node_shape,                       shape,
    node_isMaster,                   bool,
    node_isSlave,                     bool,
    node_posMaster,                  tuple<bool, vec<2>>,
    node_numberOfSlaves,             int,
    node_indexSlave,                 tuple<int, int>,
    node_myRadian,                   double,
    node_vecMyRadian,                vec<2>,
    node_vecMyVedor,                 vec<2>,
    node_checkIndex,                 bool,
    node_maxNumberOfSlaves,          int,
    node_secondReturn,               bool,
    node_flagDistance,               bool,
    node_collisionAvoidanceSlaves,   vec<2>,
    node_collisionAvoidanceMaster,   vec<2>,
    node_exactExpectedPosition,      vec<2>,

    /// Notiamo che vengono utilizzati gli stessi dati di
    /// Circular Forces per produrre i grafici di simulazione,
    /// questi dati ci serviranno, oltre che ad analizzare le
    /// singole strategie, al fine di comparare i due
    /// algoritmi di controllo proposti
    position_error,                  double,
    elastic_force_slaves,            double,
    elastic_force_master,            double,
    slaves_velocity,                 double,
    master_velocity,                 double,
    expected_dist_master_slave,     double,
    effective_dist_master_slave,    double,
    max_dist_master_slave,          double,
    min_dist_master_slave,          double,
    expected_dist_slave_slave,       double,
    max_dist_slave_slave,           double,
    min_dist_slave_slave,           double,

    /// next e previous slave sono gli slave
    /// vicini fisicamente ad uno specifico slave
    /// nel perimetro della circonferenza
    next_slave_distance,             double,
    previous_slave_distance,         double
>;
```

/// File: circular_position.hpp

```

///! @brief The maximum communication range between nodes
constexpr size_t communication_range = 200;
///! @brief Constant minimum number of nodes to form a circle
constexpr int minNodesToFormCircle = 5;
///! @brief Number of people in the area.
constexpr int node_num = minNodesToFormCircle + 5;
///! @brief Distance CircularCrown-slave.
constexpr double distanceCircularCrown = communication_range -
                                         ((communication_range / 100) * 30);
///! @brief Distance master-slave.
constexpr double distanceMasterSlave = distanceCircularCrown -
                                         ((distanceCircularCrown / 100) * 50);
///! @brief Constant pi-greco.
constexpr double pi = 3.14159265358979323846;
///! @brief Minimum distance between devices to avoid colliding.
constexpr double minDistance = ((2 * distanceMasterSlave * pi) /
                                 (node_num - 1)) * 0.5;
///! @brief Id master.
constexpr int master_uid = 0;
///! @brief Hardness constant for elastic force Master-Slave.
constexpr double hardnessMasterSlave = (distanceMasterSlave / 10000) * 2;
///! @brief Hardness constant for elastic force CircularCrown-Slave.
constexpr double hardnessCircularCrown = (distanceCircularCrown / 10000) * 2;
///! @brief Hardness constant for elastic force Slave-Slave.
constexpr double hardnessSlaveSlave = 0.01;
///! @brief Max corner for master movement in rectangle
constexpr vec<2> maxCorner = make_vec(0, 0);
///! @brief Min corner for master movement in rectangle
constexpr vec<2> minCorner = make_vec(500, 500);
///! @brief Max value period for master moviment in rectangle
constexpr double maxPeriod = 0.1;
///! @brief Percentage reductor of velocity for devices
constexpr double reductor = 0.025;
///! @brief Percentage reductor of velocity for master
constexpr double reductorMaster = 0.5;
///! @brief Max master velocity
constexpr int masterVelocity = 5;
///! @brief Percentage increment force if distance devices is < minimum distance,
///! high collision risk
constexpr double incrementForce = 50;
///! @brief Max slaves velocity
constexpr int maxVelocitySlaves = 6;
///! @brief Increment slaves angular acceleration
constexpr double incrementAcceleration = 0.0;

```

circularPosition_setup

Gli elementi fino a **node_checkIndex** saranno utilizzati per calcolare e rappresentare la circonferenza. Questi elementi sono fondamentali per il corretto funzionamento dell'algoritmo. Gli elementi da **node_isMaster** a **node_numberOfSlaves** conservano lo stesso significato di quelli con gli stessi nomi precedentemente descritti nella versione *Circular Forces*.

Osserviamo una variazione significativa nella metodologia adottata per definire il nucleo del nostro programma. In precedenza, la circonferenza veniva generata utilizzando principi di forze elastiche, mentre ora viene costruita mediante il **calcolo della posizione precisa** utilizzando l'indice, precedentemente usato per il calcolo dell'errore di posizione, in modo da **decentralizzare** l'approccio al problema.

L'introduzione della variabile **node_checkIndex** è essenziale, poiché il nodo deve attribuire un valore univoco alla sua posizione una volta che è in grado di comunicare con il nodo master. L'indice deve essere univoco al fine di prevenire **collisioni di posizioni**, ovvero situazioni in cui due nodi cercano contemporaneamente di occupare la stessa posizione sulla circonferenza, lasciando spazi vuoti inutilizzati. La variabile **node_checkIndex** è responsabile per il rilevamento di collisioni di indici, e quando questo flag risulterà essere impostato a *true*, l'indice del nodo sarà ricalcolato per risolvere la collisione.

Le variabili **node_maxNumberOfSlaves**, che rappresenta il numero massimo di *slave* costituenti lo *swarm* nella storia della simulazione, e **node_secondReturn**, variabile booleana per identificare un nodo che è uscito dalla circonferenza e intende rientrarci, vengono impiegate per la ricalibrazione dell'indice, sempre tramite l'ausilio della variabile **node_checkIndex**, nel caso in cui la quantità di nodi *slave* all'interno della formazione diminuisca durante l'esecuzione della simulazione.

Il metodo di **collision avoidance** implementato sfrutta diversi elementi:

- **node_flagDistance** rileva le collisioni, segnalando quando un nodo si avvicina ad un altro a una distanza inferiore a quella configurata come minima;
- **node_collisionAvoidanceMaster** e **node_collisionAvoidanceSlaves** determinano le forze coinvolte. Queste variabili sono utilizzate per creare un campo vettoriale, basato sui principi dei potenziali artificiali, allo scopo di guidare il comportamento dei nodi al fine di evitare collisioni, analogamente a quanto avviene nell'algoritmo *Circular Forces*.

circular_position

Nel file **"circular_position"**, vengono dichiarate delle costanti, elencate nell'estratto di codice sopra, che saranno oggetto di modifica al fine di condurre esperimenti e testare il codice. Queste costanti sono essenziali per la fase di inizializzazione e la gestione complessiva del sistema. Inoltre, all'interno di questo file si trova il nucleo centrale dell'algoritmo.

5.4 Descrizione dell'algoritmo

```
//! @brief Main function.
MAIN() {
    using namespace tags;
    node.storage(node_color{}) = color(PINK);

    //! Fase 1
    initialization(CALL);

    //! Fase 2
    node.storage(node_maxNumberOfSlaves{}) = old(CALL, 0, [&](int b){
        return max(b, (node.storage(node_numberOfSlaves{})));
    });

    field <tuple<int, int>> identifierWrongIndex = nbr(CALL,
        node.storage(node_indexSlave{}));
}
```

```

        bool flagIndex = all_hood(CALL, map_hood([])
            (tuple<int, int> t, tuple<int, int> myValue) {
                if (get<1>(myValue) == get<1>(t) && get<0>(myValue) != get<0>(t)) {
                    return false;
                }else{
                    return true;
                }
            }, identifierWrongIndex, node.storage(node_indexSlave{}));

        if(!(flagIndex && decrementIndex(CALL))) {
            checkIndex(CALL);
        }else{
            node.storage(node_checkIndex{}) = false;
        }

        if (node.storage(node_isSlave{})) {

            //! Fase 3
            calculateMyCorner(CALL);

            //! Fase 4
            collisionAvoidance(CALL);

            //! Fase 5
            errorCalculator(CALL);
        }

        /// Stabilizzazione delle forze interagenti attraverso l'uso
        /// di un attrito simulato
        if(node.velocity() > 0){
            node.velocity() -= node.velocity() * reductor;
        }

        //! Continuazione fase 5
        distanceDevices(CALL);
        if(node.storage(node_isSlave{})) {
            node.storage(slaves_velocity{}) = norm(node.velocity());
        }else{
            node.storage(master_velocity{}) = norm(node.velocity());
        }
    }
}

```

Il programma può essere suddiviso in cinque fasi distinte:

- 1. Inizializzazione;**
- 2. Controllo dell'indice;**
- 3. Calcolo della posizione esatta;**
- 4. Applicazione delle forze;**
- 5. Calcolo del margine d'errore.**

5.4.1 Inizializzazione

La selezione del master e l'assegnazione iniziale dei parametri relativi alla sua posizione e al numero degli *slave* all'interno della circonferenza avvengono secondo il medesimo approccio impiegato nella versione precedente.

Ci concentriamo quindi sull'inizializzazione dell'indice.

```

FUN void initialization(ARGS) {
    // ...
    //! Inizializzazione dell'indice
    field <tuple <bool, vec <2>>> identifierMaster = nbr(CALL, t);
    tuple<bool, vec<2>> identified = max_hood(CALL, identifierMaster);
    if (get<0>(identified)) { //! Inizio zona critica
        node.storage(node_posMaster{}) = identified;
        /**Quando un nodo identifica la posizione del master gli viene
         * assegnato un indice che equivale all'ordine di arrivo all'interno
         * della circonferenza*/
        if(!(node.storage(node_secondReturn{}))) {

            //! Costruzione iniziale della circonferenza
            int maxIndex = nbr(CALL, 0, [&](field<int> indexes) {
                int maxIndex = max_hood(CALL, indexes);
                if (get<1>(node.storage(node_indexSlave{})) == 0
                    && node.storage(node_isSlave{}) {
                    return maxIndex + 1;
                } else {
                    return maxIndex;
                }
            });
            if ((get<1>(node.storage(node_indexSlave{}))) == 0
                && node.storage(node_isSlave{}) {
                node.storage(node_indexSlave{}) = make_tuple(node.uid, maxIndex);
            }
        }else{

            //! Ricostruzione della circonferenza
            if ((get<1>(node.storage(node_indexSlave{}))) == 0
                && node.storage(node_isSlave{}) {
                get<1>(node.storage(node_indexSlave{})) =
                    node.storage(node_numberOfSlaves{});
            }
        }
    } //! Fine zona critica

    if (node.storage(node_isMaster{})) {
        rectangle_walk(CALL, minCorner, maxCorner, masterVelocity, maxPeriod);
        if(node.velocity() > 0){
            node.velocity() -= node.velocity() * reductorMaster;
        }
    }
}

```

L’attribuzione dell’indice all’interno del contesto di *Circular Position* riveste un ruolo fondamentale per il corretto funzionamento dell’algoritmo di controllo. Per queste ragioni, la sezione del codice in cui si verifica questa operazione è identificata come **sezione critica**. Ciò è dovuto alla possibilità di errori di assegnamento. Affronteremo e gestiremo questa eventualità in una fase successiva.

Cosa succede all’interno della sezione critica?

All’interno della regione critica, vengono gestite due casistiche distinte, tramite la variabile **node_secondReturn**, ciascuna correlata ad uno specifico stato in cui si trova lo *swarm*:

1. **Costruzione iniziale della circonferenza:** ogni *slave* all’interno del sistema, comunica l’indice massimo attualmente presente. Nel caso in cui ci sia un nuovo *slave*, cioè un nodo privo di un indice che si unisce al sistema, si riserva l’assegnazione dell’indice successivo al valore massimo già in uso tra gli *slave* presenti nella circonferenza;
2. **Ricostruzione *n-esima* della circonferenza:** in questa situazione, è importante notare che la circonferenza è stata precedentemente configurata e che il numero attuale di *slave* con cui il *master* comunica è determinato dall’inclusione del nodo *slave* che sta rientrando in quel momento. Pertanto, nessun altro *slave* già presente nella circonferenza avrà quel numero specifico, che viene riservato ed assegnato al nodo *slave* che sta rientrando nel gruppo.

Successivamente, esamineremo questa suddivisione per valutare la sua efficacia pratica.

Movimento del Master

Il movimento del *master* è determinato tramite l’impiego della funzione di libreria **rectangle_walk**, seguendo lo stesso approccio adottato nella versione *Circular Forces*.

5.4.2 Controllo dell’indice

In questa fase, vengono affrontate le anomalie derivanti dall’assegnazione iniziale degli indici. Qui di seguito è riportato nuovamente il frammento di codice all’interno del *main* adoperato per affrontare le possibili problematiche.

```

node.storage(node_maxNumberOfSlaves{}) = old(CALL, 0, [&](int b){
    return max(b, (node.storage(node_numberOfSlaves{})));
});

//! Rilevazione dell’errore di indice
field <tuple<int, int>> identifierWrongIndex = nbr(CALL,
    node.storage(node_indexSlave{}));
bool flagIndex = all_hood(CALL, map_hood([
    (tuple<int, int> t, tuple<int, int> myValue) {
        if (get<1>(myValue) == get<1>(t) && get<0>(myValue) != get<0>(t)) {
            return false;
        } else{
            return true;
        }
    },
    identifierWrongIndex, node.storage(node_indexSlave{})));

//! Rilevazione della casistica per la gestione dell’errore di indice
if(!(flagIndex && decrementIndex(CALL))) {
    //! Correzione dell’errore di indice
}

```

```

        checkIndex(CALL);
    }else{
        node.storage(node_checkIndex{}) = false;
    }
}

```

Possiamo suddividere questa fase in ulteriori sottofasi:

- **Rilevazione dell'errore di indice;**
- **Rilevazione della casistica per la gestione dell'errore;**
- **Correzione dell'errore di indice.**

Rilevazione dell'errore di indice

I nodi *slave* comunicano il proprio indice ai nodi adiacenti al fine di verificarne l'unicità. Per effettuare questa operazione, si adotta una logica di ***AND***, in cui, se esiste una coppia di valori $<id\ slave;\ indice\ slave>$ per cui il mio indice è uguale a quello del vicino, ma il mio id è diverso da quello del vicino, si deduce che due nodi distinti condividono lo stesso indice. In questo caso, entrambi i nodi restituiranno un valore ***false*** come esito della verifica sull'assenza di duplicati. Verrà deciso successivamente quale dei due nodi collisi dovrà cambiare il proprio indice. Questo risultato, memorizzato nella variabile *flagIndex*, rappresenta uno dei due componenti per avviare il processo di ricalcolo dell'indice.

Rilevazione della casistica per la gestione dell'errore di indice

```

FUN bool decrementIndex(ARGS){
    using namespace tags;

    //! Determinazione casistica
    int difference = node.storage(node_maxNumberOfSlaves{}) -
                     node.storage(node_numberOfSlaves{});
    if (get<1>(node.storage(node_indexSlave{})) > node.storage(node_numberOfSlaves{}) &&
        && difference > 0) {

        //! Caso della ricostruzione n-esima della circonferenza
        get<1>(node.storage(node_indexSlave{})) = 1;
        return false;
    }
    //! Caso della costruzione iniziale della circonferenza
    return true;
}

```

Se la differenza impostata nel codice è maggiore di zero, significa che alcuni *slave* sono usciti dal sistema. In questo caso, dobbiamo considerare di aggiornare l'indice attuale degli *slave* rimasti, ma lo faremo solo se l'indice corrente supera il numero massimo indice consentito, che corrisponderà al numero di *slave* rimasti all'interno della circonferenza. L'aggiornamento dell'indice comporta il ripristino del valore iniziale 1. In seguito, un nuovo indice valido sarà assegnato da un processo di ricalcolo dell'indice. È importante notare che il nodo non ha informazioni sulle posizioni libere nella circonferenza e, pertanto, dovrà cercare in tutto il perimetro per trovare uno spazio vuoto, questo è il motivo per cui l'eventuale aggiornamento dei nodi *slave* restanti parte dall'indice minimo consentito. Tale fenomeno è particolarmente evidente quando il numero di *slave* è elevato.

Ora abbiamo entrambi gli elementi per determinare se avviare una procedura di ricalcolo dell'indice. Vediamo nel dettaglio i 4 casi in cui possiamo ricadere:

Tabella 5.1: *Logica del ricalcolo dell'indice*

Logica	Ricalcolo	Descrizione
<i>flagIndex == false AND decrementIndex == true</i>	Si	Siamo nella fase di costruzione iniziale e un nodo, nella fase di inizializzazione, ha configurato il suo indice con un valore uguale a quello di un altro nodo: collisione di indici
<i>flagIndex == true AND decrementIndex == false</i>	Si	Siamo nella fase di ricostruzione e un nodo che eccede il valore massimo è stato portato al valore iniziale
<i>flagIndex == true AND decrementIndex == true</i>	No	Il nodo ha un indice univo. Ciò implica che ha trovato una posizione univoca da occupare
<i>flagIndex == false AND decrementIndex == false</i>	Si	Il nodo nella fase di costruzione iniziale si trova nella situazione di collisione di indici , nel mentre il gruppo all'interno della circonferenza diminuisce e ricadiamo nella seconda casistica

Correzione dell'errore di indice

```

FUN void checkIndex(ARGS){
    using namespace tags;
    if(node.storage(node_isSlave{})) {
        if ((get<1>(node.storage(node_indexSlave{}))) > 0) {
            field <tuple<int, int>> f = nbr(CALL, node.storage(node_indexSlave__));
            tuple<int, int> check = max_hood(CALL, f);
            if (get<1>(node.storage(node_indexSlave{})) == get<1>(check) &&
                get<0>(node.storage(node_indexSlave{})) != get<0>(check)) {
                node.storage(node_checkIndex{}) = true;
                if (!(node.storage(node_secondReturn{}))) {
                    get<1>(node.storage(node_indexSlave{})) += 1;
                }else{
                    get<1>(node.storage(node_indexSlave{})) -= 1;
                }
            }
        }
    }
}

```

Si effettua una verifica preliminare per determinare se un nodo detiene il ruolo di *slave*. Questo poiché il *master* non è dotato di un indice, in quanto tale informazione non è rilevante per il suo funzionamento.

Dopo aver confermato la sua condizione di *slave*, il nodo comunica il proprio indice attuale a tutti gli altri *slave* con cui è in grado di stabilire una comunicazione. Successivamente, procede all'identificazione del valore massimo tra gli indici condivisi.

Nel momento in cui uno *slave* rileva la presenza di un indice duplicato rispetto ad un altro, attiva il *flag di ricalcolo*, denotato dall'elemento **node.storage(node_checkIndex)**, al fine di avviare la procedura di ricalcolo. Pertanto, se entrambi gli *slave* rilevano la collisione degli indici durante il passo precedente e si trovano a questo punto allo stesso momento, il nodo con l'identificativo più alto sarà responsabile di modificare il proprio indice.

Osserviamo ora la distinzione pratica tra le due casistiche delineate: **costruzione iniziale della circonferenza, ricostruzione n-esima della circonferenza**. Tuttavia, è importante specificare che nella circostanza di *ricostruzione n-esima della circonferenza*, si riscontrano due tipi di nodi:

1. Il primo tipo è rappresentato da nodi che rimangono all'interno del sistema ma il cui indice viene ripristinato al valore iniziale. Per questi nodi, il processo di ricalcolo segue un procedimento "standard";
2. Il secondo tipo comprende nodi che ritornano nel sistema dopo essere stati temporaneamente assenti. Questi nodi seguono una metodologia inversa rispetto a quella standard nel processo di ricalcolo.

Il processo di ricalcolo standard, utilizzato oltre per la *ricostruzione n-esima della circonferenza* anche nella casistica della *costruzione iniziale della circonferenza*, può essere definito come una procedura in cui l'indice di un nodo che ha subito una collisione viene incrementato di una unità ad ogni ciclo. Il nodo coinvolto percorre sistematicamente gli indici all'interno della circonferenza uno per uno, al fine di individuare un nuovo indice disponibile. Questo procedimento si basa sull'assunzione che l'indice libero cercato sia situato a una posizione successiva rispetto all'indice precedentemente coinvolto nella collisione.

In contrasto, il processo di ricalcolo non standard, utilizzato solo nella casistica di *ricostruzione n-esima della circonferenza*, adotta l'approccio opposto. Ciò è dovuto al fatto che, in questa casistica, anche l'assegnazione iniziale dell'indice è effettuata in modo antitetico, come abbiamo precedentemente visto nella fase di inizializzazione.

5.4.3 Calcolo della posizione esatta

Riprendiamo i concetti precedentemente introdotti nella sezione 4.4.3 per la suddivisione equa della circonferenza.

```

FUN void calculateMyCorner(ARGs){
    using namespace tags;
    if(get<0>(node.storage(node_posMaster{}))) {

        //! Calcolo diretto della posizione esatta occupata
        double myRadian = (get<1>(node.storage(node_indexSlave{}))) *
                          ((2 * pi) / node.storage(node_numberOfSlaves__));
        node.storage(node_myRadian{}) = myRadian;
        double x = std::sin(myRadian) * distanceMasterSlave;
        double y = std::cos(myRadian) * distanceMasterSlave;

        //! Introduzione della corona circolare come posizione di
        //! appoggio per il ricalcolo dell'indice
        if (node.storage(node_checkIndex{})) {
            x = std::sin(myRadian) * distanceCircularCrown;
            y = std::cos(myRadian) * distanceCircularCrown;
        }
    }
}

```

```

    }

    vec<2> vecMyRadiant = make_vec(x, y);
    node.storage(node_vecMyRadiant{}) = vecMyRadiant;

    ///! Traslazione del cerchio attorno al master
    vec<2> dist = get<1>(node.storage(node_posMaster{})) - node.position();
    vec<2> versore = vecMyRadiant + dist;
    node.storage(node_vecMyVorsore{}) = versore;

    if (!(isnan(versore[0]) && isnan(versore[1]))) {

        ///! Accelerazione angolare
        node.propulsion() = versore/norm(versore);
    }

    ///! Incremento dell'accelerazione posizionale impostata sempre a 0, tranne
    ///! che in uno specifico scenario che tratteremo successivamente ...
    node.propulsion() += node.propulsion() * incrementAcceleration;
}

}

```

Una volta identificata con precisione la posizione che deve essere occupata da un nodo *slave*, ovvero aver identificato un’assegnazione di un indice univoco e corretto, il nodo procederà a calcolare la sua posizione relativa e l’angolo rispetto al *master* utilizzando il valore dell’indice come valore costante che non cambia fino a che quest’ultimo è univoco e corretto. Di conseguenza, il nodo manterrà costantemente la medesima posizione, data da un calcolo esatto, all’interno della circonferenza durante l’intera simulazione, eliminando così il fenomeno delle rotazioni.

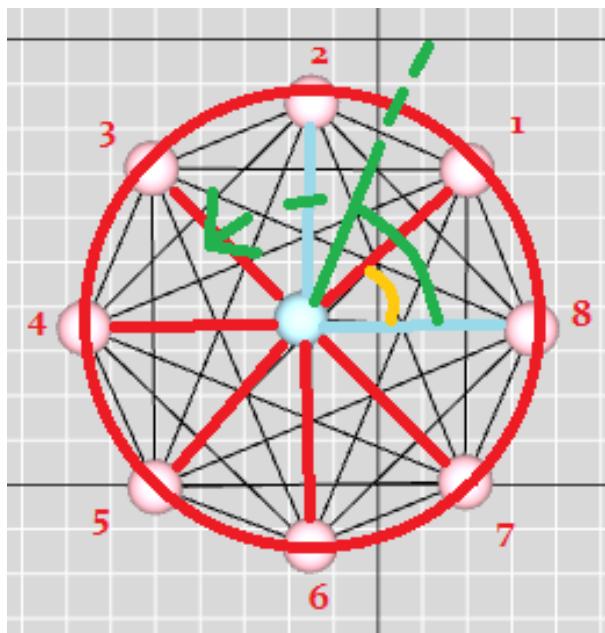


Figura 5.1: Circonferenza formata da Circular Position rispetto ad una formazione teorica

Notiamo la precisione con cui i nodi *slave* si posizionano rispetto al nodo *master*.

Traslazione della circonferenza attorno al master

Attraverso il **calcolo diretto della posizione esatta** (*vedere il riferimento nel codice subito sopra*), si determina la misura in *radiani* dell'angolo formato tra uno *slave* e l'origine degli assi cartesiani.

Con traslazione della circonferenza attorno al master si intende che facciamo in modo tale che l'origine degli assi sia la posizione del nodo *master*.

Il master è considerato quindi l'origine di un sistema di assi cartesiani "virtuale" (*disegnato in azzurro nella figura 5.1*). Facciamo ora un esempio pratico.

Tabella 5.2: *Esempio di applicazione del meccanismo di traslazione della circonferenza*

Id slave	Coordinate di posizione nel perimetro calcolate
1	[47.55; 15.45]
2	[-0.12; 50]
3	[-47.55; 15.45]
4	[-29.38; -40.45]
5	[29.38; -40.45]

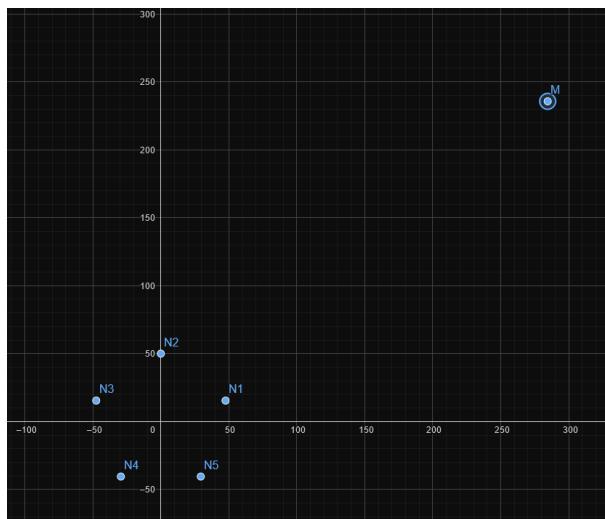


Figura 5.2: *Rappresentazione grafica del meccanismo di traslazione della circonferenza su un sistema di assi cartesiani*

Sommendo al vettore radiente la posizione del *master*, viene creato un nuovo sistema "virtuale" di coordinate in cui il *master* rappresenta l'origine.

Come viene detto al nodo slave di raggiungere la posizione risultante?

Tramite un vettore unitario, nel codice chiamato **versore**, che indica la direzione verso la quale il nodo deve muoversi e la lunghezza del percorso per raggiungere la destinazione, tenendo conto della posizione corrente rispetto la posizione del *master* e dell'angolo calcolato.

Introduzione della corona circolare come posizione di appoggio per il ricalcolo dell'indice

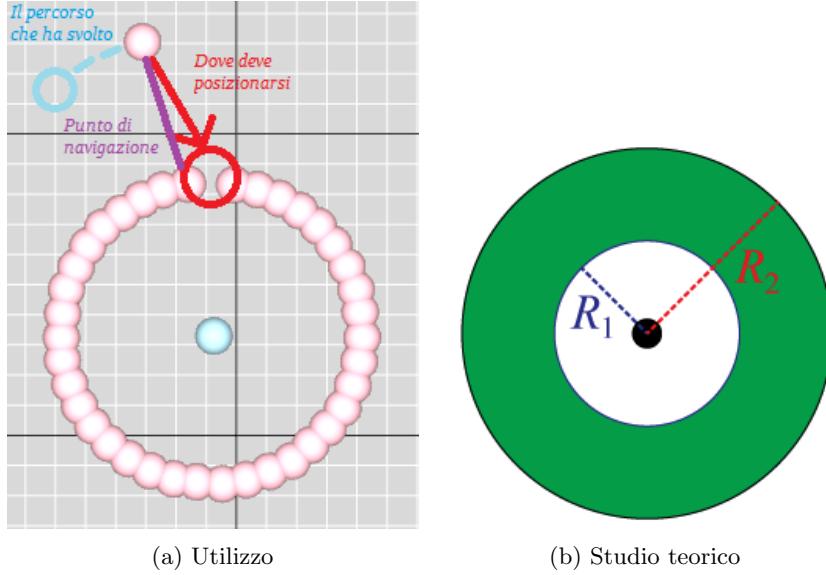


Figura 5.3: *Concetto di Corona Circolare*

Nella fase precedente, abbiamo discusso del processo di ricalcolo dell'indice, nel quale un nodo *slave* esplora tutti gli indici lungo la circonferenza. Questo implica che il nodo *slave* si sposta gradualmente da una posizione all'altra lungo la circonferenza ricoprendole tutte. Questo procedimento potrebbe portare a significative sovrapposizioni durante il ricalcolo del nodo *slave* coinvolto in una collisione di indici.

Per evitare sovrapposizioni viene utilizzato il concetto di **Corona Circolare**, in cui il nodo risiede fino a che non trova una posizione libera all'interno della circonferenza principale, ovvero fino a che non acquisisce un valore univoco dell'indice.

I calcoli sono analoghi a quelli precedentemente introdotti, semplicemente viene impostato un **raggio** differente.

5.4.4 Applicazione delle forze

```
FUN void collisionAvoidance(ARGs){
    using namespace tags;
    if (node.storage(node_isSlave{})) {
        if(get<0>(node.storage(node_posMaster{}))) {
            bool flag = compare(distance(get<1>(node.storage(node_posMaster{})),
                                         node.position()), distanceDevices);
            vec<2> elasticMaster = make_vec(0, 0);
            vec<2> v = get<1>(node.storage(node_posMaster{})) - node.position();
            if (!node.storage(node_checkIndex{})) {
                elasticMaster = v * (1 - distanceMasterSlave / (norm(v)))
                               * hardnessMasterSlave;
            } else {
                elasticMaster = v * ((1 - distanceCircularCrown / (norm(v))))
                               * hardnessCircularCrown;
            }
        }
    }
}
```

```

    //! Distanza minima fissa costante che i dispositivi devono
    //! mantenere sempre
    if(flag){
        elasticMaster += elasticMaster * incrementForce;
    }
    node.storage(node_collisionAvoidanceMaster{}) = elasticMaster;
    node.storage(elastic_force_master{}) = norm(node.storage(
        node_collisionAvoidanceMaster{}));

    //! Viene considerata come limite critico il 90% della suddivisione
    //! in parti uguali per permettere un margine d'errore del 10%
    double distanceSlaveSlave = (2 * distanceMasterSlave * pi) /
        node.storage(node_numberOfSlaves{});
    tuple<bool, vec<2>> elasticSlave = sum_hood(CALL, map_hood([])
    (vec<2> v, double d, double l, double constAvoid) {
        tuple<bool, vec<2>> t = make_tuple(false, make_vec(0, 0));
        double criticalLimit = l * 0.9;
        if (d < criticalLimit) {
            get<1>(t) = v * ((1 - 1 / (norm(v))) * hardnessSlaveSlave);

            //! Vi è un ulteriore livello di criticità individuato
            //! da una distanza fissa costante minima che i nodi devono
            //! sempre mantenere fra loro
            if(d < constAvoid){
                get<1>(t) += get<1>(t) * incrementForce;
            }
        }
        return t;
    }, node.nbr_vec(), node.nbr_dist(), distanceSlaveSlave, distanceDevices),
    tuple<bool, vec<2>>{});
    node.storage(node_collisionAvoidanceSlaves{}) = get<1>(elasticSlave);
    node.storage(elastic_force_slaves{}) = norm(node.storage(
        node_collisionAvoidanceSlaves{}));
    if(node.storage(node_collisionAvoidanceSlaves{}) != make_vec(0, 0)) {
        node.storage(node_flagDistance{}) = true;
    }else{
        node.storage(node_flagDistance{}) = false;
    }

    node.propulsion() += node.storage(node_collisionAvoidanceMaster{});
    node.propulsion() += node.storage(node_collisionAvoidanceSlaves{});
}
}
}

```

Come anticipato nell'introduzione di questo capitolo, il meccanismo in questione si basa sul medesimo principio fondamentale del **potenziale artificiale**, con l'introduzione delle stesse forze interagenti spiegate precedentemente nel contesto delle *Circular Forces*.

Collision Avoidance è visto come un'estensione del concetto di **Obstacle Avoidance**¹. Questo meccanismo è visto come tale poiché viene implementato attraverso l'uso di regioni

¹Obstacle Avoidance: descritto nella sezione 4.4.2

di attivazione rispetto al precedente.

Regione di attivazione

È possibile suddividere la regione di attivazione in due strati, che definiremo come **livelli di criticità**.

- **Livello 1 di criticità:** il primo livello è strutturato in base ad un *intervallo di tolleranza* per gli errori consentiti nelle misurazioni delle distanze tra i dispositivi. In particolare, in relazione alla distanza tra il nodo *master* e gli altri nodi *slave*, supponiamo sia necessario rispettare una precisione assoluta, quindi con un margine d'errore pari a 0, conseguentemente il *livello 1 di criticità* è impostato allo 0% rispetto alla distanza richiesta tra *master* e *slave* impostata durante la configurazione. Al contrario, tra i nodi *slave* è consentito un margine d'errore modesto. Questo compromesso è necessario poiché la disposizione dei nodi *slave* viene ricalibrata ad ogni round e può sperimentare notevoli fluttuazioni, spesso causate dall'inclusione di un nuovo nodo *slave* all'interno dello *swarm*. Ciò può portare ad oscillazioni significative che richiedono una maggiore tolleranza agli errori, nello specifico tra gli *slave* il *livello 1 di criticità* è impostato al 10% rispetto alla distanza che devono mantenere per formare una circonferenza equamente divisa.
- **Livello 2 di criticità:** viene fissata una distanza minima come vincolo che deve essere rispettato costantemente tra gli agenti. Questa distanza minima è definita in modo tale da essere inferiore alla distanza più breve possibile tra gli agenti, che si verifica quando essi sono posizionati ad una distanza uniforme lungo una circonferenza completa, con esattamente $n - 1$ nodi *slave* in formazione. Per garantire il rispetto di questa distanza minima, viene introdotta una variabile di incremento della forza.

5.4.5 Calcolo del margine d'errore

Il margine d'errore è individuato analogamente a come avviene in *Circular Forces*, ovvero tramite l'utilizzo di due differenti approcci: **Errore delle posizioni** e **Errore delle distanze**.

Esamineremo ora le modifiche apportate al calcolo dell'errore di posizione in merito alla differente logica adottata per la costruzione della circonferenza.

Errore delle posizioni

```
FUN void errorCalculator(ARGS){  
    using namespace tags;  
    if(get<0>(node.storage(node_posMaster{}))) {  
        double myRadian = (get<1>(node.storage(node_indexSlave{}))) * ((2 * pi) /  
            node.storage(node_numberOfSlaves{}));  
        double x = std::sin(myRadian) * distanceMasterSlave;  
        double y = std::cos(myRadian) * distanceMasterSlave;  
        vec<2> vecMyRadian = make_vec(x, y);  
  
        vec<2> exactPosition = vecMyRadian + get<1>(node.storage(node_posMaster{}));  
        double error = distance(exactPosition, node.position());  
  
        node.storage(node_exactExpectedPosition{}) = exactPosition;  
        node.storage(position_error{}) = error;  
    }  
}
```

In precedenza, era necessario determinare la posizione occupata da un nodo *slave*, mentre adesso disponiamo di una posizione univoca derivante dai passi precedenti. L'errore di posizione è determinato mediante il calcolo della suddivisione equa della circonferenza.

In base alla logica impiegata nella creazione della circonferenza, non dovrebbero emergere incongruenze nelle posizioni. I discostamenti (per lo più temporanei) dovrebbero essere dovuti al ritardo dello *slave* nel seguire i movimenti del master.

5.5 Analisi

Procediamo all'analisi e ai test analoghi a quelli svolti per l'algoritmo *Circular Forces* anche per l'algoritmo *Circular Position*. Condurremo in un secondo momento una valutazione conclusiva comparativa tra i due algoritmi in relazione ai differenti scenari considerati.

5.5.1 Test *Nominal*

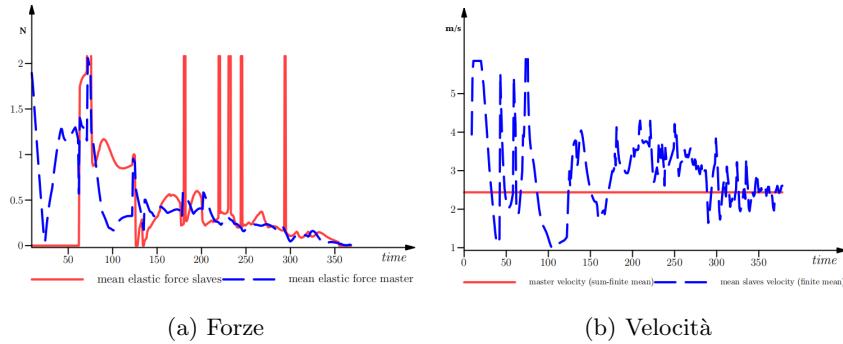


Figura 5.4: *Velocità e forze interagenti in Circular Position test Nominal*

Configurazione Nominal:

```
distanceMasterSlave = 50% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

Al fine di analizzare in dettaglio i picchi osservati nel primo grafico, suddividiamo il fenomeno in fasi distinte.

Prendiamo in considerazione un picco specifico rilevato durante la simulazione e riportiamo i dati pertinenti registrati in quel momento.

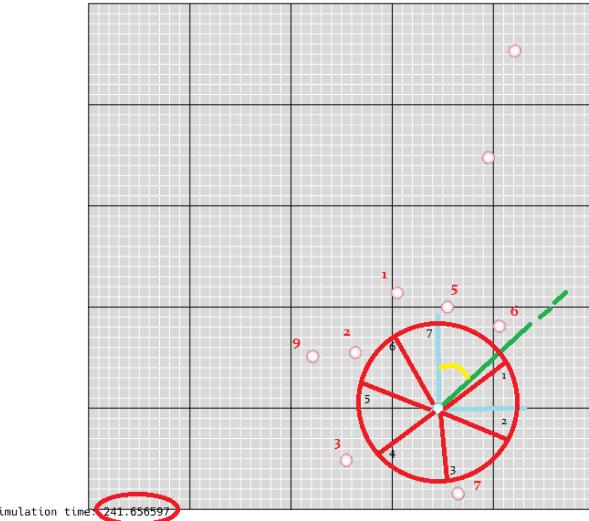


Figura 5.5: Fase 1: circonferenza in costruzione durante la simulazione Nominal con Circular Position

Tabella 5.3: Dati fase 1 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position

Identificativo slave	Indice di posizione	Elastic force slaves	flag distance
1	1	0.36N	true
2	6	0.39N	true
3	4	0.00N	false
5	2	0.41×10^{-1} N	true
6	7	0.35N	true
7	3	0.00N	false
9	5	0.47N	true

Ogni nodo *slave* è tenuto a raggiungere la posizione calcolata corrispondente al suo indice di posizione, al fine di stabilire un posizionamento angolare specifico rispetto al nodo *master*. Il raggiungimento di tale posizione potrebbe causare collisioni tra dispositivi: **vi è quindi, durante il processo di sistemazione dei dispositivi slave in formazione, la necessità di attivare una forza interagente al fine di prevenire collisioni.**

In base all'illustrazione fornita nella figura 5.5, che rappresenta una suddivisione angolare di una circonferenza con indici associati ai dispositivi *slave*, si osserva che solamente i dispositivi con identificatori 3 e 7 occupano una posizione adeguata. È interessante notare che questi due dispositivi sono gli unici a presentare una forza applicata pari a 0N e a riportare il flag di rilevazione di collisione (**flag distance**) impostato a *false*.

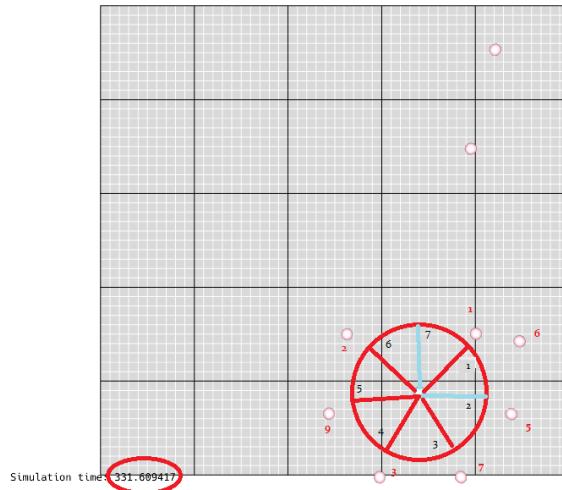


Figura 5.6: *Fase 2: circonferenza in costruzione durante la simulazione Nominal con Circular Position*

Tabella 5.4: *Dati fase 2 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position*

Identificativo <i>slave</i>	Indice di posizione	Elastic force <i>slaves</i>	flag distance
1	1	0.41N	true
2	6	0.00N	false
3	4	0.00N	false
5	2	0.11N	true
6	7	0.43N	true
7	3	0.00N	false
9	5	0.00N	false

La maggior parte dei dispositivi *slave* seguono la disposizione corrispondente al loro indice di posizione, ad eccezione del nodo 6, il quale è in fase di spostamento per raggiungere la posizione 7 a lui assegnata. Nell'istante di simulazione catturato, il nodo 6 si trova tra il nodo 1 e il nodo 5. Ciò conduce a una rilevazione di collisione in base alle distanze previste tra questi tre nodi.

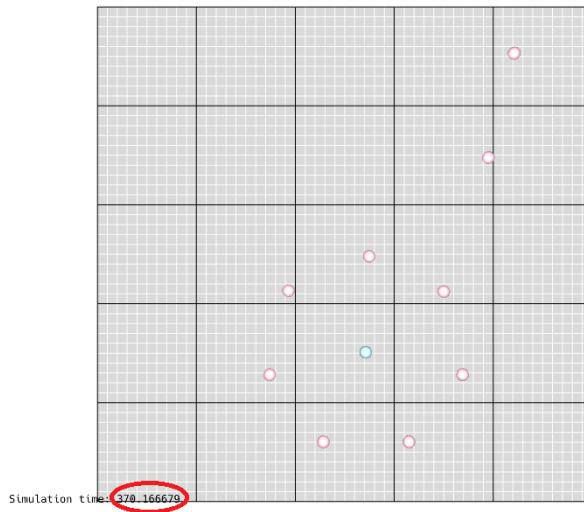


Figura 5.7: Fase 3: circonferenza in costruzione durante la simulazione Nominal con Circular Position

Tabella 5.5: Dati fase 3 relativi alle forze applicate catturati durante un fenomeno di picco grafico rilevato nel test Nominal con Circular Position

Identificativo slave	Indice di posizione	Elastic force slaves	flag distance
1	1	0.00N	false
2	6	0.00N	false
3	4	0.00N	false
5	2	0.00N	false
6	7	0.00N	false
7	3	0.00N	false
9	5	0.00N	false

Nella seconda fase del processo, si osserva che la maggior parte dei dispositivi *slave* sono disposti correttamente, e si può notare una significativa riduzione, sia dal grafico che dai dati ricavati, delle forze applicate rispetto alla fase iniziale. Gradualmente, tutte queste forze convergeranno verso lo zero, quando tutti i dispositivi saranno riusciti a stabilire la loro posizione corretta, come è chiaramente visibile nella terza fase.

Esaminiamo ora il grafico che rappresenta l'errore delle posizioni rispetto ad una circonferenza ideale, che in *Circular Position* rappresenta un errore esatto per comprendere al meglio il comportamento dell'algoritmo.

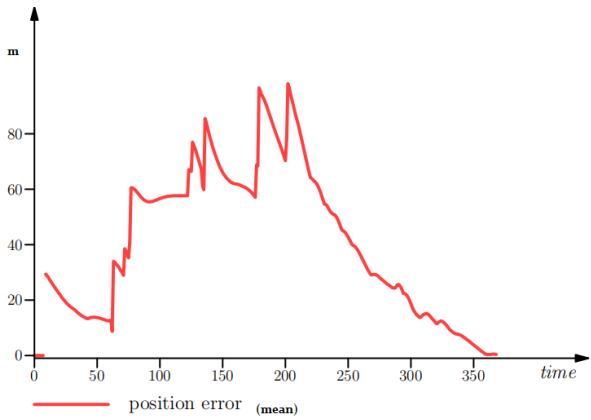


Figura 5.8: *Errore delle posizioni in Circular Position test Nominal*

Osserviamo che, in linea con quanto precedentemente discusso, l'errore di posizione tende a zero con l'avvicinarsi dei nodi alla loro corretta posizione angolare sulla circonferenza. Questo fenomeno di convergenza dell'errore di posizione può essere interpretato come una manifestazione della precisione crescente del sistema.

Al tempo massimo di simulazione considerato in questi grafici, la circonferenza non ha raggiunto la sua completa realizzazione, incorporando tutti i 10 nodi presenti in simulazione. Vediamo il risultato finale del test *Nominal*, in cui la circonferenza raggiunge la sua configurazione finale e completa.

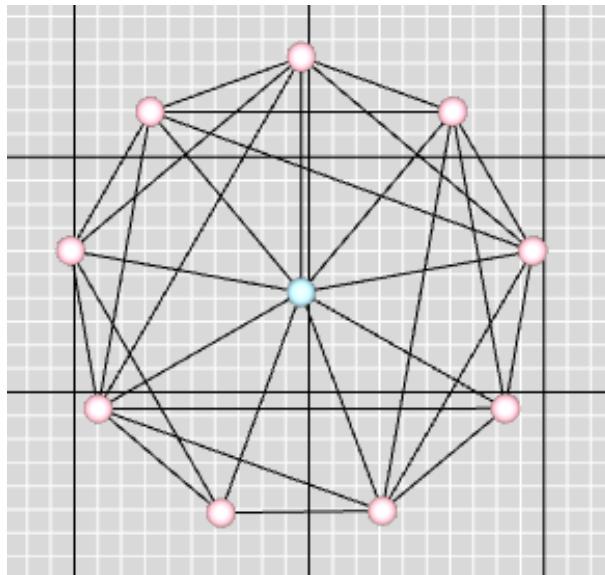


Figura 5.9: *Risultato finale del test Nominal con Circular Position*

5.5.2 Test *Slow Slaves Always Fast Master*

In questo esperimento, ci concentreremo sulle difficoltà di *Circular Position*, nell'adattarsi in uno scenario in cui tutti i dispositivi si spostano alla stessa velocità. Il nostro obiettivo è valutare quanto i dispositivi *slave* siano meno efficienti rispetto al dispositivo *master* quando operano alla stessa velocità. Nell'esperimento precedente, abbiamo osservato che quando il

dispositivo *master* viaggia a una velocità inferiore rispetto ai dispositivi *slave*, il sistema funziona correttamente senza problemi.

In questo nuovo test, esploreremo la situazione in cui il dispositivo *master* non è soggetto a riduzioni della velocità e si muove alla stessa velocità dei dispositivi *slave*.

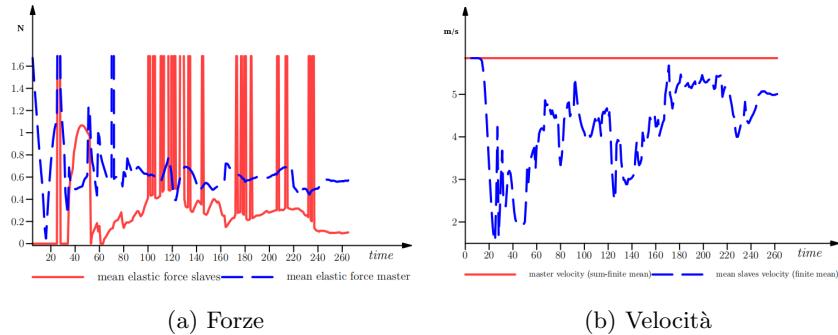


Figura 5.10: *Velocità e forze interagenti in Circular Position test Slow Slaves Always Fast Master*

Configurazione Slow Slaves Always Fast Master:

```
distanceMasterSlave = 50% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.0;
maxVelocitySlaves = 6;
masterVelocity = 6;
```

Osserviamo che le forze vengono applicate in modo costante, seguendo le regole di applicazione in *Circular Position*, questa condizione implica che i nodi tendono ad avvicinarsi tra loro al fine di stabilire un assetto ottimale. Inoltre notiamo che la velocità del nodo *master* rimane costantemente superiore rispetto a quella dei nodi *slave*, suggerendo che quest'ultimi non riescono a stare dietro ai movimenti del nodo *master*.

Verifichiamo quanto appena detto attraverso i grafici che illustrano il margine d'errore.

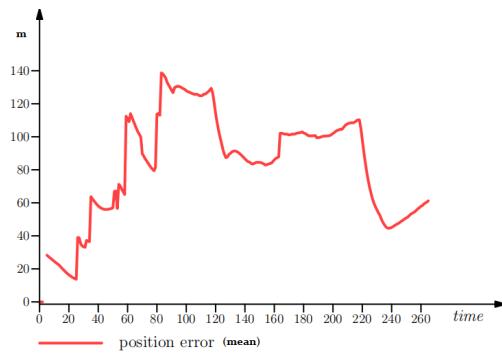


Figura 5.11: *Errore delle posizioni in Circular Position test Slow Slaves Always Fast Master*

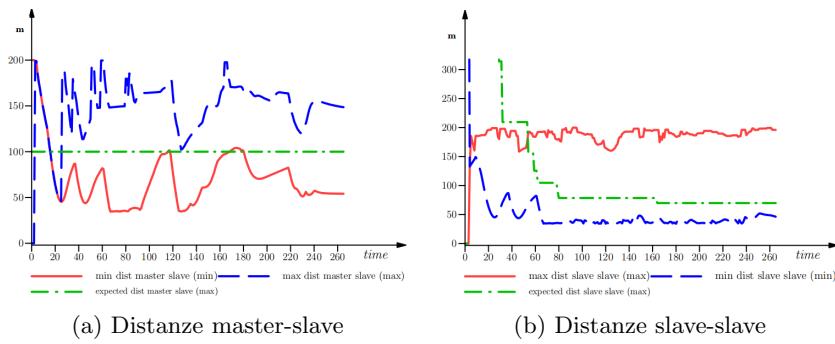


Figura 5.12: *Errore delle distanze in Circular Position test Slow Slaves Always Fast Master*

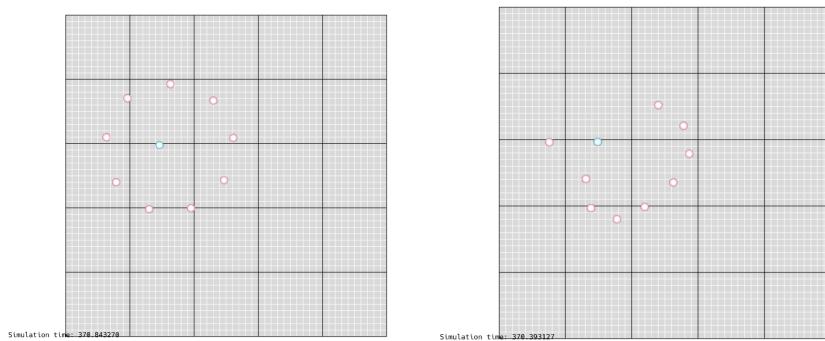
Nel periodo di osservazione considerato, non emergono segni di convergenza verso le distanze desiderate e la situazione non mostra stabilità. Questa constatazione è supportata dall'analisi degli errori di posizione calcolati, che evidenziano un andamento non conforme alle aspettative. Precedentemente, abbiamo visto come *Circular Forces* si adattata alle diverse velocità impostate sui dispositivi attraverso la modifica di una costante, permettendo alle forze di aumentare o diminuire a seconda dello scenario in cui vengono applicate.

Come mai invece Circular Position non ha questa caratteristica?

Le forze che operano all'interno del contesto di *Circular Position* sono vincolate da una specifica regione di attivazione, in cui l'incremento della forza viene regolato solamente nella regione di attivazione a livello 2 di criticità. Per questo motivo risultano troppo deboli per mantenere la formazione compatta; ma da un altro punto di vista è importante notare che queste forze non sono progettate con l'intento principale di mantenere la formazione geometrica, ma con lo scopo primario di evitare collisioni tra i dispositivi.

Pertanto, si può attribuire l'origine del problema all'accelerazione assegnata ai nodi subito dopo aver completato il calcolo dei rispettivi angoli che è insufficiente rispetto al movimento del *master*. Aumentare questa accelerazione mediante un incremento percentuale potrebbe rappresentare una possibile soluzione.

Eseguiamo ora un tentativo di aumento del 90% dell'accelerazione in esame, che chiameremo *accelerazione posizionale*, subito dopo la sua assegnazione.



(a) Accelerazione posizionale incrementata del 90% (b) Accelerazione posizionale non incrementata

Figura 5.13: Accelerazioni posizionali di Circular Position nel test Slow Slaves Always Fast Master

Osserviamo un significativo miglioramento. Ora esaminiamo attentamente i grafici in cui l'accelerazione angolare è incrementata del 90% per validare le differenze osservate nella simulazione.

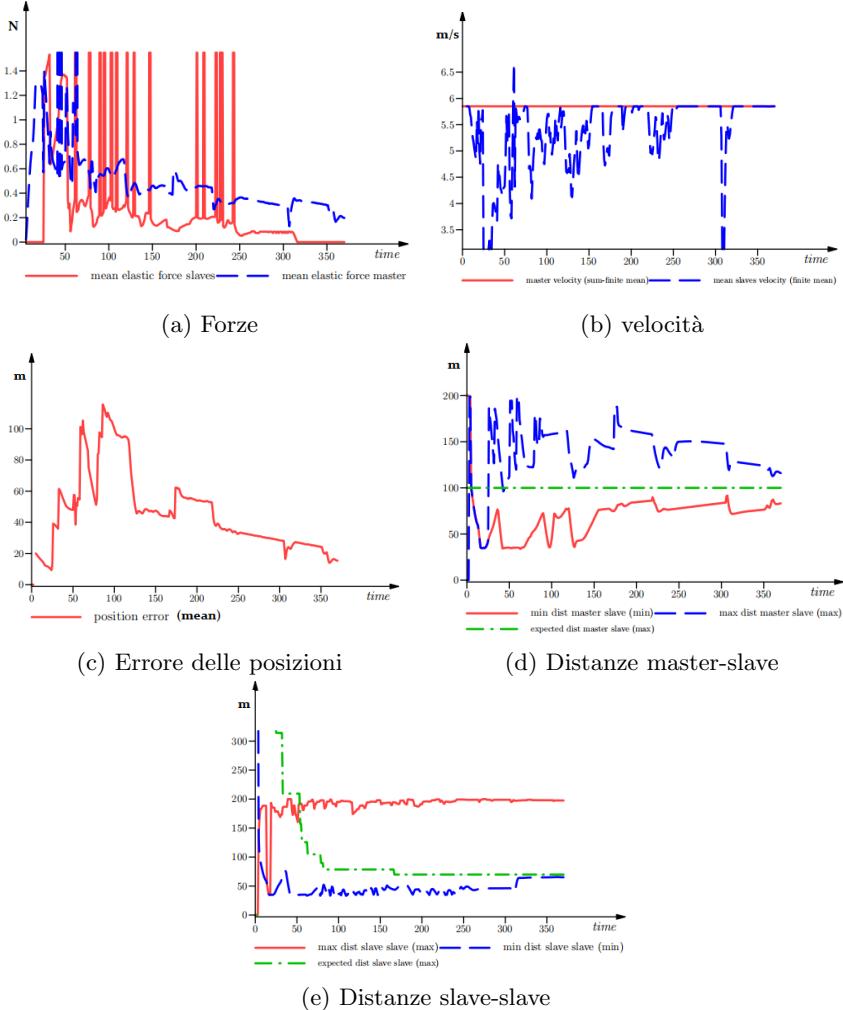


Figura 5.14: Analisi dell'accelerazione angolare aumentata di Circular Position nel test Slow Slaves Always Fast Master

Nella prima parte dell'analisi dei grafici relativi alle forze e alle velocità, notiamo che le forze applicate sono in scala inferiori rispetto al caso precedente. Inoltre, spesso osserviamo che la velocità dei dispositivi *slave* coincide con quella del dispositivo *master*. Questo suggerisce un aumento dell'efficienza dei dispositivi *slave* rispetto all'implementazione dell'accelerazione precedente, specialmente per quanto riguarda il loro movimento coordinato rispetto al dispositivo *master*.

Nella seconda parte dell'analisi dei grafici, concentrata sull'errore commesso, la valutazione conferma quanto appena esposto.

5.5.3 Test *Large Radius*

Esamineremo l'algoritmo *Circular Position* in un contesto in cui il raggio della circonferenza è ampio. Questo comporterà ad un calcolo delle distanze attese certe volte uguali o per-

sino maggiori al range di comunicazione. In tale situazione, osserveremo che si verificherà un peggioramento delle prestazioni dell'algoritmo a causa di un fenomeno che chiameremo **fenomeno di dispersione**.

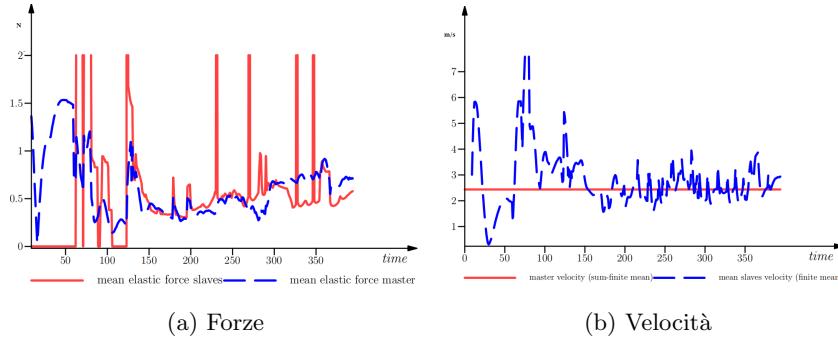


Figura 5.15: *Velocità e forze interagenti in Circular Position test Large Radius*

Configurazione Large Radius:

```
distanceMasterSlave = 75% of communication_range;
node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

Va ricordato che nell'algoritmo *Circular Position*, le forze non sono finalizzate al mantenimento della formazione, sono forze *Collision Avoidance* e pertanto non vengono applicate costantemente. Per tale ragione, nel corso di questo test, il grafico risultante delle forze non manifesta alcuna variazione significativa rispetto ai test precedenti.

Introduciamo ora i grafici successivi.

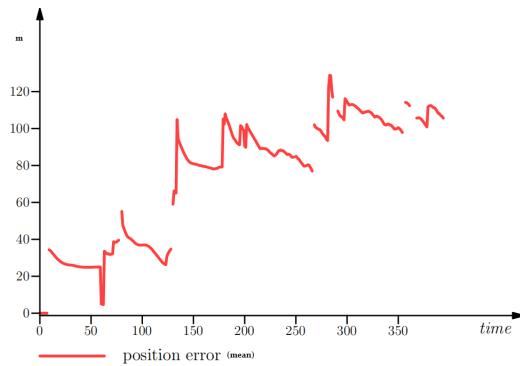


Figura 5.16: *Errore delle posizioni in Circular Position test Large Radius*

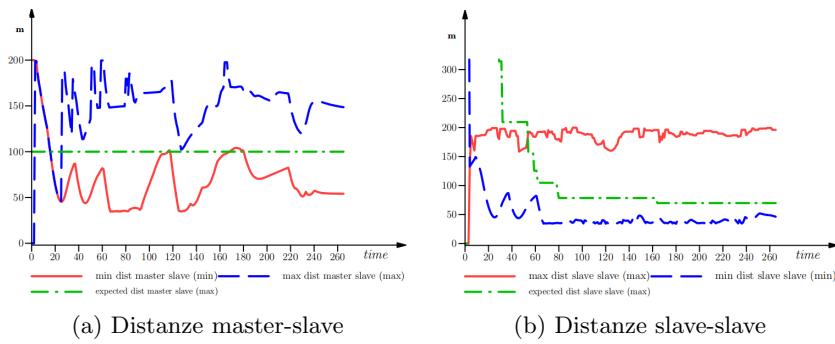


Figura 5.17: *Errore delle distanze in Circular Position test Large Radius*

Nel grafico dell'errore di posizione si osservano delle interruzioni nella curva, è il **fenomeno di dispersione** anticipato precedentemente. Questo fenomeno si manifesta quando i nodi *slave*, nel raggiungere la posizione corretta identificata tramite il loro indice univoco, si disperdonano all'interno del perimetro della circonferenza, causando la perdita di connessione, quindi un'interruzione nella comunicazione, tra di essi.

La dispersione comporta rilevanti errori dovuti alla sovrapposizioni di indici dei nodi *slave* appena entrati con quelli già all'interno dello *swarm*.

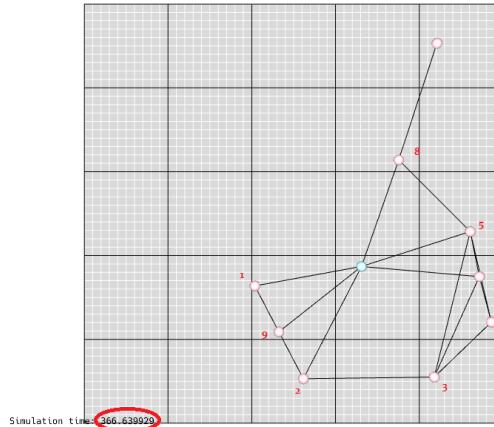


Figura 5.18: *Fenomeno di dispersione in Circular Position test Large Radius*

Tabella 5.6: *Dati relativi all'errore delle posizioni catturati al tempo massimo di simulazione preso in considerazione nei grafici del test Large Radius con Circular Position*

Identificativo slave	Indice di posizione	Check Index	Position Error
1	4	179.81m	false
2	6	156.54m	false
3	4	88.69m	false
5	3	149.60m	false
6	2	14.96m	false
7	1	177.69m	false
8	8	49.91m	false
9	1	275.21m	false

La sovrapposizione di indici, come ci aspettavamo, è presente durante lo svolgimento della simulazione del test *Large Radius*; in particolare, sono state rilevate collisioni negli indici dei nodi 1 con 3 e 7 con 9. È importante notare che il nostro metodo di verifica, tramite il flag **Check Index**, non segnala alcuna collisione. Questo metodo è stato precedentemente esaminato ed è progettato per rilevare indici duplicati al fine di consentire la successiva assegnazione di nuovi indici univoci.

Non viene segnalata alcuna collisione poiché i nodi con indici duplicati tra loro non sono connessi e, di conseguenza, non sono in grado di comunicare e rilevare la duplicazione dei valori.

La mancanza di comunicazione tra nodi, almeno tra quelli fisicamente vicini, comporta disfunzioni nel processo di assegnazione degli indici. Nel test *Large Radius* nel contesto di *Circular Forces*, la connessione tra i nodi non era un requisito essenziale, ma piuttosto una raccomandazione. Invece, nel contesto di *Circular Position*, è obbligatorio stabilire connessioni almeno tra i nodi fisicamente vicini, altrimenti la corretta costruzione della circonferenza risulta impossibile.

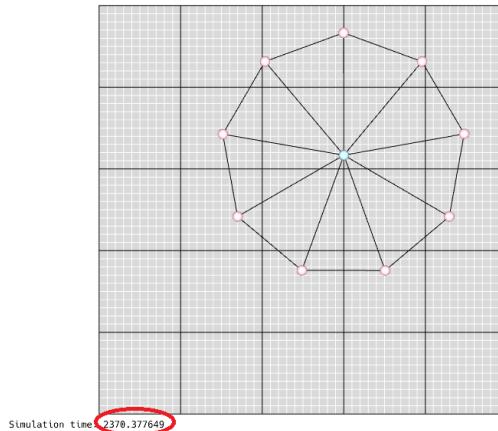


Figura 5.19: Risultato finale del test *Large Radius* con *Circular Position*

Quando i dispositivi *slave* riescono finalmente a stabilire una connessione con i dispositivi *slave* posizionati di fianco, alla loro destra e sinistra, la costruzione della circonferenza avviene in modo impeccabile (*si veda la figura 5.19 relativa al tempo 2370*).

Il test *Large Radius* con *Circular Position*, non è considerabile come un fallimento, ma piuttosto uno scenario in cui l'algoritmo dimostra una notevole inefficienza, manifestando una convergenza molto ritardata verso il risultato atteso.

5.5.4 Test *Busy Formation*

In questo ultimo test, *Circular Position* si troverà nella necessità di disporre numerosi dispositivi *slave* all'interno di un perimetro limitato.

In precedenza abbiamo discusso il concetto di *dispersione*; ora introduciamo il fenomeno contrario in questo test. Riscontreremo una **dispersione minima**, il che consentirà ai nodi *slave* di posizionarsi rapidamente al posto giusto con un impiego energetico ridotto.

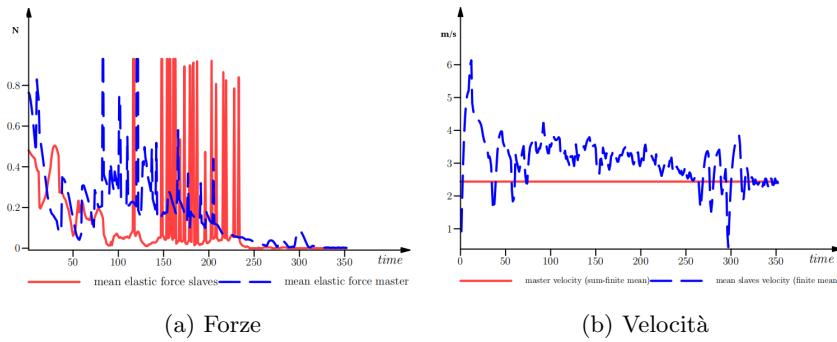


Figura 5.20: *Velocità e forze interagenti in Circular Position test Busy Formation*

Configurazione Busy Formation low second level of activation region:
`distanceMasterSlave = 30% of communication_range;`

```
//! Configurata in questa maniera anche nei precedenti test
minDistance = ((2 * distanceMasterSlave * pi) / (node_num - 1)) * 0.5;

node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

Osserviamo che il termine della fase di disposizione dei dispositivi non solo comporta delle minori forze interagenti, ma l'algoritmo converge verso uno stato di stabilità significativamente prima rispetto ai test precedenti.

Consideriamo ora un aspetto rilevante da esaminare in relazione alle forze applicate. La ridotta scala dell'asse delle ordinate è attribuibile non solo alla limitata dispersione, che comporta minori spostamenti all'interno del perimetro al fine di raggiungere la posizione desiderata, ma anche a un ulteriore parametro di configurazione che il sistema di *Collision Avoidance* impiega. In particolare parliamo della costante che regola la regione di attivazione di livello 2 di criticità nel sistema di *Collision Avoidance*. Essa equivale al 50% del risultato della suddivisione equa della circonferenza completa, ovvero contando la presenza di tutti gli *slave* in simulazione. In presenza di numerosi dispositivi *slave* all'interno di una circonferenza notevolmente ridotta, il valore in questione risulta eccessivamente limitato, con l'aumento delle forze che entra in gioco troppo tardi, nel momento in cui i dispositivi *slave* si sfiorano. Variamo questo dato come segue.

Configurazione Busy Formation unified level of activation region:
`distanceMasterSlave = 30% of communication_range;`

```
minDistance = ((2 * distanceMasterSlave * pi) / (node_num - 1)) * 0.9;

node_num = 10;
reduction = 0.025;
reductorMaster = 0.5;
maxVelocitySlaves = 6;
masterVelocity = 5;
```

Sostanzialmente abbiamo così unificato i due livelli della regione di attivazione in uno solo: il livello 1 di criticità evolve dinamicamente e corrisponde al 90% della distanza attesa

calcolata tra i dispositivi *slave* rispetto al numero degli *slave* presenti in un dato istante all'interno dello *swarm*. Quando tutti i nodi *slave* presenti in simulazione faranno parte della formazione, il livello 1 di criticità avrà lo stesso valore del livello 2 di criticità.

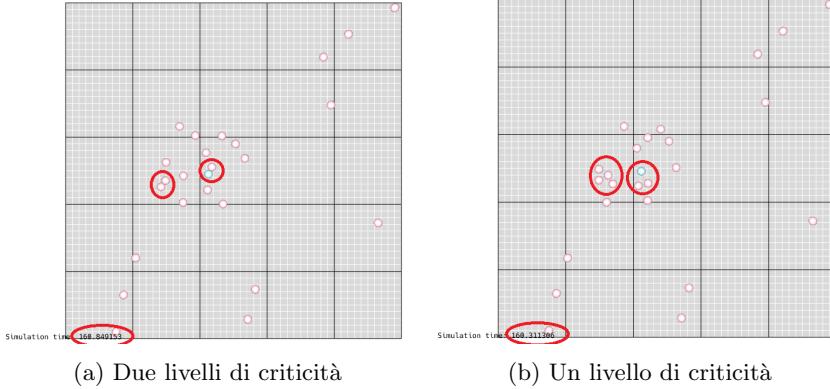


Figura 5.21: Diverse implementazioni della regione di attivazione del sistema Collision Avoidance in Circular Position nel test Busy Formation

La modifica di questo elemento di configurazione non incide sul risultato finale del test, è però una modifica consigliata. Essa comporta delle variazioni unicamente sotto l'aspetto riguardante le forze applicate durante l'evolversi della simulazione, vediamo in che modo.

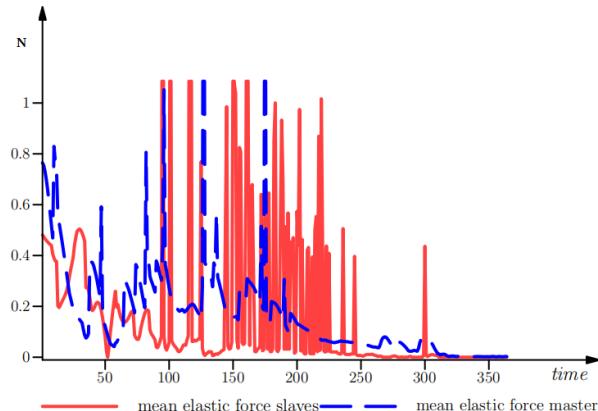


Figura 5.22: Forze applicate con un solo livello all'interno della regione di attivazione del sistema Collision Avoidance in Circular Position nel test Busy Formation

Come previsto, le forze applicate sono aumentate, ma ora ci troviamo di nuovo in un sistema sicuro.

Ora procediamo all'analisi della precisione con cui la circonferenza è stata costruita, mantenendo intatti i due livelli di criticità all'interno della regione di attivazione del sistema *Collision Avoidance*. Teniamo conto del caso in assenza dell'accelerazione posizionale incrementata dato che tale modifica non ha alcun impatto sul risultato finale ottenuto, viene così garantita una parità strutturale rispetto ai test precedenti.

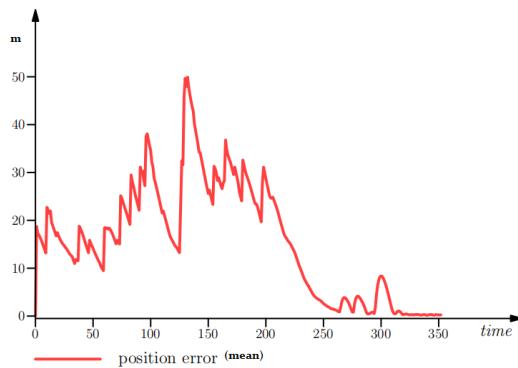


Figura 5.23: *Errore delle posizioni in Circular Position test Busy Formation*

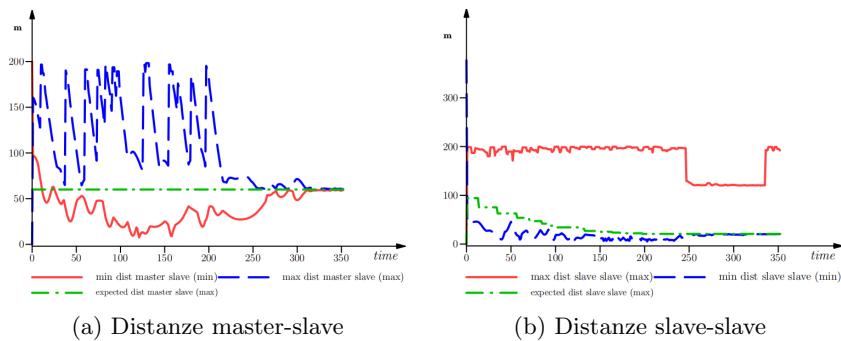


Figura 5.24: *Errore delle distanze in Circular Position test Busy Formation*

Osserviamo che a volte la distanza minima tra i dispositivi si riduce in modo significativo, pur senza mai raggiungere un valore nullo. Ciò rafforza la teoria precedentemente esposta. Nonostante questo aspetto, è importante notare che la circonferenza viene costruita correttamente, raggiungendo uno stato di stabilità con una notevole precisione.
Vediamo il risultato finale del test *Busy Formation*, in cui la circonferenza raggiunge la sua configurazione finale e completa.

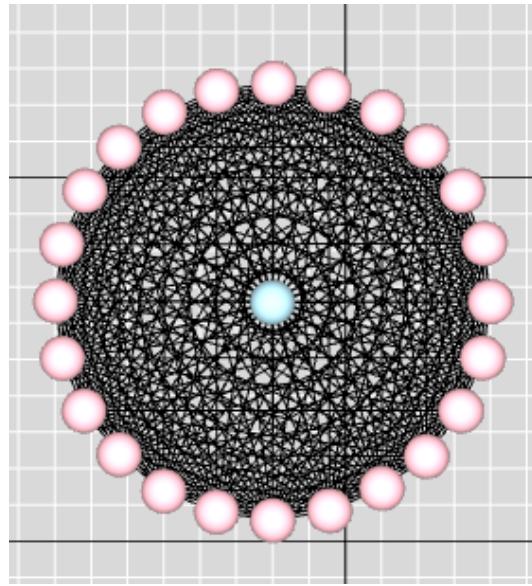


Figura 5.25: *Risultato finale del test Busy Formation con Circular Position*

5.6 Valutazione Conclusiva

Abbiamo osservato che l'algoritmo *Circular Position* è in grado di adattarsi a tutti gli scenari considerati, mantenendo un alto grado di precisione nella costruzione e nel mantenimento della formazione. Tuttavia, nel test *Large Radius* l'algoritmo richiede un considerevole periodo di tempo per costruire la circonferenza.

Ogni test ha rivelato caratteristiche e aspetti distintivi dell'algoritmo. Adesso procederemo a condurre un'analisi comparativa al fine di determinare se l'algoritmo *Circular Position* dimostra effettivamente una capacità superiore nel ridurre il margine d'errore e ottimizzare il consumo di energia rispetto all'algoritmo *Circular Forces*.

Capitolo 6

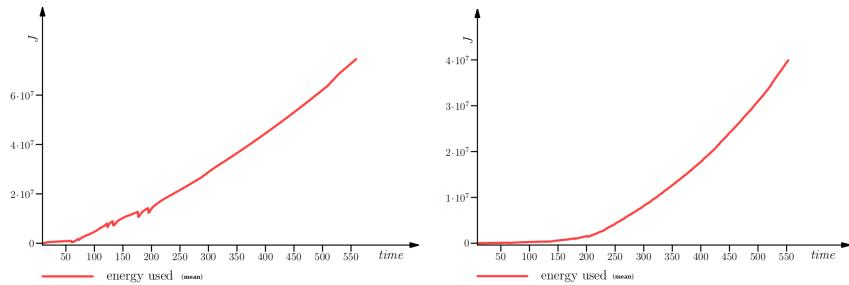
Analisi Comparativa

Ora eseguiremo un'analisi comparativa dei test precedentemente condotti, al fine di valutare e confrontare i dati raccolti per identificare relazioni, similitudini, differenze, vantaggi o svantaggi tra i due algoritmi proposti.

Verrà analizzato più nel dettaglio solamente il test *Nominal*. Per gli altri test, i risultati della comparazione emergono chiaramente dai capitoli precedenti, nei quali uno dei due algoritmi riscontra difficoltà o problematiche.

6.1 Analisi comparativa: Test *Nominal*

6.1.1 Consumi energetici dei dispositivi



(a) Energia utilizzata dai dispositivi *slaves* in *Circular Forces* (b) Energia utilizzata dai dispositivi *slaves* in *Circular Position*

Figura 6.1: *Consumo di energia a confronto test Nominal*

Il calcolo utilizzato per la stima del consumo energetico in base al lavoro svolto dall'agente, tralasciando le variabili specifiche del dispositivo su cui viene applicato che potrebbero influenzare tale calcolo, è descritto come segue:

$$ConsumoEnergetico(E) = Lavoro(W) = Forza(F) \times Distanza(d)$$

Descriviamo i passaggi applicati al fine di elaborare i grafici visualizzati sopra.

```
/// @brief Main function.
MAIN() {
```

```

    ...
    if(node.storage(node_isSlave)){
        ///! FATTORE MOLTIPLICATIVO CHE RAPPRESENTA LA FORZA APPLICATA:
        ///! è la somma dell'accelerazione risultante al termine delle fasi
        ///! dell'applicazione dell'algoritmo
        node.storage(sum_propulsion{}) = old(CALL, 0, [&](double propulsion){
            return propulsion + norm(node.propulsion());
        });

        ///! FATTORE MOLTIPLICATIVO CHE RAPPRESENTA LA DISTANZA:
        ///! è la somma della distanza percorsa rispetto alla volta precedente
        ///! cui è stato aggiornato l'elemento node_startPosition (al più
        ///! una volta ogni round)
        if(!(get<0>(node.storage(node_posMaster{})))){
            ///! Se non sono ancora in formazione salvo come valore di partenza
            ///! la posizione in cui è apparso il dispositivo all'interno della
            ///! griglia all'avvio della simulazione
            node.storage(node_startPosition{}) = node.position();
        }else{
            ///! Se sono in formazione identifico la distanza percorsa rispetto
            ///! l'ultimo aggiornamento di node_startPosition come la distanza tra
            ///! la posizione node_startPosition e quella attuale e la sommo al valore
            ///! distanza calcolato fino a quel momento
            node.storage(distanceTravelled{}) +=
            distance(node.storage(node_startPosition{}), node.position());
            ///! Aggiorno il valore di partenza con la posizione attuale
            node.storage(node_startPosition{}) = node.position();
        }
    }
}

```

I valori risultanti sono i fattori moltiplicativi nella formula precedentemente enunciata. Questi valori vengono quindi moltiplicati tra loro, il risultato ottenuto viene successivamente sommato e infine mediato per il numero di *slave* all'interno dello *swarm*.

Otteniamo così il *consumo energetico medio* di un dispositivo slave nel test *Nominal*.

Dai grafici ottenuti, emerge che dal punto di vista del *consumo energetico*, *Circular Position* sembra essere la scelta più vantaggiosa.

Durante il periodo di simulazione considerato nell'analisi dei grafici, *Circular Forces* raggiunge un valore di energia superiore a $6 \cdot 10^7 J$, mentre *Circular Position* raggiunge un valore approssimativamente pari a $4 \cdot 10^7 J$.

6.1.2 Valutazione della circonferenza costruita

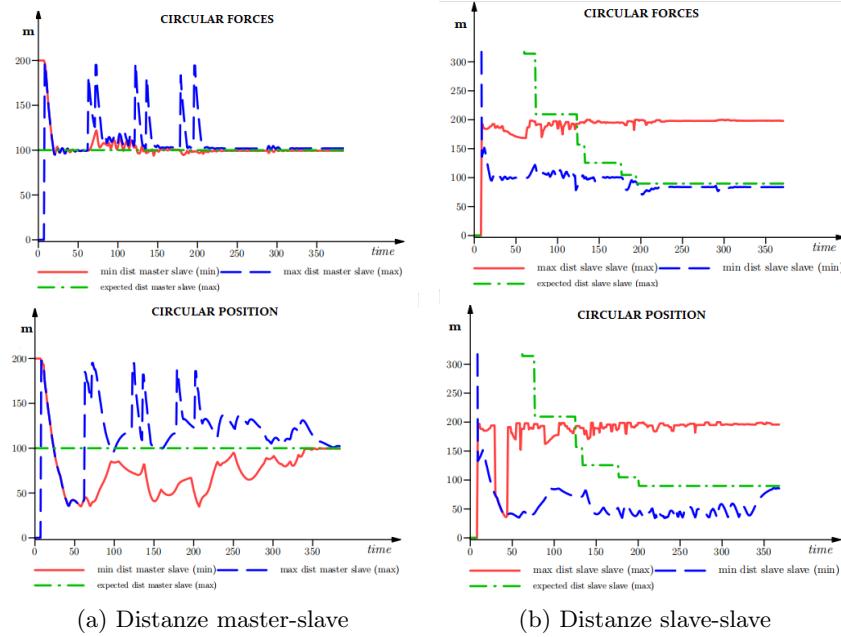


Figura 6.2: *Errore delle distanze a confronto test Nominal*

Sulla base di queste rappresentazioni grafiche, è possibile effettuare una valutazione della precisione della circonferenza costruita e della velocità con cui questo processo avviene.

La circonferenza creata dimostra un alto grado di precisione in entrambi gli algoritmi, i quali giungono a convergere verso una condizione in cui le distanze tra i dispositivi si allineano con le previsioni attese.

Notiamo però il fatto che l'algoritmo *Circular Forces* manifesta una convergenza più rapida verso tale risultato, raggiungendo, di conseguenza, uno stato di stabilità con maggiore certezza. Questo fatto emerge chiaramente dall'analisi dei grafici relativi all'errore di posizione e alle distanze massime e minime tra i dispositivi rispetto alle distanze attese.

Questa differenza è attribuibile al fatto che, nell'algoritmo *Circular Position*, i nodi *slave* richiedono un numero maggiore di iterazioni per stabilirsi all'interno della circonferenza. Quando la posizione assegnata è significativamente distante rispetto alla loro posizione d'entrata, questo comporta un maggior tempo impiegato per il loro posizionamento definitivo, creando ritardi nella stabilizzazione.

Capitolo 7

Conclusione

In conclusione, la scelta tra i due algoritmi dipende dalle esigenze dell'utilizzatore. Un algoritmo non è universalmente superiore rispetto all'altro, poiché il contesto e gli obiettivi influiscono sull'appropriata selezione.

7.1 Possibili Evoluzioni e Ottimizzazioni

7.1.1 Movimento del *Master*

Ricordiamo che, nei test condotti, il controllo del movimento del *master* veniva gestito tramite l'utilizzo dell'attrito. Tuttavia, come precedentemente anticipato nella **sezione 4.4.1**, è superfluo applicare questo meccanismo. Basterebbe semplicemente assegnare una velocità adeguata alla funzione di libreria `rectangle_walk`. Il meccanismo utilizzato non comporta alcuna alterazione nel risultato finale e in termini di efficienza derivante dalla sua implementazione; tuttavia, per una questione di precisione e coerenza, sarebbe opportuno apportare questa modifica.

7.1.2 Revisione del sistema di *Collision Avoidance*

All'interno del contesto dell'algoritmo *Circular Position*, è evidente che il sistema di *Collision Avoidance* ha un impatto negativo sulle prestazioni complessive dell'algoritmo.

Mediante una revisione della regione di attivazione delle forze all'interno del sistema e una diversa gestione dell'intensità con cui queste sono applicate, si potrebbero apportare miglioramenti significativi al funzionamento dell'algoritmo *Circular Position*.

Il secondo livello all'interno della regione di attivazione è definito dalla costante `minDistance` nella configurazione dell'algoritmo ed è reputata come la distanza minima critica che i dispositivi non possono oltrepassare. Il valore di questa è del 50% della distanza minima calcolabile in presenza di tutti gli *slave* all'interno della circonferenza.

Il primo livello della regione di attivazione è caratterizzato da un raggio più esteso ed ha una forza applicata minore rispetto al secondo livello. Il valore di questa non è costante, cambia durante la simulazione ed è del 90% della distanza calcolata in base al numero attuale di *slave* coinvolti nella formazione.

L'incremento della forza applicata tramite la costante `incrementForce`, definita nella configurazione dell'algoritmo, è effettuato esclusivamente nel secondo livello della regione di attivazione.

Il primo livello di criticità permette, all'interno del perimetro della circonferenza, di avere un basso, se non nullo margine d'errore, riguardo la precisione della posizione effettiva occupata dal nodo *slave* rispetto a quella ideale calcolata matematicamente. La forza applicata

in questo livello, pur avendo un'intensità molto bassa, limita la libertà di movimento del dispositivo su cui è applicata; questa limitazione risulta vantaggiosa nel **mantenimento** della formazione con una configurazione il più precisa possibile. Tuttavia, tale limitazione rappresenta un ostacolo durante la fase di **costruzione** della formazione. Nel caso in cui un nodo *slave*, appena entrato nello *swarm*, deve raggiungere la posizione a lui assegnata e questa sia situata ad una distanza considerevole e vi sono già altri nodi *slave* all'interno della formazione lungo il suo percorso, tali nodi possono limitare il suo spostamento, rallentando il raggiungimento del suo *goal*.

Teoricamente, un possibile approccio per ottimizzare le prestazioni complessive dell'algoritmo potrebbe essere quello di mantenere solo il secondo livello di criticità nella regione d'attivazione.

7.1.3 Considerazioni sull'analisi svolta

È stata condotta un'analisi preliminare allo scopo di acquisire una comprensione basilare del funzionamento complessivo dei due algoritmi. Per una valutazione più accurata, sarebbe opportuno eseguire una serie estesa di test su più scenari. Comunque, in base ai test eseguiti finora, entrambi gli algoritmi operano in coerenza con le aspettative previste.

Si prevedeva che, dal punto di vista dell'efficienza energetica e dell'accuratezza nella realizzazione di una circonferenza geometricamente ideale, l'algoritmo *Circular Position* sarebbe stato più efficiente. D'altra parte ci aspettavamo che l'algoritmo *Circular Forces*, richiedendo una gestione più semplice, non presentasse vincoli significativi, come ad esempio il vincolo della comunicazione tra i dispositivi *slave* all'interno della formazione.

Bibliografia

- [1] Martina Mammarella, Lorenzo Comba, Alessandro Biglia, Fabrizio Dabbene, and Paolo Gay. *Cooperation of unmanned systems for agricultural applications: A theoretical framework.*
- [2] Huang Yao, Rongjun Qin, and Xiaoyu Chen. *Unmanned aerial vehicle for remote sensing applications—A review.*
- [3] McAllister W., Osipychev D., Davis A., and Chowdhary G. *Agbots: Weeding a field with a team of autonomous robots.*
- [4] Beal, Viroli, and Pianini. *Aggregate Programming for the Internet of Things.*
- [5] G. Audrito. *FCPP: an efficient and extensible Field Calculus framework.*
- [6] Marco Vergani. *Controllo predittivo distribuito per la navigazione e il sensing di sistemi multi agente.*

Ringraziamenti

Desidero esprimere la mia più grande gratitudine innanzitutto ai miei genitori Enzo e Dina, che mi hanno sostenuto sia economicamente che moralmente durante questi anni, a mio fratello maggiore Domenico e alla sua fidanzata Alessia, sempre presenti nei momenti in cui ho avuto bisogno.

Un ringraziamento anche ai miei amici, che hanno contribuito a rendere questi anni più sereni. In particolare, vorrei ringraziare i miei amici dell'università Alessandro, Andrea e Carmine e il mio ragazzo Patric, che non solo hanno reso le giornate in dipartimento meno pesanti, ma mi hanno anche sempre aiutata nello studio quando mi sono trovata in difficoltà. Ringrazio inoltre il mio ragazzo per il costante sostegno e l'incoraggiamento che mi ha mostrato nel corso di questi anni.

Ringrazio i Professori Gianluca Torta, Giorgio Audrito e Daniele Bortoluzzi per avermi dato l'opportunità di realizzare questo lavoro che va oltre ad un lavoro volto alla semplice stesura di una tesi triennale. Mi hanno seguita costantemente spronandomi a fare sempre meglio e aiutandomi a crescere sotto l'aspetto professionale.

Infine, ringrazio il resto della mia famiglia, che è sempre stata orgogliosa e felice per i miei successi. Vorrei in particolare menzionare mia zia Pasqualina, che ha anche dedicato del tempo a farmi una lezione sulla trigonometria, contribuendo alla mia comprensione di come svolgere il lavoro sviluppato in questa tesi.